Halvor Bjørstad

Implementing RVSDG as a dialect of MLIR

Master's thesis in Computer Science (MTDT)
Supervisor: Magnus Själander
Co-supervisor: David Metz
June 2023

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Halvor Bjørstad

Implementing RVSDG as a dialect of MLIR

Master's thesis in Computer Science (MTDT)
Supervisor: Magnus Själander
Co-supervisor: David Metz
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Sammendrag/Abstract

Norsk
Slutten på Dennard-skaleringen og det påfølgende skiftet til flerkjernede og heterogene prosessorarkitekturer har flyttet ansvaret for å øke ytelsen til programvare over på programmerere og verktøyene deres. Et behov for kompilatorer som automatisk kan utnytte den iboende parallelliteten i programmer og den spesialiserte maskinvaren som man finner i moderne prosessorer har derfor blitt mye viktigere. For å kunne gjennomføre slike optimaliseringer, er det viktig at kompilatoren bruker en mellomrepresentasjon (IR) som gjør det mulig å finne ut hvilken kode som kan parallelliseres.

Den regionaliserte verdi-/tilstands-avhengighetsgrafen (RVSDG) er en dataflytbasert IR. Den representerer beregninger som noder og verdi- og tilstandsavhengigheter som kanter mellom nodene. Enkelte kontrollflytstrukturer er eksplisitt representert ved hjelp av noder med regioner, som igjen inneholder subgrafer. Denne programrepresentasjonen eksponerer parallelliteten til programmer og forenkler dataflytbaserte optimaliseringer.

MLIR er et rammeverk for å implementere domenespesifikke IR-er. Det inneholder standard infrastruktur og er grunnlaget for et økosystem med IR-dialekter som modellerer forskjellige domener på forskjellige abstraksjonsnivåer. Disse dialektene kan sameksistere i en gitt IR, noe som gjør det mulig å representere tradisjonell kode og akseleratorspesifikk kode i samme datastruktur.

Denne oppgaven viser at RVSDG kan representeres som en dialekt av MLIR. Dette gjøres både ved å presentere en teoretisk representasjon av RVSDG i MLIR, og ved å implementere en RVSDG-dialekt i MLIR. Denne dialekten vil gjøre RVSDG mer tilgjengelig for eksterne kompilatorprosjekter og vil samtidig gi fremtidig forskning på RVSDG tilgang til infrastrukturen og økosystemet tilbudt av MLIR.

English

The end of Dennard scaling and subsequent rise of multicore and heterogeneous architectures has shifted the burden of improving program performance over to programmers and their tools. In particular, creating compilers which can exploit the latent parallelism of programs has become far more important. For such optimizations to be possible, the compiler needs an intermediate representation (IR) that makes it possible to detect code that is eligible for parallelization.

The Regionalized Value State Dependence Graph (RVSDG) is a data-flow centric IR that encodes a program as a graph of nodes, representing computations, connected by edges, representing data or state dependencies. Control flow constructs are represented as nodes containing regions that hold IR subgraphs. Representing a program in this manner exposes the inherent parallelism of the program and simplifies data-flow based compiler optimizations.

MLIR is a framework for creating domain specific IRs. It provides common infrastructure as well as an ecosystem of IR dialects that each model their own domain and/or level of abstraction. These dialects are allowed to coexist in the IR, enabling progressive lowering of code and the unified representation of traditional code and accelerator-specific code.

This thesis shows empirically that RVSDG can be implemented as a dialect of MLIR, and presents both the conceptual mapping from RVSDG to MLIR and the implementation of the RVSDG dialect. Having this dialect available makes RVSDG more accessible to other compiler developers, and also gives future research into RVSDG access to the infrastructure of MLIR.

Contents

List of Figures

List of Tables

CHAPTER 1
Introduction

The end of Dennard scaling [1] brought with it the end of frequency scaling, and caused single-threaded performance to stagnate [2]. This has caused processor designs to shift more and more towards multicore and heterogeneous architectures to better exploit decreasing transistor sizes without blowing their power budget. Program performance based on such architectural improvements are, however, not as easily exploited as a simple processor frequency increase. It is up to the programmers to create programs that take full advantage of and scale with modern systems. Since this is not always a simple task, and different processors may have completely different architectures, the creation of tools that can aid programmers in exposing parallelism and exploiting specialized hardware is of vital importance.

MLIR (Multi Level Intermediate Representation) is a framework for implementing domain specific compiler Intermediate Representations (IRs) [3]. It provides out-of-the-box infrastructure such as pass management, IR verification, and IR serialization in addition to a framework for defining operations and data types. MLIR also introduces the concept of IR dialects. These are logical groupings of operations, attributes, and types which model a specific domain. In MLIR different dialects are allowed to coexist at all levels of the IR, enabling progressive lowering from high level IR to target code. Coexisting dialects also makes it possible for MLIR to model IR for heterogeneous architectures by allowing accelerator-specific code to coexist with a more traditional IR. Additionally, progressive lowering gives the compiler more opportunities to discover optimizations [3]. MLIR was originally created as an IR framework for machine learning tools, but it was eventually realized that it could have applications in the wider industry [3].

The Regionalized Value State Dependence Graph (RVSDG) presented by Bahmann et al. [4] and later expanded on by Reissmann et al. [5] is an IR structure that seeks to model programs at all levels of abstraction as a unified data-flow based data structure. RVSDG has been shown to have no intrinsic limitations to the control flow it can represent [4] and can be constructed from more traditional Control Flow Graphs (CFGs) in a reasonable amount of time [4, 5]. Reissmann et al. also provided an implementation of the IR in the form of the JLM research compiler [5, 6] to prove the feasibility for using RVSDG as an IR for optimizing compilers. The RVSDG models programs as demand-dependence graphs, which should both expose the latent parallelism of programs and make it possible to create simple but powerful data-flow optimization passes on the IR [5].

This thesis seeks to find out if the RVSDG IR can be implemented as a dialect of MLIR, and if so, how good of a fit they are for each other. An empirical approach was taken to answer these questions, and an MLIR dialect was created to model RVSDG. There are several reasons for wanting RVSDG implemented in MLIR. The first is to increase the accessibility of RVSDG for compiler developers. While JLM [5] provides a good proof of concept, its tight coupling to LLVM IR and its proprietary infrastructure makes it unsuitable as a basis for future, more high level, compiler projects. The second reason is to get access to the pre-existing dialects and infrastructure of MLIR for future development and research into RVSDG.

The contributions of this thesis are as follows:

- A conceptual mapping from RVSDG to MLIR is presented
- An MLIR dialect that captures said mapping is created and presented

CHAPTER 2

Theory

This chapter will cover some concepts related to LLVM, MLIR, and RVSDG in greater detail. This chapter draws heavily from the "Background" chapter in the pre-study report [7].

2.1 LLVM

LLVM started as a modular compiler framework presented by Chris Lattner in 2002 [8]. Today LLVM is an umbrella project that encompasses several subprojects [9] including LLVM Core (the original framework), Clang (C/C++ compiler), LLDB (Debugger), MLIR, which will be discussed in section 2.2, and several other projects. These are used in all manner of compiler and code analysis projects, both open source, academically, and in industry [10]. LLVM is built around its own intermediate representation called LLVM IR, including target- and source- independent optimizations, and code-generation for a number of different CPUs [11]. In theory, this structure should simplify the process of building compilers for new languages significantly; All that is needed for getting it running is to create a compiler frontend that consumes source code and generates LLVM IR, and the LLVM framework handles the rest.

The LLVM IR is a low-level, Single Static Assignment (SSA) IR [12, 8]. In terms of available types and operations, it is not too different from a RISC-style instruction set, but it also retains some high-level information to enable more advanced optimization and analysis. LLVM IR is however not suited, or intended [8], to be a generic compiler IR. Due to its inherent low-level nature, it is unable to directly capture high-level, language specific concepts which are needed to perform language specific analysis and optimization. This has caused several compiler projects to create their own domain-specific IRs that lie between the high-level source language and LLVM IR [3]. While this approach has worked reasonably well, its necessity highlights a flaw in the LLVM framework; Instead of simply creating a frontend that "naively" converts source code to LLVM IR, compiler developers will also have to create their own IR with associated infrastructure to be able to perform high-level optimizations.

2.2 MLIR

MLIR (Multi Level Intermediate Representation) is a subproject under the LLVM umbrella [9] that seeks to address the shortcomings of LLVM IR by providing a framework for implementing domain-specific IRs and transformations [3]. It does this by providing an extensible ecosystem of IR dialects that model different levels of abstraction, a declarative interface for defining new levels of abstraction, and common infrastructure such as pass management, IR verification, documentation building, and printing and parsing logic [3]. MLIR also standardizes the use of SSA-values.

2.2.1 TableGen

TableGen is a tool under the LLVM project whose goal is to help humans maintain and develop records of domain specific information [13]. It reads a tableGen source file, parses it, and passes the result to a domain specific backend which converts the data into a format more suitable for direct usage. In MLIR, tableGen is used as a declarative interface for defining Ops [14, 15], types [16], attributes [16], interfaces [17], certain graph rewrites [18], and a few other things. Under the hood, most of these things are specializations of C++ classes. These could be written by hand, but using tableGen to automatically define them instead saves dialect authors from writing a lot of boilerplate code, at the cost of having to learn tableGen. MLIR also makes quite heavy use of strings to identify various structures and concepts at run time [14]. TableGen helps developers manage these strings and ensures changes are properly propagated throughout the project.

A tableGen source file consists of a list of records. Each record has a unique name, a list of values, and a list of super classes, where the list of values is the actual information being encoded. There are two main types of records: classes and definitions. Definitions are concrete records that will be instantiated by the backend, while classes are abstract records that make it possible for record creators to factor out common behavior or create additional levels of abstraction. TableGen does provide more complex features that makes it more of a general purpose programming language [19], although they will not be important for this thesis.

2.2.2 Dialects

Dialects are the main way extensibility is managed in MLIR. They are logical groupings of operations, types, and attributes [3]. Dialects generally model either a particular level of abstraction or a specific domain. Some examples of dialects that are a part of MLIR by default are the “arith”-dialect which holds basic mathematical operations [20], the “memref”-dialect which holds generic operations related to interacting with computer memory [21], and the “llvm”-dialect which models a subset of LLVM IR [22]. Different dialects can also coexist on all levels of the IR [3]. This enables compiler developers to pick and choose components from different dialects when designing their IRs. Developers are also able to create their own dialects to fit their own use-cases. When doing this, developers may choose to incorporate ops, types, or attributes from other dialects, which can save a decent amount of development and maintenance effort.

2.2.3 Ops

Operations, or *Ops* as they are commonly called in MLIR, are the main semantic units of MLIR [3]. They are used for modelling everything from modules and functions, to loops and conditional execution. Ops take zero or more *operands* and produce zero or more *results*, all of which are typed SSA-values [3]. They can also take several *attributes*, which are values that are known at compile time [3].

Ops can also contain a number of *regions* which are used to model nested program structure. These regions contain a list of *blocks*, which in turn contain a list of Ops. The blocks within any given region form a control flow graph, where terminator Ops in each block can pass control to another block. Blocks also have a list of typed *arguments*, which are can be used as SSA-values within the block. When control is transferred to a block, the previous Op or block that held control decides the values of the next block's arguments. Ops do this implicitly, while other blocks pass the new argument values as operands to their terminator Ops. Passing of block arguments is used to replace LLVMs phi-function [3, 23], which is used to maintain SSA-form under conditional execution.

As of writing, there exists two main types of regions in MLIR: Single Static Assignment Control Flow Graph (SSACFG)-regions, and graph-regions [24]. SSACFG regions can have multiple blocks which form a CFG as mentioned above and the Ops within each block are, somewhat informally, expected to run sequentially. A consequence of this is that Ops in SSACFG-regions can only access SSA-values that were produced before them in the IR. Graph-regions on the other hand only have one block and model Ops that happen in an arbitrary order. Their primary use is for modelling data-flow graphs, and as such they allow cyclic dependencies between Ops.

While the standard MLIR dialects offer a decent range of different Ops, these are not the only ones available. The registry of Ops is fully extensible, and dialect authors can add as many Ops as they need for their dialects [3]. The semantics of operands, results, attributes, and regions are also entirely dependent on the Op they are attached to [3]. This gives dialect authors a lot of freedom in regard to modelling different domains, while still retaining a common interface for interacting with Ops from different dialects.

MLIR Ops have a set of constraints that define what makes a valid Op instance. The most common of these constraints come in the form of type and count constraints on inputs, outputs, and regions. Most of these can be defined declaratively through TableGen [14]. If these constraints prove insufficient for properly verifying an Op, it is possible for dialect authors to create their own verifiers using C++.

2.2.4 Types

All SSA-values in MLIR have a data-type [3]. This can be anything from simple integer types to complex composite types like structs, vectors, or tensors. These types encode information about the values that are known at compile time, and are specified by the Op that created the value or the block which defines it as an argument [3]. The standard MLIR dialects provide several common types such as integers, floats, memory references, compact vectors, and tensors. In addition to

```
// builtin.module is an Op with a single region with a single block
builtin.module {
    // arith.constant takes a literal integer attribute and produces
    // one result of type i32.
    %val1 = arith.constant 1: i32
    %val2 = arith.constant 2: i32

    // arith.addi and arith.muli has two SSA operands and produces
    // one result.
    %sum = arith.addi %val1, %val2: i32
    %product = arith.muli %val1, %val2: i32
}
```

Figure 2.1: MLIR assembly showing some different standard Ops

these, the type system is fully extensible, which allows dialect authors to define their own types to better model their domain of interest.

2.2.5 Attributes

Attributes are typed values which contain information about Ops that is known at compile time. Unlike SSA-values they are not typed using the MLIR type system, but are instead typed using C++ types. The semantics of a given attribute is specified entirely from the Op it's attached to.

Figure 2.2 gives an overview of how the different MLIR concept link to each other. Lines should be read from the end with no feet. The cardinality of the different relationships is shown through the line "feet": Double dashed lines indicate a relation to exactly one entity, while the crows foot with a circle indicates a relation to zero or more entities. E.g Ops have regions which contain blocks which in turn contain Ops. Block arguments are essentially SSA-values which all have exactly one type. In addition the purple dialect box shows how a dialect is a grouping of types, ops, and attributes.

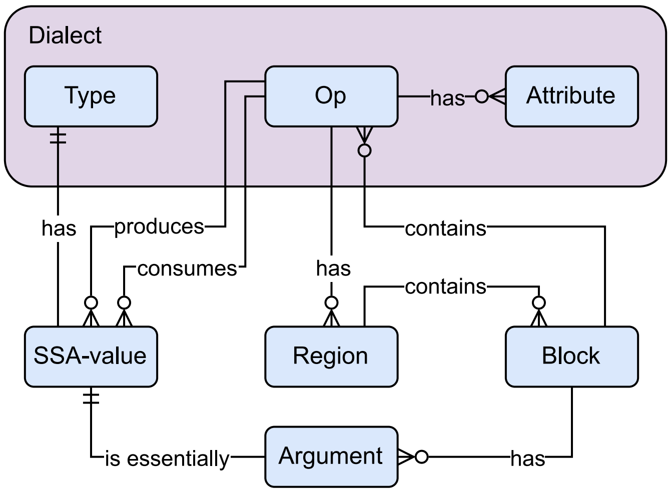


Figure 2.2: Graph showing relation between MLIR elements

2.2.6 Passes

A pass is a mechanism for traversing, and potentially rewriting, an IR graph. Different passes can perform different tasks, such as analysis, transformation, or optimization. MLIR provides a set of common passes, as well as frameworks for specifying pass pipelines and implementing new passes [3, 25]. All passes are by default Op agnostic, meaning they can run on any Op, but it is also possible to create passes that only run on specific Ops. MLIR also provides a framework for defining different analyses. These are conceptually quite similar to passes, except that they lazily compute information about a specific Op instead of performing graph transformations [25]. After an analysis has been performed, it is cached until a pass invalidates it. Passes can query analyses for the Op being processed to determine how it should be processed.

2.2.7 Traits and interfaces

An MLIR graph can contain Ops and types that are defined in different dialects and even exist on completely different layers of abstraction. Since passes can traverse any Op anywhere in the graph, they have to be able to query potentially very different Ops or types in a unified way. To combat this problem, MLIR introduces traits and interfaces [3, 26, 17]. *Traits* are used to model simple properties that an Op will express unconditionally. Properties such as "has no side effects", and "is terminator" are typically modelled as traits. More nuanced behavior can be modelled generically using *interfaces*, which provide a set of interface methods that must be implemented by the Op or type. These are then made available to passes that interested querying Ops for specific information. Some examples of interfaces provided by MLIR out of the box are callable/call Op interfaces [17] and side effect interfaces [27].

2.2.8 Progressive lowering

One of the core concept in MLIR is progressive lowering [3]. This means that high level IRs shouldn't be lowered directly to machine code, but that it should instead be lowered step by step through the different levels of abstraction until a set of dialects suitable for target code generation is reached. Lowering in this manner helps support a wide array of programming models and target architectures [3], and enables the creation of pass pipelines that perform optimizations at multiple levels of abstraction [3]. It also encourages the creation of passes that can work across different levels of abstraction, which in turn makes reusing passes easier.

To exit MLIR, a subset of LLVM IR is provided in the form of the "llvm"-dialect [22]. Once MLIR IR has been completely lowered to this dialect, usually through several other dialects, the IR can be exported to actual LLVM IR and compiled to runnable code.

2.2.9 IR representations

MLIR IR is designed to exist in three different forms: an in-memory form suitable for programmatic creation and manipulation, a compact serialized form used for storage and transport, and a human-readable textual form (MLIR assembly) which allows for easier inspection and debugging by developers [24]. These forms all represent the same information, and each of them can be converted into any of the others. All three forms must also be capable of dealing with the extensible nature of MLIR. The in-memory and serialized forms do this by directly representing Ops, regions, types, SSA values etc. without taking the semantics of the different constructs into account [28]. By default, the textual format also uses a similar structure, but since it needs to be human-readable to be useful, MLIR also offers mechanisms for creating custom representations of Ops, types, and attributes. Formats can either be specified declaratively using a format string [15], or by writing custom printers and parsers in C++. The format strings can contain string literals which are printed and parsed directly, and field names which print-/parse the value of some field of the entity being printed. The format strings also accept directives, which are functions that can optionally take some parameters and handle printing and parsing of special cases [15]. Dialect authors may create their own directives as well if the provided mechanisms prove insufficient for their applications, and they don't feel like creating fully custom printers/parsers. An example of some built-in Ops in both the pretty and generic format can be seen in Figure 2.3.

```
module {
    %val1 = arith.constant 5 : i32
    %val2 = arith.constant 4 : i32
    %sum = arith.addi %val1, %val2 : i32
}
```

(a) Pretty format

```
"builtin.module"() ({
  %0 = "arith.constant"() {value = 5 : i32} : () -> i32
  %1 = "arith.constant"() {value = 4 : i32} : () -> i32
  %2 = "arith.addi"(%0, %1) : (i32, i32) -> i32
}) : () -> ()
```

(b) Generic format

Figure 2.3: MLIR "pretty" format vs. generic format

2.2.10 Software engineering

MLIR is fundamentally designed around the concepts of user extensibility and compatibility between layers of abstraction [3], but how does it accomplish this from a software engineering point of view?

Extending MLIR primarily involves specializing C++ base classes. This holds for most concepts in MLIR, including dialects, Ops, types, attributes, traits, interfaces, and passes. The base classes provide the class members and hooks for interacting with the different MLIR structures generically. They are extended polymorphically, and generally wish to use implementations from the derived classes. This could be accomplished by using run-time polymorphism through C++ virtual methods, but MLIR has in general opted for static polymorphism through the "Curiously Recurring Template Pattern" (CRTP) [17, 29]. This pattern involves creating a base class that takes a class derived from itself as a template parameter. The derived classes then inherit from the base class and pass themselves as the template parameter. This gives the base class access to the implementations in the derived class. A simplified example of how this pattern can be used to call specialized Op verifiers can be seen in Figure 2.4. Using this pattern allows nearly all type resolution to be performed at compile time, avoiding the run-time cost of virtual method calls. Dialect classes are an exception to this rule, as they use virtual calls instead.

```
template<class ConcreteType>
class OperationBase {
    ...
    static bool verify(Operation *op){
        return llvm::cast<ConcreteType>(op).verify();
    }
    ...
}

class MyOperation: public OperationBase<MyOperation> {
public:
    ...
    bool verify() {
        printf("MyOperation verified!");
        return true;
    }
    ...
}
```

Figure 2.4: Example of the Curiously Recurring Template Pattern

Specializing an MLIR Op does not add any class members that are not already provided by the base class. In fact, all Ops are identical with regard to storage. Specialized Op classes are actually smart pointer-like objects that reference the storage object for the specific Op and provide a semantically meaningful interface on top of it. This also holds for types and attributes, but creating storage for these construct can be a little more challenging. For non-parametric, or singleton, types and attributes, no extra work is needed. Parametric types and attributes, however, need a little more care since they also need to store their parameters. This is done by creating a specialized storage class that performs all necessary allocations and memory handling that is required to store an instance of the type or attribute. When using tableGen to define types or attributes, these storage classes are automatically generated for types and attributes that only use simple parameters. For more complex parameters, MLIR provides dialect authors with the capability of letting the parameters themselves tell the storage class how they should be allocated and stored. This makes reusing complex parameters a straight forward task once it has been properly implemented once.

Since parametrized types and attributes often appear with the same parameters multiple times in the same IR graph, their storage objects are made unique. The creation and management of immortal unique storage objects is the responsibility of an MLIRContext object. This object also holds a registry of registered dialects and Ops, as well as a thread pool used for various multithreaded operations performed by MLIR.

Passes are, like most things in MLIR, implemented by specializing a CRTP base class. Each pass is anchored to exactly one Op, and is only allowed to inspect and modify that particular Op and Ops contained in its regions. Passes are in addition allowed to inspect, but not modify, the state of direct ancestors of the anchoring Op. A consequence of this is that any pass that requires information about sibling Ops must be anchored to the parent of the Op whose siblings must be queried. Additionally, passes are not allowed to maintain global mutable state or preserve mutable state between runs of the pass, and must be copy-constructable [25]. These restrictions exist to make it possible for the MLIR pass infrastructure to schedule passes to run in parallel, which can increase compiler performance on modern multicore systems.

Configuring and scheduling pass pipelines is handled through the use of pass managers. Pass managers schedule passes at a particular level of nesting. Every pass pipeline has a top level pass manager which acts as an entry-point to the pipeline [25]. This top level pass manager keeps track of the passes that should be run on the highest level of nesting, but it can also contain nested pass managers which keep track of passes that should be run on the next level of nesting. These nested pass managers can again contain their own nested pass managers, enabling the creation of pass pipelines that operate on any specific level of nesting. Occasionally it may be useful for a pass pipeline to be scheduled as part of another pass, or to create passes that need to operate at a level of nesting that can not be statically known. MLIR supports the creation and scheduling of such dynamic pass pipelines by allowing passes to run arbitrary pass pipelines on the Op being operated on or any of its children [25].

Passes will often want to analyze the Ops they are working on to guide their transformations. These analyses can be needed by several passes, and could therefore be useful to cache. MLIR handles this through the use of analysis classes and analysis managers. Analysis classes are, unlike passes, not based defined by deriving from a CRTP base class. They are instead nearly entirely defined by the analysis author, with the only restrictions being that they need a particular constructor signature, that they are not allowed to modify the IR graph, and that they are only allowed to inspect the Op the analysis is run on, its direct descendants, and its direct ancestors [25]. Analyses can be queried through an analysis manager and will, once queried, remain cached until invalidated. An analysis will generally be invalidated by any pass being scheduled on the Op the analysis was run on, although passes may declare analyses that are known to be preserved.

2.3 RVSDG

The Regionalized Value State Dependence Graph (RVSDG) is a class of compiler Intermediate Representations (IR) which aims to model both inter- and intraprocedural computation [4, 5]. It is a data-flow centric IR and takes the form of a directed, acyclic, hierarchical multigraph, which makes it well suited for performing data-flow based analyses and optimizations when compared to more traditional Control Flow Graphs. In this graph, computation is modelled as nodes and data-dependencies as edges. Nodes have a tuple of inputs and a tuple of outputs, which model the dependencies and results of the computation, respectively. Each input is connected to exactly one output, but each output can be connected to an arbitrary number of inputs. Ordering of side-effecting operations is modelled using state-edges. RVSDG has two classes of nodes: simple nodes which only have inputs and outputs, and structural nodes which additionally have at least one region. Regions consist of a tuple of region arguments, a tuple of region results, and a sub-graph built up of nodes and edges. Structural nodes are used to model hierarchical programming structures such as loops, conditional execution, and functions.

RVSDG has several properties that makes it an attractive IR for optimizing compilers. Since the graph-structure is inherently in SSA-form, passes that reestablish SSA-form are not necessary [5]. Rediscovering and recanonicalizing control flow structures isn't necessary either, since these are explicitly encoded in the graph [5]. RVSDG is also highly flexible with regard to which level of abstraction it operates on, since computation is primarily performed by simple nodes which are free to model high- or low-level operations [4]. RVSDG can also be used to model independent computations, including independent side-effecting operations, and is also capable of representing all levels of a program as a single, unified data structure [5].

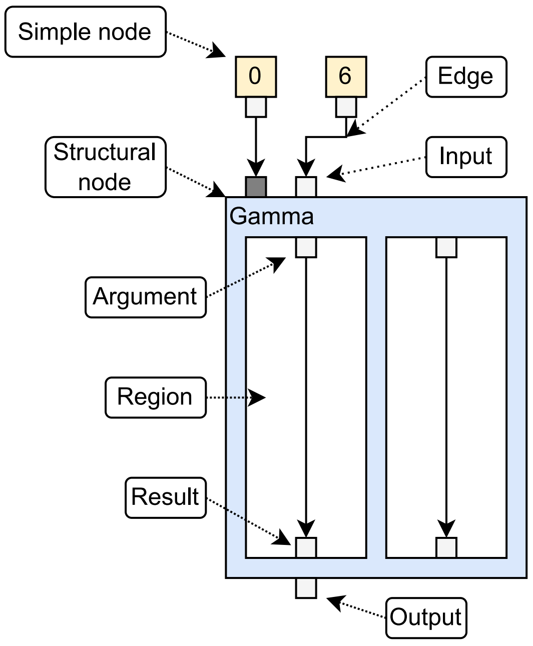


Figure 2.5: RVSDG terminology visualized

2.3.1 Node types

Simple nodes model primitive operations [4]. They have a tuple of inputs and a tuple of outputs, which map to the arguments and results of the modelled operation. These nodes do not contain any regions. Examples of operations that could be modelled by these nodes include arithmetic operations such as integer additions, subtractions, or multiplications, memory loads and stores, bitwise operations, and comparisons.

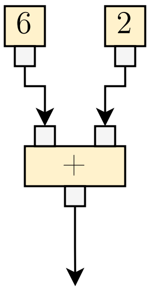
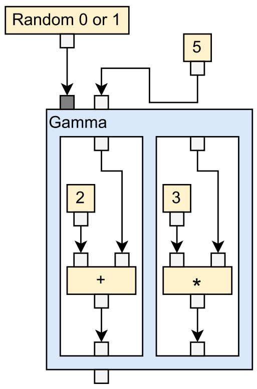


Figure 2.6: Simple nodes modelling the expression 6+2

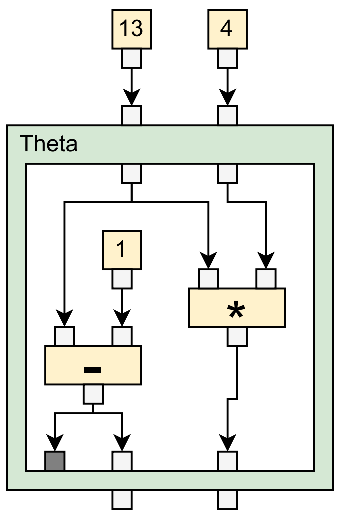
Gamma(γ)-nodes are structural nodes which model conditional execution, such as if-else or switch statements [4]. They have two or more regions which represent the different paths the execution may take. The first input to the gamma-node is a predicate that decides which region should be executed. The other inputs are context values that are needed by the nodes within the regions and map directly to the region arguments. Region results are mapped directly to the outputs of the gamma-node.



```
int i = random 0 or 1;
int x = 5;
if (i == 0) {
    x = x+2;
} else if (i == 1) {
    x = x*3;
}
```

Figure 2.7: Gamma node modelling an if-else-statement

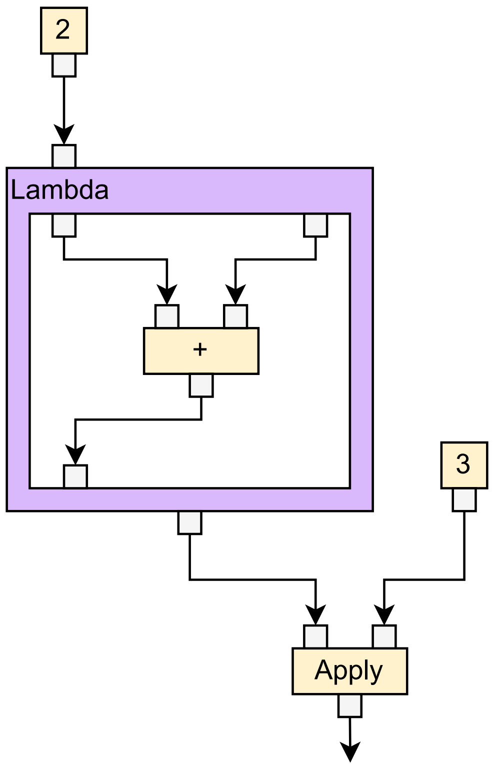
Theta(θ)-nodes are used to model tail-controlled loops [4]. They contain exactly one region which models the loop body. The inputs to the node model context values needed by the nodes in the loop body. They map directly to the region arguments of the first iteration of the loop. For iterations after the first, the arguments take the values passed to the region results in the previous iteration. The theta body has an additional region result, which is a predicate that decides whether the loop performs another iteration. After the last iteration, the values passed to the region results are mapped onto the outputs of the theta-node. It is possible to model head-controlled loops by combining gamma- and theta-nodes.



```
int n = 13;
int a = 4
do {
 a = n * a;
 n = n-1;
} while (n > 0)
```

Figure 2.8: Theta node modelling a do-while-loop

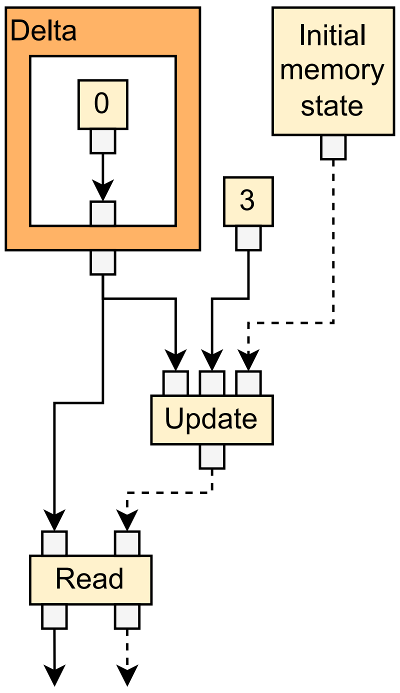
Lambda(λ)-nodes model function definitions [5]. The inputs to a lambda node model context-values that are needed by the function, while the output of the node is a reference to the lambda-node itself. This reference can be passed to the special apply-node together with any function parameters to call the function, with the results of the call being mapped to the outputs of the apply-node. Lambda-nodes have a single region which models the function body. The region arguments represent both the required context values and function arguments. Values passed to the region results are the return values of the function.



```
int inc_value = 2;
int f(int value) {
    return value + inc_value;
}
f(3);
```

Figure 2.9: Lambda node modelling an "add N" function

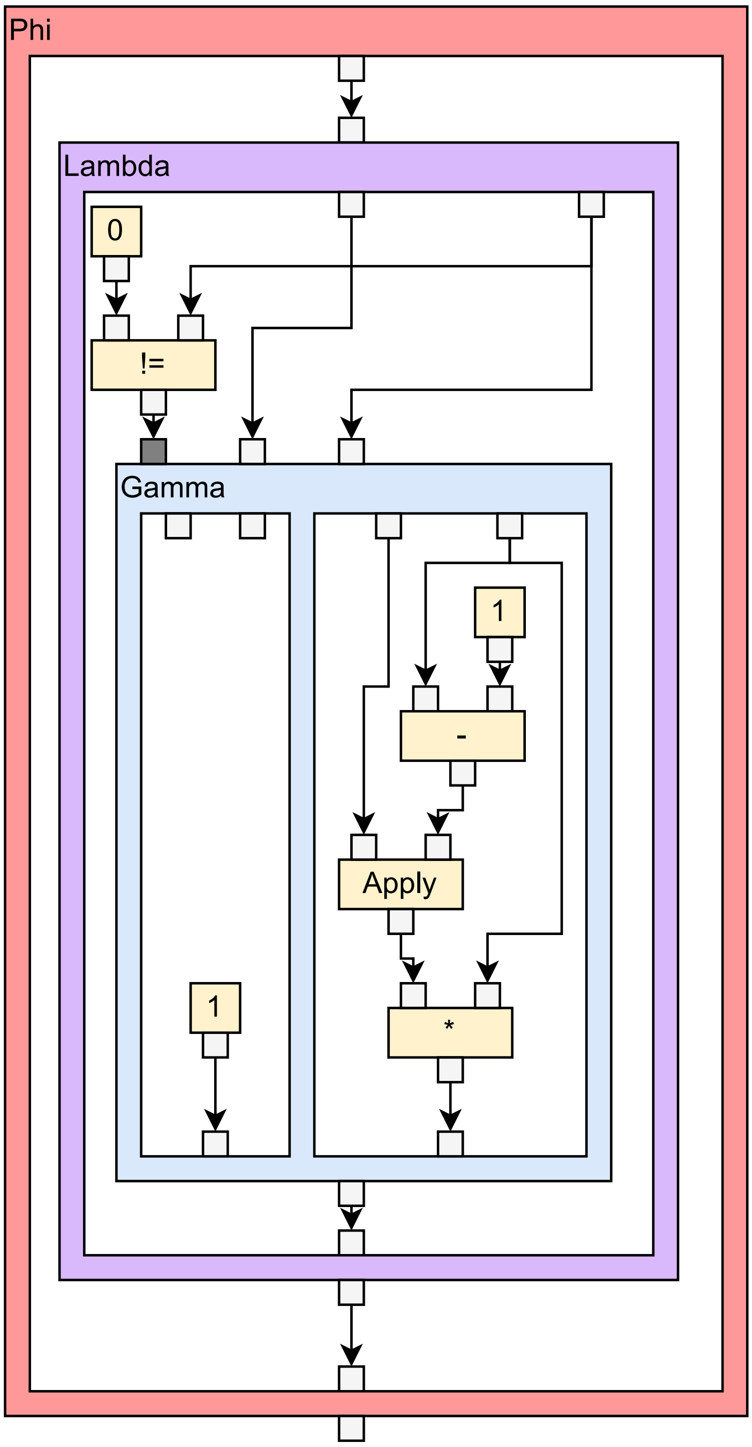
Delta(δ)-nodes model global variables [5]. The inputs of the delta-node model the external dependencies of the variable, and their output is a reference to the delta-node. The delta node has a single region which models the expression used to give the variable its value.



```
int delta = 0;
delta = 3;
delta;
```

Figure 2.10: Delta node modelling code. Dashed edges are state edges

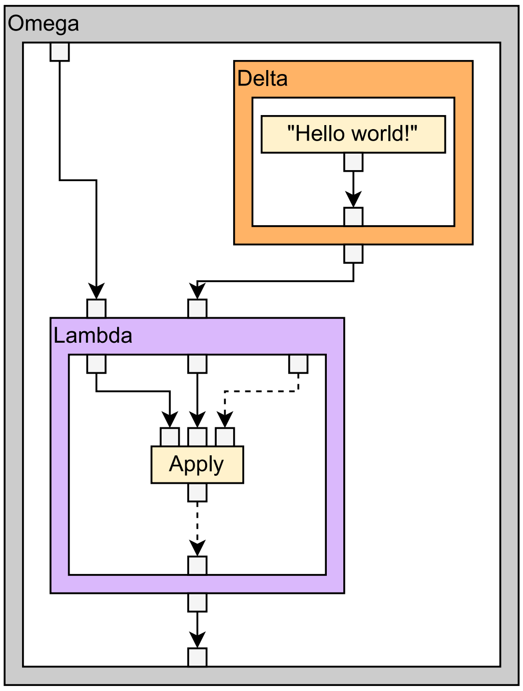
Phi(ϕ)-nodes are used to express environments with mutual recursion [5]. Since the RVSDG is directed and acyclic, it is not possible for two nodes to take each other's outputs as inputs, or for any node to take its own output as an input. This poses a problem when attempting to model recursive functions, as they would require a reference to themselves to be passed as a context value, which is impossible in a pure DAG. The phi-node is used to solve this problem. It has a single region which models the mutually recursive environment. The results of this region map both to the outputs of the phi-node and back to the region arguments. This mapping of region results to region arguments is the mechanism that enables the modelling of recursive program structures, while mapping them to the outputs allows external nodes to access them. The inputs of the phi-node model context values required by the enclosed nodes, and map directly to the non-recursive region arguments.



```
int f(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * f(n-1);
    }
}
```

Figure 2.11: Phi node modelling a recursive factorial function

Omega(ω)-nodes are the unit of compilation in RVSDG [5] and model a module, including external imports and module exports. Omega-nodes are always the top-level nodes in RVSDG, and as such they have no inputs or outputs. They contain a single region which represents the code in the compilation unit. The arguments for this region model external imports, while values passed to the region results are exported from the module.



```
include print;
char message[] = "Hello world!";

void greeter(){
    print(message);
}

export greeter;
```

Figure 2.12: Omega node modelling a program module. The imported value is a reference to some printing method. The defined lambda is also made available to other modules. Dashed lines are state edges

Term	**Description**
MLIR	The MLIR project with associated tools and libraries
MLIR IR	MLIR based IR in any of its three representations
MLIR assembly	Textual representation of MLIR IR
RVSDG	The Regionalized Value State Dependence Graph IR as a theoretical construct.
RVSDG dialect	The MLIR dialect implemented in this thesis that models RVSDG.
Node ports	Collective term for RVSDG node inputs, outputs, region arguments, and region results.

Table 2.1: Terms with descriptions

CHAPTER 3

Methods

This chapter describes how the library was developed, and details how the lessons learned in the pre-study [7] were applied. How RVSDG was initially planned to be represented in MLIR will also be explained here.

3.1 Development environment

All software has been developed and tested on a university-provided virtual machine running the following software:

- Operating system: Ubuntu 22.04
- LLVM version: 16.0.0
- MLIR version: 16.0.0
- Clang 16.0.0
- CMake 3.25.2
- Ninja 1.10.1

A dockerfile for setting up a development container can be found in the main code repository. Links to the code repositories can be found in Appendix A, and more concrete instructions for building, running, and developing the code can be found in Appendix B.

3.2 Project structure

The project was structured as an out-of-tree MLIR dialect library, as opposed to the in-tree structure that was used in the pre-study [7]. This was done for two main reasons. The first was to reduce development friction by having related files exist close to each other in the source tree, and to not have to rebuild MLIR on every update. During the pre-study, a lot of time was wasted navigating the file system or waiting for the compiler to re-link MLIR [7], time that could have been better spent on development. The second reason was to improve the usability of the library. By having the dialect as a standalone library, it is possible to use it with pre-built LLVM and MLIR libraries. Since building these libraries requires quite powerful hardware [7] this lowers the barrier of entry for working on the library by quite a bit. A more in-depth analysis of the pros and cons of in-tree and out-of-tree can be found in the pre-study report [7].

In addition to the RVSDG dialect, a separate but dependent JLM-compatibility dialect was also created. This was done to model certain operations and types that were not available in an RVSDG-compatible form through the default MLIR dialects. These Ops and types could in theory have been a part of the RVSDG dialect, but since they were not a part of pure RVSDG, it was decided to move them to their own dialect. This subject is disussed further in section 5.3.

3.3 Representation of RVSDG in MLIR

During the pre-study, a seemingly workable way to represent RVSDG in MLIR was discovered but not fully implemented [7]. This mapping formed the basis for the development of the dialect library.

RVSDG nodes are modelled using MLIR Ops. This mapping was chosen since MLIR Ops are conceptually quite similar to RVSDG nodes. They are both used to model computation or structure, they both take SSA-values as inputs, and produce SSA-values as outputs, and both can contain regions with subgraphs. Using this mapping also enables the use of pre-existing side-effect-free Ops as simple nodes, which drastically reduces the amount of work needed to model useful programs.

Structural node regions in RVSDG are modelled as single-block SSACFG regions in MLIR, with the RVSDG region arguments being represented by the arguments of the MLIR block, and the RVSDG region results being represented by the operands to the block terminator Op. While RVSDG is a data-flow graph, and MLIR graph-regions are created to model data-flow graphs, it makes more sense to use SSACFG-regions when modelling RVSDG due to RVSDG being acyclic. If graph-regions had been used, an extra check would be required to ensure that the regions don't contain cycles. MLIR graph-regions also don't allow block arguments, which would prevent their usage for representing RVSDG region arguments.

Mappings between structural node inputs, node outputs, region arguments, and region results are encoded by how they are ordered. If a structural node has inputs that map to its region arguments, the first input will map to the first region argument, the second input maps to the second argument etc.

RVSDG edges are modelled as passing MLIR SSA-values between Ops. This mapping more or less follows from the mapping of nodes to Ops, as edges in RVSDG already represent the passing of SSA-values. Modelling state edges uses the same mechanism, but a custom MLIR type has to be created to know that the SSA-value represents a state edge. Since Ops modelling simple side-effecting RVSDG nodes also need to consume and produce these state-types, it is not possible to re-use Ops from other dialects directly. While the pre-study report mentions that creating a generic "shim-Op" that wraps side-effecting Ops and add state edges would be the best way forward [7], it was instead decided to implement state-edge compatible versions of any side-effecting Ops that may be needed.

The outputs of lambda- and delta-nodes also requires some special consideration, since they are references to the nodes themselves. There were primarily two ways to represent this. The first option was to use MLIR symbol tables. This is a mechanism for creating symbols which can be referenced by name in a non-SSA manner [30] and is the way the default "func" dialect handles function references [31]. The second option is to create MLIR types for lambda- and delta-references. These types would need to carry some additional information. The delta-reference type would need to be aware of the datatype of the variable modelled by the delta-node. The lambda-reference type would in a similar vein be required to know the function signature of the lambda-node. For this project, the second option was chosen since it kept the mapping from edges to passing of SSA-values consistent.

The predicates for gamma- and theta-nodes were decided to be modelled as a custom control type. This mirrors the implementation used in JLM. The control type is instantiated to be able to represent N different options, and an SSA-value of the control type represents exactly one of these N options. A mapping operation was introduced to convert arbitrary integer types into these control types. This choice was primarily made to increase compatibility with JLM, but also makes it possible to ensure that no out-of-bounds predicates can make it past the Op verifier.

RVSDG	**MLIR**
Simple node	Op with no regions
Structural node	Op with regions
Node input	Op operand
Node output	Op results
Node region	Single-block SSACFG region
Region argument	Block argument
Region result	Operand of block terminator Op
Edge	Passing of SSA-values
State edge	SSA-value with custom state type

Table 3.1: Conceptual mapping from RVSDG to MLIR

3.4 Testing and validation

Testing of the dialect library was primarily done by manually writing MLIR assembly using the implemented Ops, types, and attributes and check if they were parsed and verified properly by MLIR. Some more complex examples were also created by creating a tool in JLM that converts JLM RVSDG to MLIR assembly. To aid in verification, RVSDG dialect aware versions of the mlir-opt tool and MLIR language server [32] were created. Mlir-opt is a woefully undocumented tool, but it serves a similar purpose to llvm-opt [33]. It reads in an MLIR assembly file, applies the specified passes to it, and outputs the resulting MLIR assembly. Roundtripping MLIR assembly through this tool is a good way to ensure that MLIR assembly printing and parsing expect the same format and that different optimizations work as intended. The MLIR language server provides IDE support for MLIR assembly, which includes syntax checking, type checking, Op verification, and the ability to view the generic MLIR assembly of Ops with a custom format. This tool provides dialect authors with a quick sanity check while creating verifiers, type constraints, and assembly formats by providing instant feedback on handwritten MLIR assembly.

To improve the readability and writability of MLIR assembly using the RVSDG dialect, it was decided to create a custom assembly format for the various structural nodes. This was mainly done using the declarative assembly format provided by MLIR [14]. Two custom assembly directives were also created to improve the format. The first of these was made to print the operand and argument lists so that the SSA-values and their respective types are written next to each other instead of being split into separate lists. This was done to improve the ergonomics of reading and writing the MLIR assembly by decreasing the required amount of eye and cursor navigation. The second custom directive was used to standardize printing of the regions of structural nodes. This directive was created to make it clear that structural node regions should only contain a single block by creating a region-centric syntax instead of a block-centric syntax. An example of the custom format compared to the generic format can be seen in Figure 3.1.

For more complete verification of the library, the plan was to roundtrip JLM IR through the RVSDG dialect. This would show that the RVSDG dialect was capable of modelling the same structures as JLM. To do this, it would be necessary to link the RVSDG dialect library in to the JLM executable, which would in turn mean that JLM and the RVSDG dialect library would need to use the same version of the different LLVM libraries. Unfortunately, at the time of coding, JLM used LLVM 14 while the RVSDG dialect library used LLVM 16. Some attempts were made, but performing either an upgrade of JLM or a downgrade of the dialect library was not deemed worth it, as going through MLIR assembly instead would still provide some validation of the dialect without requiring either project to change their LLVM version.

```
rvsdg.omegaNode (): {
    %ctx0 = arith.constant 4: i32
    %ctx1 = arith.constant 5.0: f32
    %l = rvsdg.lambdaNode <(i32)->(f32)> (%ctx0:i32, %ctx1:f32):
        (%arg: i32, %ctx0: i32, %ctx1:f32): {
        %0 = arith.muli %ctx0, %arg: i32
        %1 = arith.sitofp %0: i32 to f32
        %2 = arith.addf %1, %ctx1: f32
        rvsdg.lambdaResult(%2:f32)
    }
    %param = arith.constant 100: i32
    %res= rvsdg.applyNode %l:<(i32)->(f32)>(%param:i32) -> f32
    rvsdg.omegaResult()
}
```

(a) Pretty format

```
"rvsdg.omegaNode"() ({
  %0 = "arith.constant"() {value = 4 : i32} : () -> i32
  %1 = "arith.constant"() {value = 5.000000e+00 : f32} : () -> f32
  %2 = "rvsdg.lambdaNode"(%0, %1) ({
  ^bb0(%arg0: i32, %arg1: i32, %arg2: f32):
    %5 = "arith.muli"(%arg1, %arg0) : (i32, i32) -> i32
    %6 = "arith.sitofp"(%5) : (i32) -> f32
    %7 = "arith.addf"(%6, %arg2) {
            fastmath = #arith.fastmath<none>
        }:(f32, f32) -> f32
    "rvsdg.lambdaResult"(%7) : (f32) -> ()
  }) : (i32, f32) -> !rvsdg.lambdaRef<(i32) -> (f32)>
  %3 = "arith.constant"() {value = 100 : i32} : () -> i32
  %4 = "rvsdg.applyNode"(%2, %3) :
       (!rvsdg.lambdaRef<(i32) -> (f32)>, i32) -> f32
  "rvsdg.omegaResult"() : () -> ()
}) : () -> ()
```

(b) Generic format

Figure 3.1: Custom RVSDG dialect format vs generic MLIR assembly

3.5 Implementation

Implementing a dialect library starts with creating a tableGen record for the dialect itself. This record defines the name of the dialect, its C++ namespace, and other dialects the dialect depends on [34]. This record also controls the creation of certain methods or method signatures on the C++ class for the dialect. The dialect record is generally quite small and is mainly used as a common reference for other records. One particular thing to note is that method calls for registering Ops, types, and attributes need to be implemented in the same compilation units where the corresponding dialect members are implemented. The RVSDG dialect library uses separate source files for implementing custom methods on Ops, types, and attributes. These files are also responsible for including the C++ source files that are generated from the tableGen records and implementing the dialect methods for properly registering Ops, types, and attributes, respectively.

Once the dialect itself is created, the next step is to implement Ops and types. These were also primarily defined by creating appropriate tableGen records [14, 16]. While most of this implementation was "by the book", some aspects are worth drawing attention to.

All non-specialized RVSDG inputs, outputs, and region results of the structural nodes are defined as a variadic list of values of any type. This lets the structural nodes work with any types from any dialect out of the box. These constraints are, however, technically not correct, since the legal types for one node port often depends on the types of another. Port dependency constraints were not implemented in tableGen, and were instead enforced by creating custom C++ Op verifiers [14]. In general, these verifier functions check that the number of ports and the types they accept match for ports which map to each other. A custom verifier was also created for the apply node to ensure that the inputs and results of the apply node match the signature of the called lambda node. A different option and some associated tradeoffs are briefly discussed in section 5.2.

The control type used for values passed as predicates to gamma- and theta-nodes was originally implemented as a standard MLIR type parameterized by an unsigned integer that represents how many options a value of the type can select between. For the gamma-node, this worked fine, as the number of regions the predicate should select between can change from instance to instance. For the theta-node, however, the predicate should always choose between exactly two outcomes: run another iteration, or don't run another iteration. While this could have been handled in a C++ verifier, it was instead decided to create a more complex tableGen type constraint that verifies that a node port has a control type with a specific number of options. This type constraint was modelled using a tableGen class, which takes an integer representing the number of options of the control type as a class parameter. When the class is instantiated as part of an Op definition, the integer is passed and a suitable predicate can be generated by tableGen. To be used in this manner, tableGen also needs to know how to build a particular instance of the control type. This is done by declaring the existence of a simple builder on the generic control type which takes the number of options as a parameter, and by having the constrained control type class inherit from the BuildableType tableGen class. When inheriting from BuildableType, the exact code needed to get the type instance must be provided as a string of text. The number of options for the constrained type is injected into this code using tableGen's built-in paste operator [19].

One of the first and most time-consuming steps in the process of creating an MLIR dialect library from scratch is the creation of a functioning build system. MLIR provides several hooks and utilities for CMake-based projects, so CMake was chosen for the dialect library. Unfortunately, the utilities provided by MLIR are barely documented. The only place any mention of these utilities were within the CMake script files that defined them. MLIR does however provide a standalone dialect example which includes a basic CMake setup. Some light modification was required to make this work properly, including changing which MLIR sub-libraries were linked. Figuring out which libraries to link in was another challenge, since the MLIR documentation does not provide an explicit list of sub-libraries and their function. The only place either of these were found were within the build scripts for the MLIR project itself, and even then no information on scope or usage was anywhere to be seen. In short, the creation of the build system was heavily based on trial and error. Available MLIR libraries were discovered by digging through the MLIR source files to extract available CMake targets, and which libraries were relevant had to be inferred from their names and source code.

A similar problem occurred when attempting to make use of the sources and headers generated by tableGen. None of the automatically generated files include any external header files they depend on, a problem made worse by the at times highly opaque coding style and nested levels of abstraction employed by MLIR. This was eventually solved using the process of elimination by adding bulks of header files where needed, and then gradually removing files until the project would no longer build. Eventually, sets of header files which contained all definitions needed by the various auto-generated files were discovered. Additional header files, each including a single auto-generated header and its dependencies, were created to contain this problem and mitigate some pain for future dialect developers.

CHAPTER 4
Results

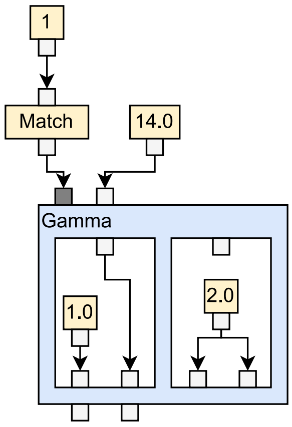
The main findings of this thesis are that RVSDG can be represented in MLIR, and that an MLIR dialect has been created to capture this representation. This dialect can interact with pre-existing non-side-effecting MLIR Ops from other dialects and seamlessly integrate them as simple RVSDG nodes. A secondary dialect for modelling JLM-specific operations and constructs was also created, but not completed. Some simple examples of the dialect being used to represent different RVSDG nodes can be seen in Figure 4.1, Figure 4.2, Figure 4.3, and Figure 4.4.

4.1 Constraints on node ports

To ensure that the order-based mappings between node inputs, node outputs, region arguments, and region results for structural nodes are valid the following constraints were placed on the structural nodes:

- Gamma nodes
  - Number of regions should be greater than or equal to 2
  - Number of options in predicate type should match number of regions
  - Number and types of region arguments should match node inputs
  - Number and types of region results should match node outputs
- Theta nodes
  - Number and types of node inputs, node outputs, region arguments, and region results (excluding the predicate) should be identical
- Lambda nodes
  - Given a function signature with n parameters of types T1, T2, . . . , Tn, and k node inputs of types I1, I2, . . . , Ik, the lambda node region should have n+k arguments of types T1, T2, . . . , Tn, I1, I2, . . . , Ik.
  - The number and types of the region results should match the return values of the function signature.

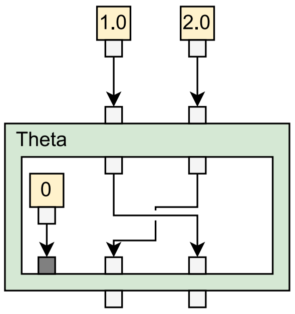
- Delta nodes
  - Number and types of node inputs should match the region arguments.
  - Node should have exactly one output and region result.
  - The type of result should be equal to the type the output references.
- Phi nodes
  - Given n inputs of type I1, I2, . . . , Tn, and k outputs of type O1, O2, . . . , Ok, the region should have k + n arguments of type O1, O2, . . . , Ok, I1, I2, . . . , Tn.
  - Number and types of results should match node outputs.



```
%pre_val = arith.constant 1: i32
%pred = rvsdg.match(%pre_val: i32) [
        #rvsdg.matchRule<0, 1-> 1>,
        #rvsdg.matchRule<default -> 0>
] -> !rvsdg.ctrl<2>

%var = arith.constant 14.0: f32
%res0, %res1 = rvsdg.gammaNode (%pred: <2>) (%var:f32):[
    (%var: f32):{
        %d = arith.constant 132.0: f32
        rvsdg.gammaResult (%d:f32, %var:f32)
    },
    (%var: f32):{
        %c = arith.constant 2.0: f32
        rvsdg.gammaResult (%c:f32, %c:f32)
    }
] -> f32, f32
```

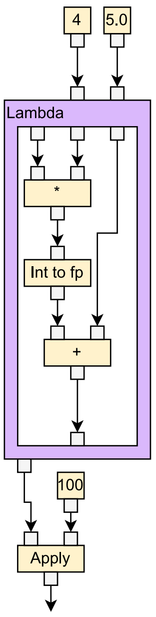
Figure 4.1: Gamma node represented in the RVSDG dialect



```
%test0 = arith.constant 1.0: f32
%test1 = arith.constant 2.0: f32

%res0:2 = rvsdg.thetaNode (%test0:f32, %test1:f32):
(%0: f32, %1: f32): {
    %predicate = rvsdg.constantCtrl 0: <2>
    rvsdg.thetaResult(%predicate) : (%1:f32, %0:f32)
} -> f32, f32
```

Figure 4.2: Theta node represented in the RVSDG dialect

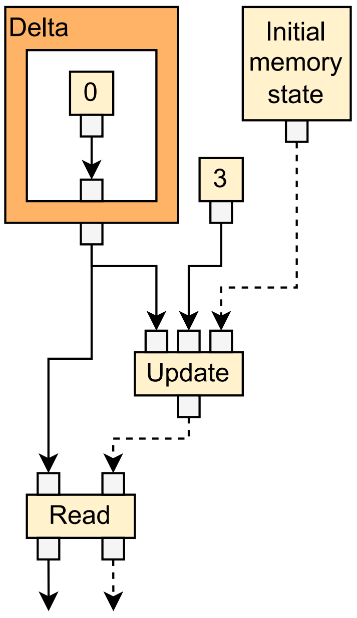


```
%ctx0 = arith.constant 4: i32
%ctx1 = arith.constant 5.0: f32

%l = rvsdg.lambdaNode <(i32)->(f32)> (%ctx0:i32, %ctx1:f32):
    (%arg: i32, %ctx0: i32, %ctx1:f32): {
    %0 = arith.muli %ctx0, %arg: i32
    %1 = arith.sitofp %0: i32 to f32
    %2 = arith.addf %1, %ctx1: f32
    rvsdg.lambdaResult(%2:f32)
}

%param = arith.constant 100: i32
%res= rvsdg.applyNode %l:<(i32)->(f32)>(%param:i32) -> f32
```

Figure 4.3: Lambda and apply node represented in the RVSDG dialect



```
%initState = --some previous memory operation--

%deltaRef = rvsdg.deltaNode():
    ():{
        %value = arith.constant 3: i32
        rvsdg.deltaResult(%value:i32)
} -> !llvm.ptr<i32>

%newVal = arith.constant 10: i32

%state = jlm.store (%deltaRef:!llvm.ptr<i32>, %newVal:i32) (%initState)
          -> !rvsdg.memState
%value, %nextState = jlm.load %deltaRef:!llvm.ptr<i32> (%state)
        -> i32, !rvsdg.memState
```

Figure 4.4: Delta node represented in the RVSDG dialect

4.2 Interoperability with pass infrastructure

The RVSDG dialect has also been shown to be able to interact with the existing pass infrastructure. By attaching the "Pure"-traits to the structural nodes, which marks them as having no side effects, the built-in Common Subexpression Elimination (CSE) pass was able to merge identical structural nodes. An example of this pass transforming MLIR assembly using the RVSDG dialect can be seen in Figure 4.5. How side effects should be marked on structural nodes will be discussed in more detail in section 5.2. A custom pass that traverses structural nodes and prints Op names in a nested fashion has also been created. This pass can be anchored to any Op marked as isolated from above, which includes all structural node Ops. It prints the name of the Op it is anchored to, followed by the names of all non-isolated from above Ops in its regions with proper indentation. If another node that is marked isolated from above is discovered in the regions, their name is not printed directly, but a new instance of the printing pass is instead scheduled on the node. This forms a dynamic pass pipeline as described in subsection 2.2.10. This printing pass was mainly created to serve as a baseline for future roundtripping with JLM.

4.3 Validation

Native roundtripping with JLM was not accomplished due to JLM and the RVSDG dialect library depending on different version of LLVM. A new tool called mlir-print was instead added to JLM which consumes LLVM IR, converts it to JLM IR, and outputs textual mlir-assembly which can then be read using the RVSDG dialect specific version of the mlir-opt tool. Examples of code represented using the dialect which were generated using mlir-print can be found in Appendix C.

Before CSE:

```
rvsdg.omegaNode (): {
    %2 = arith.constant 10: i32
    %3 = arith.constant 10: i32

    %theta_output, %theta_output2 = rvsdg.thetaNode (%2: i32, %3: i32):
        (%2: i32, %3: i32): {
            %predicate = rvsdg.constantCtrl 0:<2>
            rvsdg.thetaResult (%predicate): (%3:i32, %2:i32)
        }->i32, i32

    %theta_output3, %theta_output4 = rvsdg.thetaNode (%2: i32, %3: i32):
        (%2: i32, %3: i32): {
            %predicate = rvsdg.constantCtrl 0:<2>
            rvsdg.thetaResult (%predicate): (%3:i32, %2:i32)
        }->i32, i32

    %4 = arith.addi %theta_output, %theta_output2: i32
    %5 = arith.addi %theta_output3, %theta_output4: i32
    rvsdg.omegaResult (%4: i32, %5: i32)
}
```

After CSE:

```
rvsdg.omegaNode (): {
  %c10_i32 = arith.constant 10 : i32
  %0:2 = rvsdg.thetaNode (%c10_i32: i32, %c10_i32: i32) :
    (%arg0: i32, %arg1: i32): {
        %2 = rvsdg.constantCtrl 0 : <2>
        rvsdg.thetaResult(%2) : (%arg1: i32, %arg0: i32)
    } -> i32, i32
  %1 = arith.addi %0#0, %0#1 : i32
  rvsdg.omegaResult (%1: i32, %1: i32)
}
```

Figure 4.5: Builtin CSE-pass simplifying RVSDG-nodes

CHAPTER 5 Discussion

5.1 Validation of dialect

The main goal of this thesis was to see if RVSDG could be implemented in MLIR, and if so, if MLIR could be a good framework for future work with RVSDG. To this end, the RVSDG dialect was created. The dialect defines MLIR Ops, types, and attributes that can be used to represent RVSDG. Its ability to properly capture all of RVSDG has however not been rigorously tested. Testing with handwritten MLIR assembly and MLIR assembly generated by the mlir-print tool added to JLM has provided some encouraging results, but this does not show that the dialect actually captures the same information as JLM. The original plan was to create a fully-featured backend for JLM that directly produces in-memory MLIR IR instead of going through MLIR assembly, and then also create a tool that could create JLM IR from MLIR IR. Having both of these tools would make it possible to roundtrip IR between JLM and MLIR and prove that they are in fact both representing the same graph. This effort was unfortunately hindered by a mismatch in LLVM version between the RVSDG dialect library, which uses LLVM/MLIR 16, and JLM, which, at the time of writing, uses LLVM 15, but at the time of coding used LLVM 14. Downgrading the LLVM version for the dialect library to LLVM/MLIR 14 would prevent the use of the tableGen and MLIR assembly language servers, both of which were tools that helped a lot with development. Since the library is also meant to be used for new compiler projects, it also makes more sense to target a recent version of LLVM. The other option would have been to upgrade JLM. Doing this would require a good understanding of how JLM is built and how its different systems interact with each other. Due to time constraints on this project, and the near impenetrable nature of template and macro-heavy c++ code, it was decided that upgrading JLM would fall outside the scope of this thesis. Fortunately, there now seems to be a renewed development effort on JLM by the original authors, which has seen JLM upgraded to LLVM 15. Hopefully it will get to LLVM 16 in not too long, enabling the creation of tools for roundtripping IR between JLM and MLIR.

5.2 Evaluating representations

The pre-study for this thesis discussed different ways to represent RVSDG in MLIR. That discussion and the representations that were suggested are summarized in section 3.3 and formed the basis of development for the RVSDG dialect library. Several of these mappings were only theorized, not implemented, in the pre-study. Since they have now been implemented, it is time to look back and see if they work as well as previously envisioned.

Mapping RVSDG nodes to MLIR Ops still seems to be the way to go. MLIR Ops are meant to model any unit of semantics (functions, nested structures, computation etc.) and are therefore the obvious choice for modelling almost anything, but the suitability for RVSDG nodes goes a little further. Both RVSDG nodes and MLIR Ops take SSA values as inputs, and produce SSA values as outputs. They can both contain zero or more regions which have a set of arguments and a way to export values from the regions. Not every aspect of RVSDG nodes and MLIR Ops lines up perfectly, however. Custom C++ Op verifiers had to be written to ensure a correct relation between inputs, outputs, region arguments, and region results. It is likely possible to avoid this by creating more complex type constraints in tableGen [14], but it might make the tableGen definitions more difficult to read and modify, which would somewhat defeat the purpose of using TableGen in the first place. Overall, the representations of Ops, inputs, outputs, region arguments, and region results have worked quite well with only some smaller complications.

The mapping between RVSDG regions and MLIR regions has some incongruities. RVSDG regions directly have arguments, results, and a sub-graph, while MLIR has the added layer of "block" between region and Ops. Arguments and Ops can actually be directly accessed through the API on the region without needing to think about blocks, but results cannot, since terminator Ops are a block-specific concept. This is not a huge problem, and could probably be solved either by introducing a specialization of the region API or by creating an Op interface for all structural nodes which handles result Op retrieval.

While working on the phi-node, an inconsistency between JLM and the RVSDG dialect was noticed; JLM encodes the mapping between inputs, outputs, arguments, and results in a way that allows it to break the constraints mentioned in section 4.1, in particular the constraints of the phi-node outputs. JLM produced a phi-node that did not output all its recursion variables. In the RVSDG dialect, this would lead to ambiguity in the case where a phi-node has multiple recursion variables of the same type, since the link between node ports is defined by their position. When creating the mlir-print tool, this problem was solved by printing out additional outputs, but this may hint at a deeper incongruity that has not yet been discovered.

In its current iteration, the RVSDG dialect has three distinct types for representing state edges: IOState, MemState, and LoopState. These were somewhat naively copied from JLM, which uses the same three distinct types. All the state types implement a common state type interface. As of writing, the state type interface does not provide any fields or methods and only serves to enable recognition of an arbitrary state type. While this method works, there might be a more elegant way to integrate it with the side effect systems present in MLIR. MLIR queries Ops for side effects through the side effect interface [27]. This interface provides methods that allow Ops to analyze themselves and return a list of side effect objects to the querying entity. A state edge type parametrized by a list of side effect objects could be created, and the side effect interface implementation on RVSDG-compliant Ops could query incoming and outgoing state edges for the side effects of the Op. This would make the system more flexible in terms of more exotic side effects and could potentially enable more optimizations as it would be possible to track which specific resources are being modified and in what way by following the state edges. One consequence of this approach would be that structural nodes which contain side-effecting Ops in their regions would themselves be marked as having side effects, even if the state edge just passes through the node. Further testing is needed to determine whether this approach to modelling different types of state edges will be an improvement over the current implementation.

JLM is an experimental compiler that was made to show that RVSDG could be constructed, optimized, and destructed in a reasonable amount of time and with decently good results [5]. In this regard it performs quite well, but it carries with it a significant amount of baggage from LLVM IR. In particular, it has certain type and operation choices that may not be optimal for use in a higher level compiler. Lambda and delta references are for instance explicitly modelled using LLVM pointers. RVSDG itself does not enforce that these references should be low level pointers, and a higher level compiler might wish to use another abstraction for their references. The RVSDG dialect unfortunately inherited this trait from JLM in an effort to easily be able to interoperate with JLM. Some approaches to tackle this problem in the RVSDG dialect have been identified: The first one is to create a high-level reference type which is used internally in the RVSDG dialect and other dialects that build on the RVSDG dialect. Another is to create a type-interface which can be implemented by the types of external dialects to allow them to model their own references and create types which operate on said references. The third option is to instead create types for the lambda node itself and model references using MLIR's pre-existing memref type [21]. The first two options are not mutually exclusive. Creating a type interface for RVSDG references and providing some default high-level implementation of said interface does not take much more work that just creating the type or just making the interface. Similarly, analyses and transformations that use the interface should just work on new implementations. The main cost of creating an interface is the work and research required to ensure that it is general enough to cover most potential use-cases, but specific enough to actually be useful for transformations. The added flexibility of the interface might however be worth the added complexity. Using MLIR memref types would make interoperability with pre-existing MLIR Ops easier and almost eliminate the maintenance effort of the reference types themselves, at the cost of flexibility. Using memref types in addition to a high level interface is also possible, although it would require special handling of memref types in some passes and analyses. A way to handle this as well is to attach the interface to the memref type, which can be done without making changes to the MLIR source code [17]. In short terms, creating a high-level reference type interface would be an extremely flexible solution that could incorporate a default high-level implementation, customized reference types from users of the dialect, and the default memref type. Reference types from other dialects could also be incorporated by attaching the interface to their reference types.

Overall, the chosen representations of RVSDG concepts in MLIR work well. They appear to be able to capture the structure of RVSDG, and traversal of the graph can be done in a straight forward manner by traversing MLIR IR's def-use chains [35]. While no RVSDG-specific optimizations have been implemented yet, there are no obvious roadblocks to doing so in the future.

5.3 The role of RVSDG in the MLIR ecosystem

While the main focus of this thesis is whether MLIR is a good framework for representing and working with RVSDG, an equally interesting question would be how RVSDG could fit into the MLIR ecosystem. What RVSDG provides is a way to represent code as a unified, data-flow based data structure suitable for performing optimizations. These are things that MLIR already provides by itself, and RVSDG does not provide too much additional benefit on these points. Where it can contribute however is with side effect based analysis and automatic parallelization. Representing the ordering of side-effecting operations as passing of SSA-values makes it easy to find the correct ordering of Ops by traversing the def-use chains of the state edges. This is not a mechanism MLIR provides by default, and order is instead enforced lexically or by special Ops that model parallel execution [24, 36, 37]. Because of this, RVSDG may be a good structure for IRs that desire automatic parallelization or other transformations that require knowledge of the ordering of side-effecting operations.

The creation of the JLM dialect as a separate dialect to the main RVSDG dialect was done to keep the RVSDG dialect itself as general purpose as possible. It does, however, also act as a small demonstration as to how the RVSDG dialect is intended to be used when developing more domain specific dialects. More specifically, it demonstrates the creation of new Ops that model computation at a given level of abstraction and interact with the RVSDG state and control types. Using a pattern like this makes the RVSDG dialect more flexible, as it does not enforce a given level of abstraction. The main drawback of this approach is that the complexity of the code will be greater than if all Ops and types were combined into a single dialect: Ops, types, and attributes will be defined by different projects. Several libraries will need to be linked. More header files will need to be included. This problem only compounds for compiler projects who wish to use several RVSDG based dialects to do progressive lowering. The flexibility and separation of duty gained by this approach should, however, be well worth the added complexity.

5.4 Future work

While the core dialect is more or less complete, there is still some work that needs doing. A good next step would be to decide how to best implement state edges. The alternative implementation discussed above where the state type holds a list of side effects and the corresponding resources they affect should be implemented and evaluated against the current system of several distinct state types.

Another aspect of the dialect that has not been thoroughly explored is its suitability as a basis for optimization passes. While RVSDG itself has been shown to be suitable for optimization [5], no transformation passes beyond some default passes included in MLIR have been implemented or tested. A good way to test this would be to re-implement the same optimizations that are already implemented in JLM. The simplest optimization to implement would probably be dead node elimination, as described by Reissmann et al. [5]. Implementing optimization passes should also reveal potential flaws in the representation of RVSDG that is used in the dialect.

As of right now, there are no good ways to enter or exit the RVSDG dialect. The mlir-print tool can generate some MLIR assembly that uses the RVSDG dialect, but preferably there should be a way to enter the dialect by directly constructing in-memory IR instead of generating text. As for exiting the MLIR dialect, there are a couple of options on the table. One approach would be to go the MLIR route and find a suitable set of dialects that the RVSDG dialect could be progressively lowered into. Alternatively, it would be possible to go back into JLM once the LLVM version problems have been resolved. Going back into JLM should be easier since the fundamental structure of the IR is near identical. After exiting to JLM, the RVSDG could be destructed into LLVM IR, which can then be converted to target code. Staying within MLIR would however make RVSDG more accessible to other compiler developers, since they could develop their entire compilation pipeline using MLIR. It would also allow compilers using RVSDG to benefit from future developments in MLIR.

JLM has shown that RVSDG can be used to represent low-level code such as LLVM IR [5], but so far there haven't really been any compiler implementations that use RVSDG to represent a program at a higher level of abstraction. Such a compiler would be a good demonstration of the RVSDG dialect, and would at the same time conclusively show that RVSDG can be used as an IR for high level languages.

CHAPTER 6

Conclusions

This thesis has presented a way to represent RVSDG in MLIR and an MLIR dialect which implements said representation. The dialect has been shown to seemingly be able to model all structural nodes and integrate side effect free Ops from other dialects as simple nodes. The creation of a domain-specific dialect that builds on top of the RVSDG dialect has also been demonstrated through the creation of the JLM dialect. Applications of pre-existing MLIR optimization passes has been shown to work in some cases through the application of the built-in Common Subexpression Elimination pass. This dialect has however not yet been fully validated. A good next step would be to perform roundtripping between JLM and the RVSDG dialect to prove that the representations can model the same structure. Additionally, it would be beneficial to implement some optimization passes that work on the RVSDG dialect, since being suitable for optimizations is one of the main selling points of RVSDG as an IR. Despite this, the RVSDG dialect shows promise as a platform for future experimentation with RVSDG, and should, with some more refinements, serve to enable other compiler developers to use RVSDG as the basis for their own IRs.

References

[1] R.H. Dennard et al. “Design of ion-implanted MOSFET’s with very small physical dimensions”. In: *IEEE Journal of Solid-State Circuits* 9.5 (Oct. 1974). Conference Name: IEEE Journal of Solid-State Circuits, pp. 256–268. ISSN: 1558-173X. DOI: 10.1109/JSSC.1974.1050511.

[2] Nico Reissmann. “Principles, Techniques, and Tools for Explicit and Automatic Parallelization”. eng. Accepted: 2019-05-14T11:27:39Z ISSN: 1503-8181. Doctoral thesis. NTNU, 2019. URL: https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/2597564 (visited on 06/05/2023).

[3] Chris Lattner et al. “MLIR: Scaling Compiler Infrastructure for Domain Specific Computation”. In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2021, pp. 2–14. DOI: 10.1109/CGO51591.2021.9370308.

[4] Helge Bahmann et al. “Perfect Reconstructability of Control Flow from Demand Dependence Graphs”. en. In: *ACM Transactions on Architecture and Code Optimization* 11.4 (Jan. 2015), pp. 1–25. ISSN: 1544-3566, 1544-3973. DOI: 10.1145/2693261. URL: https://dl.acm.org/doi/10.1145/2693261 (visited on 06/02/2023).

[5] Nico Reissmann et al. “RVSDG: An Intermediate Representation for Optimizing Compilers”. In: *ACM Transactions on Embedded Computing Systems* 19.6 (Nov. 2020), pp. 1–28. DOI: 10.1145/3391902. URL: https://doi.org/10.1145/3391902.

[6] Nico Reissmann. *phate/jlm-eval-suite*. original-date: 2019-07-01T12:43:52Z. May 2022. URL: https://github.com/phate/jlm-eval-suite (visited on 12/03/2022).

[7] Halvor Bjørstad. *Implementing RVSDG in MLIR: The beginning of a journey.* Specialization project. NTNU, Dec. 2022. URL: https://github.com/Riphiphip/rapport_fordypningsprosjekt_H22/releases/download/v1.0.0/Fordypningsprosjekt_H22.pdf.

[8] Chris Lattner. “LLVM: An Infrastructure for Multi-Stage Optimization”. *See* http://llvm.cs.uiuc.edu. MA thesis. Urbana, IL: Computer Science Dept., University of Illinois at Urbana-Champaign, Dec. 2002.

[9] *The LLVM Compiler Infrastructure Project.* URL: https://llvm.org/ (visited on 11/23/2022).

[10] *LLVM users.* URL: https://llvm.org/Users.html (visited on 11/23/2022).

[11] *The LLVM Compiler Infrastructure Project.* URL: https://llvm.org/Features.html (visited on 04/25/2023).

[12] *LLVM Language Reference Manual — LLVM 16.0.0git documentation.* URL: https://llvm.org/docs/LangRef.html (visited on 11/23/2022).

[13] *TableGen Overview — LLVM 16.0.0git documentation.* URL: https://llvm.org/docs/TableGen/ (visited on 12/02/2022).

[14] *Operation Definition Specification (ODS) - MLIR.* URL: https://mlir.llvm.org/docs/OpDefinitions/ (visited on 12/02/2022).

[15] *Operation Definition Specification (ODS) - MLIR.* URL: https://mlir.llvm.org/docs/DefiningDialects/Operations/#declarative-assembly-format (visited on 12/09/2022).

[16] *Defining Dialect Attributes and Types - MLIR.* URL: https://mlir.llvm.org/docs/DefiningDialects/AttributesAndTypes/ (visited on 05/24/2023).

[17] *Interfaces - MLIR.* URL: https://mlir.llvm.org/docs/Interfaces/ (visited on 04/28/2023).

[18] *Table-driven Declarative Rewrite Rule (DRR) - MLIR.* URL: https://mlir.llvm.org/docs/DeclarativeRewrites/ (visited on 05/24/2023).

[19] *1 TableGen Programmer's Reference — LLVM 17.0.0git documentation.* URL: https://llvm.org/docs/TableGen/ProgRef.html (visited on 05/24/2023).

[20] *'arith' Dialect - MLIR.* URL: https://mlir.llvm.org/docs/Dialects/ArithOps/ (visited on 04/26/2023).

[21] *'memref' Dialect - MLIR.* URL: https://mlir.llvm.org/docs/Dialects/MemRef/ (visited on 04/26/2023).

[22] *'llvm' Dialect - MLIR.* URL: https://mlir.llvm.org/docs/Dialects/LLVM/ (visited on 11/25/2022).

[23] *LLVM Language Reference Manual — LLVM 17.0.0git documentation.* URL: https://llvm.org/docs/LangRef.html#phi-instruction (visited on 04/27/2023).

[24] *MLIR Language Reference - MLIR.* URL: https://mlir.llvm.org/docs/LangRef/ (visited on 11/30/2022).

[25] *Pass Infrastructure - MLIR.* URL: https://mlir.llvm.org/docs/PassManagement/#preserving-analyses (visited on 04/28/2023).

[26] *Traits - MLIR.* URL: https://mlir.llvm.org/docs/Traits/ (visited on 04/28/2023).

[27] *Side Effects & Speculation - MLIR.* URL: https://mlir.llvm.org/docs/Rationale/SideEffectsAndSpeculation/ (visited on 05/22/2023).

[28] *MLIR Bytecode Format - MLIR.* URL: https://mlir.llvm.org/docs/BytecodeFormat/#mlir-encoding (visited on 05/01/2023).

[29] *Chapter 2: Emitting Basic MLIR - MLIR.* URL: https://mlir.llvm.org/docs/Tutorials/Toy/Ch-2/ (visited on 06/06/2023).

[30] *Symbols and Symbol Tables - MLIR.* URL: https://mlir.llvm.org/docs/SymbolsAndSymbolTables/ (visited on 12/02/2022).

[31] *'func' Dialect - MLIR.* URL: https://mlir.llvm.org/docs/Dialects/Func/ (visited on 12/02/2022).

[32] *MLIR : Language Server Protocol - MLIR.* URL: https://mlir.llvm.org/docs/Tools/MLIRLSP/ (visited on 05/07/2023).

[33] *opt - LLVM optimizer — LLVM 17.0.0git documentation.* URL: https://llvm.org/docs/CommandGuide/opt.html (visited on 05/07/2023).

[34] *Defining Dialects - MLIR*. URL: `https://mlir.llvm.org/docs/DefiningDialects/` (visited on 05/31/2023).

[35] *Understanding the IR Structure - MLIR*. URL: `https://mlir.llvm.org/docs/Tutorials/UnderstandingTheIRStructure/` (visited on 05/22/2023).

[36] *'omp' Dialect - MLIR*. URL: `https://mlir.llvm.org/docs/Dialects/OpenMPDialect/` (visited on 05/23/2023).

[37] *'async' Dialect - MLIR*. URL: `https://mlir.llvm.org/docs/Dialects/AsyncDialect/` (visited on 05/23/2023).

APPENDICES

APPENDIX A

GitHub repositories

The code for this project can be found on my personal GitHub account

GitHub repository links

- Main code repository: `https://github.com/Riphiphip/mlir_rvsdg`
- JLM fork: `https://github.com/Riphiphip/jlm/tree/mlir-print`

APPENDIX B

Information for developers

This appendix details information that should be useful for continuing work on the RVSDG dialect library. In particular, it will cover project setup, building the code, and how to navigate the codebase. Up-to-date instructions can also be found in the repository's README file. Any instructions written here have ONLY been tested on Ubuntu 22.04.

Getting started

What's in the box?

The following can be found in the RVSDG dialect library GitHub repository:

- The RVSDG dialect library which is a representation of pure RVSDG in MLIR
  - TableGen files can be found under ‘include/RVSDG‘
  - Header files can be found under ‘include/RVSDG‘
  - C++ source files can be found under ‘lib/RVSDG‘
- The JLM dialect library which builds on the RVSDG library and contains structures used for interoperation between RVSDG and JLM
  - TableGen files can be found under ‘include/JLM‘
  - Header files can be found under ‘include/JLM‘
  - C++ source files can be found under ‘lib/JLM‘
- The ‘rvsdg-opt‘ tool which is a tool for parsing MLIR assembly that contains RVSDG and JLM dialects, and performing various transformations on the RVSDG
  - The tool can be found under ‘rvsdg-opt‘
- The ‘rvsdg-lsp-server‘ tool which is an expansion of the MLIR language server that adds support for the RVSDG and JLM dialects
  - The tool can be found under ‘rvsdg-lsp-server‘

Software dependencies

- LLVM 16.0.0
- MLIR 16.0.0
- Build tools:
  - Clang 16 or another C++ compiler with support for C++17 and #pragma once
  - CMake $\geq$ 3.13.4
  - Ninja-build

Building the project

- Ensure all software dependencies have been installed and are working correctly
- Set the LLVM_DIR and MLIR_DIR environment variables to the paths containing the CMake directories for your LLVM and MLIR installations, respectively.
  - The value for LLVM_DIR can often be found by running

    ```
    llvm-config-16 --cmakedir
    ```

  - The value for MLIR_DIR can often be found by running

    ```
    llvm-config-16 --prefix
    ```

    and appending /lib/cmake/mlir to the result
- Create a directory named ‘build‘ within the project root directory
- Run the following commands from inside the ‘build‘ directory:

  ```
  cmake .. -GNinja
  cmake --build .
  ```

- After the build has completed, the library binaries can be found in build/lib and the ‘rvsdg-opt‘ and ‘rvsdg-lsp-server‘ tools can be found under build/bin
- A package containing library binaries and headers can be created by running

  ```
  ninja package
  ```

  from within the build directory

Dev container

The GitHub repository for the dialect library contains a dockerfile which can be used to build a development container for working on the RVSDG dialect library. This container contains all required library and tool dependencies and builds the RVSDG dialect library and mlir-print tool, both as a sanity check for the container itself, and so they can be used as reference during development. The repository also provides a *devcontainer.json* file which provides a set of recommended Visual Studio Code settings and extensions. Combined, these files can be used with the VSCode dev container extension to easily set up and maintain a development environment for the library.

Using the development container requires a local installation of Docker.

Using rvsdg-opt and rvsdg-lsp-server

The rvsdg-lsp-server is not meant to be used directly, but in conjunction with an IDE which supports it. It provides intellisense, hover information, code completion and other utilities. The language server is confirmed to work with Visual Studio Code's MLIR extension.

The rvsdg-opt tool is a command line utility based on mlir-opt, but incorporating the RVSDG and JLM dialects. This tool can be used to parse and run passes on MLIR assembly that uses the RVSDG and JLM dialects. Some examples of common uses are:

- Roundtrip

  ```
  rvsdg-opt --no-implicit-module <file> | rvsdg-opt
  ```

- Run Op name printing pass

  ```
  rvsdg-opt --rvsdg-print-export --no-implicit-module <file>
  ```

- Print as generic MLIR assembly

  ```
  rvsdg-opt --mlir-print-op-generic --no-implicit-module <file>
  ```

The "–no-implicit-module" flag prevents MLIR from wrapping the IR in a top-level "builtin.module" Op. This role is taken by the omega-node in the RVSDG dialect.

APPENDIX C

Larger examples

These examples have been generated by the mlir-print tool that was added to JLM for this project.

test-fprintf

This example shows a program that calls the external function fprintf represented in the RVSDG dialect. The field types for the _IO_FILE structs have been removed for readability.

C source code

```c
#include <stdio.h>
int main()
{
    fprintf(stderr, "%d * %d = %d", 3, 3, 9);
}
```

RVSDG dialect representation

```
rvsdg.omegaNode (
    %0: !llvm.ptr<!llvm.ptr<!llvm.struct<"struct._IO_FILE", ...>>>,
    %1: !rvsdg.lambdaRef<
            (!llvm.ptr<!llvm.struct<"struct._IO_FILE", ...>>,
 ↪  !llvm.ptr<i8>, !jlm.varargList, !rvsdg.ioState,
 ↪  !rvsdg.memState, !rvsdg.loopState)
            ->
            (i32, !rvsdg.ioState, !rvsdg.memState,
 ↪  !rvsdg.loopState)
        >
): {
    %2 = rvsdg.deltaNode():
    (): {
        %3 = arith.constant 0: i8
        %4 = arith.constant 37: i8
        %5 = arith.constant 32: i8
        %6 = arith.constant 61: i8
        %7 = arith.constant 37: i8
        %8 = arith.constant 100: i8
        %9 = arith.constant 42: i8
        %10 = arith.constant 32: i8
        %11 = arith.constant 32: i8
        %12 = arith.constant 100: i8
        %13 = arith.constant 100: i8
        %14 = arith.constant 32: i8
        %15 = arith.constant 37: i8
        %16 = llvm.mlir.undef: !llvm.array<13 x i8>
        %17 = llvm.insertvalue %15, %16[0]: !llvm.array<13 x i8>
        %18 = llvm.insertvalue %12, %17[1]: !llvm.array<13 x i8>
        %19 = llvm.insertvalue %10, %18[2]: !llvm.array<13 x i8>
        %20 = llvm.insertvalue %9, %19[3]: !llvm.array<13 x i8>
        %21 = llvm.insertvalue %14, %20[4]: !llvm.array<13 x i8>
        %22 = llvm.insertvalue %7, %21[5]: !llvm.array<13 x i8>
        %23 = llvm.insertvalue %13, %22[6]: !llvm.array<13 x i8>
        %24 = llvm.insertvalue %11, %23[7]: !llvm.array<13 x i8>
        %25 = llvm.insertvalue %6, %24[8]: !llvm.array<13 x i8>
        %26 = llvm.insertvalue %5, %25[9]: !llvm.array<13 x i8>
        %27 = llvm.insertvalue %4, %26[10]: !llvm.array<13 x i8>
        %28 = llvm.insertvalue %8, %27[11]: !llvm.array<13 x i8>
        %29 = llvm.insertvalue %3, %28[12]: !llvm.array<13 x i8>
        rvsdg.deltaResult(%29:!llvm.array<13 x i8>)
    }->!llvm.ptr<!llvm.array<13 x i8>>
    %30 = rvsdg.lambdaNode <(!rvsdg.ioState, !rvsdg.memState,
↪   !rvsdg.loopState)->(i32, !rvsdg.ioState, !rvsdg.memState,
↪   !rvsdg.loopState)> (
        %1:
↪   !rvsdg.lambdaRef<(!llvm.ptr<!llvm.struct<"struct._IO_FILE",
↪   ...>>, !llvm.ptr<i8>, !jlm.varargList, !rvsdg.ioState,
↪   !rvsdg.memState, !rvsdg.loopState) -> (i32, !rvsdg.ioState,
↪   !rvsdg.memState, !rvsdg.loopState)>,
        %0: !llvm.ptr<!llvm.ptr<!llvm.struct<"struct._IO_FILE",
↪   ...>>>,
        %2: !llvm.ptr<!llvm.array<13 x i8>>
    ):
        (%31: !rvsdg.ioState, %32: !rvsdg.memState, %33:
↪   !rvsdg.loopState, %34:
↪   !rvsdg.lambdaRef<(!llvm.ptr<!llvm.struct<"struct._IO_FILE",
↪   ...>>, !llvm.ptr<i8>, !jlm.varargList, !rvsdg.ioState,
↪   !rvsdg.memState, !rvsdg.loopState) -> (i32, !rvsdg.ioState,
↪   !rvsdg.memState, !rvsdg.loopState)>, %35:
↪   !llvm.ptr<!llvm.ptr<!llvm.struct<"struct._IO_FILE", ...>>>,
↪   %36: !llvm.ptr<!llvm.array<13 x i8>>): {
            %37, %38 = jlm.load %35:
↪   !llvm.ptr<!llvm.ptr<!llvm.struct<"struct._IO_FILE", ...>>>
↪   (%32) -> !llvm.ptr<!llvm.struct<"struct._IO_FILE", ...>>,
↪   !rvsdg.memState
            %39 = arith.constant 0: i32
            %40 = arith.constant 9: i32
            %41 = arith.constant 3: i32
            %42 = arith.constant 3: i32
            %43 = arith.constant 0: i64
            %44 = arith.constant 0: i64
            %45 = llvm.getelementptr %36[%44, %43]:
↪   (!llvm.ptr<!llvm.array<13 x i8>>, i64, i64) -> !llvm.ptr<i8>
            %46 = jlm.createVarargs (%42: i32, %41: i32, %40: i32)
↪   -> !jlm.varargList
            %47, %48, %49, %50 = rvsdg.applyNode %34 :
↪   !rvsdg.lambdaRef<(!llvm.ptr<!llvm.struct<"struct._IO_FILE",
↪   ...>>, !llvm.ptr<i8>, !jlm.varargList, !rvsdg.ioState,
↪   !rvsdg.memState, !rvsdg.loopState) -> (i32, !rvsdg.ioState,
↪   !rvsdg.memState, !rvsdg.loopState)>(%37 :
↪   !llvm.ptr<!llvm.struct<"struct._IO_FILE", ...>>, %45 :
↪   !llvm.ptr<i8>, %46 : !jlm.varargList, %31 : !rvsdg.ioState, %38
↪   : !rvsdg.memState, %33 : !rvsdg.loopState) -> i32,
↪   !rvsdg.ioState, !rvsdg.memState, !rvsdg.loopState
            rvsdg.lambdaResult(%39:i32, %48:!rvsdg.ioState,
↪   %49:!rvsdg.memState, %50:!rvsdg.loopState)
        }
    rvsdg.omegaResult(%2:!llvm.ptr<!llvm.array<13 x i8>>,
↪   %30:!rvsdg.lambdaRef<(!rvsdg.ioState, !rvsdg.memState,
↪   !rvsdg.loopState) -> (i32, !rvsdg.ioState, !rvsdg.memState,
↪   !rvsdg.loopState)>)
}

```

test-do-while

This example shows a program that calculates and prints the sum of all positive integers lower than 5.

C source code

```c
#include <assert.h>
#include <stdio.h>

static unsigned int f(unsigned int n) {
        unsigned int s = 0;
        do {
                s += n;
        } while(n--);

        return s;
}

int main() {
        unsigned int s = f(5);
        printf("%d\n", s);
        assert(s == 15);
}
```

RVSDG dialect representation

```
rvsdg.omegaNode (%0: !rvsdg.lambdaRef<(!llvm.ptr<i8>,
 ↪  !llvm.ptr<i8>, i32, !llvm.ptr<i8>, !rvsdg.ioState,
 ↪  !rvsdg.memState, !rvsdg.loopState) -> (!rvsdg.ioState,
 ↪  !rvsdg.memState, !rvsdg.loopState)>, %1:
 ↪  !rvsdg.lambdaRef<(!llvm.ptr<i8>, !jlm.varargList,
 ↪  !rvsdg.ioState, !rvsdg.memState, !rvsdg.loopState) -> (i32,
 ↪  !rvsdg.ioState, !rvsdg.memState, !rvsdg.loopState)>): {
    %2 = rvsdg.lambdaNode <(i32, !rvsdg.ioState, !rvsdg.memState,
 ↪  !rvsdg.loopState)->(i32, !rvsdg.ioState, !rvsdg.memState,
 ↪  !rvsdg.loopState)> ():
        (%3: i32, %4: !rvsdg.ioState, %5: !rvsdg.memState, %6:
 ↪  !rvsdg.loopState): {
            %7 = llvm.mlir.undef: i32
            %8 = arith.constant 0: i32
            %9, %10, %11, %12, %13, %14 = rvsdg.thetaNode(%3: i32,
 ↪  %8: i32, %7: i32, %4: !rvsdg.ioState, %5: !rvsdg.memState, %6:
 ↪  !rvsdg.loopState):
            (%15: i32, %16: i32, %17: i32, %18: !rvsdg.ioState,
 ↪  %19: !rvsdg.memState, %20: !rvsdg.loopState): {
                %21 = llvm.add %16, %15: i32
                %22 = arith.constant 0: i32
                %23 = arith.constant -1: i32
            %24 = llvm.add %15, %23: i32
            %25 = llvm.icmp "ne" %15, %22: i32
            %26 = rvsdg.match(%25 : i1) [
                #rvsdg.matchRule<1 -> 1>,
                #rvsdg.matchRule<default -> 0>
            ] -> !rvsdg.ctrl<2>
            %27, %28, %29 = rvsdg.gammaNode(%26: !rvsdg.ctrl<2>)(%24: i32, %15: i32, %21: i32, %16: i32): [
                (%30: i32, %31: i32, %32: i32, %33: i32): {
                    %34 = rvsdg.constantCtrl 0: !rvsdg.ctrl<2>
                    rvsdg.gammaResult(%34:!rvsdg.ctrl<2>, %31:i32, %33:i32)
                },
                (%35: i32, %36: i32, %37: i32, %38: i32): {
                    %39 = rvsdg.constantCtrl 1: !rvsdg.ctrl<2>
                    rvsdg.gammaResult(%39:!rvsdg.ctrl<2>, %35:i32, %37:i32)
                }
            ]->!rvsdg.ctrl<2>, i32, i32
            rvsdg.thetaResult(%27): (%28:i32, %29:i32, %21:i32, %18:!rvsdg.ioState, %19:!rvsdg.memState, %20:!rvsdg.loopState)
        }->i32, i32, i32, !rvsdg.ioState, !rvsdg.memState, !rvsdg.loopState
        rvsdg.lambdaResult(%11:i32, %12:!rvsdg.ioState, %13:!rvsdg.memState, %14:!rvsdg.loopState)
    }
%40 = rvsdg.deltaNode():
(): {
    %41 = arith.constant 40: i8
    %42 = arith.constant 110: i8
    %43 = arith.constant 0: i8
    %44 = arith.constant 105: i8
    %45 = arith.constant 97: i8
    %46 = arith.constant 109: i8
    %47 = arith.constant 32: i8
    %48 = arith.constant 41: i8
    %49 = arith.constant 116: i8
    %50 = arith.constant 110: i8
    %51 = arith.constant 105: i8
    %52 = llvm.mlir.undef: !llvm.array<11 x i8>
    %53 = llvm.insertvalue %51, %52[0]: !llvm.array<11 x i8>
    %54 = llvm.insertvalue %50, %53[1]: !llvm.array<11 x i8>
    %55 = llvm.insertvalue %49, %54[2]: !llvm.array<11 x i8>
    %56 = llvm.insertvalue %47, %55[3]: !llvm.array<11 x i8>
    %57 = llvm.insertvalue %46, %56[4]: !llvm.array<11 x i8>
    %58 = llvm.insertvalue %45, %57[5]: !llvm.array<11 x i8>
    %59 = llvm.insertvalue %44, %58[6]: !llvm.array<11 x i8>
    %60 = llvm.insertvalue %42, %59[7]: !llvm.array<11 x i8>
```

```
        %61 = llvm.insertvalue %41, %60[8]: !llvm.array<11 x i8>
        %62 = llvm.insertvalue %48, %61[9]: !llvm.array<11 x i8>
        %63 = llvm.insertvalue %43, %62[10]: !llvm.array<11 x i8>
        rvsdg.deltaResult(%63:!llvm.array<11 x i8>)
    }->!llvm.ptr<!llvm.array<11 x i8>>
    %64 = rvsdg.deltaNode():
    (): {
        %65 = arith.constant 0: i8
        %66 = arith.constant 99: i8
        %67 = arith.constant 46: i8
        %68 = arith.constant 115: i8
        %69 = arith.constant 116: i8
        %70 = arith.constant 45: i8
        %71 = arith.constant 47: i8
        %72 = arith.constant 115: i8
        %73 = arith.constant 119: i8
        %74 = arith.constant 115: i8
        %75 = arith.constant 101: i8
        %76 = arith.constant 101: i8
        %77 = arith.constant 47: i8
        %78 = arith.constant 101: i8
        %79 = arith.constant 99: i8
        %80 = arith.constant 46: i8
        %81 = arith.constant 105: i8
        %82 = arith.constant 116: i8
        %83 = arith.constant 115: i8
        %84 = arith.constant 47: i8
        %85 = arith.constant 116: i8
        %86 = arith.constant 116: i8
        %87 = arith.constant 101: i8
        %88 = arith.constant 115: i8
        %89 = arith.constant 116: i8
        %90 = arith.constant 45: i8
        %91 = arith.constant 100: i8
        %92 = arith.constant 111: i8
        %93 = arith.constant 116: i8
        %94 = arith.constant 45: i8
        %95 = arith.constant 104: i8
        %96 = arith.constant 108: i8
        %97 = llvm.mlir.undef: !llvm.array<32 x i8>
        %98 = llvm.insertvalue %80, %97[0]: !llvm.array<32 x i8>
        %99 = llvm.insertvalue %77, %98[1]: !llvm.array<32 x i8>
        %100 = llvm.insertvalue %93, %99[2]: !llvm.array<32 x i8>
        %101 = llvm.insertvalue %76, %100[3]: !llvm.array<32 x i8>
        %102 = llvm.insertvalue %72, %101[4]: !llvm.array<32 x i8>
        %103 = llvm.insertvalue %85, %102[5]: !llvm.array<32 x i8>
        %104 = llvm.insertvalue %74, %103[6]: !llvm.array<32 x i8>
        %105 = llvm.insertvalue %71, %104[7]: !llvm.array<32 x i8>
        %106 = llvm.insertvalue %79, %105[8]: !llvm.array<32 x i8>
        %107 = llvm.insertvalue %70, %106[9]: !llvm.array<32 x i8>
        %108 = llvm.insertvalue %69, %107[10]: !llvm.array<32 x i8>
        %109 = llvm.insertvalue %75, %108[11]: !llvm.array<32 x i8>
        %110 = llvm.insertvalue %68, %109[12]: !llvm.array<32 x i8>
        %111 = llvm.insertvalue %82, %110[13]: !llvm.array<32 x i8>
        %112 = llvm.insertvalue %83, %111[14]: !llvm.array<32 x i8>
        %113 = llvm.insertvalue %84, %112[15]: !llvm.array<32 x i8>
        %114 = llvm.insertvalue %86, %113[16]: !llvm.array<32 x i8>
        %115 = llvm.insertvalue %87, %114[17]: !llvm.array<32 x i8>
        %116 = llvm.insertvalue %88, %115[18]: !llvm.array<32 x i8>
        %117 = llvm.insertvalue %89, %116[19]: !llvm.array<32 x i8>
        %118 = llvm.insertvalue %90, %117[20]: !llvm.array<32 x i8>
        %119 = llvm.insertvalue %91, %118[21]: !llvm.array<32 x i8>
        %120 = llvm.insertvalue %92, %119[22]: !llvm.array<32 x i8>
        %121 = llvm.insertvalue %94, %120[23]: !llvm.array<32 x i8>
        %122 = llvm.insertvalue %73, %121[24]: !llvm.array<32 x i8>
        %123 = llvm.insertvalue %95, %122[25]: !llvm.array<32 x i8>
        %124 = llvm.insertvalue %81, %123[26]: !llvm.array<32 x i8>
        %125 = llvm.insertvalue %96, %124[27]: !llvm.array<32 x i8>
        %126 = llvm.insertvalue %78, %125[28]: !llvm.array<32 x i8>
        %127 = llvm.insertvalue %67, %126[29]: !llvm.array<32 x i8>
        %128 = llvm.insertvalue %66, %127[30]: !llvm.array<32 x i8>
        %129 = llvm.insertvalue %65, %128[31]: !llvm.array<32 x i8>
        rvsdg.deltaResult(%129:!llvm.array<32 x i8>)
    }->!llvm.ptr<!llvm.array<32 x i8>>
    %130 = rvsdg.deltaNode():
    (): {
        %131 = arith.constant 0: i8
        %132 = arith.constant 53: i8
        %133 = arith.constant 49: i8
        %134 = arith.constant 32: i8
        %135 = arith.constant 61: i8
        %136 = arith.constant 32: i8
        %137 = arith.constant 61: i8
        %138 = arith.constant 115: i8
        %139 = llvm.mlir.undef: !llvm.array<8 x i8>
        %140 = llvm.insertvalue %138, %139[0]: !llvm.array<8 x i8>
        %141 = llvm.insertvalue %136, %140[1]: !llvm.array<8 x i8>
        %142 = llvm.insertvalue %137, %141[2]: !llvm.array<8 x i8>
        %143 = llvm.insertvalue %135, %142[3]: !llvm.array<8 x i8>
        %144 = llvm.insertvalue %134, %143[4]: !llvm.array<8 x i8>
        %145 = llvm.insertvalue %133, %144[5]: !llvm.array<8 x i8>
        %146 = llvm.insertvalue %132, %145[6]: !llvm.array<8 x i8>
        %147 = llvm.insertvalue %131, %146[7]: !llvm.array<8 x i8>
        rvsdg.deltaResult(%147:!llvm.array<8 x i8>)
    }->!llvm.ptr<!llvm.array<8 x i8>>
    %148 = rvsdg.deltaNode():
```

```
    (): {
        %149 = arith.constant 0: i8
        %150 = arith.constant 10: i8
        %151 = arith.constant 100: i8
        %152 = arith.constant 37: i8
        %153 = llvm.mlir.undef: !llvm.array<4 x i8>
        %154 = llvm.insertvalue %152, %153[0]: !llvm.array<4 x i8>
        %155 = llvm.insertvalue %151, %154[1]: !llvm.array<4 x i8>
        %156 = llvm.insertvalue %150, %155[2]: !llvm.array<4 x i8>
        %157 = llvm.insertvalue %149, %156[3]: !llvm.array<4 x i8>
        rvsdg.deltaResult(%157:!llvm.array<4 x i8>)
    }->!llvm.ptr<!llvm.array<4 x i8>>
    %158 = rvsdg.lambdaNode <(!rvsdg.ioState, !rvsdg.memState,
↪   !rvsdg.loopState)->(i32, !rvsdg.ioState, !rvsdg.memState,
↪   !rvsdg.loopState)> (%1: !rvsdg.lambdaRef<(!llvm.ptr<i8>,
↪   !jlm.varargList, !rvsdg.ioState, !rvsdg.memState,
↪   !rvsdg.loopState) -> (i32, !rvsdg.ioState, !rvsdg.memState,
↪   !rvsdg.loopState)>, %148: !llvm.ptr<!llvm.array<4 x i8>>, %2:
↪   !rvsdg.lambdaRef<(i32, !rvsdg.ioState, !rvsdg.memState,
↪   !rvsdg.loopState) -> (i32, !rvsdg.ioState, !rvsdg.memState,
↪   !rvsdg.loopState)>, %130: !llvm.ptr<!llvm.array<8 x i8>>, %64:
↪   !llvm.ptr<!llvm.array<32 x i8>>, %40: !llvm.ptr<!llvm.array<11
↪   x i8>>, %0: !rvsdg.lambdaRef<(!llvm.ptr<i8>, !llvm.ptr<i8>,
↪   i32, !llvm.ptr<i8>, !rvsdg.ioState, !rvsdg.memState,
↪   !rvsdg.loopState) -> (!rvsdg.ioState, !rvsdg.memState,
↪   !rvsdg.loopState)>):
        (%159: !rvsdg.ioState, %160: !rvsdg.memState, %161:
↪   !rvsdg.loopState, %162: !rvsdg.lambdaRef<(!llvm.ptr<i8>,
↪   !jlm.varargList, !rvsdg.ioState, !rvsdg.memState,
↪   !rvsdg.loopState) -> (i32, !rvsdg.ioState, !rvsdg.memState,
↪   !rvsdg.loopState)>, %163: !llvm.ptr<!llvm.array<4 x i8>>, %164:
↪   !rvsdg.lambdaRef<(i32, !rvsdg.ioState, !rvsdg.memState,
↪   !rvsdg.loopState) -> (i32, !rvsdg.ioState, !rvsdg.memState,
↪   !rvsdg.loopState)>, %165: !llvm.ptr<!llvm.array<8 x i8>>, %166:
↪   !llvm.ptr<!llvm.array<32 x i8>>, %167: !llvm.ptr<!llvm.array<11
↪   x i8>>, %168: !rvsdg.lambdaRef<(!llvm.ptr<i8>, !llvm.ptr<i8>,
↪   i32, !llvm.ptr<i8>, !rvsdg.ioState, !rvsdg.memState,
↪   !rvsdg.loopState) -> (!rvsdg.ioState, !rvsdg.memState,
↪   !rvsdg.loopState)>): {
            %169 = arith.constant 15: i32
            %170 = arith.constant 0: i64
            %171 = arith.constant 0: i64
            %172 = arith.constant 5: i32
            %173 = llvm.mlir.undef: i32
        %174, %175, %176, %177 = rvsdg.applyNode %164 : !rvsdg.lambdaRef<(i32, !rvsdg.ioState, !rvsdg.memState, !rvsdg.loopState) -> (i32, !rvsdg.ioState, !rvsdg.memState, !rvsdg.loopState)>(%172 : i32, %159 : !rvsdg.ioState, %160 : !rvsdg.memState, %161 : !rvsdg.loopState) -> i32, !rvsdg.ioState, !rvsdg.memState, !rvsdg.loopState
        %178 = llvm.getelementptr %163[%171, %170]: (!llvm.ptr<!llvm.array<4 x i8>>, i64, i64) -> !llvm.ptr<i8>
        %179 = jlm.createVarargs (%174: i32) -> !jlm.varargList
        %180 = llvm.icmp "eq" %174, %169: i32
        %181 = rvsdg.match(%180 : i1) [
            #rvsdg.matchRule<1 -> 1>,
            #rvsdg.matchRule<default -> 0>
        ] -> !rvsdg.ctrl<2>
        %182, %183, %184, %185 = rvsdg.applyNode %162 : !rvsdg.lambdaRef<(!llvm.ptr<i8>, !jlm.varargList, !rvsdg.ioState, !rvsdg.memState, !rvsdg.loopState) -> (i32, !rvsdg.ioState, !rvsdg.memState, !rvsdg.loopState)>(%178 : !llvm.ptr<i8>, %179 : !jlm.varargList, %175 : !rvsdg.ioState, %176 : !rvsdg.memState, %177 : !rvsdg.loopState) -> i32, !rvsdg.ioState, !rvsdg.memState, !rvsdg.loopState
        %186, %187, %188, %189 = rvsdg.gammaNode(%181: !rvsdg.ctrl<2>)(%166: !llvm.ptr<!llvm.array<32 x i8>>, %165: !llvm.ptr<!llvm.array<8 x i8>>, %167: !llvm.ptr<!llvm.array<11 x i8>>, %173: i32, %168: !rvsdg.lambdaRef<(!llvm.ptr<i8>, !llvm.ptr<i8>, i32, !llvm.ptr<i8>, !rvsdg.ioState, !rvsdg.memState, !rvsdg.loopState) -> (!rvsdg.ioState, !rvsdg.memState, !rvsdg.loopState)>, %183: !rvsdg.ioState, %185: !rvsdg.loopState, %184: !rvsdg.memState): [
            (%190: !llvm.ptr<!llvm.array<32 x i8>>, %191: !llvm.ptr<!llvm.array<8 x i8>>, %192: !llvm.ptr<!llvm.array<11 x i8>>, %193: i32, %194: !rvsdg.lambdaRef<(!llvm.ptr<i8>, !llvm.ptr<i8>, i32, !llvm.ptr<i8>, !rvsdg.ioState, !rvsdg.memState, !rvsdg.loopState) -> (!rvsdg.ioState, !rvsdg.memState, !rvsdg.loopState)>, %195: !rvsdg.ioState, %196: !rvsdg.loopState, %197: !rvsdg.memState): {
                %198 = arith.constant 0: i64
                %199 = arith.constant 0: i64
                %200 = arith.constant 20: i32
                %201 = arith.constant 0: i64
                %202 = arith.constant 0: i64
                %203 = arith.constant 0: i64
                %204 = arith.constant 0: i64
                %205 = llvm.getelementptr %191[%204, %203]: (!llvm.ptr<!llvm.array<8 x i8>>, i64, i64) -> !llvm.ptr<i8>
                %206 = llvm.getelementptr %190[%202, %201]: (!llvm.ptr<!llvm.array<32 x i8>>, i64, i64) -> !llvm.ptr<i8>
```

```
                    %207 = llvm.getelementptr %192[%199, %198]:
↪   (!llvm.ptr<!llvm.array<11 x i8>>, i64, i64) -> !llvm.ptr<i8>
                    %208, %209, %210 = rvsdg.applyNode %194 :
↪   !rvsdg.lambdaRef<(!llvm.ptr<i8>, !llvm.ptr<i8>, i32,
↪   !llvm.ptr<i8>, !rvsdg.ioState, !rvsdg.memState,
↪   !rvsdg.loopState) -> (!rvsdg.ioState, !rvsdg.memState,
↪   !rvsdg.loopState)>(%205 : !llvm.ptr<i8>, %206 : !llvm.ptr<i8>,
↪   %200 : i32, %207 : !llvm.ptr<i8>, %195 : !rvsdg.ioState, %197 :
↪   !rvsdg.memState, %196 : !rvsdg.loopState) -> !rvsdg.ioState,
↪   !rvsdg.memState, !rvsdg.loopState
                    rvsdg.gammaResult(%193:i32,
↪   %208:!rvsdg.ioState, %210:!rvsdg.loopState,
↪   %209:!rvsdg.memState)
                },
                (%211: !llvm.ptr<!llvm.array<32 x i8>>, %212:
↪   !llvm.ptr<!llvm.array<8 x i8>>, %213: !llvm.ptr<!llvm.array<11
↪   x i8>>, %214: i32, %215: !rvsdg.lambdaRef<(!llvm.ptr<i8>,
↪   !llvm.ptr<i8>, i32, !llvm.ptr<i8>, !rvsdg.ioState,
↪   !rvsdg.memState, !rvsdg.loopState) -> (!rvsdg.ioState,
↪   !rvsdg.memState, !rvsdg.loopState)>, %216: !rvsdg.ioState,
↪   %217: !rvsdg.loopState, %218: !rvsdg.memState): {
                    %219 = arith.constant 0: i32
                    rvsdg.gammaResult(%219:i32,
↪   %216:!rvsdg.ioState, %217:!rvsdg.loopState,
↪   %218:!rvsdg.memState)
                }
            ]->i32, !rvsdg.ioState, !rvsdg.loopState,
↪   !rvsdg.memState
            rvsdg.lambdaResult(%186:i32, %187:!rvsdg.ioState,
↪   %189:!rvsdg.memState, %188:!rvsdg.loopState)
        }
    rvsdg.omegaResult(%148:!llvm.ptr<!llvm.array<4 x i8>>,
↪   %130:!llvm.ptr<!llvm.array<8 x i8>>,
↪   %64:!llvm.ptr<!llvm.array<32 x i8>>,
↪   %40:!llvm.ptr<!llvm.array<11 x i8>>,
↪   %158:!rvsdg.lambdaRef<(!rvsdg.ioState, !rvsdg.memState,
↪   !rvsdg.loopState) -> (i32, !rvsdg.ioState, !rvsdg.memState,
↪   !rvsdg.loopState)>)
}
```

test-rectfct

This example shows a program consists of one self-recursive function and two mutually recursive functions.

C++ source code

```
#include <stdio.h>

void f(unsigned n) {
        printf("%d\n", n);
        if (n > 0)
                f(n-1);
}

unsigned y(unsigned);

unsigned x(unsigned n) {
        return y(n);
}

unsigned y(unsigned n) {
        return x(n);
}

int main() {
        f(10);
        return 0;
}
```

RVSDG dialect representation

```
rvsdg.omegaNode (%0: !rvsdg.lambdaRef<(!llvm.ptr<i8>,
↪   !jlm.varargList, !rvsdg.ioState, !rvsdg.memState,
↪   !rvsdg.loopState) -> (i32, !rvsdg.ioState, !rvsdg.memState,
↪   !rvsdg.loopState)>): {
    %1, %2 = rvsdg.phiNode():
    (%3: !rvsdg.lambdaRef<(i32, !rvsdg.ioState, !rvsdg.memState,
↪   !rvsdg.loopState) -> (i32, !rvsdg.ioState, !rvsdg.memState,
↪   !rvsdg.loopState)>, %4: !rvsdg.lambdaRef<(i32, !rvsdg.ioState,
↪   !rvsdg.memState, !rvsdg.loopState) -> (i32, !rvsdg.ioState,
↪   !rvsdg.memState, !rvsdg.loopState)>): {
        %5 = rvsdg.lambdaNode <(i32, !rvsdg.ioState,
↪   !rvsdg.memState, !rvsdg.loopState)->(i32, !rvsdg.ioState,
↪   !rvsdg.memState, !rvsdg.loopState)> (%4: !rvsdg.lambdaRef<(i32,
↪   !rvsdg.ioState, !rvsdg.memState, !rvsdg.loopState) -> (i32,
↪   !rvsdg.ioState, !rvsdg.memState, !rvsdg.loopState)>):
        (%6: i32, %7: !rvsdg.ioState, %8: !rvsdg.memState, %9:
↪ !rvsdg.loopState, %10: !rvsdg.lambdaRef<(i32, !rvsdg.ioState,
↪ !rvsdg.memState, !rvsdg.loopState) -> (i32, !rvsdg.ioState,
↪ !rvsdg.memState, !rvsdg.loopState)>): {
            %11, %12, %13, %14 = rvsdg.applyNode %10 :
↪ !rvsdg.lambdaRef<(i32, !rvsdg.ioState, !rvsdg.memState,
↪ !rvsdg.loopState) -> (i32, !rvsdg.ioState, !rvsdg.memState,
↪ !rvsdg.loopState)>(%6 : i32, %7 : !rvsdg.ioState, %8 :
↪ !rvsdg.memState, %9 : !rvsdg.loopState) -> i32, !rvsdg.ioState,
↪ !rvsdg.memState, !rvsdg.loopState
            rvsdg.lambdaResult(%11:i32, %12:!rvsdg.ioState,
↪ %13:!rvsdg.memState, %14:!rvsdg.loopState)
        }
    %15 = rvsdg.lambdaNode <(i32, !rvsdg.ioState,
↪ !rvsdg.memState, !rvsdg.loopState)->(i32, !rvsdg.ioState,
↪ !rvsdg.memState, !rvsdg.loopState)> (%3: !rvsdg.lambdaRef<(i32,
↪ !rvsdg.ioState, !rvsdg.memState, !rvsdg.loopState) -> (i32,
↪ !rvsdg.ioState, !rvsdg.memState, !rvsdg.loopState)>):
        (%16: i32, %17: !rvsdg.ioState, %18: !rvsdg.memState,
↪ %19: !rvsdg.loopState, %20: !rvsdg.lambdaRef<(i32,
↪ !rvsdg.ioState, !rvsdg.memState, !rvsdg.loopState) -> (i32,
↪ !rvsdg.ioState, !rvsdg.memState, !rvsdg.loopState)>): {
            %21, %22, %23, %24 = rvsdg.applyNode %20 :
↪ !rvsdg.lambdaRef<(i32, !rvsdg.ioState, !rvsdg.memState,
↪ !rvsdg.loopState) -> (i32, !rvsdg.ioState, !rvsdg.memState,
↪ !rvsdg.loopState)>(%16 : i32, %17 : !rvsdg.ioState, %18 :
↪ !rvsdg.memState, %19 : !rvsdg.loopState) -> i32,
↪ !rvsdg.ioState, !rvsdg.memState, !rvsdg.loopState
            rvsdg.lambdaResult(%21:i32, %22:!rvsdg.ioState,
↪ %23:!rvsdg.memState, %24:!rvsdg.loopState)
        }
    rvsdg.phiResult(%5:!rvsdg.lambdaRef<(i32, !rvsdg.ioState,
↪ !rvsdg.memState, !rvsdg.loopState) -> (i32, !rvsdg.ioState,
↪ !rvsdg.memState, !rvsdg.loopState)>, %15:!rvsdg.lambdaRef<(i32,
↪ !rvsdg.ioState, !rvsdg.memState, !rvsdg.loopState) -> (i32,
↪ !rvsdg.ioState, !rvsdg.memState, !rvsdg.loopState)>)
}->!rvsdg.lambdaRef<(i32, !rvsdg.ioState, !rvsdg.memState,
↪ !rvsdg.loopState) -> (i32, !rvsdg.ioState, !rvsdg.memState,
↪ !rvsdg.loopState)>, !rvsdg.lambdaRef<(i32, !rvsdg.ioState,
↪ !rvsdg.memState, !rvsdg.loopState) -> (i32, !rvsdg.ioState,
↪ !rvsdg.memState, !rvsdg.loopState)>
%25 = rvsdg.deltaNode():
(): {
    %26 = arith.constant 0: i8
    %27 = arith.constant 10: i8
    %28 = arith.constant 100: i8
    %29 = arith.constant 37: i8
    %30 = llvm.mlir.undef: !llvm.array<4 x i8>
    %31 = llvm.insertvalue %29, %30[0]: !llvm.array<4 x i8>
    %32 = llvm.insertvalue %28, %31[1]: !llvm.array<4 x i8>
    %33 = llvm.insertvalue %27, %32[2]: !llvm.array<4 x i8>
    %34 = llvm.insertvalue %26, %33[3]: !llvm.array<4 x i8>
    rvsdg.deltaResult(%34:!llvm.array<4 x i8>)
  }->!llvm.ptr<!llvm.array<4 x i8>>
  %35 = rvsdg.phiNode(%0: !rvsdg.lambdaRef<(!llvm.ptr<i8>,
↪ !jlm.varargList, !rvsdg.ioState, !rvsdg.memState,
↪ !rvsdg.loopState) -> (i32, !rvsdg.ioState, !rvsdg.memState,
↪ !rvsdg.loopState)>, %25: !llvm.ptr<!llvm.array<4 x i8>>):
  (%36: !rvsdg.lambdaRef<(i32, !rvsdg.ioState, !rvsdg.memState,
↪ !rvsdg.loopState) -> (!rvsdg.ioState, !rvsdg.memState,
↪ !rvsdg.loopState)>, %37: !rvsdg.lambdaRef<(!llvm.ptr<i8>,
↪ !jlm.varargList, !rvsdg.ioState, !rvsdg.memState,
↪ !rvsdg.loopState) -> (i32, !rvsdg.ioState, !rvsdg.memState,
↪ !rvsdg.loopState)>, %38: !llvm.ptr<!llvm.array<4 x i8>>): {
    %39 = rvsdg.lambdaNode <(i32, !rvsdg.ioState,
↪ !rvsdg.memState, !rvsdg.loopState)->(!rvsdg.ioState,
↪ !rvsdg.memState, !rvsdg.loopState)> (%38:
↪ !llvm.ptr<!llvm.array<4 x i8>>, %37:
↪ !rvsdg.lambdaRef<(!llvm.ptr<i8>, !jlm.varargList,
↪ !rvsdg.ioState, !rvsdg.memState, !rvsdg.loopState) -> (i32,
↪ !rvsdg.ioState, !rvsdg.memState, !rvsdg.loopState)>, %36:
↪ !rvsdg.lambdaRef<(i32, !rvsdg.ioState, !rvsdg.memState,
↪ !rvsdg.loopState) -> (!rvsdg.ioState, !rvsdg.memState,
↪ !rvsdg.loopState)>):
      (%40: i32, %41: !rvsdg.ioState, %42: !rvsdg.memState,
↪ %43: !rvsdg.loopState, %44: !llvm.ptr<!llvm.array<4 x i8>>,
↪ %45: !rvsdg.lambdaRef<(!llvm.ptr<i8>, !jlm.varargList,
↪ !rvsdg.ioState, !rvsdg.memState, !rvsdg.loopState) -> (i32,
↪ !rvsdg.ioState, !rvsdg.memState, !rvsdg.loopState)>, %46:
↪ !rvsdg.lambdaRef<(i32, !rvsdg.ioState, !rvsdg.memState,
↪ !rvsdg.loopState) -> (!rvsdg.ioState, !rvsdg.memState,
↪ !rvsdg.loopState)>): {
        %47 = jlm.createVarargs (%40: i32) ->
↪ !jlm.varargList
        %48 = arith.constant 0: i32
        %49 = arith.constant 0: i64
        %50 = arith.constant 0: i64
        %51 = llvm.getelementptr %44[%50, %49]:
↪ (!llvm.ptr<!llvm.array<4 x i8>>, i64, i64) -> !llvm.ptr<i8>
        %52 = llvm.icmp "ugt" %40, %48: i32
        %53 = rvsdg.match(%52 : i1) [
          #rvsdg.matchRule<1 -> 1>,
          #rvsdg.matchRule<default -> 0>
        ] -> !rvsdg.ctrl<2>
```

```
            %54, %55, %56, %57 = rvsdg.applyNode %45 :
↪ !rvsdg.lambdaRef<(!llvm.ptr<i8>, !jlm.varargList,
↪ !rvsdg.ioState, !rvsdg.memState, !rvsdg.loopState) -> (i32,
↪ !rvsdg.ioState, !rvsdg.memState, !rvsdg.loopState)>(%51 :
↪ !llvm.ptr<i8>, %47 : !jlm.varargList, %41 : !rvsdg.ioState, %42
↪ : !rvsdg.memState, %43 : !rvsdg.loopState) -> i32,
↪ !rvsdg.ioState, !rvsdg.memState, !rvsdg.loopState
            %58, %59, %60 = rvsdg.gammaNode(%53:
↪ !rvsdg.ctrl<2>)(%40: i32, %46: !rvsdg.lambdaRef<(i32,
↪ !rvsdg.ioState, !rvsdg.memState, !rvsdg.loopState) ->
↪ (!rvsdg.ioState, !rvsdg.memState, !rvsdg.loopState)>, %55:
↪ !rvsdg.ioState, %56: !rvsdg.memState, %57: !rvsdg.loopState): [
                (%61: i32, %62: !rvsdg.lambdaRef<(i32,
↪ !rvsdg.ioState, !rvsdg.memState, !rvsdg.loopState) ->
↪ (!rvsdg.ioState, !rvsdg.memState, !rvsdg.loopState)>, %63:
↪ !rvsdg.ioState, %64: !rvsdg.memState, %65: !rvsdg.loopState): {
                    rvsdg.gammaResult(%63:!rvsdg.ioState,
↪ %64:!rvsdg.memState, %65:!rvsdg.loopState)
                },
                (%66: i32, %67: !rvsdg.lambdaRef<(i32,
↪ !rvsdg.ioState, !rvsdg.memState, !rvsdg.loopState) ->
↪ (!rvsdg.ioState, !rvsdg.memState, !rvsdg.loopState)>, %68:
↪ !rvsdg.ioState, %69: !rvsdg.memState, %70: !rvsdg.loopState): {
                    %71 = arith.constant 1: i32
                    %72 = llvm.sub %66, %71: i32
                    %73, %74, %75 = rvsdg.applyNode %67 :
↪ !rvsdg.lambdaRef<(i32, !rvsdg.ioState, !rvsdg.memState,
↪ !rvsdg.loopState) -> (!rvsdg.ioState, !rvsdg.memState,
↪ !rvsdg.loopState)>(%72 : i32, %68 : !rvsdg.ioState, %69 :
↪ !rvsdg.memState, %70 : !rvsdg.loopState) -> !rvsdg.ioState,
↪ !rvsdg.memState, !rvsdg.loopState
                    rvsdg.gammaResult(%73:!rvsdg.ioState,
↪ %74:!rvsdg.memState, %75:!rvsdg.loopState)
                }
            ]->!rvsdg.ioState, !rvsdg.memState,
↪ !rvsdg.loopState
            rvsdg.lambdaResult(%58:!rvsdg.ioState,
↪ %59:!rvsdg.memState, %60:!rvsdg.loopState)
        }
    rvsdg.phiResult(%39:!rvsdg.lambdaRef<(i32, !rvsdg.ioState,
↪ !rvsdg.memState, !rvsdg.loopState) -> (!rvsdg.ioState,
↪ !rvsdg.memState, !rvsdg.loopState)>)
}->!rvsdg.lambdaRef<(i32, !rvsdg.ioState, !rvsdg.memState,
↪ !rvsdg.loopState) -> (!rvsdg.ioState, !rvsdg.memState,
↪ !rvsdg.loopState)>
```

```
    %76 = rvsdg.lambdaNode <(!rvsdg.ioState, !rvsdg.memState,
↪   !rvsdg.loopState)->(i32, !rvsdg.ioState, !rvsdg.memState,
↪   !rvsdg.loopState)> (%35: !rvsdg.lambdaRef<(i32, !rvsdg.ioState,
↪   !rvsdg.memState, !rvsdg.loopState) -> (!rvsdg.ioState,
↪   !rvsdg.memState, !rvsdg.loopState)>):
        (%77: !rvsdg.ioState, %78: !rvsdg.memState, %79:
↪   !rvsdg.loopState, %80: !rvsdg.lambdaRef<(i32, !rvsdg.ioState,
↪   !rvsdg.memState, !rvsdg.loopState) -> (!rvsdg.ioState,
↪   !rvsdg.memState, !rvsdg.loopState)>): {
            %81 = arith.constant 0: i32
            %82 = arith.constant 10: i32
            %83, %84, %85 = rvsdg.applyNode %80 :
↪   !rvsdg.lambdaRef<(i32, !rvsdg.ioState, !rvsdg.memState,
↪   !rvsdg.loopState) -> (!rvsdg.ioState, !rvsdg.memState,
↪   !rvsdg.loopState)>(%82 : i32, %77 : !rvsdg.ioState, %78 :
↪   !rvsdg.memState, %79 : !rvsdg.loopState) -> !rvsdg.ioState,
↪   !rvsdg.memState, !rvsdg.loopState
            rvsdg.lambdaResult(%81:i32, %83:!rvsdg.ioState,
↪   %84:!rvsdg.memState, %85:!rvsdg.loopState)
        }
    rvsdg.omegaResult(%25:!llvm.ptr<!llvm.array<4 x i8>>,
↪   %35:!rvsdg.lambdaRef<(i32, !rvsdg.ioState, !rvsdg.memState,
↪   !rvsdg.loopState) -> (!rvsdg.ioState, !rvsdg.memState,
↪   !rvsdg.loopState)>, %1:!rvsdg.lambdaRef<(i32, !rvsdg.ioState,
↪   !rvsdg.memState, !rvsdg.loopState) -> (i32, !rvsdg.ioState,
↪   !rvsdg.memState, !rvsdg.loopState)>, %2:!rvsdg.lambdaRef<(i32,
↪   !rvsdg.ioState, !rvsdg.memState, !rvsdg.loopState) -> (i32,
↪   !rvsdg.ioState, !rvsdg.memState, !rvsdg.loopState)>,
↪   %76:!rvsdg.lambdaRef<(!rvsdg.ioState, !rvsdg.memState,
↪   !rvsdg.loopState) -> (i32, !rvsdg.ioState, !rvsdg.memState,
↪   !rvsdg.loopState)>)
}

```

NTNU
Norwegian University of
Science and Technology