Compilation Project

Dev documentation
Les Quatre MousquetEngineers
Master M1 MOSIG, Grenoble Universities

Cristian HARJA Yassine JAZOUANI Lina MARSS0
Clément MOMMESSIN

29/01/2015

1 Architecture of our compiler

1.1 Langage

For this project, we were torn between OCaml and Java. As one member didn't have any solid background of OCaml, we have wisely chosen Java. For the IDE, we have chosen to use IntelliJ IDEA.

1.2 File management

In the mini_camel folder, we have created 8 packages:

1. 'comp': in which we have all the files related to the compilor (FreeVariablesVisitor, AssemblyGenerator ...)
2. 'ir': in which we have all the files related to the Intermediate Representation
3. 'visit': in which we have all the files related to the different transformations and optimisations which we will describe in the following section.
4. 'type': in which we have all the files related to the typechecking of an expression
5. 'ast': in which we have all the files related the Ast.
6. 'util': in which we have all the files that help us visiting the AST Tree for all kind of transformations.
7. 'knorm': in which we have all the files related to the K-Normalization which generates us a K-AST Tree
8. We also have 4 files for manual testing in the main folder.

For our unit tests, we have created a new folder in which we have the unit tests for all packages. These test files are in the /src/test/java/mini_camel/visit.

2 Functionalities implemented

- Detection of free variables
- Type Checking
- Alpha Conversion
- Beta Reduction
- Constant Folding
- Inlining
- Elimination of unused let-expression
- K-normalization
- Intermediate Code Generation
- Assembly Code generation (without float)

3 Choice of algorithms

We already had a parser given by our teacher. This parser produced an Abstract Syntax Tree (AST) which had to be type checked. We checked all free variables. Then we applied all the transformations to simplify the AST, generated the Intermediate Representation, and then generate the Assembly Code.

Let's see how the different parts of our compilor is implemented.

3.1 Transformations

Globally, for all transformations and pre-processing, we have chosen to use 3 different interfaces, which are:

- Visitor
- Visitor 1
- Visitor 2

The differences between these 3 patterns are the returned result :

1. Visitor.java doesn't return any value
2. Visitor1.java returns a value
3. Visitor2.java returns a value but also takes an argument which is mainly used for context and symbol tables.

Thanks to these different interfaces, it's possible to visit the tree and to do some computation depending on whether we are in presence of a AstAdd, AstEq etc...

3.1.1 Inlining

To apply inlining to our AstTree, we first select only the none recursive functions.
One example of none recursive function would be

```
let rec f x = x + 1 in f 3;;
```

Then once we have a list of functions needed to be inlined, we just recursively visit the AstTree and apply the following transformations:

- If we have an AstLetRec, we just fill in a list.
- If we have an AstApp, we check if the following function can be inlined and then we inline it.

3.1.2 Elimination

To apply elimination of useless declarations,we visit all the tree, then:

- If we have a variable declaration, we fill a list called left with the ID of the declared variable
- Each time a variable is seen at the right side of an equality, this variable cannot be eliminated.

Here is an example of elimination:
This expression:

```
let x = 3 in print_int(3);;
```

leads to this one:

```
print_int(3);;
```

3.2 K-normalization

Implemented in `src/main/java/mini_camel/visit/KNormalize.java`, recursively traverses the AST and usually relies on the `KNormalizeHelper.insert_let*` functions to help build the K-normalized AST.

Note that the transformed program is now represented in a new hierarchy of classes, very similar to the `Ast*`, but simplified to be very specific about its structures (for example, `AstAdd` allows any 2 sub-expressions as its operands, but `mini_camel.knorm.KAdd` only allows 2 identifiers as its operand, which should contain the results of the expressions).

The `insert_let*` functions take K-normalized code as arguments and, if they can't be used as operands in another expression (because they are not identifiers, but constants or complex expressions), they generate one `let` for each complex expression, and then calls `KNormalizeHelper.Handler*.apply(...)` (user-supplied callback) to generate code that actually uses the operands (which are now guaranteed to be identifiers).

3.3 Closure conversion

Based on K-normalization, extends it with a few special instructions (`ApplyClosure`, `ApplyDirect` and `ClosureMake`).
The implementation in `src/main/java/mini_camel/visit/ClosureConv.java` looks for special nodes in the K-normalized code: `visit(KLetRec)` detects closures (and possibly converts them) and `visit(KAdd)` determines how a function should be called.

3.4 Intermediate Code Generation

To go from the AST which has been K-normalized to the Generation of Assembly Code, we need first to define a bridge bewteen these 2 representation which is the Intermediate Representation (IR).
This representation divides the AST into separated functions (one main, and as many functions as there are "let rec"). Each body of a function is a list of instructions (Instr) applied to operands (Operand) that makes this representation close to the Assembly Code.

The list of defined instructions are :

- ASSIGN for assignment
- ADD_I, SUB_I, ADD_F, SUB_F, MUL_F, DIV_F for integer and float computation
- LABEL, JUMP, BRANCH for boolean expressions and labels for functions
- CALL, RETURN, CLS_MAKE and CLS_APPLY for calling, returning functions with values, making and applying closures
- ARRAY_NEW, ARRAY_GET, ARRAY_PUT for making and using arrays

We can notice the existence of instructions on floats that are defined for the IR but not handled yet during the Assembly Code Generation.
The implementation of Tuples is handled by representing them as arrays.

3.5 Assembly Code Generation

The assembly architecture generated is ARM assembly.

Roles for each registers :

- r0 - r2 are 'intern' registers used by the system to load/store values and variables, containing adresses, indexes for loops or accessing memory and other internally processes of the compiler.
- r3 - r10 are "scratch registers" containing local variables
- r11 is used to store the return value of each function. If the return value of a function is "void" this register remains untouched during a call to this function.
- r12 is the frame pointer (FP), for each entering fnction the caller's FP is pushed in the stack and the nexw FP is initialized to point on that cell of the stack.
  FP is used to access to the parameters of the called function, and easily return from the call.

Register allocation :
There is a mapping from indexes of registers (3 to 10) to the name of the variable stored in the register and a cursor pointing to some index. There is also a list of variables that are in the data section of the assembly code.
When all registers are used and we need another one, the variable stored in the register pointed by the cursor is spilled in the memory, and the variable is added in the list, so that the register is now free and the cursor is moved to the next index.

4 Problem encountered and solution found

- We had some minor problems concerning the IR. We couldn't find a simple and efficient way to implement it. Now, we use the interface Visitor3.java that uses the same methods used in the other interfaces : visit(AstXXX)
- We have some issues during the assembly code generation for float. We have some ideas but nothing concrete right now.
- We had some issues concerning inlining. Indeed, when we do a `print_int` or `print_newline`, we had a transformation that transforms these expression into to `?vX = print`. Then, when elimination was applied, since the variable `?vX` was never used, it eliminated the print. We have then chosen to modify our elimination. Now, when the variable begins with `?v`, it doesn't eliminate it.
- k-norm

5 Future development and improvements

5.1 Preprocessing

- Let Reduction

5.2 Assembly Code Generation

- Handle float computation
- Modify the register allocator to spill local variables in the stack instead of the data section
- Merge the function "check_heap" into the function "malloc"
- Enable/disable the heap management : don't add the code for the heap initialization and the function "malloc" if the programm never uses Arrays or Tuples