University of Liège
Faculty of Applied Sciences

Sushi++ compiler
Project report [INFO2049]

Magera Floriane
1st master in computer engineering
Option : Computer systems and networks
s111295
Servais Fabrice
1st master in computer engineering
Option : Computer systems and networks
s111093
Mormont Romain
1st master in computer engineering
Option : Intelligent systems
s110940

Academic year 2014-2015

Table des matières

1 Introduction

In the context of the course INFO0085, we had to develop a compiler for a handmade programming language called **Sushi++**. This language is garbage-collected and halfway between a functional and an imperative language of which the keywords are inspired from the *sushi food* lexical field. The first part of the report describes both syntax and semantic of the language (section 2) and the second details the compiler organization and algorithms (section 3). The Section 4 presents the organization of the small runtime library handling the garbage collector. Finally, in the Section **??**, the improvements that could be made are discussed.

2 Language

2.1 General information

2.2 Type system

The *Sushi++* language is a strongly and statically typed language. Nevertheless, to alleviate the programmer's work, the type system is made as unobtrusive as possible and types are inferred during compilation. The only presence of types in the language is the *function parameters hinting* mechanism : the parameter type can be specified next to the parameter name in the a function declaration (see Section 2.3.1). The available types are :

- `bool` : *true* or *false*
- `char` : a character
- `int` : an integer $\in [-2147483648, 2147483647]$
- `float` : a single-precision floating point value $\in [-3.403 \times 10^{38}, 3.403 \times 10^{38}]$
- `array` : an array of elements (see Section 2.2.1)
- `list` : a list of elements (see Section 2.2.2)
- `string` : a string
- `function` : a function
- `void` : return type for function that doesn't return anything

2.2.1 Arrays

Idea The `array` type represents a sequence of elements having a given type T and that are stored sequentially in the memory. This structure has the same behaviour as a vector data structure in terms of complexity. The *Sushi++* arrays cannot stored any type of data, T can only be one among : `int`, `float`, `char`, `string`, `bool`. The index of an array is an integer in the interval $[0, \text{array size}]$. Arrays are passed to function and returned by reference (they are never copied).

Syntax To construct an array, its elements must be listed between the array delimiters `#[` and `]#` and separated by commas :

- array of integers : `#[ 1, 2, 3, 4 ]#`
- array of strings : `#[ "str1", "str2", "str3", "str4" ]#`
- assigning a variable : `maki a = #[ true, false ]#`

To access an element in the array, the C-like array-access operator `[]` can be used with a valid index.

```
maki first_element = array[0]
```

A set of built-in functions are provided to the programmer to handle arrays. They are listed as follows in a C-like format :

- `int array_size_T(array A)` : return the size of the array

- `void array_clear_T(array A)` : empty the array
- `void array_push_T(array A, T element)` : push an element at the last position of the array
- `T array_pop_T(array A)` : pop the element at the last position of the array
- `T array_get_T(array A, int i)` : return the element at the position `i` of the array
- `void array_set_T(array A, int i, T element)` : replace the element at the position `i` by `element`
- `void array_insert_T(array A, int i, T element)` : insert `element` at the position `i`

Memory Arrays are **heap-allocated** and their memory is managed at runtime with a garbage collector (see Section ??).

2.2.2 Lists

The construction of lists follows the same principles as the construction of array except that the delimiters are the curly brackets characters :

- array of integers : { `1, 2, 3, 4` }
- array of strings : { `"str1", "str2", "str3", "str4"` }
- assigning a variable : `maki a =` { `true, false` }

Lists *cannot* be accessed with the array-access operator. Nevertheless, a set of built-in functions are provided for interacting with them (inspired from standard C++ lists interface) :

- `int list_size(List l)` : return the size of the list
- `bool list_empty(List l)` : return true if the list is empty, false otherwise
- `void list_clear(List l)` : clear the content of the list
- `T list_front_T(List l)` : return the element at the front of the list
- `T list_back_T(List l)` : return the element at the back of the list
- `T list_pop_front_T(List l)` : remove and return the element at the front of the list
- `T list_pop_back_T(List l)` : remove and return the element at the back of the list
- `void list_push_front_T(List l)` : push an element at the front of the list
- `void list_push_back_T(List l)` : push an element at the back of the list
- `T list_get_T(List l, int pos)` : get the element at the given position of the list
- `void list_insert_T(List l, int pos, T elem)` : insert an element at the given position of the list
- `T list_remove_T(List l, int pos)` : remover the element at the given position of the list

Memory Lists are **heap-allocated** and their memory is managed at runtime with a garbage collector (see Section ??).

2.3 Declarations

The *Sushi++* language provides declaration syntax for both functions and variables. The declaration keyword `maki` is common to both declaration. The declared structured can be used using an identifier. The *Sushi++* identifiers must match the following regular expression " `[a-zA-Z]([-0-9A-Za-z_]*[0-9A-Za-z_])*`".

2.3.1 Functions

Idea The function declared with the `maki` keyword are called **named function** in opposition to anonymous function (or *soy functions*, see Section 2.4.1). As soon as it is declared, a function can be either called (see Section 2.4.2) or used as expression (passed as parameter). A declared named function is accessible in the scope in which it is declared and its nested scopes (including the function own scope).

Syntax The named function declaration syntax is the following :

`maki` *function_name* {*parameter_name*[`<` *type* `>`]} `:` *function_body* `;;`

The *type* element must be one of the types listed in the Section 2.2 except `void`.

2.3.2 Variables

Idea A variable must be assigned a value when it is declared. It is accessible in the scope in which it is declared and its nested scopes but **cannot be captured** in a function. A variable can only by reassigned a value having the same type as the one initially assigned.

Syntax Several variables can be declared with a single `maki`, the different declarations must be separated by commas :

`maki` *variable_name* = expression {, *variable_name* = expression }

2.4 Expressions

2.4.1 Soy functions

Idea Soy functions or anonymous functions are functions that are not bound to an identifier. They can either be used as value (passed as parameters), be called (see Section 2.4.2) or stored into a variable.

Syntax A soy function is declared by using the `soy` keyword :

`(soy` {*parameter*[`<`*type*`>`]} `:` *function_body* `)`

Various usages :

- Storing an anonymous function in a variable : `maki f = (soy x : nori x)`
- Passing an anonymous function as parameter : `call func (soy x : nori x) a`
- Calling an anonymous function : `call (soy x : nori x) 1`

2.4.2 Function calls

Idea A function call is triggered with the `call` keyword followed by a function name or an expression that can be evaluated as a function. The keyword `call` is meant to prevent the ambiguity between a function call and a variable utilisation. A function call must be braced when its arguments are placed on several lines or if it is embedded into another expression.

Syntax The syntax for calling a function is the following :

`call` (*function_name*|soy_expression) { argument }

Various usages :

- Calling a function : `call func param1 param2`
- Calling an anonymous function : `call (soy x : nori x) 1`
- Embedded call : `a = (call func c d)`

2.4.3 Operators

The *Sushi++* language provides a set of operators for expressing operations on flat types. These operators are listed in the Table 1 which describes their properties. The precedence and associativity was inspired from the C language. Therefore, in general, operators are left associative except for the assignment[1] and exponentiation. The precedence number given in the table decreases with the priority of the operator.

1. so that they can be chained

Op.	**Arity**	**Assoc.**	**Prec.**	**Comment**	**Operand types**	
					Operand 1	Operand 2
`op ++`	1	/	1	Postfix increment	{`int`, `float`}	
`op --`	1	/	1	Postfix decrement	{`int`, `float`}	
`++ op`	1	/	2	Prefix increment	{`int`, `float`}	
`-- op`	1	/	2	Prefix decrement	{`int`, `float`}	
`op1 ** op2`	2	right	3	Exponent	{`int`, `float`}	`int`
`op1 . op2`	2	right	3	String concatenation	`string`	`string`
`- op`	1	/	4	Unary minus	{`int`, `float`}	
`~ op`	1	/	5	Bitwise not	`int`	
`! op`	1	/	5	Logical not	`bool`	
`op1 * op2`	2	left	6	Mutliplication	{`int`, `float`}	same as `op1`
`op1 / op2`	2	left	6	Division	{`int`, `float`}	same as `op1`
`op1 % op2`	2	left	6	Modulo	`int`	`int`
`op1 + op2`	2	left	7	Addition	{`int`, `float`}	same as `op1`
`op1 - op2`	2	left	7	Substraction	{`int`, `float`}	same as `op1`
`op1 >> op2`	2	left	8	Right shift	`int`	`int`
`op1 << op2`	2	left	8	Left shift	`int`	`int`
`op1 < op2`	2	left	9	Greater than	{`int`, `float`}	same as `op1`
`op1 > op2`	2	left	9	Equal to	{`int`, `float`}	same as `op1`
`op1 <= op2`	2	left	9	Less or equal to	{`int`, `float`}	same as `op1`
`op1 >= op2`	2	left	9	Greater or equal to	{`int`, `float`}	same as `op1`
`op1 != op2`	2	left	10	Not equal to	{`int`, `float`, `bool`}	same as `op1`
`op1 == op2`	2	left	10	Equal to	{`int`, `float`, `bool`}	same as `op1`
`op1 & op2`	2	left	11	Bitwise and	`int`	`int`
`op1 ^ op2`	2	left	12	Bitwise xor	`int`	`int`
`op1 | op2`	2	left	13	Bitwise or	`int`	`int`
`op1 && op2`	2	left	14	Logical and	`bool`	`bool`
`op1 || op2`	2	left	15	Logical or	`bool`	`bool`
`op1 += op2`	2	right	16	Addition assignment	{`int`, `float`}	same as `op1`
`op1 -= op2`	2	right	16	Substraction assignment	{`int`, `float`}	same as `op1`
`op1 *= op2`	2	right	16	Mutliplication assignment	{`int`, `float`}	same as `op1`
`op1 /= op2`	2	right	16	Division assignment	{`int`, `float`}	same as `op1`
`op1 %= op2`	2	right	16	Modulo assignment	`int`	`int`
`op1 **= op2`	2	right	16	Exponent assignment	{`int`, `float`}	`int`
`op1 .= op2`	2	right	16	Concatenation assignment	`string`	`string`

TABLE 1 – Sushi++ operators

2.5 Statements

2.5.1 Loops

The *Sushi++* language provides three kinds of loop.

Condition-controlled loop This loop is called `roll` and behaves like a C while loop. The scope containing this code fragment is delimited by the end of line following the condition and the scope delimiter `;;`. The condition must be an expression of type `bool`, otherwise the compiler will return a type error. . The syntax is the following :

```
roll condition
    roll scope
;;
```

Counter-controlled loop This loop is called `for` and behaves like a C for loop. Again, the condition must be an expression of type `bool`. Moreover, the initializer and update part must contain a modifying expressions (assignment or increment). The syntax is the following :

```
for init expression, condition, update expression
    for scope
;;
```

Collection-controlled loop This loop is called `foreach` and is made for iterating over lists. At the first iteration, the element at the front of the list is stored in the iteration variable. Then, at each iteration, the same goes for next value till the list was completely covered. The syntax is the following :

```
foreach list expression as variable
    foreach scope
;;
```

2.5.2 Conditionals

2.5.3 Switch

3 Compiler

3.1 General information

The *Sushi++* compiler is implemented in standard C++ (version c++11). It is organized around the `SppCompiler` class which coordinates the various phases. The execution of the compiler can be tuned with various parameters (verbosity, optimization, input file path, output file path,...). The parameters parsing and analysis is encapsulated into the `CompilerSettings` object which provides a set of functions for knowing the values of the parameters.
The compiler follows the classical architecture :

- Lexical and syntax analysis
- Semantic analysis
- Intermediate code generation
- Optimization and executable generation

The first phase uses `flex` and `bison` to generate an abstract syntax tree as an `ast::AbstractSyntaxTree` object. The tree nodes classes are defined in the folder `ast/nodes` and are all subclasses of the `ast::ASTNode` class. While the base class implements a generic tree structure by storing pointers to parent and children nodes, the derived classes contains accessors to children defining the actual abstract syntax tree structure. To perform operations on the tree during the second and third phases, the visitor pattern was implemented. The visitors base class is `visitor::ASTVisitor` in the folder `ast/visitor` and every pass on the tree is implemented in its own visitor class :

1. Function table populating and termination checking (`visitor::FunctionTableVisitor`)
2. Variable table populating (`visitor::VariableTableVisitor`)
3. Type inference (`visitor::TypeInferenceVisitor`)
4. Code generation (`visitor::CodeGenVisitor`)

As the initial visitor class (`visitor::PrintASTVisitor`) was designed parameter-free and that refactoring it would have taken too much time. Passing parameters is handled with the `visitor::VisitorParameters` class.
The errors formatting and storing is encapsulated by the class `ErrorHandler` to which are passed the various errors generated during a phase.

3.2 Lexical and syntax analysis

At the beginning of the project, the choice was made to use `bison` and `flex` (versions `3.0.2` and `2.5.35` respectively) for the syntax and lexical analysis phases. This was motivated by the fact that many projects (including the integrated project) were ongoing during the development of this part of the compiler. So we decided to use existing tools to reduce the workload so that we could deliver a function syntax and lexical analysis for the first deadline. Among the existing tools, we have decided to use `bison` and `flex` because our compiler is developed in C++ and that integration would be easier.

The parser is implemented in the bison file `parser/spp_parser.y` and the lexer in `scanner/spp_scanner.lex`. Here are the elements that are worth to be mentioned :

- the tree is built bottom up (as bison implements a bottom-up parser). Every time a rule is matched, the corresponding node is allocated and constructed. At construction, both its children and some location information (as `ast::NodeLocation` objects) are given to the node through the constructor.
- at the starting rule of the grammar, the abstract syntax tree object is allocated, his root is set with the node at the top of the built tree and is given to the `SppCompiler` object.
- adding errors to the error handler is done in the `yyerror` error function so that the built-in error reporting of bison can be used
- the operator precedence and associativity are handled with the dedicated bison constructs (see Table 1)
- the semantic type of the nodes is unfortunately `void*` (see Section 5.1)
- as the grammar is accepted by bison without conflicts **it is** LALR(1) and therefore **LR(1)**
- in the lexer file, a catch-all rule was added to generate the lexical errors when an unknown sequence is found

3.3 Semantic analysis

3.3.1 Scope checking

In order to perform scope checking, we implemented two visitors. The first one aim's is mainly to discover function's declarations and is named `FunctionTableVisitor`, while the second one `SymbolTableVisitor` will be the one to detect most semantic errors. The two visitors visit the abstract syntax tree and fill in the two symbol tables we maintain. There is one for the functions and one for the variables. The first one is filled by the first visitor and the second one by the second visitor. A little flaw is that the symbol table destined for variables may contain functions too. Sometimes we can only distinguish between a variable and a function at the inference step (in the case of an function passed as an argument for example).

The `FunctionTableVisitor` is also used for termination checking, as will be explained in the next paragraph. There was a need to visit two times the tree because a function may be called before its definition. So at this level, the visitor adds informations about the functions defined in the code in the part of the symbol table corresponding to the scope containing the function's declaration. The information yielded is the name, the location and the argument's name of the function. Note that the arguments are also added in the variable symbol table in the scope corresponding to the function body.

The `SymbolTableVisitor` fills the variable symbol table and checks that each variable was well declared and that each function call corresponds to a function repertoried by the `FunctionTableVisitor`. We decided to forbid closures.

3.3.2 Termination checking

The termination checking is implemented in the `FunctionTableVisitor`. We made it as simple and light as possible : we implemented it with booleans. In our language, all the instructions that may contain a return are represented by the non-terminal token Statement. Once we know this, in order to check termination, we only need to test whether a function's scope does contain a statement that terminates in any case. So the loops, the break and the continue instruction does not fulfill this criteria. The conditional statement may return in any case only if there is a default case (else), idem for the menu statement. At the scope checking step, we don't know the types of return, so the only distinction we make is void or non void function. A void function may not contain a return in any case and still may be correct, so in that case, there is nothing to check. We added one more restriction which is a function must either return nothing or must return something in every case.

About the implementation : generally in a function we test the presence of a return statement (empty or not), so there are two booleans for this purpose. Then locally, in each statement, we will test if there is a local returning statement (so if there is an else if statement, we test if there is a return statement in its scope). When entering a new scope, we first suppose that the scope will return in any case, then if inside the scope there is one statement non returning in any case, then we know that the scope won't return in any case. In the same way if there is a returning statement followed by code, we know that this code will never be executed. We trigger an error for dead code. At the end of the exploration of the sons of the function declaration, we will trigger an error if there is a non empty return in the function, and if the function does not return something in every case.

3.3.3 Type checking and inference

As the *Sushi++* language is statically typed, the types must be checked at compile-time. This consists in checking the following properties :

- operands of an operators have valid types according to the Table 1.
- parameters of a function have the expected types
- function always returns an element of the same type, or it always returns nothing
- expressions used in statements have have a valid type (boolean for conditions or loop guardians for instance)
- variable can only be reassigned a value of the same type as the one initially assigned
- ...

As *Sushi++* is free from type annotation, these checks cannot be performed directly and the types of expressions and identifiers have to be inferred.

Formally, the type inference problem can be as formulated as follows : given a set of types $\mathcal{T}$ defined for the programming language, a set of expressions $\mathcal{E}$ and a set of identifiers $\mathcal{I}$ defined in a program, inferring the types consists in assigning a type $t \in \mathcal{T}$ for any identifier $i \in \mathcal{I}$ and expression $e \in \mathcal{E}$ used in the program (if the expression has no effect or if the identifier is not used in the program than knowing his type is not relevant). In the literature, a classical approach for assigning types is the Hindley-Milner (or Damas-Milner) algorithm. This algorithm assigns types to language constructs using a set of deduction rules and derivations. Unfortunately, this algorithm was designed for purely functional languages and is not directly applicable to the *Sushi++* language.

Alternatively, the type inference can be seen as problem of **constraints generation and unification**. The types associated with the various program elements are represented by variables. Some constraints bringing in these variables are generated from some rules encoding the language type semantic and these constrains are unified to find the actual type of the program elements. Two sub-problems have therefore to be addressed to design the type inference algorithm : on the one hand, an unification algorithm has to be defined. On the other hand, for each *Sushi++* construct, the type semantic has to be encoded into a rule defining the constraints to be generated.

Framework The algorithm associates a type variable, symbolized by a greek letters, to every language typed construct (including function parameters, return type and data structure elements' type). A variable is either resolved, meaning that it is associated with a valid type, or unresolved. A valid type is either a flat type (integer, float, char, string, bool or void) or a structured type of which the type parameters are valid types (array, list or function). A variable is also associated a set of types that it can be assigned and such a set is called *hints* (the hints of a variable α is noted $\mathcal{H}_\alpha$). The hint system is a way to encode operator polymorphism and function parameter hinting. An unresolved type variable α can be resolved if $\mathcal{H}_\alpha$ contains only one flat type.

Unification The goal of the unification is to "*find a substitution for all type variables that make the expressions identical*" (taken from document [2a], slide 8). The algorithm used in the compiler is given in Listing 1 and uses the following function :

- `is_unresolved(`α`)` : returns false if the variable is resolved, true otherwise
- `is_function(`α`)` : returns true if the variable contains a function type, false otherwise
- `count_parameters(`α`)` : given a variable containing a function type, return the number of parameters of this function
- `get_return_type(`α`)` : given a variable containing a function type, return the type variable containing the return type of this function
- `is_array(`α`)` : return true if the variable contains an array type, false otherwise
- `is_list(`α`)` : return true if the variable contains an list type, false otherwise
- `get_datastructure_type(`α`)` : given a variable containing an array or a list type, return the type variable containing the type of its elements

The worst-case complexity of the unification algorithm is $\Theta(n)$ where n is the number of parameters which intervenes in the types contained in α and β. This can be an issue when α and β both contains a function of which one of the parameters is a function taking itself as argument a function taking itself as argument a function, etc. Nonetheless, in the other cases, n often equals to 1 or 2.

```
// Unification algorithm
unify ( $\alpha$, $\beta$ )
{
  if is_unresolved($\alpha$) || is_unresolved($\beta$)
  {
    if $\mathcal{H}_\alpha \cap \mathcal{H}_\beta = \emptyset$ // check hints compatiblity
      throw error("incompatible hints")

    // update hints
    $\mathcal{H}_\beta$ = $\mathcal{H}_\alpha$ = $\mathcal{H}_\alpha \cap \mathcal{H}_\beta$
    // add indirection between the variables
    if is_unresolved($\alpha$) { $\alpha$ = $\beta$ } else { $\beta$ = $\alpha$ }

    return
  }
  // both variables are resolved : the valid types must be compatibles
  if is_function($\alpha$) && is_function($\beta$) // variables are functions
  {
    if count_paramters($\alpha$) != count_parameters($\beta$)
      throw error("function types should have the same number of parameters")

    for each parameters types variables $\gamma$ of function $\alpha$ and $\delta$ of function $\beta$
      unify($\gamma$, $\delta$)

    unify(get_return_type($\alpha$), get_return_type($\beta$))

    return
  }

  // variables are both uniparameter types
  if ( is_array($\alpha$) && is_array($\beta$) ) || ( is_list($\alpha$) && is_list($\beta$) )
  {
    $\gamma$ = get_datastructure_type($\alpha$)
    $\delta$ = get_datastructure_type($\beta$)
    unify($\gamma$, $\delta$)
  }

  // variables contains the same flat type
  if is_flat($\alpha$) && is_flat($\beta$) && $\alpha$ = $\beta$
    return

  throw error("types cannot be unified")
}
```

Listing 1 – Unification algorithm

Constraints generation The constraints generation can be said to be syntax-directed. A set of type variables are passed to a node of the abstract syntax tree as inherited attributes and some others are created by the node itself. The first set contains the information about the types expected by the parent while the second contains locale type information. Then, the node performs unification on these variables to signal the types it is expecting. It can also create new variables to pass to its children as inherited attributes. If the unification fails, then a type error must be signaled by the compiler. If the whole abstract syntax tree is traversed without type error, the program can still be invalid in terms of typing. Indeed, a program is only valid if every *useful* typed construct is assigned a type, yet some variables could still be unresolved. In this case, the compiler should signal a lack of expressiveness in the input program.
As the visitor performs only one pass on every node, running through the tree is linear with its size. Therefore, a large upper bound on the type inference algorithm time complexity is $\Theta(N * n)$ where N is the number of nodes of the tree and n is the number of sub-parameters of the largest function type to be handled during the parsing. This complexity corresponds to the case when each node has to unify the largest function type. Yet, as it is impossible that every node requires to perform this unification and that the unification will often involve "*small*" types, the type inference algorithm is almost linear on average.

Implementation The type inference is implemented in the `ast/visitor/TypeInferenceVisitor.cpp` file and uses the classes defined in the `inference/` folder. The main idea is to store all variables and their associated type into a **type symbol table**. This table is implemented as a structure mapping a variable name to an object representing the type associated with this variable (of class `TypeSymbol`). The creation of variables consists in adding entries in this table and unification in updating it.
A key element of the table design is that, if two variables were unified, updating the type of one of them must be automatically reflected in the other. To achieve this, the elements mapped in the type symbol table are `TypeLink` objects introducing a level of indirection between a variable name and its actual type. The only attribute of a TypeLink is simply a pointer to another TypeSymbol. The class hierarchy of type symbols is described hereafter.

Type symbols The types symbols are objects designed to represents types. The base class `TypeSymbol` is derived into the classes `TypeLink` and `TerminalTypeSymbol` which represents respectively a link to a type symbol and an element that can be located at the end of a type link objects chain. The latter is then derived into the classes `TypeVariable` which represents a type variable (or an unresolved type) and `Type` which is the base type of any actual type class :

- `FlatType` : base class for any flat type (`Int`, `Bool`, `Float`, `String`, `Char` and `Void`).
- `UniparameterType` : base class for type having only one type parameter (`Array` and `List`). This class holds the type parameter which is represented by a type link object.
- `Function` : class for function types. This class stores a vector of `TypeLink` objects to represent the type of the parameters and another one for the function return type.

The `ShallowType` enumeration represents the possible types as integers. The hint sets are represented by `TypeHints` objects (class defined in `inference/Types.hpp`) which represents the set as a bitmask of `ShallowTypes` allowing efficient operations (intersection, union, removal and checking if a ShallowType belongs to it). Every `TerminalTypeSymbol` is assigned a hint set object.

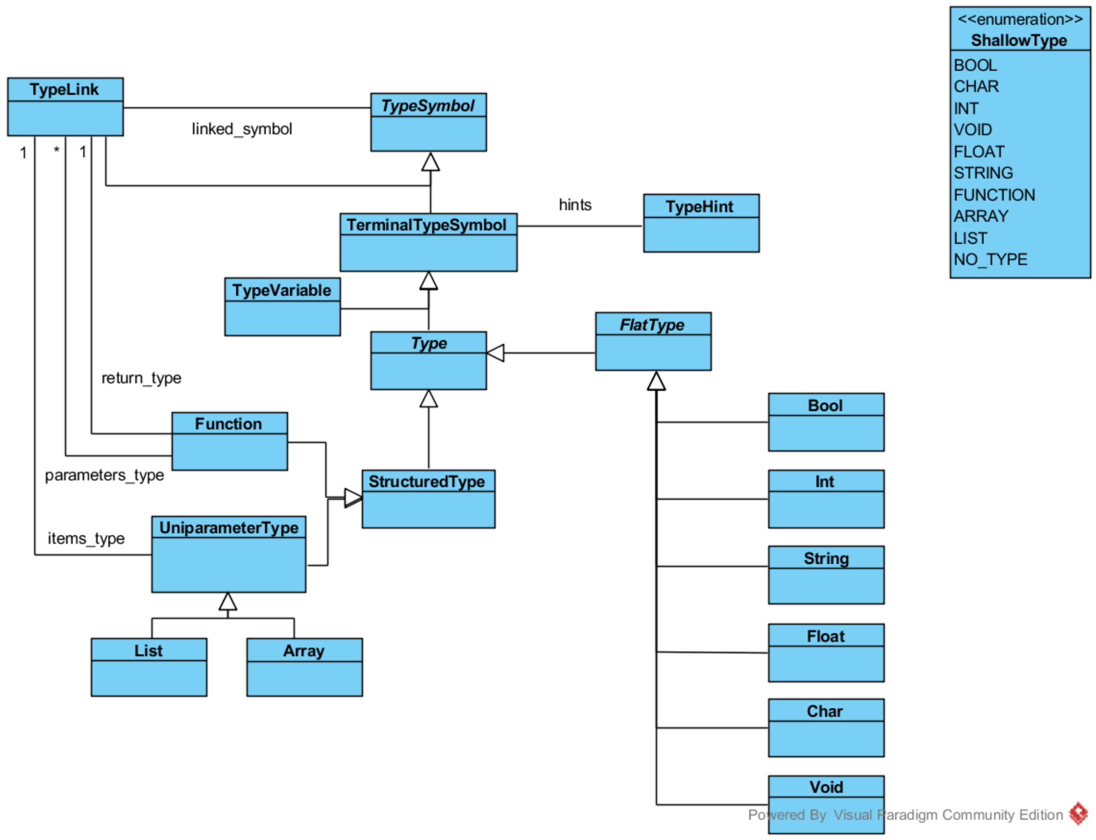

FIGURE 1 – Type symbols class diagram

Type symbol table The type symbol table inherits from an instantiation of the C++ standard hashtable template class, `std::unordered_map<std::string,TypeLink>`, and encapsulates the following operations :

- unification between two variables
- unification between a variable and a flat type
- update of the hints of a type variable
- creation of new variables
- construction of type symbol objects for variables, functions, arrays and lists
- construction of a type object for the code generation phase
- generation of unique type variable names

The table stores two kind of variables : the ones bounded to the program's identifiers (so-called *bounded* type variables) and the ones that are not (so-called *pure* type variables). As every variable has to be uniquely identified, the generation of variable names have to be done carefully :

- pure variable names are numbers generated with a counter and is therefore collision free as long as the counter does not overflow
- as a in a valid program, two identifiers having the same name cannot be declared in the same scope, a unique name for bounded variables can be constructed from these two elements. An additional character, '@' is added between the identifier name and the scope id for the sake of readability. The name of a bounded variable is therefore generated as follows :

  `identifier@scope_id`

Syntax-directed type inference As every other semantic analysis pass, the type inference is implemented with another visitor in `TypeInferenceVisitor.cpp`. Each node corresponds to a language construct of which the type semantic must be translated in the type symbol table. Moreover, a node can receive from its parent and pass to his children a set of variable (for instance, if an expression is expected to return a value of a given type, the corresponding type variable could be passed from the parent to the expression node). Therefore, designing the type inference for a node results in answering five questions :

1. **Inheritance** : which type variables are inherited from the parent node ?
2. **Table insertion** : which variables must be created in the type symbol table and with which terminal type symbols have they to be associated ?
3. **Unification** : which pairs of variables have to be unified ? Does a variable have to be unified with a flat type ?
4. **Hinting** : has the hints of a variable to be updated ?
5. **Transmission** : which variables must be given to the children nodes ?

As these elements must be defined for every node, the complete list is not detailed here. Nevertheless, three examples are given : the *+* operator, the *if* statement and the function declaration.

- **Operator +** : this node has two children, the two operands
  - *Inheritance* : α is the type that should be returned by the operation
  - *Table insertion* : no type variable insertion
  - *Unification* : no unification
  - *Hinting* : as '+' can only take integer or float operands, we have : $\mathcal{H}_\alpha = \mathcal{H}_\alpha \cap \{\texttt{int}, \texttt{float}\}$
  - *Transmission* : as the type returned by the addition is the same as the type of the operands α is passed the both of them
- ***Statement if*** : this node has two children, the condition expression and the scope
  - *Inheritance* : α is the type that should be returned by a `nori` statement in the *if* scope
  - *Table insertion* : β is the type returned by the condition expression

  — *Unification* : as the condition expression must return a boolean value, the following unification must be performed : $unify(\beta, \texttt{bool})$
  — *Hinting* : no hinting
  — *Transmission* : β is passed to the condition expression node and α to the scope node
- **Function declaration** : `maki func_name` $\texttt{p}_1$ `...` $\texttt{p}_n$ `: scope ;;` (for instance defined in the scope 1 and its scope having id 2)
  — *Inheritance* : no inheritance
  — *Table insertion* : *func_name@1* is the type of the function, p_i*@2* is the type of the i^{th} parameter and α is the return type
  — *Unification* : the function name must be unified with the function type :

$$unify(\textit{func\_name@1}, p_1\textit{@2} \ldots p_n\textit{@2} \rightarrow \gamma)$$

  — *Hinting* : each hinted parameter should be hinted. If the hint is a flat type, an unification can be performed instead
  — *Transmission* : γ to the scope node

The *almost* complete list of the nodes and their inference actions is given in the `Inference.pdf` document attached to the submitted archive.

Error handling When an error occurs, the unification errors are added to the error handler with a message indicating the type causing the error, and the one that was expected. If an error occurred during type inference, the next stages are bypassed. As said above, a program passing the type inference without error might not be a valid program, nevertheless an explicit check of this possible problem wasn't implemented but will trigger an exception in the intermediate code generation phase.

3.4 Code generation

The code generation part is done manually, we do not use the LLVM library due to the lack of documentation and its difficulty. Nevertheless, it was really interesting to handle this part.

It is divided into two main modules :

- A visitor going through the nodes of the AST,
- A builder aggregating the lines of LLVM code.

3.4.1 Code construction

An LLVM code for a program is structured that way : a file is represented as a `Module`, which contains functions, global variables and external declarations. Functions are represented as `FunctionBlock`, having a name, a return type and arguments. In a function, one also have blocks beginning with a label, these are `BasicBlock`. This class contains all the methods to generate a specific line or operation in the code.

In order to facilitate and encapsulate the data needed for these methods, one creates a type which is transmitted in all of these : `Value`. From this class inherits types : `codegen::Function`, `codegen::Variable` and `codegen::Constant`. They need to implement the `str_value()` abstract method, which has as purpose to return the string value corresponding to the class, for instance a `Variable` should return the concatenation of '%' with its name, a `Constant` its value and a `Function` its signature (although this is not used in practice). A simplified class diagram is shown in FIGURE 2.

The main class that handles the creation and aggregation of the code is `Builder`. It contains a vector of `Modules`. In our case, there is only one module possible, the handling of multiple files (hence modules) is an improvement.

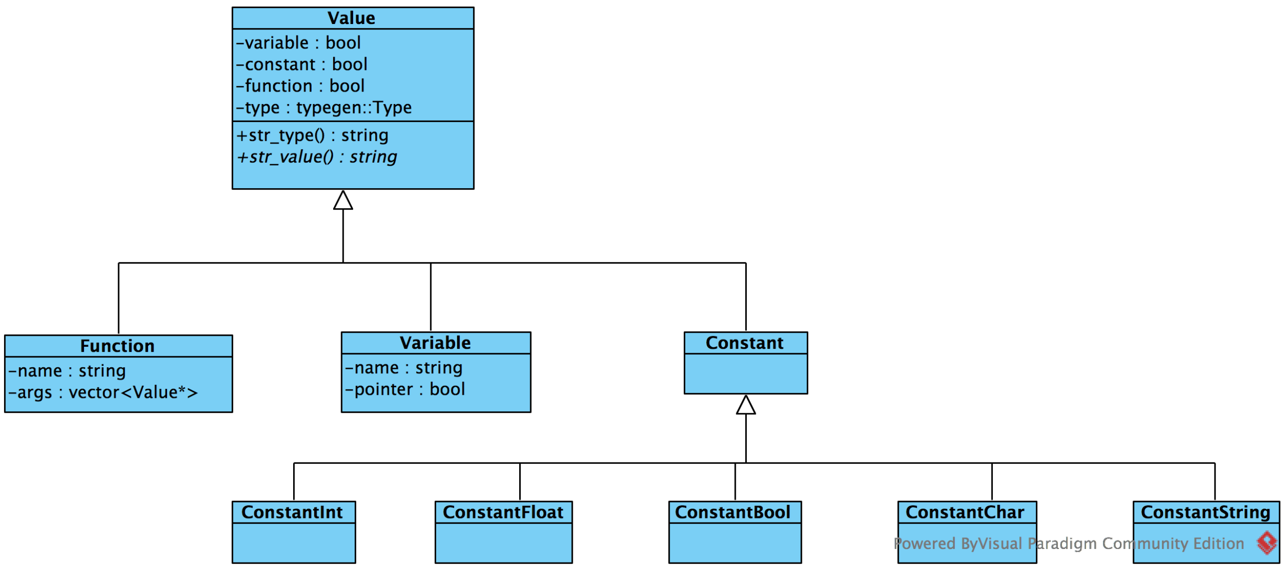

FIGURE 2 – Simplified diagram of code generation types

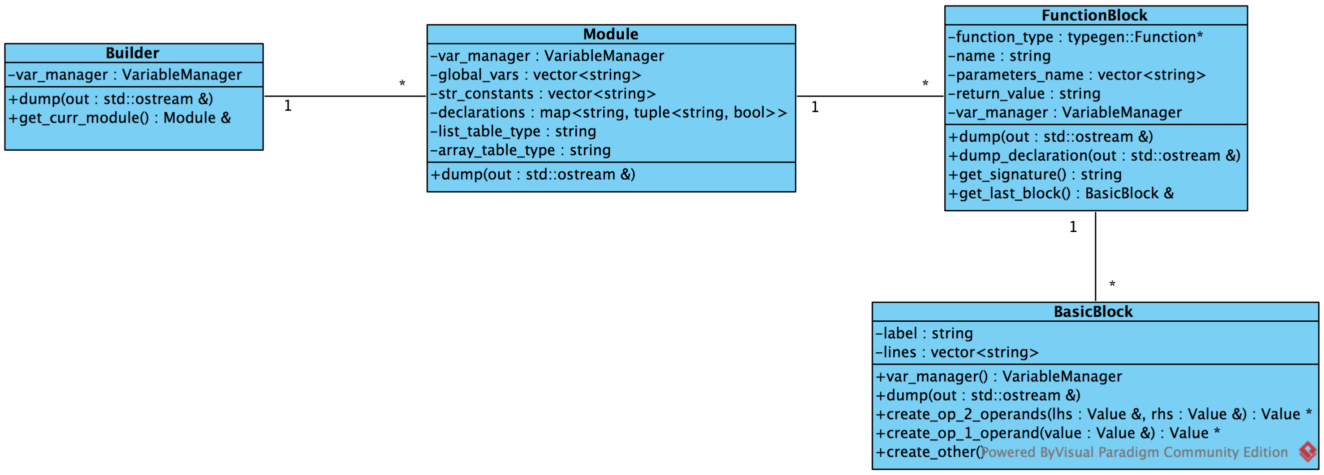
Builder
-var_manager : VariableManager
+dump(out : std::ostream &)
+get_curr_module() : Module &
1
*
Module
-var_manager : VariableManager
-global_vars : vector<string>
-str_constants : vector<string>
-declarations : map<string, tuple<string, bool>>
-list_table_type : string
-array_table_type : string
+dump(out : std::ostream &)
1
*
FunctionBlock
-function_type : typegen::Function*
-name : string
-parameters_name : vector<string>
-return_value : string
-var_manager : VariableManager
+dump(out : std::ostream &)
+dump_declaration(out : std::ostream &)
+get_signature() : string
+get_last_block() : BasicBlock &
1
*
BasicBlock
-label : string
-lines : vector<string>
+var_manager() : VariableManager
+dump(out : std::ostream &)
+create_op_2_operands(lhs : Value &, rhs : Value &) : Value *
+create_op_1_operand(value : Value &) : Value *
+create_other()
Powered ByVisual Paradigm Community Edition

FIGURE 3 – Simplified diagram of the `Builder` architecture

Another noticeable module is the {`Label`,`Variable`}`Manager`. It has to handle the name of the labels (or variables). Indeed, since we are in a SSA form, variables cannot be re-assigned. In a similar way, two labels cannot have the same name. The design is pretty simple, it is only a mapping of the name of the variable and a number which is basically the number of time the variable had been assigned. Thanks to that, one can easily retrieve a new name of variable with the same prefix, for instance, if "var_name" has already been used, `string insert_variable(string)` will give the next available name with the prefix : "var_name.1".

A simplified diagram is shown in FIGURE 3.

3.4.2 Code generation

After having handled how to write lines of LLVM, we can go to the part where we manipulate this tool to generate code depending on the source code.

The generation is done through a visitor on the AST : `CodeGenVisitor`. It contains an instance of the `Builder`, as well as some kind of cursor to where it should add lines next, so which function of which module. It also contains a reference to the symbol tables and the type table, in addition to information for the built-in fonctions. To handle the passing of `Value`'s though the nodes (when it is visited bottom-up), one uses a common stack where the returned values are pushed.

At the construction, the object initializes some global variables, types and functions related to the arrays and lists from the runtime. It also adds a `main` function.

The values we are manipulating are all stored in memory, we are thus managing pointers in LLVM. The values are then loaded from these pointers, or are the results of functions. Typically, in this case, the value of a variable is loaded from it, given to the function, and its result is stored in a new allocated zone in memory. This pointer is given to the node's parent and is responsible to load the value again and store it in the correct place.

Operators : The sons of the node are visited, putting either a `Variable` or a `Constant` on the stack. If it is a `Variable`, the value is loaded in a temporary LLVM variable. The values are given to a `create` method of the `Builder` (more precisely, of the current `BasicBlock`). The result is then stored in a new location in memory. This result is also pushed on the stack.

Assignment : It basically does a `load` and a `store` just after. In the case of an array or a list, the counter of references to it is updated.

Constant token : It only creates a `Constant` of the correct type.

Datastructure non-terminal : In this case, one has to use the functions of the runtime. These are hardcoded in the block.

Declaration non-terminal :

- Function : A new `FunctionBlock` is created, containing the name, the return value and the arguments of the function, optained by visitig the sons of the node. One also has to set the current function block of the visitor to it, hence, the following lines to be written will be in that function.
- Variable : First visit the expression and the identifier. It will then create a pointer where the value of the expression is stored. In the case of an array or a list, the reference counter is also updated.

Expression non-terminal : It basically only visits the children of the node. In the case of a datastructure access, it has to use de built-in functions, to be then stored in memory.

FunctionCall non-terminal : It first visits its children to retrieve the identifier and the parameters. It also retrieves the return type from the type table. For the arguments, one has to load the values from the pointers. One can afterwards store the result of the call of the function in memory, if the function return type is not void.

Program non-terminal : The main node is Scope, it has to handle the change of scope id. Also, at the end of the scope, the arrays and the lists allocated has to be freed. One has thus to go through the `RemoveReferenceFlags` to see which datastructure has to be freed, and generate the associated lines.

Statement non-terminal :

- Return : The reasoning is the same as before, where it simply loads the needed values and call the corresponding method in the block to create a return.
- Conditional/If/Elesif/Else : It first appends the condition of the if to the current block and creates a conditional branch depending on the value of the condition, if the result is true, branch on the block with hte prefix "if_true", and on "if_false" otherwise. The block "if_true" is first created, fulfilled, add a branch to the "end_if" block and then the "if_false" block is inserted. If there is any Elseif or Else condition, they are inserted in the same way. Finally, the "end_if" block is inserted.

3.5 Optimization and machine code generation

The last compile phase is implemented using the C standard `system` function which "*invokes the command processor to execute a command*" (taken from [3a]). Through this function, calls to LLVM utilities are done to optimize the intermediate representation and to generate the executable.

3.5.1 Optimization

The LLVM optimizer `opt` command is used. The optimization performed are the following :

- `mem2reg` : try to use registers instead of memory (as our generation puts everything into memory, this pass is very useful)
- `tailcallelim` : try to perform tail call elimination (transform tail call recursive functions into iterative functions)
- `inline` : try to inline functions from callee to caller (bottom-up)
- `constprop` : try to propagates and merges constants intervening in the program
- `dce` : detect and try to get rid of the dead code blocks (sometimes generated by the compiler, or by the optimizer)

By default, the compiler doesn't optimize the generated code. The flag `-t` must be given to enable these optimizations.

3.5.2 Machine code generation

The compiler chains the following commands :

- `clang` : to generate, from source, the LLVM code of the runtime libraries (`.c` → `.ll`)
- `llvm-as` : to compile human readable LLVM to bitcode assembler (`.ll` → `.bc`)
- `llc` : to compile bitcode assembler to machine assembler (`.bc` → `.s`)
- `gcc` : to generate the executable from the `.s` files

The commands `clang`, `llvm-as` and `llc` are extracted from the LLVM `3.5` bundle and the version of `gcc` is `4.8.2`. Yet, nothing fuzzy is done with these commands and using a more recent version should also work (even with older versions for the LLVM commands). To clean remove the useless generated files, the unix commands `mv` and `rm` are also used.

4 Runtime

Initially, the runtime was supposed to provide support for :

- **memory management** for arrays and lists (i.e. a garbage collector)
- **closures** : to be able to capture variables in functions, a runtime data structure is required to keep a mapping between variables and their addresses
- **basic functionalities** : IO operations and data structure handling

The *closures* functionality was dropped due to a lack of time but the others were successfully implemented. Three runtime libraries were implemented :

- `array_runtime.h` : interface to the array management runtime
- `list_runtime.h` : interface to the list management runtime
- `support.h` : interface providing basic operations such as IO (print)

The list and array runtimes are implemented using the same principle. A object of type `struct array_table` (reps. `struct list_table`) contains all the allocated arrays (resp. lists) which are identified by a unique number. A set of operations are then provided by the interfaces to interact with an array (resp. a list) given the table object and the array id (resp. list id). Some of these operations are exposed to the *Sushi++* programmer while others (such as reference counting, allocation,...) aren't.
The memory is therefore handled by reference counting through the functions `array_add_reference` and

`array_rm_reference` (resp. `list_add_reference` and `list_rm_reference`) that must be called in the LLVM generated program to signal that a data structure is referenced by a new variable or it is not referenced anymore. When the counter drops to 0, the object is automatically freed.
The underlying implementation of the `struct array_table` (resp. `struct list_table`), stores the arrays (resp. lists) in a linked list which is not optimal but discussed in the Section

5 Limitations and possible improvements

5.1 Lexical and syntax analysis

As mentioned in the Section 3.2, the semantic type of the node is `void*`. This choice was done juste before the first deadline because using pointers to actual node objects was triggering weird compile-errors and we weren't able to solve them in another way than using the type `void*`. This is unfortunate because it makes the code longer and less readable due to type conversion. When we (re)tried, later on, to use actual nodes objects, it seemed to work but we decided not to do it because it would take too much time. An improvement in terms of readability would be to use proper types then.
As the error reporting of `bison` is used, we rely on the underlying implementation. Therefore, even though we tried to display as much errors as possible, the error reporting is quite erratic. Sometimes, one error will be displayed even if the program contains more, sometimes it will display all of them. Moreover, we discovered a segmentation fault in the *bison* implementation for some erroneous program (which we didn't have time to debug). Adding some `error` rules in the parser would allow a more fine-grained error reporting.
Another problem is the column counting which is buggy. While the line count is accurate, the lexer fails at counting the starting and ending column of a token. Therefore, all the error reports throughout the execution of the program fails at given the correct column of an error. This is why we have disable a interesting feature of the error handler which consisted in displaying the program line triggering an error.

5.2 Scope and termination checking

5.3 Type Inference

The type inference is quite satisfying even if it can sometimes fail for programs that aren't expressive enough. Few improvements can still be performed :

- adding information in the **type error messages** : for the moment, the messages only report the location where the type checking fails, it would be interesting to know where was generated the constraint that led to the error
- adding a **check for unresolved types** : then the user could be noticed about the lack of expressiveness of its program with precise information

5.4 Optimization and machine code generation

As the optimization is performed by another application, improving it is impossible. Nevertheless, some others optimizations passes could be applied (as the `opt` command provides a lot more than 5 optimization passes).
As far as the machine code generation is concern, two problems arise :

- **system dependence** : as the compiler uses unix commands, it could only be used on unix-based operating systems which is configured so that `gcc` and LLVM commands can be used. Removing these dependencies would allow to port the application on other platform.
- **file dependence and generation** : on the one hand, the compiler needs the implementation files of the runtime to be able to execute machine code generation and therefore couldn't be used as a standalone executable. On the other hand, the compiler generates a lot of files without checking their names. Therefore, finding a way to embed the runtime files into the executable would allow an easier use of the compiler and a system that checks existing file names would prevent troubles caused by collisions.

5.5 Runtime

As said in the Section 4, the internal data structure (called *table*) used to store arrays and lists is itself a list. Therefore, accessing an element (list or array) has the a complexity $\mathcal{O}(n)$ where n is the number of objects in the table. Moreover, every function of the interface has to access a particular object in the table before performing operations, this resulting in a search at every interface function call. This is of course extremely inefficient. The reason why the list implementation was chosen is mainly lack of time. As a simply linked list is easy to implement, we could focus on the code generation instead. A major improvement would then be the reimplementation of the tables with an adequate data structure such as an ordered vector (ordered on the array/list id), a tree or a hashtable. Nevertheless, as implementation details don't leak through the interface, this can be done without changing the code generation.
Also, initially, we wanted to implement closures but we dropped this feature due to lack of time. A improvement in the usability of the language would be the implementation of such a mechanisms. This would include the implementation of another runtime module for handling it (and some parts of the compiler would also have to be modified).

A Sources

1. **Lexical and syntax analysis** :
   (a) Flex documentation, "*Lexical Analysis With Flex*", `http://goo.gl/wIJglV`
   (b) Charles Donnely, Richard Stallman, "*Bison, Using the Yacc-compatible Parser Generator*", `http://goo.gl/rp2kpM`
2. **Type inference** :
   (a) Paul N. Hilfinger, "*Lecture #22 : Type Inference and Unification*", `https://goo.gl/RR8Eme`, UC Berekeley, course *CS164* : Programming Languages and Compilers.
   (b) Cyril Soldani, "*Monomorphic type inference*", `http://goo.gl/Kveyvf`
3. **Optimization and machine code generation**
   (a) CPlusPlus.com, "*`system` - C++ reference*", `http://goo.gl/XkM9rr`
   (b) LLVM documentation, "*LLVM's Analysis and Transform Passes*", `http://goo.gl/P8kyxX`