Staging for general purpose languages

Thais Baudon

Univ Rennes

Abstract. Reliable and easy to maintain source code rarely yields efficient target programs. Conversely, efficient code is usually hard to write, hard to maintain and prone to errors. One way to retain both good programming practices and optimal performance is to write a high-level program generator that outputs an efficient, specialized target program. Staging is an approach that combines code generation and target instructions in a single meta-program, by delaying the evaluation of target expressions. LMS (lightweight modular staging) is a library-based framework that acts as an embedded compiler for such meta-programs. It is particularly well suited for embedded domain-specific languages, which can be built on top of the framework. However, general purpose languages require a different approach, since they are typically not restricted to a superset of another language. The approach presented in this paper is to extend LMS with a frontend that maps a small subset of Scala to LMS intermediate representation, and a backend that generates x86 instructions from this representation. This allows to effectively use LMS as a compiler for a restricted general purpose language. Future work could extend this approach to more expressive general purpose languages.

Introduction

Good programming practices and program performance are often at odds. While code featuring high-level constructs, abstract structures and generic software components is easy to write and maintain, it can significantly hinder program performance. On the other hand, optimized, low-level code yields highly efficient programs, but is harder to write, to maintain and prone to errors.

In order to maintain both efficient software development and acceptable performance, one can write a high-level program generator that outputs a low-level target program, rather than the target program by hand. This is known as generative programming. Lightweight modular staging (LMS) is an approach that enables generator and target code to be combined in a single staged program. This approach is particularly well suited to embedded domain-specific languages. However, general purpose languages require a different approach, since they are typically built as a separate 'standalone'language, rather than on top of another language.

The approach presented in this paper extends the LMS framework with a frontend and a backend to compile a small general purpose language to LMS intermediate representation, then to x86 assembly. Section 1 describes the LMS framework and the approaches it is based on. Sections 2 and 3 present the frontend and x86 backend. Finally, section 4 discusses limitations and future work.

1 Background

LMS (*Lightweight Modular Staging*) is a library-based generative programming and compiler framework. It provides a relatively simple and reliable way to partially evaluate programs.

1.1 Partial evaluation

Partial evaluation generates a specialized version of a program, which is potentially much faster than the general version [1]. Let p be a program that takes two inputs i_1 and i_2. Instead of evaluating the application of p to a pair of inputs (j_1, j_2) in a single step, it is also possible to evaluate it in two steps: first partially evaluate p with input j_1 to produce a new program r, and then evaluate r with input j_2 to produce the result. r is a specialized version of p for the particular value j_1 of the first input. It is called a residual program and is often much faster than the general program p. For example, partially evaluating the power

```
def power(b: Int, n: Int): Int = {
  if (n == 0)
    1
  else
    b * power(b, n - 1)
}

// After partial evaluation for n = 3
def power(b: Int): Int = b * b * b
```

function in 1.1 with $n = 3$ yields a specialized cube function.

Partial evaluation has several useful applications, including multistage programming and Futamura projections.

1.2 Generative programming and staging

Writing reliable and maintainable programs often involves using high-level abstract structures and generic code components. However, these usually significantly hinder program performance. Conversely, highly specialized code with little or no high-level structures produces efficient programs, but is significantly more error-prone and difficult to write and maintain [2].

Generative programming aims to achieve both acceptable performance and maintainable programs. Its principle is to write a generic high-level program generator that outputs a specialized and highly efficient target program, rather than a low-level program directly.

This allows to use high-level, generic programming abstractions when writing the program generator, yet keep the target program specialized and highly efficient. Indeed, high-level structures are only used in the generator code, while the snippets of target code are written in a low-level, efficient style.

One way to write such a program generator is *multistage programming* (also known as *staging*). A multistage program contains some *staged* expressions, whose evaluation is explicitly delayed. This allows to express code generation instructions and target code in a single meta-program, by staging the expressions that are part of the generated code. The target program is generated by evaluating the code generation instructions in the meta-program, i.e. partially evaluating it.

1.3 Futamura projections

For most programming languages, writing a compiler is much more complex than writing an interpreter. However, given a way to partially evaluate any program, Futamura projections allow to essentially turn any interpreter into an equivalent compiler [3].

For any program source p, let p^* be a compiled version of p. Then a compiler can be seen as a function that maps any program p to its compiled equivalent p^*:

$$\text{compiler} : p \mapsto p^*$$

First projection An interpreter can be seen as a function that, given a program p and some dynamic input x, returns the computation described by p applied to x:

$$\text{interpreter} : (p, x) \mapsto p^*(x)$$

Partially evaluating the interpreter with a particular program p yields a specialized interpreter that maps any input x to $p^*(x)$:

$$\text{interpreter}_p : x \mapsto p^*(x)$$

In other words, specializing an interpreter for a particular program yields a compiled version of that program:

$$\text{interpreter}_p := p^*$$

Second projection Let specializer be a program that, given an interpreter and an object program, outputs a compiled version of that program by specializing the interpreter, as in the first projection:

$$\text{specializer} : (i, p) \mapsto i_p = p^*$$

Then specializing the specializer for a particular interpreter yields a tool that maps any program to a compiled version of that program, i.e. a compiler:

$$\text{specializer}_i : p \mapsto p^*$$

$$\text{specializer}_i = \text{compiler}$$

Third projection A specializer can also be seen as an interpreter interpreter. Therefore, it is possible to specialize it for itself. This yields a tool that maps any interpreter to an equivalent compiler:

$$\text{specializer}(\text{specializer}, i) = \text{specializer}_i = \text{compiler}$$

$$\text{specializer}_{\text{specializer}} : \text{interpreter} \mapsto \text{compiler}$$

1.4 Lightweight modular staging

Overview Staging and Futamura projections can significantly lower the complexity of writing certain programs. However, these techniques require a relatively simple way to stage expressions when writing the program generator and a tool that can reliably partially evaluate the program.

Lightweight Modular Staging (LMS) [4] is a framework that meets these two requirements.

LMS uses types to distinguish code generation instructions from staged expressions, taking advantage of the Scala type system. Staged expressions that represent the later computation of an expression of type T are of type Rep[T] in the program generator. For example, staging the base expression in the power function (1.4) yields a power function specialized for some exponent after partial evaluation (1.4). LMS acts as an embedded compiler. Unlike a 'standalone' com-

```
def power(b: Rep[Int], n: Int): Rep[Int] = {
  if (n == 0) 1
  else b * power(b, n - 1)
}

val x: Rep[Int] = someComputation()
power(x, 5)
```

piler, it doesnt compile the whole program, but instead provides an implementation for staged expressions. Instead of evaluating expressions, this implementation generates graph nodes that represent their computation. Therefore, compiling the staged program yields a graph representation of the partially evaluated program. Staged expressions are still present in that graph as nodes, whereas other expressions have been evaluated and define the structure of the graph.

An embedded code generator can then schedule nodes and generate a program according to this schedule, in some target language (e.g. Scala or C).

```
def power(b: Int): Int = {
  b * b * b * b * b
}
val x: Int = someComputation()
power(x)
```

Intermediate representation The intermediate representation of a program in LMS is a directed graph. Its nodes represent the program statements, while its edges represent the effect dependencies between these statements.

Effect dependencies are dependencies that result from two statements' effects (e.g. accessing a mutable object), as opposed to data dependencies. For example, in 1.4, the statement s_2 has an effect dependency on s_1 ('write after read'), because they both access the same mutable variable a. In 1.4, s_2 cannot be scheduled before s_1 because it uses its result. However, the dependency between these two statements is a data dependency, rather than an effect dependency.

```
var a = 0
val b = a + 1 // s1
a = 2 // s2
```

```
var a = 0
val b = a + 1 // s1
val c = b * 2 // s2
```

Unlike a control flow graph, which contains fixed 'basic blocks', and similarly to a 'sea of nodes' [5], nesting dependencies are represented as edges from a node to a block's *effect input*, which is a special symbol that 'anchors' nodes into the block. All nodes are in the same graph, regardless of which scope they were created in. Therefore, a node that was created inside a nested scope but has no effect dependency has no link to the corresponding block, which means that it is not forced inside the block and could be scheduled outside of it. A node is forced inside a block if, and only if, it depends on this block's effect input or on another node forced inside the same block. This enables more flexible scheduling than a control flow graph. However, unlike a 'traditional' sea of nodes, the graph retains the structure of the original program (e.g. loops, functions, etc.). Graph generation does not involve translation of high-level control structures to lower-level structures. This enables to apply optimizations and scheduling algorithms to the graph without carrying out dataflow analysis, since there is no need to retrieve information about the original program structure.

```
var x = 0
while (x < 10) {
  val a = 2 // can be moved outside the loop
  x += a // effectful (reads and writes x), must stay inside the loop
}
```

Fig. 1. LMS graph representation of 1.4

Code motion and scheduling Scheduling each node as a statement in the target program is done by recursively applying the *code motion* algorithm to each block, starting with the global program block [4]. This algorithm, given a block and a list of unscheduled nodes, determines which of these nodes are 'toplevel' in this block, as opposed to nodes that belong in a nested block. Code motion is then recursively applied to all blocks that belong to toplevel nodes.

The algorithm attempts to schedule each node in a block that will minimize its frequency of execution. Frequency of execution is determined relatively to a given source node for each node reachable from it. Reachable nodes include data dependencies, effect dependencies, and nodes used in blocks that belong to the source node. Nodes used in a block are its result and the nodes that have been determined to belong in that block by the effect system (see 2.5).

The frequency map of a source node maps each reachable node to one of three frequency values: *cold*, *warm* and *hot*. Nodes used in conditional branches are expected to be executed less often than the conditional node itself, and are therefore considered *cold* nodes. Nodes used in loops or functions are, on average, executed more often than the loop or function node itself, and are considered *hot* nodes. All other reachable nodes are assumed to be executed roughly as often as the source node, and are considered *warm* nodes. Therefore, a minimal frequency of execution is achieved by scheduling as many nodes inside toplevel conditionals and as few nodes inside loops and functions as possible. The code motion algorithm first determines which nodes are reachable through a 'warm' (resp. 'cold') path, that is, a path which doesnt (resp. does) cross any toplevel conditional. The initial set of reachable nodes contains the nodes used in the current block, which are toplevel by definition. The algorithm then browses each unscheduled node (in reverse topological order) and determines whether their reachable nodes are reachable through a warm or cold path.

The second step of the algorithm schedules the reachable nodes in the current scope (toplevel) or in a nested scope. Nodes that are part of a warm path, by definition, cannot be scheduled inside any toplevel conditional. In order to minimize their frequency, each of these nodes is scheduled in toplevel if it is available in the current scope, that is if, and only if, it doesn't depend on any unbounded variable. Nodes that are part of a cold path but outside any warm path can be pushed inside a toplevel conditional, and are therefore scheduled in a nested scope. Some nodes are not reachable through any path. Such nodes are not

```
def codeMotion(block, scope, innerNodes):
  newScope = boundSyms(block) ++ scope

  warmPath = usedSyms(block)
  coldPath = empty

  for node in innerNodes.reverse:
    if node in warmPath:
      if available(node):
        for (r, f) in frequencyMap(node):
          if (f == cold), coldPath += r
          else warmPath += r

      // node will be scheduled inside a loop
      else warmPath ++= node.reachableNodes

    if node in coldPath:
      coldPath ++= node.reachableNodes


  toplevel = Nil
  nested = Nil
  for node in innerNodes.reverse:
    if node in warmPath:
      if available(node), toplevel = node :: toplevel
      else nested = node :: nested
    else if node in coldPath, nested = node :: nested

  for node in toplevel, visit(node, newScope, nested)
```

Fig. 2. Code motion algorithm

used anywhere in the block and don't need to be scheduled. This provides some amount of dead code elimination during scheduling, with no specific optimization pass needed.

2 Frontend

Miniscala is a subset of Scala that does not feature 'advanced' constructs (e.g. classes, polymorphism. . .). Although not a very expressive one, it is a general purpose language and is not embedded in another language. This section details the implementation of a frontend that generates an equivalent LMS graph from any Miniscala program.

The Miniscala parser outputs an abstract syntax tree (AST) of the input program. The frontend then maps AST nodes to graph nodes. It also computes the dependencies (edges) of each node. The result is a graph representation of the program.

2.1 Intermediate representation implementation

Each node consists of an uniquely identifying symbol, a label for the type of statement it represents and a list of inputs that correspond to the statements arguments. A node input is either an expression (constant or symbol) or a block, which consists of a list of unbound input symbols, a result expression and an effect input symbol. Each node also has an effect summary, which carries information about the nodes dependencies, as well as which effects the node has. Graph 'edges' are represented by the nodes' effect summaries.

Blocks also have an effect summary, which carries information about the nested nodes' effects and is used for computing the effect summary of the node that contains the block. It also determines which nodes are used in the block. These nodes must be scheduled exactly in this block to preserve program semantics.

The effect summary of a new node depends on the other nodes in the current scope. The scope of a block represents the set of nodes that are inside that block, but outside any nested block. It contains the effect input symbol of the block, which provides an 'entry point' to the block. Each node's dependencies are restricted to symbols in the same scope. The current scope is represented by some metadata that keeps track of the current block, as well as some of its nodes, depending on the effect system (see 2.5).

2.2 Abstract syntax tree to graph

Constant nodes in the abstract syntax tree (AST) are mapped to constant expressions. For other AST nodes, the frontend creates a new graph node through the *reflect* operation and maps the AST node to the new node's identifying symbol. Children AST nodes are mapped to the new node's inputs.

Children nodes that can be computed independently from the parent node are mapped to 'toplevel' expressions and evaluated in the current scope. On the other hand, children nodes whose computation is directed by the parent node (functions, loops and conditional branches) are mapped to the contents of a new block and must be evaluated in a new nested scope. This is done by the *reify* operation.

2.3 Reflect and reify

The graph is generated through two main operations: *reflect* and *reify*.

The *reflect* operation creates a node and computes its effect summary according to the effect system, then adds the new node to the graph.

The *reify* operation takes two arguments: an arity and a function that, given *arity* input expressions, evaluates a tree in the current scope. Its purpose is to create a new block and reflect its contents in a new scope. It also computes the block's effect summary, which depends on the nested nodes' effects. Depending on the effect system, some metadata holds information about the effects of the nodes added to the graph so far. Before applying the block function to the inputs, this metadata is saved and reset to empty data. It is restored at the end of the reify operation. Therefore, the block's effect summary can be computed using this metadata, which contains precisely the information about the effects of the nodes that were created by the block function, i.e. the statements nested in the block. Because the metadata is restored afterwards, the individual statements nested inside the block do not change the outer scope's effect information; their effects will instead be associated with the node that contains the block.

2.4 Language restrictions

Since Miniscala is a somewhat restricted language, it is supported by relatively simple effect systems and a restricted frontend. In order to map a 'full' general purpose language to LMS, the frontend would likely need to be extended.

The frontend supports all Miniscala features (except the full type system) with metadata only. For example, mutually recursive functions are supported through a global map that assigns a placeholder symbol for every undefined function encountered, so that this symbol is assigned to the function when it is defined.

However, more advanced features may make it necessary to extend LMS intermediate representation itself to support a more expressive language.

2.5 Effect systems

The selected effect system determines which effect dependencies (edges) exist between nodes. Each effect system enforces the necessary dependencies to preserve program semantics. More fine-grained effect systems add fewer extra dependencies to allow for more flexibility in the scheduling algorithm.

```
def printf(s: String): Unit = (); // external

def echo(): Unit = printf("text");
def f(x: Int): Int = x - 5;

var i = 0;
var j = 1;
var p = 0;
while (i < 10) {
  j = j * 2;
  i = i + 1;
  p = 3
};
var a = j - 7;
var b = f(2);
j = b;
echo();
a
```

Linearized This effect system schedules each node in the sequential order of the program: every node depends on the previous node in the current scope, or on the current block's effect input if no other node exists in the scope.

Each block carries a dependency to the last node created in its scope; this ensures that every node in the scope is scheduled inside this block.

Effectful/pure This effect system distinguishes *pure* statements from *effectful* statements. Any statement that creates, reads or writes a mutable variable or that has a 'global' effect (such as input/output operations) is effectful.

Effectful statements are scheduled in sequential order, as in the linearized effect system. However, pure statements do not have any effect dependency and can be moved around freely. Each block carries a dependency to the last effectful node created in its scope, if it exists. As a result, all effectful nodes created in this scope will be scheduled in this block.

A node may also have a *latent effect*, if it has effectful block inputs or if it calls an effectful function. Functions' effect summaries are retrieved from a global cache that maps function symbols to associated definitions. Since these blocks are executed each time the node is executed, if one of them is effectful, then the node is effectful as well.

Effect keys Effect keys allow to distinguish different effects categories. A statement may be effectful for the following effect keys, possibly simultaneously:

- x where x is an existing mutable object, if it reads or writes x;
- *STORE* if it creates a new mutable object;
- *CTRL* if it has 'global' effects, such as printing.

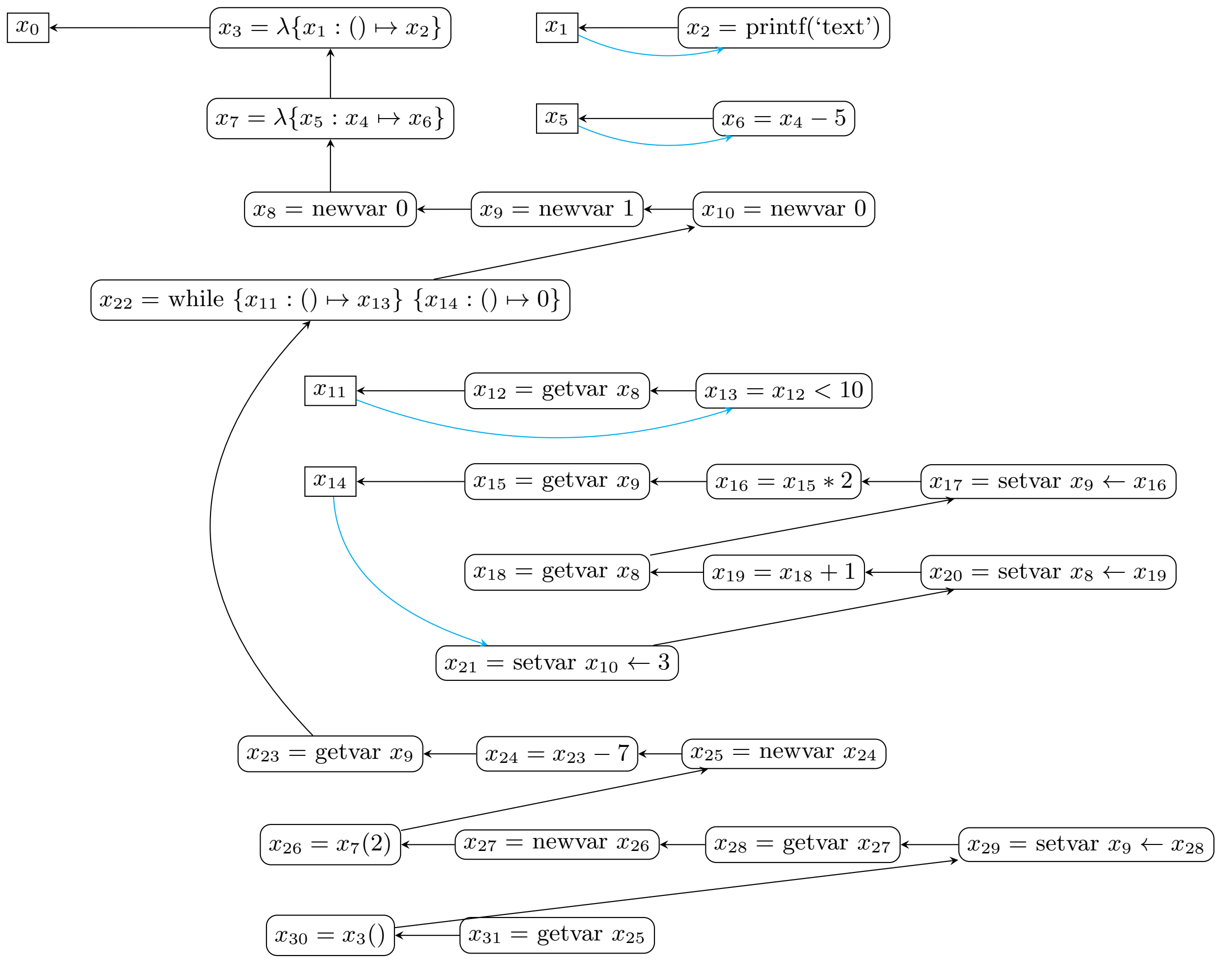
x_0
$x_3 = \lambda\{x_1 : () \mapsto x_2\}$
x_1
$x_2 = \text{printf('text')}$
$x_7 = \lambda\{x_5 : x_4 \mapsto x_6\}$
x_5
$x_6 = x_4 - 5$
$x_8 = \text{newvar } 0$
$x_9 = \text{newvar } 1$
$x_{10} = \text{newvar } 0$
$x_{22} = \text{while } \{x_{11} : () \mapsto x_{13}\} \ \{x_{14} : () \mapsto 0\}$
x_{11}
$x_{12} = \text{getvar } x_8$
$x_{13} = x_{12} < 10$
x_{14}
$x_{15} = \text{getvar } x_9$
$x_{16} = x_{15} * 2$
$x_{17} = \text{setvar } x_9 \leftarrow x_{16}$
$x_{18} = \text{getvar } x_8$
$x_{19} = x_{18} + 1$
$x_{20} = \text{setvar } x_8 \leftarrow x_{19}$
$x_{21} = \text{setvar } x_{10} \leftarrow 3$
$x_{23} = \text{getvar } x_9$
$x_{24} = x_{23} - 7$
$x_{25} = \text{newvar } x_{24}$
$x_{26} = x_7(2)$
$x_{27} = \text{newvar } x_{26}$
$x_{28} = \text{getvar } x_{27}$
$x_{29} = \text{setvar } x_9 \leftarrow x_{28}$
$x_{30} = x_3()$
$x_{31} = \text{getvar } x_{25}$

Fig. 3. Graph generated for 2.5

A node effectful for some set of effect keys depends on the last effectful node for each of these keys if it exists, or else on the current block's effect input.

A global set of local definitions keeps track of variables created in the current scope. Mutable variables created inside a block are removed from this block's effect keys, since these don't have any effect outside the block.

Read/write keys Effects are further divided into two categories: read and write effects. A node that writes to a key must be scheduled after the key's last write node. It also cannot be scheduled before any of the key's read nodes that were created after the last write. However, it does not have to be scheduled *after* these nodes, meaning that they can be removed from the scheduled without affecting the write node. The write node has a *soft dependency* on each of these nodes, as opposed to a *hard dependency* that doesn't allow the node to be scheduled if it depends on an unscheduled node.

A node that reads a key only depends on this key's last write node: as long as the value of a mutable object is unchanged, the order of its read-only accesses does not alter program semantics.

Each block carries hard dependencies to the last write node for each of its write keys.

CTRL and *STORE* keys are considered exclusively write and read keys, respectively. Indeed, memory allocations can be reordered without altering the program's semantics, which is not the case for global effects (e.g. printing).

3 Backend

This section details the implementation of a backend that compiles the generated graph to x86 assembly.

3.1 x86 code generation

The x86 code generator is built directly on top of the code motion algorithm. It generates one or more corresponding x86 statement(s) each time a new node is scheduled. Blocks are delimited by labels and jumps created by the parent node. Each function is mapped to its own sequence of statements.

At this point, mutable variables are not mapped to physical memory yet, and are instead represented by *virtual registers*. The output also contains placeholders for sequences of instructions that cannot be generated before assigning physical locations to virtual registers.

NoSep is inserted between two instructions that must follow each other with no other instruction in between in order to preserve semantics. For example, inserting a *NoSep* between a comparison and a conditional jump prevents register allocation from altering program behavior with extra instructions between them.

```
def x1() = {
  val x3 = printf "text"
  x3
}
var x8 = 0
var x9 = 1
var x10 = 0
x22 = while {
  val x12 = x8
  val x13 = x12 < 10
  x13
} {
  val x15 = x9
  val x16 = x15 * 2
  x9 = x16
  val x18 = x8
  val x19 = x18 + 1
  x8 = x19
  x10 = 3
  0
}
val x23 = x9
val x24 = x23 - 7
var x25 = x24
val x30 = x1()
val x31 = x25
x31
```

Fig. 4. Statement scheduling for 2.5

```
  x1:
  # SaveCalleeSave
  # ScopeBegin
  addq $8, %rsp
  subq $8, %rsp
  # SaveCallerSave
  movq $str0, %rdi
  movb $0, %al
  call printf
  # RestoreCallerSave
  movq %rax, x3
  movq x3, %rax
  # ScopeEnd
  # RestoreCalleeSave
  ret

  start:
  # ScopeBegin
  x0:
  movq $0, x8
  movq $1, x9
  movq $0, x10
  # ScopeBegin
  jmp x11
  x14:
    movq x9, x15
    movq x15, x16
    imulq $2, x16
    movq x16, x9
    movq x8, x18
    movq x18, x19
    addq $1, x19
    movq x19, x8
    movq $3, x10
  x11:
    movq x8, x12
    cmp $10, x12
    # NoSep
    setl %al
    movzbq %al, x13
  movq x13, x22
  cmp $0, x13
  # NoSep
  jnz x14
  # ScopeEnd
  movq x9, x23
  movq x23, x24
  subq $7, x24
  movq x24, x25
  # SaveCallerSave
  call x1
  # RestoreCallerSave
  movq %rax, x30
  movq x25, x31
movq x31, %rax
# ScopeEnd
ret
```

Fig. 5. Code generated for 2.5 with read/write effects

Caller-save registers The contents of some x86 registers, called *caller-save* registers, may be modified during a function call. Each caller-save register that contains a value must therefore be saved before calling any function and restored after the call. After register allocation, the *SaveCallerSave* (resp. *RestoreCallerSave*) placeholder is replaced with a sequence of *push* (resp. *pop*) instructions that save (resp. restore) each caller-save register used in the surrounding scope.

Callee-save registers Other x86 registers, called *callee-save* registers, may not be modified by a function call. Therefore, any of these registers used in a function must be saved at the beginning of the function and restored at the end. After register allocation, save and restore instructions replace *SaveCalleeSave* and *RestoreCalleeSave* placeholders.

Scope delimiters If all physical registers are already assigned to a virtual register and a new value needs to be mapped to a physical location, the register allocation algorithm may evict the contents of a register on the stack and use the register for the new value. This results in the insertion of a *push* instruction before the statement that writes the new value. During execution, the register will effectively be freed for the new value. However, if these instructions are executed again, the register's contents will be evicted again, even though no new value needs to be stored. As a result, the stack will grow to an unknown size and it will be impossible to restore it to its original state (without wasting a register as an accumulator). A better solution is to evict the register's contents before entering the innermost surrounding loop, and to restore them after exiting the loop. If no loop surrounds the instructions, the beginning and end of the surrounding function are used instead. The eviction (resp. restore) instructions for all evicted registers are represented by a *ScopeBegin* (resp. *ScopeEnd*) placeholder. Since every loop or function contains the same number (zero or one) of these placeholders and they both push or pop the same number of values, the stack is always restored to its initial size.

3.2 Register allocation

Virtual registers must then be mapped to physical memory locations. The register allocation algorithm used in this backend uses a ring buffer of registers as a cache for the top of the stack.

This algorithm is applied to each function individually. Physical registers that are already used in the function are excluded from the register buffer. Global tables keep track of the contents of the stack and the physical registers.

For each statement (or group of two statements separated by *NoSep*), virtual register operands are replaced with real operands. If the virtual register has already been assigned to a physical register or stack location, then it is replaced with the corresponding register or memory operand. Otherwise, it is assigned to the next register in the buffer. If the next register is not free, its contents need to be pushed onto the stack. The register is added to a list of physical registers that must be saved before the current loop and restored afterwards. The register allocation process then has to be restarted from the beginning of the current loop, since the location of the previous contents has changed.

Most two-operand x86 instructions need at least one register operand. If a statement ends up with no physical register operand, then some instructions are added before and after the statement to use the next register as a temporary location for one the operands, and to save and restore it if needed.

Each remaining placeholder is then replaced by a sequence of *push* or *pop* instructions for each register in the list of emptied registers.

Fig. 6. Code generated for 2.5 after register allocation

4 Evaluation

Miniscala does not support some abstract features such as classes and polymorphism. The frontend supports every Miniscala feature with extra metadata during graph generation (such as an undefined function table for mutually recursive functions). However, supporting a more expressive general purpose language may require to extend LMS intermediate representation itself, with new node types or more information attached to nodes.

Even though the Miniscala type system supports many other types, the frontend currently only supports integers, booleans and integer or boolean arrays. Floating-point numbers would likely require some type information to be added to LMS nodes, in order to generate floating-point x86 instructions for statements operating on floating-point values.

LMS intermediate representation already allows optimizations such as dead code elimination or common subexpression elimination to be applied to the graph. It may be possible to extend the frontend and the IR so that the graph contains control-flow information, in order to simplify optimization passes.

In the same way, some information (e.g. variable liveness ranges) could be added to the graph for use by the register allocation component of the backend, in order to reduce the amount of program analysis needed. More advanced register allocation algorithms, such as linear scan or graph coloring may improve generated program performance.

Experimental evaluation is needed to compare target program performance between different effect systems, register allocation algorithms, and between the backend and general purpose language compilers.

Conclusion

Staging is an approach that can simplify software development while maintaining optimal performance.

LMS provides a framework to use this approach in embedded domain-specific languages. The approach presented in this paper provides an interface to LMS for a restricted general purpose language.

Future work would allow to target more expressive languages, take advantage of more LMS features and evaluate target program performance.

References

1. N.D. Jones, C.K. Gomard, and P. Sestoft, Partial Evaluation and Automatic Program Generation. Prentice Hall International, June 1993. xii + 415 pages. ISBN 0-13-020249-5.
2. Tiark Rompf and Martin Odersky. 2012. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. Commun. ACM 55, 6 (2012), 121130.
3. Yoshihiko Futamura. 1971. Partial Evaluation of Computation Process - An approach to a Compiler-Compiler. Transactions of the Institute of Electronics and Communication Engineers of Japan 54-C, 8 (1971), 721–728.
4. Tiark Rompf. 2012. Lightweight Modular Staging and Embedded Compilers: Abstraction Without Regret for High-Level High-Performance Programming. Ph.D. Dissertation. EPFL.
5. Cliff Click and Michael Paleczny. A Simple Graph-Based Intermediate Representation. In Intermediate Representations Workshop, pages 3549, 1995.