University of Glasgow | School of Computing Science

Compilation as a Bigraphical Reactive System

Borislav Kratchanov (2401649K)

Friday 14th April, 2023

ABSTRACT

Compilers are getting increasingly complex software systems. Hence, we need better abstractions to deal with this complexity. This paper implements the graph stages of an optimising compiler as a bigraphical reactive system, using declarative reaction rules. The approach has an overall shorter implementation, however the compilation speed is 6 to 7 orders of magnitude slower, and it applies fewer optimisations, compared to traditional compilers. It concludes that more modelling features and improved simulation tools are needed to make this approach a viable abstraction for compilers.

1. INTRODUCTION

Compilation has been an integral part of the software development process since its invention, allowing for an invaluable layer of abstraction between a program's source code and its binaries. Software projects continue to grow in complexity and size, and processor architectures are becoming increasingly complex and diverse [19]. This is reflected by the active compiler research and development to accommodate these growing demands. Despite this, the abstractions used in the implementation of production compilers for imperative languages has not changed significantly in the past decade. This has led to an unsustainable increase in their complexity, requiring new abstractions to be developed.

A substantial part of this complexity comes from the need of the compiler to perform optimisations on the program being compiled [25]. To expose optimisation opportunities, compilers use a variety of intermediate representations, many of which use graph concepts like nodes and edges. This allows for a direct parallel to be drawn between optimisation application and graph transformations.

This project explores the practicality of using a graph transformation system as a layer of abstraction in compilation. Instead of representing the graph intermediate representation using low-level imperative constructs like structs and pointers, it calls for representing them as a declarative graph transformation system. This allows for transformations to be defined as declarative graph rewriting rules and then applied using a graph rewriting engine, rather than imperatively iterating and manipulating objects and pointers.

This approach puts a greater focus on the effect of the transformation, rather than its implementation. It also reduces the opportunity for introducing bugs in the compiler by writing the code for iterating and transforming the graph only once – inside the graph transformation engine. A nice addition is the ability to automatically draw the transformations in a graphical form, enabling easier and more precise communication. It also opens the possibility for verifying the correctness of the transformations. This leads to the main *hypothesis:* It is practical to express and optimise compiler graph intermediate representations using a general-purpose graph transformation system.

Research Contributions:

- Model the compiler intermediate representation RVSDG as a bigraph – a universal graph model – in Section 4.
- Transform between graph intermediate representations using bigraph rewrite rules in Section 4.2.
- Implement compiler optimisations as bigraph rewrite rules in section 5 and apply them using the bigraph simulation and analysis engine BigraphER to real-world instances in Section 6.3.
- Show the graph transformation implementation is often, but not always, shorter than an imperative implementation in Section 6.1.
- Show that the simulation speed of the graph transformation system makes the approach currently impractical in Section 6.2.

The paper continues with Preliminaries (Section 2). Then, the technologies used in the compiler and how they tie together are described in Section 3. The translation of textual LLVM IR to bigraph RVSDG IR is in Section 4, followed by the way optimisations are expressed in Section 5. In the end, the implementation is compared to existing ones in Section 6, followed by conclusions in Section 7, and the paper's connection with related work in Section 8.

2. PRELIMINARIES

First, this section gives a brief overview of compilers. Then, it introduces the relevant Intermediate Representations used by compilers and finishes with a brief introduction to bigraphs and bigraphical reactive systems.

2.1 Compilers

Compilation can be roughly split in two stages [10] – the front-end, focusing on parsing the source code into an intermediate representation – and the back-end, focusing on generating binaries from this intermediate representation. These two stages are further broken down in 5 sub-stages each, illustrated in Figure 1.

The project focuses on 3 of these sub-stages. The first is Context Handling, which connects different program constructs with relevant information. An example of this is connecting a goto statement with its label. The second stage is Intermediate Representation Generation, which translates the source program into an equivalent Intermediate Representation. The third one is Intermediate Representation

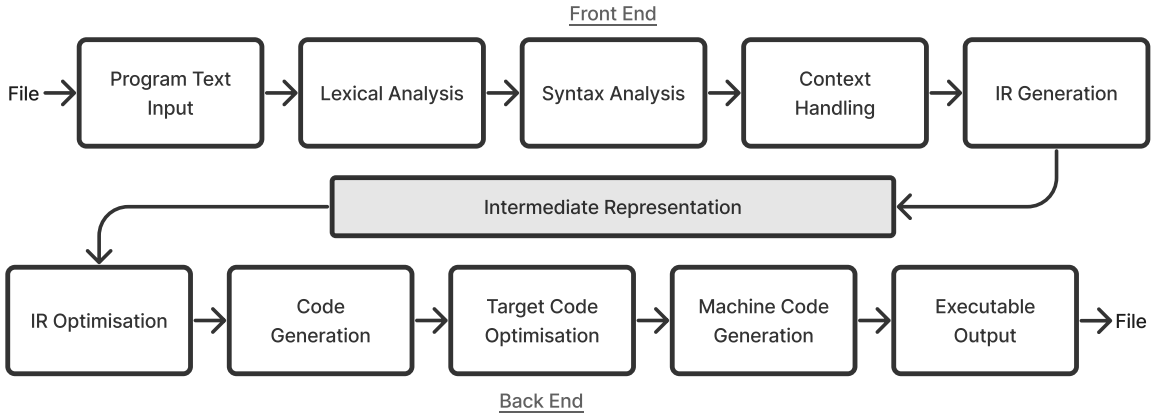


Figure 1: The pipeline of a modern compiler (adapted from [10]). The front-end reads in the program and makes sure its text follows the language rules. The back-end optimises the program and outputs its binary.

Optimisation, which changes parts of the intermediate representation to be more efficient – reducing the binary runtime, size, or both.

2.2 Intermediate Representations

As seen on Figure 1, an important focus for compilers is optimising the code being compiled. Hence, compilers translate the program to different Intermediate Representations (IRs) that makes applying optimisations easier.

IRs can be split in two broad categories – Linear and Graphical [25], which can often be mixed. Linear IR usually consists of an ordered list of instructions similar to assembly, which can have different level of abstraction. Linear IR instructions are strictly ordered to ensure correct execution of side-effects. Examples of Linear IR are Java Bytecode and Python Bytecode, as well as LLVM IR introduced in the next section. Many Linear IRs are in Static Single Assignment form (SSA) [9], which allows for a variable to only be assigned once. This makes tracking the value dependencies between instructions straight-forward.

Graphical IRs take the form of a graph, with nodes and edges defined differently depending on the specific IR. One of the first, and currently still most common, Graphical IR is the Control Flow Graph (CFG) [1]. A CFG is a directed graph, where the nodes (referred to as Blocks) represent a linear sequence of program instructions, usually using Linear IR. The edges in the CFG represent all of the possible execution paths between the different blocks. An example CFG is presented on Figure 2c.

Many optimisations rely on discovering loops within a program, however the CFG has no explicit way of denoting them. Hence, structural analysis is performed on the CFG to discover loops and branches [23]. This procedure is used to construct another Graphical IR, namely the Control Tree (CT), which specifies these constructs (Figure 2d).

Structural analysis cannot be performed directly on all CFG. CFG that are amenable to structural analysis are known as reducible. CFGs that contain irreducible control flow need to be first converted to a reducible form, for which many approaches have been described [11, 26, 4]. All of the approaches look for the Strongly Connected Components (SCC) in the CFG, which are subgraphs in which every node can be reached from every other node by following the directed edges (including reaching itself). Then, the entry and exit nodes are counted, and depending on their count, the SCC is transformed.

2.3 LLVM Framework and IR

LLVM[1] is an open-source compilation framework [14], used by compilers such as Clang[2]. Its modular design has enabled multiple front- and back-ends to be developed for it independently, supporting many languages and computer architectures respectively. Compilers using the LLVM framework have a similar architecture to the one described on Figure 1.

A central aspect of the framework is the LLVM IR[3], which has the form of a CFG, with SSA Linear IR as the body of each block. The Linear IR has the form of RISC assembly language (Figure 2b).

2.4 RVSDG

Another Graph IR is the Regionalised Value State Dependence Graph (RVSDG)[21]. It constrains the control flow of the program to be reducible, simplifying the application of optimisations. It also expresses instructions as graph Nodes, explicitly encoding the dependencies between them using edges. This removes the need to strictly order instructions that are independent, reducing the amount of time spent during compilation analysing their relations [21].

The RVSDG represents all computation using Nodes. Different types of Nodes represent different programming structures. They are split into two categories – Inter- and Intra-procedural. Inter-procedural Nodes represent, among others, constructs like global variables – Delta Nodes, and functions – Lambda Nodes. Intra-procedural Nodes represent instructions – Simple Nodes, branching – Gamma Nodes, and tail-controlled loops – Theta Nodes.

Each Node (apart from the root – the Omega Node) is contained within a Region (Figure 2e). Nodes in each Region depend only on computations performed in their parent Regions. This keeps the structure of the graph acyclic, simplifying optimisations. At the same time, each Node (apart from Simple Nodes) contains a Region, which specifies the computation it represents. In effect, the RVSDG's control flow structure is a tree Graph, with an Omega Node as the

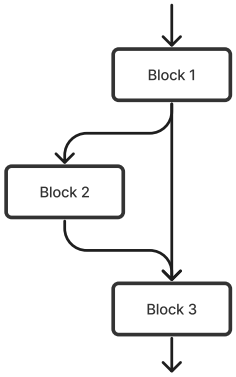
[1]LLVM is the name of the framework itself, and is no longer used as an acronym.
[2]`https://clang.llvm.org`
[3]`https://llvm.org/docs/LangRef.html`

```c
int main(){
    int a = 1;
    int b = 2;

    if (b>a){
        b = a;
    }
}
```

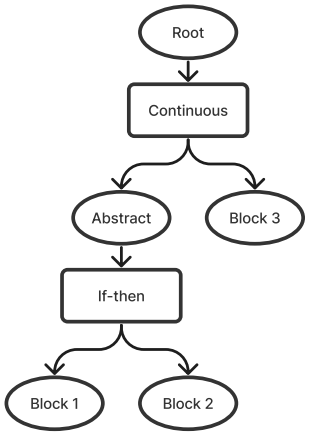
(a) C Source code

```
                ; Block 1
[...]
  %4 = load i32, i32* %3, align 4
  %5 = load i32, i32* %2, align 4
  %6 = icmp sgt i32 %4, %5
  br i1 %6, label %7, label %9

7:      ; Block 2
  %8 = load i32, i32* %2, align 4
  store i32 %8, i32* %3, align 4
  br label %9

9:      ; Block 3
  %10 = load i32, i32* %1, align 4
  ret i32 %10
}
```

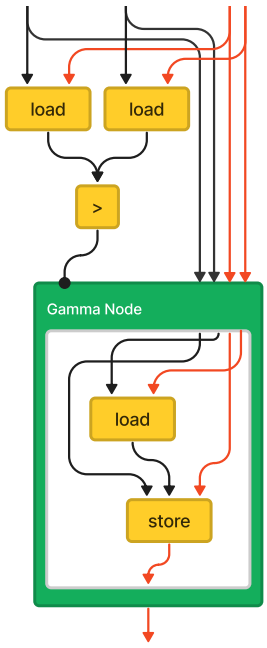
(b) LLVM Linear IR



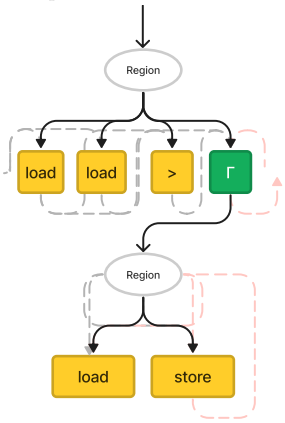
(c) Control Flow Graph



(d) Control Tree – Block 1 and Block 2 are matched to an if-then CT node from the CFG on 2c and are replaced by an abstract CFG node. The abstract CFG node and Block 3 are matched to a continuous CT node and the CT is complete.



(e) Part of RVSDG Region. Value dependencies are in black, state dependencies are in red. The yellow rectangles are Simple Nodes.



(f) RVSDG as tree
NB: The dependencies are shown with a dotted line to avoid overwhelming the graph.

Figure 2: The program in 2a represented as different IR. Note that some IR had to be abbreviated to fit.

root, and Simple Nodes as leaves (Figure 2f).

Whitin each Region, Nodes are connected to each other using Dependency Edges (referred to as Dedges in the rest of the paper), which represent the Value and State dependencies between the instructions. These dependency edges can connect to one or more Nodes, so in effect the RVSDG as a whole is a multigraph.

An RVSDG is built from an SSA CFG in two major stages:

1. Inter-procedural translation – identifies the functions and global variables in the program and translates them into RVSDG Nodes, assigning each of them a CFG representing the computation they perform.

2. Intra-procedural translation – takes each CFG assigned to an Inter-procedural Node and converts it into RVSDG Nodes and Regions, in 4 steps:

   (a) Control Flow Restructuring – converts irreducible control flow in the CFG to reducible control flow.

   (b) Structural Analysis – builds a CT, identifying the needed RVSDG Gamma and Theta Nodes and their respective Regions. A comprehensive description of this stage is in Chapter 4 of [24].

   (c) Demand Annotation – discovers the dependencies between the Nodes.

   (d) Control Tree Translation – emits the RVSDG Nodes.

2.5 Bigraphs

Bigraphs are a formalism representing the spatial and interaction relationships between entities described by Milner [18]. It has a set of nodes, which are referred to as controls, with two structures defined over them. Note that the term *control* is not related to the concepts of Control Flow Graph and Control Tree described earlier. In the rest of the work, the bigraph concept of a *control* will be used with a lowercase letter, whereas the IR concept of Control will be used capitalised, always as a modifier (i.e., *Control Tree*, or *Control Flow Graph*), and as part of an abbreviation whenever possible (i.e., *CT* or *CFG*). The term Node is not used in relation to bigraphs to avoid confusion with RVSDG Nodes.

Figure 3 has an example bigraph with all relevant features. The place graph of a bigraph models the topological relationship between the controls. It is a directed tree graph and it is usually visualised by containment, but this is equivalent to its tree visualisation (Appendix A). It has two powerful abstraction features. The grey squares are sites and represent any possible place sub-graph. The dotted rectangle around is a region and it represents any possible parent place graph. Note that bigraph regions are not related to the concept of RVSDG Regions described in the previous section. To avoid confusion, the word will be used to refer to RVSDG Regions by default and will be capitalised in this case. When referring to bigraph regions, that will always be specified.

The green edges in the bigraph represent the link graph. It is a multigraph modelling the interaction and non-spatial relationships between the controls. Outer names represent a connection to any possible link outside of the specified bigraph. Note that the figures in the paper break the convention of always writing the outer names on the top when visualising them to keep the figures less cluttered.

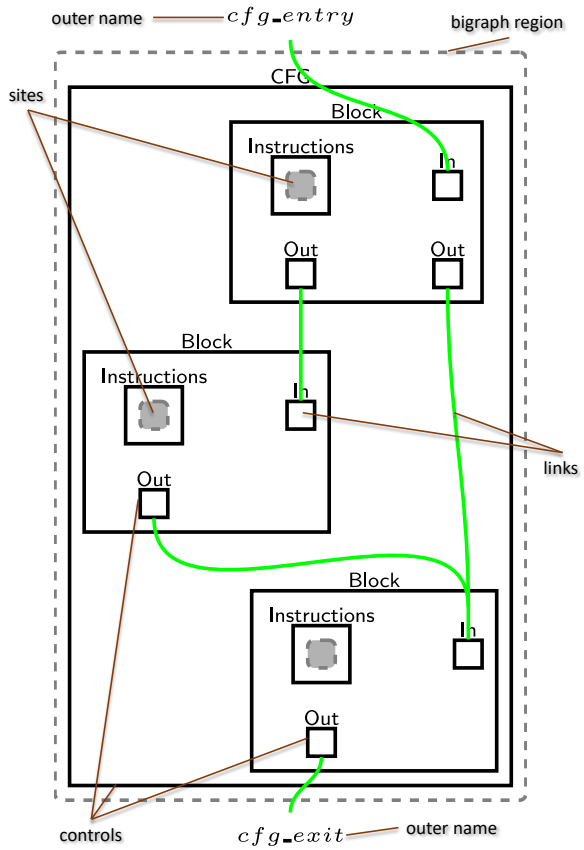


Figure 3: The control flow graph from Figure 2c expressed as a bigraph. A control contained within another control is its child in the place graph. It can also be equivalently visualised as a tree (Appendix A). The link graph is not directed, so direction is expressed using the controls In and Out.
Note that the outer names are expressed both on the top and the bottom, following the flow of execution.

2.6 Bigraphical Reactive System

Bigraphs are often used as part of a Bigraphical Reactive System (BRS) to model complex systems [5, 12, 27]. Computation can be performed using a BRS by defining an initial state – a bigraph, and a set of reaction rules. The reaction rules are checked against the initial state and they are applied to it, creating a new state. Then the rules are matched against the new state, and so on. If multiple rules match a state, the rule to be applied is chosen non-deterministically.

A reaction rule defines a transformation on a bigraph. Each reaction rule has a left-hand side and a right-hand side, each defined with a bigraph. The LHS bigraph needs to be matched to an equivalent subgraph in the bigraph corresponding to a state. If it is successfully matched, it is replaced with the bigraph in the right-hand side. Examples of reaction rules can be found on Figures 6, 7, 9.

Extensions to BRS have been defined. One extension used in the paper is reaction rule priorities [3], which allows for defining which rules should be applied before other rules. Another one used is reaction rule conditions [2], which define if a bigraph structure needs to be present (or to be not present) in the bigraph representing the state outside of the specified LHS of the reaction rule.

3. DESIGN

An optimising compiler expressing its graph IR and transformations using a BRS is implemented to explore the practicality of the approach. The compiler is positioned within the LLVM framework, reducing significantly the overhead of implementing a full front- and back-end from scratch.

The definition of bigraphs was inspired by the need to model systems that have a local and hyperlocal aspect [18]. Namely, these days processes are often able to communicate to each other using local channels, like shared memory, and hyper-local channels, like using the internet. This model can also be applied to program execution. It has a local aspect – the sequential incrementation of the program counter, and a hyper-local aspect – the ability to randomly access and jump to different places in memory. Hence, a Bigraphical Reactive System (BRS) is a natural fit for the domain.

Another important consideration for the compiler is the choice of graph Intermediate Representation (IR) to express the computation and apply optimisations to. The default choice of using a CFG would have been sufficient, however bigraphs allow for much more flexibility than what can be expressed using a CFG due to its linear component within blocks. The RVSDG was chosen due to its structure of sibling nodes within a Region connected by dependency edges, which can be naturally expressed as a bigraph.

3.1 Pipeline

The pipeline of the compiler is illustrated on Figure 4. The bigraph aspect of the compiler is implemented using the bigraph reaction and analysis engine BigraphER [22]. It is used because it has a concise declarative syntax for defining bigraphs, and reactions rules on the bigraphs. It also supports reaction rule priorities, conditions and parameterisation, which simplify the development process.

Before entering the bigraph domain, the compiler performs two transformations to the source code. The first one is converting it into LLVM Intermediate Representation. LLVM IR is in a SSA CFG form by definition, so this allows directly translating it to RVSDG using the steps described in Section 2.4. Also, LLVM is open source, so there are many available tools to parse and use it, and there are many front-ends and back-ends using LLVM. Positioning the bigraph transformation system within the LLVM framework allows it to be generalised for different languages and architectures with little to no extra effort.

Having the SSA CFG, the next step performed by the compiler is Inter-Procedural Translation. This stage is performed using a Python program that reads in the LLVM IR and outputs an Inter-Procedural RVSDG in BigraphER format, with CFG assigned to each function and global variable Region. Python was chosen due to its powerful string manipulation features, which allows it to output a textual representation of the bigraph, enabling debugging efforts. It also has support for parsing LLVM IR using the third-party library llvmlite[4]. As its focus is on outputting LLVM IR, it lacks many features for parsing LLVM IR compared to libraries for other platforms (like Rust or C++). This requires manually parsing many constructs, but this is a worthwhile trade-off due to the additional flexibility of using Python.

[4] `https://github.com/numba/llvmlite`

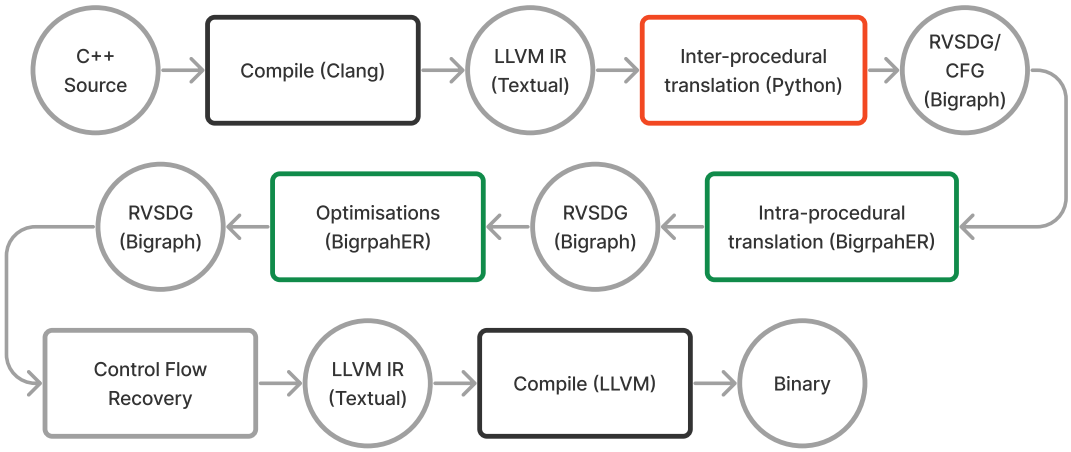


Figure 4: Pipeline of the implemented compiler. The inter- and intra-procedural translation stages is implemented to express the RVSDG IR using bigraphs. The Optimisation stage is implemented to explore the practicality of applying optimisations using graph rewriting rules.

The Inter-Procedural Translation leaves the body of the functions as a CFG, so the next step is to perform Intra-Procedural Translation to complete the conversion to RVSDG, by applying a set of bigraph reaction rules on the bigraph defined in the previous stage. This stage is confluent [1], so it is enough to only simulate one possible execution of the BRS and output the last state it reaches.

This state is then used as the initial state of the optimisation stage. The compiler optimisations are also implemented as a set of bigraph reaction rules, and it can be run in two different ways. The implementation can be accessed at `https://github.com/Bobby612/L5-Project`.

The final two stages would include converting the RVSDG bigraph back to LLVM and assembling it to binary. They were deprioritised, as they do not directly relate to the contributions of the paper.

4. MODELLING

This section describes how LLVM IR is translated into an RVSDG expressed as a bigraph. Inter-procedural Translation translates functions and global variables to bigraph RVSDG Nodes using a Python program. Intra-procedural Translation is split in two – the linear LLVM instructions are translated to RVSDG Simple Nodes within LLVM blocks using the Python program. Then, the LLVM CFG is translated to Gamma and Theta Nodes using bigraph rewrite rules.

4.1 Inter-procedural Translation

It focuses only on programs without mutual recursion. The extra steps for supporting it is described in other work [21], but it is relatively complex without being insightful.

RVSDG Nodes as Bigraphs

All RVSDG Nodes are modelled using the same format (visualised on Figure 5) to allow for generalisations to be made when appropriate. Each Node is represented by a bigraph control named Node. Its child level contains the different sets and attributes defining the Node. The first one is NodeType, whose child control defines the type of the

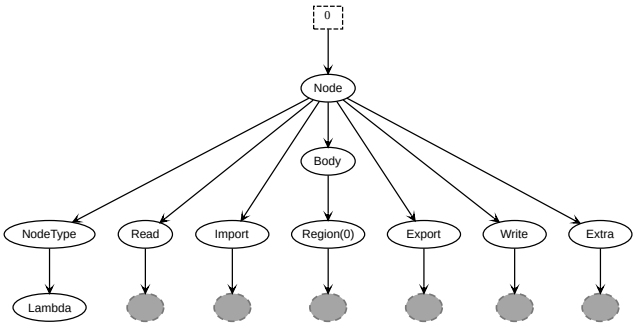


Figure 5: Place graph of an RVSDG Lambda Node expressed as a bigraph (in tree visualisation style). The structure is the same for Delta and Theta Nodes. Simple Nodes have no Regions and no Import/Export sets. Gamma Nodes can have multiple Regions, and each Region has exactly one Import set and one Export set corresponding to it.

RVSDG Node. Each Node also has a Read control and a Write control, which represent the sets of the Node's input and output dependencies.

The Body control represents the set of Regions that the Node has, if any. The Region control is parameterised to keep track of their order. The Node also has an Import and Export control for each Region. To make their identification with a Region easier, they also have a parameterised ID child control. Parameterising the Region control has required more parameterised reaction rules to be implemented than potentially necessary, which has not been an issue for the compiler so far, as it supports constructs with limited number of Regions. However, this will warrant a redesign if it is expanded to support compiling programs that require any number of Regions to be possible for Gamma Nodes.

The last child control of each Node is Extra, which is a set that contains information that is not defined in the RVSDG, but is important to keep track of to aid Control Flow reconstruction. Such information is data types and compiler options specific to the node.

The only Node that does not strictly follow this pattern is the Omega Node. It represents the program compiled as a whole, so it cannot depend on anything, and nothing depends on it, so it has no Read and Write set. Also, no optimisations should be applied to it, so to make that impossible, it is the only Node represented by its own control – Omega.

RVSDG Node features are summarised in Appendix B.

RVSDG Dependencies

Dependencies between Nodes are represented by Dedge controls (abbreviation for Dependency Edge), which have links between each other and are contained in the Read, Write, Import, and Export controls. For example, a function defines itself, so each Lambda Node has a Dedge in its Write set representing it. If a function calls another function in its body, it will have a Dedge in its Read set connected to the definition's Write Dedge by a link.

Conversely, the dependencies within a Node are tracked with a Loc control (for Location). This separate convention is done because dependency edges need to be kept local to a Region control. Local bigraphs have been proposed [17], but no support is available yet. Hence, the invariant is kept true by implementing extra rules. Continuing the above example, the function call inside the Region needs to express its dependency on the Lambda node of the function being called. Hence, the function call has a Dedge in its Read set that is connected to a Dedge in the Region's Import set. The Dedge in the Import set is then connected to the Dedge in the Read set of the Lambda Node using their child Loc control. Dependency edges that need to have a property defined within the Node (e.g., their type) are also connected to that property using the link to the Loc control.

4.2 Intra-procedural Translation

The next step of the pipeline is performing intra-procedural translation to represent the functions' body with RVSDG constructs by adapting the procedure described in section 2.4. The Control Flow Restructuring stage is described in other work [21], but is not performed because it is complex without contributing to the research aims. Hence, the compiler assumes the input CFG has no irreducible control flow.

Demand Annotation and Control Tree Translation were interleaved, with Demand Annotation being performed after each new RVSDG Node is created during Control Tree translation. This interleaved stage was split into two, with Simple Nodes being created before Structural Analysis, and Theta and Gamma Nodes being performed after Structural Analysis. The three stages are therefore:

1. Simple Node CTT and DA
2. Structural Analysis
3. Theta and Gamma Node CTT and DA

This adaptation allowed for the compiler to fully enter the bigraph domain at the second stage.

Simple Nodes

As bigraphs do not have an inherent order and support for variable names, the instructions of each block are translated to a partial RVSDG Region by the Python program.

This involved translating each instruction into a Simple Node expressed as a bigraph and connecting its Dedges. Simple Nodes follow the same structure of all other Nodes, but instead of Regions in their Body control, they have an Instruction control, parameterised on the LLVM's instruction opcode. Consequently, they also have no Import and Export controls. Following the same structure allows for reaction rules to be written that work for both Simple and non-Simple nodes.

Their Dedges are also organised in the same way, and are defined based on the dependencies each instruction has, as defined in LLVM. In a nutshell, each LLVM instruction outputs at most a single value to a virtual register, which corresponds to a Dedge in the Simple Node's Write set. Each LLVM instruction also depends on a set of values, so the corresponding Dedges are connected to the Dedges these are output from.

Some instructions also change and/or depend on the state of the program, so they are given extra Dedges connecting them to the rest of the stateful computations in the correct order. Such instructions are e.g. Load and Store – loading a value from an address can have a different effect whether it is done before or after a store. In effect, Value Dedges keep track of the values produced by instructions, and State Dedges keep track of their side-effects.

The state model of the compiler assumes each memory address to have its own state. This allows for independent state computations to be optimised separately. For this to work, an assumption the compiler makes is that no pointer operation is performed, such that a location is accessed without being ordered by its state. This could be violated, for example, if two separate variables point to the same address. Modifications to them would be ordered separately, which leads to a race condition. This could be mitigated in the future by analysing the code for any such cases and ordering them with additional State Dedges.

The State Dedges have the same bigraphical structure as the ones responsible for the value dependencies and are differentiated from them only by having a different type in the Extra set of the Node. This allows them to be manipulated by the same reaction rules for value edges when the distinction is immaterial.

In LLVM IR, constants are denoted instead of registers and labels within the instruction they are used, so an instruction using a constant would have fewer dependencies. To make all Simple Nodes with a specific opcode have the same number of Dedges, constants were also represented as Simple Nodes.

Creating Simple Nodes is part of the Control Tree Translation stage described in other works [24]. It is partially performed at the very beginning to enter the bigraph domain and explore the conversion between graph IR using bigraph rewrite rules. As LLVM instructions are represented either textually, or using bytecode, they need to be translated. RVSDG Simple Nodes were adapted to be one such translation, so they are created now, rather than at the end. The features of Simple Nodes are illustrated on Figure 8. The supported instructions are summarised in Appendix C.

CFG Blocks

Each LLVM CFG block is translated to a Block control that has 4 child controls. The first one is a body control, which contains the partial RVSDG Region described above. It also has an Interface control, which connects it to the other Blocks. Each Block has a single Entry point and zero

or more exit points, represented by corresponding controls.

The final two child controls in the Block are Import and Export, which have Dedges that connect to the Simple Nodes inside its body that have dependencies from outside of the block. This way the Nodes inside each block already follow the needed invariant that no Dedges are connected outside of the Region. The Dedges in the Import and Export set are also connected to the Import and Export Dedges of the function using their child Loc controls.

This tracking of dependencies partially performs the Demand Annotation stage of the Intra-Procedural translation. In other work, it is performed as a separate stage in the end of the compilation process, however here it is interleaved with the other stages. This is necessary because the CFG Blocks are combined in the Control Tree Translation stage, so having their dependencies tracked from the beginning allows them to be ordered correctly. This is further elaborated on in the following sections. Following the original order of the stages should be possible, however it would require introducing more concepts to the bigraph model, unnecessarily complicating it.

Structural Analysis

From this point on, the program is fully represented as a bigraph, so the transformation from CFG to CT and finally to RVSDG Nodes is performed as a BRS and constitute one of the paper's contributions.

The first rule initialises the CT inside a Lambda Region by adding a Ct control in it acting as its root. It has the lowest priority in this stage to make sure each structural tree is built separately, so its construction trace is clear if debugging is needed. It also has a condition, making sure the CT is not initialised multiple times, which is also useful for debugging.

Then, a rule matching a structure in the CFG is applied. There are 5 structures in total that need to be matched – continuous blocks, if-then-else constructs, if-then constructs, self-loops, and while loops. Switch statements are omitted, but their implementation should be similar.

As an example, focus on the rule matching if-then control flow (Figure 6) Its LHS matches two blocks, in addition to the initialised Ct. The first block has an Exit control connected to the second block's Entry control. The first block also has another Exit control, which is connected to the same Entry control the second block's Exit is connected (but that Entry itself is not matched, so it is represented by an outer name). In the RHS, these two blocks are combined into a single abstract block in the CFG. This abstract block is connected to a new CT Node representing the if-then control flow. The CT Node contains a control that denotes its type and its two children blocks. The rule is parameterised on the order of the two exit nodes in the interface of the if-then node, as the two orders are equivalent. The other 4 rules are structured in a similar way (note that the while-loop rule translates the CFG construct into a combination of self-loop and if-then CT Nodes, so it does not have its own CT Node).

When an abstract block is added to a new CT Node, the already existing CT Node connected to it is moved to its place by a fix rule, which has the highest priority.

This procedure terminates because the number of Control Flow edges is reduced at each step. This is relevant because the rule matching self-loops does not reduce the number of CFG blocks, however it still reduces the number of connections, as its resultant abstract block no longer has a self-edge. Once there is a single abstract CFG block left, it is removed by a finish rule, to allow for the next stage to start.

Translation of Theta and Gamma Nodes

The rest of the Translation phase and the rest of the Demand Annotation phase are interleaved. There is one Translation reaction rule for each CT Node defined and a set of fix reaction rules ensuring their Dedges are correctly connected (i.e., the Demand Annotation itself is still correct). These sets of rules can be divided in two based on function – reaction rules creating a new RVSDG Nodes, and reaction rules combining RVSDG Regions.

The first group of reaction rules takes an if-then, if-then-else, or self-loop CT Node and turns it into an RVSDG Region with the corresponding RVSDG Node in it. As an example, consider the if-then rule (Figure 7). In its LHS, it matches an if-then CT Node, and it creates a corresponding Gamma Node out of the CT Block parameterised with 1. It also uses that block's Import and Export sets as the Gamma Node's Import and Read, and Export and Write sets respectively.

After this, a set of 23 fix rules are applied to the new Node (not all are needed in every case). The first 4 ensure the new Node has the correct Read and Write Dedges. Then, the node needs to be connected to its soon-to-be parent Region (which in the case of the if-then construct is the CT Block parameterised with 0). There are 12 cases that need to be covered, however they overlap, so they were implemented with 5 rules (Appendix D.1). This is in effect performing Demand Annotation for this new Node. The next 13 rules ensure the newly created RVSDG Node is consistent – e.g., the Dedges in the Read and Import set are now connected directly because they were copied, rather than through their Loc child controls, so they are split and reconnected. The next two fix rules connect the conditional that controls the newly created Node. The last fix rule combines the new Node with its parent Region and removes the CT Node, moving a level up in the CT.

Gamma and Theta Nodes follow the same format as the one described for Lambda and Delta Nodes. The only addition is that they have an extra Dedge, which connects to the result of the expression controlling the Node. Namely, a Theta Node models a self-loop (AKA a do-while loop), so it has a conditional expression in its Region, which is evaluated to decide if the loop should be continued or broken. Hence, it has a Branch Dedge in the Export set of its Region. Conversely, a Gamma Node models an if-then, if-then-else, or a switch construct, so it has a conditional expression in its parent Region, which is connected to it via a Branch Dedge in its Read set. These features are summarised in Appendix B and a Gamma node with two Simple Nodes in one of its Regions is illustrated on Figure 8.

Combining Regions

The second group of reaction rules takes a continuous block CT Node and combines the two RVSDG Region segments it represents. Its first rule is only used as an initialisation, so each pair of continuous blocks are combined separately, enabling debugging. Then, fix rules are employed to reconnect their Imports and Exports. Each dependency can either be in the Import set of the Region (if it is used in-

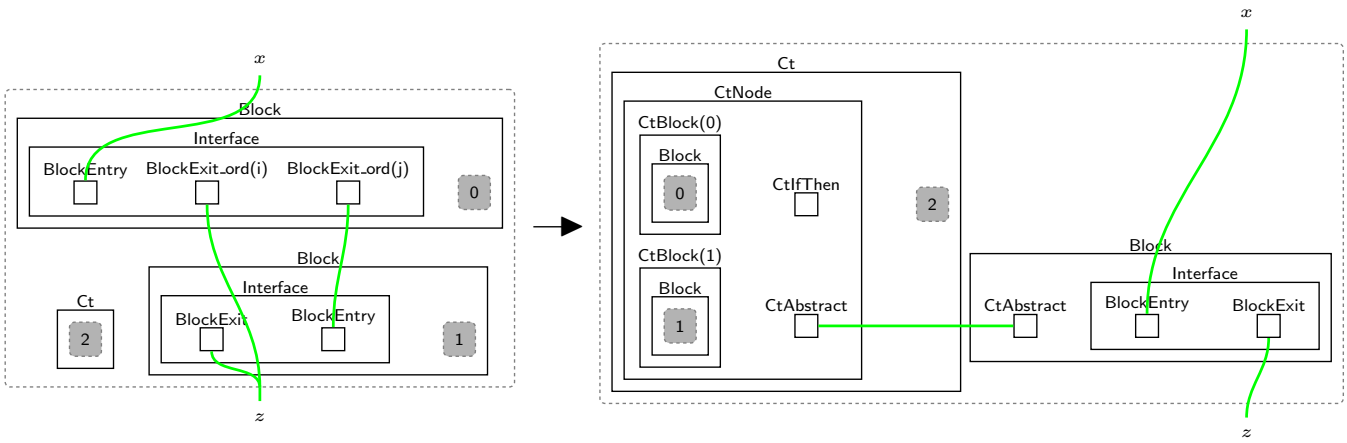


Figure 6: Rule matching if-then structure in CFG and turning it into a CT Node. The controls BlockEntry and BlockExit in this figure perform a similar function to In and Out in Figure 3. Here, BlockExit controls are further parameterised (i and j) to keep track where the control flows when the condition is true and where when it is false.
Also, compare RHS to 2d – the CtNode marked as CtIfThen node by a child control and the two child CtBlocks.
Note that z is an outer name despite being placed on the bottom, so it can follow the control flow. Sites are also numbered to keep track where each sub-graph is moved by the rule.

side), in its Export set (if it is created inside), or both (if it is modified inside – particularly true for state dependencies). This leaves us with 15 combinations that need to be addressed across the two Region segments. Those 15 combinations can be split into sub-tasks which overlap, so they were implemented using only 4 reaction rules. This split can be found in Appendix D.2. The final fix rule combines the correctly connected sub-Regions into a single (sub-)Region and moves a level up in the CT.

When there is a single level left in the CT, its imports and exports are connected to the ones of its parent Lambda Region by two rules, and the level is finally combined with it as well, completing the Intra-procedural Translation. The output of this stage is an RVSDG modelled as a bigraph in the BigraphER textual format, which can then be used as the initial state of the optimisation stage.

5. OPTIMISATIONS

The compiler implements 5 optimisations using bigraph rewrite rules. 1 of them generalises to all of its possible applications, whereas the other 4 are restricted to specific cases. They were chosen from the optimisations that the RVSDG compiler jlm[5] implements to aid their comparison in implementation complexity (section 6.1).

5.1 Node Pull-in

If the result of a computation is used only in one branch of execution, it can be moved to be part of this branch. This way it is executed only when the branch is executed, speeding up the execution of other branches. In an RVSDG, branching is represented by Gamma nodes, so if a Node's Write set is only ever used in one Region of a Gamma node, it can be moved in that Region.

This is implemented using one main parameterised rule and a set of helper rules. In the main rule, the Node is moved to the appropriate region of the Gamma node. Its Read set of dependency edges is also copied in both the Read set of the Gamma node and the Import set of that Region. This rule is visualised on Figure 9. The helper rules make sure to disconnect the copied Dedges from the Node.

A limitation of the implementation is that it works only on Nodes that have a single dependency edge in their Write set – i.e., they have a single result. In theory, the optimisation should work on Nodes that have any number of Dependency Edges in their Write set (like Gamma or Theta Nodes).

A way to do it would be to implement it using multiple rules in a Mark-and-Sweep fashion – mark all of the Dependency Edges that are used in the way required by the rule, and if all of them are marked apply the rule.

This was attempted for one of the optimisations, which exposed another limitation of bigraphs – the lack of conditions on specific sites [2], which is needed at the sweep stage. Namely, once all appropriate Dependency Edges are marked in the Write set of a Node, you need a way to apply the condition to the whole Write set (by representing it as a site), but not on any other sites present in the LHS. Currently, the condition is applied to all sites.

5.2 Node Push-out – Theta

If a computation performed inside a (do-while) loop depends only on values that are not modified inside the loop, the computation can be performed before the loop. This way it is performed only once, instead of at every loop iteration. In the context of the RVSDG, this would mean for a Node inside a Theta Node's Region to be moved if its Read set of dependency edges depend only on Edges that are not modified within the Theta Region, and therefore not Exported/Written by the Theta edge. For idempotent operations (like a Store instruction performed with the same value to the same location), this condition is relaxed only to their value dependence.

The optimisation is performed using three main rules covering three separate cases – Nodes that have one Dependency Edge in their Read set, Nodes that have two, and a rule that covers Store Simple nodes explicitly. The second of these is illustrated in Appendix E.1. Like the Node Pull-in

[5]https://github.com/phate/jlm

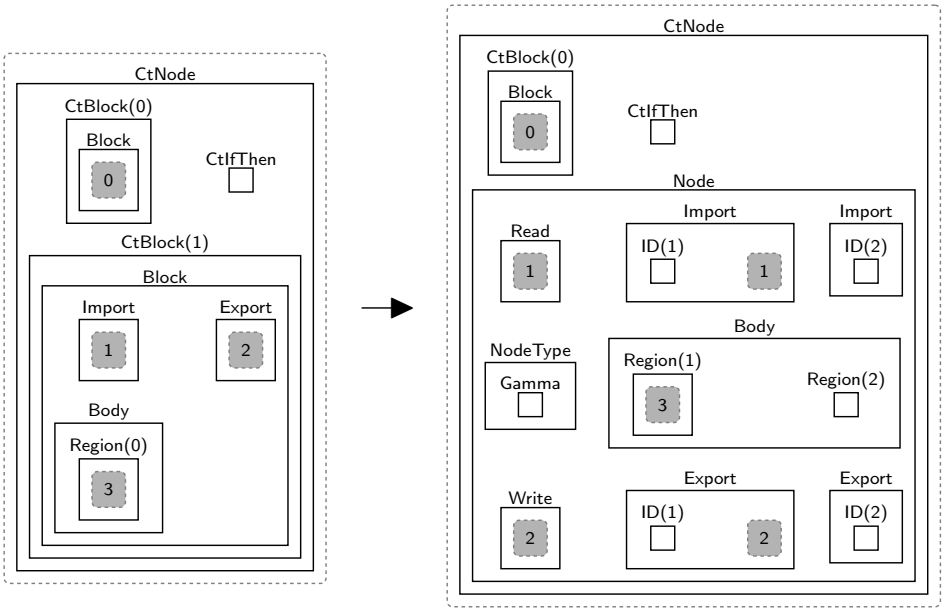


Figure 7: Rule matching If-then CT Node and turning it into a Gamma Node. Compare the similar structure of the CTNode on the LHS to the one on the RHS on Figure 6. Here, CtBlock 0 is matched with the same level of abstraction, but CtBlock 1 is further specified with its child cotrols. This allows their sites to be matched and copied in the RHS, where site 1 and site 2 are present multiple times. Any links connected to the controls corresponding to the copied controls will be linked after copying, so fix rules are needed to disconnect them.

optimisation, the implementation is limited in terms of the number of Dedges that the Node pushed out can have. The optimisation makes use of one helper rule, which removes any unneeded Dependency Edges from the Read set of the Theta Node, if present after pushing out the Node.

5.3 Node Push-out – Gamma

If the same computation is performed in all branches of execution, it can be performed before the branching. Hence, if the same computation is present in all Regions of a Gamma Node, it can be moved out of the Gamma node. The Node that would be pushed out needs to have all of its Dependency Edges in the Read set connected directly to the Region's Import set, and it needs to represent the same computation.

A consideration for this optimisation is the way equivalent computations are represented. In general, this optimisation should work for any computation, no matter how complex, which would mean that the body of the pushed-out Node should be equal. There is no concise BigraphER syntax that currently supports this feature, so the focus is on only matching Simple Nodes using a parameter.

Another consideration is whether the computation is commutative. In general, all Theta and Gamma Nodes have commutative dependencies, as the computation order is explicitly represented by the Nodes inside their Regions. This is not true for Simple Nodes, which fall in three categories:

1. Commutative – Simple Nodes representing commutative computations, e.g. addition or multiplication
2. Pseudo-commutative – Simple Nodes that have more than one dependency, but they are from different types, so they cannot be mixed up. e.g., the Simple node representing the Load instruction – it has one State and one Value dependency which cannot be swapped.
3. Non-commutative – representing computations that are not commutative, e.g. division.

Two main parameterised rules were implemented for this optimisation. One for commutative Simple Nodes with two dependencies (visualised in Appendix E.2), and one for non-commutative Simple Nodes with three dependencies. The former is parameterised only on the instruction type, whereas the latter is parameterised on the order of the Dedges as well.

5.4 Common Node Elimination

This optimisation removes duplicated computations. This is represented in the model as having two nodes representing the same computation in a region connected to the same dependencies. This optimisation has the same limitations as the Gamma Node Push-out optimisations, regarding both expressing the same computation and working on Nodes with different-sized Read and Write sets.

One rule is implemented for Simple Nodes of commutative instructions with two dependencies (Appendix E.3).

5.5 Invariant Value Redirection

After the application of different optimisations (e.g. Gamma Push-out), some Theta or Gamma nodes may be left with a dependency for an edge that is no longer needed. This RVSDG-specific optimisation removes that dependency to enable other optimisations in the parent region.

This optimisation was implemented with one main rule (visualised in Appendix E.4) and 5 helper rules.

6. EVALUATION

This section compares the described so far compiler in terms of implementation size and number of optimisations applied to gain insight in the practicality of the approach.

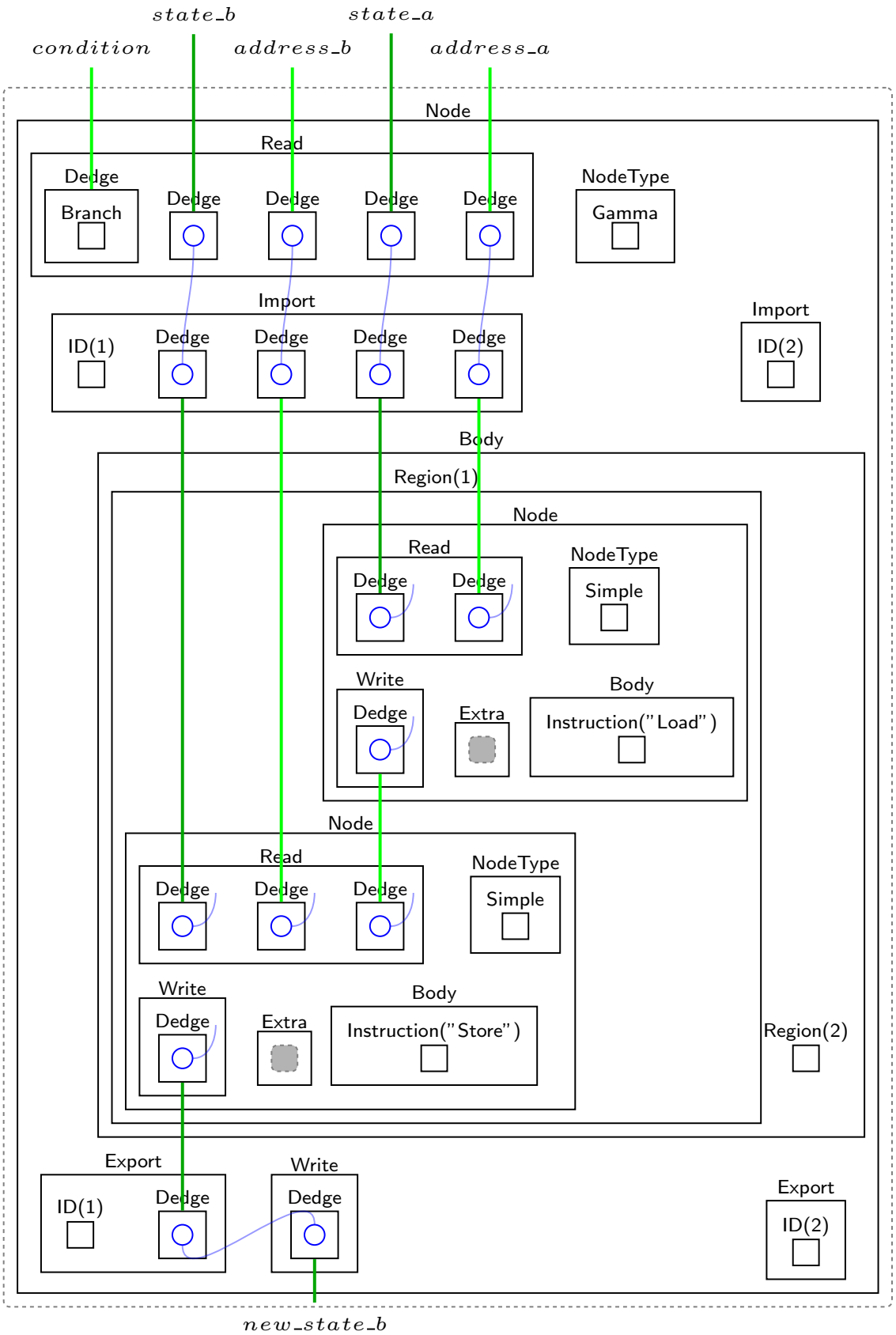


Figure 8: Example of a Gamma Node with two Simple Nodes in one of its Regions. Compare to the RVSDG Gamma Node on Figure 2e. The state dependencies are shaded darker to help them stand out, however they are modelled in the same way as the value dependencies. The blue circles are the Loc controls, which connect the Dedges within the nodes using the thinner blue links. In Figure 2e this is done by ordering them, which is not practical with bigraphs due to their lack of inherent order of sibling controls. There is also the branch Dedge in the Gamma's Read set, which connects the Gamma Node with the result of its condition (visualised as a dot in Figure 2e). *Note:* the links between the Loc controls inside the Simple Nodes are not drawn to avoid cluttering the diagram, however they would be connected to other controls within the Node.

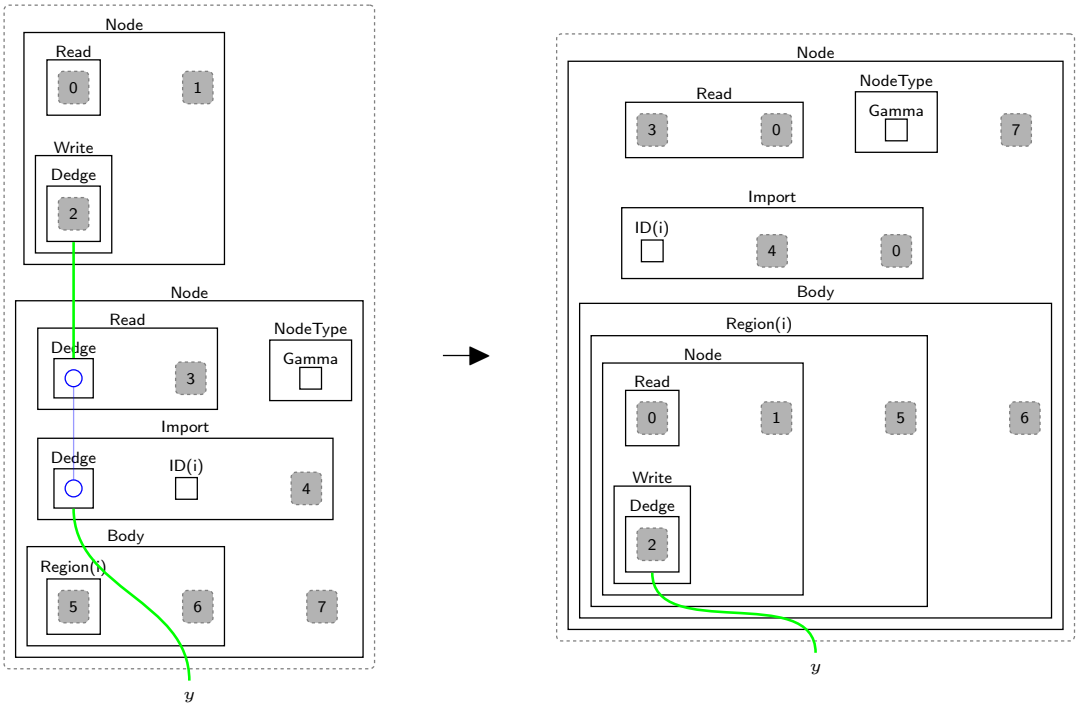


Figure 9: Node Pull-in main rule – it matches the Node on top on the LHS and moves it inside the Region of the Gamma Node on the bottom. This works because its only Write Dedge is connected only to Dedges inside that Region (enforced by the closed links), expressed by outer name y being connected to a Dedge in the Import set of that Region. The sites are numbered to keep track of which one goes where – notably, site 0 is copied in the RHS to the Import and the Read set of the Gamma Node. Any Dedges within those sites would be connected with all of their links after that, so fix rules are used to disconnect the unnecessary links.

6.1 Implementation Code Complexity

Comparing implementation complexity has been a topic of study [28] and debate [7] in the computing science community, with no reached consensus for best practices. What is more, the set of possible metrics is further limited by the declarative nature of BigraphER – measures like cyclomatic number [16] or control flow complexity [29] are not applicable when there is no concept of control flow in the language. When it comes to metrics relating to programming language concepts present in BigraphER, the different programming paradigm makes comparisons similarly uninsightful. For example, BigraphER has statements, however they can vary in size and complexity more than what is usual for imperative languages, so statement count is not a useful metric.

Therefore, the metric chosen was Lines of Code (LoC). Despite its limitations, it has been useful in software engineering for cost and effort prediction (with models like COCOMO [6]), so it should give us some insight into the implementation complexity. When used in these models, LoC has strict definitions depending on the programming language, however such definition does not exist for BigraphER, so each non-blank line is counted. To ensure this approach is not biased by formatting style, the number of keywords and identifiers are also counted.

The implementation was compared against the RVSDG compiler jlm[6] [21], written in C++. Only specific modules were compared, without including code shared between them for both implementations. This gives us insight into any strengths and weaknesses the two approaches might have and it controls for the fact that jlm has a lot more features implemented. For BigraphER optimisation rules that have variants working on different types of instructions or numbers of dependency edges, only the largest one was included, as it was assumed that better abstractions in the future would allow for all of cases to be included in a similarly sized rule. Also, not all optimisations had multiple variants implemented, so the size variation between the rules would be deceptive if all variants are counted. That being said, jlm is also likely to have implemented restrictions of the optimisations, as it applied none of the test instances (Section 6.3), so the comparison is not invalidated by this.

The results of the comparison can be seen on Figure 10, showing a much shorter implementation when using BigraphER for most, but not all modules. The modules that have a shorter C++ implementation are also smaller overall for both implementations, which alludes to reaching a point of diminishing returns for simple transformations.

This comparison shows there is potential for reducing the complexity of production compilers by introducing graph-based expressions, however more research is needed to make decisive recommendations.

[6] `https://github.com/phate/jlm`

6.2 Instances and Compilation Time

A commonly used [31, 21, 30] set of instances used for evaluating compiler performance and optimisations is Poly-

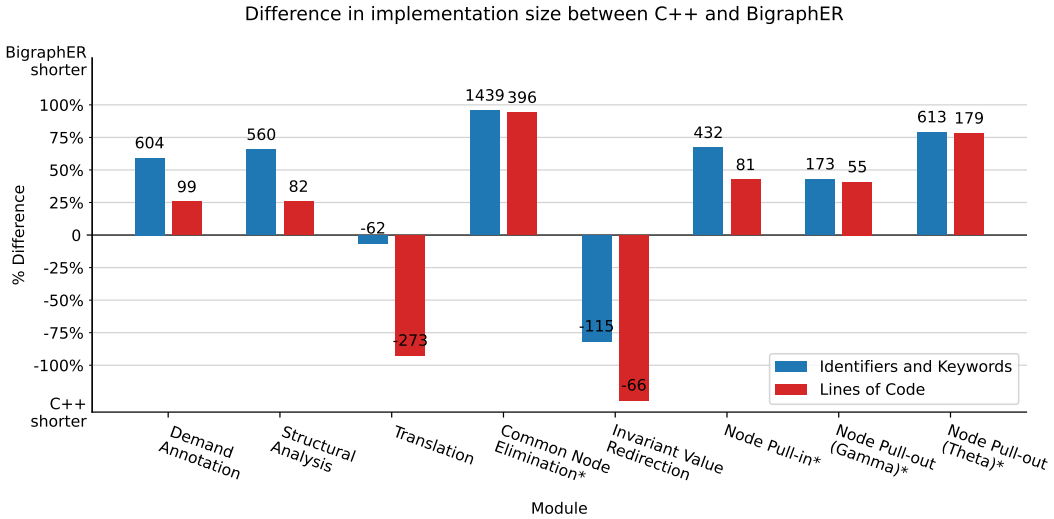


Figure 10: The % difference of *Lines of Code*, and *Number of Identifiers and Keywords* between the two implementations. The BigraphER implementation is shorter for all modules except for Translation and Invariant Value Redirection.
Note that optimisations with an asterisk do not generalise to all possible applications in their BigraphER implementation.

bench[7]. Unfortunately, even the smallest instances correspond to bigraph RVSDG models which are too big for the MiniCard solver used by BigraphER, crashing it from high memory use. Hence, the smallest 6 kernel functions (with a number of instructions ranging from 83 to 110) were extracted from the evaluation set and are optimised in isolation (`kernel_bicg`, `kernel_gemm`, `kernel_jacobi1d`, `kernel_syrk`, `kernel_trisolv`, `kernel_trmm`). They represent the core computation performed by each of its instances and are the only ones that could be compiled in under 12 hours.

Even then, the BRS instances took on average over 7 hours each to simulate. For context, they took under a tenth of a second each to compile using either jlm or llvm. With a slowdown of 6 to 7 orders of magnitude, it is clear that the approach is not feasible at the moment for any practical use, even without controlled runtime measurements.

To gain insight into the reason behind the performance gap, we need to compare the size of the bigraph model with the size of its representation in the C++ runtime. To do this, we can compare the number of smallest constructs needed to represent the instances. This means comparing the number of controls with the total number of fields in BigraphER and C++ respectively. This proved to be challenging for the LLVM framework. Runtime measurements do not provide the granularity needed, measuring only object allocations. Estimating them statically cannot provide the same level of accuracy, however a lower bound could be set as each instruction has at least 9 fields.

BigraphER provides these statistics directly, so the average number of controls needed per instruction to represent each instance as bigraph RVSDG was measured to be 31 (st. dev 1.6) for the evaluation instances.

This makes the LLVM implementation use at most 3.4 times fewer constructs. More accurate measurements and an investigation into these differences are left for future work. The focus of the comparison is to show if the slowdown of 6 orders of magnitude can be explained by an inefficient encoding. A three times larger program is not only reasonable, but available in the benchmark set (and quite small in practice), so the performance of the solver is likely to be the cause of the slow compilation times.

An observation was made that the rate the solver reaches new states increased with the number of states that were visited, so it is possible that the issue is in the solver matching all possible rule application sites. As more optimisations are applied, fewer rules can be matched.

[7] `https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/`

6.3 Optimisations Applied

This section aims to check if the implemented compiler discovers and applies a similar number of optimisation opportunities to existing compilers. Ideally, it would have been compared to jlm, as it has equivalent optimisations implemented and it also uses the RVSDG IR. Unfortunately, jlm was not applying any optimisations on the chosen instances, despite them being also used for its evaluation in its introductory paper [21]. That may be due to their reduced size coupled with jlm's relatively stricter state model.

Hence, the LLVM opt module was used instead. 3 of its optimisations were identified to be similar to the ones implemented. Loop Invariant Code Motion (-licm option) moves instructions outside of loops similar to the Theta Node Push-out, Code Sinking (-sink option) moves instructions to further blocks in the CFG to avoid unnecessary execution similar to Node Pull-in, and Combine Redundant Instructions (-instcombine option) performs, among others, common node elimination. The number of optimisation applications for both compilers are summarised on Figure 11.

On average, LLVM performed over twice as many Node Push-outs as the implemented compiler. This was likely because the implemented compiler is unable to push out the same Nodes from nested while-loops. Namely, a while-loop is represented as a Theta Node inside a Gamma Node, so the Theta Node Push-out can only push the Nodes to the Gamma Region, rather than its parent Region. The implementation, however, performs Node Pull-in optimisations even though LLVM performs none. This is likely due to the

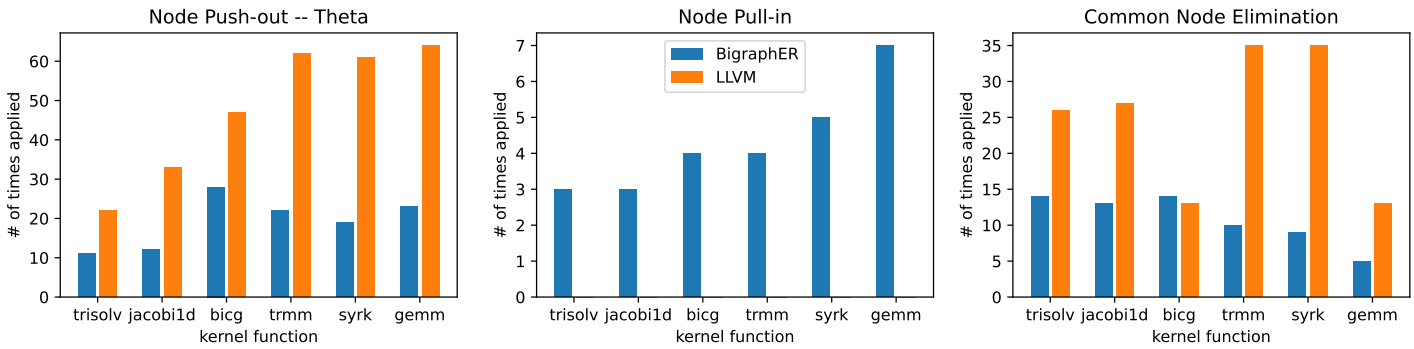


Figure 11: Comparison between the number of Nodes/instructions each optimisation was applied to by the implemented compiler and LLVM respectively.

extra Gamma Nodes present from its while-loop representation. Opt's Combine Redundant Instructions performs much better on average than Common Node Elimination, which is partially because of the extra instructions pushed to the same Block, and partially because of its broader scope.

7. CONCLUSIONS AND FUTURE WORK

This paper set out to explore the practicality of expressing the graph-based stages of a compiler as a graph transformation system. To achieve this, it models the IR RVSDG as a bigraph and it optimises it using bigraph rewrite rules. The facets of practicality explored were the size of the implementation and its compilation performance.

Most of the implemented optimisations are constrained to work on a subset of their domain because of unsupported by bigraphs abstractions and generalisations. This leads to a worse optimisation performance compared to a production compiler. To bring the approach to parity, bigraph features like local links [17] and conditions on links [2] would need to be defined and implemented in the future.

The implementation is on average shorter than the equivalent C++ implementation, which suggests it may also be less complex. Considering the deficiencies of the static measures of code complexity, a user study comparing the two implementations involving learners and educators would be needed to make a definitive judgement.

The compilation time was impractical for any real-world sized instances due to the performance of the rewrite engine. To take advantage of declarative graph transformations for compilers, significant improvements need to be made to the solvers used for matching rewrite rules to bigraphs. Better solvers would also allow for more complex analysis to be performed, like the optimal optimisation order for an instance could be found by making a full exploration of the BRS.

The correctness of rewriting rules can also be proven by showing that each rule does not change the semantics of the graph being transformed. This would eliminate the potential for whole classes of bugs and significantly reduce the trusted code-base.

8. RELATED WORK

Declarative rewrite rules are already used for applying optimisations in compilers for functional programming languages [20, 8]. Applying them to compilers for imperative languages has proven more challenging due to the need to take side-effects into account. One way to tackle the problem is to apply the rewrites on CFG IR and apply it only to computations with specific side-effects and dependencies [13]. This is expressed using conditional guards, restricting what features of the IR are present before and after it. Other work in the field has applied guarded rewrite rules on C-like IR [15], with even more complex conditions, like performing analysis on pointer operations.

Expressing optimisations as rewrite rules has also allowed for the optimisation implementation to be proven for correctness automatically in [15], by showing each rewrite rule does not change the semantics of the program. This significantly decreases the possibility for compilation bugs because it reduces the trusted codebase in the compiler to just the prover, even if new optimisations are introduced.

Acknowledgements. I would like to thank my supervisor, Dr Blair Archibald, for introducing me to bigraphs and inspiring me throughout my project. I would also like to thank my family for their unwavering support throughout my higher education and beyond.

9. REFERENCES

[1] F. E. Allen. Control flow analysis. In R. S. Northcote, editor, *Proceedings of a Symposium on Compiler Optimization, Urbana-Champaign, Illinois, USA, July 27-28, 1970*, pages 1–19. ACM, 1970.

[2] B. Archibald, M. Calder, and M. Sevegnani. Conditional Bigraphs. In F. Gadducci and T. Kehrer, editors, *Graph Transformation*, volume 12150, pages 3–19. Springer International Publishing, Cham, 2020. Series Title: Lecture Notes in Computer Science.

[3] J. C. M. Baeten, J. A. Bergstra, J. W. Klop, and W. P. Weijland. Term-Rewriting Systems with Rule Priorities. *Theor. Comput. Sci.*, 67(2&3):283–301, 1989.

[4] H. Bahmann, N. Reissmann, M. Jahre, and J. C. Meyer. Perfect Reconstructability of Control Flow from Demand Dependence Graphs. *ACM Trans. Archit. Code Optim.*, 11(4):66:1–66:25, 2014.

[5] S. Benford, M. Calder, T. Rodden, and M. Sevegnani. On Lions, Impala, and Bigraphs: Modelling Interactions in Physical/Virtual Spaces. *ACM Trans. Comput. Hum. Interact.*, 23(2):9:1–9:56, 2016.

[6] B. W. Boehm, Clark, Horowitz, Brown, Reifer, Chulani, R. Madachy, and B. Steece. *Software Cost*

Estimation with Cocomo II with Cdrom. Prentice Hall PTR, USA, 1st edition, 2000.

[7] J. Cherniavsky and C. Smith. On Weyuker's axioms for software complexity measures. *IEEE Transactions on Software Engineering*, 17(6):636–638, June 1991.

[8] A. Chlipala. An Optimizing Compiler for a Purely Functional Web-Application Language. *SIGPLAN Not.*, 50(9):10–21, Aug. 2015. Place: New York, NY, USA Publisher: Association for Computing Machinery.

[9] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, Oct. 1991. Place: New York, NY, USA Publisher: Association for Computing Machinery.

[10] D. Grune, K. van Reeuwijk, H. E. Bal, C. J. H. Jacobs, and K. Langendoen. *Modern Compiler Design*. Springer New York, New York, 2 edition, July 2012.

[11] J. Janssen and H. Corporaal. Making Graphs Reducible with Controlled Node Splitting. *ACM Trans. Program. Lang. Syst.*, 19(6):1031–1052, 1997.

[12] J. Krivine, R. Milner, and A. Troina. Stochastic Bigraphs. In A. Bauer and M. W. Mislove, editors, *Proceedings of the 24th Conference on the Mathematical Foundations of Programming Semantics, MFPS 2008, Philadelphia, PA, USA, May 22-25, 2008*, volume 218 of *Electronic Notes in Theoretical Computer Science*, pages 73–96. Elsevier, 2008.

[13] D. Lacey and O. de Moor. Imperative Program Transformation by Rewriting. In R. Wilhelm, editor, *Compiler Construction*, Lecture Notes in Computer Science, pages 52–68, Berlin, Heidelberg, 2001. Springer.

[14] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *CGO*, pages 75–88, San Jose, CA, USA, Mar. 2004.

[15] S. Lerner, T. D. Millstein, and C. Chambers. Automatically proving the correctness of compiler optimizations. In R. Cytron and R. Gupta, editors, *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003*, pages 220–231. ACM, 2003.

[16] T. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec. 1976.

[17] R. Milner. Local Bigraphs and Confluence: Two Conjectures. *Electronic Notes in Theoretical Computer Science*, 175(3):65–73, June 2007.

[18] R. Milner. *The Space and Motion of Communicating Agents*. Cambridge University Press, 2009.

[19] S. Mittal. A Survey of Techniques for Architecting and Managing Asymmetric Multicore Processors. *ACM Comput. Surv.*, 48(3):45:1–45:38, 2016.

[20] S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *2001 Haskell Workshop*. ACM SIGPLAN, Sept. 2001. Edition: 2001 Haskell Workshop.

[21] N. Reissmann, J. C. Meyer, H. Bahmann, and M. Själander. RVSDG: An Intermediate Representation for Optimizing Compilers. *ACM Transactions on Embedded Computing Systems*, 19(6):49:1–49:28, Dec. 2020.

[22] M. Sevegnani and M. Calder. BigraphER: Rewriting and Analysis Engine for Bigraphs. In S. Chaudhuri and A. Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, volume 9780 of *Lecture Notes in Computer Science*, pages 494–501. Springer, 2016.

[23] M. Sharir. Structural Analysis: A New Approch to Flow Analysis in Optimizing Compilers. *Comput. Lang.*, 5(3):141–153, 1980.

[24] J. Stanier. *Removing and restoring control flow with the Value State Dependence Graph*. PhD Thesis, University of Sussex, UK, 2012.

[25] J. Stanier and D. Watson. Intermediate representations in imperative compilers: A survey. *ACM Comput. Surv.*, 45(3):26:1–26:27, 2013.

[26] S. Unger and F. Mueller. Handling Irreducible Loops: Optimized Node Splitting vs. DJ-Graphs. In R. Sakellariou, J. A. Keane, J. R. Gurd, and L. Freeman, editors, *Euro-Par 2001: Parallel Processing, 7th International Euro-Par Conference Manchester, UK August 28-31, 2001, Proceedings*, volume 2150 of *Lecture Notes in Computer Science*, pages 207–220. Springer, 2001.

[27] L. Walton and M. F. Worboys. A Qualitative Bigraph Model for Indoor Space. In N. Xiao, M.-P. Kwan, M. F. Goodchild, and S. Shekhar, editors, *Geographic Information Science - 7th International Conference, GIScience 2012, Columbus, OH, USA, September 18-21, 2012. Proceedings*, volume 7478 of *Lecture Notes in Computer Science*, pages 226–240. Springer, 2012.

[28] E. Weyuker. Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, 14(9):1357–1365, Sept. 1988. Conference Name: IEEE Transactions on Software Engineering.

[29] M. R. Woodward, M. A. Hennell, and D. Hedley. A Measure of Control Flow Complexity in Program Text. *IEEE Trans. Software Eng.*, 5(1):45–50, 1979.

[30] X. Wu, M. Kruse, P. Balaprakash, H. Finkel, P. D. Hovland, V. Taylor, and M. W. Hall. Autotuning PolyBench benchmarks with LLVM Clang/Polly loop optimization pragmas using Bayesian optimization. *Concurr. Comput. Pract. Exp.*, 34(20), 2022.

[31] T. C. d. S. Xavier and A. F. d. Silva. Exploration of Compiler Optimization Sequences Using a Hybrid Approach. *Comput. Informatics*, 37(1):165–185, 2018.

APPENDIX

A. BIGRAPH EXAMPLE VISUALISED AS TREE

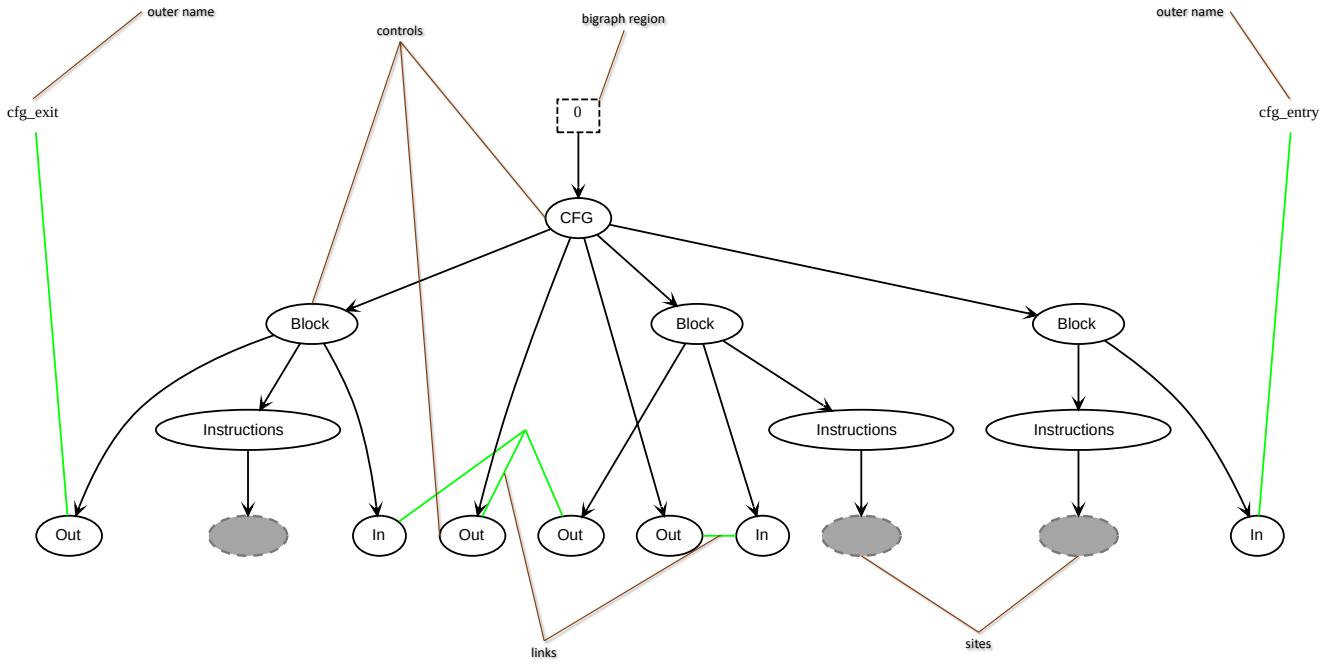


Figure 12: The CFG bigraph example from Figure 3 visualised as a tree.

B. IMPLEMENTED RVSDG NODES

Node	Construct	Read	Write	Regions	Import	Export	Extra
Lambda (defined)	Function (defined in the program)	Edges to Lambda and Delta nodes it depends on	The function itself	One	Its dependencies; the function's arguments	Return value dependency; state Dedges	Function Type, Export dependency edge Type
Lambda (imported)	Function (defined outside of the program)	Import Dedge in Omega node	The function itself	One (Empty)	Empty	Empty	Function Type
Delta	Global Variable	Edges to Global Variables it depends on (if any)	The variable itself	One	Global Variables it depends on (if any)	The variable (statically) defined in the region	Variable Type
Omega	The Program itself	N/A	N/A	One	External dependencies	The Main function	Data Types of External Dependencies
Theta	Self-loop (AKA do-while loop)	Value and State that Nodes in its Region depend on	Value and State Written in its Region	One	Value and State that Nodes in the Region depend on	Value and State written in the Region; *The loop condition*	None
Gamma	If-then, if-then-else, or switch* statement	Value and State that Nodes in all of its Regions depend on; *The branching condition*	Value and State Written in all of its Regions	Two or more (last one is always default/else)	Value and State that Nodes in the specific Region depend on	Value and State Written in each specific Region	None
Simple	Instruction, Constant	Value and State it depends on	Value and State it writes	N/A	N/A	N/A	Dependency types and other instruction options

Table 1: Summary of the implemented RVSDG Nodes and their features

C. IMPLEMENTED LLVM INSTRUCTIONS

The following table summarises the supported instructions by the compiler. They were implemented on an as-needed basis.

Instruction Format	**Implemented Opcodes**	**Read (value)**	**Read (state)**	**Write (value)**	**Write (state)**	**Extra**
Function call	call	Function arguments	Method order	Function return value (if any)	New method order	Types
Extend value to a type	sext	Value to extend	—	Extended value	—	Types
Get element pointer	getelementptr	Address and offset	—	Pointer	—	Types and Options
Allocate memory	alloca		—	Address of allocated memory	State of the allocated memory	Type
Conversion instructions	bitcast; sitofp	Value to convert	—	Converted value	—	Type
Return	ret	Value to return	—	The return export of the function region	—	Type
Compare	icmp	Values to compare	—	The comparison result	—	Type, comparison operation
Conditional break	br	Condition	—	—	—	—
Load	load	Address	Address state	Loaded value	—	Type
Store	store	Address, value to store	Address state	—	New address state	Type
Arithmetic	add; sub; mul; shl; srem; urem; sdiv; fadd; fsub; fmul; fdiv; frem	The values to be calculated	—	Calculated value	—	Type and options

Table 2: A summary of the LLVM instructions supported by the developed compiler.
NB: The instruction format groups in the table is based on observation, rather than on any official grouping.

D. COMBINING DEPENDENCIES

D.1 Adding New Node

When a new RVSDG Node needs to be added to its parent region, there are 12 cases that need to be considered. Note that the Node is added at the ”end” of the Region – if any of its dependencies are changed in the Region, it should follow them. Each of the 12 cases can be split into parts, which are highlighted on Table 3. As the parts repeat themselves, all cases can be addressed using 5 reaction rules in 4 priority classes.

Region \Node	Read	Write	Read and Write
Import	Connect Read to Import	Add Write to Export	Connect Read to Import and Add Write to Export
Export	Connect Read to Export	Connect Write to Export	Connect Read to Export and Connect Write to Export
Import and Export	Connect Read to Export	Connect Write to Export	Connect Read to Export and Connect Write to Export
Neither	Add Read to Import	Add Write to Export	Add Read to Import and Add Write to Export

Table 3

In summary, the 5 rules implemented are (in order of priority):

1. Connect Read to Export – Connect the Read Dedge to Export Dedge
2. Connect Write to Export – Connect the Export Dedge to the Write Dedge
3. Connect Read to Import – Connect the Read Dedge to the Import Dedge
4. Add Write to Export and Add Read to Import – Add Dedge to Import/Export and connect it to the respective Dedge in the Node.

D.2 Combining Region Segments

When two continuous blocks need to be combined into a single block, the subregions need to be combined. This requires the dependencies of each node to be connected correctly. There are 15 cases for each Dependency, based on whether it is present in the Import, Export, or both of each Region, which are summarised in Table 4. When split into steps, these 15 cases can be implemented using 4 standalone rewrite rules, and 4 tasks implemented as part of other rules.

Block 2 \Block 1	Import 1	Export 1	Import and Export 1	None 1
Import 2	Connect Import 2 to Import 1, Remove Import 2	Connect Export 1 to Import 2, Remove Import 2; Copy Export 1	Connect Export 1 to Import 2, Remove Import 2; Copy Export 1 and Import 1	Copy Import 2
Export 2	Copy Import 1 and Export 2	Remove Export 1, Copy Export 2	Remove Export 1, Copy Import 1, Copy Export 2	Copy Export 2
Import and Export 2	Connect Import 2 to Import 1, Remove Import 2; Copy Export 2	Connect Export 1 to Import 2, Remove Export 1 and Import 2; Copy Export 2,	Connect Export 1 to Import 2, Remove Export 1 and Import 2; Copy Import 1 and Export 2	Copy Import 1 and Export 2
None 2	Copy Import 1	Copy Export 1	Copy Import 1 and Export 1	—

Table 4

As some tasks are represented by rules that can match other tasks, it was important to define their priorities as well:

1. Connect Export 1 to Import 2, Remove Export 1 and Import 2 – matches if a dependency is in all of Export 1, Import 2, and Export 2
2. Connect Export 1 to Import 2, Remove Import 2 – matches if a dependency is in Export 1 and Import 2
3. Connect Import 2 to Import 1, Remove Import 2 – matches if a dependency is in Import 1 and Import 2
4. Remove Export 1 – matches if the dependency is present in Export 1 and Export 2
5. Copy Import 1, Import 2, Export 1, Export 2 – Implemented as part of the continuous block finish reaction rule

E. OPTIMISATION REACTION RULES

E.1 Node Push-out – Theta

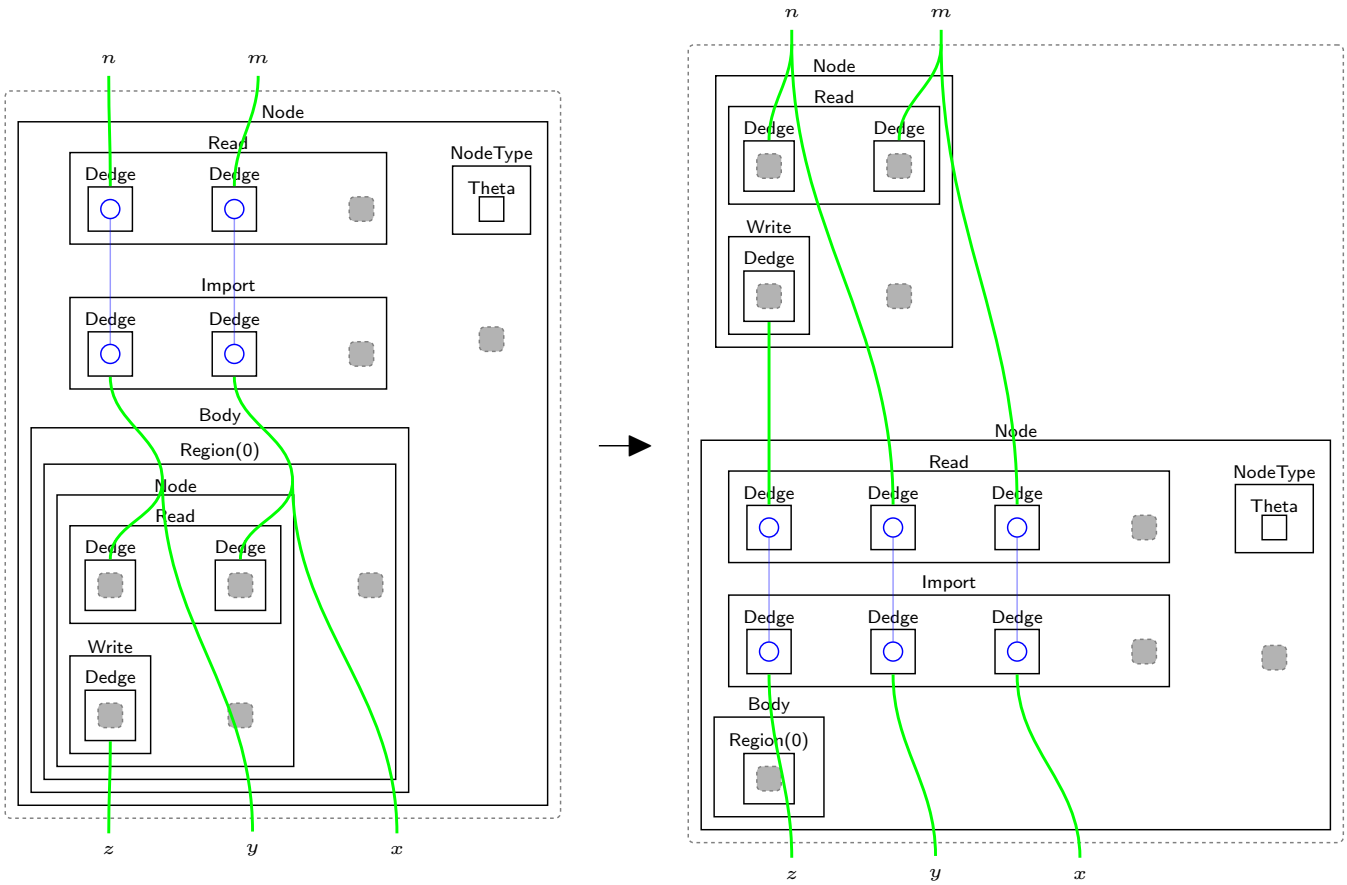


Figure 13: Node Push-out for Theta Nodes – initial rule

E.2 Node Push-out – Gamma

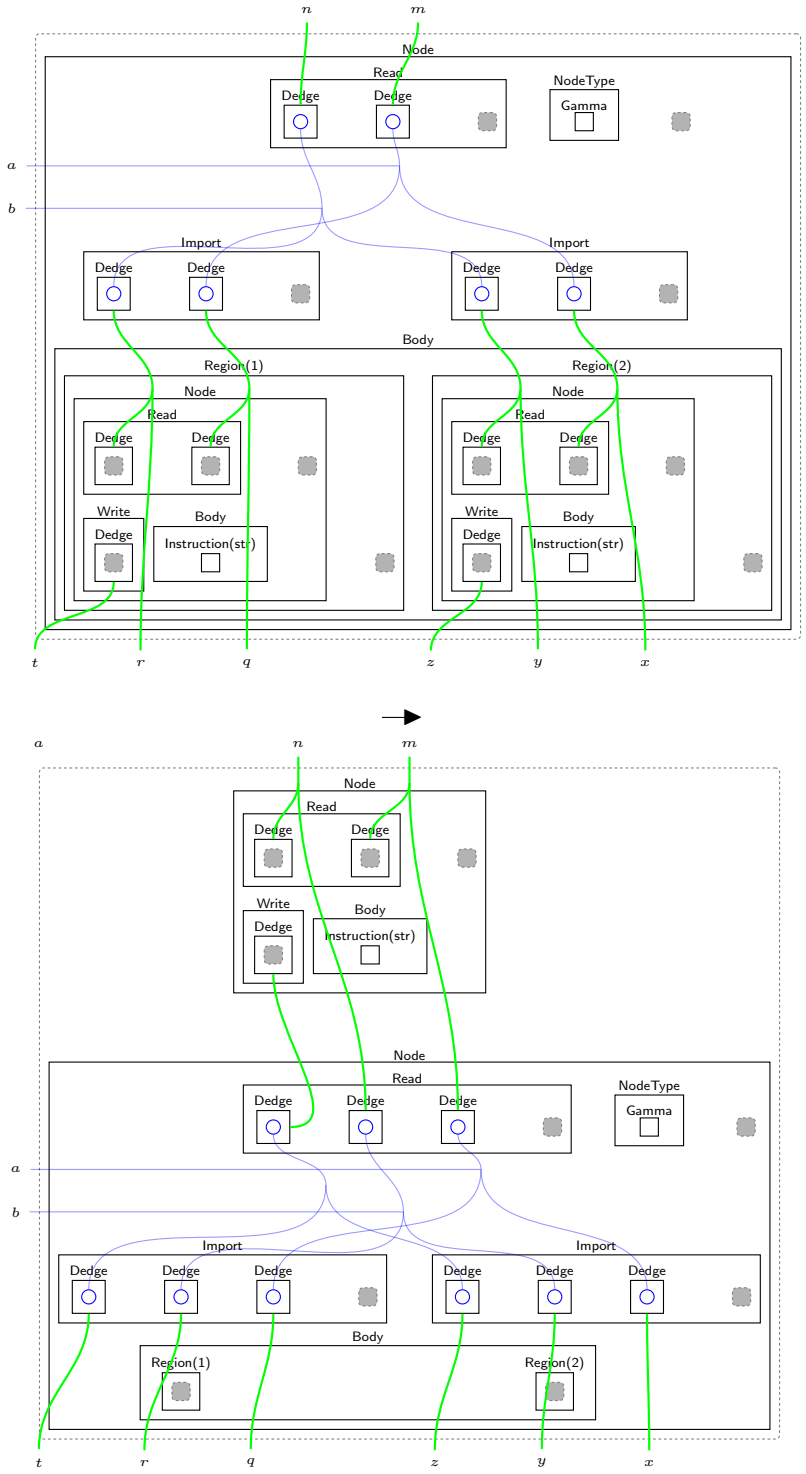


Figure 14: Node Push-out for Gamma Nodes – initial rule

E.3 Common Node Elimination

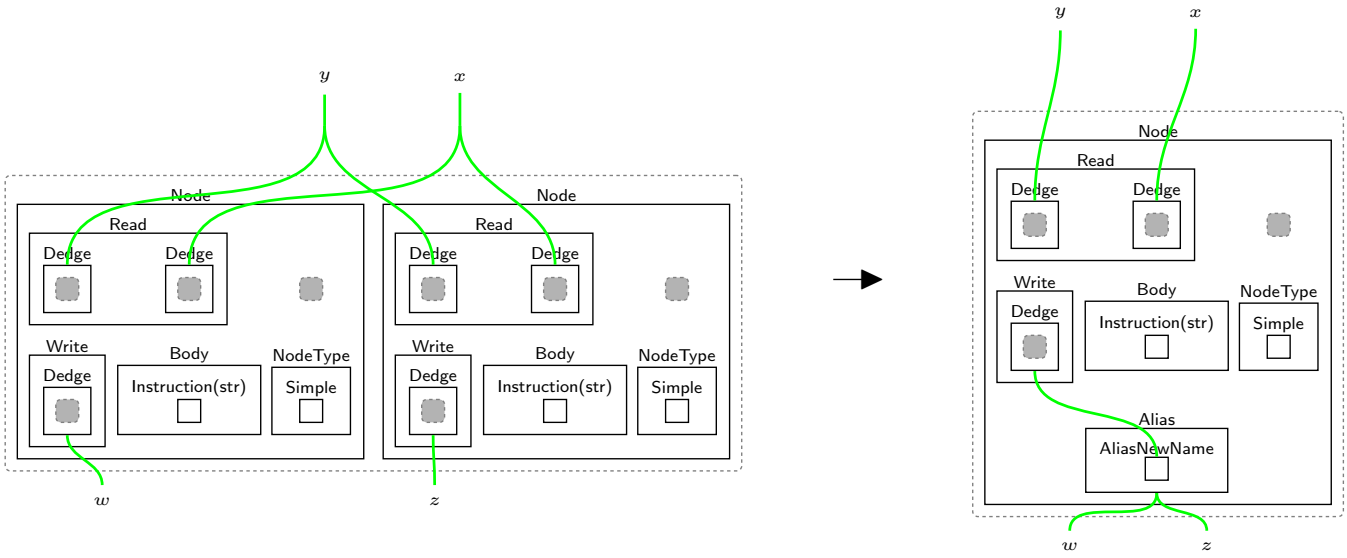


Figure 15: Common Node Elimination – initial rule

E.4 Invariant Value Redirection

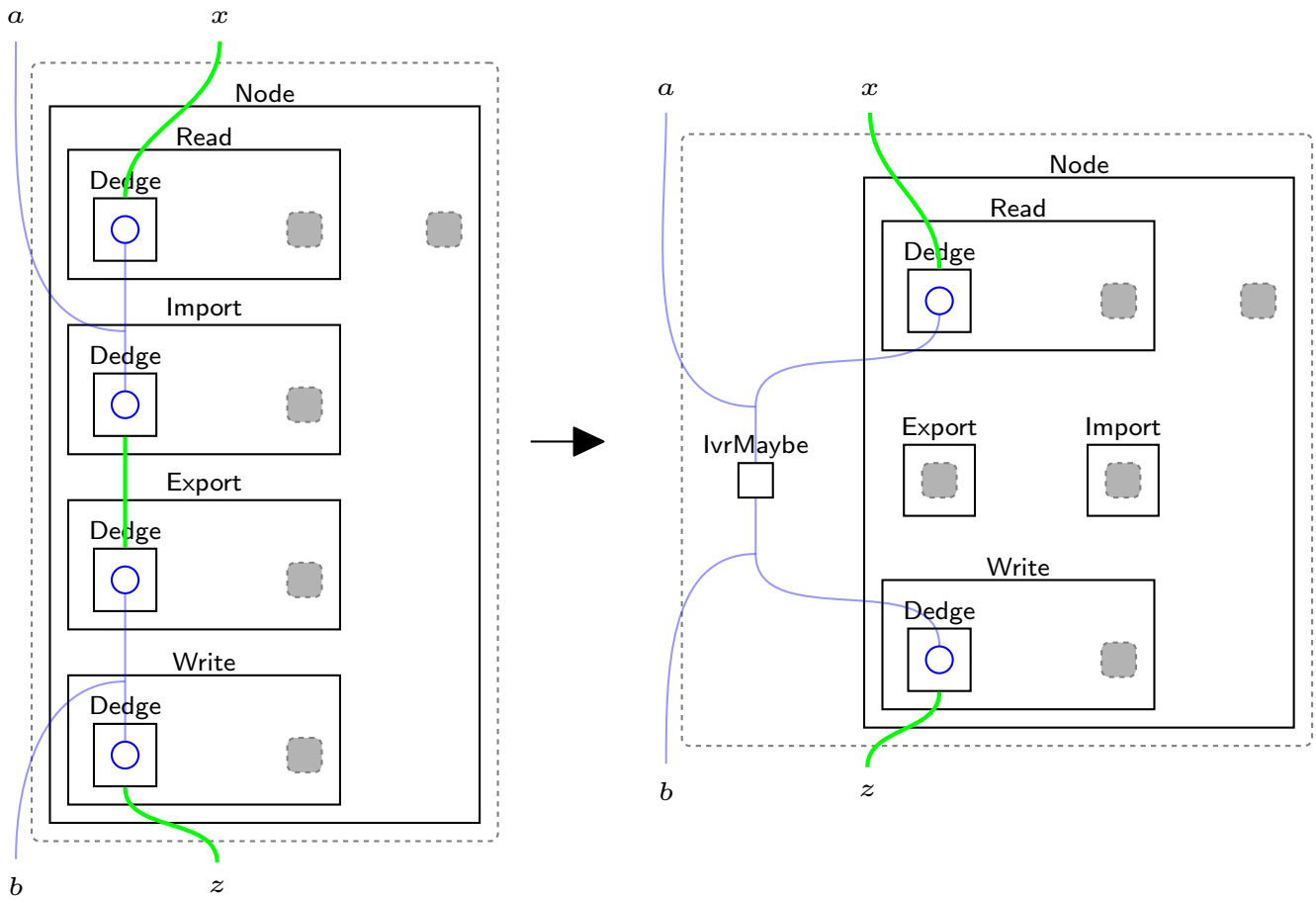


Figure 16: Invariant Value redirection – initial rule