LMS Code Walk

TIARK'S GROUP

Lightweight Modular Staging (LMS) is a generative programming tool achieving the multi-stage programming (staging). LMS_clean is a remake of the LMS project, aiming at a more flexible design and extension with better support for LMS IR transformation and optimization. This documentation is a code walk of the LMS_clean repo, hoping to explain the implementation (in high-level or in details) to people who are interested in learning and using LMS. This is different from a tutorial since it will dive into the core implementation of LMS and offers more insights than simply how to use LMS.

1 INTRODUCTION

Multi-Stage Programming (or staging) is a programming language concept that allows various parts of the programs to run in different stages. It allows users to code with high-level abstractions (trait, classes, high-order functions), but still gain highly-efficient code after the abstractions are executed (staged away). This offers both high productivity and high performance of the target program, thus the slogan "abstract without regret".

Lightweight Modular Staging (LMS) is a staging tool built in Scala. In LMS, type information is used to distinguish the evaluation stages (i.e., All Rep[T] typed values and expressions are in code generation for the next stage.) Simply speaking, LMS is a compiler. However, LMS does not have a laxer or parser to transform the input program into intermediate representations (IR). Instead, the IR is generated via executing the input program. All the Rep[T] typed expressions in the input program evaluate to LMS IR. This can be considered as the LMS frontend. Then the LMS backend compiles the LMS IR to target programs.

2 LMS IR

In the file core/backend.scala, object Backend, the core LMS IR is described.

```
abstract class Def // Definition: used in right-hand-side of all nodes
abstract class Exp extends Def
case class Sym(n: Int) extends Exp   // Symbol
case class Const(x: Any) extends Exp // Constant
case class Block(in: List[Sym], res: Exp, ein: Sym, eff: EffectSummary) extends Def
case class Node(n: Sym, op: String, rhs: List[Def], eff: EffectSummary)
```

2.1 Sea Of Nodes

The LMS IR follows the "sea of nodes" design (cite?), where the IR is composed of a list of `Nodes`, and the `Blocks` do not explicitly scope the nodes. Instead, the `Blocks` describe their *inputs* (via `in: List[Sym]`), *result* (via `res: Exp`), *world* (via `ein: Sym`), and *effects* (via `eff: EffectSummary`).

The *world* (by `ein: Sym`) is probably unfamiliar to many readers. It is used to scope the nodes that belong to this block. That is to say, if a node depends on the world of a block, then it should be scheduled within that block. For example:

Author's address: Tiark's Group.

XXXX-XXXX/2020/6-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

`Give an example?`

If a node depends on the inputs of a block, very likely it has to be scheduled in the block too. So both inputs and world are scoping the beginning of the block. That is why the world is also called the "effect input". So if the world marks the beginning of a block, which component of the Block marks the end of a block? The answer is the result and the effects of the block. When scheduling the nodes for a block, we can start from the result and effects, and pick nodes that are depended on by them.

Why using the sea of node IR in LMS? One of the key reasons is that it offers easy optimization of code-motion. Since the blocks do not explicitly scope the nodes, whether a node is scheduled in or out of a block is flexibly determined based on effects, using frequency, and et al. It does make IR traversal and transformation different from IRs with explicit scopes, which we will talk about in core/traversal.scala.

2.2 Effect Summary

In Section 2.1, we talked about the dependencies between nodes and blocks (via EffectSummary). EffectSummary can be course-grained or fine-grained. More fine-grained effect summary will have higher compilation time cost, but provide more optimized target program.

First of all, we want to be sure of the two intertwined concepts here: effects and dependencies. Effects refer to node behaviors such as printing, variable read, variable write, et al. Dependencies refer to both data dependencies and dependencies caused by effects, such as printing cannot be missed or reordered, and we cannot reorder two writes on the same variable.

The general work flow is that at node constructions, the effects of nodes are tracked. using the node effects we compute the dependencies. The dependencies are then used to schedule nodes for each block. Our current EffectSummary is like this:

```
case class EffectSummary(sdeps: Set[Sym], hdeps: Set[Sym], rkeys: Set[Exp], wkeys: Set[Exp])
```

It tracks soft dependencies (via `sdeps: Set[Sym]`), hard dependencies (via `hdeps: Set[Sym]`), read keys (via `Set[Exp]`), and write keys (via `Set[Exp]`). Hard dependencies are the normal dependencies. If node A hard-depends on node B, then scheduling A ensures that B is scheduled before A. Soft dependencies are soft in the sense that, if node A soft-depends on node B, node B cannot be scheduled after A. However, scheduling A does not ensure that B is scheduled. Read keys and write keys are easy to understand: they track `Exp`s that are read or written to by the node or block. However, the read keys and write keys can also be `Const`, indicating that they have other semantics related to other types of effects. For instance, printing nodes have write effect on `Const("CTLR")`, which ensures that printings are all scheduled with the order unchanged. Allocating variables/arrays have read effects on `Const("Store")`.

3 LMS GRAPH

In this section, we will talk about how LMS Graph are constructed using the LMS IR components. It shows how the LMS IRs are used in constructing LMS Graphs, and how effects and dependencies are tracked and generated. All LMS snippets are function. As the result, all LMS Graph have a list of nodes (already in topological order) and a block describing the function. That is captured by the

```
case class Graph(val nodes: Seq[Node], val block: Block, val globalDefsCache: immutable.Map[Sym,No
```

at core/backend.scala. The LMS Graph is constructed by `class GraphBuilder` at core/backend.scala.

Besides the basic functionality of storing nodes, searching nodes by symbols, and generating fresh symbols, GraphBuilder offers two keys functionalities

(1) Building nodes by the `reflect*` family of methods.

(2) Building blocks by the `reify*` family of methods.

Note that the `reify*` family of methods not only generate the Block object, but also the nodes that are used in the block. However, the nodes are not explicitly scoped in the block, but rather implicitly scoped via effect summaries. This implicit scoping allows flexible code motion as long as effects and dependencies are respected.

3.1 High-level Design of the Effect System

3.2 Code Walk: reflectEffect

The core method in the `reflect*` family of methods is the reflectEffect method.

```
def reflectEffect(s: String, as: Def*)(readEfKeys: Exp*)(writeEfKeys: Exp*): Exp
```

3.3 Code Walk: reify

4 SIMPLE FRONTEND

With a proper definition of LMS IR and the facility to build LMS Graph in core/backend.scala, we can build a frontend that construct LMS Graph. The LMS frontend should feature the Rep[T] type, which allows the staged programs to be type checked. However, in this section, we are going to introduce a very simple frontend that can just nit the LMS Graph, without the iconic Rep[T] type. This simple frontend is not of much use in production, but it shows the simple essence of LMS front, i.e., being able to construct LMS Graph with various control flows.

The basic ways to construct LMS graphs are through the `reflect*` and `reify*` family of functions. However, the frontend should allow the users to construct LMS graphs via unary and binary operations, `If`, `While`, `FUN`, and et al. That is the main purpose of the simple FrontEnd class.

```
class FrontEnd {

    var g: GraphBuilder = null // LMS graph is built in here

    case class BOOL(x: Exp) // boolean wrapper of LMS EXP, supporting ! op
    case class INT(x: Exp)  // int-like wrapper of LMS EXP, supporting arithmetic op
    case class ARRAY(x: Exp) // array wrapper of LMS Exp, supporting array access.
    case class VAR(x: Exp) // variable wrapper of LMS Exp, supporting variables

    // supporting conditional. a and b are executed in reify*.
    // then the returned blocks are used in reflect* to build the conditional node
    def IF(c: BOOL)(a: => INT)(b: => INT): INT = {...}

    // supporting loop. both c and b are executed in reify*,
    // then the returned blocks are used in reflect* to build the loop node
    def WHILE(c: => BOOL)(b: => Unit): Unit = {...}

    // supporting application. just creating an app node.
    def APP(f: Exp, x: INT): INT = INT(g.reflect("@", f, x.x))

    // If we just need to support in-graph, non-recursive functions, this is enough.
    // It builds a lambda node with the `f' reified into the block,
    // then it returns a scala function that can be applied to create APP construct
    def FUN(f: INT => INT): INT => INT = {
      val fn = INT(g.reflect("lambda", g.reify {xn: Exp => f(INT(xn)).x} ) )
      (x: INT) => APP(fn.x, x)
    }
}
```

It would be interesting to find out how to achieve in-graph, recursive functions. If we want to support recursive functions, we have to create a lambda node that uses the lambda Exp within the lambda block.

```scala
val fn = Sym(g.fresh)
g.reflect(fn, "lambda", g.reify(???))
```

In order to be able to use the same `fn` in the block of the lambda, we need to be able to construct `APP(fn, xn)` within the `g.reify`.

```scala
val f1 = (x: INT) => APP(fn, x)
```

But we don't know how the `f1` is recursively used. That has to come from user code `f`.

```scala
def FUN(f) = {
    ...
    g.reflect(fn,"lambda",g.reify(xn: Exp => f(f1, INT(xn)).x))()
}
// user code `f' decides how `f1' (the in-graph lambda) is recursively used.
```

To put everything together and offer a non-recursive API:

```scala
def FUN(f: INT => INT): INT => INT = FUN((_,x) => f(x))

def FUN(f: ((INT=>INT),INT) => INT): INT => INT = {
    val fn = Sym(g.fresh)
    val f1 = (x: INT) => APP(fn,x)
    g.reflect(fn,"lambda",g.reify(xn => f(f1,INT(xn)).x))()
    f1
}
```

A use case might be

```scala
val fac = FUN { (f, n) =>
        IF (n !== 0) {
        n * f(n-1) // recursive call
    } {
        1
    }
}
```

Finally, the `program` function reifies the user provided snippet (of type INT => INT) and returns the LMS Graph.

5 NORMAL FRONTEND

After discussing the simple frontend in Section 4, we want to show what a normal frontend looks like (in the object Adaptor at core/stub.scala).

object Adapter extends Frontend typeMap funTable

emitCommon // code gen ???

class MyGraphBuilder extends GraphBuilder

Base EmbeddedControls (macro and virtualization) OverloadHack (for hacky overload) ClosureCompare (compare LMS closures)

5.1 Base: Introducing Rep[T]

In the Base trait, the code establish the iconic Rep[T] of LMS. Previously in Frontend class, we have seen one way to wrap around core.backend.Exp so that we can construct LMS Graph via unary operators, binary operators, and et al. What is to be further provided in Base trait is the ability to use Rep[T]. Similarly Rep[T] is built on top of core.backend.Exp. The core.backend.Exp do not have types. The types are added via user code and type inferencing, and then tracked in a data structure called typeMap (such as Adaptor.typeMap).

```scala
trait Base extends EmbeddedControls with OverloadHack with ClosureCompare {
    type Rep[+T] = Exp[T]  // type name aliasing

    abstract class Exp[+T] // track LMS IR for non-variables
    abstract class Var[T]  // track LMS IR for variables
```

```
    // The Wrap class and method that build Rep[T] typed expression with type tracking
    case class Wrap[+A:Manifest](x: lms.core.Backend.Exp) extends Exp[A] {
        Adapter.typeMap(x) = manifest[A]
    }
    def Wrap[A;Manifest](x: lms.core.Backend.Exp): Exp[A] = {
        if (manifest[A] == manifest[Unit]) Const(()).asInstanceOf[Exp[A]]
        else new Wrap[A](x)
    }

    // The WrapV class for Var[T]
    case class WrapV[A:Manifest](x: lms.core.Backend.Exp) extends Var[A] {
        Adapter.typeMap(x) = manifest[A]
    }
}
```

5.2 Base: Better Handling of Functions

In the simple frontend, we see that the handling of recursive functions is a bit awkward. How do we make it better with a better frontend? In Base trait, we express the type of in-graph function better:

```
def fun[A:Manifest, B:Manifest](f: Rep[A]=>Rep[B]): Rep[A=>B] =
    Wrap[A=>B](__fun(f, 1, xn => Unwrap(f(Wrap[A](xn(0))))))
```

Unfortunately, we have to implement multiple `fun`s for different function arities, which we will elide here. The `__fun` function reifies the argument into a function block.

```
def __fun[T:Manifest](f: AnyRef, arity: Int, gf: List[Backend.Exp] => Backend.Exp): Backend.Exp =
    // use canonicalize to get the unique representation of any Scala function
    val can = canonicalize(f)
    Adapter.funTable.find(_._2 == can) match {
        case Some((funSym, _)) =>
            funSym // Easy case: found the function in funTable
        case _ =>
            // Step 1. set up "lambdaforward" node with 2 new fresh Syms
            val fn = Backend.Sym(Adapter.g.fresh)
            val fn1 = Backend.Sym(Adapter.g.fresh)
            Adapter.g.reflect(fn, "lambdaforward", fn1)()

            // Step 2. register (fn, can) in funTable, so that recursive calls
            //    will find fn as the function Sym. Reify the block.
            // Note: it might seem strange why/how recursive calls re-enter this __fun() function.
            //    The reason is that in user code, recursive functions have to be written as
            //    lazy val f = fun{...} or def f = fun{...}, in which case the recursive calls
            //    will re-enter the `fun` call.
            Adapter.funTable = (fn, can)::Adapter.funTable
            val block = Adapter.g.reify(arity, gf)

            // Step 3. build the "lambda" node with fn1 as the function name
            //    fix the funTable such that it pairs (fn1, can) for non-recursive uses.
            val res = Adapter.g.reflect(fn1,"lambda",block)(hardSummary(fn))
            Adapter.funTable = Adapter.funTable.map {
            case (fn2, can2) => if (can == can2) (fn1, can) else (fn2, can2)
            }
            res
    }
}
```