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 `Node`s, and the `Block`s do not explicitly scope the nodes. Instead, the `Block`s describe their *inputs* (via `in: List[Sym]`), *result* (via `res: Exp`), *input-effect* (via `ein: Sym`), and *effects* (via `eff: EffectSummary`).

Using "sea of nodes" has perfound implications to LMS IR. The advantage is that the scopes of nodes are determined dynamically based on correctness and using frequency, which allows easy

Author's address: Tiark's Group.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2020 Association for Computing Machinery.
XXXX-XXXX/2020/6-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

code-motion and optimization. However, using "sea of nodes" means that we must have a way to dynamically determine what nodes are in each block. We compute that based on *dependencies*. It does make IR traversal and transformation different from IRs with explicit scopes, which we will talk about in core/traversal.scala.

Dependencies controls how nodes are scheduled. There are some simple rules about it. If node A depends on node B, then scheduling A means B must be scheduled before A. If a block's results or effects depend on node A, then A must be schedule within or before this block. If a node depends on the input-effect or the inputs of a block, then the node is bound by this block (i.e., it must be scheduled within this block or an inner block).

Here is one example of scala snippet:

```
def snippet(x: Int, y: Int) = {
  var idx = 0
  var agg = 0
  while (idx < x) {
    val p = y * y
    agg = agg + p * idx
    idx += 1
  }
  agg
}
```

If we determine that every statement from this snippet depends on its previous statement, we can have a list of nodes and blocks with the following dependencies.

```
Graph(nodes, Block([x0, x1], x19, x2, [dep: x19]))

nodes = [
  x3 = "var_new" 0    [dep: x2]   // idx
  x4 = "var_new" 0    [dep: x3]   // agg

  x18 = "W" Block([], x7, x5, [dep: x7])
            Block([], (), x8, [dep: x17]) [dep: x4]

    x6 = "var_get" x3       [dep: x5]
    x7 = "<" x6 x0          [dep: x6]

    x9 = "*" x1 x1          [dep: x8]
    x10 = "var_get" x4      [dep: x9]
    x11 = "var_get" x3      [dep: x10]
    x12 = "*" x9 x11        [dep: x11]
    x13 = "+" x10 x12       [dep: x12]
    x14 = "var_set" x4 x13  [dep: x13]
    x15 = "var_get" x3      [dep: x14]
    x16 = "+" x15 1         [dep: x15]
    x17 = "var_set" x3 x16  [dep: x16]

  x19 = "var_get" x4  [dep: x18]
]
```

In this example, we first have two nodes that declare to new variables (x3 and x4). Then we have a while loop (x18) that has two blocks (the condition block and the loop body block). The condition block has input-effect x5 and result x7. So it scopes node x6 and x7 into the condition block. The loop body block has input-effect x8 and effects x17, so it scopes node x9 to x17 in the loop body block. The while loop node depends on x4, and the final node x19 depends on x18. Thus a linear ordering of all nodes with two blocks is fully determined by the dependencies.

2.2 Effects in Categories

In Section 2.1, we talked about the dependencies between nodes and blocks (via EffectSummary). In that example, the effect summary simply stores the depdendencies of the nodes. That is a very coarse-grained effect summary, and more fine-grained effect summaries can offer more optimization and flexible code motions. But before going into the details of the fine-grained effect summary, let's first get clear of two related and intertwined concepts: 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.

To make our effect summary more fine-grained, the first step is to realize that there are different kinds of effects. The effects of printing is clearly not so related to the effect of variable reading, thus there should be no scheduling enforcement between them. We chose to support the following categories of effects:

(1) Statements that create mutable objects belong to the category keyed by STORE.
(2) Statements that read/write a mutable object belong to the category keyed by the symbol of this object (result of an allocation node).
(3) For all remaining effectful statements (such as printf), they belong to the category keyed by CTRL (for control flow dependent).

2.2.1 Latent Effects. Effects can be latent, such as the effects of a function block. The function block may have printing statements and variable reads/writes, but they are not happening until they are called in function applications. Tracking latent effects can be tricky if we allow first-order functions, where functions can be returned from other functions and control flows, passed in as parameters, and stored in data structures such as arrays, lists, and et al.

2.2.2 Aliasing and Borrowing. Handling the read and write effects of data structures can be really tricky too, when considering aliasing and borrowing.

```
// aliasing
val arr = new Array[Int](10)
val arr2 = arr.slice(0, 5)
arr2(0) = 5
printf("%d\n", arr(0))

// borrowing
val arr = new Array[Int](10)
var arr2 = arr
arr2(0) = 5
printf("%d\n", arr(0))
```

These are currently unsolved problems in LMS clean :).

2.2.3 From Effects to Dependencies. After collection read and write effects, we need to compute dependencies from them. The rules for computing dependencies from effects are listed below:

(1) Read-After-Write (RAW): there should be a hard dependency from R to W, since the correctness of the R depends on the W to be scheduled)
(2) Write-After-Read (WAR): there should be a soft dependency from W to R, since the correctness of the W does not depend on the R to be scheduled, but the order of them matters.
(3) Write-After-Write (WAW): the first idea is to generate a soft dependency, since the second W simply overwrite the first W. However, write to array is more complicated, such as arr(0)

= 1; arr(1) = 2, where the second W doesn't overwrite the first W, and both Ws have to be generated. For now, we just issue a hard dependency from the second W to the first W.

(4) Read-After-Read (RAR): there is no effect dependency between them.

Note that we introduced soft dependencies in the rules. 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.

2.2.4 Const Effects. The STORE and CTRL keys are also handled in our read/write system in a case-by-case manner. For instances, allocating heap memory can be modeled as a read on the STORE key, if we don't care about the order of heap allocation. However, printf should be modeled as write on the CTRL key, since all prints should be scheduled in the same order. Some trickier cases include getting a random number from a pseudo-random generator. It really depends on the expected behavior.

More details about effect system can be found and Gregory's write up: https://hackmd.io/_-VGqPBiR3qToam7YTDdRw?view.

3 BUILDING EFFECTSUMMARY IN LMS GRAPH

Our fine-grained current EffectSummary is like below. 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]`).

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

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.

The core reflect method is defined as below (with some simplification):

```
def reflectEffect(s: String, as: Def*)(readEfKeys: Exp*)(writeEfKeys: Exp*): Exp = {
    // simple pre-construction optimization
    rewrite(s, as.toList) match {
      case Some(e) => e // found optimization (resulting in pure expressions only)
      case None => // no available optimization
        val (latent_ref, latent_wef) = getLatentEffect(s, as:_*)
        val (reads, writes) = ((latent_ref ++ readEfKeys).toSet, (latent_wef ++ writeEfKeys).toSet

        if (reads.nonEmpty || writes.nonEmpty) {
            // build node with the help of `gatherEffectDeps'
            val (prevHard, prevSoft) = gatherEffectDeps(reads, writes, s, as:_*)
            val summary = EffectSummary(prevSoft, prevHard, reads, writes)
            val res = reflect(Sym(fresh), s, as:_*)(summary)

            // update effect environments (curEffects, curLocalReads, and curLocalWrites)
            curLocalReads ++= reads
            curLocalWrites ++= writes
            for (key <- reads) {
              val (lw, lrs) = curEffects.getOrElse(key, (curBlock, Nil))
```

```
              curEffects += key -> (lw, res::lrs)
              if (key == STORE) curEffects += res -> (res, Nil)
            }
            for (key <- writes) { curEffects += key -> (res, Nil) }
            res
        } else {
            // We can run Common Subexpression Elimination (CSE) for pure nodes
            findDefinition(s,as) match {
                case Some(n) => n.n
                case None =>
                reflect(Sym(fresh), s, as:_*)()
            }
        }
    }
}
```

This method first tries to run pre-construction optimization via `rewrite`. The optimized result is a pure Exp that can be returned directly. If not optimized, the method computes the latent effects via `getLatentEffect` helper method, and then combine the results with the user provided `readEfkeys` and `writeEfKeys`. If the effect keys are empty, the methods go to the else branch and try Common Subexpression Elimination (CSE) via `findDefinition` helper method. Otherwise, the method compute dependencies from the effect keys via `gatherEffectDeps` method and then creates the node. The last thing to do is updating the effect environments, including `curEffects`, `curLocalReads`, and `curLocalWrites`. The `curEffects` is a map of type: Exp -> (Sym, List[Sym]), which tracks the last write and the reads after last write for each Exp key.

The getLatentEffect family of methods are like below. They essentially recursively call each other to dig out all latent effects of nodes.

```
def getLatentEffect(op: String, xs: Def*): (Set[Exp], Set[Exp]) = (op, xs) match {
    case ("lambda", _) => (Set[Exp](), Set[Exp]()) // no latent effect for function declaration
    case ("@", (f: Sym)+:args) => getApplyLatentEffect(f, args:_*)._1
    case _ => getLatentEffect(xs:_*)
}
def getApplyLatentEffect(f: Sym, args: Def*): ((Set[Exp], Set[Exp]), Option[Exp]) = {
    // Just collecting the latent effects of arguments
    val (reads, writes) = getLatentEffect(args: _*)

    // the freads/fwrites are read/write keys of the function (excluding parameters)
    // the preads/pwrites are read/write keys of the function parameters (they are Set[Int] as ind
    // the res is the result of the function body. It is needed because the result of the body can
    //     we need to get the latent effects of.
    val ((freads, fwrites), (preads, pwrites), res) = getFunctionLatentEffect(f)

    // For @ we need to replace the effect on parameters to the actual arguments.
    // the asInstanceOf seems unsafe at first glance. However, it is not a problem since a standal
    // should never be an argument in function application.
    ((reads ++ freads ++ preads.map(args(_).asInstanceOf[Exp]), writes ++ fwrites ++ pwrites.map(a
}
def getFunctionLatentEffect(f: Exp): ((Set[Exp], Set[Exp]),(Set[Int], Set[Int]), Option[Exp]) = fi
      case Some(Node(_, "lambda", List(b:Block), _)) =>
        getEffKeysWithParam(b)
      case Some(Node(_, "lambdaforward", _, _)) => // what about doubly recursive?
        ((Set[Exp](), Set[Exp](Const("CTRL"))), (Set[Int](), Set[Int]()), None)
      case None => // FIXME: function argument? fac-01 test used for recursive function...
        ((Set[Exp](), Set[Exp](Const("CTRL"))), (Set[Int](), Set[Int]()), None)
      case Some(Node(_, "@", (f: Sym)+:args, _)) =>
        val ((rk, wk), Some(f_res)) = getApplyLatentEffect(f, args: _*)
        val ((rk2, wk2), (prk2, pwk2), f_res_res) = getFunctionLatentEffect(f_res)
        ((rk ++ rk2, wk ++ wk2), (prk2, pwk2), f_res_res)
      case Some(Node(_, "?", c::Block(ins, out, ein, eout)::Block(ins2, out2, ein2, eout2)::Nil, _
```

```
        val ((rk, wk), (prk, pwk), _) = getFunctionLatentEffect(out)
        val ((rk2, wk2), (prk2, pwk2), _) = getFunctionLatentEffect(out2)
        ((rk ++ rk2, wk ++ wk2), (prk ++ prk2, pwk ++ pwk2), None) // FIXME(feiw)
      case Some(e) => ???
}
def getLatentEffect(xs: Def*): (Set[Exp], Set[Exp]) =
    xs.foldLeft((Set[Exp](), Set[Exp]())) { case ((r, w), x) =>
      val (ref, wef) = getLatentEffect(x)
      (r ++ ref, w ++ wef)
    }
def getLatentEffect(x: Def): (Set[Exp], Set[Exp]) = x match {
    case b: Block => getEffKeys(b)
    case s: Sym => findDefinition(s) match {
        case Some(Node(_, "lambda", (b@Block(ins, out, ein, eout))::_, _)) => getEffKeys(b)
        case _ => (Set[Exp](), Set[Exp]())
    }
    case _ => (Set[Exp](), Set[Exp]())
}
```

However, the complex code here is not yet fully correct. There are several issues:

(1) We have a specical function (getApplyLatentEffect) to dig out latent effects of functions (since they are applied via "@" syntax). However, the LMS IR may have other syntax to apply a function, such as ("map", List(array, f)). It is still not clear when to dig latent effects out and when to not.
(2) Depending on whether we want to support first-order functions, the functions may be wrapped in complex data structures (as array element) or returned from other functions or conditionals. The getFunctionLatentEffect function might have too many cases to check and too many branches to consider, which is both expensive and inaccurate.
(3) The read and write effects on data structures are currently at the most coarse granularity (for the whole data structure). Also the aliasing and borrowing effects are not yet considered.

Potential solutions are listed here:

(1) be conservative and cheap at some cases with a "stop the world" effect
(2) track aliasing with some frontend constraint (like rust, separation logic, regions)

The gatherEffectDeps method computes dependencies from effect keys:

```
def gatherEffectDeps(reads: Set[Exp], writes: Set[Exp], s: String, as: Def*): (Set[Sym], Set[Sym])
    val (prevHard, prevSoft) = (new mutable.ListBuffer[Sym], new mutable.ListBuffer[Sym])
    // gather effect dependencies 1): handle the write keys
    for (key <- writes) {
      curEffects.get(key) match {
        case Some((lw, lr)) =>
          val (sdeps, hdeps) = gatherEffectDepsWrite(s, as, lw, lr)
          prevSoft ++= sdeps; prevHard ++= hdeps
        case _ =>
          // write has hard dependencies on declaration (if declared locally) or block (if declare
          prevHard += latest(key);
      }
    }
    // gather effect dependencies 2): handling of reifyHere
    // reifyHere is an Effect Optimization for conditionals (if and switch)
    // it allows the block of conditionals to be aware of the `curEffects` out of the block
    // The exact demonstration of the optimization is in test IfDCETest "if_effect_reifyHere".
    if (reifyHere) prevHard += curBlock
    // gather effect dependencies 3): handle read keys (i.e., reads have hard dependencies on prev
    for (key <- reads) {
      prevHard += getLastWrite(key)
    }
    (prevHard.toSet, prevSoft.toSet)
```

```
}
```

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. The withBlockScopedEnv method is the subroutine that saves old effect environments.

```
def reify(arity: Int, f: List[Exp] => Exp, here: Boolean = false): Block =
withBlockScopedEnv(here){
    val args = (0 until arity).toList.map(_ => Sym(fresh))
    val res = f(args)
    // remove local definitions from visible effect keys
    val reads = curLocalReads.filterNot(curLocalDefs)
    val writes = curLocalWrites.filterNot(curLocalDefs)
    var hard = writes.map(curEffects.map(_)._1)
    if (curEffects.map contains res) // if res is a local mutable (e.g. Array)
      hard += curEffects.map(res)._1
    if (hard.isEmpty)
      hard = Set(curBlock)

    Block(args, res, curBlock, EffectSummary(Set(), hard, reads, writes))
}
```

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.

```
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`.

```
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`.

```
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:

```
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

```
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 **??**, 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

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).

```
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
    }
}
```

Although the __fun function provided a nice solution to recursive functions, it does add complexity to code generations and LMS graph transformations, which we will cover later. The Base trait also provides topFun, which is a variant of fun that is supposed to be lifted to top level (for C code generation). There are still many limitations to topFun related to closure conversion, recursion, and so on.

The Base trait also provide macro support for native if, while, and var, by extending the EmbeddedControls and providing the following methods:

```
def __ifThenElse[T:Manifest](c: Rep[Boolean], a: => Rep[T], b: => Rep[T]): Rep[T]
def __whileDo(c: => Rep[Boolean], b: => Rep[Unit]): Rep[Unit]
def var_new[T:Manifest](x: Rep[T]): Var[T]
def __assign[T:Manifest](lhs: Var[T], rhs: T): Unit
```

With the macro (@virtualize), native if, while, and var are syntactically transformed to these functions, which are then converted to IF, WHILE, and WrapV[T].

The Base trait also support misc ops, boolean ops, timing ops, comment ops, unchecked ops, and et al.

5.3 Other Rep[T] Handling

Since a Type T object can have many methods, we need to support those methods for Type Rep[T] objects. The way is to implement

(1) implicit conversion from T, Rep[T], and Var[T] to a class (TOpsCls) wrapping Rep[T].
(2) all the desired methods in that class (TOpsCls).

An example of ordering is given below. All these traits have to extend Base or be implemented in Base trait directly, to be able to use Rep[T]. This pattern of code is used for many types that are supported in LMS IR as Rep[T], including primitives (such as Int, Float, and Double) and some data structures (such as List, Tuple, Array, and Map)

```
trait OrderingOps extends Base with OverloadHack {
  implicit def orderingToOrderingOps[T:Ordering:Manifest](n: T) = new OrderingOpsCls(unit(n))
  implicit def repOrderingToOrderingOps[T:Ordering:Manifest](n: Rep[T]) = new OrderingOpsCls(n)
  implicit def varOrderingToOrderingOps[T:Ordering:Manifest](n: Var[T]) = new OrderingOpsCls(readV

  class OrderingOpsCls[T:Ordering:Manifest](lhs: Rep[T]){
    def <  (rhs: Rep[T])(implicit pos: SourceContext) =
      Wrap[Boolean](Adapter.g.reflect("<", Unwrap(lhs), Unwrap(rhs)))
    def <= (rhs: Rep[T])(implicit pos: SourceContext) =
      Wrap[Boolean](Adapter.g.reflect("<=", Unwrap(lhs), Unwrap(rhs)))
  }
```