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 **??**) and the second details the compiler organization and algorithms (section **??**). 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 **??**). 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 **??**)
- `list` : a list of elements (see Section **??**)
- `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

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 **??**). As soon as it is declared, a function can be either called (see Section **??**) 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 **??** 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 **??**) 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 **??**.

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

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.

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

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 **??**.
- 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 algorithm : on the one hand, for each *Sushi++* construct, the type semantic has to be encoded into a rule defining the constraints to be generated. On the other hand, an unification algorithm has to be defined.

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 [**??**], slide 8). The algorithm is given in Listing **??** 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 $\Theta(n)$

```
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

Implementation

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.