University of Liège
Faculty of Applied Sciences

Sushi++ compiler
Project report [INFO2049]

Magera Floriane
1st master in computer engineering
Option : Computer systems and networkss
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 **??** 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 2.4.1). The available types are :

- `bool` : *true* or *false* (default : *true*)
- `char` : a character (default : *0*)
- `int` : an integer $\in [-2147483648, 2147483647]$ (default : *0*)
- `float` : a single-precision floating point value $\in [-3.403 \times 10^{38}, 3.403 \times 10^{38}]$ (default : *0.0*)
- `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 4.1).

2.2.2 Lists

2.3 Functions

2.4 Declarations

The *Sushi++* language provides declaration syntax for both functions and variables. The declaration keyword `maki` is common to both declaration.

2.4.1 Functions

Idea The function declared with the `maki` keyword are called **named function** in opposition to anonymous function (or *soy functions*, see Section 2.5.1). As soon as it is declared, a function can be either called (see Section 2.5.2) 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`.

2.4.2 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 }

2.5 Expressions

2.5.1 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 2.5.2) 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`

2.5.2 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)`

2.5.3 Operators

The *Sushi++* language provides a set of operators for expressing operations on flat types. Some of them are polymorphic as they can operate on multiple types. These operators are listed in the Table 1.

2.6 Statements

2.6.1 Loops

2.6.2 Conditionals

2.6.3 Switch

3 Compiler

3.1 General information

3.2 Lexical and syntax analysis

3.3 Semantic analysis

3.3.1 Scope checking

3.3.2 Termination checking

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.

Op.	Arity	Comment	Operand types	
			Operand 1	Operand 2
`op1 + op2`	2	**Addition**	{`int`, `float`}	same as `op1`
`op1 - op2`	2	**Substraction**	{`int`, `float`}	same as `op1`
`op1 * op2`	2	**Mutliplication**	{`int`, `float`}	same as `op1`
`op1 / op2`	2	**Division**	{`int`, `float`}	same as `op1`
`op1 % op2`	2	**Modulo**	`int`	`int`
`op1 ** op2`	2	**Exponent**	{`int`, `float`}	`int`
`- op`	1	**Unary minus**	{`int`, `float`}	
`++ op`, `op ++`	1	**Prefix/postfix increment**	{`int`, `float`}	
`-- op`, `op --`	1	**Prefix/postfix decrement**	{`int`, `float`}	
`op1 & op2`	2	**Bitwise and**	`int`	`int`
`op1 | op2`	2	**Bitwise or**	`int`	`int`
`op1 ^ op2`	2	**Bitwise xor**	`int`	`int`
`op1 >> op2`	2	**Right shift**	`int`	`int`
`op1 << op2`	2	**Left shift**	`int`	`int`
`~ op`	1	**Bitwise not**	`int`	
`op1 && op2`	2	**Logical and**	`bool`	`bool`
`op1 || op2`	2	**Logical or**	`bool`	`bool`
`! op`	1	**Logical not**	`bool`	
`op1 != op2`	2	**Not equal to**	{`int`, `float`, `bool`}	same as `op1`
`op1 == op2`	2	**Equal to**	{`int`, `float`, `bool`}	same as `op1`
`op1 < op2`	2	**Greater than**	{`int`, `float`}	same as `op1`
`op1 > op2`	2	**Equal to**	{`int`, `float`}	same as `op1`
`op1 <= op2`	2	**Less or equal to**	{`int`, `float`}	same as `op1`
`op1 >= op2`	2	**Greater or equal to**	{`int`, `float`}	same as `op1`
`op1 . op2`	2	**String concatenation**	`string`	`string`
`op1 += op2`	2	**Addition assignment**	{`int`, `float`}	same as `op1`
`op1 -= op2`	2	**Substraction assignment**	{`int`, `float`}	same as `op1`
`op1 *= op2`	2	**Mutliplication assignment**	{`int`, `float`}	same as `op1`
`op1 /= op2`	2	**Division assignment**	{`int`, `float`}	same as `op1`
`op1 %= op2`	2	**Modulo assignment**	`int`	`int`
`op1 **= op2`	2	**Exponent assignment**	{`int`, `float`}	`int`
`op1 .= op2`	2	**Concatenation assignment**	`string`	`string`

TABLE 1 – Sushi++ operators

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 [1a], 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, the complexity is $\mathcal{O}(1)$.

```
// Unification algorithm
unify ( α, β )
{
  if is_unresolved(α) || is_unresolved(β)
  {
    if ℋα ∩ ℋβ = ∅ // check hints compatiblity
      throw error("incompatible hints")

    // update hints
    ℋβ = ℋα = ℋα ∩ ℋβ
    // add indirection between the variables
    if is_unresolved(α) { α = β } else { β = α }

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

    for each parameters types variables γ of function α and δ of function β
      unify(γ, δ)

    unify(get_return_type(α), get_return_type(β))

    return
  }

  // variables are both uniparameter types
  if ( is_array(α) && is_array(β) ) || ( is_list(α) && is_list(β) )
  {
    γ = get_datastructure_type(α)
    δ = get_datastructure_type(β)
    unify(γ, δ)
  }

  // variables contains the same flat type
  if is_flat(α) && is_flat(β) && α = β
    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. These variables contains the information about the types expected by the parent. 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.

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` : base class for any actual type. 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 is an enumeration representing 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.

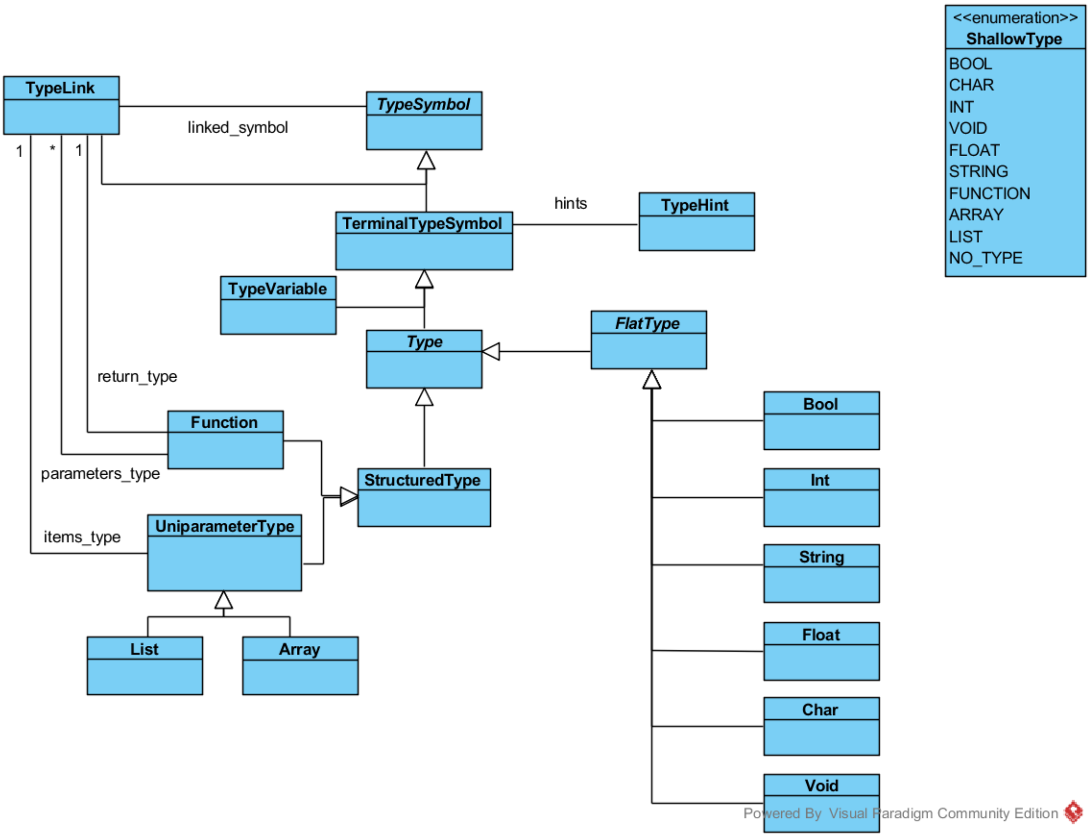

FIGURE 1 – Type symbols class diagram

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 of 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 mostly. The name of a bounded variable is therefore generated as follows :

  `identifier@scope_id`

Syntax-directed type inference Given the architecture described above, the type inference can be summarized into three kinds of actions :

1. **Table insertion** : which variables must be created in the type symbol table and with which terminal type symbols have they to be associated ?
2. **Unification** : which pairs of variables have to be unified ? does a variable have to be unified with a flat type ?
3. **Hinting** : has the hints of a variable to be updated ?

3.4 Code generation

3.4.1 Code generation

3.4.2 Code construction

4 Runtime

4.1 Garbage collector

5 Flaws and possible improvements

A Sources

1. **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 Diagrams