Inlining and Local Function Definition

Compiler Construction '18 Final Report

Lionel Pellier Maxime Pisa
EPFL
{firtname.lastname}@epfl.ch

1. Introduction

We first implemented a compiler following the given pipeline: Lexer, Parser, Name Analysis, Type Checking and Code Generation.

The parser step not only gives meaning to the input through the recursive descent LL1 parsing, it also constructs the Abstract Syntax Tree(AST): the tree representation of the abstract syntactic structure of the source code. By focusing on the rewriting of those trees, we can implement optimizations allowing our compiler to move load from the runtime to the compile time. This is what we do with the inlining optimization and the constant folding. The inlining consists in replacing a call to function by the body of the function assured that the function has been declared as an inlined function, typically through the use of the "inline" keyword. This allows to reduce overheads induced by a call to a function and it is generally a good optimization for short functions that are mainly defined for modularization purposes. This optimization makes even more sense when used with the constant folding: an optimization that allows to evaluate constant expressions at the compile time therefore decreasing the number of operations and registers needed at runtime.

We also modified the compiler to add local function definitions support to our language.

Once the inlining and the local functions are implemented, it is interesting to use them together. Indeed, "inlining is particularly useful when applied to auxiliary functions that only exist for clarity. While inlining can lead to code explosion when applied too liberally, note that inlining a non-recursive function that is only called in a single location will strictly reduce code size and potentially lead to more efficient code". Automatically inlining such functions allow us to make the most out of our extension.

2. Examples

2.1 Inlining and constant folding

Here is an example of a source code making use of the "inline" keyword before it compiles. Without our extension, the program would rest as such after compilation (excluding the "inline" keyword of course).

```
object TestInlining {
  inline def abs(n: Int): Int = {
    if (n < 0){
      −n
    }
    else{
      n
    }
  }

  inline def times2(n: Int): Int = {
    2 ∗ n
  }

  inline def func(n: Int): Int ={
    2 + 2 + times2(times2(n+1))
  }

  val a: Int = abs(123);
  val b: Int = abs(−456);
  val c: Int = times2(3);
  val d: Int = func(1)
}
```

However there is a lot of room for optimizations in such a program with the inlining + constant folding extension. Below is the program once the source code has been through our extended parser that rewrote the AST.

```
object TestInlining {
  inline def abs(n: Int): Int = {
    (if((n < 0)) {
      n
```

```
    } else {
      n
    })
  }

  inline def times2(n: Int): Int = {
    (2 * n)
  }

  inline def func(n: Int): Int = {
    (4 + (4 * (n + 1)))
  }

  val a: Int = 123;
  val b: Int = 456;
  val c: Int = 6;
  val d: Int = 12
}
```

2.2 Inner definitions

The implementation of local function definitions allows us to use this kind of function (here the example use a recursive function to not trigger our automatic inlining feature):

```
def foo(n: Int): Int = {
    def plus1(n: Int): Int = {
      if(4 < n) {n}
      else { plus1(n + 1)}
    }
    plus1(n)
  }

  foo(1)
```

2.3 Automatic inlining of inner functions

If we take the source code of the example presented in the project description:

```
def foo(n: Int): Int = {
    def plus1(n: Int): Int = { n + 1 }
    inline def times2(n: Int): Int = { 2 * n }
    plus1(times2(times2(n)))
  }
```

We have once we rewrote the program:

```
def foo(n: Int): Int = {
    inline def plus1(n: Int): Int = {(n + 1)}
    inline def times2(n: Int): Int = {(2 * n)}
    ((4 * n) + 1)
  }
```

Here the plus1 function has been turned into an inlined function.

3. Implementation

3.1 Theoretical Background

Constant folding is a classical compiler optimization that has been largely discussed. We took the approach mentioned in the CS-402 course from the University of Rhode Island [1] that consists in rewriting the AST for the binary operations even though their examples were simpler than the cases we were dealing with.

3.2 Implementation Details

3.2.1 Change in the architecture

We needed to be able to differentiate inlined functions from the standard ones. To do so we changed the case classes FunDef and FunSig allowing them to take an *isInlined* boolean parameter. We also needed to differentiate local functions from normal ones and to be able to tell what local functions a function had. To do so we added to the case class FunDef a list of Fundef and a boolean parameter *isLocal* and added a second boolean parameter to FunSig *isLocal*. This results in changes across all the project to adapt to the new constructors.

3.2.2 Lexer

As our extension relies on the use of a new keyword, we extended the list of tokens in Tokens.scala. This induces an addition of a case in the pattern matching on the keywords in the Lexer.scala file.

3.2.3 Parser

This is the part of the pipeline that contains the bulk of our work on the inlining and constant folding. The Amy grammar as well as the ASTConstructorLL1 also had to be changed to support local function definition.

Inlining

The first part consisted in being able to parse the new keyword. Hence we modified the Parser.scala file by adding a second production rule for the FunDef production consisting of the INLINE() token + the first rule.

We adapted the constructDef0 method of the ASTConstructorLL1 to take care of the new rule. When a parsed function is an inlined one, we need to inform the constrExpr function that it needs to constant fold the content of the function, that also applies for the

[1] https://homepage.cs.uri.edu/faculty

arguments. To do so, we added a cstFolding boolean parameter to the constrExpr function and to the helper functions (constrExpr1, constrExpr2, ...) to be able to propagate this value. Moreover, we add an entry to a local mapping *inlinedFunctions* that maps the name of an inlined function to a tuple made of the FunDef and the unconstructed body (NodeOrLeaf[Token]). We use this mapping when dealing with the Call production.

When calling a function, we check whether this function is inlined or not through the inlinedFunctions mapping. If it is not the case, then as before we return a Call Expr otherwise we need to be able to construct the body of the function. The tricky part here is that we need to be able to replace all parameters by the arguments that we evaluated (with constant folding as it is an inlined function). For example we need to take care that the following code:

```
inline def times2(n: Int): Int = {
  2 * n
}

inline def func(n: Int): Int ={
  times2(times2(n+1))
}
```

becomes (without constant folding):

```
inline def times2(n: Int): Int = {
  2 * n
}

inline def func(n: Int): Int ={
  (2 * (2 * (n + 1)))
}
```

instead of:

```
inline def times2(n: Int): Int = {
  2 * n
}

inline def func(n: Int): Int ={
  2 * n
}
```

The last sample of code is what we would get if we trivially replaced the call to the times2 function by its body. The problem is that we cannot use the body of the funDef object as it has already been constructed in an Expr preventing any modifications. This why we also stored the NodeOrLeaf[Token] body in the mapping's tuple. However if we just applied the constrExpr on it, we would get the same result that the body stored in funDef. That is why we rely on another mapping *inlinedLocals* that maps a Name (of parameter) to an Expr. Before constructing the body, we add an entry in this mapping for each argument where the names are fetched from the *params* field contained in the funDef object and the Expr from myargs: the arguments that we constructed with constructList and constructExpr. We make use of the inlinedLocals mapping in another production rule of Call, the one that is used when a function call of its arguments. Instead of using the Variable(name) Expr, we use the Expr stored in the mapping. With this the inlining is achieved but we still need to take care of the constant folding.

Constant Folding

As we decided to follow the approach described in the last part for the constant folding, our work is focused on the constructOpExpr where we call our constantFolding function when needed (cstFolding = true). For each operation, we pattern match on all the binary operations. From there, the most trivial thing that we can do is reducing an operation to its result when operands have the same type [1]. To be able to use the operands we extended the Expr trait with functions to typecast an IntLiteral into Int, a BooleanLiteral into Boolean or a StringLiteral into String.
However it is not enough since we also need to take into account cases where operands do not have the same type but where a reduction is still possible [2]. Note that the left operand of the first operation having the same literal type than the left operand of the second operation is not a sufficient condition to apply constant folding as we can be in a case where the priority of the operation would change the result of the operation[3].
To address this issue, we reduce such operations only when both operations have the same level of priority. Choosing to apply a reduction when the priority of second operation is inferior or equal to the first one would have implied to implement the distributivity of some operations and that would have not make any sense since it would not decrease the total number of operations.

There are corner cases to the method we chose for the constant folding. Indeed if we did nothing more we could not apply constant folding on unary operations such as Neg and Not or the conditional statement. We solved those issues by interpreting the result of the constant folded constrExpr.

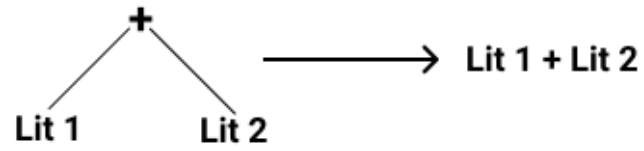

Figure 1. Constant folding for two literals

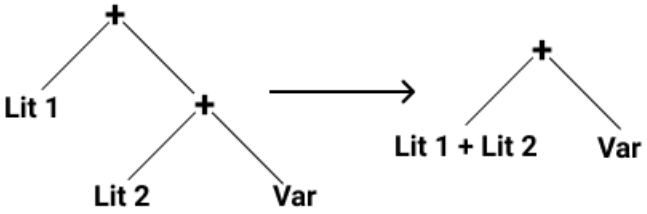

Figure 2. Constant folding where operands do not have the same type

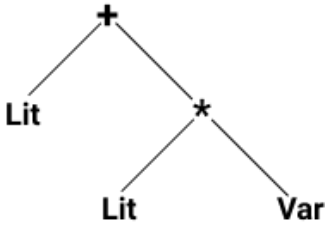

Figure 3. Constant folding is not possible due to priority of the operations

The Neg and Not operations were initially defined as OP(constrExpr(expr)). Instead, we check if the result of constrExpr(expr) has been constant folded to a BooleanLiteral in the case of the Not operation and a IntLiteral for the Neg operation. If this is the case we recreate a literal for which the apply the operation on the value otherwise there is no need to change the result of OP(constrExpr(expr)).

When encountering a If production rule we construct its condition. If the condition is reduced to a BooleanLiteral having a value equals to true then we return the Expr for the thenn part, if it is reduced to a BooleanLiteral having a value equals to false we return the Expr for the elze part. Otherwise we return a If Expr.

Local functions

We added to the Amy grammar a new rule which allowed us to have a local list of function definition before the expression of a function.

```
'FunDefLocal ::= 'FunDef ~ 'FunDefLocal | epsilon(),
```

We added to the existing rules for function our keyword to enable local function definition.

```
LBRACE() ~ 'Expr ~ RBRACE()
```

Became:

```
LBRACE() ~ 'FunDefLocal ~ 'Expr ~ RBRACE()
```

To support this change we modified the ASTConstructorLL1, more precisely we modified the ConstructDef0 method to call a new method called constructFunDefLocal, which constructs every local function using the also new method constructFunDef, and returns a List of FunDef. This list is then stored is the original fonction definition. Note that this process of local function constructions is recursive allowing us to define local functions of local functions.

Automatically inlining local functions

To automatically inline a function one needs to know:

- if the function is a local function
- if the inlining of the function can leads to code explosion, that is if the function is recursive.

The structure we implemented in our parser allow us to easily check whether a function is a local one or not. The main focus is then to find the recursions in a program which is not trivial as even though we have the name of the function we want to call, we do not have the name of the caller.

Our approach consists in a two pass kind of algorithm relying on two data structures: a MutableList that stores the name of each function that has been called in the parsing of the Call production, and a HashMap that maps a function name to the list of the functions it uses (the MutableList converted to List).

By constructing the body of a function a first time, we fill our data structures. We then check if there is any recursions among the calls through a function we implemented: *shouldInlineRecur*. This function takes four arguments: *start*, *current*, *actualDepth* and *maxDepth*. The first argument corresponds to the name of the function which we want to know if we should inline it or not, the second argument *current* is the current name of the function being verified, it is initially empty and the third and last arguments are to prevent very long calculations. The function verifies that the name of the function *start* isn't called in its body. If it is, it returns false, if it isn't it recursively call itself on every one

of the function that his body calls changing the *current* parameter to the name of the function called and incrementing the *actualDepth* parameter. The *maxDepth* parameter serves to prevent crazy long computations for example in the case that there is a call to a library function that call many other functions. This way any looping calls or recursions are identified and the initial function isn't deemed safe to inline. Of course it there are no function calls in the body true is returned.

If a functions meets the two conditions stated above, we add its funDef and body to the *inlinedFunctions* mapping and we reconstruct its body, this time allowing the constant folding. With this, a non recursive inner function will be inlined without the use of the inline keyword.

3.2.4 Name Analysis

To support local function definition the class NameAnalyser had to be changed. First the discovery of function definitions and their addition to the symbol table needed to be added. For this we added a recursive local function addLocalDefsToSymbolTable. This appropriately named function takes a list of local function definitions and adds them to the symbol table, then recursively for each local function definition of the list, calls itself on its local function definitions. The function transformFunDef naturally had to be modified it also calls a new recursive method called transformFunDefLocals which processes all local function definitions. We wanted to implement correct scoping and to achieve this the method transformFunDefLocals adds to the locals of its own body a binding from the name of every local function to a new fresh identifier. This is used in transformExpr in the case of a function call. After having retrieved the signature of the function a second test is made to see if the function name is in fact in the locals. This prevents local functions from being called elsewhere than their scope.

3.2.5 Type Checking

The local function needed to be typechecked. For this we simply defined a new function localDefsTypeCheck which recursively typechecks every local function.

3.2.6 Code Generation

The cgModule function had to be modified to not only generate code for all function but also to generate code for their local function definitions.This was fairly simple we added a function called getLocalFun which recursively generated code for every local function.

4. Possible Extensions

As stated in the original description of the project, our local functions only have access to their own parameters and locals, but not the surrounding functions parameters or locals. It would be interesting to go further and change that making it possible for a local function to have access to local functions defined at the same level.

Automatically apply inlining is a powerful feature that can go beyond the local functions. Modern compilers uses different kind of heuristic to be able to decide when it is useful to automatically apply inlining. It would be possible to go further by making a review of the existing inlining heuristics and implement them.

References