Compilation Project

Development Documentation

Master 1 INFO & MOSIG, UGA and Grenoble Inp
Team1
Ayan Hore, Lyubomyr Polyuha,Tuo Zhao, Kritika Mehta, Sitan Xiao,
17/01/2018

1 Architecture compiler

1.1 Language

For this project we have chosen Java as the programming language and ECLIPSE IDE.

1.2 File management

In the Compiler/SRC folder we have the following packages:

1) Visiteur: It contains all the Visitor classes used in all the frontend transformations
2) Types: It contains all the types used
3) TypeChecker: It consists of TypeChecking class which is partially implemented
4) Tool: It contains the class Id for Id used in expressions
5) Expression: It contains all the expressions classes used
6) Heights: It contains the Height and HeightVisitor class which computes the height
7) K_nor: It contains the K_nor class which performs K-Normalization and gives the K-normalized AST
8) Alpha_Conversion: It contains the file Alpha_Con which performs alpha conversion on the output of K normalized AST
9) Reduction_nested: It contains the file Reduction_N which performs let reduction on the AST output by other transformation
10) Closure_Conversion: It contains the file Closure_Con which performs Closure conversion on the AST output by other transformation
11) ASML Code Generation and ASML : These contain the code for ASML Generation
12) Registers: It contains the Registers file which contains all the information about a register and its initialization.
13) Instructions: It consists of all the instructions used in the ARM generation

14) ARMGen: It consists of all the files related to ARM generation and the data structure conversion

Functionalities implemented:

- K normalization
- Alpha Conversion
- Reduction of let expression
- Closure Conversion
- ASML Generation
- ARM Generation

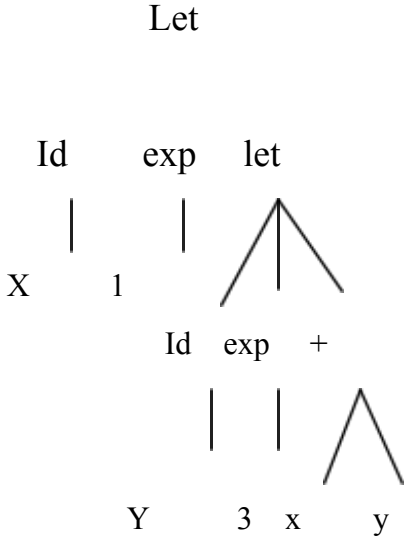
2 Choice of algorithms

The parser provided to us generates and an Abstract Syntax Tree (AST) . The transformations like K normalization, Alpha Conversion, Closure Conversion, Reduction of Let was applied to simplify the AST. The ASML is generated and then the ARM Code.

2.1 K-normalization

New fresh variables are used to replace all nested expressions.

For example,



For the function of let rec, we have to get the parameters(id, type,args,e).

2.2 Alpha Conversion

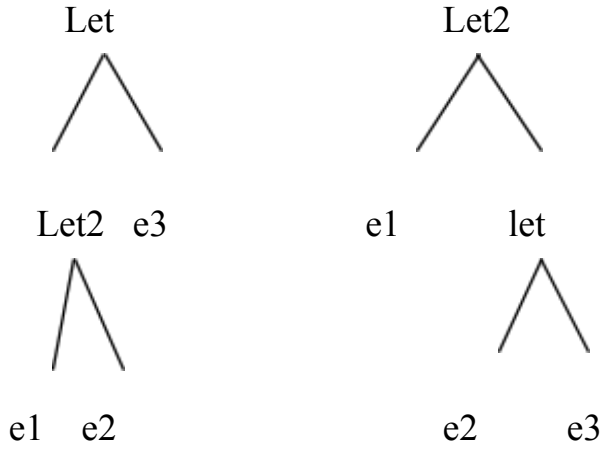
Every variable defined by let or function parameters should be different. HashMap is used for this purpose which stores all the variables. For the expressions containing id or a list of id's, it is checked if the variable is already present in the HashMap . If it is present, a new id is generated otherwise the original is used and saved in the HashMap.

2.3 Reduction of Let expression

All the let definitions are flattened but not the let rec.

For example,

For "Let":



For "let tuple":

There are three situations like the below

E1 is a let:

let (id1,id2,...) = (let y = e1 in e2) in e3 -> let y = e1 in let (id1,id2,...) = e2 in e3

E1 is a let tuple:

let (id1,id2,...) = (let (idy1,idy2,...) = e1 in e2) in e3-> let (idy1,idy2,...) = e1 in let (id1,id2,...) = e2 in e3

E1 is a let rec:

let (id1,id2,...) = (let fd = e1 in e2) in e3-> let fd = e1 in let (id1,id2,...) = e2 in e3

2.4 Closure Conversion

In the step, a label is assigned to a function and the label is printed. A loop is used to check every parameter of a function if there is no function in parameter.

2.5 ASML Generation

All the expression_asml in ASML and the relation among those expression_aml are created, example, inherit relation.

```
fundefs:
| LET UNDERSC EQUAL asmt
| LET LABEL EQUAL FLOAT fundefs
| LET LABEL formal_args EQUAL asml fundefs
```

Then the output of ASML can be printed in the console and put into a local file. The transformation process gives the ASML and then data structure is filled accordingly for the backend.

2.6 ARM Code Generation

The expressions obtained after front end transformation need to be converted to fill a data structure which acts as a bridge between the 2 representations. The AST is divided into separate functions which is a list of instructions and operands to which the instructions are applied. This representation is then used to generate the ARM code.

We obtain ARM code from the intermediate representation. It is performed in 2 steps

Register Allocation:

There are 2 types of registers available, r4-12 for the local variables and r0-r3 for the arguments. Initially, the very basic allocation strategy to assign different register to each variable was used which stops once we run out of registers. Now the registers are allocated and once they are non empty, the variables and arguments are spilled to the stack and indirect addressing is used [fp, i] where i is the offset. To implement this, a function is initialized with all the available registers and all the variables and arguments are assigned a register and the register is removed from the list of registers available for the function. If there are no registers available, the variables are spilled to stack and offset is calculated accordingly.

The second step is the generation of ARM code from the frontend output. We use a data structure the store the output from the frontend transformations. The data structure consists of different types of instructions that are encountered. The data structure conversion file takes the transformed AST expressions and returns the instructions along with filling the variable lists and instructions lists for the functions. Registers are allocated accordingly.

3 Problem encountered and solution found

1. There was a problem integrating the front end and back end and how to fill the data structure for the ARM generation. Now, a data structure conversion class is used which takes the transformed AST output as the input and extracts the instructions from the expression to add it to the data structure.

4 Future development and improvements

4.1 ARM Code Generation

The register allocation strategy can be enhanced and better strategy like linear scan can be used. The Arm generation is only implemented for arithmetic expressions, call to functions, simple first order functions and conditional branch (if-then-else); the implementation can be extended to float and should be able to handle tuples, arrays, closure.

4.2 Type checking

Type checking is only partially implemented and it can be improved upon.