daScript Reference Manual

Release 0.2 beta

Anton Yudintsev

Jul 16, 2024

CONTENTS

CHAPTER

ONE

INTRODUCTION

Daslang is a high-performance, strong and statically typed scripting language, designed to be high-performance as an embeddable “scripting” language for real-time applications (like games).

Daslang offers a wide range of features like strong static typing, generic programming with iterative type inference, Ruby-like blocks, semantic indenting, native machine types, ahead-of-time “compilation” to C++, and fast and simplified bindings to C++ program.

It’s philosophy is build around a modified Zen of Python.

- *Performance counts.*
- *But not at the cost of safety.*
- *Unless is explicitly unsafe to be performant.*
- *Readability counts.*
- *Explicit is better than implicit.*
- *Simple is better than complex.*
- *Complex is better than complicated.*
- *Flat is better than nested.*

Daslang is supposed to work as a “host data processor”. While it is technically possible to maintain persistent state within a script context (with a certain option set), Daslang is designed to transform your host (C++) data/implement scripted behaviors.

In a certain sense, it is pure functional - i.e. all persistent state is out of the scope of the scripting context, and the script’s state is temporal by its nature. Thus, the memory model and management of persistent state are the responsibility of the application. This leads to an extremely simple and fast memory model in Daslang itself.

1.1 Performance.

In a real world scenarios, it’s interpretation is 10+ times faster than LuaJIT without JIT (and can be even faster than LuaJIT with JIT). Even more important for embedded scripting languages, its interop with C++ is extremely fast (both-ways), an order of magnitude faster than most other popular scripting languages. Fast calls from C++ to Daslang allow you to use Daslang for simple stored procedures, and makes it an ECS/Data Oriented Design friendly language. Fast calls to C++ from Daslang allow you to write performant scripts which are processing host (C++) data, and rely on bound host (C++) functions.

It also allows Ahead-of-Time compilation, which is not only possible on all platforms (unlike JIT), but also always faster/not-slower (JIT is known to sometimes slow down scripts).

Daslang already has implemented AoT (C++ transpiler) which produces code more or less similar with C++11 performance of the same program.

Table with performance comparisons on a synthetic samples/benchmarks.

1.2 How it looks?

Mandatory fibonacci samples:

```
def fibR(n)
   if (n < 2)
       return n
   else
       return fibR(n - 1) + fibR(n - 2)

def fibI(n)
   var last = 0
   var cur = 1
   for i in range(0, n - 1)
       let tmp = cur
       cur += last
       last = tmp
   return cur
```

The same samples with curly brackets, for those who prefer this type of syntax:

```
def fibR(n) {
    if (n < 2) {
        return n;
    } else {
        return fibR(n - 1) + fibR(n - 2);
    }
}
def fibI(n) {
    var last = 0;
    var cur = 1;
    for i in range(0, n-1); {
        let tmp = cur;
        cur += last;
        last = tmp;
    }
    return cur;
}
```

Please note, that semicolons(';') are mandatory within curly brackets. You can actually mix both ways in your codes, but for clarity in the documentation, we will only use the pythonic way.

1.3 Generic programming and type system

Although above sample may seem to be dynamically typed, it is actually generic programming. The actual instance of the fibI/fibR functions is strongly typed and basically is just accepting and returning an `int`. This is similar to templates in C++ (although C++ is not a strong-typed language) or ML. Generic programming in Daslang allows very powerful compile-time type reflection mechanisms, significantly simplifying writing optimal and clear code. Unlike C++ with it's SFINAE, you can use common conditionals (if) in order to change the instance of the function depending on type info of its arguments. Consider the following example:

```
def setSomeField(var obj; val)
    static_if typeinfo(has_field<someField> obj)
        obj.someField = val
```

This function sets *someField* in the provided argument *if* it is a struct with a *someField* member.

(For more info, see *Generic programming*).

1.4 Compilation time macros

Daslang does a lot of heavy lifting during compilation time so that it does not have to do it at run time. In fact, the Daslang compiler runs the Daslang interpreter for each module and has the entire AST available to it.

The following example modifies function calls at compilation time to add a precomputed hash of a constant string argument:

```
[tag_function_macro(tag="get_hint_tag")]
class GetHintFnMacro : AstFunctionAnnotation
    [unsafe] def override transform ( var call : smart_ptr<ExprCall>;
        var errors : das_string ) : ExpressionPtr
        if call.arguments[1] is ExprConstString
            let arg2 = reinterpret<ExprConstString?>(call.arguments[1])
            var mkc <- new [[ExprConstUInt() at=arg2.at, value=hash("{arg2.value}")]]
            push(call.arguments, ExpressionPtr(mkc))
            return <- ExpressionPtr(call)
        return [[ExpressionPtr]]
```

1.5 Features

Its (not) full list of features includes:

- strong typing
- Ruby-like blocks and lambda
- tables
- arrays
- string-builder
- native (C++ friendly) interop
- generics
- classes

- macros, including reader macros
- semantic indenting
- ECS-friendly interop
- easy-to-extend type system
- etc.

CHAPTER

TWO

THE LANGUAGE

2.1 Lexical Structure

2.1.1 Identifiers

Identifiers start with an alphabetic character (and not the symbol '_') followed by any number of alphabetic characters, '_' or digits ([0-9]). Daslang is a case sensitive language meaning that the lowercase and uppercase representation of the same alphabetic character are considered different characters. For instance, "foo", "Foo" and "fOo" are treated as 3 distinct identifiers.

2.1.2 Keywords

The following words are reserved as keywords and cannot be used as identifiers:

struct	class	let	def	while	if
static_if	else	for	recover	true	false
new	typeinfo	type	in	is	as
elif	static_elif	array	return	null	break
try	options	table	expect	const	require
operator	enum	finally	delete	deref	aka
typedef	with	cast	override	abstract	upcast
iterator	var	addr	continue	where	pass
reinterpret	module	public	label	goto	implicit
shared	private	smart_ptr	generator	yield	unsafe
assume	explicit	sealed	static	inscope	fixed_array
typedecl					

The following words are reserved as type names and cannot be used as identifiers:

bool	void	string	auto	int	int2
int3	int4	uint	bitfield	uint2	uint3
uint4	float	float2	float3	float4	range
urange	block	int64	uint64	double	function
lambda	int8	uint8	int16	uint16	tuple
variant	range64	urange64			

Keywords and types are covered in detail later in this document.

2.1.3 Operators

Daslang recognizes the following operators:

`+=`	`-=`	`/=`	`*=`	`%=`	`|=`	`^=`	`<<`
`>>`	`++`	`--`	`<=`	`<<=`	`>>=`	`>=`	`==`
`!=`	`->`	`<-`	`??`	`?.`	`?[`	`<|`	`|>`
`:=`	`<<<`	`>>>`	`<<<=`	`>>>=`	`=>`	`+`	`@@`
`-`	`*`	`/`	`%`	`&`	`|`	`^`	`>`
`<`	`!`	`~`	`&&`	`||`	`^^`	`&&=`	`||=`
`^^=`							

2.1.4 Other tokens

Other significant tokens are:

`{`	`}`	`[`	`]`	`.`	`:`
`::`	`'`	`;`	`"`	`]]`	`[[`
`[{`	`}]`	`{{`	`}}`	`@`	`$`
`#`					

2.1.5 Literals

Daslang accepts integer numbers, unsigned integers, floating and double point numbers and string literals.

`34`	Integer number(base 10)
`0xFF00A120`	Unsigned Integer number(base 16)
`0753`	Integer number(base 8)
`'a'`	Integer number
`1.52`	Floating point number
`1.e2`	Floating point number
`1.e-2`	Floating point number
`1.52d`	Double point number
`1.e2d`	Double point number
`1.e-2d`	Double point number
`"I'm a string"`	String
`" I'm a multiline verbatim string "`	String

Pesudo BNF:

```
IntegerLiteral          ::=   [1-9][0-9]* | '0x' [0-9A-Fa-f]+ | ''' [.]+ ''' | 0[0-7]+
FloatLiteral            ::=   [0-9]+ '.' [0-9]+
FloatLiteral            ::=   [0-9]+ '.' 'e'|'E' '+'|'-' [0-9]+
StringLiteral           ::=   '"'[.]* '"'
VerbatimStringLiteral   ::=   '@''"'[.]* '"'
```

2.1.6 Comments

A comment is text that the compiler ignores, but is useful for programmers. Comments are normally used to embed annotations in the code. The compiler treats them as white space.

A comment can be `/*` (slash, asterisk) characters, followed by any sequence of characters (including new lines), followed by the `*/` characters. This syntax is the same as ANSI C:

```
/*
This is
a multiline comment.
This lines will be ignored by the compiler.
*/
```

A comment can also be `//` (two slash) characters, followed by any sequence of characters. A new line not immediately preceded by a backslash terminates this form of comment. It is commonly called a *"single-line comment"*:

```
// This is a single line comment. This line will be ignored by the compiler.
```

2.1.7 Semantic Indenting

Daslang follows semantic indenting (much like Python). That means that logical blocks are arranged with the same indenting, and if a control statement requires the nesting of a block (such as the body of a function, block, if, for, etc.), it has to be indented one step more. The indenting step is part of the options of the program. It is either 2, 4 or 8, but always the same for whole file. The default indenting is 4, but can be globally overridden per project.

2.2 Values and Data Types

Daslang is a strong, statically typed language. All variables have a type. Daslang's basic POD (plain old data) data types are:

```
int, uint, float, bool, double, int64, uint64
int2, int3, int4, uint2, uint3, uint4, float2, float3, float4
```

All PODs are represented with machine register/word. All PODs are passed to functions by value.

Daslang's storage types are:

```
int8, uint8, int16, uint16 - 8/16-bits signed and unsigned integers
```

They can't be manipulated, but can be used as storage type within structs, classes, etc.

Daslang's other types are:

```
string, das_string, struct, pointers, references, block, lambda, function pointer,
array, table, tuple, variant, iterator, bitfield
```

All Daslang's types are initialized with zeroed memory by default.

2.2.1 Integer

An integer represents a 32-bit (un)signed number:

```
let a = 123    // decimal, integer
let u = 123u   // decimal, unsigned integer
let h = 0x0012 // hexadecimal, unsigned integer
let o = 075    // octal, unsigned integer

let a = int2(123, 124)     // two integers type
let u = uint2(123u, 124u) // two unsigned integer type
```

2.2.2 Float

A float represents a 32-bit floating point number:

```
let a = 1.0
let b = 0.234
let a = float2(1.0, 2.0)
```

2.2.3 Bool

A bool is a double-valued (Boolean) data type. Its literals are `true` and `false`. A bool value expresses the validity of a condition (tells whether the condition is true or false):

```
let a = true
let b = false
```

All conditionals (if, elif, while) work only with the bool type.

2.2.4 String

Strings are an immutable sequence of characters. In order to modify a string, it is necessary to create a new one.

Daslang's strings are similar to strings in C or C++. They are delimited by quotation marks(`"`) and can contain escape sequences (`\t`, `\a`, `\b`, `\n`, `\r`, `\v`, `\f`, `\\`, `\"`, `\'`, `\0`, `\x<hh>`, `\u<hhhh>` and `\U<hhhhhhhh>`):

```
let a = "I'm a string\n"
let a = "I'm also
    a multi-line
        string\n"
```

Strings type can be thought of as a 'pointer to the actual string', like a 'const char *' in C. As such, they will be passed to functions by value (but this value is just a reference to the immutable string in memory).

`das_string` is a mutable string, whose content can be changed. It is simply a builtin handled type, i.e., a std::string bound to Daslang. As such, it passed as reference.

2.2.5 Table

Tables are associative containers implemented as a set of key/value pairs:

```
var tab: table<string; int>
tab["10"] = 10
tab["20"] = 20
tab["some"] = 10
tab["some"] = 20 // replaces the value for 'some' key
```

(see *Tables*).

2.2.6 Array

Arrays are simple sequences of objects. There are static arrays (fixed size) and dynamic arrays (container, size is dynamic). The index always starts from 0:

```
var a = [[int[4] 1; 2; 3; 4]] // fixed size of array is 4, and content is [1, 2, 3, 4]
var b: array<string>          // empty dynamic array
push(b,"some")                // now it is 1 element of "some"
```

(see *Arrays*).

2.2.7 Struct

Structs are records of data of other types (including structs), similar to C. All structs (as well as other non-POD types, except strings) are passed by reference.

(see *Structs*).

2.2.8 Classes

Classes are similar to structures, but they additionally allow built-in methods and rtti.

(see *Classes*).

2.2.9 Variant

Variant is a special anonymous data type similar to a struct, however only one field exists at a time. It is possible to query or assign to a variant type, as well as the active field value.

(see *Variants*).

2.2.10 Tuple

Tuples are anonymous records of data of other types (including structs), similar to a C++ std::tuple. All tuples (as well as other non-POD types, except strings) are passed by reference.

(see *Tuples*).

2.2.11 Enumeration

An enumeration binds a specific integer value to a name, similar to C++ enum classes.

(see *Enumerations*).

2.2.12 Bitfield

Bitfields are an anonymous data type, similar to enumerations. Each field explicitly represents one bit, and the storage type is always a uint. Queries on individual bits are available on variants, as well as binary logical operations.

(see *Bitfields*).

2.2.13 Function

Functions are similar to those in most other languages:

```
def twice(a: int): int
    return a + a
```

However, there are generic (templated) functions, which will be 'instantiated' during function calls by type inference:

```
def twice(a)
    return a + a

let f = twice(1.0) // 2.0 float
let i = twice(1)   // 2 int
```

(see *Functions*).

2.2.14 Reference

References are types that 'reference' (point to) some other data:

```
def twice(var a: int&)
    a = a + a
var a = 1
twice(a) // a value is now 2
```

All structs are always passed to functions arguments as references.

2.2.15 Pointers

Pointers are types that 'reference' (point to) some other data, but can be null (point to nothing). In order to work with actual value, one need to dereference it using the dereference or safe navigation operators. Dereferencing will panic if a null pointer is passed to it. Pointers can be created using the new operator, or with the C++ environment.

```
def twice(var a: int&)
    a = a + a
def twicePointer(var a: int?)
    twice(*a)

struct Foo
    x: int

def getX(foo: Foo?)  // it returns either foo.x or -1, if foo is null
   return foo?.x ?? -1
```

2.2.16 Iterators

Iterators are a sequence which can be traversed, and associated data retrieved. They share some similarities with C++ iterators.

(see *Iterators*).

2.3 Statements

A Daslang program is a simple sequence of statements:

```
stats ::= stat [';'|'\n'] stats
```

Statements in Daslang are comparable to those in C-family languages (C/C++, Java, C#, etc.): there are assignments, function calls, program flow control structures, etc. There are also some custom statements like blocks, structs, and initializers (which will be covered in detail later in this document). Statements can be separated with a new line or ';'.

2.3.1 Visibility Block

```
visibility_block ::= indent (stat)* unindent
visibility_block ::= '{' (stat)* '}'
```

A sequence of statements delimited by indenting or curly brackets ({ }) is called a visibility_block.

2.3.2 Control Flow Statements

Daslang implements the most common control flow statements: `if, while, for`

true and false

Daslang has a strong boolean type (bool). Only expressions with a boolean type can be part of the condition in control statements.

if/elif/else statement

```
stat ::= 'if' exp '\n' visibility_block (['elif' exp '\n' visibility_block])*  ['else
↪' '\n' visibility_block]
```

Conditionally executes a statement depending on the result of an expression:

```
if a > b
    a = b
elif a < b
    b = a
else
    print("equal")
```

while statement

```
stat ::= 'while' exp '\n' indent stat
```

Executes a statement while the condition is true:

```
while true
    if a<0
        break
```

2.3.3 Ranged Loops

for

```
stat ::= 'for' iterator 'in' [rangeexp] '\n' visibility_block
```

Executes a loop body statement for every element/iterator in expression, in sequenced order:

```
for i in range(0, 10)
    print("{i}")        // will print numbers from 0 to 9

// or

let arr: array<int>
resize(arr, 4)
for i in arr
    print("{i}")        // will print content of array from first element to last

// or

var a: array<int>
var b: int[10]
resize(a, 4)
```

(continues on next page)

(continued from previous page)

```
for l, r in a, b
    print("{l}=={r}")  // will print content of a array and first 4 elements of array␣
↪b

// or

var tab: table<string; int>
for k, v in keys(tab), values(tab)
    print("{k}:{v}")   // will print content of table, in form key:value
```

Iterable types are implemented via iterators (see *Iterators*).

2.3.4 break

```
stat ::= 'break'
```

The break statement terminates the execution of a loop (`for` or `while`).

2.3.5 continue

```
stat ::= 'continue'
```

The continue operator jumps to the next iteration of the loop, skipping the execution of the rest of the statements.

2.3.6 return

```
stat ::= return [exp]
stat ::= return <- exp
```

The return statement terminates the execution of the current function, block, or lambda, and optionally returns the result of an expression. If the expression is omitted, the function will return nothing, and the return type is assumed to be void. Returning mismatching types from same function is an error (i.e., all returns should return a value of the same type). If the function's return type is explicit, the return expression should return the same type.

Example:

```
def foo(a: bool)
    if a
      return 1
    else
      return 0.f  // error, different return type

def bar(a: bool): int
    if a
      return 1
    else
      return 0.f  // error, mismatching return type

def foobar(a)
    return a  // return type will be same as argument type
```

In generator blocks, return must always return boolean expression, where false indicates end of generation.

'return <- exp' syntax is for move-on-return:

```
def make_array
    var a: array<int>
    a.resize(10)   // fill with something
    return <- a    // return will return

let a <- make_array() //create array filled with make_array
```

2.3.7 yield

Yield serves similar purpose as `return` for generators (see *Generators*).

It is similar to return syntax, but can only be used inside `generator` blocks.

Yield must always produce a value which matches that of the generator:

```
let gen <- generator<int>() <| $()
    yield 0          // int 0
    yield 1          // int 1
    return false
```

2.3.8 Finally statement

```
stat ::= finally visibility-block
```

Finally declares a block which will be executed once for any block (including control statements). A finally block can't contain `break`, `continue`, or `return` statements. It is designed to ensure execution after 'all is done'. Consider the following:

```
def test(a: array<int>; b: int)
    for x in a
        if x == b
            return 10
     return -1
finally
     print("print anyway")

def test(a: array<int>; b: int)
    for x in a
        if x == b
            print("we found {x}")
            break
    finally
         print("we print this anyway")
```

Finally may be used for resource de-allocation.

It's possible to add code to the finally statement of the block with the `defer` macro:

```
require daslib/defer

def foo
```

(continues on next page)

(continued from previous page)

```
    print("a\n")
finally
    print("b\n")

def bar
    defer <|
        print("b\n")
    print("a\n")
```

In the example above, functions `foo` and `bar` are semantically identical. Multiple `defer` statements occur in reverse order.

The `defer_delete` macro adds a delete statement for its argument, and does not require a block.

2.3.9 Local variables declaration

```
initz ::= id [:type] [= exp]
initz ::= id [:type] [<- exp]
initz ::= id [:type] [:= exp]
scope ::= `inscope`
ro_stat ::= 'let' [scope] initz
rw_stat ::= 'var' [scope] initz
```

Local variables can be declared at any point in a function. They exist between their declaration and the end of the visibility block where they have been declared. `let` declares read only variables, and `var` declares mutable (read-write) variables.

Copy =, move `->`, or clone `:=` semantics indicate how the variable is to be initialized.

If `inscope` is specified, the `delete id` statement is added in the finally section of the block, where the variable is declared. It can't appear directly in the loop block, since finally section of the loop is executed only once.

2.3.10 Function declaration

```
stat ::= 'def' id ['(' args ')'] [':' type ] visibility_block

arg_decl = [var] id (',' id)* [':' type]
args ::= (arg_decl)*
```

Declares a new function. Examples:

```
def hello
    print("hello")

def hello(): bool
    print("hello")
    return false

def printVar(i: int)
    print("{i}")

def printVarRef(i: int&)
    print("{i}")

```

(continues on next page)

(continued from previous page)

```
def setVar(var i: int&)
    i = i + 2
```

2.3.11 try/recover

```
stat ::= 'try' stat 'recover' visibility-block
```

The try statement encloses a block of code in which a panic condition can occur, such as a fatal runtime error or a panic function. The try-recover clause provides the panic-handling code.

It is important to understand that try/recover is not correct error handling code, and definitely not a way to implement control-flow. Much like in the Go language, this is really an invalid situation which should not normally happen in a production environment. Examples of potential exceptions are dereferencing a null pointer, indexing into an array out of bounds, etc.

2.3.12 panic

```
stat ::= 'panic' '(' [string-exp] ')'
```

Calling `panic` causes a runtime exception with string-exp available in the log.

2.3.13 global variables

```
stat ::= 'let|var' { shared } {private} '\n' indent id '=' expression
stat ::= 'let|var' { shared } {private} '\n' indent id '<-' expression
stat ::= 'let|var' { shared } {private} '\n' indent id ':=' expression
```

Declares a constant global variable. This variable is initialized once during initialization of the script (or each time when script init is manually called).

`shared` indicates that the constant is to be initialized once, and its memory is shared between multiple instances of the Daslang context.

`private` indicates that the variable is not visible outside of its module.

2.3.14 enum

```
enumerations ::= ( 'id' ) '\n'
stat ::= 'enum' id indent enumerations unindent
```

Declares an enumeration (see *Constants & Enumerations*).

2.3.15 Expression statement

```
stat ::= exp
```

In Daslang every expression is also allowed to be a statement. If so, the result of the expression is thrown away.

2.4 Expressions

2.4.1 Assignment

```
exp := exp '=' exp
exp := exp '<-' exp
exp := exp ':=' exp
```

Daslang implements 3 kind of assignment: copy assignment(=):

```
a = 10
```

Copy assignment is equivalent of C++ memcpy operation:

```
template <typename TT, typename QQ>
__forceinline void das_copy ( TT & a, const QQ b ) {
    static_assert(sizeof(TT)<=sizeof(QQ),"can't copy from smaller type");
    memcpy(&a, &b, sizeof(TT));
}
```

"Move" assignment:

```
var b = new Foo
var a: Foo?
a <- b
```

Move assignment nullifies source (b). It's main purpose is to correctly move ownership, and optimize copying if you don't need source for heavy types (such as arrays, tables). Some external handled types can be non assignable, but still moveable.

Move assignment is equivalent of C++ memcpy + memset operations:

```
template <typename TT, typename QQ>
__forceinline void das_move ( TT & a, const QQ & b ) {
    static_assert(sizeof(TT)<=sizeof(QQ),"can't move from smaller type");
    memcpy(&a, &b, sizeof(TT));
    memset((TT *)&b, 0, sizeof(TT));
}
```

"Clone" assignment:

```
a := b
```

Clone assignment is syntactic sugar for calling clone(var a: auto&; b: auto&) if it exists or basic assignment for POD types. It is also implemented for das_string, array and table types, and creates a 'deep' copy.

Some external handled types can be non assignable, but still cloneable (see *Clone*).

2.4.2 Operators

.. Operator

```
expr1 \.\. expr2
```

This is equivalent to *interval(expr1,expr2)*. By default *interval(a,b:int)* is implemented as *range(a,b)*, and *interval(a,b:uint)* is implemented as *urange(a,b)*. Users can define their own interval functions or generics.

?: Operator

```
exp := exp_cond '?' exp1 ':' exp2
```

This conditionally evaluate an expression depending on the result of an expression. If expr_cond is true, only exp1 will be evaluated. Similarly, if false, only exp2.

?? Null-coalescing operator

```
exp := exp1 '??' exp2
```

Conditionally evaluate exp2 depending on the result of exp1. The given code is equivalent to:

```
exp := (exp1 '!=' null) '?' *exp1 ':' exp2
```

It evaluates expressions until the first non-null value (just like | operators for the first 'true' one).

Operator precedence is also follows C# design, so that ?? has lower priority than |

?. and ?[- Null-propagation operator

```
exp := value '?.' key
```

If the value is not null, then dereferences the field 'key' for struct, otherwise returns null.

```
struct TestObjectFooNative
    fooData : int

struct TestObjectBarNative
    fooPtr: TestObjectFooNative?
    barData: float

def test
    var a: TestObjectFooNative?
    var b: TestObjectBarNative?
    var idummy: int
    var fdummy: float
    a?.fooData ?? idummy = 1 // will return reference to idummy, since a is null
    assert(idummy == 1)

    a = new TestObjectFooNative
    a?.fooData ?? idummy = 2 // will return reference to a.fooData, since a is now␣
↪not null
    assert(a.fooData == 2 & idummy == 1)
```

(continues on next page)

(continued from previous page)

```
    b = new TestObjectBarNative
    b?.fooPtr?.fooData ?? idummy = 3 // will return reference to idummy, since while␣
↪b is not null, but b.?barData is still null
    assert(idummy == 3)

    b.fooPtr <- a
    b?.fooPtr?.fooData ?? idummy = 4 // will return reference to b.fooPtr.fooData
    assert(b.fooPtr.fooData == 4 & idummy == 3)
```

Additionally, null propagation of index ?[can be used with tables:

```
var tab <- {{ "one"=>1; "two"=> 2 }}
let i = tab?["three"] ?? 3
print("i = {i}\n")
```

It checks both the container pointer and the availability of the key.

Arithmetic

```
exp:= 'exp' op 'exp'
```

Daslang supports the standard arithmetic operators `+, -, *, / and %`. It also supports compact operators `+=, -=, *=, /=, %=` and increment and decrement operators `++ and --`:

```
a += 2
// is the same as writing
a = a + 2
x++
// is the same as writing
x = x + 1
```

All operators are defined for numeric and vector types, i.e (u)int* and float* and double.

Relational

```
exp:= 'exp' op 'exp'
```

Relational operators in Daslang are : `==, <, <=, >, >=, !=`.

These operators return true if the expression is false and a value different than true if the expression is true.

Logical

```
exp := exp op exp
exp := '!' exp
```

Logical operators in Daslang are : `&&, ||, ^^, !, &&=, ||=, ^^=`.

The operator `&&` (logical and) returns false if its first argument is false, or otherwise returns its second argument. The operator `||` (logical or) returns its first argument if is different than false, or otherwise returns the second argument.

The operator `^^` (logical exclusive or) returns true if arguments are different, and false otherwise.

It is important to understand, that && and || will not necessarily 'evaluate' all their arguments. Unlike their C++ equivalents, &&= and ||= will also cancel evaluation of the right side.

The '!' (negation) operator will return false if the given value was true, or false otherwise.

Bitwise Operators

```
exp:= 'exp' op 'exp'
exp := '~' exp
```

Daslang supports the standard C-like bitwise operators `&`, `|`, `^`, `~`, `<<`, `>>`, `<<<`, `>>>`. Those operators only work on integer values.

Pipe Operators

```
exp:= 'exp' |> 'exp'
exp:= 'exp' <| 'exp'
```

Daslang supports pipe operators. Pipe operators are similar to 'call' expressions where the other expression is first argument.

```
def addX(a, b)
    assert(b == 2 || b == 3)
    return a + b
def test
    let t = 12 |> addX(2) |> addX(3)
    assert(t == 17)
    return true
```

```
def addOne(a)
    return a + 1

def test
    let t =  addOne() <| 2
    assert(t == 3)
```

The `lpipe` macro allows piping to the previous line:

```
require daslib/lpipe

def main
    print()
    lpipe() <| "this is string constant"
```

In the example above, the string constant will be piped to the print expression on the previous line. This allows piping of multiple blocks while still using significant whitespace syntax.

Operators precedence

post++ post-- . -> ?. ?[*(deref)	highest
|> <|	
is as	
- + ~ ! ++ --	
??	
/ * %	
+ -	
<< >> <<< >>>	
< <= > >=	
== !=	
&	
^	
|	
&&	
^^	
||	
? :	
+= = -= /= *= %= &= |= ^= <<= >>= <- <<<= >>>= &&= ||= ^^=	...
=>	
',' comma	lowest

2.4.3 Array Initializer

```
exp := '[['type[] [explist] ']]'
```

Creates a new fixed size array:

```
let a = [[int[] 1; 2]]      // creates array of two elements
let a = [[int[2] 1; 2]]     // creates array of two elements
var a = [[auto 1; 2]]       // creates which fully infers its own type
let a = [[int[2] 1; 2; 3]] // error, too many initializers
var a = [[auto 1]]          // int
var a = [[auto[] 1]]        // int[1]
```

Arrays can be also created with array comprehensions:

```
let q <- [[ for x in range(0, 10); x * x ]]
```

Similar syntax can be used to initialize dynamic arrays:

```
let a <- [{int[3] 1;2;3 }]                      // creates and initializes array<int>
let q <- [{ for x in range(0, 10); x * x }]     // comprehension which initializes␣
↪array<int>
```

Only dynamic multi-dimensional arrays can be initialized (for now):

```
var a <- [[auto [{int 1;2;3}]; [{int 4;5}]]]     // array<int>[2]
var a <- [{auto [{int 1;2;3}]; [{int 4;5}]}]     // array<array<int>>
```

(see *Arrays*, *Comprehensions*).

2.4.4 Struct, Class, and Handled Type Initializer

```
struct Foo
  x: int = 1
  y: int = 2

let fExplicit = [[Foo x = 13, y = 11]]              // x = 13, y = 11
let fPartial  = [[Foo x = 13]]                      // x = 13, y = 0
let fComplete = [[Foo() x = 13]]                    // x = 13, y = 2 with 'construct'␣
↪syntax
let aArray    = [[Foo() x=11,y=22; x=33; y=44]]     // array of Foo with 'construct'␣
↪syntax
```

Initialization also supports optional inline block:

```
var c = [[ Foo x=1, y=2 where $ ( var foo ) { print("{foo}"); } ]]
```

Classes and handled (external) types can also be initialized using structure initialization syntax. Classes and handled types always require constructor syntax, i.e. ().

(see *Structs*, *Classes*, *Handles*).

2.4.5 Tuple Initializer

Create new tuple:

```
let a = [[tuple<int;float;string> 1, 2.0, "3"]]     // creates typed tuple
let b = [[auto 1, 2.0, "3"]]                        // infers tuple type
let c = [[auto 1, 2.0, "3"; 2, 3.0, "4"]]           // creates array of tuples
```

(see *Tuples*).

2.4.6 Variant Initializer

Variants are created with a syntax similar to that of a structure:

```
variant Foo
    i : int
    f : float

let x = [[Foo i = 3]]
let y = [[Foo f = 4.0]]
let a = [[Foo[2] i=3; f=4.0]]    // array of variants
let z = [[Foo i = 3, f = 4.0]]   // syntax error, only one initializer
```

(see *Variants*).

2.4.7 Table Initializer

Tables are created by specifying key => value pairs separated by semicolon:

```
var a <- {{ 1=>"one"; 2=>"two" }}
var a <- {{ 1=>"one"; 2=>2 }}        // error, type mismatch
```

(see *Tables*).

2.5 Temporary types

Temporary types are designed to address lifetime issues of data, which are exposed to Daslang directly from C++.

Let's review the following C++ example:

```
void peek_das_string(const string & str, const TBlock<void,TTemporary<const char *>> &
↪ block, Context * context) {
    vec4f args[1];
    args[0] = cast<const char *>::from(str.c_str());
    context->invoke(block, args, nullptr);
}
```

The C++ function here exposes a pointer a to c-string, internal to std::string. From Daslang's perspective, the declaration of the function looks like this:

```
def peek ( str : das_string; blk : block<(arg:string#):void> )
```

Where string# is a temporary version of a Daslang string type.

The main idea behind temporary types is that they can't *escape* outside of the scope of the block they are passed to.

Temporary values accomplish this by following certain rules.

Temporary values can't be copied or moved:

```
def sample ( var t : das_string )
    var s : string
    peek(t) <| $ ( boo : string# )
        s = boo // error, can't copy temporary value
```

Temporary values can't be returned or passed to functions, which require regular values:

```
def accept_string(s:string)
    print("s={s}\n")

def sample ( var t : das_string )
    peek(t) <| $ ( boo : string# )
        accept_string(boo) // error
```

This causes the following error:

```
30304: no matching functions or generics accept_string ( string const&# )
candidate function:
        accept_string ( s : string const ) : void
                invalid argument s. expecting string const, passing string const&#
```

Values need to be marked as `implicit` to accept both temporary and regular values. These functions implicitly promise that the data will not be cached (copied, moved) in any form:

```
def accept_any_string(s:string implicit)
    print("s={s}\n")

def sample ( var t : das_string )
    peek(t) <| $ ( boo : string# )
        accept_any_string(boo)
```

Temporary values can and are intended to be cloned:

```
def sample ( var t : das_string )
    peek(t) <| $ ( boo : string# )
        var boo_clone : string := boo
        accept_string(boo_clone)
```

Returning a temporary value is an unsafe operation.

A pointer to the temporary value can be received for the corresponding scope via the `safe_addr` macro:

```
require daslib/safe_addr

def foo
    var a = 13
    ...
    var b = safe_addr(a)    // b is int?#, and this operation does not require unsafe
    ...
```

2.6 Built-in Functions

Builtin functions are function-like expressions that are available without any modules. They implement inherent mechanisms of the language, in available in the AST as separate expressions. They are different from standard functions (see built-in functions).

2.6.1 Invoke

invoke (*block_or_function*, *arguments*)
: `invoke` calls a block, lambda, or pointer to a function (*block_or_function*) with the provided list of arguments.

(see *Functions*, *Blocks*, *Lambdas*).

2.6.2 Misc

assert (*x*, *str*)
: `assert` causes an application-defined assert if the *x* argument is false. `assert` can and probably will be removed from release builds. That's why it will not compile if the *x* argument has side effects (for example, calling a function with side effects).

verify (*x*, *str*)
: `verify` causes an application-defined assert if the *x* argument is false. The `verify` check can be removed from release builds, but execution of the *x* argument stays. That's why verify, unlike `assert`, can have side effects in evaluating *x*.

static_assert (*x*, *str*)
: `static_assert` causes the compiler to stop compilation if the *x* argument is false. That's why *x* has to be a compile-time known constant. ``static_assert``s are removed from compiled programs.

concept_assert (*x*, *str*)
`concept_assert` is similar to `static_assert`, but errors will be reported one level above the assert. That way applications can report contract errors.

debug (*x*, *str*)
`debug` prints string *str* and the value of *x* (like print). However, `debug` also returns the value of *x*, which makes it suitable for debugging expressions:

```
let mad = debug(x, "x") * debug(y, "y") + debug(z, "z") // x*y + z
```

2.7 Clone

Clone is designed to create a deep copy of the data. Cloning is invoked via the clone operator `:=`:

```
a := b
```

Cloning can be also invoked via the clone initializer in a variable declaration:

```
var x := y
```

This in turn expands into `clone_to_move`:

```
var x <- clone_to_move(y)
```

(see *clone_to_move*).

2.7.1 Cloning rules and implementation details

Cloning obeys several rules.

Certain types like blocks, lambdas, and iterators can't be cloned at all.

However, if a custom clone function exists, it is immediately called regardless of the type's cloneability:

```
struct Foo
    a : int

def clone ( var x : Foo; y : Foo )
    x.a = y.a
    print("cloned\n")

var l = [[Foo a=1]]
var cl : Foo
cl := l                    // invokes clone(cl,l)
```

Cloning is typically allowed between regular and temporary types (see *Temporary types*).

POD types are copied instead of cloned:

```
var a,b : int
var c,d : int[10]
a := b
c := d
```

This expands to:

```
a = b
c = d
```

Handled types provide their own clone functionality via `canClone`, `simulateClone`, and appropriate `das_clone` C++ infrastructure (see *Handles*).

For static arrays, the `clone_dim` generic is called, and for dynamic arrays, the `clone` generic is called. Those in turn clone each of the array elements:

```
struct Foo
    a : array<int>
    b : int

var a, b : array<Foo>
b := a
var c, d : Foo[10]
c := d
```

This expands to:

```
def builtin`clone ( var a:array<Foo aka TT> explicit; b:array<Foo> const )
    resize(a,length(b))
    for aV,bV in a,b
        aV := bV

def builtin`clone_dim ( var a:Foo[10] explicit; b:Foo const[10] implicit explicit )
    for aV,bV in a,b
        aV := bV
```

For tables, the `clone` generic is called, which in turn clones its values:

```
var a, b : table<string;Foo>
b := a
```

This expands to:

```
def builtin`clone ( var a:table<string aka KT;Foo aka VT> explicit; b:table<string;Foo>
↪ const )
    clear(a)
    for k,v in keys(b),values(b)
        a[k] := v
```

For structures, the default `clone` function is generated, in which each element is cloned:

```
struct Foo
    a : array<int>
    b : int
```

This expands to:

```
def clone ( var a:Foo explicit; b:Foo const )
    a.a := b.a
    a.b = b.b   // note copy instead of clone
```

For tuples, each individual element is cloned:

```
var a, b : tuple<int;array<int>;string>
b := a
```

This expands to:

```
def clone ( var dest:tuple<int;array<int>;string> -const; src:tuple<int;array<int>;
↪string> const -const )
    dest._0 = src._0
    dest._1 := src._1
    dest._2 = src._2
```

For variants, only the currently active element is cloned:

```
var a, b : variant<i:int;a:array<int>;s:string>
b := a
```

This expands to:

```
def clone ( var dest:variant<i:int;a:array<int>;s:string> -const; src:variant<i:int;
↪a:array<int>;s:string> const -const )
    if src is i
        set_variant_index(dest,0)
        dest.i = src.i
    elif src is a
        set_variant_index(dest,1)
        dest.a := src.a
    elif src is s
        set_variant_index(dest,2)
        dest.s = src.s
```

2.7.2 clone_to_move implementation

`clone_to_move` is implemented via regular generics as part of the builtin module:

```
def clone_to_move(clone_src:auto(TT)) : TT -const
    var clone_dest : TT
    clone_dest := clone_src
    return <- clone_dest
```

Note that for non-cloneable types, Daslang will not promote **:=** initialize into `clone_to_move`.

2.8 Unsafe

The `unsafe` keyword denotes unsafe contents, which is required for operations, but could potentially crash the application:

```
unsafe
    let px = addr(x)
```

Expressions (and subexpressions) can also be unsafe:

```
let px = unsafe(addr(x))
```

Unsafe is followed by a block which can include those operations. Nested unsafe sections are allowed. Unsafe is not inherited in lambda, generator, or local functions; it is, however, inherited in local blocks.

Individual expressions can cause a *CompilationError::unsafe* error, unless they are part of the unsafe section. Additionally, macros can explicitly set the *ExprGenFlags::alwaysSafe* flag.

The address of expression is unsafe:

```
unsafe
    let a : int
    let pa = addr(a)
    return pa                                // accessing *pa can potentially corrupt␣
↪stack
```

Lambdas or generators require unsafe sections for the implicit capture by move or by reference:

```
var a : array<int>
unsafe
    var counter <- @ <| (extra:int) : int
        return a[0] + extra                  // a is implicitly moved
```

Deleting any pointer requires an unsafe section:

```
var p = new Foo()
var q = p
unsafe
    delete p                                 // accessing q can potentially corrupt␣
↪memory
```

Upcast and reinterpret cast require an unsafe section:

```
unsafe
    return reinterpret<void?> 13             // reinterpret can create unsafe pointers
```

Indexing into a pointer is unsafe:

```
unsafe
    var p = new Foo()
    return p[13]                             // accessing out of bounds pointer can␣
↪potentially corrupt memory
```

A safe index is unsafe when not followed by the null coalescing operator:

```
var a = {{ 13 => 12 }}
unsafe
    var t = a?[13] ?? 1234                   // safe
    return a?[13]                            // unsafe; safe index is a form of 'addr'␣
↪operation
                                             // it can create pointers to temporary␣
↪objects
```

Variant `?as` on local variables is unsafe when not followed by the null coalescing operator:

```
unsafe
    return a ?as Bar                         // safe as is a form of 'addr' operation
```

Variant `.?field` is unsafe when not followed by the null coalescing operator:

```
unsafe
    return a?.Bar                            // safe navigation of a variant is a form␣
↪of 'addr' operation
```

Variant `.field` is unsafe:

```
unsafe
    return a.Bar                            // this is potentially a reinterpret cast
```

Certain functions and operators are inherently unsafe or marked unsafe via the [unsafe_operation] annotation:

```
unsafe
    var a : int?
    a += 13                                 // pointer arithmetic can create invalid␣
↪pointers
    var boo : int[13]
    var it = each(boo)                      // each() of array is unsafe, for it does␣
↪not capture
```

Moving from a smart pointer value requires unsafe, unless that value is the 'new' operator:

```
unsafe
    var a <- new TestObjectSmart()          // safe, its explicitly new
    var b <- someSmartFunction()            // unsafe since lifetime is not obvious
    b <- a                                  // safe, values are not lost
```

Moving or copying classes is unsafe:

```
def foo ( var b : TestClass )
    unsafe
        var a : TestClass
        a <- b                              // potentially moving from derived class
```

Local class variables are unsafe:

```
unsafe
    var g = Goo()                           // potential lifetime issues
```

2.9 implicit

implicit keyword is used to specify that type can be either temporary or regular type, and will be treated as defined. For example:

```
def foo ( a : Foo implicit )     // a will be treated as Foo, but will also accept Foo
↪# as argument
def foo ( a : Foo# implicit )    // a will be treated as Foo#, but will also accept␣
↪Foo as argument
```

Unfortunately implicit conversions like this are unsafe, so *implicit* is unsafe by definition.

2.10 Table

Tables are associative containers implemented as a set of key/value pairs:

```
var tab: table<string; int>
tab["10"] = 10
tab["20"] = 20
tab["some"] = 10
tab["some"] = 20  // replaces the value for 'some' key
```

There are several relevant builtin functions: `clear`, `key_exists`, `get`, and `erase`. `get` works with block as last argument, and returns if value has been found. It can be used with the rbpipe operator:

```
var tab <- {{ "one"=>1; "two"=>2 }}
let found = get(tab,"one") <| $(val)
            assert(val==1)
    assert(found)
```

There is non constant version available as well:

```
[export]
def main
    var tab <- {{ "one"=>1; "two"=>2 }}
    let found = get(tab,"one") <| $(var val)
        val = 123
    let t = tab["one"]
    assert(t==123)
```

Tables (as well as arrays, structs, and handled types) are passed to functions by reference only.

Tables cannot be assigned, only cloned or moved.

```
def clone_table(var a, b: table<string, int>)
  a := b       // a is not deep copy of b
  clone(a, b) // same as above
  a = b        // error

def move_table(var a, b: table<string, int>)
  a <- b  //a is no points to same data as b, and b is empty.
```

Table keys can be not only strings, but any other 'workhorse' type as well.

Tables can be constructed inline:

```
let tab <- {{ "one"=>1; "two"=>2 }}
```

This is syntax sugar for:

```
let tab : table<string;int> <- to_table_move([[tuple<string;int>[2] "one"=>1; "two"=>
↪2]])
```

Alternative syntax is:

```
let tab <- table("one"=>1, "two"=>2)
let tab <- table<string; int>("one"=>1, "two"=>2)
```

Table which holds no associative data can also be declared:

```
var tab : table<int>
tab |> insert(1)        // this is how we insert a key into such table
```

Table can be iterated over with the `for` loop:

```
var tab <- {{ "one"=>1; "two"=>2 }}
for key, value in keys(tab), values(tab)
    print("key: {key}, value: {value}\n")
```

Table which holds no associative data only has keys.

2.11 Array

An array is a sequence of values indexed by an integer number from 0 to the size of the array minus 1. An array's elements can be obtained by their index.

```
var a = [[int[4] 1; 2; 3; 4]] // fixed size of array is 4, and content is [1, 2, 3, 4]
assert(a[0] == 1)

var b: array<int>
push(b,1)
assert(b[0] == 1)
```

Alternative syntax is:

```
var a = fixed_array(1,2,3,4)
var a = fixed_array<int>(1,2,3,4)
```

There are static arrays (of fixed size, allocated on the stack), and dynamic arrays (size is dynamic, allocated on the heap):

```
var a = [[int[4] 1; 2; 3; 4]] // fixed size of array is 4, and content is [1, 2, 3, 4]
var b: array<string>          // empty dynamic array
push(b, "some")               // now it is 1 element of "some"
b |> push("some")             // same as above line, but using pipe operator
```

Dynamic sub-arrays can be created out of any array type via range indexing:

```
var a  = [[int[4] 1; 2; 3; 4]]
let b <- a[1..3]               //  b is [{int 2;3}]
```

In reality *a[b]*, where b is a range, is equivalent to *subarray(a, b)*.

Resizing, insertion, and deletion of dynamic arrays and array elements is done through a set of standard functions (see built-in functions).

The relevant builtin functions are: `push`, `push_clone`, `emplace`, `reserve`, `resize`, `erase`, `length`, `clear`, and `capacity`.

Arrays (as well as tables, structures, and handled types) are passed to functions by reference only.

Arrays cannot be copied; only cloned or moved.

```
def clone_array(var a, b: array<string>)
  a := b      // a is not a deep copy of b
  clone(a, b) // same as above
```

(continues on next page)

(continued from previous page)

```
def move_array(var a, b: array<string>)
  a <- b  // a is no points to same data as b, and b is empty.
```

Arrays can be constructed inline:

```
let arr = [[auto 1.; 2.; 3.; 4.5]]
```

This expands to:

```
let arr : float[4] = [[float[4] 1.; 2.; 3.; 4.5]]
```

Dynamic arrays can also be constructed inline:

```
let arr <- [{auto "one"; "two"; "three"}]
```

This is syntactic equivalent to:

```
let arr : array<string> <- to_array_move([[string[3] "one"; "two"; "three"]])
```

Alternative syntax is:

```
let arr <- array(1., 2., 3., 4.5)
let arr <- array<float>(1., 2., 3., 4.5)
```

If only one element is specified, local data construction is of that element:

```
let i1 = [[int 1]]      // same is the line bellow
let i2 = 1
```

To create an array of an unspecified type, use [] syntax:

```
let ai1 <- [[int[] 1]]  // actually [[int[1] 1]]
let ai2 <- [[auto[] 1]  // same as above
```

When array elements can't be copied, use `push_clone` to insert a clone of a value, or `emplace` to move it in.

`resize` can potentially create new array elements. Those elements are initialized with 0.

`reserve` is there for performance reasons. Generally, array capacity doubles, if exceeded. `reserve` allows you to specify the exact known capacity and significantly reduce the overhead of multiple `push` operations.

It's possible to iterate over an array via a regular `for` loop:

```
for x in [[int[] 1;2;3;4]]
        print("x = {x}\n")
```

Additionally, a collection of unsafe iterators is provided:

```
def each ( a : auto(TT)[] ) : iterator<TT&>
def each ( a : array<auto(TT)> ) : iterator<TT&>
```

The reason both are unsafe operations is that they do not capture the array.

Search functions are available for both static and dynamic arrays:

```
def find_index ( arr : array<auto(TT)> implicit; key : TT )
def find_index ( arr : auto(TT)[] implicit; key : TT )
def find_index_if ( arr : array<auto(TT)> implicit; blk : block<(key:TT):bool> )
def find_index_if ( arr : auto(TT)[] implicit; blk : block<(key:TT):bool> )
```

2.12 Function

Functions pointers are first class values, like integers or strings, and can be stored in table slots, local variables, arrays, and passed as function parameters. Functions themselves are declarations (much like in C++).

2.12.1 Function declaration

Functions are similar to those in most other typed languages:

```
def twice(a: int): int
    return a+a
```

Completely empty functions (without arguments) can be also declared:

```
def foo
    print("foo")

//same as above
def foo()
    print("foo")
```

Daslang can always infer a function's return type. Returning different types is a compilation error:

```
def foo(a:bool)
    if a
        return 1
    else
        return 2.0  // error, expecting int
```

Publicity

Functions can be *private* or *public*

```
def private foo(a:bool)

def public bar(a:float)
```

If not specified, functions inherit module publicity (i.e. in public modules functions are public, and in private modules functions are private).

Function calls

You can call a function by using its name and passing in all its arguments (with the possible omission of the default arguments):

```
def foo(a, b: int)
    return a + b

def bar
    foo(1, 2) // a = 1, b = 2
```

Named Arguments Function call

You can also call a function by using its name and passing all aits rguments with explicit names (with the possible omission of the default arguments):

```
def foo(a, b: int)
    return a + b

def bar
    foo([a = 1, b = 2])  // same as foo(1, 2)
```

Named arguments should be still in the same order:

```
def bar
    foo([b = 1, a = 2])  // error, out of order
```

Named argument calls increase the readability of callee code and ensure correctness in refactorings of the existing functions. They also allow default values for arguments other than the last ones:

```
def foo(a:int=13; b: int)
    return a + b

def bar
    foo([b = 2])  // same as foo(13, 2)
```

Function pointer

Pointers to a function use a similar declaration to that of a block or lambda:

```
function_type ::= function { optional_function_type }
optional_function_type ::= < { optional_function_arguments } { : return_type } >
optional_function_arguments := ( function_argument_list )
function_argument_list := argument_name : type | function_argument_list ; argument_
↪name : type

function < (arg1:int;arg2:float&):bool >
```

Function pointers can be obtained by using the `@@` operator:

```
def twice(a:int)
    return a + a

let fn = @@twice
```

When multiple functions have the same name, a pointer can be obtained by explicitly specifying signature:

```
def twice(a:int)
    return a + a

def twice(a:float)  // when this one is required
    return a + a

let fn = @@<(a:float):float> twice
```

Function pointers can be called via `invoke`:

```
let t = invoke(fn, 1)  // t = 2
```

Nameless functions

Pointers to nameless functions can be created with a syntax similar to that of lambdas or blocks (see *Blocks*):

```
let fn <- @@ <| ( a : int )
    return a + a
```

Nameless local functions do not capture variables at all:

```
var count = 1
let fn <- @@ <| ( a : int )
    return a + count            // compilation error, can't locate variable count
```

Internally, a regular function will be generated:

```
def _localfunction_thismodule_8_8_1`function ( a:int const ) : int
        return a + a

let fn:function<(a:int const):int> const <- @@_localfunction_thismodule_8_8_1`function
```

Generic functions

Generic functions are similar to C++ templated functions. Daslang will instantiate them during the infer pass of compilation:

```
def twice(a)
    return a + a

let f = twice(1.0)  // 2.0 float
let i = twice(1)    // 2 int
```

Generic functions allow code similar to dynamically-typed languages like Python or Lua, while still enjoying the performance and robustness of strong, static typing.

Generic function addresses cannot be obtained.

Unspecified types can also be written via `auto` notation:

```
def twice(a:auto)   // same as 'twice' above
    return a + a
```

Generic functions can specialize generic type aliases, and use them as part of the declaration:

```
def twice(a:auto(TT)) : TT
    return a + a
```

In the example above, alias TT is used to enforce the return type contract.

Type aliases can be used before the corresponding auto:

```
def summ(base : TT; a:auto(TT)[] )
    var s = base
    for x in a
        s += x
    return s
```

In the example above, TT is inferred from the type of the passed array a, and expected as a first argument base. The return type is inferred from the type of s, which is also TT.

Function overloading

Functions can be specialized if their argument types are different:

```
def twice(a: int)
    print("int")
    return a + a
def twice(a: float)
    print("float")
    return a + a

let i = twice(1)    // prints "int"
let f = twice(1.0)  // prints "float"
```

Declaring functions with the same exact argument list is compilation time error.

Functions can be partially specialized:

```
def twice(a:int)        // int
    return a + a
def twice(a:float)      // float
    return a + a
def twice(a:auto[])     // any array
    return length(a)*2
def twice(a)            // any other case
    return a + a
```

Daslang uses the following rules for matching partially specialized functions:

1. Non-auto is more specialized than auto.
2. If both are non-auto, the one without a cast is more specialized.
3. Ones with arrays are more specialized than ones without. If both have an array, the one with the actual value is more specialized than the one without.
4. Ones with a base type of autoalias are less specialized. If both are autoalias, it is assumed that they have the same level of specialization.
5. For pointers and arrays, the subtypes are compared.
6. For tables, tuples and variants, subtypes are compared, and all must be the same or equally specialized.

7. For functions, blocks, or lambdas, subtypes and return types are compared, and all must be the same or equally specialized.

When matching functions, Daslang picks the ones which are most specialized and sorts by substitute distance. Substitute distance is increased by 1 for each argument if a cast is required for the LSP (Liskov substitution principle). At the end, the function with the least distance is picked. If more than one function is left for picking, a compilation error is reported.

Function specialization can be limited by contracts (contract macros):

```
[expect_any_array(blah)]  // array<foo>, [], or dasvector`.... or similar
def print_arr ( blah )
    for i in range(length(blah))
        print("{blah[i]}\n")
```

In the example above, only arrays will be matched.

Its possible to do boolean logic operations on the contracts:

```
[expect_any_tuple(blah) || expect_any_variant(blah)]
def print_blah ...
```

In the example above print_blah will accept any tuple or variant. Available logic operations are *!*, *&&*, || and ^^.

LSP can be explicitly prohibited for a particular function argument via the *explicit* keyword:

```
def foo ( a : Foo explicit ) // will accept Foo, but not any subtype of Foo
```

Default Parameters

Daslang's functions can have default parameters.

A function with default parameters is declared as follows:

```
def test(a, b: int; c: int = 1; d: int = 1)
    return a + b + c + d
```

When the function *test* is invoked and the parameters *c* or *d* are not specified, the compiler will generate a call with default value to the unspecified parameter. A default parameter can be any valid compile-time const Daslang expression. The expression is evaluated at compile-time.

It is valid to declare default values for arguments other than the last one:

```
def test(c: int = 1; d: int = 1; a, b: int) // valid!
    return a + b + c + d
```

Calling such functions with default arguments requires a named arguments call:

```
test(2, 3)           // invalid call, a,b parameters are missing
test([a = 2, b = 3]) // valid call
```

Default arguments can be combined with overloading:

```
def test(c: int = 1; d: int = 1; a, b: int)
    return a + b + c + d
def test(a, b: int) // now test(2, 3) is valid call
    return test([a = a, b = b])
```

2.12.2 OOP-style calls

There are no methods or function members of structs in Daslang. However, code can be easily written "OOP style" by using the right pipe operator `|>`:

```
struct Foo
    x, y: int = 0

def setXY(var thisFoo: Foo; x, y: int)
    thisFoo.x = x
    thisFoo.y = y
...
var foo:Foo
foo |> setXY(10, 11)   // this is syntactic sugar for setXY(foo, 10, 11)
setXY(foo, 10, 11)     // exactly same as above line
```

(see *Structs*).

2.12.3 Tail Recursion

Tail recursion is a method for partially transforming recursion in a program into iteration: it applies when the recursive calls in a function are the last executed statements in that function (just before the return).

Currently, Daslang doesn't support tail recursion. It is implied that a Daslang function always returns.

2.12.4 Operator Overloading

Daslang allows you to overload operators, which means that you can define custom behavior for operators when used with your own data types. To overload an operator, you need to define a special function with the name of the operator you want to overload. Here's the syntax:

```
def operator <operator>(<arguments>) : <return_type>
    # Implementation here
```

In this syntax, <operator> is the name of the operator you want to overload (e.g. +, -, *, /, ==, etc.), <arguments> are the parameters that the operator function takes, and <return_type> is the return type of the operator function.

For example, here's how you could overload the == operator for a custom struct called iVec2:

```
struct iVec2:
    x, y: int

def operator==(a, b: iVec2)
    return (a.x == b.x) && (a.y == b.y)
```

In this example, we define a structure called iVec2 with two integer fields (x and y).

We then define an operator== function that takes two parameters (a and b) of type iVec2. This function returns a bool value indicating whether a and b are equal. The implementation checks whether the x and y components of a and b are equal using the == operator.

With this operator overloaded, you can now use the == operator to compare iVec2 objects, like this:

```
let v1 = iVec2(1, 2)
let v2 = iVec2(1, 2)
let v3 = iVec2(3, 4)
```

(continues on next page)

(continued from previous page)

```
print("{v1==v2}") # prints "true"
print("{v1==v3}") # prints "false"
```

In this example, we create three iVec2 objects and compare them using the == operator. The first comparison (v1 == v2) returns true because the x and y components of v1 and v2 are equal. The second comparison (v1 == v3) returns false because the x and y components of v1 and v3 are not equal.

2.12.5 Overloading the '.' and '?.' operators

Daslang allows you to overload the dot . operator, which is used to access fields of structure or a class. To overload the dot . operator, you need to define a special function with the name operator . Here's the syntax:

```
def operator.(<object>: <type>; <name>: string) : <return_type>
    # Implementation here
```

Alternatively you can specify field explicitly:

```
def operator.<name> (<object>: <type>) : <return_type>
    # Implementation here
```

In this syntax, <object> is the object you want to access, <type> is the type of the object, <name> is the name of the field you want to access, and <return_type> is the return type of the operator function.

Operator ?. works in a similar way.

For example, here's how you could overload the dot . operator for a custom structure called Goo:

```
struct Goo
    a: string

def operator.(t: Goo, name: string) : string
    return "{name} = {t . . a}"

def operator. length(t: Goo) : int
    return length(t . . a)
```

In this example, we define a struct called Goo with a string field called a.

We then define two operator. functions:

The first one takes two parameters (t and name) and returns a string value that contains the name of the field or method being accessed (name) and the value of the a field of the Goo object (t.a). The second one takes one parameter (t) and returns the length of the a field of the Goo object (t.a). With these operators overloaded, you can now use the dot . operator to access fields and methods of a Goo object, like this:

```
var g = [[Goo a ="hello"]]
var field = g.a
var length = g.length
```

In this example, we create an instance of the Goo struct and access its world field using the dot . operator. The overloaded operator. function is called and returns the string "world = hello". We also access the length property of the Goo object using the dot . operator. The overloaded operator. length function is called and returns the length of the a field of the Goo object (5 in this case).

The . . syntax is used to access the fields of a structure or a class while bypassing overloaded operations.

2.12.6 Overloading accessors

Daslang allows you to overload accessors, which means that you can define custom behavior for accessing fields of your own data types. Here is an example of how to overload the accessor for a custom struct called Foo:

```
struct Foo
    dir : float3
def operator . length ( foo : Foo )
    return length(foo.dir)
def operator . length := ( var foo:Foo; value:float )
    foo.dir = normalize(foo.dir) * value
[export]
def main
    var f = [[Foo dir=float3(1,2,3)]]
    print("length = {f.length} // {f}\n")
    f.length := 10.
    print("length = {f.length} // {f}\n")
```

It now has accessor *length* which can be used to get and set the length of the *dir* field.

Classes allow to overload accessors for properties as well:

```
class Foo
    dir : float3
    def const operator . length
        return length(dir)
    def operator . length := ( value:float )
        dir = normalize(dir) * value
```

2.13 Modules

Modules provide infrastructure for code reuse, as well as mechanism to expose C++ functionality to Daslang. A module is a collection of types, constants, and functions. Modules can be native to Daslang, as well as built-in.

To request a module, use the `require` keyword:

```
require math
require ast public
require daslib/ast_boost
```

The `public` modifier indicates that included model is visible to everything including current module.

Module names may contain `/` and `.` symbols. The project is responsible for resolving module names into file names (see *Project*).

2.13.1 Native modules

A native module is a separate Daslang file, with an optional `module` name:

```
module custom       // specifies module name
...
def foo             // defines function in module
...
```

If not specified, the module name defaults to that of the file name.

Modules can be *private* or *public*:

```
module Foo private

module Foo public
```

Default publicity of the functions, structures, or enumerations are that of the module (i.e. if the module is public and a function's publicity is not specified, that function is public).

2.13.2 Builtin modules

Builtin modules are the way to expose C++ functionality to Daslang (see *Builtin modules*).

2.13.3 Shared modules

Shared modules are modules that are shared between compilation of multiple contexts. Typically, modules are compiled anew for each context, but when the 'shared' keyword is specified, the module gets promoted to a builtin module:

```
module Foo shared
```

That way only one instance of the module is created per compilation environment. Macros in shared modules can't expect the module to be unique, since sharing of the modules can be disabled via the code of policies.

2.13.4 Module function visibility

When calling a function, the name of the module can be specified explicitly or implicitly:

```
let s1 = sin(0.0)            // implicit, assumed math::sin
let s2 = math::sin(0.0)      // explicit, always math::sin
```

If the function does not exist in that module, a compilation error will occur. If the function is private or not directly visible, a compilation error will occur. If multiple functions match implicit function, compilation error will occur.

Module names _ and __ are reserved to specify the *current module* and the *current module only*, respectively. Its particularly important for generic functions, which are always instanced as private functions in the current module:

```
module b

[generic]
def from_b_get_fun_4()
    return _::fun_4()       //  call `fun_4', as if it was implicitly called from b

[generic]
def from_b_get_fun_5()
    return __::fun_5()      // always b::fun_5
```

Specifying an empty prefix is the same as specifying no prefix.

Without the _ or __ module prefixes, overwritten functions would not be visible from generics. That is why the `:=` and `delete` operators are always replaced with `_::clone` or `_::finalize` calls.

2.14 Block

Blocks are nameless functions which captures the local context by reference. Blocks offer significant performance advantages over lambdas (see *Lambda*).

The block type can be declared with a function-like syntax:

```
block_type ::= block { optional_block_type }
optional_block_type ::= < { optional_block_arguments } { : return_type } >
optional_block_arguments := ( block_argument_list )
block_argument_list := argument_name : type | block_argument_list ; argument_name :␣
↪type

block < (arg1:int;arg2:float&):bool >
```

Blocks capture the current stack, so blocks can be passed, but never returned. Block variables can only be passed as arguments. Global or local block variables are prohibited; returning the block type is also prohibited:

```
def goo ( b : block )
    ...

def foo ( b : block < (arg1:int; arg2:float&) : bool >
    ...
```

Blocks can be called via `invoke`:

```
def radd(var ext:int&;b:block<(var arg:int&):int>):int
    return invoke(b,ext)
```

Typeless blocks are typically declared via pipe syntax:

```
goo() <|                                        // block without arguments
    print("inside goo")
```

Similarly typed blocks are typically declared via pipe syntax:

```
var v1 = 1                                      // block with arguments
res = radd(v1) <| $(var a:int&):int
    return a++
```

Blocks can also be declared via inline syntax:

```
res = radd(v1, $(var a:int&) : int { return a++; }) // equivalent to example above
```

There is a simplified syntax for blocks that only contain a return expression:

```
res = radd(v1, $(var a:int&) : int => a++ )          // equivalent to example above
```

If a block is sufficiently specified in the generic or function, block types will be automatically inferred:

```
res = radd(v1, $(a) => a++ )                         // equivalent to example above
```

Nested blocks are allowed:

```
def passthroughFoo(a:Foo; blk:block<(b:Foo):void>)
    invoke(blk,a)

```

(continues on next page)

(continued from previous page)

```
passthrough(1) <| $ ( a )
    assert(a==1)
    passthrough(2) <| $ ( b )
        assert(a==1 && b==2)
        passthrough(3) <| $ ( c )
            assert(a==1 && b==2 && c==3)
```

Loop control expressions are not allowed to cross block boundaries:

```
while true
    take_any() <|
        break                   // 30801, captured block can't break outside the block
```

Blocks can have annotations:

```
def queryOne(dt:float=1.0f)
    testProfile::queryEs() <| $ [es] (var pos:float3&;vel:float3 const)  // [es] is␣
↪annotation
        pos += vel * dt
```

Block annotations can be implemented via appropriate macros (see *Macro*).

Local block variables are allowed:

```
var blk = $ <| ( a, b : int )
    return a + b
verify ( 3 == invoke(blk,1,2) )
verify ( 7 == invoke(blk,3,4) )
```

They can't be copied, or moved.

2.15 Lambda

Lambdas are nameless functions which capture the local context by clone, copy, or reference. Lambdas are slower than blocks, but allow for more flexibility in lifetime and capture modes (see *Blocks*).

The lambda type can be declared with a function-like syntax:

```
lambda_type ::= lambda { optional_lambda_type }
optional_lambda_type ::= < { optional_lambda_arguments } { : return_type } >
optional_lambda_arguments := ( lambda_argument_list )
lambda_argument_list := argument_name : type | lambda_argument_list ; argument_name :␣
↪type

lambda < (arg1:int;arg2:float&):bool >
```

Lambdas can be local or global variables, and can be passed as an argument by reference. Lambdas can be moved, but can't be copied or cloned:

```
def foo ( x : lambda < (arg1:int;arg2:float&):bool > )
    ...
    var y <- x
    ...
```

Lambdas can be invoked via `invoke`:

```
def inv13 ( x : lambda < (arg1:int):int > )
    return invoke(x,13)
```

Lambdas are typically declared via pipe syntax:

```
var CNT = 0
let counter <- @ <| (extra:int) : int
    return CNT++ + extra
let t = invoke(counter,13)
```

There are a lot of similarities between lambda and block declarations. The main difference is that blocks are specified with `$` symbol, while lambdas are specified with `@` symbol. Lambdas can also be declared via inline syntax. There is a similar simplified syntax for the lambdas containing return expression only. If a lambda is sufficiently specified in the generic or function, its types will be automatically inferred (see *Blocks*).

2.15.1 Capture

Unlike blocks, lambdas can specify their capture types explicitly. There are several available types of capture:

- by copy
- by move
- by clone
- by reference

Capturing by reference requires unsafe.

By default, capture by copy will be generated. If copy is not available, unsafe is required for the default capture by move:

```
var a1 <- [{int 1;2}]
var a2 <- [{int 1;2}]
var a3 <- [{int 1;2}]
unsafe  // required do to capture of a1 by reference
        var lam <- @ <| [[&a1,<-a2,:=a3]]
                push(a1,1)
                push(a2,1)
                push(a3,1)
        invoke(lam)
```

Lambdas can be deleted, which cause finalizers to be called on all captured data (see *Finalizers*):

```
delete lam
```

Lambdas can specify a custom finalizer which is invoked before the default finalizer:

```
var CNT = 0
var counter <- @ <| (extra:int) : int
    return CNT++ + extra
finally
    print("CNT = {CNT}\n")
var x = invoke(counter,13)
delete counter                    // this is when the finalizer is called
```

2.15.2 Iterators

Lambdas are the main building blocks for implementing custom iterators (see *Iterators*).

Lambdas can be converted to iterators via the `each` or `each_ref` functions:

```
var count = 0
let lam <- @ <| (var a:int &) : bool
    if count < 10
        a = count++
        return true
    else
        return false
for x,tx in each(lam),range(0,10)
    assert(x==tx)
```

To serve as an iterator, a lambda must

- have single argument, which is the result of the iteration for each step
- have boolean return type, where `true` means continue iteration, and `false` means stop

A more straightforward way to make iterator is with generators (see *Generators*).

2.15.3 Implementation details

Lambdas are implemented by creating a nameless structure for the capture, as well as a function for the body of the lambda.

Let's review an example with a singled captured variable:

```
var CNT = 0
let counter <- @ <| (extra:int) : int
    return CNT++ + extra
```

Daslang will generated the following code:

Capture structure:

```
struct _lambda_thismodule_7_8_1
    __lambda : function<(__this:_lambda_thismodule_7_8_1;extra:int const):int> = @@_
↪lambda_thismodule_7_8_1`function
    __finalize : function<(__this:_lambda_thismodule_7_8_1? -const):void> = @@_lambda_
↪thismodule_7_8_1`finalizer
    CNT : int
```

Body function:

```
def _lambda_thismodule_7_8_1`function ( var __this:_lambda_thismodule_7_8_1; extra:int␣
↪const ) : int
    with __this
        return CNT++ + extra
```

Finalizer function:

```
def _lambda_thismodule_7_8_1`finalizer ( var __this:_lambda_thismodule_7_8_1? explicit␣
↪)
    delete *this
    delete __this
```

Lambda creation is replaced with the ascend of the capture structure:

```
let counter:lambda<(extra:int const):int> const <- new<lambda<(extra:int const):int>>␣
↪[[CNT = CNT]]
```

The C++ Lambda class contains single void pointer for the capture data:

```
struct Lambda {
    ...
    char *      capture;
    ...
};
```

The rational behind passing lambda by reference is that when delete is called

1. the finalizer is invoked for the capture data
2. the capture is replaced via null

The lack of a copy or move ensures there are not multiple pointers to a single instance of the captured data floating around.

2.16 Struct

Daslang uses a structure mechanism similar to languages like C/C++, Java, C#, etc. However, there are some important difference. Structures are first class objects like integers or strings and can be stored in table slots, other structures, local variables, arrays, tuples, variants, etc., and passed as function parameters.

2.16.1 Struct Declaration

A structure object is created through the keyword `struct`:

```
struct Foo
    x, y: int
    xf: float
```

Sturctures can be `private` or `public`:

```
struct private Foo
    x, y: int

struct public Bar
    xf: float
```

If not specified, structures inherit module publicity (i.e. in public modules structures are public, and in private modules structures are private).

Structure instances are created through a 'new expression' or a variable declaration statement:

```
let foo: Foo
let foo: Foo? = new Foo
```

There are intentionally no member functions. There are only data members, since it is a data type itself. Structures can handle members with a function type as data (meaning it's a function pointer that can be changed during execution). There are initializers that simplify writing complex structure initialization. Basically, a function with same name as the structure itself works as an initializer. The compiler will generate a 'default' initializer if there are any members with an initializer:

```
struct Foo
    x: int = 1
    y: int = 2
```

Structure fields are initialized to zero by default, regardless of 'initializers' for members, unless you specifically call the initializer:

```
let fZero : Foo      // no initializer is called, x, y = 0
let fInited = Foo() // initializer is called, x = 1, y = 2
```

Structure field types are inferred, where possible:

```
struct Foo
    x = 1    // inferred as int
    y = 2.0    // inferred as float
```

Explicit structure initialization during creation leaves all uninitialized members zeroed:

```
let fExplicit = [[Foo x=13]]  // x = 13, y = 0
```

The previous code example is syntactic sugar for:

```
let fExplicit: Foo
fExplicit.x = 13
```

Post-construction initialization only needs to specify overwritten fields:

```
let fPostConstruction = [[Foo() x=13]]  // x = 13, y = 2
```

The previous code example is syntactic sugar for:

```
let fPostConstruction: Foo
fPostConstruction.x = 13
fPostConstruction.y = 2
```

The "Clone initializer" is useful pattern for creating a clone of an existing structure when both structures are on the heap:

```
def Foo ( p : Foo? )                // "clone initializer" takes pointer to existing␣
↪structure
    var self := *p
    return <- self
...
let a = new [[Foo x=1, y=2.]]       // create new instance of Foo on the heap,␣
↪initialize it
let b = new Foo(a)                  // clone of b is created here
```

2.16.2 Structure Function Members

Daslang doesn't have embedded structure member functions, virtual (that can be overridden in inherited structures) or non-virtual. Those features are implemented for classes. For ease of Objected Oriented Programming, non-virtual member functions can be easily emulated with the pipe operator |>:

```
struct Foo
    x, y: int = 0

def setXY(var thisFoo: Foo; X, Y: int)
    with thisFoo
        x = X
        y = Y

var foo: Foo
foo |> setXY(10, 11)   // this is syntactic sugar for setXY(foo, 10, 11)
setXY(foo, 10, 11)     // exactly same thing as the line above
```

Since function pointers are a thing, one can emulate 'virtual' functions by storing function pointers as members:

```
struct Foo
    x, y: int = 0
    set = @@setXY

def setXY(var thisFoo: Foo; X, Y: int)
    with thisFoo
        x = X
        y = Y
...
var foo: Foo = Foo()
foo->set(1, 2)  // this one can call something else, if overridden in derived class.
                // It is also just syntactic sugar for function pointer call
invoke(foo.set, foo, 1, 2)  // exactly same thing as above
```

This makes the difference between virtual and non-virtual calls in the OOP paradigm explicit. In fact, Daslang classes implement virtual functions in exactly this manner.

2.16.3 Inheritance

Daslang's structures support single inheritance by adding a ' : ', followed by the parent structure's name in the structure declaration. The syntax for a derived struct is the following:

```
struct Bar: Foo
    yf: float
```

When a derived structure is declared, Daslang first copies all base's members to the new structure and then proceeds with evaluating the rest of the declaration.

A derived structure has all members of its base structure. It is just syntactic sugar for copying all the members manually first.

2.16.4 Alignment

Structure size and alignment are similar to that of C++:

- individual members are aligned individually
- overall structure alignment is that of the largest member's alignment

Inherited structure alignment can be controlled via the [cpp_layout] annotation:

```
[cpp_layout (pod=false)]
struct CppS1
    vtable : void?              // we are simulating C++ class
    b : int64 = 21
    c : int = 3

[cpp_layout (pod=false)]
struct CppS2 : CppS1            // d will be aligned on the class bounds
    d : int = 4
```

2.16.5 OOP

There is sufficient amount of infrastructure to support basic OOP on top of the structures. However, it is already available in form of classes with some fixed memory overhead (see *Classes*).

It's possible to override the method of the base class with override syntax. Here an example:

```
struct Foo
    x, y: int = 0
    set = @@Foo_setXY

def Foo_setXY(var this: Foo; x, y: int)
    this.x = x
    this.y = y

struct Foo3D: Foo
    z: int = 3
    override set = cast<auto> @@Foo3D_setXY

def Foo3D_setXY(var thisFoo: Foo3D; x, y: int)
    thisFoo.x = x
    thisFoo.y = y
    thisFoo.z = -1
```

It is safe to use the `cast` keyword to cast a derived structure instance into its parent type:

```
var f3d: Foo3D = Foo3D()
(cast<Foo> f3d).y = 5
```

It is unsafe to cast a base struct to it's derived child type:

```
var f3d: Foo3D = Foo3D()
def foo(var foo: Foo)
    (cast<Foo3D> foo).z = 5  // error, won't compile
```

If needed, the upcast can be used with the `unsafe` keyword:

```
struct Foo
    x: int

struct Foo2:Foo
    y: int

def setY(var foo: Foo; y: int)  // Warning! Can make awful things to your app if its␣
↪not really Foo2
    unsafe
        (upcast<Foo2> foo).y = y
```

As the example above is very dangerous, and in order to make it safer, you can modify it to following:

```
struct Foo
    x: int
    typeTag: uint = hash("Foo")

struct Foo2:Foo
    y: int
    override typeTag: uint = hash("Foo2")

def setY(var foo: Foo; y: int)  // this won't do anything really bad, but will panic␣
↪on wrong reference
    unsafe
        if foo.typeTag == hash("Foo2")
            (upcast<Foo2> foo).y = y
            print("Foo2 type references was passed\n")
        else
            assert(false, "Not Foo2 type references was passed\n")
```

2.17 Tuple

Tuples are a concise syntax to create nameless data structures:

```
tuple ::= tuple < element_list >
element_list ::= nameless_element_list | named_element_list
nameless_element_list ::= type | nameless_element_list ';' type
named_element_list := name : type | named_element_list ';' name : type
```

Two tuple declarations are the same if they have the same number of types, and their respective types are the same:

```
var a : tuple<int; float>
var b : tuple<i:int; f:float>
a = b
```

Tuple elements can be accessed via nameless fields, i.e. _ followed by the 0 base field index:

```
a._0 = 1
a._1 = 2.0
```

Named tuple elements can be accessed by name as well as via nameless field:

```
b.i = 1          // same as _0
b.f = 2.0        // same as _1
b._1 = 2.0       // _1 is also available
```

Tuples follow the same alignment rules as structures (see *Structures*).

Tuple alias types can be constructed the same way as structures. For example:

```
tuple Foo
    a : int
    b : float
```

It's the same as:

```
typedef Foo = tuple<a:int;b:float>
```

Tuples can be constructed using the tuple constructor, for example:

```
var a = [[auto 1,2.0,"3"]]
var b = [[tuple<int;float;string> 1, 2.0, "3"]]
```

Alternative syntax is:

```
var a = tuple(1,2.0,"3")
var b = tuple<int;float;string>(1, 2.0, "3")
```

both auto a full type specification can be used to construct a tuple. Array of tuples can be constructed using similar syntax, with a ; as a separator:

```
var a = [[auto 1, 2.0, "3"; 4, 5.0, "6"]]
```

There is a shortcut syntax for constructing tuples, where the tuple is returned:

```
return 1, 2.0, "3"  // same as return [[auto 1, 2.0, "3"]]
```

Tuples can be expanded upon the variable declaration, for example:

```
var [[a, b, c]] = [[auto 1, 2.0, "3"]]
```

In this case only one variable is created, as well as for 'assume' expressions. I.e:

```
var a`b`c = [[auto 1, 2.0, "3"]]
assume a  = a`b`c._0
assume b  = a`b`c._1
assume c  = a`b`c._2
```

2.18 Variant

Variants are nameless types which provide support for values that can be one of a number of named cases, possibly each with different values and types:

```
var t : variant<i_value:uint;f_value:float>
```

There is a shorthand type alias syntax to define a variant:

```
variant U_F
    i_value : uint
    f_value : float

```

(continues on next page)

(continued from previous page)

```
typedef
    U_F = variant<i_value:uint;f_value:float> // exactly the same as the declaration␣
↪above
```

Any two variants are the same type if they have the same named cases of the same types in the same order.

Variants hold the `index` of the current case, as well as the value for the current case only.

The current case selection can be checked via the `is` operator, and accessed via the `as` operator:

```
assert(t is i_value)
assert(t as i_value == 0x3f800000)
```

The entire variant selection can be modified by copying the properly constructed variant of a different case:

```
t = [[U_F i_value = 0x40000000]]     // now case is i_value
t = [[U_F f_value = 1.0]]            // now case is f_value
```

Accessing a variant case of the incorrect type will cause a panic:

```
t = [[U_F i_value = 0x40000000]]
return t as f_value                  // panic, invalid variant index
```

Safe navigation is available via the `?as` operation:

```
return t ?as f_value ?? 1.0          // will return 1.0 if t is not f_value
```

Cases can also be accessed in an unsafe manner without checking the type:

```
unsafe
    t.i_value = 0x3f800000
    return t.f_value                    // will return memory, occupied by f_value -␣
↪i.e. 1.0f
```

The current index can be determined via the `variant_index` function:

```
var t : U_F
assert(variant_index(t)==0)
```

The index value for a specific case can be determine via the `variant_index` and `safe_variant_index` type traits. `safe_variant_index` will return -1 for invalid indices and types, whereas `variant_index` will report a compilation error:

```
assert(typeinfo(variant_index<i_value> t)==0)
assert(typeinfo(variant_index<f_value> t)==1)
assert(typeinfo(variant_index<unknown_value> t)==-1) // compilation error

assert(typeinfo(safe_variant_index<i_value> t)==0)
assert(typeinfo(safe_variant_index<f_value> t)==1)
assert(typeinfo(safe_variant_index<unknown_value> t)==-1)
```

Current case selection can be modified with the unsafe operation `safe_variant_index`:

```
unsafe
    set_variant_index(t, typeinfo(variant_index<f_value> t))
```

2.18.1 Alignment and data layout

Variants contain the 'index' of the current case, followed by a union of individual cases, similar to the following C++ layout:

```
struct MyVariantName {
    int32_t __variant_index;
    union {
        type0   case0;
        type1   case1;
        ...
    };
};
```

Individual cases start from the same offset.

The variant type is aligned by the alignment of its largest case, but no less than that of an int32.

2.19 Class

In Daslang, classes are an extension of structures designed to provide OOP capabilities. Classes provides single parent inheritance, abstract and virtual methods, initializers, and finalizers.

The basic class declaration is similar to that of a structure, but with the `class` keyword:

```
class Foo
    x, y : int = 0
    def Foo                                 // custom initializer
        Foo`set(self,1,1)
    def set(X,Y:int)                        // inline method
        x = X
        y = Y
```

The initializer is a function with a name matching that of a class. Classes can have multiple initializer with different arguments:

```
class Foo
    ...
    def Foo(T:int)                          // custom initializer
        self->set(T,T)
    def Foo(X,Y:int)                        // custom initializer
        Foo`set(self,X,Y)
```

Finalizers can be defined explicitly as void functions named `finalize`:

```
class Foo
    ...
    def finalize                            // custom finalizer
        delFoo ++
```

Alternative syntax is:

```
class Foo
    ...
    def operator delete                     // custom finalizer
        delFoo ++
```

There are no guarantees that a finalizer is called implicitly (see *Finalizers*).

Derived classes need to override methods explicitly, using the `override` keyword:

```
class Foo3D : Foo
    z : int = 13
    def Foo3D                           // overriding default initializer
        Foo`Foo(self)                   // call parents initializer explicitly
        z = 3
    def override set(X,Y:int)           // overriding method variable
        Foo`set(self,X,Y)               // calling generated method function directly
        z = 0
```

Classes can define abstract methods using the `abstract` keyword:

```
class FooAbstract
    def abstract set(X,Y:int) : void              // inline method
```

Abstract functions need to be fully qualified, including their return type. Class member functions are inferred in the same manner as regular functions.

Sealed functions cannot be overridden. The `sealed` keyword is used to prevent overriding:

```
class Foo3D : Foo
    def sealed set(X,Y:int )     // subclasses of Foo3D can no longer override this␣
↪method
        xyz = X + Y
```

Sealed classes can not be inherited from. The `sealed` keyword is used to prevent inheritance:

```
class sealed Foo3D : Foo         // Foo3D can no longer be inherited from
    ...
```

A pointer named `self` is available inside any class method.

Classes can be created via the `new` operator:

```
var f = new Foo()
```

Local class variables are unsafe:

```
unsafe
    var f = Foo()        // unsafe
```

Class methods can be invoked using `->` syntax:

```
f->set(1,2)
```

A specific version of the method can also be called explicitly:

```
Foo`set(*f,1,2)
```

Class methods can be constant:

```
class Foo
    dir : float3
    def const length
        return length(dir)  // dir is const float3 here
```

Class methods can be operators:

```
class Foo
    dir : float3
    def Foo ( x,y,z:float )
        dir = float3(x,y,z)
    def Foo ( d:float3 )
        dir = d
    def const operator . length
        return length(dir)
    def operator . length := ( value:float )
        dir = normalize(dir) * value
    def const operator + ( other:Foo )
        return Foo(dir + other.dir)
```

Class fields can be declared static, i.e. shared between all instances of the class:

```
class Foo
    static count : int = 0
    def Foo
        count ++
    def finalize
        count --
```

Class methods can be declared static. Static methods don't have access to 'self' but can access static fields:

```
class Foo
    static count : int = 0
    def static getCount : int
        return count

    let count = Foo`getCount()  // they can be accessed outside of class
```

2.19.1 Implementation details

Class initializers are generated by adding a local `self` variable with *construct* syntax. The body of the method is prefixed via a `with self` expression. The final expression is a `return <- self`:

```
def Foo ( X:int const; Y:int const ) : Foo
    var self:Foo <- [[Foo()]]
    with self
        Foo`Foo(self,X,Y)
    return <- self
```

Class methods and finalizers are generated by providing the extra argument `self`. The body of the method is prefixed with a `with self` expression:

```
def Foo3D`set ( var self:Foo3D; X:int const; Y:int const )
    with self
        Foo`set(self,X,Y)
        z = 0
```

Calling virtual methods is implemented via invoke:

```
invoke(f3d.set,cast<Foo> f3d,1,2)
```

Every base class gets an `__rtti` pointer, and a `__finalize` function pointer. Additionally, a function pointer is added for each member function:

```
class Foo
        __rtti : void? = typeinfo(rtti_classinfo type<Foo>)
        __finalize : function<(self:Foo):void> = @@_::Foo'__finalize
        x : int = 0
        y : int = 0
        set : function<(self:Foo;X:int const;Y:int const):void> = @@_::Foo`set
```

`__rtti` contains rtti::TypeInfo for the specific class instance. There is helper function in the rtti module to access class_info safely:

```
def class_info ( cl ) : StructInfo const?
```

The `finalize` pointer is invoked when the finalizer is called for the class pointer. That way, when delete is called on the base class pointer, the correct version of the derived finalizer is called.

2.20 Constants, Enumerations, Global variables

Daslang allows you to bind constant values to a global variable identifier. Whenever possible, all constant global variables will be evaluated at compile time. There are also enumerations, which are strongly typed constant collections similar to enum classes in C++.

2.20.1 Constant

Constants bind a specific value to an identifier. Constants are exactly global variables. Their value cannot be changed.

Constants are declared with the following syntax:

```
let
  foobar = 100
let
  floatbar = 1.2
let
  stringbar = "I'm a constant string"
let blah = "I'm string constant which is declared on the same line as variable"
```

Constants are always globally scoped from the moment they are declared. Any subsequential code can reference them.

You can not change such global variables.

Constants can be `shared`:

```
let shared blah <- [{string "blah"; "blahh"; "blahh"}]
```

Shared constants point to the same memory in different instances of *Context*. They are initialized once during the first context initialization.

2.20.2 Global variable

Mutable global variables are defined as:

```
var
  foobar = 100
var barfoo = 100
```

Their usage can be switched on and off on a per-project basis via *CodeOfPolicies*.

Local static variables can be declared via the *static_let* macro:

```
require daslib/static_let

def foo
    static_let <|
        var bar = 13
    bar = 14
```

Variable `bar` in the example above is effectively a global variable. However, it's only visible inside the scope of the corresponding *static_let* macro.

Global variables can be *private* or *public*

```
let public foobar = 100

let private barfoo = 100
```

If not specified, structures inherit module publicity (i.e. in public modules global variables are public, and in private modules global variables are private).

2.20.3 Enumeration

An enumeration binds a specific value to a name. Enumerations are also evaluated at compile time and their value cannot be changed.

An enum declaration introduces a new enumeration to the program. Enumeration values can only be compile time constant expressions. It is not required to assign specific value to enum:

```
enum Numbers
    zero     // will be 0
    one      // will be 1
    two      // will be 2
    ten = 9+1 // will be 10, since its explicitly specified
```

Enumerations can be *private* or *public*:

```
enum private Foo
    fooA
    fooB

enum public Bar
    barA
    barB
```

If not specified, enumeration inherit module publicity (i.e. in public modules enumerations are public, and in private modules enumerations are private).

An enum name itself is a strong type, and all enum values are of this type. An enum value can be addressed as 'enum name' followed by exact enumeration

```
let one: Numbers = Numbers one
```

An enum value can be converted to an integer type with an explicit cast

```
let one: Numbers = Numbers one
assert(int(one) == 1)
```

Enumerations can specify one of the following storage types: `int`, `int8`, `int16`, `uint`, `uint8`, or `uint16`:

```
enum Characters : uint8
    ch_a = 'A'
    ch_b = 'B'
```

Enumeration values will be truncated down to the storage type.

The *each_enum* iterator iterates over specific enumeration type values. Any enum element needs to be provided to specify the enumeration type:

```
for x in each_enum(Characters ch_a)
        print("x = {x}\n")
```

2.21 Bitfield

Bitfields are a nameless types which represent a collection of up to 32-bit flags in a single integer:

```
var t : bitfield < one; two; three >
```

There is a shorthand type alias syntax to define a bitfield:

```
bitfield bits123
    one
    two
    three

typedef
    bits123 = bitfield<one; two; three> // exactly the same as the declaration above
```

Any two bitfields are the same type and represent 32-bit integer:

```
var a : bitfield<one; two; three>
var b : bitfield<one; two>
b = a
```

Individual flags can be read as if they were regular bool fields:

```
var t : bitfield < one; two; three >
assert(!t.one)
```

If alias is available, bitfield can be constructed via alias notation:

```
assert(t==bits123 three)
```

Bitfields can be constructed via an integer value. Limited binary logical operators are available:

```
var t : bitfield < one; two; three > = bitfield(1<<1) | bitfield(1<<2)
assert(!t.one && t.two && t.three)
assert("{t}"=="(two|three)")
t ^= bitfield(1<<1)
```

2.22 Comprehension

Comprehensions are concise notation constructs designed to allow sequences to be built with other sequences.

The syntax is inspired by that of a for loop:

```
comprehension ::= array_comprehension | iterator_comprehension
array_comprehension ::= [{ any_comprehension }]
iterator_comprehension := [[ any_comprehension ]]
any_comprehension ::= for argument_list in source_list; result { ; where optional_
↪clause }
argument_list ::= argument | argument_list ',' argument
source_list ::= iterable_expression | source_list ',' iterable_expression
```

Comprehension produces either an iterator or a dynamic array, depending on the style of brackets:

```
var a1 <- [[for x in range(0,10); x]]    // iterator<int>
var a2 <- [{for x in range(0,10); x}]    // array<int>
```

A `where` clause acts as a filter:

```
var a3 <- [{for x in range(0,10); x; where (x & 1) == 1}]   // only odd numbers
```

Just like a for loop, comprehension can iterate over multiple sources:

```
var a4 <- [{for x,y in range(0,10),a1; x + y; where x==y }] // multiple variables
```

Iterator comprehension may produce a referenced iterator:

```
var a = [[int 1;2;3;4]]
var b <- [[for x in a; a]]  // iterator<int&> and will point to captured copy of the␣
↪elements of a
```

Regular lambda capturing rules are applied for iterator comprehensions (see *Lambdas*).

Internally array comprehension produces an `invoke` of a local block and a for loop; whereas iterator comprehension produces a generator (lambda). Array comprehensions are typically faster, but iterator comprehensions have less of a memory footprint.

2.23 Iterator

Iterators are objects which can traverse over a sequence without knowing the details of the sequence's implementation.

The Iterator type is defined as follows:

```
iterator_type ::= iterator < type >

iterator<int>            // iterates over integer
iterator<const Foo&>     // iterates over Foo by reference
```

Iterators can be moved, but not copied or cloned.

Iterators can be created via the `each` function from a range, static array, or dynamic array. `each` functions are unsafe because the iterator does not capture its arguments:

```
unsafe
    var it <- each ( [[int 1;2;3;4]] )
```

The most straightforward way to traverse an iterator is with a `for` loop:

```
for x in it             // iterates over contents of 'it'
    print("x = {x}\n")
```

For the reference iterator, the `for` loop will provide a reference variable:

```
var t = [[int 1;2;3;4]]
for x in t      // x is int&
    x ++        // increases values inside t
```

Iterators can be created from lambdas (see *Lambda*) or generators (see *Generator*).

Calling `delete` on an iterator will make it sequence out and free its memory:

```
var it <- each_enum(Numbers one)
delete it                          // safe to delete

var it <- each_enum(Numbers one)
for x in it
    print("x = {x}\n")
delete it                          // its always safe to delete sequenced out␣
↪iterator
```

Loops and iteration functions do it automatically.

2.23.1 builtin iterators

Table keys and values iterators can be obtained via the `keys` and `values` functions:

```
var tab <- {{ "one"=>1; "two"=>2 }}
for k,v in keys(tab),values(tab)        // keys(tab) is iterator<string>
    print("{k} => {v}\n")               // values(tab) is iterator<int&>
```

It is possible to iterate over each character of the string via the `each` function:

```
unsafe
    for ch in each("hello,world!")      // string iterator is iterator<int>
        print("ch = {ch}\n")
```

It is possible to iterate over each element of an enumeration via the `each_enum` function:

```
enum Numbers
    one
    two
    ten = 10

for x in each_enum(Numbers one)         // argument is any value from said enumeration
    print("x = {x}\n")
```

2.23.2 builtin iteration functions

The `empty` function checks if an iterator is null or already sequenced out:

```
unsafe
    var it <- each ( [[int 1;2;3;4]] )
    for x in it
        print("x = {x}\n")
    verify(empty(it))            // iterator is sequenced out
```

More complicated iteration patterns may require the `next` function:

```
var x : int
while next(it,x)        // this is semantically equivalent to the `for x in it`
    print("x = {x}\n")
```

Next can only operate on copyable types.

2.23.3 low level builtin iteration functions

`_builtin_iterator_first`, `_builtin_iterator_next`, and `_builtin_iterator_close` address the regular lifecycle of the iterator. A semantic equivalent of the for loop can be explicitly written using these operations:

```
var it <- each(range(0,10))
var i : int
var pi : void?
unsafe
    pi = reinterpret<void?> ( addr(i) )
if _builtin_iterator_first(it,pi)
    print("i = {i}\n")
    while _builtin_iterator_next(it,pi)
        print("i = {i}\n")
    _builtin_iterator_close(it,pi)
```

`_builtin_iterator_iterate` is one function to rule them all. It acts like all 3 functions above. On a non-empty iterator it starts with 'first', then proceeds to call *next* until the sequence is exhausted. Once the iterator is sequenced out, it calls *close*:

```
var it <- each(range(0,10))
var i : int
var pi : void?
unsafe
    pi = reinterpret<void?> ( addr(i) )
while _builtin_iterator_iterate(it,pi)      // this is equivalent to the example above
    print("i = {i}\n")
```

2.23.4 next implementation details

The function `next` is implemented as follows:

```
def next ( it:iterator<auto(TT)>; var value : TT& ) : bool
    static_if !typeinfo(can_copy type<TT>)
        concept_assert(false, "requires type which can be copied")
    static_elif typeinfo(is_ref_value type<TT>)
        var pValue : TT - & ?
        unsafe
            if _builtin_iterator_iterate(it, addr(pValue))
                value = *pValue
                return true
            else
                return false
    else
        unsafe
            return _builtin_iterator_iterate(it, addr(value))
```

It is important to notice that builtin iteration functions accept pointers instead of references.

2.24 Generator

Generators allow you to declare a lambda that behaves like an iterator. For all intents and purposes, a generator is a lambda passed to an `each` or `each_ref` function.

Generator syntax is similar to lambda syntax:

```
generator ::= `generator` < type > $ ( ) block
```

Generator lambdas must have no arguments. It always returns boolean:

```
let gen <- generator<int>() <| $()  // gen is iterator<int>
    for t in range(0,10)
        yield t
    return false                    // returning false stops iteration
```

The result type of a `generator` expression is an iterator (see *Iterators*).

Generators output iterator values via `yield` expressions. Similar to the return statement, move semantic `yield <-` is allowed:

```
return <- generator<TT> () <| $ ()
    for w in src
        yield <- invoke(blk,w)  // move invoke result
    return false
```

Generators can output ref types. They can have a capture section:

```
unsafe                                                  // unsafe due to capture of␣
↪src by reference
    var src = [[int 1;2;3;4]]
    var gen <- generator<int&> [[&src]] () <| $ ()      // capturing src by ref
        for w in src
            yield w                                     // yield of int&
        return false
```

(continues on next page)

(continued from previous page)

```
    for t in gen
        t ++
    print("src = {src}\n")  // will output [[2;3;4;5]]
```

Generators can have loops and other control structures:

```
let gen <- generator<int>() <| $()
    var t = 0
    while t < 100
        if t == 10
            break
        yield t ++
    return false

let gen <- generator<int>() <| $()
    for t in range(0,100)
        if t >= 10
            continue
        yield t
    return false
```

Generators can have a `finally` expression on its blocks, with the exception of the if-then-else blocks:

```
let gen <- generator<int>() <| $()
    for t in range(0,9)
        yield t
    finally
        yield 9
    return false
```

2.24.1 implementation details

In the following example:

```
var gen <- generator<int> () <| $ ()
    for x in range(0,10)
        if (x & 1)==0
            yield x
    return false
```

A lambda is generated with all captured variables:

```
struct _lambda_thismodule_8_8_1
    __lambda : function<(__this:_lambda_thismodule_8_8_1;_yield_8:int&):bool const> =␣
↪@@_::_lambda_thismodule_8_8_1`function
    __finalize : function<(__this:_lambda_thismodule_8_8_1? -const):void> = @@_::_
↪lambda_thismodule_8_8_1`finalizer
    __yield : int
    _loop_at_8 : bool
    x : int // captured constant
    _pvar_0_at_8 : void?
    _source_0_at_8 : iterator<int>
```

A lambda function is generated:

```
[GENERATOR]
def _lambda_thismodule_8_8_1`function ( var __this:_lambda_thismodule_8_8_1; var _
↪yield_8:int& ) : bool const
    goto __this.__yield
    label 0:
    __this._loop_at_8 = true
    __this._source_0_at_8 <- __::builtin`each(range(0,10))
    memzero(__this.x)
    __this._pvar_0_at_8 = reinterpret<void?> addr(__this.x)
    __this._loop_at_8 &&= _builtin_iterator_first(__this._source_0_at_8,__this._pvar_
↪0_at_8,__context__)
    label 3: /*begin for at line 8*/
    if !__this._loop_at_8
            goto label 5
    if !((__this.x & 1) == 0)
            goto label 2
    _yield_8 = __this.x
    __this.__yield = 1
    return /*yield*/ true
    label 1: /*yield at line 10*/
    label 2: /*end if at line 9*/
    label 4: /*continue for at line 8*/
    __this._loop_at_8 &&= _builtin_iterator_next(__this._source_0_at_8,__this._pvar_0_
↪at_8,__context__)
    goto label 3
    label 5: /*end for at line 8*/
    _builtin_iterator_close(__this._source_0_at_8,__this._pvar_0_at_8,__context__)
    return false
```

Control flow statements are replaced with the `label` + `goto` equivalents. Generators always start with `goto __this.yield`. This effectively produces a finite state machine, with the `yield` variable holding current state index.

The `yield` expression is converted into a copy result and return value pair. A label is created to specify where to go to next time, after the `yield`:

```
_yield_8 = __this.x                    // produce next iterator value
__this.__yield = 1                     // label to go to next (1)
return /*yield*/ true                  // return true to indicate, that iterator␣
↪produced a value
label 1: /*yield at line 10*/          // next label marker (1)
```

Iterator initialization is replaced with the creation of the lambda:

```
var gen:iterator<int> <- each(new<lambda<(_yield_8:int&):bool const>> [[_lambda_
↪thismodule_8_8_1]])
```

2.25 Finalizer

Finalizers are special functions which are called in exactly two cases:

`delete` is called explicitly on a data type:

```
var f <- [{int 1;2;3;4}]
delete f
```

Lambda based iterator or generator is sequenced out:

```
var src <- [{int 1;2;3;4}]
var gen <- generator<int&> [[<-src]] () <| $ ()
    for w in src
        yield w
    return false
for t in gen
    print("t = {t}\n")
// finalizer is called on captured version of src
```

By default finalizers are called recursively on subtypes.

If memory models allows deallocation, standard finalizers also free the memory:

```
options persistent_heap = true

var src <- [{int 1;2;3;4}]
delete src                      // memory of src will be freed here
```

Custom finalizers can be defined for any type by overriding the `finalize` function. Generic custom finalizers are also allowed:

```
struct Foo
    a : int

def finalize ( var foo : Foo )
    print("we kill foo\n")

var f = [[Foo a = 5]]
delete f                    // prints 'we kill foo' here
```

2.25.1 Rules and implementation details

Finalizers obey the following rules:

If a custom `finalize` is available, it is called instead of default one.

Pointer finalizers expand to calling `finalize` on the dereferenced pointer, and then calling the native memory finalizer on the result:

```
var pf = new Foo
unsafe
    delete pf
```

This expands to:

```
def finalize ( var __this:Foo?& explicit -const )
    if __this != null
        _::finalize(deref(__this))
        delete /*native*/ __this
        __this = null
```

Static arrays call `finalize_dim` generically, which finalizes all its values:

```
var f : Foo[5]
delete f
```

This expands to:

```
def builtin`finalize_dim ( var a:Foo aka TT[5] explicit )
    for aV in a
        _::finalize(aV)
```

Dynamic arrays call `finalize` generically, which finalizes all its values:

```
var f : array<Foo>
delete f
```

This expands to:

```
def builtin`finalize ( var a:array<Foo aka TT> explicit )
    for aV in a
        _::finalize(aV)
    __builtin_array_free(a,4,__context__)
```

Tables call `finalize` generically, which finalizes all its values, but not its keys:

```
var f : table<string;Foo>
delete f
```

This expands to:

```
def builtin`finalize ( var a:table<string aka TK;Foo aka TV> explicit )
    for aV in values(a)
        _::finalize(aV)
    __builtin_table_free(a,8,4,__context__)
```

Custom finalizers are generated for structures. Fields annotated as [[do_not_delete]] are ignored. `memzero` clears structure memory at the end:

```
struct Goo
    a : Foo
    [[do_not_delete]] b : array<int>

var g <- [[Goo]]
delete g
```

This expands to:

```
def finalize ( var __this:Goo explicit )
    _::finalize(__this.a)
    __::builtin`finalize(__this.b)
    memzero(__this)
```

Tuples behave similar to structures. There is no way to ignore individual fields:

```
var t : tuple<Foo; int>
delete t
```

This expands to:

```
def finalize ( var __this:tuple<Foo;int> explicit -const )
    _::finalize(__this._0)
    memzero(__this)
```

Variants behave similarly to tuples. Only the currently active variant is finalized:

```
var t : variant<f:Foo; i:int; ai:array<int>>
delete t
```

This expands to:

```
def finalize ( var __this:variant<f:Foo;i:int;ai:array<int>> explicit -const )
    if __this is f
        _::finalize(__this.f)
    else if __this is ai
        __::builtin`finalize(__this.ai)
    memzero(__this)
```

Lambdas and generators have their capture structure finalized. Lambdas can have custom finalizers defined as well (see *Lambdas*).

Classes can define custom finalizers inside the class body (see *Classes*).

2.26 String Builder

Instead of formatting strings with variant arguments count function (like printf), Daslang provides String builder functionality out-of-the-box. It is both more readable, more compact and more robust than printf-like syntax. All strings in Daslang can be either string literals, or *built strings*. Both are written with "", but string builder strings also contain any expression in curly brackets '{}':

```
let str1 = "String Literal"
let str2 = "str1={str1}"  // str2 will be "str1=String Literal"
```

In the example above, str2 will actually be compile-time defined, as the expression in {} is compile-time computable. But generally, they can be run-time compiled as well. Expressions in {} can be of any type, including handled extern type, provided that said type implements `DataWalker`. All PODs in Daslang do have `DataWalker` 'to string' implementation.

In order to make a string with {} inside, one has to escape curly brackets with '':

```
print("Curly brackets=\{\}")  // prints Curly brackets={}
```

2.27 Generic Programming

Daslang allows omission of types in statements, functions, and function declaration, making writing in it similar to dynamically typed languages, such as Python or Lua. Said functions are *instantiated* for specific types of arguments on the first call.

There are also ways to inspect the types of the provided arguments, in order to change the behavior of function, or to provide reasonable meaningful errors during the compilation phase. Most of these ways are achieved with s

Unlike C++ with its SFINAE, you can use common conditionals (if) in order to change the instance of the function depending on the type info of its arguments. Consider the following example:

```
def setSomeField(var obj; val)
    if typeinfo(has_field<someField> obj)
        obj.someField = val
```

This function sets `someField` in the provided argument *if* it is a struct with a `someField` member.

We can do even more. For example:

```
def setSomeField(var obj; val: auto(valT))
    if typeinfo(has_field<someField> obj)
        if typeinfo(typename obj.someField) == typeinfo(typename type valT -const)
            obj.someField = val
```

This function sets `someField` in the provided argument *if* it is a struct with a `someField` member, and only if `someField` is of the same type as `val`!

2.27.1 typeinfo

Most type reflection mechanisms are implemented with the typeinfo operator. There are:

- `typeinfo(typename object)` // returns typename of object
- `typeinfo(fulltypename object)` // returns full typename of object, with contracts (like !const, or !&)
- `typeinfo(sizeof object)` // returns sizeof
- `typeinfo(is_pod object)` // returns true if object is POD type
- `typeinfo(is_raw object)` // returns true if object is raw data, i.e., can be copied with memcpy
- `typeinfo(is_struct object)` // returns true if object is struct
- `typeinfo(has_field<name_of_field> object)` // returns true if object is struct with field name_of_field
- `typeinfo(is_ref object)` // returns true if object is reference to something
- `typeinfo(is_ref_type object)` // returns true if object is of reference type (such as array, table, das_string or other handled reference types)
- `typeinfo(is_const object)` // returns true if object is of const type (i.e., can't be modified)
- `typeinfo(is_pointer object)` // returns true if object is of pointer type, i.e., int?

All typeinfo can work with types, not objects, with the `type` keyword:

```
typeinfo(typename type int) // returns "int"
```

2.27.2 auto and auto(named)

Instead of ommitting the type name in a generic, it is possible to use an explicit `auto` type or `auto(name)` to type it:

```
def fn(a: auto): auto
    return a
```

or

```
def fn(a: auto(some_name)): some_name
    return a
```

This is the same as:

```
def fn(a)
    return a
```

This is very helpful if the function accepts numerous arguments, and some of them have to be of the same type:

```
def fn(a, b) // a and b can be of different types
    return a + b
```

This is not the same as:

```
def fn(a, b: auto) // a and b are one type
    return a + b
```

Also, consider the following:

```
def set0(a, b; index: int) // a is only supposed to be of array type, of same type as␣
↪b
    return a[index] = b
```

If you call this function with an array of floats and an int, you would get a not-so-obvious compiler error message:

```
def set0(a: array<auto(some)>; b: some; index: int) // a is of array type, of same␣
↪type as b
    return a[index] = b
```

Usage of named `auto` with `typeinfo`

```
def fn(a: auto(some))
    print(typeinfo(typename type some))

fn(1) // print "const int"
```

You can also modify the type with delete syntax:

```
def fn(a: auto(some))
    print(typeinfo(typename type some -const))

fn(1) // print "int"
```

2.27.3 type contracts and type operations

Generic function arguments, result, and inferred type aliases can be operated on during the inference.

const specifies, that constant and regular expressions will be matched:

```
def foo ( a : Foo const )   // accepts Foo and Foo const
```

==*const* specifies, that const of the expression has to match const of the argument:

```
def foo ( a : Foo const ==const )   // accepts Foo const only
def foo ( var a : Foo ==const )     // accepts Foo only
```

-*const* will remove const from the matching type:

```
def foo ( a : array<auto -const> )  // matches any array, with non-const elements
```

specifies that only temporary types are accepted:

```
def foo ( a : Foo# )     // accepts Foo# only
```

-*#* will remove temporary type from the matching type:

```
def foo ( a : auto(TT) )         // accepts any type
    var temp : TT -# := a;       // TT -# is now a regular type, and when `a` is␣
↪temporary, it can clone it into `temp`
```

& specifies that argument is passed by reference:

```
def foo ( a : auto& )            // accepts any type, passed by reference
```

==*&* specifies that reference of the expression has to match reference of the argument:

```
def foo ( a : auto& ==& )   // accepts any type, passed by reference (for example␣
↪variable i, even if its integer)
def foo ( a : auto ==& )    // accepts any type, passed by value     (for example␣
↪value 3)
```

-*&* will remove reference from the matching type:

```
def foo ( a : auto(TT)& )        // accepts any type, passed by reference
    var temp : TT -& = a;        // TT -& is not a local reference
```

[] specifies that the argument is a static array of arbitrary dimension:

```
def foo ( a : auto[] )           // accepts static array of any type of any size
```

-*[]* will remove static array dimension from the matching type:

```
def take_dim( a : auto(TT) )
    var temp : TT -[]            // temp is type of element of a
// if a is int[10] temp is int
// if a is int[10][20][30] temp is still int
```

implicit specifies that both temporary and regular types can be matched, but the type will be treated as specified. *implicit* is _UNSAFE_:

```
def foo ( a : Foo implicit )     // accepts Foo and Foo#, a will be treated as Foo
def foo ( a : Foo# implicit )    // accepts Foo and Foo#, a will be treated as Foo#
```

explicit specifies that LSP will not be applied, and only exact type match will be accepted:

```
def foo ( a : Foo )              // accepts Foo and any type that is inherited from␣
↪Foo directly or indirectly
def foo ( a : Foo explicit )     // accepts Foo only
```

2.27.4 options

Multiple options can be specified as a function argument:

```
def foo ( a : int | float )    // accepts int or float
```

Optional types always make function generic.

Generic options will be matched in the order listed:

```
def foo ( a : Bar explicit | Foo )    // first will try to match exactly Bar, than␣
↪anything else inherited from Foo
```

|# shortcat matches previous type, with temporary flipped:

```
def foo ( a : Foo |# )    // accepts Foo and Foo# in that order
def foo ( a : Foo# |# )   // accepts Foo# and Foo in that order
```

2.27.5 typedecl

Consider the following example:

```
struct A
    id : string

struct B
    id : int

def get_table_from_id(t : auto(T))
    var tab : table<typedecl(t.id); T>   // NOTE typedecl
    return <- tab

[export]
def main
    var a : A
    var b : B
    var aTable <- get_table_from_id(a)
    var bTable <- get_table_from_id(b)
    print("{typeinfo(typename aTable)}\n")
    print("{typeinfo(typename bTable)}\n")
```

Here table is created with a key type of *id* field of the provided struct. This feature allows to create types based on the provided expression type.

2.27.6 generic tuples and type<> expressions

Consider the following example:

```
tuple Handle
    h : auto(HandleType)
    i : int

def make_handle ( t : auto(HandleType) ) : Handle
    var h : type<Handle> // NOTE type<Handle>
    return h

def take_handle ( h : Handle )
    print("count = {h.i} of type {typeinfo(typename type<HandleType>)}\n")

[export]
def main
    let h = make_handle(10)
    take_handle(h)
```

In the function make_handle, the type of the variable h is created with the type<> expression. type<> is inferred in context (this time based on a function argument). This feature allows to create types based on the provided expression type.

Generic function take_handle takes any Handle type, but only Handle type tuple.

This carries some similarity to the C++ template system, but is a bit more limited due to tuples being weak types.

2.28 Macros

In Daslang, macros are the machinery that allow direct manipulation of the syntax tree.

Macros are exposed via the daslib/ast module and daslib/ast_boost helper module.

Macros are evaluated at compilation time during different compilation passes. Macros assigned to a specific module are evaluated as part of the module every time that module is included.

2.28.1 Compilation passes

The Daslang compiler performs compilation passes in the following order for each module (see *Modules*):

1. Parser transforms das program to AST
    1. If there are any parsing errors, compilation stops
2. *apply* is called for every function or structure
    1. If there are any errors, compilation stops
3. Infer pass repeats itself until no new transformations are reported
    1. Built-in infer pass happens
        1. *transform* macros are called for every function or expression
    2. Macro passes happen
4. If there are still any errors left, compilation stops
5. *finish* is called for all functions and structure macros

6. Lint pass happens
    1. If there are any errors, compilation stops
7. Optimization pass repeats itself until no new transformations are reported
    1. Built-in optimization pass happens
    2. Macro optimization pass happens
8. If there are any errors during optimization passes, compilation stops
9. If the module contains any macros, simulation happens
    1. If there are any simulation errors, compilation stops
    2. Module macro functions (annotated with *_macro*) are invoked
        1. If there are any errors, compilation stops

Modules are compiled in *require* order.

2.28.2 Invoking macros

The `[_macro]` annotation is used to specify functions that should be evaluated at compilation time . Consider the following example from daslib/ast_boost:

```
[_macro,private]
def setup
    if is_compiling_macros_in_module("ast_boost")
        add_new_function_annotation("macro", new MacroMacro())
```

The *setup* function is evaluated after the compilation of each module, which includes ast_boost. The `is_compiling_macros_in_module` function returns true if the currently compiled module name matches the argument. In this particular example, the function annotation `macro` would only be added once: when the module *ast_boost* is compiled.

Macros are invoked in the following fashion:

1. Class is derived from the appropriate base macro class
2. Adapter is created
3. Adapter is registered with the module

For example, this is how this lifetime cycle is implemented for the reader macro:

```
def add_new_reader_macro ( name:string; someClassPtr )
    var ann <- make_reader_macro(name, someClassPtr)
    this_module() |> add_reader_macro(ann)
    unsafe
        delete ann
```

2.28.3 AstFunctionAnnotation

The AstFunctionAnnotation macro allows you to manipulate calls to specific functions as well as their function bodies. Annotations can be added to regular or generic functions.

add_new_function_annotation adds a function annotation to a module. There is additionally the [function_macro] annotation which accomplishes the same thing.

AstFunctionAnnotation allows several different manipulations:

```
class AstFunctionAnnotation
    def abstract transform ( var call : smart_ptr<ExprCallFunc>; var errors : das_
↪string ) : ExpressionPtr
    def abstract verifyCall ( var call : smart_ptr<ExprCallFunc>; args,
↪progArgs:AnnotationArgumentList; var errors : das_string ) : bool
    def abstract apply ( var func:FunctionPtr; var group:ModuleGroup;␣
↪args:AnnotationArgumentList; var errors : das_string ) : bool
    def abstract finish ( var func:FunctionPtr; var group:ModuleGroup; args,
↪progArgs:AnnotationArgumentList; var errors : das_string ) : bool
    def abstract patch ( var func:FunctionPtr; var group:ModuleGroup; args,
↪progArgs:AnnotationArgumentList; var errors : das_string; var astChanged:bool& ) :␣
↪bool
    def abstract fixup ( var func:FunctionPtr; var group:ModuleGroup; args,
↪progArgs:AnnotationArgumentList; var errors : das_string ) : bool
    def abstract lint ( var func:FunctionPtr; var group:ModuleGroup; args,
↪progArgs:AnnotationArgumentList; var errors : das_string ) : bool
    def abstract complete ( var func:FunctionPtr; var ctx:smart_ptr<Context> ) : void
    def abstract isCompatible ( var func:FunctionPtr; var types:VectorTypeDeclPtr;␣
↪decl:AnnotationDeclaration; var errors:das_string ) : bool
    def abstract isSpecialized : bool
    def abstract appendToMangledName ( func:FunctionPtr; decl:AnnotationDeclaration;␣
↪var mangledName:das_string ) : void
```

transform lets you change calls to the function and is applied at the infer pass. Transform is the best way to replace or modify function calls with other semantics.

verifyCall is called durng the *lint* phase on each call to the function and is used to check if the call is valid.

apply is applied to the function itself before the infer pass. Apply is typically where global function body modifications or instancing occurs.

finish is applied to the function itself after the infer pass. It's only called on non-generic functions or instances of the generic functions. finish is typically used to register functions, notify C++ code, etc. After this, the function is fully defined and inferred, and can no longer be modified.

patch is called after the infer pass. If patch sets astChanged to true, the infer pass will be repeated.

fixup is called after the infer pass. It's used to fixup the function's body.

lint is called during the *lint* phase on the function itself and is used to verify that the function is valid.

complete is called during the *simulate* portion of context creation. At this point Context is available.

isSpecialized must return true if the particular function matching is governed by contracts. In that case, isCompatible is called, and the result taken into account.

isCompatible returns true if a specialized function is compatible with the given arguments. If a function is not compatible, the errors field must be specified.

appendToMangledName is called to append a mangled name to the function. That way multiple functions with the same type signature can exist and be differentiated between.

Lets review the following example from *ast_boost* of how the macro annotation is implemented:

```
class MacroMacro : AstFunctionAnnotation
    def override apply ( var func:FunctionPtr; var group:ModuleGroup;␣
↪args:AnnotationArgumentList; var errors : das_string ) : bool
        compiling_program().flags |= ProgramFlags needMacroModule
        func.flags |= FunctionFlags init
        var blk <- new [[ExprBlock() at=func.at]]
        var ifm <- new [[ExprCall() at=func.at, name:="is_compiling_macros"]]
        var ife <- new [[ExprIfThenElse() at=func.at, cond<-ifm, if_true<-func.body]]
        push(blk.list,ife)
        func.body <- blk
        return true
```

During the *apply* pass the function body is appended with the if is_compiling_macros() closure. Additionally, the init flag is set, which is equivalent to a _macro annotation. Functions annotated with [macro] are evaluated during module compilation.

2.28.4 AstBlockAnnotation

AstBlockAnnotation is used to manipulate block expressions (blocks, lambdas, local functions):

```
class AstBlockAnnotation
    def abstract apply ( var blk:smart_ptr<ExprBlock>; var group:ModuleGroup;␣
↪args:AnnotationArgumentList; var errors : das_string ) : bool
    def abstract finish ( var blk:smart_ptr<ExprBlock>; var group:ModuleGroup; args,
↪progArgs:AnnotationArgumentList; var errors : das_string ) : bool
```

add_new_block_annotation adds a function annotation to a module. There is additionally the [block_macro] annotation which accomplishes the same thing.

apply is called for every block expression before the infer pass.

finish is called for every block expression after infer pass.

2.28.5 AstStructureAnnotation

The AstStructureAnnotation macro lets you manipulate structure or class definitions via annotation:

```
class AstStructureAnnotation
    def abstract apply ( var st:StructurePtr; var group:ModuleGroup;␣
↪args:AnnotationArgumentList; var errors : das_string ) : bool
    def abstract finish ( var st:StructurePtr; var group:ModuleGroup;␣
↪args:AnnotationArgumentList; var errors : das_string ) : bool
    def abstract patch ( var st:StructurePtr; var group:ModuleGroup;␣
↪args:AnnotationArgumentList; var errors : das_string; var astChanged:bool& ) : bool
    def abstract complete ( var st:StructurePtr; var ctx:smart_ptr<Context> ) : void
```

add_new_structure_annotation adds a function annotation to a module. There is additionally the [structure_macro] annotation which accomplishes the same thing.

AstStructureAnnotation allows 2 different manipulations:

```
class AstStructureAnnotation
    def abstract apply ( var st:StructurePtr; var group:ModuleGroup;␣
↪args:AnnotationArgumentList; var errors : das_string ) : bool
    def abstract finish ( var st:StructurePtr; var group:ModuleGroup;␣
↪args:AnnotationArgumentList; var errors : das_string ) : bool
```

`apply` is invoked before the infer pass. It is the best time to modify the structure, generate some code, etc.

`finish` is invoked after the successful infer pass. Its typically used to register structures, perform RTTI operations, etc. After this, the structure is fully inferred and defined and can no longer be modified afterwards.

`patch` is invoked after the infer pass. If patch sets astChanged to true, the infer pass will be repeated.

`complete` is invoked during the *simulate* portion of context creation. At this point Context is available.

An example of such annotation is *SetupAnyAnnotation* from daslib/ast_boost.

2.28.6 AstEnumerationAnnotation

The `AstStructureAnnotation` macro lets you manipulate enumerations via annotation:

```
class AstEnumerationAnnotation
    def abstract apply ( var st:EnumerationPtr; var group:ModuleGroup;␣
↪args:AnnotationArgumentList; var errors : das_string ) : bool
```

`add_new_enumeration_annotation` adds a function annotation to a module. There is additionally the `[enumeration_macro]` annotation which accomplishes the same thing.

`apply` is invoked before the infer pass. It is the best time to modify the enumeration, generate some code, etc.

2.28.7 AstVariantMacro

`AstVariantMacro` is specialized in transforming `is`, `as`, and `?as` expressions.

`add_new_variant_macro` adds a variant macro to a module. There is additionally the `[variant_macro]` annotation which accomplishes the same thing.

Each of the 3 transformations are covered in the appropriate abstract function:

```
class AstVariantMacro
    def abstract visitExprIsVariant     ( prog:ProgramPtr; mod:Module?; expr:smart_ptr
↪<ExprIsVariant> ) : ExpressionPtr
    def abstract visitExprAsVariant     ( prog:ProgramPtr; mod:Module?; expr:smart_ptr
↪<ExprAsVariant> ) : ExpressionPtr
    def abstract visitExprSafeAsVariant ( prog:ProgramPtr; mod:Module?; expr:smart_ptr
↪<ExprSafeAsVariant> ) : ExpressionPtr
```

Let’s review the following example from daslib/ast_boost:

```
// replacing ExprIsVariant(value,name) => ExprOp2('==",value.__rtti,"name")
// if value is ast::Expr*
class BetterRttiVisitor : AstVariantMacro
    def override visitExprIsVariant(prog:ProgramPtr; mod:Module?;expr:smart_ptr
↪<ExprIsVariant>) : ExpressionPtr
        if isExpression(expr.value._type)
            var vdr <- new [[ExprField() at=expr.at, name:="__rtti", value <- clone_
↪expression(expr.value)]]
```

(continues on next page)

(continued from previous page)

```
            var cna <- new [[ExprConstString() at=expr.at, value:=expr.name]]
            var veq <- new [[ExprOp2() at=expr.at, op:="==", left<-vdr, right<-cna]]
            return veq
        return [[ExpressionPtr]]

// note the following ussage
class GetHintFnMacro : AstFunctionAnnotation
    def override transform ( var call : smart_ptr<ExprCall>; var errors : das_string␣
↪) : ExpressionPtr
        if call.arguments[1] is ExprConstString     // HERE EXPRESSION WILL BE␣
↪REPLACED
            ...
```

Here, the macro takes advantage of the ExprIsVariant syntax. It replaces the `expr is TYPENAME` expression with an `expr.__rtti = "TYPENAME"` expression. The `isExpression` function ensures that *expr* is from the *ast::Expr** family, i.e. part of the Daslang syntax tree.

2.28.8 AstReaderMacro

`AstReaderMacro` allows embedding a completely different syntax inside Daslang code.

`add_new_reader_macro` adds a reader macro to a module. There is additionally the `[reader_macro]` annotation, which essentially automates the same thing.

Reader macros accept characters, collect them if necessary, and return an *ast::Expression*:

```
class AstReaderMacro
    def abstract accept ( prog:ProgramPtr; mod:Module?; expr:ExprReader?; ch:int;␣
↪info:LineInfo ) : bool
    def abstract visit ( prog:ProgramPtr; mod:Module?; expr:smart_ptr<ExprReader> ) :␣
↪ExpressionPtr
```

Reader macros are invoked via the `% READER_MACRO_NAME ~ character_sequence` syntax. The `accept` function notifies the correct terminator of the character sequence:

```
var x = %arr~\{\}\w\x\y\n%% // invoking reader macro arr, %% is a terminator
```

Consider the implementation for the example above:

```
[reader_macro(name="arr")]
class ArrayReader : AstReaderMacro
    def override accept ( prog:ProgramPtr; mod:Module?; var expr:ExprReader?; ch:int;␣
↪info:LineInfo ) : bool
        append(expr.sequence,ch)
        if ends_with(expr.sequence,"%%")
            let len = length(expr.sequence)
            resize(expr.sequence,len-2)
            return false
        else
            return true
    def override visit ( prog:ProgramPtr; mod:Module?; expr:smart_ptr<ExprReader> ) :␣
↪ExpressionPtr
        let seqStr = string(expr.sequence)
        var arrT <- new [[TypeDecl() baseType=Type tInt]]
        push(arrT.dim,length(seqStr))
        var mkArr <- new [[ExprMakeArray() at = expr.at, makeType <- arrT]]
```

(continues on next page)

(continued from previous page)

```
        for x in seqStr
            var mkC <- new [[ExprConstInt() at=expr.at, value=x]]
            push(mkArr.values,mkC)
        return mkArr
```

The accept function macro collects symbols in the sequence. Once the sequence ends with the terminator sequence %%, accept returns false to indicate the end of the sequence.

In visit, the collected sequence is converted into a make array [[int ch1; ch2; ..]] expression.

More complex examples include the JsonReader macro in daslib/json_boost or RegexReader in daslib/regex_boost.

2.28.9 AstCallMacro

AstCallMacro operates on expressions which have function call syntax or something similar. It occurs during the infer pass.

add_new_call_macro adds a call macro to a module. The [call_macro] annotation automates the same thing:

```
class AstCallMacro
    def abstract preVisit ( prog:ProgramPtr; mod:Module?; expr:smart_ptr
↪<ExprCallMacro> ) : void
    def abstract visit ( prog:ProgramPtr; mod:Module?; expr:smart_ptr<ExprCallMacro>␣
↪) : ExpressionPtr
    def abstract canVisitArguments ( expr:smart_ptr<ExprCallMacro> ) : bool
```

apply from daslib/apply is an example of such a macro:

```
[call_macro(name="apply")]  // apply(value, block)
class ApplyMacro : AstCallMacro
    def override visit ( prog:ProgramPtr; mod:Module?; var expr:smart_ptr
↪<ExprCallMacro> ) : ExpressionPtr
        ...
```

Note how the name is provided in the [call_macro] annotation.

preVisit is called before the arguments are visited.

visit is called after the arguments are visited.

canVisitArguments is called to determine if the macro can visit the arguments.

2.28.10 AstPassMacro

AstPassMacro is one macro to rule them all. It gets entire module as an input, and can be invoked at numerous passes:

```
class AstPassMacro
    def abstract apply ( prog:ProgramPtr; mod:Module? ) : bool
```

make_pass_macro registers a class as a pass macro.

add_new_infer_macro adds a pass macro to the infer pass. The [infer] annotation accomplishes the same thing.

add_new_dirty_infer_macro adds a pass macro to the *dirty* section of infer pass. The [dirty_infer] annotation accomplishes the same thing.

Typically, such macros create an AstVisitor which performs the necessary transformations.

2.28.11 AstTypeInfoMacro

AstTypeInfoMacro is designed to implement custom type information inside a typeinfo expression:

```
class AstTypeInfoMacro
    def abstract getAstChange ( expr:smart_ptr<ExprTypeInfo>; var errors:das_string )␣
↪: ExpressionPtr
    def abstract getAstType ( var lib:ModuleLibrary; expr:smart_ptr<ExprTypeInfo>;␣
↪var errors:das_string ) : TypeDeclPtr
```

add_new_typeinfo_macro adds a reader macro to a module. There is additionally the [typeinfo_macro] annotation, which essentially automates the same thing.

getAstChange returns a newly generated ast for the typeinfo expression. Alternatively, it returns null if no changes are required, or if there is an error. In case of error, the errors string must be filled.

getAstType returns the type of the new typeinfo expression.

2.28.12 AstForLoopMacro

AstForLoopMacro is designed to implement custom processing of for loop expressions:

```
class AstForLoopMacro
    def abstract visitExprFor ( prog:ProgramPtr; mod:Module?; expr:smart_ptr<ExprFor>␣
↪) : ExpressionPtr
```

add_new_for_loop_macro adds a reader macro to a module. There is additionally the [for_loop_macro] annotation, which essentially automates the same thing.

visitExprFor is similar to that of *AstVisitor*. It returns a new expression, or null if no changes are required.

2.28.13 AstCaptureMacro

AstCaptureMacro is designed to implement custom capturing and finalization of lambda expressions:

```
class AstCaptureMacro
    def abstract captureExpression ( prog:Program?; mod:Module?; expr:ExpressionPtr;␣
↪etype:TypeDeclPtr ) : ExpressionPtr
    def abstract captureFunction ( prog:Program?; mod:Module?; var lcs:Structure?;␣
↪var fun:FunctionPtr ) : void
```

add_new_capture_macro adds a reader macro to a module. There is additionally the [capture_macro] annotation, which essentially automates the same thing.

captureExpression is called when an expression is captured. It returns a new expression, or null if no changes are required.

captureFunction is called when a function is captured. This is where custom finalization can be added to the *final* section of the function body.

2.28.14 AstCommentReader

`AstCommentReader` is designed to implement custom processing of comment expressions:

```
class AstCommentReader
    def abstract open ( prog:ProgramPtr; mod:Module?; cpp:bool; info:LineInfo ) : void
    def abstract accept ( prog:ProgramPtr; mod:Module?; ch:int; info:LineInfo ) : void
    def abstract close ( prog:ProgramPtr; mod:Module?; info:LineInfo ) : void
    def abstract beforeStructure ( prog:ProgramPtr; mod:Module?; info:LineInfo ) :␣
↪void
    def abstract afterStructure ( st:StructurePtr; prog:ProgramPtr; mod:Module?;␣
↪info:LineInfo ) : void
    def abstract beforeStructureFields ( prog:ProgramPtr; mod:Module?; info:LineInfo␣
↪) : void
    def abstract afterStructureField ( name:string; prog:ProgramPtr; mod:Module?;␣
↪info:LineInfo ) : void
    def abstract afterStructureFields ( prog:ProgramPtr; mod:Module?; info:LineInfo )␣
↪: void
    def abstract beforeFunction ( prog:ProgramPtr; mod:Module?; info:LineInfo ) : void
    def abstract afterFunction ( fn:FunctionPtr; prog:ProgramPtr; mod:Module?;␣
↪info:LineInfo ) : void
    def abstract beforeGlobalVariables ( prog:ProgramPtr; mod:Module?; info:LineInfo␣
↪) : void
    def abstract afterGlobalVariable ( name:string; prog:ProgramPtr; mod:Module?;␣
↪info:LineInfo ) : void
    def abstract afterGlobalVariables ( prog:ProgramPtr; mod:Module?; info:LineInfo )␣
↪: void
    def abstract beforeVariant ( prog:ProgramPtr; mod:Module?; info:LineInfo ) : void
    def abstract afterVariant ( name:string; prog:ProgramPtr; mod:Module?;␣
↪info:LineInfo ) : void
    def abstract beforeEnumeration ( prog:ProgramPtr; mod:Module?; info:LineInfo ) :␣
↪void
    def abstract afterEnumeration ( name:string; prog:ProgramPtr; mod:Module?;␣
↪info:LineInfo ) : void
    def abstract beforeAlias ( prog:ProgramPtr; mod:Module?; info:LineInfo ) : void
    def abstract afterAlias ( name:string; prog:ProgramPtr; mod:Module?;␣
↪info:LineInfo ) : void
```

`add_new_comment_reader` adds a reader macro to a module. There is additionally the `[comment_reader]` annotation, which essentially automates the same thing.

`open` occurs when the parsing of a comment starts.

`accept` occurs for every character of the comment.

`close` occurs when a comment is over.

`beforeStructure` and `afterStructure` occur before and after each structure or class declaration, regardless of if it has comments.

`beforeStructureFields` and `afterStructureFields` occur before and after each structure or class field, regardless of if it has comments.

`afterStructureField` occurs after each field declaration.

`beforeFunction` and `afterFunction` occur before and after each function declaration, regardless of if it has comments.

`beforeGlobalVariables` and `afterGlobalVariables` occur before and after each global variable declaration, regardless of if it has comments.

`afterGlobalVariable` occurs after each individual global variable declaration.

`beforeVariant` and `afterVariant` occur before and after each variant declaration, regardless of if it has comments.

`beforeEnumeration` and `afterEnumeration` occur before and after each enumeration declaration, regardless of if it has comments.

`beforeAlias` and `afterAlias` occur before and after each alias type declaration, regardless or if it has comments.

2.28.15 AstSimulateMacro

AstSimulateMacro is designed to customize the simulation of the program:

```
class AstSimulateMacro
    def abstract preSimulate ( prog:Program?; ctx:Context? ) : bool
    def abstract simulate ( prog:Program?; ctx:Context? ) : bool
```

`preSimulate` occurs after the context has been simulated, but before all the structure and function annotation simulations.

`simulate` occurs after all the structure and function annotation simulations.

2.28.16 AstVisitor

`AstVisitor` implements the visitor pattern for the Daslang expression tree. It contains a callback for every single expression in prefix and postfix form, as well as some additional callbacks:

```
class AstVisitor
    ...
    // find
        def abstract preVisitExprFind(expr:smart_ptr<ExprFind>) : void          //␣
↪prefix
        def abstract visitExprFind(expr:smart_ptr<ExprFind>) : ExpressionPtr    //␣
↪postifx
    ...
```

Postfix callbacks can return expressions to replace the ones passed to the callback.

PrintVisitor from the *ast_print* example implements the printing of every single expression in Daslang syntax.

`make_visitor` creates a visitor adapter from the class, derived from `AstVisitor`. The adapter then can be applied to a program via the `visit` function:

```
var astVisitor = new PrintVisitor()
var astVisitorAdapter <- make_visitor(*astVisitor)
visit(this_program(), astVisitorAdapter)
```

If an expression needs to be visited, and can potentially be fully substituted, the `visit_expression` function should be used:

```
expr <- visit_expression(expr,astVisitorAdapter)
```

2.29 Reification

Expression reification is used to generate AST expression trees in a convenient way. It provides a collection of escaping sequences to allow for different types of expression substitutions. At the top level, reification is supported by multiple call macros, which are used to generate different AST objects.

Reification is implemented in daslib/templates_boost.

2.29.1 Simple example

Let's review the following example:

```
var foo = "foo"
var fun <- qmacro_function("madd") <| $ ( a, b )
    return $i(foo) * a + b
print(describe(fun))
```

The output would be:

```
def public madd (  a:auto const;  b:auto const ) : auto
    return (foo * a) + b
```

What happens here is that call to macro `qmacro_function` generates a new function named *madd*. The arguments and body of that function are taken from the block, which is passed to the function. The escape sequence $i takes its argument in the form of a string and converts it to an identifier (ExprVar).

2.29.2 Quote macros

Reification macros are similar to `quote` expressions because the arguments are not going through type inference. Instead, an ast tree is generated and operated on.

qmacro

`qmacro` is the simplest reification. The input is returned as is, after escape sequences are resolved:

```
var expr <- qmacro(2+2)
print(describe(expr))
```

prints:

```
(2+2)
```

qmacro_block

`qmacro_block` takes a block as an input and returns unquoted block. To illustrate the difference between `qmacro` and `qmacro_block`, let's review the following example:

```
var blk1 <- qmacro <| $ ( a, b ) { return a+b; }
var blk2 <- qmacro_block <| $ ( a, b ) { return a+b; }
print("{blk1.__rtti}\n{blk2.__rtti}\n")
```

The output would be:

```
ExprMakeBlock
ExprBlock
```

This is because the block sub-expression is decorated, i.e. (ExprMakeBlock(ExprBlock (. . .))), and `qmacro_block` removes such decoration.

qmacro_expr

`qmacro_expr` takes a block with a single expression as an input and returns that expression as the result. Certain expressions like *return* and such can't be an argument to a call, so they can't be passed to `qmacro` directly. The work around is to pass them as first line of a block:

```
var expr <- qmacro_block <|
    return 13
print(describe(expr))
```

prints:

```
return 13
```

qmacro_type

`qmacro_type` takes a type expression (type<. . . >) as an input and returns the subtype as a TypeDeclPtr, after resolving the escape sequences. Consider the following example:

```
var foo <- typeinfo(ast_typedecl type<int>)
var typ <- qmacro_type <| type<$t(foo)?>
print(describe(typ))
```

TypeDeclPtr foo is passed as a reification sequence to `qmacro_type`, and a new pointer type is generated. The output is:

```
int?
```

qmacro_function

`qmacro_function` takes two arguments. The first one is the generated function name. The second one is a block with a function body and arguments. By default, the generated function only has the *FunctionFlags generated* flag set.

qmacro_variable

`qmacro_variable` takes a variable name and type expression as an input, and returns the variable as a VariableDeclPtr, after resolving the escape sequences:

```
var vdecl <- qmacro_variable("foo", type<int>)
print(describe(vdecl))
```

prints:

```
foo:int
```

2.29.3 Escape sequences

Reification provides multiple escape sequences, which are used for miscellaneous template substitution.

$i(ident)

`$i` takes a `string` or `das_string` as an argument and substitutes it with an identifier. An identifier can be substituted for the variable name in both the variable declaration and use:

```
var bus = "bus"
var qb <- qmacro_block <|
    let $i(bus) = "busbus"
    let t = $i(bus)
print(describe(qb))
```

prints:

```
let  bus:auto const = "busbus"
let  t:auto const = bus
```

$f(field-name)

`$f` takes a `string` or `das_string` as an argument and substitutes it with a field name:

```
var bar = "fieldname"
var blk <- qmacro_block <|
    foo.$f(bar) = 13
print(describe(blk))
```

prints:

```
foo.fieldname = 13
```

$v(value)

`$v` takes any value as an argument and substitutes it with an expression which generates that value. The value does not have to be a constant expression, but the expression will be evaluated before its substituted. Appropriate *make* infrastructure will be generated:

```
var t = [[auto 1,2.,"3"]]
var expr <- qmacro($v(t))
print(describe(expr))
```

prints:

```
[[1,2f,"3"]]
```

In the example above, a tuple is substituted with the expression that generates this tuple.

$e(expression)

`$e` takes any expression as an argument in form of an `ExpressionPtr`. The expression will be substituted as-is:

```
var expr <- quote(2+2)
var qb <- qmacro_block <|
    let foo = $e(expr)
print(describe(qb))
```

prints:

```
let foo:auto const = (2 + 2)
```

$b(array-of-expr)

`$b` takes an `array<ExpressionPtr>` or `das::vector<ExpressionPtr>` aka `dasvector`smart_ptr`Expression` as an argument and is replaced with each expression from the input array in sequential order:

```
var qqblk : array<ExpressionPtr>
for i in range(3)
    qqblk |> emplace_new <| qmacro(print("{$v(i)}\n"))
var blk <- qmacro_block <|
    $b(qqblk)
print(describe(blk))
```

prints:

```
print(string_builder(0, "\n"))
print(string_builder(1, "\n"))
print(string_builder(2, "\n"))
```

$a(arguments)

`$a` takes an `array<ExpressionPtr>` or `das::vector<ExpressionPtr>` aka `dasvector`smart_ptr`Expression` as an argument and replaces call arguments with each expression from the input array in sequential order:

```
var arguments <- [{ExpressionPtr quote(1+2); quote("foo")}]
var blk <- qmacro <| somefunnycall(1,$a(arguments),2)
print(describe(blk))
```

prints:

```
somefunnycall(1,1 + 2,"foo",2)
```

Note how the other arguments of the function are preserved, and multiple arguments can be substituted at the same time.

Arguments can be substituted in the function declaration itself. In that case $a expects `array<VariablePtr>`:

```
var foo <- [{VariablePtr
    new [[Variable() name:="v1", _type<-qmacro_type(type<int>)]];
    new [[Variable() name:="v2", _type<-qmacro_type(type<float>), init<-qmacro(1.2)]]
```

(continues on next page)

(continued from previous page)

```
}]
var fun <- qmacro_function("show") <| $ ( a: int; $a(foo); b : int )
    return a + b
print(describe(fun))
```

prints:

```
def public add ( a:int const; var v1:int; var v2:float = 1.2f; b:int const ) : int
    return a + b
```

$t(type)

`$t` takes a `TypeDeclPtr` as an input and substitutes it with the type expression. In the following example:

```
var subtype <- typeinfo(ast_typedecl type<int>)
var blk <- qmacro_block <|
    var a : $t(subtype)?
print(describe(blk))
```

we create pointer to a subtype:

```
var a:int? -const
```

$c(call-name)

`$c` takes a `string` or `das_string` as an input, and substitutes the call expression name:

```
var cll = "somefunnycall"
var blk <- qmacro ( $c(cll)(1,2) )
print(describe(blk))
```

prints:

```
somefunnycall(1,2)
```

2.30 Pattern matching

In the world of computer programming, there is a concept known as pattern matching. This technique allows us to take a complex value, such as an array or a variant, and compare it to a set of patterns. If the value fits a certain pattern, the matching process continues and we can extract specific values from that value. This is a powerful tool for making our code more readable and efficient, and in this section we'll be exploring the different ways that pattern matching can be used in Daslang.

In Daslang pattern matching is implement via macros in the *daslib/match* module.

2.30.1 Enumeration Matching

Daslang supports pattern matching on enumerations, which allows you to match the value of an enumeration with specific patterns. You can use this feature to simplify your code by eliminating the need for multiple if-else statements or switch statements. To match enumerations in Daslang, you use the match keyword followed by the enumeration value, and a series of if statements, each representing a pattern to match. If a match is found, the corresponding code block is executed.

Example:

```
enum Color
    Black
    Red
    Green
    Blue

def enum_match (color:Color)
    match color
        if Color Black
            return 0
        if Color Red
            return 1
        if _
            return -1
```

In the example, the enum_match function takes a Color enumeration value as an argument and returns a value based on the matched pattern. The if Color Black statement matches the Black enumeration value, the if Color Red statement matches the Red enumeration value, and the if _ statement is a catch-all that matches any other enumeration value that hasn't been explicitly matched.

2.30.2 Matching Variants

Variants in Daslang can be matched using the match statement. A variant is a discriminated union type that holds one of several possible values, each of a different type.

In the example, the IF variant has two possible values: i of type int, and f of type float. The variant_as_match function takes a value of type IF as an argument, and matches it to determine its type.

The if _ as i statement matches any value and assigns it to the declared variable i. Similarly, the if _ as f statement matches any value and assigns it to the declared variable f. The final if _ statement matches any remaining values, and returns "anything".

Example:

```
variant IF
    i : int
    f : float

def variant_as_match (v:IF)
    match v
        if _ as i
            return "int"
        if _ as f
            return "float"
        if _
            return "anything"
```

Variants can be matched in Daslang using the same syntax used to create new variants.

Here's an example:

```
def variant_match (v : IF)
    match v
        if [[IF i=$v(i)]]
            return 1
        if [[IF f=$v(f)]]
            return 2
        if _
            return 0
```

In the example above, the function variant_match takes a variant v of type IF. The first case matches v if it contains an i and binds the value of i to a variable i. In this case, the function returns 1. The second case matches v if it contains an f and binds the value of f to a variable f. In this case, the function returns 2. T he last case matches anything that doesn't match the first two cases and returns 0.

2.30.3 Declaring Variables in Pattern Matching

In Daslang, you can declare variables in pattern matching statements, including variant matching. To declare a variable, use the syntax $v(decl) where decl is the name of the variable being declared. The declared variable is then assigned the value of the matched pattern.

This feature is not restricted to variant matching, and can be used in any pattern matching statement in Daslang. In the example, the if $v(as_int) statement matches the variant value when it holds an integer and declares a variable as_int to store the value. Similarly, the if $v(as_float) statement matches the variant value when it holds a floating-point value and declares a variable as_float to store the value.

Example:

```
variant IF
    i : int
    f : float

def variant_as_match (v:IF)
    match v
        if $v(as_int) as i
            return as_int
        if $v(as_float) as f
            return as_float
        if _
            return None
```

2.30.4 Matching Structs

Daslang supports matching structs using the match statement. A struct is a composite data type that groups variables of different data types under a single name.

In the example, the Foo struct has one member a of type int. The struct_match function takes an argument of type Foo, and matches it against various patterns.

The first match if [[Foo a=13]] matches a Foo struct where a is equal to 13, and returns 0 if this match succeeds. The second match if [[Foo a=$v(anyA)]] matches any Foo struct and binds its a member to the declared variable anyA. This match returns the value of anyA if it succeeds.

Example:

```
struct Foo
    a : int

def struct_match (f:Foo)
    match f
        if [[Foo a=13]]
            return 0
        if [[Foo a=$v(anyA)]]
            return anyA
```

2.30.5 Using Guards

Daslang supports the use of guards in its pattern matching mechanism. Guards are conditions that must be satisfied in addition to a successful pattern match.

In the example, the AB struct has two members a and b of type int. The guards_match function takes an argument of type AB, and matches it against various patterns.

The first match if [[AB a=$v(a), b=$v(b)]] && (b > a) matches an AB struct and binds its a and b members to the declared variables a and b, respectively. The guard condition b > a must also be satisfied for this match to be successful. If this match succeeds, the function returns a string indicating that b is greater than a.

The second match if [[AB a=$v(a), b=$v(b)]] matches any AB struct and binds its a and b members to the declared variables a and b, respectively. No additional restrictions are placed on the match by means of a guard. If this match succeeds, the function returns a string indicating that b is less than or equal to a.

Example:

```
struct AB
    a, b : int

def guards_match (ab:AB)
    match ab
        if [[AB a=$v(a), b=$v(b)]] && (b > a)
            return "{b} > {a}"
        if [[AB a=$v(a), b=$v(b)]]
            return "{b} <= {a}"
```

2.30.6 Tuple Matching

Matching tuples in Daslang is done with double square brackets and uses the same syntax as creating a new tuple. The type of the tuple must be specified or auto can be used to indicate automatic type inference.

Here is an example that demonstrates tuple matching in Daslang:

```
def tuple_match ( A : tuple<int;float;string> )
    match A
        if [[auto 1,_,"3"]]
            return 1
        if [[auto 13,...]]       // starts with 13
            return 2
        if [[auto ...,"13"]]     // ends with "13"
            return 3
        if [[auto 2,...,"2"]]    // starts with 2, ends with "2"
            return 4
```

(continues on next page)

(continued from previous page)

```
        if _
            return 0
```

In this example, a tuple A of type tuple<int;float;string> is passed as an argument to the function tuple_match. The function uses a match statement to match different patterns in the tuple A. The if clauses inside the match statement use double square brackets to specify the pattern to be matched.

The first pattern to be matched is [[auto 1,_,"3"]]. The pattern matches a tuple that starts with the value 1, followed by any value, and ends with the string "3". The _ symbol in the pattern indicates that any value can be matched at that position in the tuple.

The second pattern to be matched is [[auto 13,…]], which matches a tuple that starts with the value 13. The … symbol in the pattern indicates that any number of values can be matched after the value 13.

The third pattern to be matched is [[auto …,"13"]], which matches a tuple that ends with the string "13". The … symbol in the pattern indicates that any number of values can be matched before the string "13".

The fourth pattern to be matched is [[auto 2,…,"2"]], which matches a tuple that starts with the value 2 and ends with the string "2".

If none of the patterns match, the _ clause is executed and the function returns 0.

2.30.7 Matching Static Arrays

Static arrays in Daslang can be matched using the double square bracket syntax, similarly to tuples. Additionally, static arrays must have their type specified, or the type can be automatically inferred using the auto keyword.

Here is an example of matching a static array of type int[3]:

```
def static_array_match ( A : int[3] )
    match A
        if [[auto $v(a);$v(b);$v(c)]] && (a+b+c)==6 // total of 3 elements, sum is 6
            return 1
        if [[int 0;...]]    // starts with 0
            return 0
        if [[int ...;13]]   // ends with 13
            return 2
        if [[int 12;...;12]]    // starts and ends with 12
            return 3
        if _
            return -1
```

In this example, the function static_array_match takes an argument of type int[3], which is a static array of three integers. The match statement uses the double square bracket syntax to match against different patterns of the input array A.

The first case, [[auto $v(a);$v(b);$v(c)]] && (a+b+c)==6, matches an array where the sum of its three elements is equal to 6. The matched elements are assigned to variables a, b, and c using the $v syntax.

The next three cases match arrays that start with 0, end with 13, and start and end with 12, respectively. The … syntax is used to match any elements in between.

Finally, the _ case matches any array that does not match any of the other cases, and returns -1 in this case.

2.30.8 Dynamic Array Matching

Dynamic arrays are used to store a collection of values that can be changed during runtime. In Daslang, dynamic arrays can be matched with patterns using similar syntax as for tuples, but with the added check for the number of elements in the array.

Here is an example of matching on a dynamic array of integers:

```
def dynamic_array_match ( A : array<int> )
    match A
        if [{auto $v(a);$v(b);$v(c)}] && (a+b+c)==6 // total of 3 elements, sum is 6
            return 1
        if [{int 0;0;0;...}]    // first 3 are 0
            return 0
        if [{int ...;1;2}]      // ends with 1,2
            return 2
        if [{int 0;1;...;2;3}]    // starts with 0,1, ends with 2,3
            return 3
        if _
            return -1
```

In the code above, the dynamic_array_match function takes a dynamic array of integers as an argument. The match statement then tries to match the elements in the array against a series of patterns.

The first pattern if [{auto $v(a);$v(b);$v(c)}] && (a+b+c)==6 matches arrays that contain three elements and the sum of those elements is 6. The $v syntax is used to match and capture the values of the elements in the array. The captured values can then be used in the condition (a+b+c)==6.

The second pattern if [{int 0;0;0;. . . }] matches arrays that start with three zeros. The . . . syntax is used to match any remaining elements in the array.

The third pattern if [{int . . . ;1;2}] matches arrays that end with the elements 1 and 2.

The fourth pattern if [{int 0;1;. . . ;2;3}] matches arrays that start with the elements 0 and 1 and end with the elements 2 and 3.

The final pattern if _ matches any array that didn't match any of the previous patterns.

It is important to note that the number of elements in the dynamic array must match the number of elements in the pattern for the match to be successful.

2.30.9 Match Expressions

In Daslang, match expressions allow you to reuse variables declared earlier in the pattern to match expressions later in the pattern.

Here's an example that demonstrates how to use match expressions to check if an array of integers is in ascending order:

```
def ascending_array_match ( A : int[3] )
    match A
        if [[int $v(x);match_expr(x+1);match_expr(x+2)]]
            return true
        if _
            return false
```

In this example, the first element of the array is matched to x. Then, the next two elements are matched using match_expr and the expression x+1 and x+2 respectively. If all three elements match, the function returns true. If there is no match, the function returns false.

2.30.10 Matching with || Expression

In Daslang, you can use the || expression to match either of the provided options in the order they appear. This is useful when you want to match a variant based on multiple criteria.

Here is an example of matching with || expression:

```
struct Bar
    a : int
    b : float

def or_match ( B:Bar )
    match B
        if [[Bar a=1, b=$v(b)]] || [[Bar a=2, b=$v(b)]]
            return b
        if _
            return 0.0
```

In this example, the function or_match takes a variant B of type Bar and matches it using the || expression. The first option matches when the value of a is 1 and b is captured as a variable. The second option matches when the value of a is 2 and b is captured as a variable. If either of these options match, the value of b is returned. If neither of the options match, 0.0 is returned.

It's important to note that for the || expression to work, both sides of the statement must declare the same variables.

2.30.11 [match_as_is] Structure Annotation

The [match_as_is] structure annotation in Daslang allows you to perform pattern matching for structures of different types. This allows you to match structures of different types in a single pattern matching expression, as long as the necessary is and as operators have been implemented for the matching types.

Here's an example of how to use the [match_as_is] structure annotation:

```
[match_as_is]
struct CmdMove : Cmd
    override rtti = "CmdMove"
    x : float
    y : float
```

In this example, the structure CmdMove is marked with the [match_as_is] annotation, allowing it to participate in pattern matching:

```
def operator is CmdMove ( cmd:Cmd )
    return cmd.rtti=="CmdMove"

def operator is CmdMove ( anything )
    return false

def operator as CmdMove ( cmd:Cmd ==const ) : CmdMove const&
    assert(cmd.rtti=="CmdMove")
    unsafe
        return reinterpret<CmdMove const&> cmd

def operator as CmdMove ( var cmd:Cmd ==const ) : CmdMove&
    assert(cmd.rtti=="CmdMove")
    unsafe
        return reinterpret<CmdMove&> cmd
```

(continues on next page)

(continued from previous page)

```
def operator as CmdMove ( anything )
    panic("Cannot cast to CmdMove")
    return [[CmdMove]]

def matching_as_and_is (cmd:Cmd)
    match cmd
        if [[CmdMove x=$v(x), y=$v(y)]]
            return x + y
        if _
            return 0.
```

In this example, the necessary is and as operators have been implemented for the CmdMove structure to allow it to participate in pattern matching. The is operator is used to determine the compatibility of the types, and the as operator is used to perform the actual type casting.

In the matching_as_and_is function, cmd is matched against the CmdMove structure using the [[CmdMove x=$v(x), y=$v(y)]] pattern. If the match is successful, the values of x and y are extracted and the sum is returned. If the match is not successful, the catch-all _ case is matched, and 0.0 is returned.

Note that the [match_as_is] structure annotation only works if the necessary is and as operators have been implemented for the matching types. In the example above, the necessary is and as operators have been implemented for the CmdMove structure to allow it to participate in pattern matching.

2.30.12 [match_copy] Structure Annotation

The [match_copy] structure annotation in Daslang allows you to perform pattern matching for structures of different types. This allows you to match structures of different types in a single pattern matching expression, as long as the necessary match_copy function has been implemented for the matching types.

Here's an example of how to use the [match_copy] structure annotation:

```
[match_copy]
struct CmdLocate : Cmd
    override rtti = "CmdLocate"
    x : float
    y : float
    z : float
```

In this example, the structure CmdLocate is marked with the [match_copy] annotation, allowing it to participate in pattern matching.

The match_copy function is used to match structures of different types. Here's an example of the implementation of the match_copy function for the CmdLocate structure:

```
def match_copy ( var cmdm:CmdLocate; cmd:Cmd )
    if cmd.rtti != "CmdLocate"
        return false
    unsafe
        cmdm = reinterpret<CmdLocate const&> cmd
    return true
```

In this example, the match_copy function takes two parameters: cmdm of type CmdLocate and cmd of type Cmd. The purpose of this function is to determine if the cmd parameter is of type CmdLocate. If it is, the function performs a type cast to CmdLocate using the reinterpret, and assigns the result to cmdm. The function then returns true to indicate that the type cast was successful. If the cmd parameter is not of type CmdLocate, the function returns false.

Here's an example of how the match_copy function is used in a matching_copy function:

```
def matching_copy ( cmd:Cmd )
    match cmd
        if [[CmdLocate x=$v(x), y=$v(y), z=$v(z)]]
            return x + y + z
        if _
            return 0.
```

In this example, the matching_copy function takes a single parameter cmd of type Cmd. This function performs a type matching operation on the cmd parameter to determine its type. If the cmd parameter is of type CmdLocate, the function returns the sum of the values of its x, y, and z fields. If the cmd parameter is of any other type, the function returns 0.

Note that the [match_copy] structure annotation only works if the necessary match_copy function has been implemented for the matching types. In the example above, the necessary match_copy function has been implemented for the CmdLocate structure to allow it to participate in pattern matching.

2.30.13 Static Matching

Static matching is a way to match on generic expressions Daslang. It works similarly to regular matching, but with one key difference: when there is a type mismatch between the match expression and the pattern, the match will be ignored at compile-time, as opposed to a compilation error. This makes static matching robust for generic functions.

The syntax for static matching is as follows:

```
static_match match_expression
    if pattern_1
        return result_1
    if pattern_2
        return result_2
    ...
    if _
        return result_default
```

Here, match_expression is the expression to be matched against the patterns. Each pattern is a value or expression that the match_expression will be compared against. If the match_expression matches one of the patterns, the corresponding result will be returned. If none of the patterns match, the result_default will be returned. If pattern can't be matched, it will be ignored.

Here is an example:

```
enum Color
    red
    green
    blue

def enum_static_match ( color, blah )
    static_match color
        if Color red
            return 0
        if match_expr(blah)
            return 1
        if _
            return -1
```

In this example, color is matched against the enumeration values red, green, and blue. If the match expression color is equal to the enumeration value red, 0 will be returned. If the match expression color is equal to the value of blah, 1 will be returned. If none of the patterns match, -1 will be returned.

Note that match_expr is used to match blah against the match expression color, rather than directly matching blah against the enumeration value.

If color is not Color first match will fail. If blah is not Color, second match will fail. But the function will always compile.

2.30.14 match_type

The match_type subexpression in Daslang allows you to perform pattern matching based on the type of an expression. It is used within the static_match statement to specify the type of expression that you want to match.

The syntax for match_type is as follows:

```
if match_type<Type> expr
    // code to run if match is successful
```

where Type is the type that you want to match and. expr is the expression that you want to match against.

Here's an example of how to use the match_type subexpression:

```
def static_match_by_type (what)
    static_match what
        if match_type<int> $v(expr)
            return expr
        if _
            return -1
```

In this example, what is the expression that is being matched. If what is of type int, then it is assigned to the variable $v and the expression expr is returned. If what is not of type int, the match falls through to the catch-all _ case, and -1 is returned.

Note that the match_type subexpression only matches types, and mismatched values are ignored. This is in contrast to regular pattern matching, where both type and value must match for a match to be successful.

2.30.15 Multi-Match

In Daslang, you can use the multi_match feature to match multiple values in a single expression. This is useful when you want to match a value based on several different conditions.

Here is an example of using the multi_match feature:

```
def multi_match_test ( a:int )
    var text = "{a}"
    multi_match a
        if 0
            text += " zero"
        if 1
            text += " one"
        if 2
            text += " two"
        if $v(a) && (a % 2 == 0) && (a!=0)
            text += " even"
        if $v(a) && (a % 2 == 1)
```

(continues on next page)

(continued from previous page)

```
            text += " odd"
    return text
```

In this example, the function multi_match_test takes an integer value a and matches it using the multi_match feature. The first three options match when a is equal to 0, 1, or 2, respectively. The fourth option matches when a is not equal to 0 and is an even number. The fifth option matches when a is an odd number. The variable text is updated based on the matching conditions. The final result is returned as the string representation of text.

It's important to note that the multi_match feature allows for multiple conditions to be matched in a single expression. This makes the code more concise and easier to read compared to using multiple match and if statements.

The same example using regular match would look like this:

```
def multi_match_test ( a:int )
    var text = "{a}"
    match a
        if 0
            text += " zero"
    match a
        if 1
            text += " one"
    match a
        if 2
            text += " two"
    match a
        if $v(a) && (a % 2 == 0) && (a!=0)
            text += " even"
    match a
        if $v(a) && (a % 2 == 1)
            text += " odd"
    return text
```

static_multi_match is a variant of *multi_match* that works with *static_match*.

2.31 Context

Daslang environments are organized into contexts. Compiling Daslang program produces the 'Program' object, which can then be simulated into the 'Context'.

***Context* consists of**

- name and flags
- functions code
- global variables data
- shared global variable data
- stack
- dynamic memory heap
- dynamic string heap
- constant string heap
- runtime debug information

- locks
- miscellaneous lookup infrastructure

In some sense *Context* can be viewed as Daslang virtual machine. It is the object that is responsible for executing the code and keeping the state. It can also be viewed as an instance of the class, which methods can be accessed when marked as [export].

Function code, constant string heap, runtime debug information, and shared global variables are shared between cloned contexts. That allows to keep relatively small profile for the context instance.

Stack can be optionally shared between multiple contexts of different type, to keep memory profile even smaller.

2.31.1 Initialization and shutdown

Through its lifetime *Context* goes through the initialization and the shutdown. Context initialization is implemented in *Context::runInitScript* and shutdown is implemented in *Context::runShutdownScript*. These functions are called automatically when *Context* is created, cloned, and destroyed. Depending on the user application and the *CodeOfPolicies*, they may also be called when *Context::restart* or *Context::restartHeaps* is called.

Its initialized in the following order.

1. All global variables are initialized in order they are declared per module.
2. All functions tagged as [init] are called in order they are declared per module, except for specifically ordered ones.
3. All specifically ordered functions tagged as [init] are called in order they appear after the topological sort.

The topological sort order for the init functions is specified in the init annotation.

- *tag* attribute specifies that function will appear during the specified pass
- *before* attribute specifies that function will appear before the specified pass
- *after* attribute specifies that function will appear after the specified pass

Consider the following example:

```
[init(before="middle")]
def a
    order |> push("a")
[init(tag="middle")]
def b
    order |> push("b")
[init(tag="middle")]
def c
    order |> push("c")
[init(after="middle")]
def d
    order |> push("d")
```

Functions will appear in the order of

1. d
2. b or c, in any order
3. a

Context shuts down runs all functions marked as [finalize] in the order they are declared per module.

2.31.2 Macro contexts

For each module which contains macros individual context is created and initialized. On top of regular functions, functions tagged as [macro] or [_macro] are called during the initialization.

Functions tagged as [macro_function] are excluded from the regular context, and only appear in the macro contexts.

Unless macro module is marked as shared, it will be shutdown after the compilation. Shared macro modules are initialized during their first compilation, and are shut down during the environment shutdown.

2.31.3 Locking

Context contains *recursive_mutex*, and can be specifically locked and unlocked with the *lock_context* or *lock_this_context* RAII block. Cross context calls *invoke_in_context* automatically lock the target context.

2.31.4 Lookups

Global variables and functions can be looked up by name or by mangled name hash on both Daslang and C++ side.

2.32 Locks

There are several locking mechanisms available in Daslang. They are designed to ensure runtime safety of the code.

2.32.1 Context locks

Context can be locked and unlocked via *lock* and *unlock* functions from the C++ side. When locked *Context* can not be restarted. *tryRestartAndLock* restarts context if its not locked, and then locks it regardless. The main reason to lock context is when data on the heap is accessed externally. Heap collection is safe to do on a locked context.

2.32.2 Array and Table locks

Array or *Table* can be locked and unlocked explicitly. When locked, they can't be modified. Calling *resize*, *reserve*, *push*, *emplace*, *erase*, etc on the locked *Array* ` will cause *panic*. Accessing locked *Table* elements via [] operation would cause *panic*.

Arrays are locked when iterated over. This is done to prevent modification of the array while it is being iterated over. *keys* and *values* iterators lock *Table* as well. *Tables* are also locked during the *find** operations.

2.32.3 Array and Table lock checking

Array and *Table* lock checking occurs on the data structures, which internally contain other *Arrays* or *Tables*.

Consider the following example:

```
var a : array < array<int> >
...
for b in a[0]
    a |> resize(100500)
```

The *resize* operation on the *a* array will cause *panic* because *a[0]* is locked during the iteration. This test, however, can only happen in runtime. The compiler generates custom *resize* code, which verifies locks:

```
def private builtin`resize ( var Arr:array<array<int> aka numT> explicit; newSize:int␣
↪const )
    _builtin_verify_locks(Arr)
    __builtin_array_resize(Arr,newSize,24,__context__)
```

The *_builtin_verify_locks* iterates over provided data, and for each *Array* or *Table* makes sure it does not lock. If its locked *panic* occurs.

Custom operations will only be generated, if the underlying type needs lock checks.

Here are the list of operations, which perform lock check on the data structures::

- a <- b
- return <- a
- resize
- reserve
- push
- push_clone
- emplace
- pop
- erase
- clear
- insert
- a[b] for the *Table*

Lock checking can be explicitly disabled

- for the *Array* or the *Table* by using *set_verify_array_locks* and *set_verify_table_locks* functions.
- for a structure type with the [skip_field_lock_check] structure annotation
- for the entire function with the [skip_lock_check] function annotation
- for the entire context with the *options skip_lock_checks*
- for the entire context with the *set_verify_context_locks* function

CHAPTER

THREE

EMBEDDING DASLANG INTO C++

3.1 Daslang Virtual Machine

3.1.1 Tree-based Interpretion

The Virtual Machine of Daslang consists of a small execution context, which manages stack and heap allocation. The compiled program itself is a Tree of "Nodes" (called SimNode) which can evaluate virtual functions. These Nodes don't check argument types, assuming that all such checks were done by compiler. Why tree-based? Tree-based interpreters are slow!

Yes, Tree-based interpreters are infamous for their low performance compared to byte-code interpreters.

However, this opinion is typically based on a comparison between AST (abstract syntax tree) interpreters of dynamically typed languages with optimized register- or stack-based optimized interpreters. Due to their simplicity of implementation, AST-based interpreters are also seen in a lot of "home-brewed" naive interpreters, giving tree-based interpreters additional bad fame. The AST usually contains a lot of data that is useless or unnecessary for interpretation, and big tree depth and complexity.

It is also hard to even make an AST interpreter of a statically typed language which will somehow benefit from statically typed data - basically each tree node visitor will still return both the value and type information in generic form.

In comparison, a good byte-code VM interpreter of a typical dynamically typed language will feature a tight core loop of all or the most frequent instructions (probably with computed goto) and additionally can statically (or during execution) infer types and optimize code for it.

Register- and **stack-based**- VMs each have their own trade-offs, notably with generally fewer generated instructions/fused instructions, fewer memory moves/indirect memory access for register-based VMs, and smaller instruction sets and easier implementation for stack-based VMs.

The more "core types" the VM has, the more instructions will probably be needed in the instruction set and/or the instruction cost increases. Although dynamically typed languages usually don't have many core types, and some can embed all their main type's values and type information in just 64bits (using NAN-tagging, for example), that still usually leaves one of these core types (table/class/object) to be implemented with associative containers lookups (unordered_map/hashmap). That is not optimal for cache locality/performance, and also makes interop with host (C++) types slow and inefficient.

Interop with host/C++ functions because of that is usually slow, as it requires complex and slow arguments/return value type conversion, and/or associative table(s) lookup.

So, typically, host functions calls are very "heavy", and programmers usually can't optimize scripts by extracting just some of functionality into C++ function - they have to re-write big chunks/loops.

Increasing the amount of core internal types can help (for example, making "float3", a typical type in game development, one of the "core" types), but this makes instruction set bigger/slower, increases the complexity of type conversion, and usually introduces mandatory indirection (due to limited bitsize of value type), which is also not good for cache locality.

However, **Daslang** *does not* interpret the AST, nor is it a dynamically typed language.

Instead, for run-time program execution it emits a different tree (Simulation Tree), which doesn't require type information to be provided for arguments/return types, since it is statically typed, and all the types are known.

For the Daslang ABI, 128bit words are used, which are natural to most of modern hardware.

The chosen representation helps branch prediction, increases cache locality, and provides a mix of stack and register based code - each 'Node' utilizes native machine registers.

It is also important to note that the amount of "types" and "instructions" doesn't matter much - what matters is the amount of different instructions used in a particular program/function.

Type conversion between the VM and C++ ABIs is straightforward (and for most of types is as simple as a move instruction), so it is very fast and cache-friendly.

It also makes it possible for the programmer to optimize particular functionality (in interpretation) by extracting it to a C++/host function - basically "fusing" instructions into one.

Adding new user-types is rather simple and not painful performance- or engineering-wise.

"Value" types have to fit into 128bits and have to be relocatable and zero-initialized (i.e. should be trivially destructible, and allow memcpy and memsetting with zeroes); all other types are "RefTypes" or "Boxed Types", which means they can be operated on in the script only as references/pointers.

There are no limits on the amount of user types, neither is there a performance impact caused by using such types (besides the obvious cost of indirection for Boxed/Ref Types).

Using generalized nodes additionally allows for a seamless mix of interpretation and Ahead of Time compiled code in the run-time - i.e. if some of the functions in the script were changed, the unchanged portion would still be running the optimized AoT code.

These are the main reasons why tree-based interpretation (not to be confused with AST-based) was chosen for the Daslang interpreter, and why its interpreter is faster than most, if not all, byte code based script interpreters.

3.1.2 Execution Context

The Daslang Execution Context is light-weight. It basically consists of stack allocation and two heap allocators (for strings and everything else). One Context can be used to execute different programs; however, if the program has any global state in the heap, all calls to the program have to be done within the same Context.

It is possible to call stop-the-world garbage collection on a Context (this call is better to be done outside the program execution; it's unsafe otherwise).

However, the cost of resetting context (i.e. deallocate all memory) is extremely low, and (depending on memory usage) can be as low as several instructions, which allows the simplest and fastest form of memory management for all of the stateless scripts - just reset the context each frame or each call. This basically turns Context heap management into form of 'bump/stack allocator', significantly simplifying and optimizing memory management.

There are certain ways (including code of policies) to ensure that none of the scripts are using global variables at all, or at least global variables which require heap memory.

For example, one can split all contexts into several cateories: one context for all stateless script programs, and one context for each GC'ed (or, additionally, `unsafe`) script. The stateless context is then reset as often as needed (for example, each 'top' call from C++ or each frame/tick), and on GC-ed contexts one can call garbage collection as soon as it is needed (using some heurestics of memory usage/performance).

Each context can be used only in one thread simultaneously, i.e. for multi-threading you will need one Context for each simultaneously running thread.

To exchange data/communicate between different contexts, use 'channels' or some other custom-brewed C++ hosted code of that kind.

3.2 Modules and C++ bindings

3.2.1 Builtin modules

Builtin modules are the way to expose C++ functionality to Daslang.

Let's look at the `FIO` module as an example. To create a builtin module, an application needs to do the following:

Make a C++ file where the module resides. Additionally, make a header for AOT to include.

Derive from the Module class and provide a custom module name to the constructor:

```
class Module_FIO : public Module {
public:
    Module_FIO() : Module("fio") {                  // this module name is ``fio``
        DAS_PROFILE_SECTION("Module_FIO");         // the profile section is there to␣
↪profile module initialization time
        ModuleLibrary lib;                         // module needs library to register␣
↪types and functions
        lib.addModule(this);                       // add current module to the library
        lib.addBuiltInModule();                    // make builtin functions visible to␣
↪the library
```

Specify the AOT type and provide a prefix with C++ includes (see *AOT*):

```
virtual ModuleAotType aotRequire ( TextWriter & tw ) const override {
    tw << "#include \"Daslang/simulate/aot_builtin_fio.h\"\n";
    return ModuleAotType::cpp;
}
```

Register the module at the bottom of the C++ file using the `REGISTER_MODULE` or `REGISTER_MODULE_IN_NAMESPACE` macro:

```
REGISTER_MODULE_IN_NAMESPACE(Module_FIO,das);
```

Use the `NEED_MODULE` macro during application initialization before the Daslang compiler is invoked:

```
NEED_MODULE(Module_FIO);
```

Its possible to have additional Daslang files that accompany the builtin module, and have them compiled at initialization time via the `compileBuiltinModule` function:

```
Module_FIO() : Module("fio") {
    ...
    // add builtin module
    compileBuiltinModule("fio.das",fio_das, sizeof(fio_das));
```

What happens here is that fio.das is embedded into the executable (via the XXD utility) as a string constant.

Once everything is registered in the module class constructor, it's a good idea to verify that the module is ready for AOT via the `verifyAotReady` function. It's also a good idea to verify that the builtin names are following the correct naming conventions and do not collide with keywords via the `verifyBuiltinNames` function:

```
Module_FIO() : Module("fio") {
    ...
    // lets verify all names
    uint32_t verifyFlags = uint32_t(VerifyBuiltinFlags::verifyAll);
    verifyFlags &= ~VerifyBuiltinFlags::verifyHandleTypes;  // we skip annotatins due␣
↪to FILE and FStat
    verifyBuiltinNames(verifyFlags);
    // and now its aot ready
    verifyAotReady();
}
```

3.2.2 ModuleAotType

Modules can specify 3 different AOT options.

`ModuleAotType::no_aot` means that no ahead of time compilation will occur for the module, as well as any other modules which require it.

`ModuleAotType::hybrid` means that no ahead of time compilation will occur for the module itself. Other modules which require this one will will have AOT, but not without performance penalties.

`ModuleAotType::cpp` means that full blown AOT will occur. It also means that the module is required to fill in `cppName` for every function, field, or property. The best way to verify it is to call `verifyAotReady` at the end of the module constructor.

Additionally, modules need to write out a full list of required C++ includes:

```
virtual ModuleAotType aotRequire ( TextWriter & tw ) const override {
    tw << "#include \"Daslang/simulate/aot_builtin_fio.h\"\n"; // like this
    return ModuleAotType::cpp;
}
```

3.2.3 Builtin module constants

Constants can be exposed via the `addConstant` function:

```
addConstant(*this,"PI",(float)M_PI);
```

The constant's type is automatically inferred, assuming type `cast` infrastructure is in place (see *cast*).

3.2.4 Builtin module enumerations

Enumerations can be exposed via *the `addEnumeration`* function:

```
addEnumeration(make_smart<EnumerationGooEnum>());
addEnumeration(make_smart<EnumerationGooEnum98>());
```

For this to work, the enumeration adapter has to be defined via the `DAS_BASE_BIND_ENUM` or `DAS_BASE_BIND_ENUM_98` C++ preprocessor macros:

```
namespace Goo {
    enum class GooEnum {
        regular
```

(continues on next page)

(continued from previous page)

```
    ,   hazardous
    };

    enum GooEnum98 {
        soft
    ,   hard
    };
}

DAS_BASE_BIND_ENUM(Goo::GooEnum, GooEnum, regular, hazardous)
DAS_BASE_BIND_ENUM_98(Goo::GooEnum98, GooEnum98, soft, hard)
```

3.2.5 Builtin module data types

Custom data types and type annotations can be exposed via the `addAnnotation` or `addStructure` functions:

```
addAnnotation(make_smart<FileAnnotation>(lib));
```

See *handles* for more details.

3.2.6 Builtin module macros

Custom macros of different types can be added via `addAnnotation`, `addTypeInfoMacro`, `addReaderMacro`, `addCallMacro`, and such. It is strongly preferred, however, to implement macros in Daslang.

See *macros* for more details.

3.2.7 Builtin module functions

Functions can be exposed to the builtin module via the `addExtern` and `addInterop` routines.

addExtern

`addExtern` exposes standard C++ functions which are not specifically designed for Daslang interop:

```
addExtern<DAS_BIND_FUN(builtin_fprint)>(*this, lib, "fprint",␣
↪SideEffects::modifyExternal, "builtin_fprint");
```

Here, the builtin_fprint function is exposed to Daslang and given the name *fprint*. The AOT name for the function is explicitly specified to indicate that the function is AOT ready.

The side-effects of the function need to be explicitly specified (see *Side-effects*). It's always safe, but inefficient, to specify `SideEffects::worstDefault`.

Let's look at the exposed function in detail:

```
void builtin_fprint ( const FILE * f, const char * text, Context * context,␣
↪LineInfoArg * at ) {
    if ( !f ) context->throw_error_at(at, "can't fprint NULL");
    if ( text ) fputs(text,(FILE *)f);
}
```

C++ code can explicitly request to be provided with a Daslang context, by adding the *Context* type argument. Making it last argument of the function makes context substitution transparent for Daslang code, i.e. it can simply call:

```
fprint(f, "boo")     // current context with be provided transparently
```

Daslang strings are very similar to C++ `char *`, however null also indicates empty string. That's the reason the *fputs* only occurs if text is not null in the example above.

Let's look at another integration example from the builtin *math* module:

```
addExtern<DAS_BIND_FUN(float4x4_translation), SimNode_ExtFuncCallAndCopyOrMove>(*this,
↪ lib, "translation",
        SideEffects::none, "float4x4_translation")->arg("xyz");
```

Here, the float4x4_translation function returns a ref type by value, i.e. *float4x4*. This needs to be indicated explicitly by specifying a templated SimNode argument for the `addExtern` function, which is `SimNode_ExtFuncCallAndCopyOrMove`.

Some functions need to return a ref type by reference:

```
addExtern<DAS_BIND_FUN(fooPtr2Ref),SimNode_ExtFuncCallRef>(*this, lib, "fooPtr2Ref",
    SideEffects::none, "fooPtr2Ref");
```

This is indicated with the `SimNode_ExtFuncCallRef` argument.

addInterop

For some functions it may be necessary to access type information as well as non-marshalled data. Interop functions are designed specifically for that purpose.

Interop functions are of the following pattern:

```
vec4f your_function_name_here ( Context & context, SimNode_CallBase * call, vec4f *␣
↪args )
```

They receive a context, calling node, and arguments. They are expected to marshal and return results, or v_zero().

`addInterop` exposes C++ functions, which are specifically designed around Daslang:

```
addInterop<
    builtin_read,               // function to register
    int,                        // function return type
    const FILE*,vec4f,int32_t   // function arguments in order
>(*this, lib, "_builtin_read",SideEffects::modifyExternal, "builtin_read");
```

The interop function registration template expects a function name as its first template argument, function return value as its second, with the rest of the arguments following.

When a function's argument type needs to remain unspecified, an argument type of `vec4f` is used.

Let's look at the exposed function in detail:

```
vec4f builtin_read ( Context & context, SimNode_CallBase * call, vec4f * args ) {
    DAS_ASSERT ( call->types[1]->isRef() || call->types[1]->isRefType() || call->
↪types[1]->type==Type::tString);
    auto fp = cast<FILE *>::to(args[0]);
    if ( !fp ) context.throw_error_at(call->debugInfo, "can't read NULL");
    auto buf = cast<void *>::to(args[1]);
```

(continues on next page)

(continued from previous page)

```
    auto len = cast<int32_t>::to(args[2]);
    int32_t res = (int32_t) fread(buf,1,len,fp);
    return cast<int32_t>::from(res);
}
```

Argument types can be accessed via the call->types array. Argument values and return value are marshalled via `cast` infrastructure (see *cast*).

3.2.8 Function side-effects

The Daslang compiler is very much an optimizin compiler and pays a lot of attention to functions' side-effects.

On the C++ side, `enum class SideEffects` contains possible side effect combinations.

`none` indicates that a function is pure, i.e it has no side-effects whatsoever. A good example would be purely computational functions like `cos` or `strlen`. Daslang may choose to fold those functions at compilation time as well as completely remove them in cases where the result is not used.

Trying to register void functions with no arguments and no side-effects causes the module initialization to fail.

`unsafe` indicates that a function has unsafe side-effects, which can cause a panic or crash.

`userScenario` indicates that some other uncategorized side-effects are in works. Daslang does not optimize or fold those functions.

`modifyExternal` indicates that the function modifies state, external to Daslang; typically it's some sort of C++ state.

`accessExternal` indicates that the function reads state, external to Daslang.

`modifyArgument` means that the function modifies one of its input parameters. Daslang will look into non-constant ref arguments and will assume that they may be modified during the function call.

Trying to register functions without mutable ref arguments and `modifyArgument` side effects causes module initialization to fail.

`accessGlobal` indicates that that function accesses global state, i.e. global Daslang variables or constants.

`invoke` indicates that the function may invoke another functions, lambdas, or blocks.

3.2.9 File access

Daslang provides machinery to specify custom file access and module name resolution.

Default file access is implemented with the `FsFileAccess` class.

File access needs to implement the following file and name resolution routines:

```
virtual das::FileInfo * getNewFileInfo(const das::string & fileName) override;
virtual ModuleInfo getModuleInfo ( const string & req, const string & from ) const␣
↪override;
```

`getNewFileInfo` provides a file name to file data machinery. It returns null if the file is not found.

`getModuleInfo` provides a module name to file name resolution machinery. Given require string *req* and the module it was called *from*, it needs to fully resolve the module:

```
struct ModuleInfo {
    string  moduleName;     // name of the module (by default tail of req)
    string  fileName;       // file name, where the module is to be found
    string  importName;     // import name, i.e. module namespace (by default same as␣
↪module name)
};
```

It is better to implement module resolution in Daslang itself, via a project.

3.2.10 Project

Projects need to export a `module_get` function, which essentially implements the default C++ `getModuleInfo` routine:

```
require strings
require daslib/strings_boost

typedef
    module_info = tuple<string;string;string> const // mirror of C++ ModuleInfo

[export]
def module_get(req,from:string) : module_info
    let rs <- split_by_chars(req,"./")                      // split request
    var fr <- split_by_chars(from,"/")
    let mod_name = rs[length(rs)-1]
    if length(fr)==0                                        // relative to local
        return [[auto mod_name, req + ".das", ""]]
    elif length(fr)==1 && fr[0]=="daslib"                   // process `daslib` prefix
        return [[auto mod_name, "{get_das_root()}/daslib/{req}.das", ""]]
    else
        pop(fr)
        for se in rs
            push(fr,se)
        let path_name = join(fr,"/") + ".das"               // treat as local path
        return [[auto mod_name, path_name, ""]]
```

The implementation above splits the require string and looks for recognized prefixes. If a module is requested from another module, parent module prefixes are used. If the root *daslib* prefix is recognized, modules are looked for from the `get_das_root` path. Otherwise, the request is treated as local path.

3.3 C++ ABI and type factory infrastructure

3.3.1 Cast

When C++ interfaces with Daslang, the *cast* ABI is followed.

- Value types are converted to and from *vec4f*, in specific memory layout
- Reference types have their address converted to and from *vec4f*

It is expected that vec4f * can be pruned to a by value type by simple pointer cast becase the Daslang interpreter will in certain cases access pre-cast data via the v_ldu intrinsic:

```
template <typename TT>
TT get_data ( vec4f * dasData ) {            // default version
    return cast<TT>::to(v_ldu((float *)dasData));
}

int32_t get_data ( vec4f * dasData ) {       // legally optimized version
    return * (int32_t *) dasData;
}
```

ABI infrastructure is implemented via the C++ cast template, which serves two primary functions:

- Casting `from` C++ to Daslang
- Casting `to` C++ from Daslang

The `from` function expects a Daslang type as an input, and outputs a vec4f.

The `to` function expects a vec4f, and outputs a Daslang type.

Let's review the following example:

```
template <>
struct cast <int32_t> {
    static __forceinline int32_t to ( vec4f x )              { return v_extract_xi(v_
↪cast_vec4i(x)); }
    static __forceinline vec4f from ( int32_t x )            { return v_cast_vec4f(v_
↪splatsi(x)); }
};
```

It implements the ABI for the int32_t, which packs an int32_t value at the beginning of the vec4f using multiplatform intrinsics.

Let's review another example, which implements default packing of a reference type:

```
template <typename TT>
struct cast <TT &> {
    static __forceinline TT & to ( vec4f a )                 { return *(TT *) v_extract_
↪ptr(v_cast_vec4i((a))); }
    static __forceinline vec4f from ( const TT & p )         { return v_cast_vec4f(v_
↪splats_ptr((const void *)&p)); }
};
```

Here, a pointer to the data is packed in a vec4f using multiplatform intrinsics.

3.3.2 Type factory

When C++ types are exposed to Daslang, type factory infrastructure is employed.

To expose any custom C++ type, use the `MAKE_TYPE_FACTORY` macro, or the `MAKE_EXTERNAL_TYPE_FACTORY` and `IMPLEMENT_EXTERNAL_TYPE_FACTORY` macro pair:

```
MAKE_TYPE_FACTORY(clock, das::Time)
```

The example above tells Daslang that the C++ type *das::Time* will be exposed to Daslang with the name *clock*.

Let's look at the implementation of the `MAKE_TYPE_FACTORY` macro:

```
#define MAKE_TYPE_FACTORY(TYPE,CTYPE) \
    namespace das { \
    template <> \
    struct typeFactory<CTYPE> { \
        static TypeDeclPtr make(const ModuleLibrary & library ) { \
            return makeHandleType(library,#TYPE); \
        } \
    }; \
    template <> \
    struct typeName<CTYPE> { \
        constexpr static const char * name() { return #TYPE; } \
    }; \
    };
```

What happens in the example above is that two templated policies are exposed to C++.

The `typeName` policy has a single static function `name`, which returns the string name of the type.

The `typeFactory` policy creates a smart pointer to Daslang the *das::TypeDecl* type, which corresponds to C++ type. It expects to find the type somewhere in the provided ModuleLibrary (see *Modules*).

3.3.3 Type aliases

A custom type factory is the preferable way to create aliases:

```
struct Point3 { float x, y, z; };

template <>
struct typeFactory<Point3> {
    static TypeDeclPtr make(const ModuleLibrary &) {
        auto t = make_smart<TypeDecl>(Type::tFloat3);
        t->alias = "Point3";
        t->aotAlias = true;
        return t;
    }
};

template <> struct typeName<Point3>   { constexpr static const char * name() { return
↪"Point3"; } };
```

In the example above, the C++ application already has a *Point3* type, which is very similar to Daslang's float3. Exposing C++ functions which operate on Point3 is preferable, so the implementation creates an alias named *Point3* which corresponds to the das Type::tFloat3.

Sometimes, a custom implementation of `typeFactory` is required to expose C++ to a Daslang type in a more native fashion. Let's review the following example:

```
  struct SampleVariant {
      int32_t _variant;
      union {
          int32_t     i_value;
          float       f_value;
          char *      s_value;
      };
  };

template <>
```

(continues on next page)

(continued from previous page)

```
struct typeFactory<SampleVariant> {
    static TypeDeclPtr make(const ModuleLibrary & library ) {
        auto vtype = make_smart<TypeDecl>(Type::tVariant);
        vtype->alias = "SampleVariant";
        vtype->aotAlias = true;
        vtype->addVariant("i_value", typeFactory<decltype(SampleVariant::i_value)>
↪::make(library));
        vtype->addVariant("f_value", typeFactory<decltype(SampleVariant::f_value)>
↪::make(library));
        vtype->addVariant("s_value", typeFactory<decltype(SampleVariant::s_value)>
↪::make(library));
        // optional validation
        DAS_ASSERT(sizeof(SampleVariant) == vtype->getSizeOf());
        DAS_ASSERT(alignof(SampleVariant) == vtype->getAlignOf());
        DAS_ASSERT(offsetof(SampleVariant, i_value) == vtype->
↪getVariantFieldOffset(0));
        DAS_ASSERT(offsetof(SampleVariant, f_value) == vtype->
↪getVariantFieldOffset(1));
        DAS_ASSERT(offsetof(SampleVariant, s_value) == vtype->
↪getVariantFieldOffset(2));
        return vtype;
    }
};
```

Here, C++ type *SomeVariant* matches the Daslang variant type with its memory layout. The code above exposes a C++ type alias and creates a corresponding TypeDecl.

3.4 Exposing C++ handled types

Handled types represent the machinery designed to expose C++ types to Daslang.

A handled type is created by deriving a custom type annotation from TypeAnnotation and adding an instance of that annotation to the desired module. For example:

```
template <typename VecT, int RowC>
class MatrixAnnotation : public TypeAnnotation {
...

typedef MatrixAnnotation<float4,4> float4x4_ann;

Module_Math() : Module("math") {
    ...
    addAnnotation(make_smart<float4x4_ann>());
```

3.4.1 TypeAnnotation

`TypeAnnotation` contains a collection of virtual methods to describe type properties, as well as methods to implement simulation nodes for the specific functionality.

`canAot` returns true if the type can appear in *AOT*:

```
virtual bool canAot(das_set<Structure *> &) const
```

`canCopy`, `canMove` and `canClone` allow a type to be copied, moved, or cloned:

```
virtual bool canMove() const
virtual bool canCopy() const
virtual bool canClone() const
```

`isPod` and `isRawPod` specify if a type is plain old data, and plain old data without pointers, respectively:

```
virtual bool isPod() const
virtual bool isRawPod() const
```

`isRefType` specifies the type ABI, i.e. if it's passed by reference or by value:

```
virtual bool isRefType() const
```

`isLocal` allows creation of a local variable of that type:

```
virtual bool isLocal() const
```

`canNew`, `canDelete` and `canDeletePtr` specify if new and delete operations are allowed for the type, as well as whether a pointer to the type can be deleted:

```
virtual bool canNew() const
virtual bool canDelete() const
virtual bool canDeletePtr() const
```

`needDelete` specifies if automatically generated finalizers are to delete this type:

```
virtual bool needDelete() const
```

`isIndexable` specifies if the index operation `[]` is allowed for the type:

```
virtual bool isIndexable ( const TypeDeclPtr & ) const
```

`isIterable` specifies if the type can be the source of a for loop:

```
virtual bool isIterable ( ) const
```

`isShareable` specifies if global variables of the type can be marked as *shared*:

```
virtual bool isShareable ( ) const
```

`isSmart` specifies, if a pointer to the type appears as a smart_ptr:

```
virtual bool isSmart() const
```

`canSubstitute` queries if LSP is allowed for the type, i.e. the type can be downcast:

```
virtual bool canSubstitute ( TypeAnnotation * /* passType */ ) const
```

`getSmartAnnotationCloneFunction` returns the clone function name for the `:=` operator substitution:

```
virtual string getSmartAnnotationCloneFunction () const { return ""; }
```

`getSizeOf` and `getAlignOf` return the size and alignment of the type, respectively:

```
virtual size_t getSizeOf() const
virtual size_t getAlignOf() const
```

`makeFieldType` and `makeSafeFieldType` return the type of the specified field (or null if the field is not found):

```
virtual TypeDeclPtr makeFieldType ( const string & ) const
virtual TypeDeclPtr makeSafeFieldType ( const string & ) const
```

`makeIndexType` returns the type of the `[]` operator, given an index expression (or null if unsupported):

```
virtual TypeDeclPtr makeIndexType ( const ExpressionPtr & /*src*/, const␣
↪ExpressionPtr & /*idx*/ ) const
```

`makeIteratorType` returns the type of the iterable variable when serving as a for loop source (or null if unsupported):

```
virtual TypeDeclPtr makeIteratorType ( const ExpressionPtr & /*src*/ ) const
```

`aotPreVisitGetField`, `aotPreVisitGetFieldPtr`, `aotVisitGetField`, and `aotVisitGetFieldPtr` generate specific AOT prefixes and suffixes for the field and pointer field dereference:

```
virtual void aotPreVisitGetField ( TextWriter &, const string & )
virtual void aotPreVisitGetFieldPtr ( TextWriter &, const string & )
virtual void aotVisitGetField ( TextWriter & ss, const string & fieldName )
virtual void aotVisitGetFieldPtr ( TextWriter & ss, const string & fieldName )
```

There are numerous `simulate...` routines that provide specific simulation nodes for different scenarios:

```
virtual SimNode * simulateDelete ( Context &, const LineInfo &, SimNode *, uint32_t )␣
↪const
virtual SimNode * simulateDeletePtr ( Context &, const LineInfo &, SimNode *, uint32_
↪t ) const
virtual SimNode * simulateCopy ( Context &, const LineInfo &, SimNode *, SimNode * )␣
↪const
virtual SimNode * simulateClone ( Context &, const LineInfo &, SimNode *, SimNode * )␣
↪const
virtual SimNode * simulateRef2Value ( Context &, const LineInfo &, SimNode * ) const
virtual SimNode * simulateGetNew ( Context &, const LineInfo & ) const
virtual SimNode * simulateGetAt ( Context &, const LineInfo &, const TypeDeclPtr &,
                                  const ExpressionPtr &, const ExpressionPtr &, uint32_
↪t ) const
virtual SimNode * simulateGetAtR2V ( Context &, const LineInfo &, const TypeDeclPtr &,
                                     const ExpressionPtr &, const ExpressionPtr &,␣
↪uint32_t ) const
virtual SimNode * simulateGetIterator ( Context &, const LineInfo &, const␣
↪ExpressionPtr & ) const
```

`walk` provides custom data walking functionality, to allow for inspection and binary serialization of the type:

```
virtual void walk ( DataWalker &, void * )
```

3.4.2 ManagedStructureAnnotation

`ManagedStructureAnnotation` is a helper type annotation template, designed to streamline the binding of a majority of C++ classes.

Lets review the following example:

```
struct Object {
    das::float3   pos;
    das::float3   vel;
    __forceinline float speed() { return sqrt(vel.x*vel.x + vel.y*vel.y + vel.z*vel.
↪z); }
};
```

To bind it, we inherit from `ManagedStructureAnnotation`, provide a name, and register fields and properties:

```
struct ObjectStructureTypeAnnotation : ManagedStructureAnnotation <Object> {
    ObjectStructureTypeAnnotation(ModuleLibrary & ml) : ManagedStructureAnnotation (
↪"Object",ml) {
        ...
```

`addField` and `addProperty` are used to add fields and properties accordingly. Fields are registered as ref values. Properties are registered with an offset of -1 and are returned by value:

```
ObjectStructureTypeAnnotation(ModuleLibrary & ml) : ManagedStructureAnnotation (
↪"Object",ml) {
    addField<DAS_BIND_MANAGED_FIELD(pos)>("position","pos");
    addField<DAS_BIND_MANAGED_FIELD(vel)>("velocity","vel");
    addProperty<DAS_BIND_MANAGED_PROP(speed)>("speed","speed");
```

Afterwards, we register a type factory and add type annotations to the module:

```
MAKE_TYPE_FACTORY(Object, Object)

addAnnotation(make_smart<ObjectStructureTypeAnnotation>(lib));
```

`addFieldEx` allows registering custom offsets or types:

```
addFieldEx ( "flags", "flags", offsetof(MakeFieldDecl, flags),␣
↪makeMakeFieldDeclFlags() );
```

That way, the field of one type can be registered as another type.

Managed structure annotation automatically implements `walk` for the exposed fields.

3.4.3 DummyTypeAnnotation

`DummyTypeAnnotation` is there when a type needs to be exposed to Daslang, but no contents or operations are allowed.

That way, the type can be part of other structures, and be passed to C++ functions which require it.

The dummy type annotation constructor takes a Daslang type name, C++ type name, its size, and alignment:

```
DummyTypeAnnotation(const string & name, const string & cppName, size_t sz, size_t al)
```

Since `TypeAnnotation` is a strong Daslang type, `DummyTypeAnnotation` allows 'gluing' code in Daslang without exposing the details of the C++ types. Consider the following example:

> send_unit_to(get_unit("Ally"), get_unit_pos(get_unit("Enemy")))

The result of `get_unit` is passed directly to `send_unit_to`, without Daslang knowing anything about the unit type (other than that it exists).

3.4.4 ManagedVectorAnnotation

`ManagedVectorAnnotation` is there to expose standard library vectors to Daslang.

For the most part, no integration is required, and vector annotations are automatically added to the modules, which register anything vector related in any form.

Vectors get registered together with the following 4 functions, similar to those of Daslang arrays:

```
push(vec, value)
pop(vec)
clear(vec)
resize(vec, newSize)
```

Vectors also expose the field `length` which returns current size of vector.

Managed vector annotation automatically implements `walk`, similar to Daslang arrays.

3.4.5 ManagedValueAnnotation

`ManagedValueAnnotation` is designed to expose C++ POD types, which are passed by value.

It expects type *cast* machinery to be implemented for that type.

3.5 Ahead of time compilation and C++ operation bindings

For optimal performance and seamless integration, Daslang is capable of ahead of time compilation, i.e. producing C++ files, which are semantically equivalent to simulated Daslang nodes.

The output C++ is designed to be to some extent human readable.

For the most part, Daslang produces AOT automatically, but some integration effort may be required for custom types. Plus, certain performance optimizations can be achieved with additional integration effort.

Daslang AOT integration is done on the AST expression tree level, and not on the simulation node level.

3.5.1 das_index

The `das_index` template is used to provide the implementation of the `ExprAt` and `ExprSafeAt` AST nodes.

Given the input type *VecT*, output result *TT*, and index type of int32_t, `das_index` needs to implement the following functions:

```
// regular index
    static __forceinline TT & at ( VecT & value, int32_t index, Context * __context__␣
↪);
    static __forceinline const TT & at ( const VecT & value, int32_t index, Context *␣
↪__context__ );
// safe index
    static __forceinline TT * safe_at ( VecT * value, int32_t index, Context * );
    static __forceinline const TT * safe_at ( const VecT * value, int32_t index,␣
↪Context * );
```

Note that sometimes more than one index type is possible. In that case, implementation for each index type is required.

Note how both const and not const versions are available. Additionally, const and non const versions of the `das_index` template itself may be required.

3.5.2 das_iterator

The `das_iterator` template is used to provide the for loop backend for the `ExprFor` sources.

Let's review the following example, which implements iteration over the range type:

```
template <>
struct das_iterator <const range> {
    __forceinline das_iterator(const range & r) : that(r) {}
    __forceinline bool first ( Context *, int32_t & i ) { i = that.from; return i!
↪=that.to; }
    __forceinline bool next  ( Context *, int32_t & i ) { i++; return i!=that.to; }
    __forceinline void close ( Context *, int32_t &   ) {}
    range that;
};
```

The `das_iterator` template needs to implement the constructor for the specified type, and also the `first`, `next`, and `close` functions, similar to that of the Iterator.

Both the const and regular versions of the `das_iterator` template are to be provided:

```
template <>
struct das_iterator <range> : das_iterator<const range> {
    __forceinline das_iterator(const range & r) : das_iterator<const range>(r) {}
};
```

Ref iterator return types are C++ pointers:

```
template <typename TT>
struct das_iterator<TArray<TT>> {
    __forceinline bool first(Context * __context__, TT * & i) {
```

Out of the box, `das_iterator` is implemented for all integrated types.

3.5.3 AOT template function

By default, AOT generated functions expect blocks to be passed as the C++ TBlock class (see *Blocks*). This creates significant performance overhead, which can be reduced by AOT template machinery.

Let's review the following example:

```
void peek_das_string(const string & str, const TBlock<void,TTemporary<const char *>> &
↪ block, Context * context) {
    vec4f args[1];
    args[0] = cast<const char *>::from(str.c_str());
    context->invoke(block, args, nullptr);
}
```

The overhead consists of type marshalling, as well as context block invocation. However, the following template can be called like this, instead:

```
template <typename TT>
void peek_das_string_T(const string & str, TT && block, Context *) {
    block((char *)str.c_str());
}
```

Here, the block is templated, and can be called without any marshalling whatsoever. To achieve this, the function registration in the module needs to be modified:

```
addExtern<DAS_BIND_FUN(peek_das_string)>(*this, lib, "peek",
    SideEffects::modifyExternal,"peek_das_string_T")->setAotTemplate();
```

3.5.4 AOT settings for individual functions

There are several function annotations which control how function AOT is generated.

The `[hybrid]` annotation indicates that a function is always called via the full Daslang interop ABI (slower), as oppose to a direct function call via C++ language construct (faster). Doing this removes the dependency between the two functions in the semantic hash, which in turn allows for replacing only one of the functions with the simulated version.

The `[no_aot]` annotation indicates that the AOT version of the function will not be generated. This is useful for working around AOT code-generation issues, as well as during builtin module development.

3.5.5 AOT prefix and suffix

Function or type trait expressions can have custom annotations to specify prefix and suffix text around the generated call. This may be necessary to completely replace the call itself, provide additional type conversions, or perform other customizations.

Let's review the following example:

```
struct ClassInfoMacro : TypeInfoMacro {
    ....
    virtual void aotPrefix ( TextWriter & ss, const ExpressionPtr & ) override {
        ss << "(void *)(&";
    }
    virtual void aotSuffix ( TextWriter & ss, const ExpressionPtr & ) override {
        ss << ")";
    }
```

Here, the class info macro converts the requested type information to *void **. This part of the class machinery allows the `__rtti` pointer of the class to remain void, without including RTTI everywhere the class is included.

3.5.6 AOT field prefix and suffix

`ExprField` is covered by the following functions in the handled type annotation (see *Handles*):

```
virtual void aotPreVisitGetField ( TextWriter &, const string & fieldName )
virtual void aotPreVisitGetFieldPtr ( TextWriter &, const string & fieldName )
virtual void aotVisitGetField ( TextWriter & ss, const string & fieldName )
virtual void aotVisitGetFieldPtr ( TextWriter & ss, const string & fieldName )
```

By default, prefix functions do nothing, and postfix functions append *.fieldName* and *->fieldName* accordingly.

Note that `ExprSafeField` is not covered yet, and will be implemented for AOT at some point.