Technical Report

UCAM-CL-TR-705
ISSN 1476-2986

Number 705

UNIVERSITY OF
CAMBRIDGE

Computer Laboratory

Optimizing compilation with the Value State Dependence Graph

Alan C. Lawrence

December 2007

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

© 2007 Alan C. Lawrence

This technical report is based on a dissertation submitted May 2007 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Churchill College.

Some figures in this document are best viewed in colour. If you received a black-and-white copy, please consult the online version if necessary.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Abstract

Most modern compilers are based on variants of the Control Flow Graph. Developments on this representation—specifically, SSA form and the Program Dependence Graph (PDG)—have focused on adding and refining data dependence information, and these suggest the next step is to use a purely data-dependence-based representation such as the VDG (Ernst et al.) or VSDG (Johnson et al.).

This thesis studies such representations, identifying key differences in the information carried by the VSDG and several restricted forms of PDG, which relate to functional programming and continuations. We unify these representations in a new framework for specifying the *sharing* of resources across a computation.

We study the problems posed by using the VSDG, and argue that existing techniques have not solved the *sequentialization* problem of mapping VSDGs back to CFGs. We propose a new compiler architecture breaking sequentialization into several stages which focus on different characteristics of the input VSDG, and tend to be concerned with different properties of the output and target machine. The stages integrate a wide variety of important optimizations, exploit opportunities offered by the VSDG to address many common phase-order problems, and unify many operations previously considered distinct.

Focusing on branch-intensive code, we demonstrate how effective control flow—sometimes superior to that of the original source code, and comparable to the best CFG optimization techniques—can be reconstructed from just the dataflow information comprising the VSDG. Further, a wide variety of more invasive optimizations involving the duplication and specialization of program elements are eased because the VSDG relaxes the CFG's overspecification of instruction and branch ordering. Specifically we identify the optimization of nested branches as generalizing the problem of minimizing boolean expressions.

We conclude that it is now practical to discard the control flow information rather than maintain it in parallel as is done in many previous approaches (e.g. the PDG).

Acknowledgements

Firstly, I would like to thank my supervisor, Professor Alan Mycroft, for his unending enthusiasm, sense of direction, encouragement, and occasional cracking-of-the-whip; and my parents, for their support and understanding throughout. Without these people this thesis would never have been possible. Thanks must also go to my grandmother for her help with accomodation, which has been invaluable.

I would also like to thank my fellow members of the Cambridge Programming Research Group, for the many discussions both on and off the subject of this thesis; and to many friends and officemates for their help throughout in preserving my sanity, or at least the will to go on...

Contents

1 Introduction **11**
- 1.1 Computers and Complexity . . . 11
- 1.2 Compilers and Abstraction . . . 11
- 1.3 Normalization: Many Source Programs to One Target Program . . . 12
- 1.4 Abstraction Inside the Compiler . . . 13
- 1.5 Trends in Intermediate Representations . . . 14
  - 1.5.1 Static Single Assignment . . . 15
  - 1.5.2 The Program Dependence Graph . . . 15
  - 1.5.3 The Value (State) Dependence Graph . . . 17
- 1.6 Optimizations on the VSDG . . . 17
- 1.7 The Sequentialization Problem . . . 18
  - 1.7.1 Sequentialization and Late Decisions . . . 18
- 1.8 Chapter Summary . . . 19

2 The Nature of the Beast **21**
- 2.1 The VSDG by Example . . . 21
  - 2.1.1 Uniformity of Expressions . . . 27
  - 2.1.2 Formal Notations . . . 27
- 2.2 The VSDG and Functional Programming . . . 27
  - 2.2.1 SSA and Strict Languages . . . 27
  - 2.2.2 Encoding the VSDG as a Functional Program . . . 28
  - 2.2.3 Evaluation Strategies . . . 29
  - 2.2.4 Higher-Order Programming . . . 31
- 2.3 The VSDG and the Program Dependence Graph . . . 32
- 2.4 An Architecture for VSDG Sequentialization . . . 32
  - 2.4.1 The VSDG and Phase-Order Problems . . . 34
  - 2.4.2 Three Phases or Two? . . . 35
  - 2.4.3 A Fresh Perspective: Continuations . . . 36
- 2.5 Definition of the PDG . . . 37
- 2.6 The VSDG: Definition and Properties . . . 39
  - 2.6.1 Places, Kinds and Sorts . . . 39

2.6.2 Transitions 40
2.6.3 Edges 41
2.6.4 Labels, Sets and Tuples 41
2.6.5 Hierarchical Petri-Nets 42
2.6.6 Loops 43
2.6.7 Well-Formedness Requirements 43
2.7 Semantics 44
2.7.1 Sequentialization by Semantic Refinement 44
2.7.2 The VSDG as a Reduction System 45
2.7.3 A Trace Semantics of the VSDG 46
2.8 Chapter Summary 48

3 Proceduralization 51
3.1 Choosing an Evaluation Strategy 53
3.2 Foundations of Translation 54
3.2.1 Naïve Algorithm 54
3.2.2 Tail Nodes—an Intuition 55
3.2.3 Normalization of Conditional Predicates 57
3.3 An Algorithmic Framework 57
3.3.1 Dominance and Dominator Trees 58
3.3.2 Gating Conditions 59
3.3.3 The Traversal Algorithm 60
3.4 Additional Operations 62
3.4.1 The γ-Ordering Transformation 62
3.4.2 Coalescing of γ-Trees 65
3.5 Worked Examples 67
3.6 Chapter Summary 71

4 PDG Sequentialization 73
4.1 Duplication-Freedom 73
4.1.1 Effect on Running Examples 74
4.2 A Special Case of PDG Sequentialization 75
4.2.1 Bipartite Graphs and Vertex Coverings 77
4.2.2 Unweighted Solution 78
4.2.3 Weights and Measures 80
4.3 Comparison with VSDG Sequentialization Techniques 82
4.3.1 Johnson's Algorithm 82
4.3.2 Upton's Algorithm 84
4.4 Comparison with Classical CFG Code Motion 85
4.4.1 Suitable Program Points 87
4.4.2 Extra Tests and Branches 87
4.4.3 Program Points and Sequentialization Phases 87
4.4.4 Lifetime Minimization 88
4.4.5 Variable Naming and Textually Identical Expressions 89
4.4.6 Optimal Code Motion on Running Examples 90
4.4.7 Other Algorithms 90

5 Intermediate Representations and Sharing **93**
5.1 Shared Operators: the PDG & VSDG, Part 2 94
5.1.1 Production of Shared Operators 97
5.2 An Explicit Specification of Sharing 98
5.3 A Semantics of Sharing Edges 101
5.3.1 An Alternative View of Proceduralization 103
5.4 Loops 105
5.4.1 Proceduralization and Loops 106
5.5 The RVSDG: a Workable PDG Alternative 106
5.5.1 Strict Nets and γ-Nets 109
5.5.2 Loops 111
5.5.3 Sharing 111
5.5.4 Explicit Representation of Register Moves 112
5.5.5 Well-Formedness Conditions 114
5.5.6 Semantics 116
5.5.7 Performing γ-Ordering on the RVSDG 116
5.5.8 Chapter Summary 117

6 Node Scheduling **121**
6.1 Johnson's Algorithm 121
6.1.1 Atomicity of γ-Regions 125
6.1.2 Node Raising and Speculation 126
6.1.3 Node Cloning and Dominance 126
6.2 Reformulating Johnson's Algorithm on the RVSDG 126
6.2.1 Hierarchy 126
6.2.2 Tail-Sharing Regions 128
6.2.3 Register Allocation 129
6.3 Simple Extensions 130
6.3.1 Heuristics for Speed over Space 130
6.3.2 Instruction Latencies 130
6.3.3 Movement Between Regions 131
6.3.4 Predicated Execution 131
6.4 Alternative Approaches 132
6.5 The Phase-Order Problem Revisited 134
6.5.1 Combining Proceduralization and Node Scheduling 136
6.6 Chapter Summary 138

7 Splitting **139**
7.1 Splitting, *à la* VSDG 139
7.1.1 Enabling Optimizations 140
7.1.2 Transformations on γ-Nodes 141
7.2 Splitting in the RVSDG 143
7.3 Splitting: a Cross-Phase Concern 144
7.4 Optimality Criteria for Splitting 147
7.4.1 Optimal Merge Placement 148
7.4.2 Control Flow Preservation 148
7.4.3 Limited Optimal Splitting 149

7.4.4 Exhaustive Optimal Splitting 150
7.5 Relation to and Application of Existing Techniques 152
7.5.1 Message Splitting 152
7.5.2 Cost-Optimal Code Motion 154
7.5.3 Speculative Code Motion 155
7.5.4 Use of the PDG 155
7.5.5 Restructuring the CFG 157

8 Conclusion 161
8.1 Open Questions 162

A Glossary of Terms 165

B State Edges and Haskell 167
B.1 The State Monad 167
B.2 Encoding Just the State Edges 168
B.3 Adding Values 170
B.4 Even the State Monad is Too Expressive 172
B.5 Well-Formedness Conditions 173
B.6 Derestricting State Edges 174
B.7 A Note on Exceptions 174

Bibliography 177

CHAPTER 1

Introduction

> A modern digital computer is perhaps the most complex toy ever created by man.
>
> *The Computer Revolution in Philosophy*, Aaron Sloman, 1978

1.1 Computers and Complexity

Clearly, the complexity of computer systems has only increased in the years since the quote above. This complexity drives much of computer science, and can be seen as the composition of two forms:

1. The *inherent complexity* in working out and specifying what the system must do precisely and unambiguously. Work with formal specification languages shows that this is still difficult, even with no allowance made for hardware or implementation concerns.

2. The *incidental complexity* of expressing this specification in a form which can be executed by a computer—for example, in a particular programming language. This brings in additional concerns of the *efficiency* of execution.

1.2 Compilers and Abstraction

Abstraction is a powerful tool for managing complexity, so it is no surprise that the use of abstractions in the construction of computer systems, specifically software, has been increasing steadily. One vital source of abstraction for overcoming incidental complexity is the *compiler*, defined by Aho et al. [AU77] as:

> A program that reads a program written in one language—the source language—and translates it into an equivalent program in another language—the target language.

We see this as providing abstraction in two distinct ways.

Firstly, the conversion between programming languages provides the abstraction(s) of the source language: for example, features such as mnemonic instructions instead of binary op-codes, user-declared variables in place of registers, data types over untyped memory locations, higher-order functions, objects, garbage collection, etc...

Secondly, the translation into an *equivalent* program—meaning one which produces the same results and side effects, including termination or otherwise—allows the compiler to *optimize* the result such that it is more efficient in space, time, or both. This moves concerns of efficiency—previously in the domain of the programmer—into the domain of the compiler, such that the programmer need think about less, and can concentrate more fully on the real-world task to be performed[1].

1.3 Normalization: Many Source Programs to One Target Program

Thus, a maxim of compiler folklore is that:

> "The most optimizing compiler is the most normalizing compiler."

Normalization refers to where many different source programs result in the same machine code after compilation—that is, where the same machine code is produced regardless of which form was written by the programmer. In this thesis, we refer to the input programs *being normalized* by the compiler. For the behaviour of the compiler to be considered correct, the two input code sequences must therefore do the same thing (and the output must implement this!), i.e. they have the same observable semantics, and so we can see them as different *representations* of the same underlying idea[2].

This allows the programmer to select from such representations according to which is the easiest to comprehend or to fit his ideas or program into, and leave the compiler to select the output according to efficiency concerns. Thus, the level of abstraction is raised.

Of course, due to decidability issues, in general the compiler cannot select the "best" output, or even list all the possible equivalents; further, the efficiency of different possible outputs may be incomparable, depending on contextual factors such as intended library usage or likely program inputs that are not known at compile-time. Or, the programmer may have written his source code using knowledge of these factors that is merely unavailable to the compiler; in such cases, compiler attempts at "optimization" and normalization may in effect undo hours of work by programmers in hand-optimizing their code and make the code *worse*. This leads to the idea of *conservative* optimization: that which cannot make the program perform worse on *any* execution; clearly, conservatism is a major limitation. Hence, runtime profiling is a better solution: observe how the code is actually used at runtime, and use this information to optimize the code with runtime recompilation. Programmers' conceptions of how code is used are frequently ill-founded, and profiling is more accurate. Such techniques allow the compiler to select an appropriate output form; and where the same or equivalent program fragments

[1] As William Wulf [Wul72] said: " More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason—including blind stupidity."

[2] Some programming languages emphasize orthogonality—the existence of exactly one program construct for each purpose—but even this does not rule out the possibility of combining the same structures in different ways to achieve the same end. Others even make it a feature that "There Is More Than One Way To Do It" (TIMTOWTDI or "Tim Toady") [Wal00]!

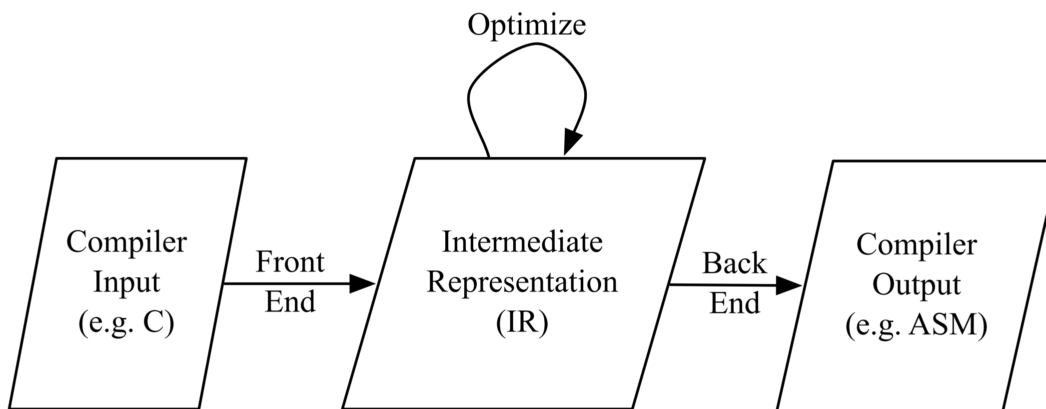


Figure 1.1: Structure of an optimizing compiler. In this thesis, the intermediate representation is the VSDG, and the compiler output can be considered a CFG of machine instructions with physical registers.

appear in multiple different places in the input, it may even choose a different output for each (so "normalization" may be a misleading term).

1.4 Abstraction Inside the Compiler

Because of this increasing demand for optimization, compilers are themselves becoming more and more complex. Correspondingly, the role of abstraction *inside* compilers is becoming more important.

Optimizing compilers are conventionally structured as shown in Figure 1.1: the *front end* of the compiler transforms the input program (e.g. source code) into an Intermediate Representation (IR); optimisation phases operate on the program in this form; and then the *rear end* transforms the program into the desired output format (e.g. a sequence of machine code instructions). *Construction* refers to the production of the IR from the input; paralleling this, we use the term *destruction* to refer to conversion of the program out of the IR to produce machine code. However, for a CFG of machine instructions with physical registers, destruction is trivial—requiring only the placement of the basic blocks into a single list—and thus we tend to see the output of the compiler as being such a CFG.

This thesis is concerned with the Intermediate Representation used, as representing the program in such an IR data structure is one of the main abstractions on which the compiler is based. In fact we can see "an Intermediate Representation" as an *abstract data type* (ADT), with operations of construction, destruction, and intervening optimization, much as "Priority Queue" is an ADT with operations of adding an element and examining and removing the smallest. Different IRs implement the ADT in different ways and are almost interchangeable, and a particular IR should be selected according to its costs for the operations that are expected—much as selecting a structure such as a Heap, Binomial Heap or Fibonacci Heap to implement a Priority Queue.

Specifically (and as a direct result of the increasing demands placed on compilers), the amount of time spent on optimization increases (in both programming and running the compiler), whereas construction and destruction continue to occur exactly once per compilation.

In most compilers, the IR used is the CFG (Control Flow Graph), whereas in this thesis,

we argue for the use of the VSDG (Value State Dependence Graph—both of these are outlined below in the next section). When we compare these two representations as to their costs, we see:

- In the **front end**, the VSDG is slightly more expensive to construct, but this is generally straightforward.
- In the **optimization** stage, although there are exceptions, generally optimizations are significantly cheaper in the VSDG, as well as being easier to write; this is discussed at more length in Section 1.6.
- In the **rear end**, the VSDG requires substantially more effort. Specifically, the VSDG must first be *sequentialized* into a CFG, and this has not previously been seen as straightforward. This is discussed in Section 1.7, and much of this thesis focuses on this problem.

As time progresses and more and more optimization is performed, an IR which facilitates optimization becomes more important, and it becomes worthwhile to spend more effort in conversion both into an IR and out of it, as this effort can be traded against increasing cost savings. Thus, we argue that switching to the VSDG as IR is a logical next step.

1.5 Trends in Intermediate Representations

Changes and augmentations to Intermediate Representations have followed a number of trends, but three main themes stand out:

Assignment becoming less important

Control flow becoming more implicit

Dependencies becoming more explicit

Early compilers used the CFG, as it was simple to construct and destruct. The CFG is a simple flow chart (familiar from process management and non-computer-science fields), with each node labelled with a machine instruction (or optionally series of instructions); for decision nodes (only) the final instruction is a branch instruction (hence in the CFG, such nodes are usually referred to as *branch nodes*). We can see that the CFG is quite primitive and very close to the hardware in all three themes:

Assignment is the sole means by which information is carried from any (non-branch) operation to another, exactly as values are carried from one machine code instruction to another by being stored in registers.

Control Flow is recorded explicitly by successor edges, thus (unnecessarily) ordering even independent statements. We can see the program counter of a traditional CPU as pointing to one node after another.

Dependencies are highly implicit; to identify them, one must perform Reaching Definitions analysis to locate the other statement(s) potentially producing the value depended on. (This is similar to the work done by the reorder buffer in a superscalar processor in order to identify parallelizable statements.)

Branches control which statements are executed; we can see this as additionally encoding information (whether the branch was taken or not) into the PC. After paths merge together, information remains only in the values assigned to variables by statements which the branch caused to be executed (or not).

We now discuss some significant alternatives and refinements; these are shown in the diagram on page 16.

1.5.1 Static Single Assignment

Many "dataflow" analyses were developed to operate on the CFG: for example, live variables, available and busy expressions, reaching definitions. These eventually led to perhaps the first major increase in abstraction: Static Single Assignment, or SSA form [CFR+91]. This adds a restriction that for each variable appearing in the CFG, statically there is a single assignment to it (usually requiring the use of additional variables). At merge points, ϕ-nodes allow merging the variables from different incoming edges to create new variables.

SSA form makes substantial advances in two of the three themes:

Assignment Each variable has only one assignment to it, so the idea that a statement might affect the future by having a side effect onto some variable is avoided. (Information is still only carried by values assigned.)

Dependencies thus become much more obvious, as for any use of a variable, the unique statement producing that value can easily be identified.

This greatly simplifies analysis and hence optimization, but some extra effort is required in production (and sequentialization, although not significantly). However, the same ordering restrictions remain.

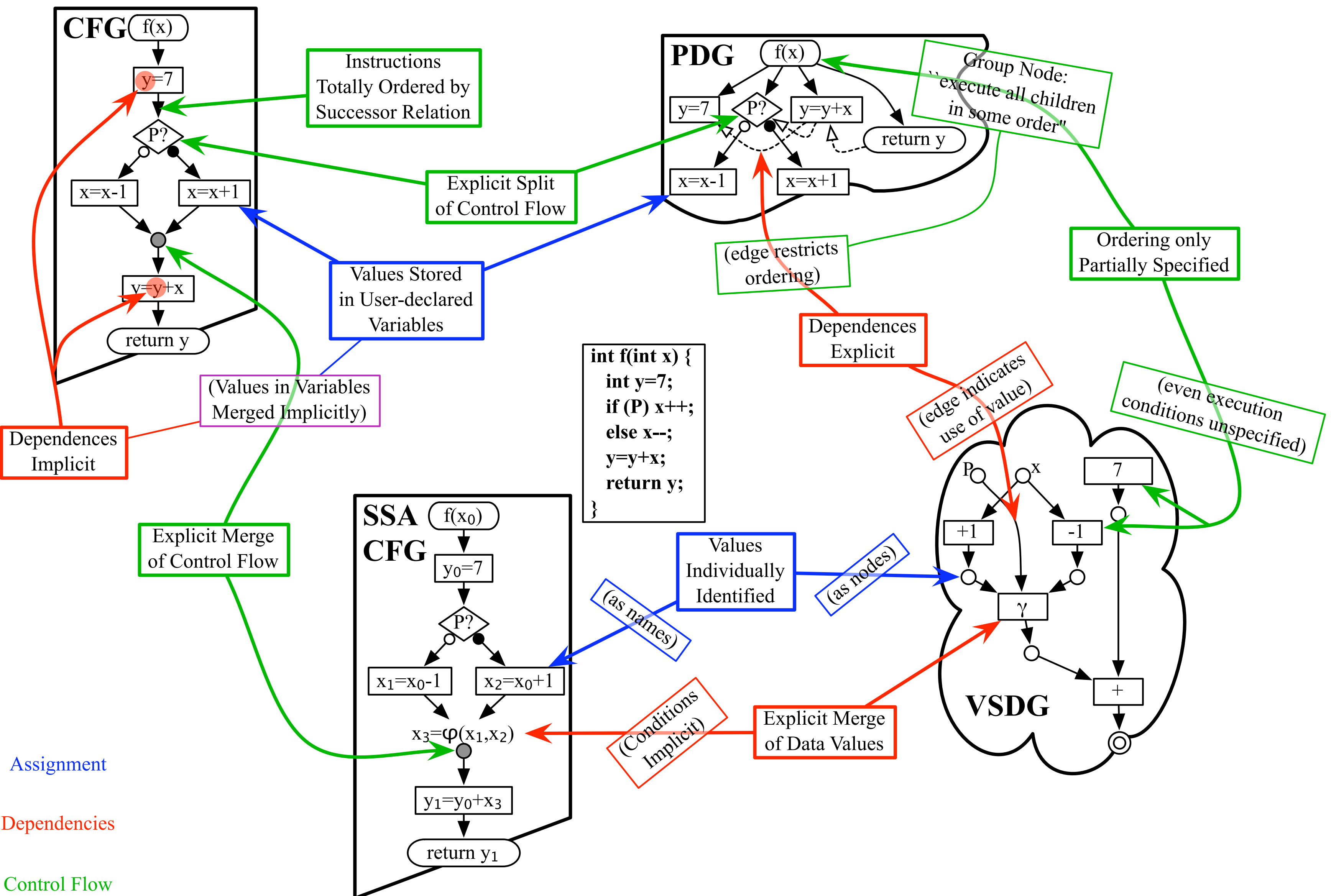
1.5.2 The Program Dependence Graph

An independent development in IRs was the Program Dependence Graph (PDG), introduced in 1987 as a combined representation of both control and data dependence. It has been widely used in analysis tasks such as program slicing [HRB88, Bin99], in parallelization and vectorization optimizations [ZSE04, BHRB89], and as a software engineering and testing tool [OO84, BH93]. Many traditional optimizations operate more efficiently on the PDG [FOW87], and we can see it makes large steps forward in two of the themes outlined earlier:

Control Flow Ordering requirements are substantially relaxed by treating statements such as `x+=1` as atomic and allowing a *group node* which represents "execute all child nodes of this node" in some order. Thus edges no longer represent control flow.

Dependencies are made more explicit than in the CFG by the use of a separate class of data dependence edges between siblings, which restrict the possible orderings. (These are analogous to the dependence edges between tasks used in project management to identify critical paths.)

(In terms of assignment, values are still carried by placing them into variables, much as in the CFG. Hence, SSA form can again be applied to simplify analysis tasks.) However, sequen-



CFG
f(x)
y=7
P?
x=x-1
x=x+1
y=y+x
return y
Instructions Totally Ordered by Successor Relation
Explicit Split of Control Flow
PDG
f(x)
y=7
P?
y=y+x
return y
x=x-1
x=x+1
Group Node: "execute all children in some order"
Values Stored in User-declared Variables
(edge restricts ordering)
Ordering only Partially Specified
Dependences Explicit
(Values in Variables Merged Implicitly)
Dependences Implicit
int f(int x) {
int y=7;
if (P) x++;
else x--;
y=y+x;
return y;
}
(edge indicates use of value)
(even execution conditions unspecified)
Explicit Merge of Control Flow
SSA CFG
f(x0)
y0=7
P?
x1=x0-1
x2=x0+1
x3=φ(x1,x2)
y1=y0+x3
return y1
Values Individually Identified
(as names)
(as nodes)
P
x
7
+1
-1
γ
+
VSDG
(Conditions Implicit)
Explicit Merge of Data Values
Assignment
Dependencies
Control Flow

tialization of the PDG is substantially more difficult, and some PDGs *cannot* be sequentialized without duplicating some of their nodes.

Some ordering restrictions—some specification of Control Flow—remain even in the PDG, however, as the following code sequences are treated as distinct:

```
{ int x = a+b; if (P) y = x;}

if (P) { int x = a+b; y = x;}
```

While this is very suitable in a machine-oriented intermediate code (because of the different timing effects), it is less appropriate for a general optimisation phase, as we may wish to make late decisions on which form to use based on, for example, register pressure *after* optimization.

1.5.3 The Value (State) Dependence Graph

Another, more recent, IR—variously known as the Value Dependence Graph (VDG) [WCES94], Value State Dependence Graph (VSDG) [JM03], or more generally, as the class of Gated Data Dependence Graphs (Gated DDGs) [Upt06]—has also been proposed, but has yet to receive widespread acceptance[3].

This IR can be seen as the logical culmination of all three trends:

Assignment is entirely absent, as it replaces executable/imperative statements with functional operations.

Control Flow is not indicated at all: the VSDG takes the viewpoint that control flow exists only to route appropriate values to appropriate operations, and records merely which values and operations are appropriate, not how this routing can be achieved.

Dependencies are indicated explicitly, including the conditions under which each occurs (i.e. under which each value is selected).

1.6 Optimizations on the VSDG

Earlier, in Section 1.4, we said the VSDG makes the optimization stages—that is, many of the analyses and transformations on which CFG compilers spend their time—much simpler and easier.

An overview of how many traditional CFG optimizations can easily be applied to the VSDG is given by Upton [Upt06]: we summarize that those where the main difficulty is in the dataflow analysis (e.g. constant propagation and folding, algebraic simplification, escape analysis) become much easier in the VSDG. In other cases (e.g. type-like analyses, devirtualization, scalar replacement, loop unrolling) using the VSDG makes little difference, and we see many such optimizations as having clear VSDG equivalents or being able to be "ported" across the IR gap with minimal changes. Indeed, Singer [Sin05] has shown how many optimizations can be done in an identical fashion on CFGs satisfying different restrictions on variable naming and reuse,

[3]The exact differences between these will be explained and reviewed later, but for now we can see the two being based on the same principles—in particular, every VDG is a VSDG. The VSDG adds explicit *state edges* to link operations which must have their ordering preserved (such as the indirect stores resulting from `*x=1; *y=2;` when the aliasing of `x` and `y` is unknown).

including SSA and SSI [Ana99] forms, and we see the same principle as applying equally to the selection of VSDG over CFG (the VSDG is implicitly in SSA form). In some cases (e.g. constant propagation), the precision of the result depends on the IR that was used, but others (e.g. dead-code elimination) work independently of the IR altogether.

Further, we argue that many optimizations can be made more effective and general by so changing or redeveloping them to take advantage of the VSDG's features. Where optimizations seem to occur implicitly by representation in the VSDG, this is a highly effective way of implementing them—although in some cases it may be merely delaying the issues until the sequentialization stage (discussed in Section 1.7.1).

1.7 The Sequentialization Problem

In Section 1.4 we also said that conversion of the VSDG to machine code was difficult. We argue previous attempts at VSDG sequentialization fall short of making use of the VSDG practical, and indeed, Upton suggests that the sequentialization problem is the major obstacle preventing the widespread usage of the VSDG in real compilers. We can see several reasons for this:

- Evaluation of nodes proceeds in a *wavefront*, i.e. a cut across the entire VSDG. This does not fit well with the sequential nature of execution in conventional computers, which have only a single program counter, identifying a single point or program location.

- Evaluation proceeds *nondeterministically*—that is, the VSDG does not specify how the wavefront moves, what nodes it passes over, or in what order (or even how many times). This issue is explored in more detail in Chapter 2, but is in stark contrast to the highly *deterministic* nature of conventional computers, where programs instruct the CPU exactly what to do and in what order[4].

- The VSDG is highly normalizing. Whilst we have discussed the benefits of this already, it also implies a cost: somehow the compiler must choose the best representation to output; this may involve some element of search.

1.7.1 Sequentialization and Late Decisions

Normalization is normally done by the compiler translating all the different source code versions into the same form in its intermediate representation, *either* by some kind of normalizing transformation (which explicitly selects one of the corresponding IR versions), *or* because the IR *cannot* distinguish between them. An example is the ordering of independent instructions: in some IRs (e.g. the CFG), these are explicitly ordered (so a normalizing transformation would be some kind of sort), but in other IRs (e.g. the PDG), the ordering is not specified.

The VSDG tends towards the latter: many optimizations, which would be separate passes in a CFG compiler, seem trivial in a VSDG framework, due to the VSDG's normalizing properties. That is, they automatically (or implicitly) occur merely by representing a program as a VSDG.

However, this does not entirely avoid the problem: rather, it merely delays the decision until the IR is transformed into some form which does make the distinction, namely until the sequentialization stage. Thus, sequentialization wraps up many such optimizations together, and concentrates their difficulty into one place. However, this approach *does* seem helpful in

[4]even if modern out-of-order CPUs may not always follow such instructions exactly!

addressing phase-order problems, which have been a particular problem for CFG compilers which (historically) tend to focus on implementing many distinct optimizations each in its own pass.

1.8 Chapter Summary

We have seen a number of trends suggesting that a switch to the VSDG would be an advantageous progression in intermediate representations and perhaps the logical "next step" in compilers. We also saw that this potential was being held back by of the difficulty of *sequentialization*: converting instances of the VSDG, perhaps highly optimized, back into good sequential CFG code.

Structure of this Thesis We tackle the problem of sequentialization by breaking it into three phases; although previous attempts have used similar ideas implicitly, they have not properly distinguished between these phases. Key to this is the use of a special form of *duplication-free* PDG (df-PDG) as a midpoint between VSDG and CFG.

Chapter 2 explains our compiler architecture and the breakdown into phases by formally defining the VSDG and PDG and reviewing the significance of duplication-freedom. Chapter 3 describes how VSDGs may be converted into PDGs suitable for input to existing techniques of PDG sequentialization. Chapter 4 reviews how these existing techniques produce df-PDGs, and compares our approach with previous techniques for VSDG sequentialization as well as classical CFG code motion. Chapter 5 reconsiders the VSDG-to-PDG conversion process from a more theoretical standpoint, and defines a new data structure, the *Regionalized VSDG* (RVSDG), as a reformulation of the PDG. This is then used in Chapter 6 which considers how df-PDGs (df-RVSDGs) may be converted to CFGs, and critically reviews the earlier breakdown of sequentialization into phases. Chapter 7 reflects on the treatment of sequentialization as a separate stage after optimization (as shown in Figure 1.1) and the difficulties remaining in this structure by studying an optimization of *splitting* which subsumes a wide variety of more aggressive CFG *restructuring* optimizations. Finally, Chapter 8 concludes and suggests further research directions.

CHAPTER 2

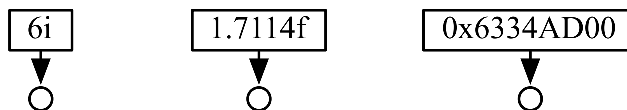
The Nature of the Beast

What the VSDG is, and How it Makes Life Difficult

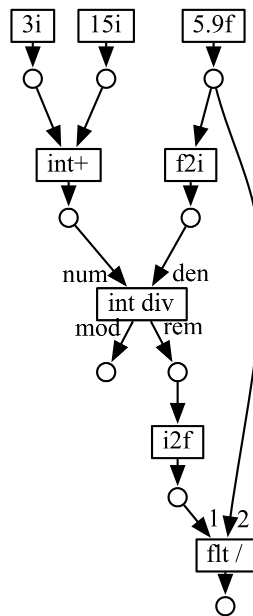
In this Chapter we will explain in detail how a program is represented as a VSDG; how this differs from other IRs, and thus what tasks the sequentialization phase must perform; and how we tackle this problem by breaking sequentialization apart into phases. The chapter concludes with formal definitions of both VSDG and PDG.

2.1 The VSDG by Example

The VSDG represents programs graphically, using nodes and edges. Historically nodes have been computations, and edges represented values. However, for this work we borrow the notation (although not the exact semantics) of Petri-Nets [Pet81]. Nodes come in two forms: *places* (round, representing values) and *transitions* (square, representing operations such as ALU instructions). Such operations include constants—transitions with no predecessors:

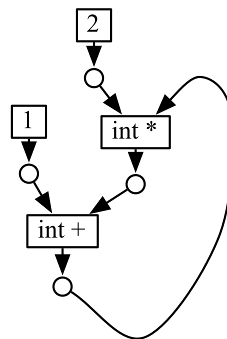


and computations, which have incoming edges to indicate the values on which they operate:

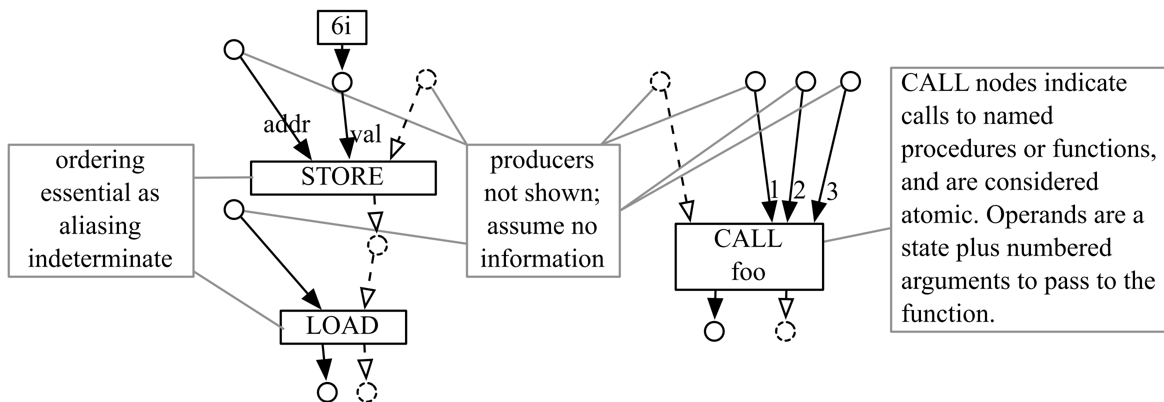


As the above example shows, operations may have multiple operands, and/or produce multiple values, and these are distinguished by edge labels. However each place is the *result* of its unique *producer* transition (this is in contrast to regular Petri Nets). The exact set of operations available is a compiler implementation detail and may depend upon the target architecture.

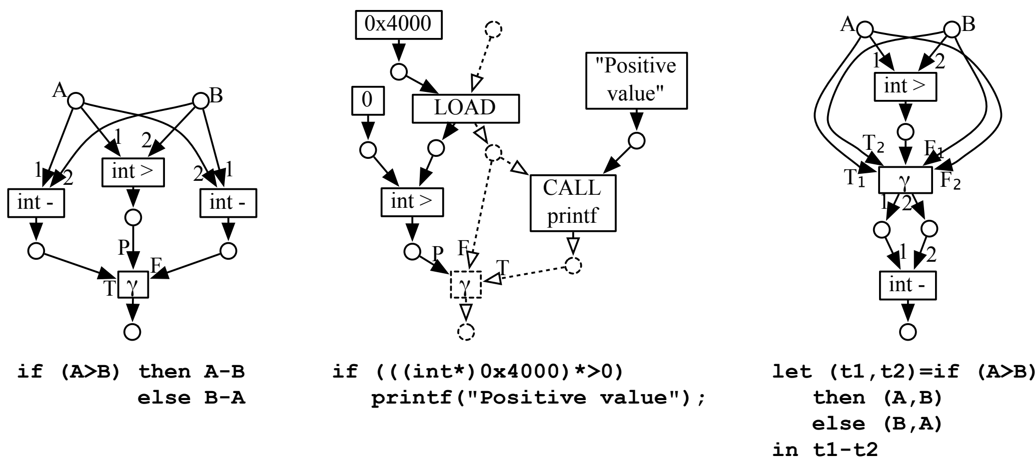
There is no need for a VSDG to be treelike: a single result may be used many times, as in the above. However there must be no cycles; the following example is *not* legal:



Instead of containing values, some *places* may instead contain *states*, which model both the contents of memory (seen as external to the VSDG) and termination (of both loops and function calls), and indicate the essential sequential dependencies between operations due to these. We draw states with dotted boundaries and connecting edges; *linearity* restrictions on the graph ensure that, dynamically, each state produced must be used exactly once:



Choices between values, states, or tuples thereof, are represented using γ-nodes:



(Note how the third example differs from the first by having only a single static subtraction.)

Places may include *arguments*, identifiable by (uniquely) having no predecessor transitions, and also include the *results* of the VSDG, drawn with double outlines. These provide a way for values and/or state to be passed into and out of the VSDG, and may be named to distinguish between them. Note that the same place may be both argument and result, as shown in Figure 2.1.

VSDGs can also be *hierarchical* graphs, in which individual *compound nodes*, drawn with a double outline, may contain entire graphs; the *flattening* operation merges such encapsulated graphs into the outer (containing) graph, as shown in Figure 2.2.

Flattening works by *quotienting* places in the containing graph with those in the contained graph, according to edge and place labels in the respective graphs. (Note that the kind—value or state—of an edge or place is a label.) The dotted group of nodes in the result records the extent of the contained group, allowing an inverse operation of *nesting*.

Figure 2.3 shows how this mechanism is used to represent loops, by using standard μ fixpoint notation to write an infinite (regular, treelike) hierarchy. As Figure 2.4 shows, this is similar to the representation of functions.

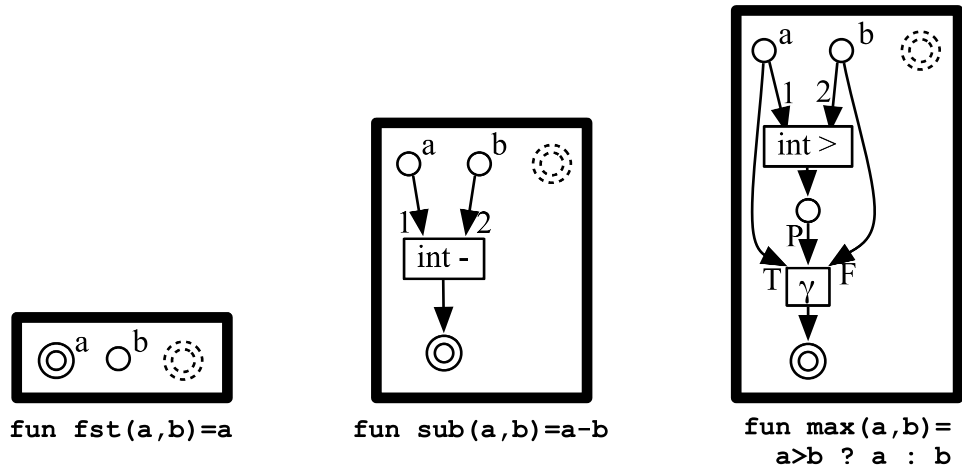


Figure 2.1: Argument and Result places are drawn with double outlines

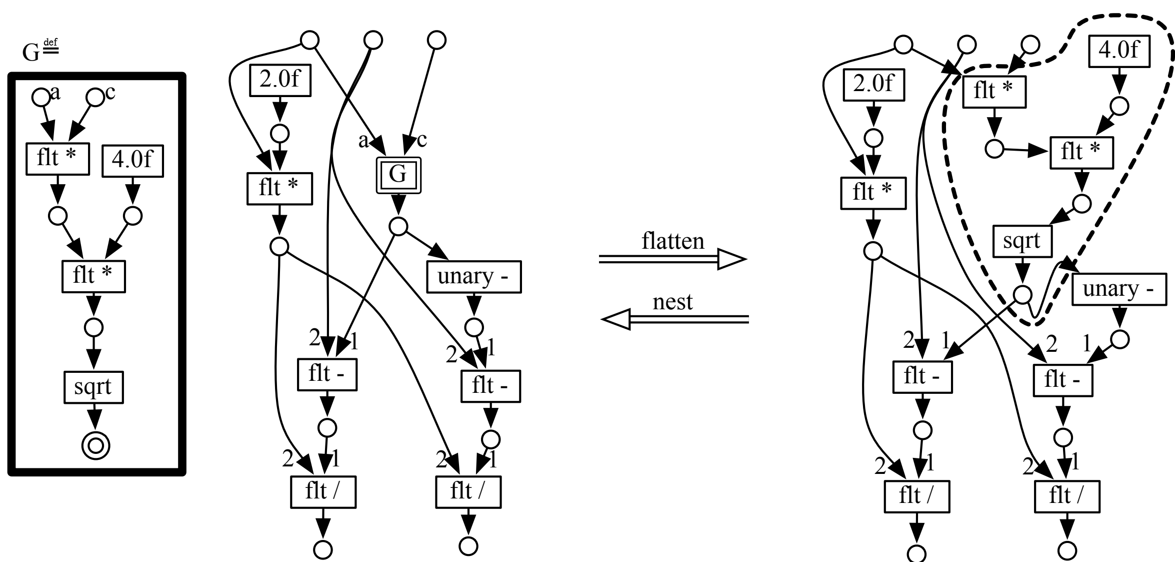


Figure 2.2: VSDGs can also be hierarchical graphs.

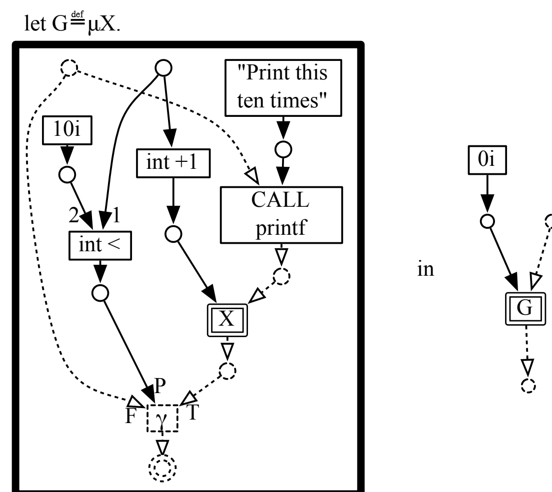


Figure 2.3: Loops are represented using the μ-operator. (The syntactic definition $G \stackrel{\text{def}}{=} \mu X.\ldots$ can alternatively be seen as a declaration `let rec G=...` $\left[{}^{G}\!/\!{}_{X}\right]$)

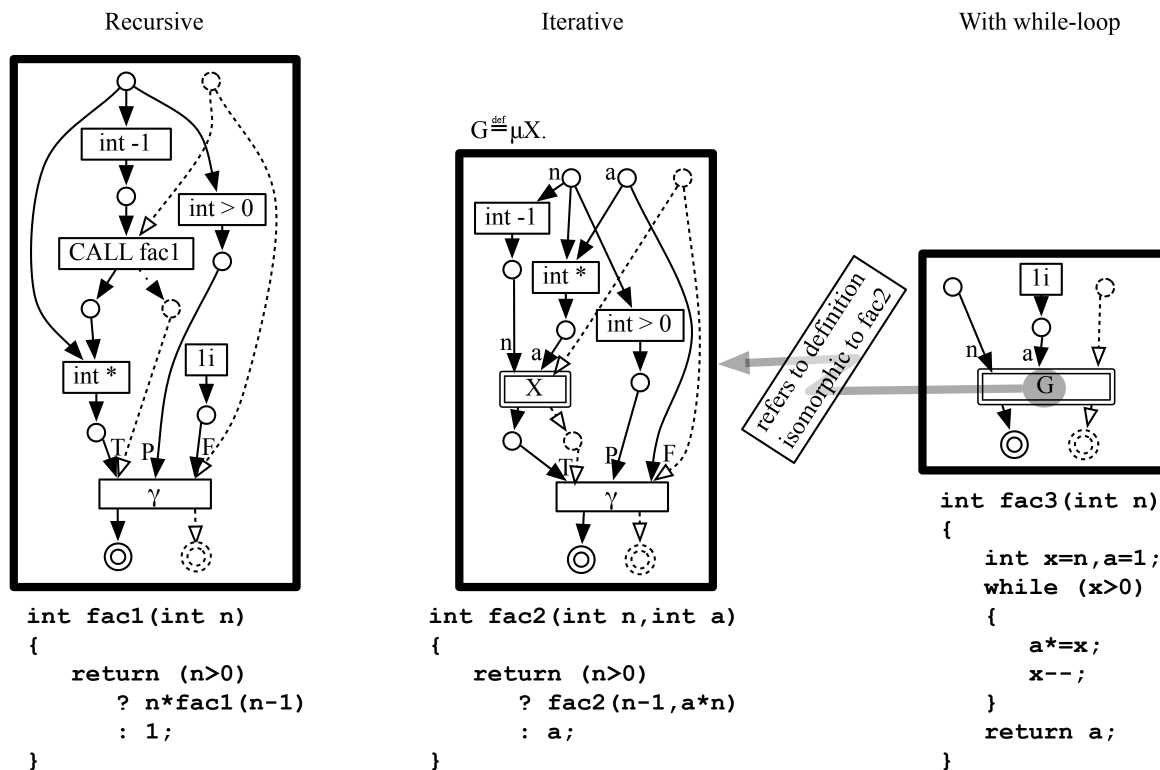


Figure 2.4: Three representations of the factorial function. Here only the first uses the stack, to make recursive function calls `fac1(...)`. The second instead uses iteration, which can be seen as a loop (*à la* tail calls in ML); the third wraps this loop in a procedure taking one fewer argument. A valid alternative form of the second VSDG exists in which the complex node in the recursive definition—shown textually as containing X—is replaced by CALL fac2; this would also use the stack (much as an “iterative” function might if it were not subject to the tail call optimization).

2.1.1 Uniformity of Expressions

A key feature of the VSDG is that γ-nodes—which implement choice between or selection of values (and states)—are just nodes like any other (arithmetic) operation. Thus, many transformations naturally treat expressions in the same way regardless of whether they are spread over multiple basic blocks or just one: the basic block structure is part of the expression. This contrasts with the CFG's two-level system where the *shape* of the graph (i.e. the structure of nodes and edges) specifies the control-flow—which implements choice between or selection of values (and states)—and the *labelling* of the nodes specifies the other (arithmetic) operations: thus, many transformations work by moving the labels (instructions) around within and between the nodes (basic blocks).

This ability to treat expressions uniformly is a recurring theme throughout this thesis, and is particularly important in Chapters 6 and 7.

2.1.2 Formal Notations

We use n to range over nodes (either places or transitions) generally;

r, s, x, y for arbitrary places or results, usually values, or σ to indicate specifically a state;

t, u, v or sometimes op for operations or transitions (functions in the mathematical sense). We tend to use `foo` and `bar` for external functions, especially those whose invocation requires and produces a *state* operand, as opposed to those which have been proven side-effect-free and always-terminating.

g ranges over γ-nodes;

P, Q range over predicates, both in the VSDG (where they are places containing value of boolean type) and more generally (including in the CFG and PDG, where P and Q can also identify a corresponding branch or predicate node);

G ranges over either graphs (including hierarchical VSDGs) or, according to context, PDG group nodes.

We borrow other notations from Petri-Nets, writing ${}^\bullet t$ for the set of *predecessors* (operands) to a transition t, and $t^\bullet$ for the set of successors (results) of t. We also write $s^\bullet$ for the set of successors (now *consumers*) of a place s, and by abuse of notation apply the same notation for sets of places, thus: $S'^\bullet = \bigcup_{s \in S'} s^\bullet$. (Hence, $t^{\bullet\bullet}$ is the set of consumers of any result of t.) However we write ${}^\circ s$ for the predecessor (producer) of a place s—recall this is a single transition not a set—and again implicitly lift this over sets of places.

2.2 The VSDG and Functional Programming

In this section we explore the connection between the VSDG and functional programming languages such as ML and Haskell. This yields some insight into the problem of sequentialising the VSDG.

2.2.1 SSA and Strict Languages

Appel [App98] has previously noted that, because SSA effectively removes the (imperative) notion of assignment, any program in SSA form is equivalent to a functional program; the equivalence can be seen as follows:

1. Liveness analysis is performed, to identify the SSA variables live at entry to each basic

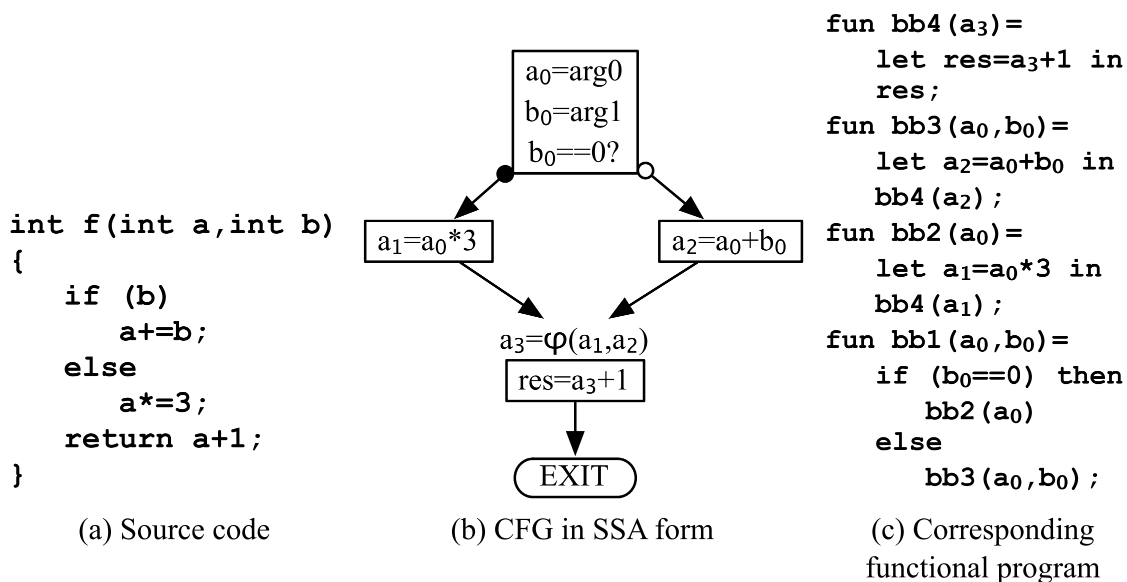


Figure 2.5: Correspondence between SSA Form and Functional Programming

block (including results of ϕ-functions, not their operands)

2. For each basic block, there is a corresponding named function, with parameters being the live variables

3. Each assignment statement $v_1 := op(v_2, v_3)$ corresponds to a `let` statement `let` v_1=$op(v_2, v_3)$

4. Each successor edge is converted into a call to the corresponding function, passing as arguments:
   - For variables (parameters) defined in the CFG by a ϕ-function, the corresponding *operand* to the ϕ-function
   - For other variables, the same variable in the calling function

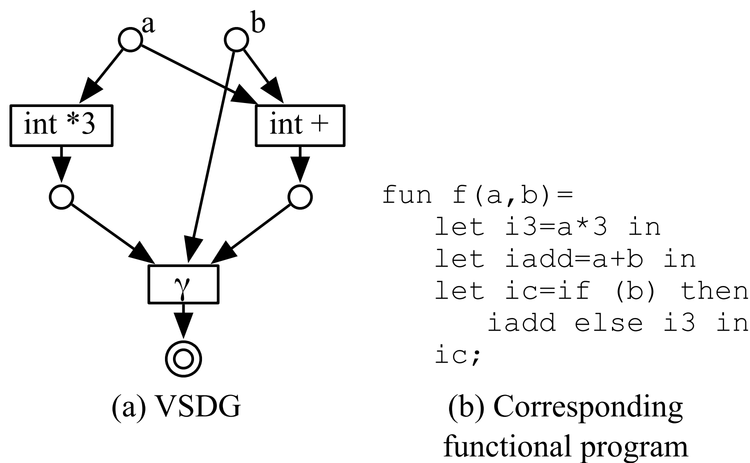
An example of this correspondence is given in Figure 2.5. However, particularly interesting for us is the equivalence of evaluation behaviour. Execution of a basic block in a CFG proceeds by evaluating the RHS, and storing the result in the LHS, of each statement in turn. This is the same as the behaviour when evaluating the corresponding function *in a strict language* (such as ML): for each `let` statement in turn, the expression is evaluated and assigned to the variable, before proceeding to the body of the `let` (i.e. the next such statement)[1]. The functional program is in fact in Administrative Normal Form (ANF) [CKZ03].

2.2.2 Encoding the VSDG as a Functional Program

Since the VSDG is implicitly in SSA form (every value has a single producer!), it is not surprising that a translation to an equivalent[2] functional program exists. The translation for stateless

[1]Even the ordering of side effects is thus preserved.

[2]Having the same observable semantics—this is discussed in Section 2.7.



(a) VSDG

(b) Corresponding functional program

Figure 2.6: Correspondence between VSDG and Functional Programming (Source code given in Figure 2.5(a).)

(VDG) fragments is as follows:

1. Order the transitions of the V(S)DG into any topological sort respecting their dependence edges. (Such an order is guaranteed to exist because the VSDG is acyclic.)
2. Assign each place s a unique variable name v_s
3. For each transition t in topological sort order, let $\overrightarrow{v_{\mathrm{i}}}$ be the set of variable names corresponding to t's operand places $^{\bullet}t$, and let $\overrightarrow{v_{\mathrm{r}}}$ be the set of variable names corresponding to its results $t^{\bullet}$. Output the statement `let` $(\overrightarrow{v_{\mathrm{r}}})$=*op*$(\overrightarrow{v_{\mathrm{i}}})$ `in` where *op* is the function implementing the node's operation. ($\overrightarrow{v_{\mathrm{i}}}$ are guaranteed to be in scope because of the topological sort.)

An example is given in Figure 2.6. (For stateful VSDGs, the same principles apply, but the translation is more complicated and given in Appendix B).

2.2.3 Evaluation Strategies

In Section 2.2.1 we observed that the evaluation behaviour of the CFG was the same as its corresponding functional program under *strict* language semantics. However, the semantics of the VSDG do not specify an exact evaluation behaviour, i.e. which transitions are executed—merely which places are *used* as part of the overall result. In fact stateless fragments of the VSDG, like expressions in pure functional programming languages, are *referentially transparent*:

> **Referential Transparency** An expression E is referentially transparent if any subexpression and its value (the result of evaluating it) can be interchanged without changing the value of E.
>
> *Definition from FOLDOC, the Free On-Line Dictionary Of Computing,* `www.foldoc.org`

A consequence of this is the *Church-Rosser theorem*:

Church-Rosser Theorem A property of a reduction system that states that if an expression can be reduced by zero or more reduction steps to either expression M or expression N then there exists some other expression to which both M and N can be reduced.

Such *reduction systems* include both the λ-calculus and the VSDG (we will use λ-calculus for this discussion, because its properties are well-known, even though we will not use any higher-order features until Chapter 5). From the perspective of the λ-calculus, this means that any term has at most one normal form, or (equivalently), any sequence of reductions that terminates in a normal form (i.e. value) will terminate in the same value. From the perspective of the VSDG, this means that the choice of which VSDG nodes are evaluated will not affect the result. Since the VSDG only describes the resulting value, *it does not specify which nodes are evaluated*.

Several *evaluation strategies* (or *reduction strategies*) have been suggested for λ-calculus terms, and these can also be applied to the VSDG:

Call-by-Name Recompute the value each time it is needed.

Call-by-Need Compute the value if and only if it is needed; but if it is, store the value, such that on subsequent occassions the value will not be recomputed but rather the stored value will be returned.

Call-by-Value Compute the value immediately, whether it is needed or not, and store the result ready for use.

We can see these strategies in a range of contexts. In the λ-calculus, call-by-value means reducing the RHS of any application *before* the application itself; call-by-name the opposite (thus, always performing the outermost reduction first); and call-by-need is as call-by-name but requires representing the expression using a DAG rather than a tree[3]. In functional programming languages, the strategies typically refer to the treatment of arguments to procedure calls; in a call-by-value language, arguments are evaluated *eagerly*, before calling, so evaluation of an expression is only avoided by it being on one arm of an `if`.

However, a more insightful perspective is obtained by considering `let` statements. These allow the same subexpression to be referenced in multiple places, for example `let x=`E `in if` E' `then x else if` E'' `then x else` E'''. In a call-by-value language (such as ML), evaluation of such a statement *first evaluates the argument to the* `let` (E in the example above); the body is then processed with `x` already containing a value. Contrastingly, in a call-by-need language such as Haskell, the argument E of the `let`-expr need be evaluated *only if* `x` *is used in the body*.

A great deal of work has been put into attempting to make lazy languages execute quickly. Observing that call-by-value is more efficient on today's sequential imperative hardware, optimizations attempt to use eager evaluation *selectively* in order to boost performance. The key issue in all cases is to preserve the termination semantics of the language, i.e. call-by-need. *Strictness analysis* [Myc80] attempts to identify expressions which *will definitely be evaluated*, and then computes their values early. Other techniques evaluate expressions *speculatively*, that is, before before they are known to be needed: termination analysis [Gie95] identifies subexpressions which will definitely terminate, and can thus be speculated safely; optimistic evaluation [Enn04] may begin evaluating any expression, but aborts upon reaching a time-limit.

[3]Significantly, call-by-need terminates if and only if call-by-name terminates, but will do so in at most the same number of reduction steps, often fewer.

(Timeouts are then adjusted adaptively by a runtime system to avoid wasting excessive cycles on computations whose results may be discarded). However despite these efforts the performance of lazy languages generally lags behind that of strict ones.

In the VSDG, the situation is similar: the compiler may select a non-lazy evaluation strategy to try to boost performance, so long as the termination semantics of call-by-need are preserved. *State edges* explicitly distinguish operations which may not be speculated—those with side-effects, or which may not terminate[4]—from all others, which may be; but just as for functional programs, the compiler must avoid *infinite* speculation, as this would introduce non-termination[5]. Moreover, the same choice remains, between the potential wasted effort of speculative evaluation, and the difficulty of arranging control flow for lazy evaluation.

Thus, whereas an (SSA) CFG corresponds to a functional program in a *strict* language, we see a VSDG as corresponding to its equivalent functional program in a *lazy* language.

2.2.4 Higher-Order Programming

Higher-order programming refers generally to the use of variables to store functions or computations, that is, to store code rather than values. Thus, a higher-order variable is one containing a *stored procedure*; this can be called or invoked as any other procedure, but the actual code that is executed depends upon the runtime value of the variable.

In the context of a programming language explicitly supporting higher-order programming, implementation probably involves such a variable containing a function pointer—i.e. the code's address in memory—with invocation being an *indirect* jump. This might seem to have little in common with the VSDG; the *Value* (State) Dependence Graph represents a computation in terms of the *values* involved, with edges showing which other values computation requires. That every node is a value, not a function, is quite clear in the Haskell translation of Section 2.2.2: every variable is of ground type.

However, as discussed in Section 2.2.3, the evaluation model of the VSDG means that *every* variable, although of ground type, implicitly stores not a value but rather the computation required to produce it. For call-by-need rather than call-by-name, we can see the stored computation as being of the form

```
if (!COMPUTED) {
   VALUE=...;
   COMPUTED=TRUE;
};
return VALUE;
```

In contrast to Haskell—we are using the VSDG for intraprocedural compilation and optimization—the implementation of such stored procedures will generally be to *inline* a copy of the stored procedure *into each call site*. (The possibility of using VSDG compilation techniques for first-order Haskell program fragments remains, however.)

[4]Potentially also those which may raise exceptions—this is discussed in Appendix B.7.

[5]Thus the semantics of the VSDG, in Section 2.7.2, insist on a *fairness* criterion which allows only a *finite* amount of speculation.

2.3 The VSDG and the Program Dependence Graph

The PDG has been mentioned already, and is defined formally in Section 2.5. We previously described the PDG as being an intermediate point between the CFG and VSDG in terms of its representation of control flow: whereas in the CFG nodes have explicit control flow successor edges, in the PDG there is *some* control flow information—roughly, "execute this under these conditions".

In comparison, the VSDG does not represent such details (recall its translation into a functional programming language, in Section 2.2.2): it deals only with the use of values (with each `let`-expr describing *how* each value may be computed from each other), rather than the execution of the statements or computations (implicitly stored in each variable) producing those values—by the principle of referential transparency, these may be executed at any time. Whereas the VSDG is functional, the PDG is *imperative*, with extra information in its CDG component. This describes *what* to compute and *when*—that is, *the Control Dependence subgraph encodes an evaluation strategy*.

A particular insight stems from the *ill-formed* PDG of Figure 2.8 on page 38. This is seen as an incomplete specification of either of the following (semantically different—e.g. consider S as `x+=1`) programs:

```
{if (P) S;} {if (P') S;}
{if (P || P') S;}
```

Observe that the first of these corresponds to call-by-name, and the second to call-by-need. Hence, the graph of Figure 2.8 is not a valid PDG because it does not specify which evaluation strategy to use, and a PDG must do so. In contrast, VSDGs of similar structure *are* legal[6], and the VSDG allows the *choice* of either strategy. (This poses problems for naïve sequentialization algorithms, considered in Section 3.2.1.)

2.4 An Architecture for VSDG Sequentialization

We tackle the sequentialization problem by breaking it into three phases, as shown in Figure 2.4:

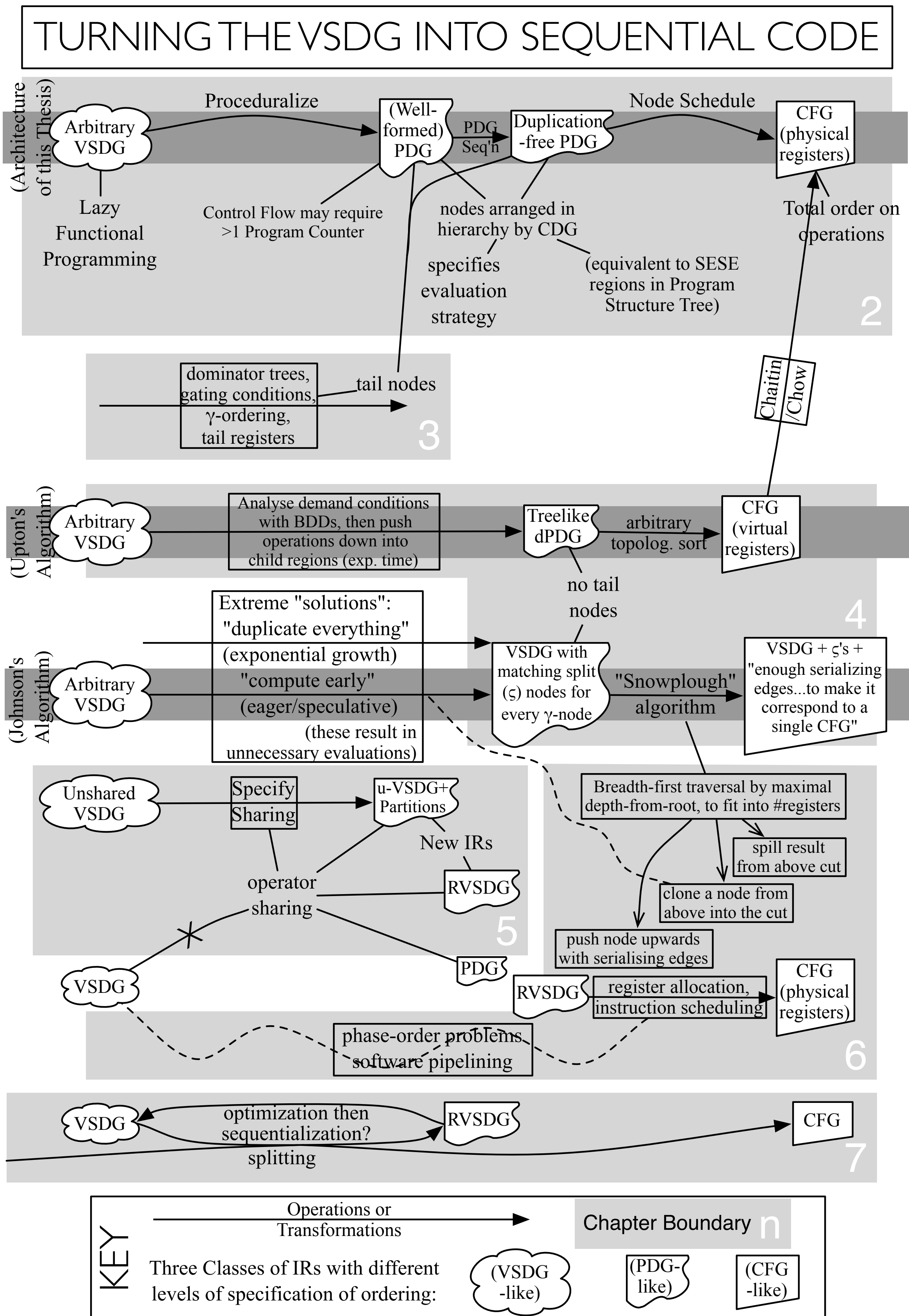
Proceduralization: conversion from (parallel) *functional* to (parallel) *imperative* form—that is, from referentially-transparent expressions into side-effecting statements. From the VSDG, this produces a PDG, and requires selection of an *evaluation strategy*, working out a structure for the program to implement this strategy, and encoding that structure in the Control Dependence Graph subgraph of the PDG (missing from the VSDG). Chapter 3 covers this in detail.

PDG Sequentialization (by existing techniques): we see this as converting from a PDG to a special form of *duplication-free* PDG or *df-PDG*, explained below. Chapter 4 reviews these techniques and discusses related issues.

Node Scheduling: conversion from *parallel* (imperative) to *serial* (imperative) form—this consists of putting the unordered program elements of the df-PDG (statements and groups) into an order, producing a CFG. Chapter 6 covers this.

Key to our approach is the use of the Program Dependence Graph (PDG, defined in Section 2.5 below), including the integration of existing techniques of PDG sequentialization.

[6]Except when S is stateful; in such cases linearity restrictions require that one behaviour is specified.


TURNING THE VSDG INTO SEQUENTIAL CODE
(Architecture of this Thesis)
Arbitrary VSDG
Proceduralize
(Well-formed) PDG
PDG Seq'n
Duplication -free PDG
Node Schedule
CFG (physical registers)
Lazy Functional Programming
Control Flow may require >1 Program Counter
nodes arranged in hierarchy by CDG
specifies evaluation strategy
(equivalent to SESE regions in Program Structure Tree)
Total order on operations
2
dominator trees, gating conditions, γ-ordering, tail registers
tail nodes
3
Chaitin /Chow
(Upton's Algorithm)
Arbitrary VSDG
Analyse demand conditions with BDDs, then push operations down into child regions (exp. time)
Treelike dPDG
arbitrary topolog. sort
CFG (virtual registers)
no tail nodes
4
Extreme "solutions": "duplicate everything" (exponential growth) "compute early" (eager/speculative) (these result in unnecessary evaluations)
(Johnson's Algorithm)
Arbitrary VSDG
VSDG with matching split (ς) nodes for every γ-node
"Snowplough" algorithm
VSDG + ς's + "enough serializing edges...to make it correspond to a single CFG"
Unshared VSDG
Specify Sharing
u-VSDG+ Partitions
New IRs
operator sharing
RVSDG
PDG
VSDG
5
Breadth-first traversal by maximal depth-from-root, to fit into #registers
spill result from above cut
clone a node from above into the cut
push node upwards with serialising edges
RVSDG
register allocation, instruction scheduling
CFG (physical registers)
phase-order problems software pipelining
6
VSDG
optimization then sequentialization?
RVSDG
splitting
CFG
7
KEY
Operations or Transformations
Chapter Boundary n
Three Classes of IRs with different levels of specification of ordering:
(VSDG -like)
(PDG- like)
(CFG -like)

PDG Sequentialization is usually seen as converting from PDG to CFG; since the PDG already contains imperative statements which can be placed directly onto nodes of a CFG, the main task is in arranging the statements—specifically, ordering the children of each group node—so that all the children execute when the group does. A key question is:

> Given a node n which is a child of two group nodes G_1 and G_2, is it possible to route executions of n resulting from G_1 and G_2 through only a single copy of n in the CFG?

The intuition is that if n can be executed *last* among the children of both G_1 and G_2, then a single CFG node suffices[7]; if not, then two copies of n—one for each parent—are required. Clearly, no ordering of the children of a group node can put every child last(!), but in a special class of PDGs known as *duplication-free* PDGs (df-PDGs), no nodes need be duplicated to make a CFG. Intuitively, these can be characterized as having control flow which can be implemented using only a single Program Counter (PC).

We mentioned above that PDG sequentialization is normally seen as transforming a PDG into a CFG: the task is to exhibit a *total ordering* of each group node's children, which can be naturally encoded into the successor relation of a CFG. However, it is possible for many different sequentializations (orderings) to exist: Ball and Horwitz [BH92] show that in any *duplication-free* PDG, every pair of siblings either *must* be placed in a particular order in the resulting CFG (property OrderFixed), or *may* be placed in either order (property OrderArbitrary). That is, the OrderFixeds form the elements of a *partial* ordering, and *every* total ordering respecting it is a valid (CFG) sequentialization.

In our compilation framework, we wish to preserve as much flexibility in ordering as possible for the node scheduler. Thus, we use existing PDG sequentialization techniques only to identify which pairs of nodes in a df-PDG are OrderFixed or OrderArbitrary; ordering within groups of OrderArbitrary nodes is left to the node scheduler. (That is, the node scheduler must take into account ordering restrictions resulting from PDG sequentialization as well as data dependence.) Instead, the main task of PDG sequentialization is to *make* the PDG duplication-free. This problem has not been well-studied, but we consider it in Chapter 4.

2.4.1 The VSDG and Phase-Order Problems

The phase-order problem is a long-standing issue in optimising compilers.

On the CFG, individual optimizations—such as register allocation, instruction scheduling for removing pipeline bubbles, or code motion—are often implemented individually, as separate passes. The phase-order problem lies in that for many pairs of these phases, each can hinder the other if done first: register allocation prevents instruction scheduling from permuting instructions assigned to the same physical registers, whereas instruction scheduling lengthens variable lifetimes which makes allocation more difficult. Thus, there is *no* good ordering for these phases.

The alternative approach is to try to combine phases together—leading to register-pressure-sensitive scheduling algorithms [NP95], for example. This makes the concept of optimality

[7] This ordering of statements allows control flow to "fall through", capturing the desirable possibility of using *shortcircuit evaluation* and generalizing the *cross-jumping* optimization on CFGs.

harder to identify, and much harder to reach. However this is only because the goals of optimality in the individual phases are discarded; and we argue that these goals have always been somewhat illusory: ultimately it is the quality of the final code that counts.

The VSDG leans towards the second approach. Its increased flexibility, and the lack of constraints to code motion it offers, have proven helpful in combining profitably some phases which have been particularly antagonistic in the past: for example common-subexpression elimination and register allocation (dealt with by Johnson [JM03]), and instruction scheduling (in Chapter 6). Important here is the way the VSDG wraps up many normalizing transformations, which would be discrete phases in CFG compilers, into the sequentialization stage (Section 1.7.1).

In this way, many of the phase-order problems of CFG compilers are reduced. They are not entirely eliminated, in that the phases identified above are not perfectly separate, and so in a truly *optimising* compiler[8], the phases would have to be combined and/or the interactions between them considered. Hence, our VSDG compiler will suffer similar, but different, phase-order problems to those of CFG compilers. However we argue that these are less severe in our case for two reasons:

1. We have fewer phases than traditional CFG compilers (as an extreme example, the Jikes RVM optimizing compiler has over 100 phases including many repetitions).

2. The phases we use have more orthogonal concerns. Specifically, proceduralization is concerned with the shape (or interval structure) of the graph (that is, nodes and subtrees are interchangeable), and is largely machine-independent, whereas node scheduling looks at the size or node-level structure of the graph, and deals with machine-dependent characteristics such as register file width and instruction latencies.

(We discuss the vestigial problems with our division into phases further in Section 6.5, including some possibilities for interleaving them at a fine level. However it is worth bearing in mind, in any attempt to truly "combine" the phases of proceduralization and node scheduling, that the task of putting unordered program elements into an order first requires knowing what program elements must be ordered!)

2.4.2 Three Phases or Two?

The division of VSDG sequentialization into three phases leaves PDG sequentialization sitting somewhat uncomfortably in the middle of the other two: at times it seems like the last part of proceduralization, and at other times like the first part of node scheduling.

PDG sequentialization seems part of node scheduling because it is converting from parallel to serial by *interleaving* the partial ordering of data dependences into a total ordering, taking into account tail-sharing. The data dependences are clearly a specification of "what to do next" *for each value*, and the total ordering is a specification of "what to do next" *with only a single program counter*. (Thus, that we need to interleave them at all, is a result of a machine-dependent characteristic—PDG sequentialization might be omitted on multicore architectures with lightweight fork-and-join mechanisms. Node Scheduling might then be used only to interleave multiple computations with identical control conditions, for which ILP suffices and thread-creation overhead may be avoided.)

[8] Practical compilers might better be described as *ameliorating*.

However, PDG sequentialization also seems part of proceduralization, because it can be seen as part of the conversion from functional to procedural: it concerns the duplication *required* to encode (into any resulting CFG) an evaluation strategy. Moreover, many of the tasks performed by proceduralization concern themselves with the size of the output program *after* PDG sequentialization, and previous techniques for VSDG sequentialization have passed through a point equivalent to the *df-PDG*—this is useful for comparison purposes, covered in Chapter 4. (However, other techniques construct only restricted *treelike* PDGs, which are trivially duplication-free, and much of the power of our architecture comes from the ability to use more general df-PDGs.)

Thus, we find it neatest to leave PDG sequentialization as a separate phase.

2.4.3 A Fresh Perspective: Continuations

Much of the task of sequentialization relates to specifying *continuations*. One normally thinks of a continuation as a record of the calling context when making a procedure call (to an explicitly named procedure, using a direct jump, or indirectly to a function pointer), such that execution can resume there after the procedure completes; typically, the continuation is the return address, passed as an extra argument. However, as we saw in Section 2.2.4, each VSDG node implicitly stores a computation; and our approach of inlining calls to such procedures stores the return continuation in the CFG successor relation (one can see this as using the memory address of each copy to identify the "return continuation" after that call site).

In the VSDG the nearest thing each node has to a continuation is the set of its consumers, i.e. the operations which *might* be performed on its value. However, this does not even make clear *which* of these consumers will ever execute in any program run.

In the PDG group nodes add an additional level of specification: the continuation of a group node is to execute *all* of its children (in some order). Although still somewhat vague, this is more precise than the VSDG.

The effect of PDG sequentialization is to specify the continuations much more precisely, including specifying continuations for the statement nodes: when the children of a group node are totally ordered, each sibling's continuation is to go on to execute the next sibling[9].

In the CFG each statement's continuation—"what to do next" after executing that statement—is explicitly specified as its control flow successor[10].

The perspective of continuations leads to two insights:

1. Whereas in PDG and VSDG, the (data) dependence edges specify *partial* orderings on sets of nodes (these can be seen as a set of continuations to be executed *in parallel*), imposing a *total ordering* (specified by a single continuation for each node, forming a chain to be executed *serially*) introduces the possibility that the "next" node in the ordering may not have been related by the partial order. That is (as in the CFG), the next node may not do anything with any value produced by the previous, but could merely perform some

[9] For the final sibling, this is the next sibling of the next highest *ancestral* group node.

[10] Branches have two continuations—one for `true` and one for `false`.

other computation which was placed afterwards arbitrarily or for some other reason. In this case, the two could have been interchanged in the total order, but this would have changed the continuation for each.

2. In both PDG and VSDG, nodes can have *multiple* continuations: in the VSDG, these correspond to program executions in which different subsets of the node's consumers will be evaluated; in the PDG, a node could have one continuation per parent group node (potentially all different, if each group node were to order its children independently). Contrastingly, in the CFG, each statement node stores *exactly one* continuation; hence, storing i distinct continuations for a node, requires i copies of that node in the CFG. A number of the optimization techniques we will see have the effect of reducing the number of distinct continuations identified for each node, and thus the number of copies of the stored procedure that are required.

The Task of PDG Sequentialization is to choose orderings such that nodes have few *distinct* continuations; duplication-freedom captures the condition under which each node has *exactly one* distinct continuation.

2.5 Definition of the PDG

A PDG consists of three kinds of nodes (below), and two kinds of edges: control dependence edges and data dependence edges. The *Control Dependence Graph* (CDG) is the subgraph consisting of all the nodes, and only the control dependence edges; the *Data Dependence Graph* (DDG) is the subgraph of all the nodes and only the data dependence edges. It is convenient to describe control dependence edges as pointing from parents to children, even though the CDG is not strictly treelike in that it may contain cross-links (and back-edges, creating cycles). The number of children depends upon the type of node:

Group Nodes G, drawn as ovals, have multiple children ($n \geq 0$); execution of a group node entails execution of *all* its children, but in any (or an unspecified) order that respects the data dependence edges between them. While group nodes may have labels, these are not significant, being used merely for comments.

Predicate Nodes P, Q, drawn as diamonds, have precisely two children: one for `true` and one for `false`. Execution of a predicate node entails execution of exactly one of its children, according to the runtime value of the predicate being tested, which is indicated by the node's label.

Statement Nodes (or **Leaf Nodes**) S, drawn as rectangles, have no children, but each is labelled with an imperative three-address statement to execute.

Values are passed between statements (and to predicates) by storing them in variables; as in the CFG, SSA form may be used but is not generally a requirement[11]. Data dependence edges are used only to restrict possible orderings among statements (e.g. potentially due to anti-dependences as well as data dependences), and run between siblings of a group node. The DDG is acyclic even in the presence of program loops, but there may be cycles involving DDG as well as CDG edges in the whole PDG.

[11] The PDG predates SSA form.

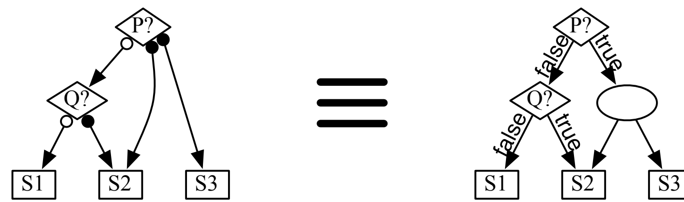


Figure 2.7: Shorthand for Children of Predicate Nodes in PDG

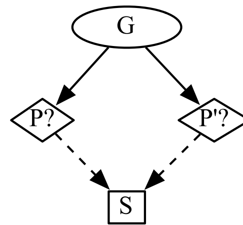


Figure 2.8: Forbidden CDG Subgraph

Note that graphically we indicate the true and false children of predicate nodes with filled and open dots, respectively, and sometimes draw multiple outgoing true or false edges rather than the single group node which must gather them together as a single child. These two shorthand notations are shown in Figure 2.7.

Several well-formedness constraints must be imposed to make a valid PDG. The first of these, which we refer to as *Ferrante-sequentializability*, is particularly important (and was discussed in Section 2.3): we require that the CDG part of the PDG must not contain subgraphs like Figure 2.8.

The remaining well-formedness constraints are easily enforced via transformations; thus, we treat these as being implicitly applied to any PDG:

Redundant Group Nodes Whenever a group node G is a child of a group node G' but has no other parents besides G', it can be removed, and all its children made control dependent directly on G'.

Postdominance of Predicate Nodes Whenever a node N is control dependent on both true and false group nodes of a single predicate P, it is said to *postdominate*[12] P. Ferrante et al. forbid the existence of such nodes [FMS88]. However, any such node can instead be made control dependent on P's parent, as shown in Figure 2.9(a). (If P has multiple parents $G_1, \ldots, G_i$, then a fresh group node can first be inserted as child of all the G_i and as unique parent of P).

Redundant Edges Whenever a node N which is a child of a group node G, is also a child of another group node G' descending from G, the control dependence edge $G' \rightarrow N$ can be removed. (The additional dependency serves no purpose, in that N will already necessarily be executed if G' is). This is shown in Figure 2.9(b).

[12]The term is used loosely as a reference to the position of the resulting node in the CFG *after* sequentialization.

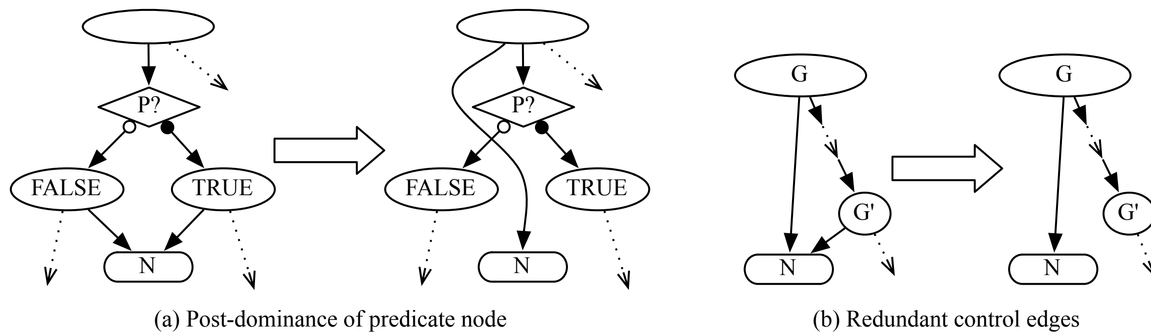


Figure 2.9: Transformations for enforcing or restoring well-formedness of PDGs

2.6 The VSDG: Definition and Properties

The VSDG is a directed labelled hierarchical Petri-net[13] $G = (S, T, E, S_{\text{in}}, S_{\text{out}})$, where:

Transitions T represent operations, covered in Section 2.6.2. These may be *complex*, i.e. containing a distinct graph G'; this hierarchy is explained in Section 2.6.5.

Places S represent the *results* of operations, covered in Section 2.6.1.

Edges $E \subseteq S \times T \cup T \times S$ represent dependencies on and production of results by operations

Arguments $S_{\text{in}} \subseteq S$ indicates those places which, upon entry to the function, contain any values and/or state passed in.

Results $S_{\text{out}} \subseteq S$ similarly indicates the places which contain any values and/or states to be returned from the Petri-net in question.

Note that sometimes we omit the arguments and results, writing simply $G = (S, T, E)$, where these are not relevant.

It is a requirement that $S \cap T = \emptyset$; this makes a petri-net a *bipartite* graph with nodes $S \cup T$ and edges E (thus we write $n \in G$ to indicate $n \in S \cup T$). Besides the restrictions this imposes on edges, other notations of petri-nets will be useful:

$$t \in T \to {}^{\bullet}t = \{s \in S \mid (s, t) \in E\}$$
$$t \in T \to t^{\bullet} = \{s \in S \mid (t, s) \in E\}$$

The first are referred to as the transition's *operands*; the second as the *results* of the transition (this will be clear from context). Note that whilst a place may be operand to many transitions, it is the result of exactly one, called its *producer* (it may *also* be a result of the net).

We now consider the various elements in more detail.

2.6.1 Places, Kinds and Sorts

Every place has a *kind*, either *state* or *value*. (States are drawn with dotted outlines).

A collection of kinds is referred to as a *sort*; specifically:

[13]This is purely a matter of representation; the VSDG's semantics, covered in Section 2.7, are *not* those of a Petri-net.

- A *value sort* is an ordered collection of n values, for some $n > 0$;
- A *state sort* is a single state;
- A *tuple sort* is an ordered collection of n values, for some $n \geq 0$, *plus* a single state. (Thus, a state sort is a tuple sort).

Sorts are often written using abbreviations v, vv, s, v^3s, etc. Thus, we may refer to a transition as having "operands of sort vvs", or "a tuple sort of results".

Two special cases of places are those used to pass values and/or state in and out of the function—the arguments and results. For example, each of the functions `fst`, `sub` and `max` in Figure 2.1 has three arguments, of sort vvs, and two results, of sort vs. Note that a single place may be both argument and result; this is true of the state argument in all three functions, but in `fst` the first argument is also returned.

2.6.2 Transitions

Transitions come in three varieties: *Complex* nodes, which contain further graphs; *Computations*; and γ*-nodes*.

Complex Nodes

These may have any sort of operands and results, and are covered in detail in Section 2.6.5.

Computations

These perform operations taken from some appropriate set of low-level operations. We tend to use RISC-like abstract machine instructions, but within this general theme, the precise set of instructions is not important[14]. Computation nodes are labelled to indicate the operation, and have some sort of inputs and outputs, depending on the operation; we can further classify these nodes into three subcategories:

- Constant nodes have no inputs, and a value sort of outputs (they do not produce state)
- ALU operations have a value sort of inputs and a value sort of outputs
- Memory operations have a tuple sort of inputs, and a tuple sort of outputs. For example, load (inputs: state, address; outputs: state, result) and store (inputs: state, address, value; outputs: state). In this class we include static and virtual calls to other functions.

A simple VSDG using only computation nodes is the `sub` function of Figure 2.1, which has a single computation with two operands and one result. (A similar computation node is in the `max` function).

γ-Nodes

A γ-node g multiplexes between two tuples of inputs according to a predicate. It has results of some sort R, a value operand predicate, and two further sets of operands true and false both of

[14]We do not address questions of *instruction selection*, although there is no reason such optimizations could not be beneficially fitted into the VSDG.

sort R. (Thus, a γ-node with a state result, has *two* state operands; all other nodes have at most one state operand). According to the runtime value of the predicate, either the true operands or the false operands are passed through to the result.

Components It is sometimes helpful to see a γ-node as set $E \subset S^3$ of *components* or triples, where each component $(s^{\mathsf{true}}, s^{\mathsf{false}}, s^{\mathrm{res}})$ gathers together *one* of the node's true operands s^{true} with the corresponding false operand s^{false} and result s^{res}. The predicate input is not considered part of any component, but part of the γ-node itself. We call a path p *componentwise* if for each γ-node g appearing on p, all edges to and from g are incident upon the same component. (Thus, g could be replaced with that component and p would still be a path.)

An example of a VSDG using only computation and γ-nodes is the `max` function of Figure 2.1, with one single-component γ-node (a total of three operands, and one result).

Trees of γ-Nodes When a group of γ-nodes all of the same arity are connected only by using results of one γ-node as true or false operands to another (without permutation), we call the resulting structure a *γ-tree*. This includes where one γ-node is used by several others, forming a DAG. (However there must be no uses of the result of any γ-node from outside the tree, except for the distinguished root node.) Such a γ-tree generalizes the idea of a γ-node by selecting from among potentially many different values or *leaves* (rather than just two) and according to potentially many predicates, but retains the *non-strict* nature of a single γ-node.

2.6.3 Edges

Like places, every edge has a kind, either state or value. The kind of an edge can be inferred from its endpoints: an edge is a state edge—indicated with a dotted line—if and only if its place endpoint is a state.

2.6.4 Labels, Sets and Tuples

A key consideration in the VSDG is that of distinguishing between multiple operands (and/or results) of a transition. Firstly, the *order* of the operands may be important: compare $s_1 - s_2$ with $s_2 - s_1$. Secondly, a single place might be an operand to a transition more than once—e.g. $s + s$.

Strictly, the VSDG handles these cases by *labelling* edges; for any given type of transition, the *set* of labels on its incoming (and outgoing) edges is fixed and particular to that type of transition. (Indeed, the *kind* of an edge can be seen as part of this label.) Thus, s `sub` s could be unambiguously represented by labelling the edges $s \xrightarrow{1} t$ and $s \xrightarrow{2} t$ (where t is the transition for sub).

In this system, $^\bullet t$ is not a set but rather an ordered *tuple* of predecessors, or a map indexed by label (the tuple of i elements can be seen as a map indexed by integers $1 \ldots i$), and similarly for $t^\bullet$, S_{in} and S_{out}. In such cases, we write $s^l \in {}^\bullet t$ to indicate that s is in $^\bullet t$ with label l (that is, $s \xrightarrow{l} t$), and similarly $s^l \in S_{\mathrm{in}}$. Graphically, we may label the edges, as well as the operand and result places of the graph (recall from Section 2.6.1 that in diagrams these can be identified by having no producers and by being drawn with double outlines, respectively).

Two special cases are complex nodes, where the labels depend upon the contained graph (covered in Section 2.6.5), and tupled γ-nodes, where the set of labels depends upon the arity

of the node. Specifically, for a γ-node with components $c_1, \ldots, c_j$ (where $c_i = (s_i^{\mathsf{true}}, s_i^{\mathsf{false}}, s_i^{\mathrm{res}})$ for $1 \leq i \leq j$), we use labels $T_1, \ldots, T_j$ for the edges from the true operands (that is, we have labelled edges $s_1^{\mathsf{true}} \xrightarrow{T_1} g, \ldots, s_j^{\mathsf{true}} \xrightarrow{T_j} g$), labels F_i for the edges from the false operands s_i^{false}, and labels i for the results s_i^{res}. Lastly the edge from the predicate operand is labelled P.

However, in practice this level of detail/precision often confuses the presentation and hampers clarity, and thus generally we leave this labelling implicit, e.g. writing ${}^\bullet t \subseteq S$, and for γ-nodes using labels T, F and P for the operands (only).

In particular, we may "lift" operations over these tuples or maps implicitly, and such lifting should be taken as being performed elementwise and respecting labelling. For example, we may write that for $S_1, S_2 \subseteq S$ some relation $S_1 \mathcal{R} S_2$ holds, where $\mathcal{R} \subseteq S \times S$; strictly, this means that:

1. Both S_1 and S_2 are maps from the same domain (i.e. with *the same set* of keys or labels)—this is usually guaranteed from context.

2. For each label l, the *corresponding* elements of S_1 and S_2 are related by $\mathcal{R}$, that is:

$$\forall l. S_1[l] \mathcal{R} S_2[l]$$

   or alternatively $S_1 \ni s_1^l \mathcal{R} s_2^l \in S_2$.

2.6.5 Hierarchical Petri-Nets

We have already mentioned that *complex nodes* $t \in T$ may contain petri-nets. This allows individual petri-nets—called *net instances* or just *nets*—to be arranged into a *hierarchy* to form a VSDG. Given such a complex node t, we write $t\langle G' \rangle \in G$ and say G' is the *contained* net in its *container* transition t which is a member of *parent* net G.

As usual, the contained net may have arguments S'_{in} and results S'_{out}, and these may be of any sort. The container transition must have operands of sort *at least* including the sort of S'_{in}, and results of sort *at most* the sort of S'_{out}. The correspondence between these is recorded using the system of labels detailed in Section 2.6.4: specifically, the arguments S'_{in} of the contained net are actually a map, keyed by a set of labels L_{in}, and each $l \in L_{\mathrm{in}}$ must also appear on exactly one edge ${}^\bullet t \to t$. Dually, let L_{out} be the set of labels on edges from t to its results $t^\bullet$; the results S'_{out} must contain an element for each $l \in L_{\mathrm{out}}$.

Flattening Merges a complex node $t\langle G' \rangle \in G$ to make a new net incorporating the bodies of both. Informally, flattening proceeds by taking the disjoint union of the two graphs, removing the container node t, and identifying the operands ${}^\bullet t$ with the arguments of the previously-contained net and similarly for the results.

More formally, let the *quotient* operation on a graph $G = (V, E)$ combine two vertices into a fresh vertex x, as follows:

$$G/(u,v) = \left(\begin{array}{c} (V \backslash \{u,v\}) \cup \{x\}, \\ \left(\begin{array}{l} \{(v_1,v_2) \in E \mid \{v_1,v_2\} \cap \{u,v\} = \emptyset\} \cup \\ \{(v',x) \mid (v',u) \in E\} \cup \\ \{(v',x) \mid (v',v) \in E\} \cup \\ \{(x,v') \mid (u,v') \in E\} \cup \\ \{(x,v') \mid (v,v') \in E\} \end{array} \right) \end{array} \right)$$

This is lifted over sets of pairs straightforwardly:

$$\begin{array}{rcl} G/\emptyset & = & G \\ G/(\{(a,b)\} \cup P) & = & (G/(a,b))/P \end{array}$$

The same principle applies to petri-nets, in which both places and transitions may be quotiented (with other places, or respectively transitions, only). For VSDGs, which contain arguments S_{in} and results S_{out} in addition to nodes and edges, the same substitution of $[^{x}/_{u}]\,[^{x}/_{v}]$ is also applied to S_{in} and S_{out}.

The result of flattening $t\langle G'\rangle \in G$ can now be obtained by the disjoint union $G \uplus G'$ followed by quotienting the operands and results (for precision, here we deal with the labels explicitly):

$$\begin{array}{l} \mathit{flatten}(t\langle (S', T', E', S'_{\text{in}}, S'_{\text{out}}\rangle \in (S, T, E, S_{\text{in}}, S_{\text{out}}) = \\ \left(\begin{array}{c} (\{1\} \times S') \cup (\{2\} \times S), \\ (\{1\} \times T') \cup (\{2\} \times (T \backslash \{t\})), \\ (\{((1,s') \xrightarrow{l} (1,t')) \mid (s' \xrightarrow{l} t') \in E'\} \cup \\ \{((1,t') \xrightarrow{l} (1,s')) \mid (t' \xrightarrow{l} s') \in E'\} \cup \\ \{((2,s') \xrightarrow{l} (2,t')) \mid (s' \xrightarrow{l} t') \in E\} \cup \\ \{((2,t') \xrightarrow{l} (2,s')) \mid (t' \xrightarrow{l} s') \in E\}), \\ \{2\} \times S_{\text{in}}, \\ \{2\} \times S_{\text{out}} \end{array}\right) / \left\{ \begin{array}{l|l} ((1,s), & (s^{l} \in S'_{\text{in}} \wedge (s' \xrightarrow{l} t) \in E) \vee \\ (2,s')) & (s^{l} \in S'_{\text{out}} \wedge (t \xrightarrow{l} s') \in E) \end{array} \right\} \end{array}$$

Observe that the restrictions on the edge and place labels above ensure that after flattening, every place has a unique producer transition, but places may still have any number $i \geq 0$ of consumers.

2.6.6 Loops

Loops are represented as *infinite* VSDGs. Although we write such structures using the μ fixpoint operator (they are regular), this is merely a syntactic representation of an infinite graph or hierarchy. By convention, we tend to μ-bind variables to net instances contained in complex nodes, thus: $t\langle \mu X.G[X]\rangle$. However, this is not essential, as the *flattening* operation may be applied (infinitely many times) to such hierarchies, yielding a single net with an infinite set of places and transitions. (This is used in Section 2.7.2 to define a semantics of loops.)

Loops generally both consume and produce state on every iteration; however loops which perform no stateful operations and are known to always terminate, may be optimized to neither consume nor produce state.

2.6.7 Well-Formedness Requirements

For a VSDG to be well-formed, we make a number of requirements, as follows (some of these have already been mentioned above).

Acyclicity There must be no (graph-theoretic) cycles in the VSDG (loops must be represented using infinite VSDGs).

States and Values Any edge incident upon a state place must be a state edge; all other edges must be value edges.

Node Arity Every place must have a unique producer (i.e. a single incoming edge $\in T \times S$), with the exception of argument places S_{in}, which must have no incoming edges. Transitions must have incoming and outgoing edges appropriate to their type.

Hierarchy For a compound node $t\langle(S', T', E', S'_{\text{in}}, S'_{\text{out}})\rangle \in (S, T, E)$, we require the *sets of labels* to be consistent, thus: $\{l \mid s^l \in S'_{\text{in}}\} \subseteq \{l \mid (s \xrightarrow{l} t) \in E\} \wedge \{l \mid s^l \in S'_{\text{out}}\} \supseteq \{l \mid (t \xrightarrow{l} s) \in E\}$

States must be used linearly Every state dynamically produced must be consumed exactly once. The exact requirements on the shape of the graph that ensure this are complex, and given in Appendix B. For now recall other properties of state specified already:

- State γ-nodes have exactly two state operands, and one state output.
- Other transitions *either* have exactly one state operand and one state result (*stateful* transitions), *or* have no state operands or results.

2.7 Semantics

2.7.1 Sequentialization by Semantic Refinement

In this thesis, two kinds of semantics are relevant.

The *observable semantics* of a program describes the action of the program in terms of a function from its inputs to its outputs (including side effects). These *must be preserved exactly*, from source code specification to machine code output, by the compiler. In some languages—according to the language specification—such semantics are sometimes considered partially nondeterministic; for example, in C

```
int x=0;
printf("%i",(x++)+(x++));
```

may produce output `0` or `1`, *according to what the compiler decides is easiest*. However, we assume that in such cases a specific choice has been made by the front-end, and that the observable semantics of a given program when represented in all IRs are fully deterministic (explained below).

However, we are also concerned with the *trace semantics* of the program in a particular IR. These are a “small-step” semantics specifying the sequence or trace of operations by which the resulting program will compute its result, indexed by the natural numbers $\mathbb{N}$. The *observable semantics* of the program, for any particular input values, are the homomorphic image of a trace for those inputs counting only observable actions.

Moreover, the various IRs we consider—VSDG, PDG, CFG—include a *degree* of nondeterminism, such that their trace semantics specify a *set* of possible traces, but the observable image is the same for each. (For example, the order in which $+$ and $-$ are performed in evaluation of $(a + b) * (c - d)$.) Our architecture for sequentialization thus works by a process of *semantic refinement* in which the range of nondeterminism is progressively *narrowed*. Each phase reduces the set of possible traces, beginning from a large set (the VSDG) to a smaller subset (the PDG) and finally to a single element (the CFG, which can be considered deterministic), with the observable image remaining identical throughout.

Optimizations performed on the IR prior to sequentialization (as shown in Figure 1.1) are allowed to change the program's trace semantics, so long as the observable semantics are preserved.

2.7.2 The VSDG as a Reduction System

We define the observable semantics of a VSDG by treating it as a reduction system (reduction systems were discussed in Section 2.2.3) under interpretation.

Specifically, let $\mathcal{V}$ be some set of values including at least true and false, and $i(\mathtt{op}) : \mathcal{V}^* \to \mathcal{V}$ be an interpretation of the arithmetic operations[15].

We allow places S to be *marked* with elements taken from $\mathcal{V}$. A transition is a *potential redex* iff all its operand places are marked and all of its result places are unmarked, except for γ-nodes, where some operands do not need markings. A potential redex is a *redex* if there is a path from it to one of the VSDG's result places that does not go through an already-reduced γ-node[16]. Each reduction step corresponds to "firing" a transition, placing a marking on each result place (and unlike normal Petri-nets, leaving operand markings untouched) as follows:

$$\begin{array}{rcl} \mathtt{op}(v_1 \in \mathcal{V}, \ldots, v_n \in \mathcal{V}) & \mapsto & i(\mathtt{op})(v_1, \ldots, v_n) \in \mathcal{V} \\ \gamma(\mathsf{true}, v \in \mathcal{V}, \circ) & \mapsto & v \\ \gamma(\mathsf{false}, \circ, v \in \mathcal{V}) & \mapsto & v \end{array}$$

Note that $\circ$ here indicates a place which may have any marking or be unmarked. For tuples of γ-nodes, the last two rules apply, but *all components of a tuple must fire together* (the tuple is only a redex if *all* components have appropriate markings on their operands).

Finite Nets For a finite VSDG G with no I/O operations, we define the observable semantics under such an interpretation as the value or tuple $(\!|G|\!) \in \mathcal{V}^*$ given by the uniquely-defined marking(s) eventually placed onto the result place(s) of G.

The semantics of a hierarchical net with complex nodes are that of its flattening, thus the division into hierarchy is not significant and serves only as a tool for structuring.

Loops and Infinite Nets For infinite nets, we use standard domain theoretic techniques of successive approximation, as follows. Here, we must insist on a *fairness* criterion:

> For any transition t which becomes a redex at some point in a sequence, there is some future point in that sequence where *either*
>
> - The redex is reduced, *or*
> - The transition becomes no longer a redex.

The latter may occur if subsequent γ-node reductions mean the transition's result will not be needed. Specifically, fairness means any arbitrary amount of speculative evaluation (e.g. of nodes in future loop iterations) is permitted, but only *finitely*; thus, such speculation may not introduce non-termination.

[15] Technically, $i(\mathtt{op}) : \mathcal{V}^* \to \mathcal{V}^*$ is more accurate, as $\|t^\bullet\| \geq 1$.

[16] Thus, transitions whose results have already been determined to be unneeded, are not redexes.

Now, let $\mathcal{D}$ be the domain consisting of values V augmented with additional element $\bot$ satisfying $\forall x.\bot \sqsubseteq x$, let $\sqcup$ defined as

$$x \sqcup y = \begin{cases} x, & \text{if } y = \bot \\ y, & \text{if } x = \bot \\ x, & \text{if } x = y \end{cases}$$

Now, for a finite VSDG G with no I/O, we write $[\![G]\!]$ for the marking placed onto its result place(s) under interpretation with values $\mathcal{D}$ and reduction rules augmented with:

$$\begin{array}{rcl} \forall \mathtt{op}.\mathtt{op}(\ldots, \bot, \ldots) & \mapsto & \bot \\ \gamma(\bot, \circ, \circ) & \mapsto & \bot \end{array}$$

The semantics of an infinite graph G are now given by lifting the above $[\![\cdot]\!]$ over successive non-hierarchical approximations:

$$[\![G]\!] = \bigsqcup_{n \in \mathbb{N}} [\![G^n]\!]$$

Where G^n is obtained from G by:

1. Flattening all the complex nodes in G, n times
2. Marking every place which is the still the result of a complex node with $\bot$
3. Marking every place with $minDFR(s) > i$ with $\bot$, where $minDFR$ is the minimum depth-from-root.

Note that this is consistent with the translation into a functional program given above in Section 2.2.2, and that for finite nets, $[\![G]\!] = (\!|G|\!)$

2.7.3 A Trace Semantics of the VSDG

In Section 2.7.1 we stated that the *trace semantics* of a VSDG were a large set. Specifically, its nondeterminism ranges over all possible evaluation behaviours (strategies) and orders, including where a given node may be evaluated more than once[17], and where a single computation occurs for *multiple* transitions if they are *equivalent*. (Transitions are equivalent if they compute the same function on equivalent inputs.)

Hash-Consing The *hash-consing* transformation and its inverse of *node cloning* convert between such nets, as shown in Figure 2.10:

- *Hash-consing* takes transition nodes t and t' that perform the same operation on identical operands (that is, $^\bullet t = {}^\bullet t'$—their operand places are *the same nodes*), and changes all consumers of results of t' to use results of t instead. That is, all edges from $s' \in t'^\bullet$ are redirected to come from the corresponding $s \in t^\bullet$ instead. (Since t' and t performed the same operation, they have the same sort of results, as defined in Section 2.6.1.) t' is then dead (unreachable in the transpose) and may be deleted; thus, hash-consing yields a smaller net.

[17] Except for stateful nodes, for which the number of evaluations is fixed and the operations discussed here do not generally apply.

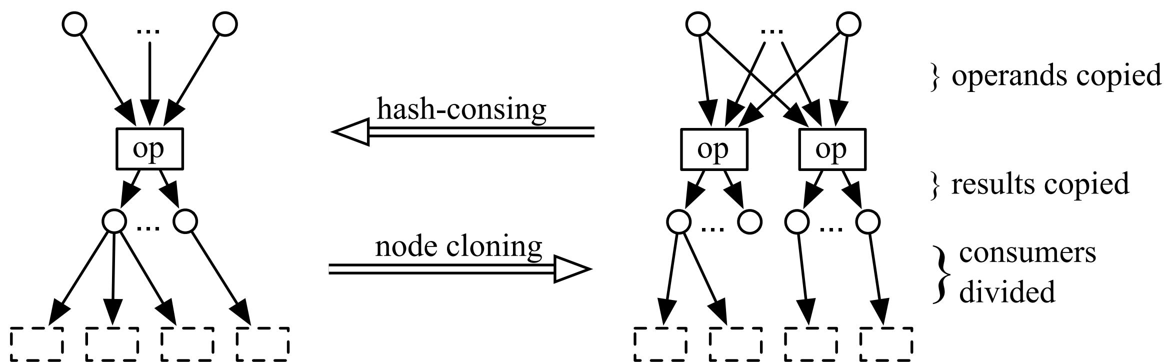


Figure 2.10: The hash-consing and node cloning transformations

- *Node cloning* thus copies a computation node t with multiple uses (edges $\subseteq t^\bullet \times T$) to make an additional transition node t' and result places $t'^\bullet$, redirects some of the edges outgoing from $s \in t^\bullet$ to come from the corresponding $s' \in t'^\bullet$.

For tupled γ-nodes, discussed in Section 2.6.2, hash-consing also includes combining or separating equivalent components within a single γ-node in analogous fashion.

Full Laziness The equivalent to hash-consing in functional programming languages is *full laziness* [Wad71]: a transformation replacing multiple textually identical expressions with uses of a single extra variable. In *fully lazy lambda lifting*, the variable is defined by a function abstraction; in *let-floating*, by a `let`-expression. (As noted in Section 2.2.4, the variables bound by `let`-expressions implicitly store computations, for which the usual range of evaluation strategies may be used, including call-by-name.)

Thus, let $h(G)$ be the *set* of VSDGs which can be obtained by applying any number of hash-consing, node-cloning, and/or flattening operations to a VSDG G (flattening operations expose additional opportunities for hash-consing and node-cloning). The *trace semantics* of G are the set of possible sequences of reduction steps[18] by which any VSDG $G' \in h(G)$ can be reduced to its observable result $[\![G']\!] = [\![G]\!]$.

Thus, for example, the VSDGs (flattened and hierarchical) of Figure 2.11 have *the same semantics* (trace and observable) as those of Figure 2.2, despite differences in their arrangement into a hierarchy and the number of nodes evaluating $2 * a$.

A particular consequence is that arbitrary amounts of hash-consing are permitted *across loop iterations*, and so VSDGs which conceptually encode different levels of software pipelining in fact have the same semantics—as shown in Figure 2.7.3. Thus, a loop represented in the VSDG does not have a level of software pipelining specified, and we can see the VSDG as normalizing between different pipelinings in the source code. The selection of an appropriate level of software pipelining is part of sequentialization, and we revisit this issue in Chapters 5 and 6.

[18]The correspondence between reduction steps and program operations will remain informal in this thesis.

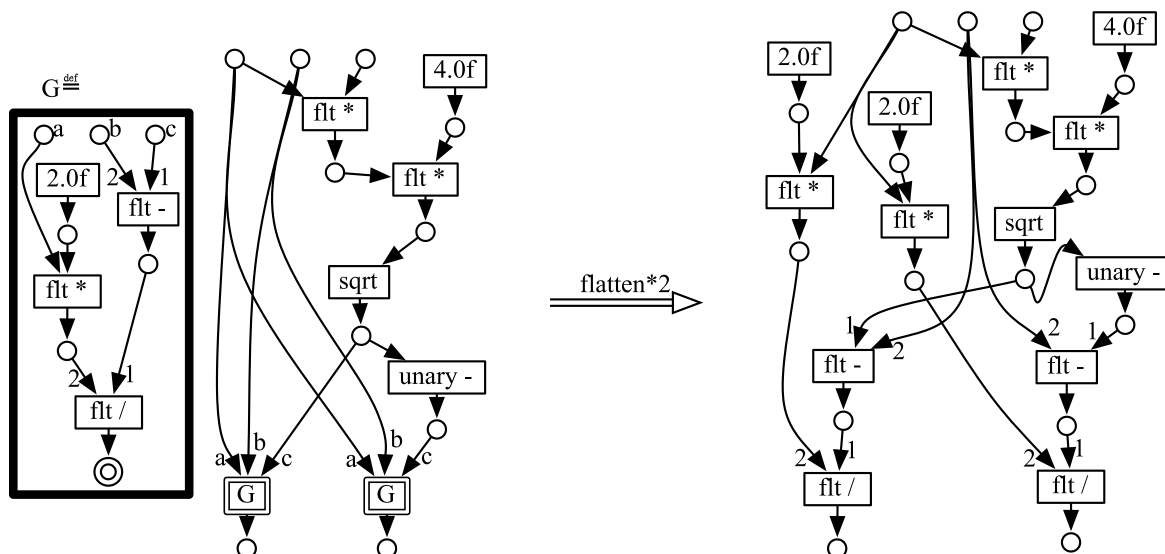
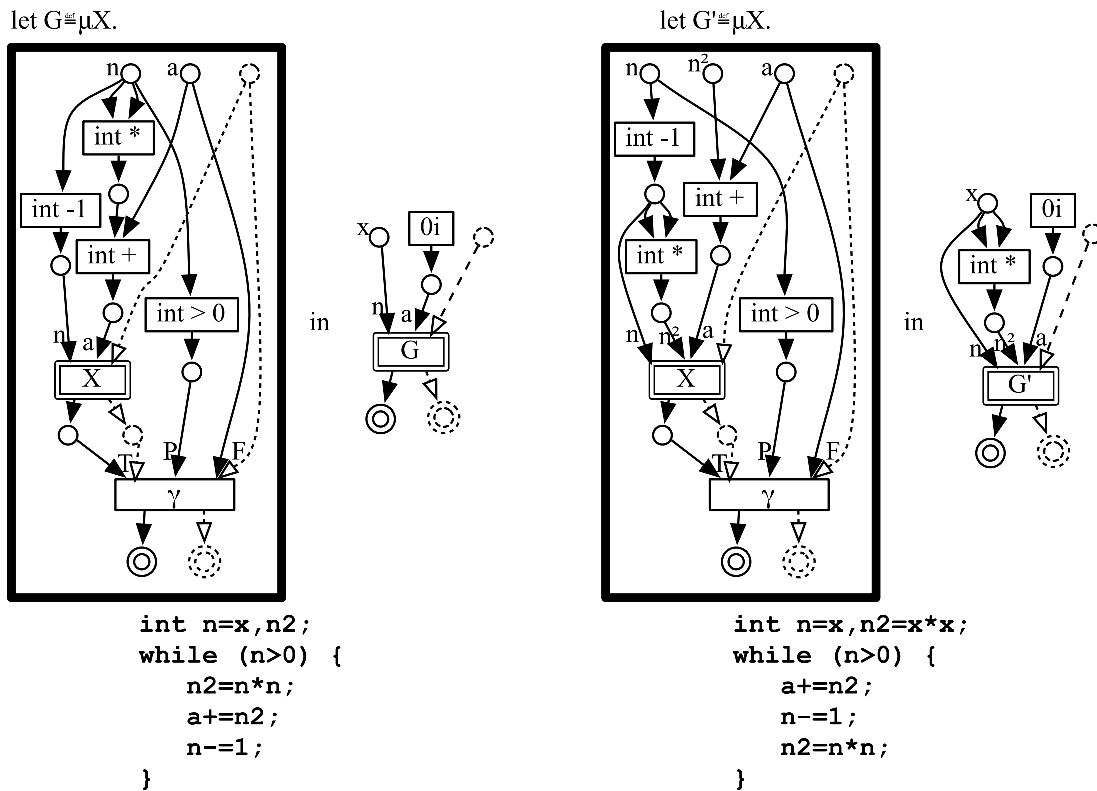


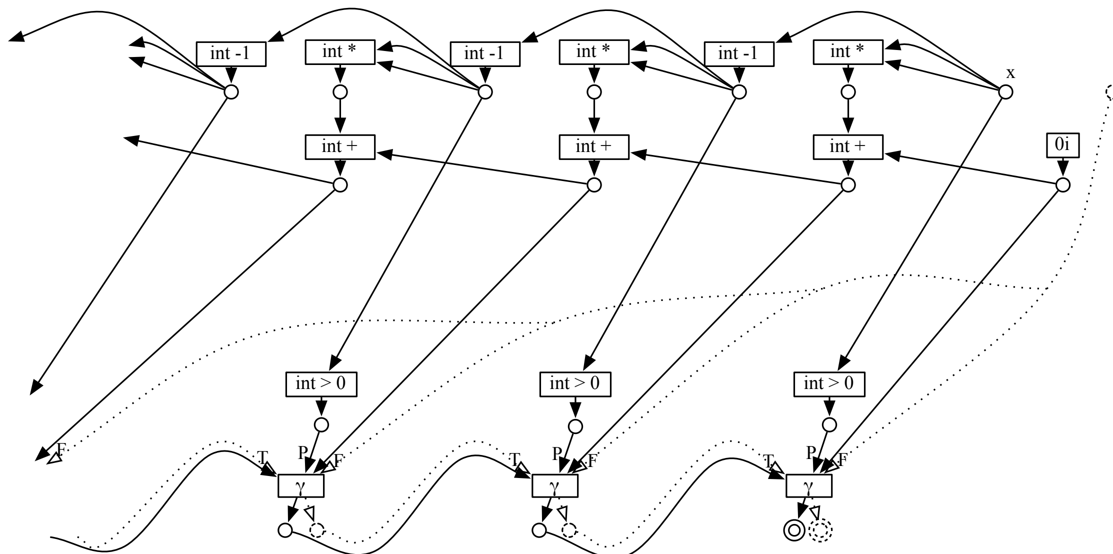
Figure 2.11: Another VSDG, semantically equivalent to Figure 2.2.

2.8 Chapter Summary

We have seen how a VSDG represents a program in the manner of a *lazy functional* programming language. Secondly, we have presented the compiler architecture which will be used in the rest of this thesis, consisting of stages of *proceduralization* (VSDG→PDG conversion), *PDG sequentialization* (enforcing the *duplication freedom* condition on PDGs) and *Node Scheduling* (df-PDG→CFG conversion). These are shown diagrammatically on page 33.



(a) Two *conceptual* software pipelinings of the same loop



(b) The infinite unrolling common to both (after flattening)

CHAPTER 3

Proceduralization

Proceduralization is the conversion from VSDG to PDG representations, including the construction of a CDG encoding an evaluation strategy, and the conversion of functional nodes to imperative statements. In this chapter, we will:

- Discuss issues relating to the choice of evaluation strategy (Section 2.2.3), resulting in choosing the lazy strategy, in Section 3.1.
- Show how a naive translation from VSDGs to PDGs, given in Section 3.2.1, is flawed with respect to both efficiency and correctness; and give an intuition as to how this can be improved using *tail nodes*, in Section 3.2.2.
- Describe in Section 3.3 an algorithmic framework for VSDG→PDG conversion, and in Section 3.4 explain the two additional operations performed by our algorithm which are key to the efficiency of the resulting PDG:
  1. Ordering parallel γ-nodes into a tree such that the branches precisely identify the executions under which each node definitely is, or definitely isn't, required. The γ-ordering transformation which achieves this is given in Section 3.4.1.
  2. Using tail nodes to reduce duplication in PDG sequentialization, paralleling the use of *cross-jumping* or *shortcircuit evaluation* on CFGs. This is achieved by γ-*coalescing*, described in Section 3.4.2.

Running Examples Throughout this chapter, illustration will be by two running examples of VSDGs which (informally) are difficult to proceduralize well and have proved challenging for previous techniques (this is discussed more thoroughly in Section 4.3). These are given in Figures 3.1 and 3.2, each with two different CFG representations of the same function and corresponding source codes; we refer to them as *exclusive* and *independent* redundancy, respectively.

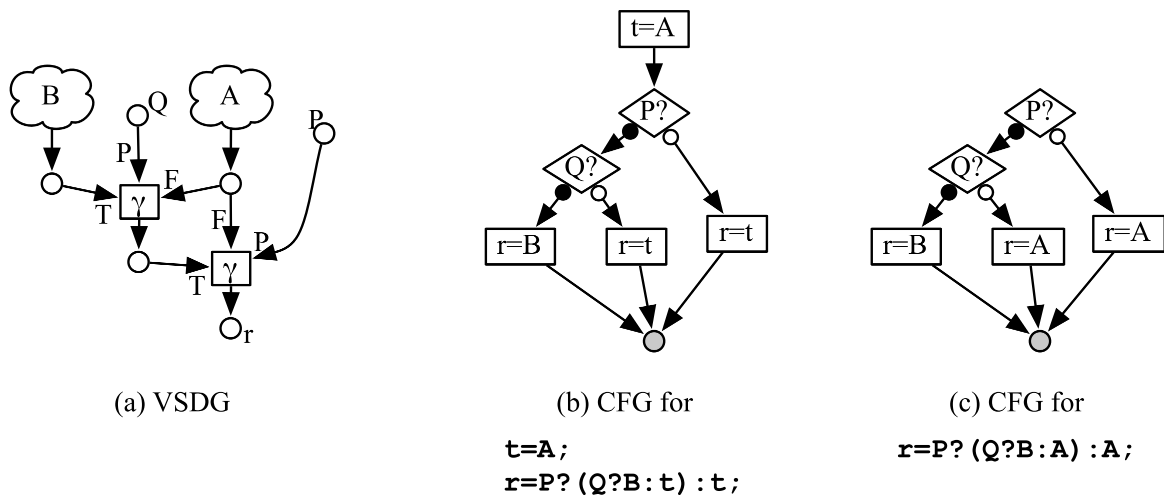


Figure 3.1: Two programs—(b) speculating, and (c) duplicating, computation of A—informally known as *exclusive* redundancy

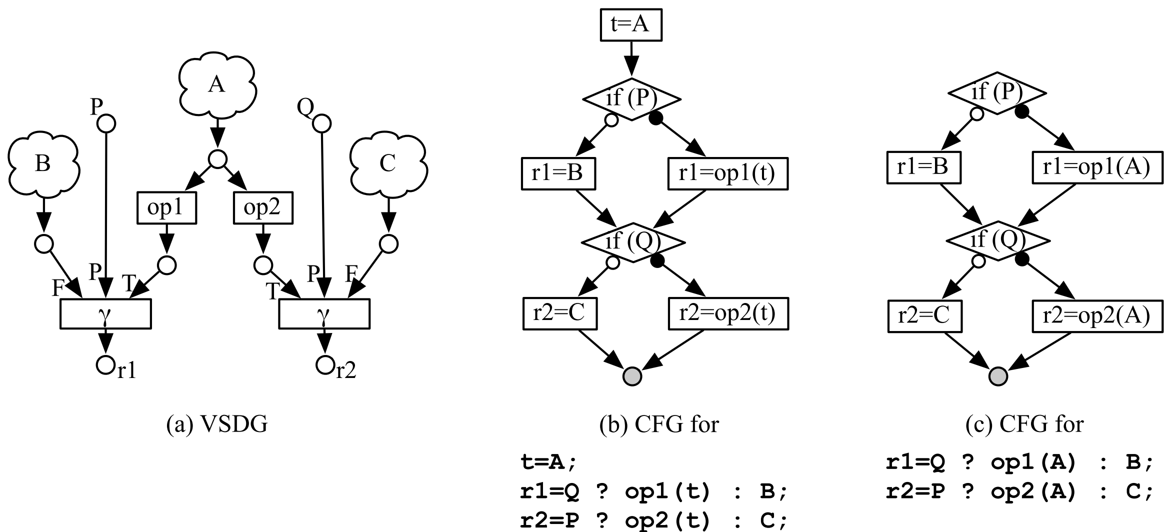


Figure 3.2: Two programs—(b) speculating, and (c) duplicating, computation of A—informally known as *independent* redundancy

Kind of redundancy	Call-by-need (optimal)	Call-by-value (speculative)	Call-by-name (duplicating)
Exclusive (Fig. 3.1)	0–1	1	0–1
Independent (Fig. 3.2)	0–1	1	0–n

Table 3.1: Evaluation strategies and their resulting number of evaluations

3.1 Choosing an Evaluation Strategy

In Section 2.2.3 we explained how the evaluation strategies familiar from the λ-calculus also apply to the VSDG, and how the selection of evaluation strategy does not affect the end result[1]. Further, part of the task of proceduralization is to construct a CDG, which selects and encodes a particular evaluation strategy into the resultant PDG. The question is, which strategy should we use?

The key consideration is the number of evaluations of any node that might result from our choice. It turns out that the two running examples of Figure 3.1 and Figure 3.2 identify two situations where choice of evaluation strategy is particularly important, so we summarize for each example how many times the node A might be evaluated under each strategy in Table 3.1.

The desirable number of evaluations is captured by Upton [Upt06] in his definition of the optimality of a sequentialization, as follows:

> A sequentialization is *optimal* if it:
>
> 1. Is *dynamically optimal*—i.e. performs no redundant computation on any path
> 2. Is *statically optimal*—i.e. has the smallest size *(number of nodes)*—amongst all dynamically optimal sequentializations.

Upton applied his definition to the entire sequentialization process; however, we will see that PDG Sequentialization affects only static optimality, and Upton's equivalent of our node scheduling was merely to order nodes in an arbitrary topological sort, which affects neither criterion. Thus, we consider the definition of dynamic optimality to apply to proceduralization specifically. Upton proves that finding an optimal sequential VSDG is NP-complete[2]; hence, in our work we will take the syntactic approximation that γ-nodes branch independently (their predicates are opaque), even when their predicate inputs are the same. Improvements to this approach are explored in Chapter 7, but this simplifying assumption is commonly used in strictness analysis [Myc80] and many CFG (dataflow) analyses such as code motion [KRS94a].

Upton's criteria strongly suggests that we should implement the lazy strategy. A further consideration is that it is the only strategy that is *safe* in the presence of state edges and stateful nodes (where call-by-name can lead to the state being mutated more than once, and call-by-value can both lead to the same state being consumed more than once, and also prevent proper termination of loops).

[1] Modulo changes to termination behaviour, and/or side-effects.

[2] The proof proceeds by constructing a tree of γ-nodes which demands the value of the vertex v only if an arbitrary boolean expression holds at runtime; thus, whether a single copy of v should be computed in advance, or a separate copy of v positioned at each usage site, depends upon whether the expression is a tautology or not, a Co-NP-complete problem. In our framework the essence of the proof stands, but the details would have to be modified (to perform a different function on v at each usage site) to avoid an optimal shortcircuit proceduralization which is simply produced by our algorithm. Since Upton's (exponential-time) algorithm does not produce such sequentializations, it is optimal only for VSDGs corresponding to certain classes of structured program!

As noted in Section 2.2.3, a second consideration is the dynamic overhead required to implementing the evaluation strategy, which tends to be greater for lazy strategies (and indeed, can outweigh the reduction in computation). We will use the policy that, *dynamically*, one test-and-branch will be performed per γ-node in the VSDG[3]. As noted in Section 2.2.4, the lazy strategy can be implemented by inlining into each usage site a copy of the thunk with a check for previous evaluation; however this produces many extra tests and branches, and so we will find an arrangement whereby *every* check can have its result determined *statically* and thus be eliminated. (Clearly this is possible, e.g. if the output CFG were to contain no control-flow merges. However that would involve an exponential size increase, even for straight-line code, and infinite for loops(!); we will see how we can do better.)

3.2 Foundations of Translation

All algorithms to translate the VSDG into a PDG have various common aspects. First, each *value* (as opposed to state) place in the VSDG is associated with a temporary variable in the PDG. It is convenient to make this a *virtual register* suitable for later colouring by a register allocator, and as such generally each place will have a distinct register.

A VSDG operation node, for example $+$, will be translated into a PDG atomic statement node containing (e.g.) $\mathsf{r}_1 = \mathsf{r}_2 + \mathsf{r}_3$ where r_1 is the temporary for its result place and r_2 and r_3 are those for its operands. We use the convention that $\overrightarrow{rt}$ indicates the register(s) containing the result(s) of transition t.

A γ-node g with predicate operand in temporary r_p is translated into a PDG predicate node testing r_p and two group nodes (one for true and one for false), and statement nodes $\overrightarrow{rg} = \overrightarrow{r_\mathsf{t}}$ and $\overrightarrow{rg} = \overrightarrow{r_\mathsf{f}}$ assigning to the result of the γ-node the temporaries for the true and false operands, respectively.

3.2.1 Naïve Algorithm

It remains to organize the statement nodes, and the predicate nodes, under the various group nodes, i.e. arranging them into some sort of hierarchy. The basis of this construction is that, if a statement S is control dependent on a group node G, then nodes producing the values used by S should also be control dependent on G (so they will be evaluated if S is). Thus we begin by making all VSDG nodes reachable from the return node *without* going along any true/false edges of γ-nodes, be control-dependent on the PDG's root node (a group node). Then, for each γ-node g, the statements $\overrightarrow{rg} = \overrightarrow{r_\mathsf{t}}$ and $\overrightarrow{rg} = \overrightarrow{r_\mathsf{f}}$ are made control-dependent on the true and false children, respectively, of the corresponding PDG predicate node, along with any other nodes reachable without going along any further true or false edges[4], and so on.

Dependency edges (both state and value) between VSDG nodes are then copied across to the PDG, where they point from statement nodes consuming values to the nodes producing them; their endpoints are then moved up the CDG hierarchy until they are between siblings.

However, these policies lead to several problems:

1. **Code Size**—the resulting PDG sequentializes only to an unnecessarily large CFG. For

[3] However, these may not correspond exactly to the tests and branches in the source code due to the possibility of different policies for γ-node tupling in VSDG construction.

[4] Recalling the normalizations of Section 2.5—any nodes already control-dependent on some ancestral group node need not be considered further.

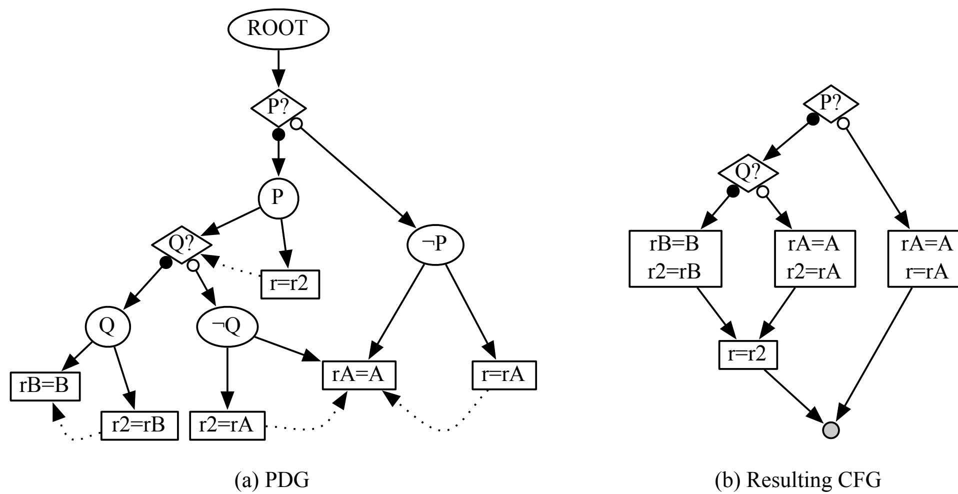


Figure 3.3: Naive translation of the VSDG of Figure 3.1

example, consider the running example VSDG of Figure 3.1. The naïve approach results in the PDG of Figure 3.3(a) (note that the control dependences of A cannot be combined into a single dependence on the root node, as the value of A is *not* always required). As we will see in Section 4.1.1, this PDG is not duplication-free; specifically, PDG sequentialization must duplicate $\mathsf{rA} = \mathsf{A}$, producing the CFG of Figure 3.3(b).

2. **Correctness** Consider again the example of Figure 3.2. The naïve policy includes control dependences for $\mathsf{rA} = \mathsf{A}$ on the true group nodes resulting from *both* γ-nodes. However, this PDG (Figure 3.4(a)) is not well-formed (specifically, it is not Ferrante-sequentializable, as defined in Section 2.5).

3. **Execution Speed** It might seem that an “obvious” fix for the previous problem is to produce two copies of the problem statement, one for each predecessor, i.e. Figure 3.4(b). However, this PDG admits executions (specifically, if both P and Q hold) in which more than one statement computing the value of A is executed. Such repeated computation is clearly redundant by Upton’s criteria (Section 3.1) and should be eliminated[5].

3.2.2 Tail Nodes—an Intuition

An intuition for an improved system comes from the widespread use of *shortcircuit evaluation* of boolean predicates (we can see the *cross-jumping* optimization on CFGs as opportunistically applying the same transformation in other cases).

It closely parallels the *tail call* optimization on functional programming languages, whereby the callee is passed the same return address as was passed to the caller (rather than pushing a new frame onto the stack in order to return into the middle of the caller), so the callee itself completes execution of the caller. The approach is to take any transition t whose results are *only* used (i.e. consumed) as true or false operands to a γ-node g. By allocating t the same result register(s) as g, the move instruction(s) $\overrightarrow{rg} = \overrightarrow{rt}$ resulting from that edge can be elided

[5]If A has side effects it may change the observable semantics of the program.

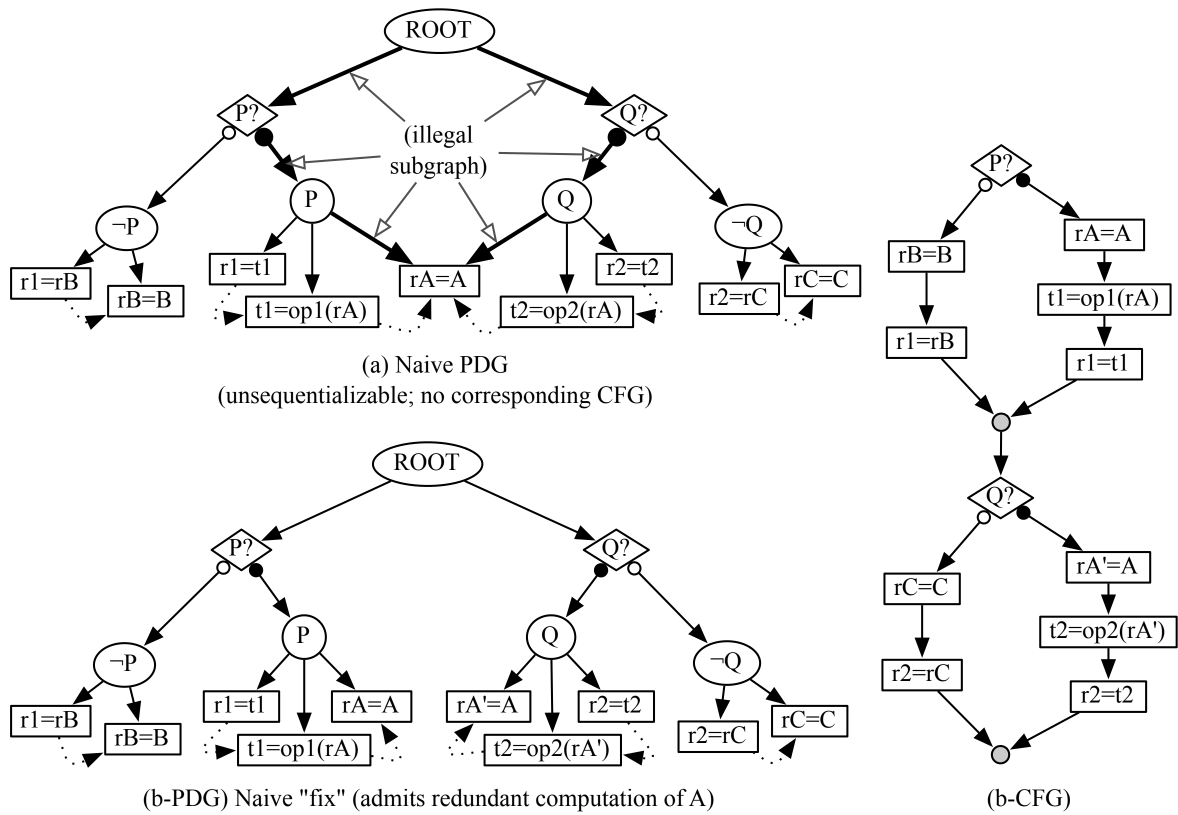
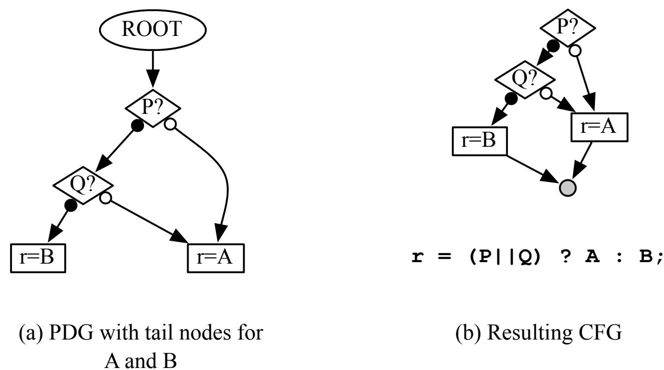


Figure 3.4: Two naive translations of the VSDG of Figure 3.2. ((b-CFG) is identical to CFG Figure 3.2(c) modulo register coalescing.)



(a) PDG with tail nodes for A and B

(b) Resulting CFG

Figure 3.5: "Tail node" register assignment

(as register assignment techniques might later try to do). This means the continuation of t's result is now the same as the continuation of g's result.

When t is another γ-node, this technique generalizes to entire γ-trees (defined in Section 2.6.2), forcing the same registers to be allocated to all γ-nodes in the tree. Thus, the continuation (future uses) of every leaf is the same as that of the root of the tree.

Applying this to the VSDG of Figure 3.1, we first take t as the upper γ-node, being returned through g the lower γ-node. This identifies both γ-nodes as being a γ-tree, sharing the same result register. We can then take t as node A, both of whose consumers are now all part of that tree, and then as node B, resulting in the PDG of Figure 3.5; observe that the shared PDG node r1 = A—called a *tail node*[6]—is not duplicated by PDG sequentialization.

3.2.3 Normalization of Conditional Predicates

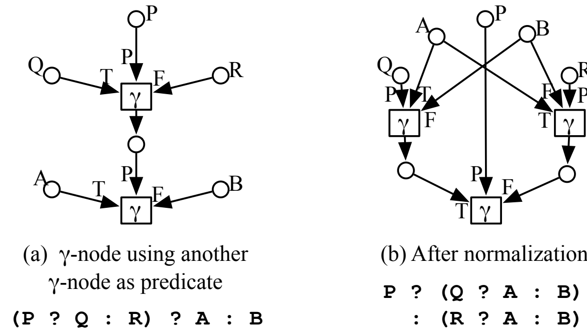
Another situation where naive treatment produces rather inefficient statements is where the predicate operand to a γ-node is the result of another γ-node, as shown in Figure 3.6(a), Instead, such γ-nodes can be rewritten, as shown in Figure 3.6(b); this is dynamically better, and will remain statically small after PDG Sequentialization due to sharing of A and B.

3.3 An Algorithmic Framework

In this section we present our framework for VSDG→PDG conversion. Central is the procedure `buildPDG`, which works by a recursive, post-order traversal of the *postdominator* tree of the VSDG's transitions, using a system of *gating conditions* computed during traversal to combine together the results of the recursive calls. Importantly, this algorithm supports additional operations (in Section 3.4) which allow the flaws of the naïve translation (discussed in Section 3.2.1) to be avoided.

First, Section 3.3.1 expounds the postdominator tree upon which the algorithm operates, together with some of its properties. Then, gating conditions are defined and explained in Section 3.3.2. Lastly, Section 3.3.3 explains the `buildPDG` procedure itself.

[6]By abuse of terminology we use the same term to refer to the VSDG node from which the PDG node or subtree was produced.



(a) γ-node using another γ-node as predicate

`(P ? Q : R) ? A : B`

(b) After normalization

`P ? (Q ? A : B) : (R ? A : B)`

Figure 3.6: Transforming conditional predicates

For the purposes of this section, the places in the VSDG are not useful: a value or state s can only be obtained by executing the unique $t = {}^\bullet s$, producing all $t^\bullet$ together. Hence, while we continue to use the notation ${}^{\circ\bullet}t$ for the transitions producing a result used as operand by t, we otherwise effectively ignore places s.

3.3.1 Dominance and Dominator Trees

Dominators are a standard concept of flowgraphs [AU77, LT79]. (A flowgraph $G = (N, E, n_0, n_\infty)$ is a graph with nodes N, edges $E \subseteq N \times N$, and distinguished *entry node* $n_0 \in N$ and *exit node* $n_\infty \in N$.) Briefly, node n dominates n', written $n \ \mathtt{dom}^* \ n'$, if all paths from the entry node n_0 to n' pass through n. Dominance is reflexive, transitive and antisymmetric, and hence constitutes a partial order. Node n *strictly dominates* n', written $n \ \mathtt{dom}^+ \ n'$, if $n \ \mathtt{dom}^* \ n'$ and $n \neq n'$. This allows the definition of the *immediate dominator* $\mathsf{idom}(n)$ of n, as the unique node n' satisfying

$$n' \ \mathtt{dom}^+ \ n \wedge \forall n'' \ \mathtt{dom}^+ \ n . n'' \ \mathtt{dom}^* \ n'$$

The *dominator tree* of a flowgraph $G = (N, E, n_0 \in N)$ is the graph with nodes N and an edge $n \rightarrow n'$ whenever $n = \mathsf{idom}(n')$. This graph is treelike, with n_0 being the root. Dominator trees can be computed in linear [Har85] or quasi-linear time [LT79].

Postdominance is the same concept but with the edges in G reversed: that is, n postdominates n', written $n \ \mathtt{pdom}^* \ n'$, if all paths from n' to the exit node n_∞ pass through n. The same ideas of strict postdominance, immediate postdominator $\mathsf{ipdom}(n)$, and postdominator tree apply.

We adapt these concepts to the VSDG, and to the notation of Petri-Nets, as follows. Firstly, we hypothesize the existence of a *return* transition $\mathfrak{r} \in T$. This merely gathers together the results S_{out} of the VSDG (that is, ${}^\bullet\mathfrak{r} = S_{\text{out}} \wedge \mathfrak{r}^\bullet = \emptyset$), and moreover allows postdominance to be defined by acting as the exit node. Specifically, $n \ \mathtt{pdom} \ n'$ if all paths from n' to $\mathfrak{r}$ (i.e. to any $s \in S_{\text{out}}$) pass through n. Secondly, we consider the concepts of dominance to apply only to transitions, such that $\mathtt{pdom} \subseteq T \times T$. Thus, $\mathsf{ipdom}(t)$ is the unique *transition* t' satisfying $t' \ \mathtt{pdom}^+ \ t \wedge \forall t'' \ \mathtt{pdom}^+ \ t . t'' \ \mathtt{pdom}^* \ t'$.

Further notation will also be useful, as follows. Write $D(t)$ for the nodes (transitions) postdominated by t, i.e. $\{t' \mid t \mathtt{pdom}^* t'\}$, or (interchangeably) the subgraph of the postdominator tree

induced by those nodes. (Thus, $D(\mathfrak{r})$ may stand for the entire postdominator tree.) Secondly, let $children(t)$ be the set $\{t' \mid \mathsf{ipdom}(t') = t\}$.

Dominator (and postdominator) trees have many useful properties. Specifically, Tu and Padua [TP95] prove a lemma on flowgraphs that for any node v, its *predecessors* are either $\mathsf{idom}(v)$, or descendants of $\mathsf{idom}(v)$. (That is, for any edge $u \to v$, $\mathsf{idom}(v)\ \mathtt{dom}^*\ u$).

Thus, on the VSDG framework, an alternative characterization is that the *producers* $^{\circ\bullet}t$ of places used by t are necessarily *children* in the postdominator tree of some ancestor of t:

$$\mathsf{ipdom}(^{\circ\bullet}t)\ \mathtt{pdom}^*\ t \tag{3.1}$$

Thus, every $t' \in {}^{\circ\bullet}D(t)$ is either a transition in $D(t)$, or a child of some ancestor of t.

Particularly key to our algorithm, in Section 3.3.3 below, is the set of *external producers* $^{\star}D(t)$, which are the transitions not in $D(t)$ which produce results used by transitions in $D(t)$:

$$^{\star}D(t) = {}^{\circ\bullet}D(t) \backslash D(t)$$

From the above, this set can be computed incrementally during traversal by Equation 3.2:

$$^{\star}D(t) = \left({}^{\circ\bullet}t \cup \bigcup_{t' \in children(t)} ({}^{\star}D(t')) \right) \backslash children(t) \tag{3.2}$$

Gating Paths The postdominator tree allows *paths* in the VSDG to be decomposed using a system of *gating paths*. Intuitively, a path from a place s to a transition t is a gating path if *all of its transitions are postdominated by* t. That is, a path $p = s \xrightarrow{*} t$ is a gating path iff $\forall t' \in p.t\ \mathtt{pdom}^*\ t'$.

The properties of postdominator trees ensure that if $u\, \mathtt{pdom}^*\, v$, then *all* paths from results $v^\bullet$ of v to u are gating paths, and thus do not leave the postdominator subtree of u:

$$v \in D(u) \Rightarrow \forall p \in v^\bullet \xrightarrow{*} u. \forall t \in p.t \in D(u)$$

3.3.2 Gating Conditions

Gating Conditions (GCs) are one of the key tools we use to control the action of our sequentialization algorithm in Section 3.3.3; they also guide the additional operations in Section 3.4.

The gating condition $\mathtt{gc}^u(v)$ describes the set of *gating paths* from results $v^\bullet$ of v to u, expressed as the runtime conditions[7] under which a result $v^\bullet$ would be used in computing u *along one of those paths*. (Recall gating paths were defined in Section 3.3.1 above.)

Gating conditions are based on the *gating functions* of Tu and Padua [TP95] with the addition of a disjunction constructor $\oplus$, and are given by the following grammar:

$c \in C ::=$	Λ	(always demanded)
$\mid$	$\emptyset$	(not demanded)
$\mid$	$\langle\!\langle ? \rangle\!\rangle(g, c_t, c_f)$	(pronounced "switch") for some γ-node g (according to the runtime value of the predicate of g, either c_t applies, or c_f does)
$\mid$	$c_1 \oplus c_2$	(disjunction: the node is demanded if either c_1 or c_2 says it is)

[7] In terms of the branching behaviour of γ-nodes—recall the syntactic approximation discussed in Section 3.1. This is formalized in Figure 3.9 below.

However, note that we treat the $\oplus$ constructor as both associative and commutative, and see the following normalizations as continuously applied:

$$\begin{array}{rcl} c \oplus \Lambda & \Rightarrow & \Lambda \\ c \oplus \emptyset & \Rightarrow & c \\ \langle ? \rangle(g, c, c) & \Rightarrow & c \end{array}$$

These mean that in any GC of the form $\ldots \oplus c_i \oplus \ldots$, all the c's must be $\langle ? \rangle$s. Further, operations will preserve the invariant that all such $\langle ? \rangle$s have different γ-nodes as their first element.

Lastly, we use two shorthand forms: g to stand for $\langle ? \rangle(g, \Lambda, \emptyset)$, and $\bar{g}$ for $\langle ? \rangle(g, \emptyset, \Lambda)$.

Construction of gating conditions (in Section 3.3) makes use of three utility functions, which are defined recursively on the structure of their GC arguments, as follows:

Catenation $c_1 \cdot c_2$ is associative but not commutative, and is defined as follows:

$$\begin{array}{rcl} \emptyset \cdot c & = & \emptyset \\ \Lambda \cdot c & = & c \\ \langle ? \rangle(g, c_t, c_f) \cdot c & = & \langle ? \rangle(g, c_t \cdot c, c_f \cdot c) \\ (c_1 \oplus c_2) \cdot c & = & (c_1 \cdot c) \oplus (c_2 \cdot c) \end{array}$$

(Note that handling the common cases of $c \cdot \emptyset = \emptyset$ and $c \cdot \Lambda = c$ explicitly, computes the same gating conditions more efficiently).

Union $c_1 \cup c_2$ is associative and commutative. Simple cases are defined as follows:

$$\begin{array}{c} \emptyset \cup c = c \cup \emptyset = c \\ \Lambda \cup c = c \cup \Lambda = \Lambda \\ \langle ? \rangle(g, c_t, c_f) \cup \langle ? \rangle(g, c'_t, c'_f) = \langle ? \rangle(g, c_t \cup c'_t, c_f \cup c'_f) \end{array}$$

In other cases, it must be that c_1 and c_2 are (potentially disjunctions of) $\langle ? \rangle$s. $c_1 \cup c_2$ identifies all the γ-nodes on both sides, and combines any $\langle ? \rangle$s *with the same γ-node* using the final rule above. If any of these result in Λ then that is the overall result, otherwise all the resulting $\langle ? \rangle$s are then combined together using $\oplus$ (thus preserving the invariant above).

Individual Edges The function $cond : (S \times T) \rightarrow C$, gives a gating condition for any $s \rightarrow t$ edge:

$$cond(e) = \begin{cases} \langle ? \rangle(g, \Lambda, \emptyset), & \text{if } e \text{ is a true edge to a } \gamma\text{-node } g \\ \langle ? \rangle(g, \emptyset, \Lambda), & \text{if } e \text{ is a false edge to a } \gamma\text{-node } g \\ \Lambda, & \text{otherwise} \end{cases}$$

3.3.3 The Traversal Algorithm

The algorithm is expressed as a procedure `buildPDG`(t) operating on a transition t, and is given in Figure 3.7. `buildPDG`(t) converts into PDG form *only* the nodes in the postdominator subtree $D(t)$, producing PDG $P(t)$.

Note that below we make extensive use of shorthand notation for edges. Specifically, given $S' \subseteq S$ and $T' \subseteq T$, we write $S' \rightarrow T'$ to indicate edges $\{s \rightarrow t \mid s \in S' \wedge t \in T'\}$, and similarly for *paths*, writing $S' \xrightarrow{*} t$ where $t \in T$.

Hierarchical Decomposition of Demand Conditions An essential task for the algorithm is to ensure that whenever a result of a transition v might be demanded to evaluate transition u (i.e. there is a path $v^\bullet \xrightarrow{*} u$ in the VSDG), *control dependence edges* connect $P(u)$ to $P(v)$, ensuring that $P(u)$ causes execution of $P(v)$ if necessary. The postdominator tree allows all paths in the VSDG to be decomposed and considered an edge at a time, as follows.

At each step of recursion, `buildPDG`(u) considers the *edges* $v^\bullet \rightarrow D(u)$ *entering* $D(u)$ from outside (thus $v \in {}^\bigstar D(u)$); as $D(u)$ moves from a single leaf node to $D(\mathfrak{r})$ being all the nodes in the VSDG, this eventually captures all edges. For a given u, such edges $v^\bullet \rightarrow D(u)$ may be broken down into one of two cases:

1. Edges $v^\bullet \rightarrow u$; these are handled by `buildPDG`(u) itself.

2. Edges entering into $D(u')$ for some child $u' \in \mathit{children}(u)$; these are handled by the recursive calls to `buildPDG`(u').

Composition of edges into *paths* is recorded using gating conditions. Specifically, `buildPDG`(u) maintains the gating conditions $\texttt{gc}^u(v)$ for all $v \in {}^\bigstar D(u)$, describing the *gating paths* $v^\bullet \xrightarrow{*} u$ (recall from Section 3.3.2 these are the paths through only transitions $t \in D(u)$). Crucially, `buildPDG`(u) does not need to handle non-gating paths $v^\bullet \rightarrow t \rightarrow t^\bullet \xrightarrow{*} u$ via other transitions $t \notin D(u)$: it merely ensures that $P(u)$ causes execution of $P(t)$ as necessary, and the PDG subtree $P(t)$ will itself execute $P(v)$ recursively.

Dominator Trees and Gating Conditions Thus, gating paths $v^\bullet \xrightarrow{*} u$ for $v \in {}^\bigstar D(u)$ can be broken into two cases:

1. Edges $e = v^\bullet \rightarrow u$. These cause v to be demanded exactly according to $\mathit{cond}(e)$ (defined in Section 3.3.2 above).

2. Routes from v to u *via* some child u' of u (i.e. $u = \texttt{ipdom}(u')$), where $v \in {}^\bigstar D(u')$. Thus, $\texttt{gc}^{u'}(v)$ describes the routes from v to u', and so $\texttt{gc}^u(u') \cdot \texttt{gc}^{u'}(v)$ describes the routes from v to u *that go through* u'.

This allows `buildPDG`(u) to compute $\texttt{gc}^u(v)$ by taking the union of the edges $v^\bullet \rightarrow u$ and the routes via each child u' of u. Further, recalling the definition of ${}^\bigstar D(u)$ in Equation 3.2, $\texttt{gc}^u(u')$, for $u' \in \mathit{children}(u)$, may be computed in the same way. For these nodes $u = \texttt{ipdom}(u')$, so *all* paths $u'^\bullet \xrightarrow{*} u$ are gating paths, and described by $\texttt{gc}^u(u')$. (Indeed, $\texttt{gc}^u(u')$ thus describes *all* paths in the whole VSDG by which u' may be demanded, as all such paths go through u.)

Connecting the PDG Fragments It is these $\texttt{gc}^u(u')$, for $u' \in \mathit{children}(u)$, which determine how $P(u)$ is produced by combining the $P(u')$. Specifically, `buildPDG`(u) calls a procedure `link` to add control dependence edges from $P(u)$ to each $P(u')$. The `link` procedure is shown in Figure 3.8; it is this procedure we modify in order to incorporate the two additional operations of our algorithm, namely γ-ordering (Section 3.4.1) and the use of tail nodes (Section 3.4.2).

Thus, for an arbitrary VSDG edge $v^\bullet \rightarrow u$, one of two cases applies:

1. $u = \texttt{ipdom}(v)$. In this case, during execution of `buildPDG`(u), the call to `link` will directly add an edge from $P(u)$ to $P(v)$ (either from the root node of $P(u)$, or if u is a γ-node, from the corresponding predicate node in $P(u)$—this is considered below).

2. $v \in {}^\star D(u)$. In this case, the edge will be handled by the call to buildPDG($\mathsf{ipdom}(v)$)—from Equation 3.1 above, this call dynamically encloses buildPDG(u). That is, the edge $v^\bullet \to u$ will (by recursive traversal) be concatenated onto all the paths $u \xrightarrow{*} \mathsf{ipdom}(v)$, and included in computation of $\mathsf{gc}^{\mathsf{ipdom}(v)}(v)$. This GC is then passed to link, which uses it to add a control dependence edge from the appropriate part of $P(\mathsf{ipdom}(v))$ to $P(v)$.

Return Values As buildPDG(t) traverses the postdominator tree, its return values are a triple:

- The set ${}^\star D(t)$;
- A *partial* PDG $P(t)$ computing the results of t into register(s) $\mathsf{v}_{t^\bullet}$. This PDG is partial in that it has no edges to the PDG statements corresponding to the $t' \in {}^\star D(t)$, and these will need to be added to make it a valid PDG. For simplicity, we assume the root node of these PDG fragments (also written $P(t)$) is a group node, containing at least:
  - For γ-nodes t, an appropriate PDG predicate node, with empty true and false child group nodes
  - For arithmetic nodes, a statement node $\mathsf{v}_{t^\bullet} = \mathsf{op}(\mathsf{v}_{^\bullet t})$.
- For each $t' \in {}^\star D(t)$, the Gating Condition $\mathsf{gc}^t(t')$, describing where edges to the PDG subtree $P(t')$ must be added to $P(t)$.

Topological Sorts A final key to the algorithm is how buildPDG(u) processes the children $\overrightarrow{u_i}$ of u in *topological sort* (top-sort) order. Specifically, recall from the definition of ${}^\star D(u_i)$ and the properties of postdominator trees (Section 3.3.1), that each $v \in {}^\star D(u_i)$ is either another child of u, or not postdominated by u. Since the VSDG is acyclic[8], we can sort the $\overrightarrow{u_i}$ so that whenever $u_i \in {}^\star D(u_j)$ then u_j comes *before* u_i. Thus, each u_i comes *after* every u_j which is on a path from u_i to u. The algorithm uses this to consider the $\overrightarrow{u_i}$ in turn, such that when processing each u_i, all u_j on paths ${u_i}^\bullet \xrightarrow{*} u$ have already been processed, and thus the gating condition $C(u_i)$ (see Figure 3.7) is the correct value for $\mathsf{gc}^u(u_i)$.

3.4 Additional Operations

In this section we show how our algorithm incorporates two additional operations into the framework of Section 3.3, called γ-*ordering* and γ-*coalescing*. These allow a correct and efficient PDG to be produced, avoiding the flaws of the naive translation discussed in Section 3.2.1.

Both operations concern how PDG subtrees are connected together during traversal, a task performed by the link procedure (shown in Figure 3.8). Thus, both operations are implemented by modifying link.

3.4.1 The γ-Ordering Transformation

The γ-ordering transformation is used to deal with cases of independent redundancy (as exemplified by Figure 3.2). Recall from Section 3.2.1 two naïve treatments of such a node n were not acceptable:

[8] In fact, this property is assured merely by the reducibility of the VSDG.

$\texttt{buildPDG}(u \in T) =$

Let $C(v) = \emptyset$ be a map from transitions $v \in {}^{\circ\bullet}D(u)$ to GCs.

Let P store the initial PDG fragment for u. //see text

Let D store a set of transitions.

Let $\overrightarrow{u_i} = children(u)$.

//1. Process edges to u

Set $D = {}^{\circ\bullet}u$. //The producers of operands of u

For each $v \in D$,

set $C(v) = \bigcup_{e \in v^\bullet \to u} cond(e)$.

//2. Recurse

For each u_i, let $({}^\star D(u_i), P(u_i), \texttt{gc}^{u_i}(v \in {}^\star D(u_i))) = \texttt{buildPDG}(u_i)$.

//3. Combine subtree results

Top-sort the $\overrightarrow{u_i}$ to respect $u_j \in {}^\star D(u_i) \Rightarrow i < j$. //see text

For each u_i in topological sort order,

//$C(u_i)$ is now a correct value for $\texttt{gc}^u(u_i)$—see text

call $\texttt{link}(P, C(u_i), P(u_i))$. //`link` is defined in Figure 3.8

for each $v \in {}^\star D(u_i)$, set $C(v) = C(v) \cup C(u_i) \cdot \texttt{gc}^{u_i}(v)$.

Set $D = (D \cup {}^\star D(u_i)) \backslash \{u_i\}$.

Remove entry for $C(u_i)$. //Edges $D(u_j) \to u_i$ are *inside* $D(u)$

Normalize P (by merging group nodes with only one parent into parent).

Return $({}^\star D(u), P(u), \texttt{gc}^u(v \in {}^\star D(u))) = (D, P, C)$.

Figure 3.7: The `buildPDG` algorithm. (Note mutable variables $C(\cdot)$, P and D.)

$\texttt{link}(G, c, G')$ adds edges from (children of) PDG group node G to G' according to $c \in C$ as follows:

- $\texttt{link}(G, \Lambda, G')$ adds a control edge from G to G'. In fact it may instead use a copy of G' modified to store its result into a different register, to enable sharing of tail nodes; this is explained in Section 3.4.2.
- $\texttt{link}(G, \langle ? \rangle(g, c^t, c^f), G')$, for γ-node g, identifies the corresponding PDG predicate node (as a child of G) and recurses on its true child with c^t and its false child with c^f (passing G' to both).
- $\texttt{link}(G, \emptyset, G')$ does nothing.
- $\texttt{link}(G, c_1 \oplus c_2, G')$ causes application of the γ-ordering transform, considered in Section 3.4.1.

Figure 3.8: The `link` procedure.

1. Adding control dependence edges to $P(n)$ from *both* PDG predicate nodes (i.e. those corresponding to each of the γ-nodes testing predicates P and Q in Figure 3.2) leads to an *illegal* PDG, as shown in Figure 3.4(a).

2. Adding control dependences from each PDG predicate node to *a different copy* of $P(n)$ leads to a legal PDG but one in which the code for n could be dynamically executed twice (as shown in Figure 3.4(b)).

(3. A third naïve treatment would be to add a control dependence edge to $P(n)$ from the group node *parent* of both PDG predicate nodes; however, this is not acceptable either, as it would cause the code for n to be speculatively executed, even in program executions in which *neither* γ-node demanded the value for n.)

In the context of our algorithmic framework, observe that such independent redundancy nodes n are identified precisely by being given gating conditions containing $\oplus$s. Thus, the naïve treatments above correspond to the following simplistic implementations of $\texttt{link}(G, \langle ? \rangle(g_1, c_1^{\mathrm{t}}, c_1^{\mathrm{f}}) \oplus \langle ? \rangle(g_2, c_2^{\mathrm{t}}, c_2^{\mathrm{f}}), P(n))$:

1. Recursively calling $\texttt{link}(G, \langle ? \rangle(g_i, c_i^{\mathrm{t}}, c_i^{\mathrm{f}}), P(n))$ for $i = 1, 2$. (These calls will then identify the appropriate PDG predicate node P_i and recursively apply c_i^{t} and c_i^{f} to its true and false children.)

2. Recursively calling $\texttt{link}(G, \langle ? \rangle(g_i, c_i^{\mathrm{t}}, c_i^{\mathrm{f}}), P(n))$ for $i = 1, 2$, but using distinct copies of $P(n)$ for each call.

(3. Adding a control edge directly from G to $P(n)$.)

Thus, we use $\oplus$ to guide an alternative mechanism avoiding these flaws, as follows.

The essence of our technique is a transformation of *γ-ordering*[9] whereby one *dominant* predicate node is ordered before the other *subsidiary* ones, paralleling the choice of dominant variable in construction of an Ordered Binary Decision Diagram.

In fact, gating conditions can be interpreted as boolean functions (namely, telling us whether the annotated node is demanded or not under those conditions), as in Figure 3.9. Note that for GCs $c_1 \oplus c_2 \oplus \ldots$ the intuitive boolean expression, of the form $E_1 \vee E_2 \vee \ldots$, has no *a priori* specification of evaluation order, but an order is required for a decision procedure for a sequential computer with a single program counter. Repeated application of γ-ordering effectively constructs an OBDD by choosing an ordering.

The transformation proceeds as follows. Let P_d be the dominant predicate node, and consider in turn each subsidiary predicate node P_s. Remove the CDG edge $G \to P_s$, and make P'_s be a clone of P_s with *the same children* (i.e. the same nodes, rather than copies thereof); then add CDG edges P_d.true $\to P_s$ and P_d.false $\to P'_s$. Repeat for each remaining subsidiary node. Finally, the original call to $\texttt{link}$ (with gating condition $\langle ? \rangle(g_d, c_d^{\mathrm{t}}, c_d^{\mathrm{f}}) \oplus \overrightarrow{\langle ? \rangle(g_s, c_s^{\mathrm{t}}, c_s^{\mathrm{f}})}$) can be completed by two recursive calls with distinct copies[10] of n, firstly to P_d.true with gating condition $c_d^{\mathrm{t}} \cup \overrightarrow{\langle ? \rangle(g_s, c_s^{\mathrm{t}}, c_s^{\mathrm{f}})}$ and secondly to P_d.false (similarly). The final effect is illustrated in Figure 3.10(a).

[9]Strictly, we order PDG predicate nodes rather than the γ-nodes to which they correspond; there are issues in performing the transformation directly on the γ-nodes themselves, which we address in Chapter 5.

[10]This is not strictly necessary in that PDG sequentialization will duplicate n if it is not already—discussed in Section 4.1.

Let $V^\gamma = \{v \in V \mid v \text{ is a } \gamma\text{-node}\}$ be the set of γ-nodes.
Let $G \subseteq V^\gamma$ encode a set of runtime conditions as the subset of γ-nodes which return their true inputs.
Now $eval[\![c \in C]\!] : \mathcal{P}(V^\gamma) \rightarrow$ `boolean` interprets gating conditions as functions from such sets of runtime conditions to booleans:

$$eval[\![c \in C]\!] = \begin{cases} \lambda G.\mathsf{true}, & \text{if } c = \Lambda \\ \lambda G.\mathsf{false}, & \text{if } c = \emptyset \\ \lambda G.\text{if } g \in G \text{ then } eval[\![c_t]\!](G) \text{ else } eval[\![c_f]\!](G), & \text{if } c = \langle\!\langle ? \rangle\!\rangle(g, c_t, c_f) \\ \lambda G.eval[\![c']\!](G) \vee eval[\![c'']\!](G), & \text{if } c = (c' \oplus c'') \end{cases}$$

Figure 3.9: Interpretation of Gating Conditions as boolean functions

A complication is that there may be paths between the dominant and subsidiary predicate nodes in the Data Dependence Graph, and untreated these lead to cycles. Instead all nodes on such paths must be removed (as siblings of P_d) and duplicated as children of both P_d.true and P_d.false; the DDG edges must be copied likewise, with those incident upon P_d redirected to the appropriate child of P_d.true or P_d.false (according to the virtual register causing the dependence). This is shown in Figure 3.10(b). Where a path exists from P_d to some P_s because the *predicate* of P_d depends upon P_s, it is not possible to make P_d be the dominant predicate node (this would requires branching according to the predicate before the predicate can be computed); some other predicate must be made dominant instead.

3.4.2 Coalescing of γ-Trees

We also wish to use tail nodes (introduced in Section 3.2.2) to deal with cases of exclusive redundancy (as exemplified by Figure 3.1)—recall that naïve treatment lead to PDG sequentialization duplicating such nodes, as in Figure 3.3.

This is achieved by coalescing the result registers of trees of γ-nodes, called *γ-coalescing*. Specifically, we use a modified version of the gating conditions of Section 3.3.2, in which every Λ may optionally be labelled with a *destination register* into which the value described by the GC should be placed: by making these registers correspond to the result places of γ-nodes, the extra register moves (e.g. r = rA and r2 = rA in Figure 3.3) causing the duplication[11] can be elided, and the computations tail-shared.

Such labelled GCs are written Λ^{r}, and are used by the `link` procedure to modify the PDG fragments whenever a call is made to $\texttt{link}(G, \Lambda^{\mathsf{r}}, G')$. In this case, rather than adding a control edge $G \rightarrow G'$, a modified version of G'—shared amongst all Λs with the same destination register—is used, in which all assignments to G's result register are replaced by assignments to r.

[11]Because the moves are shared less than the computation upon which they depend—this is explained in Section 4.1.

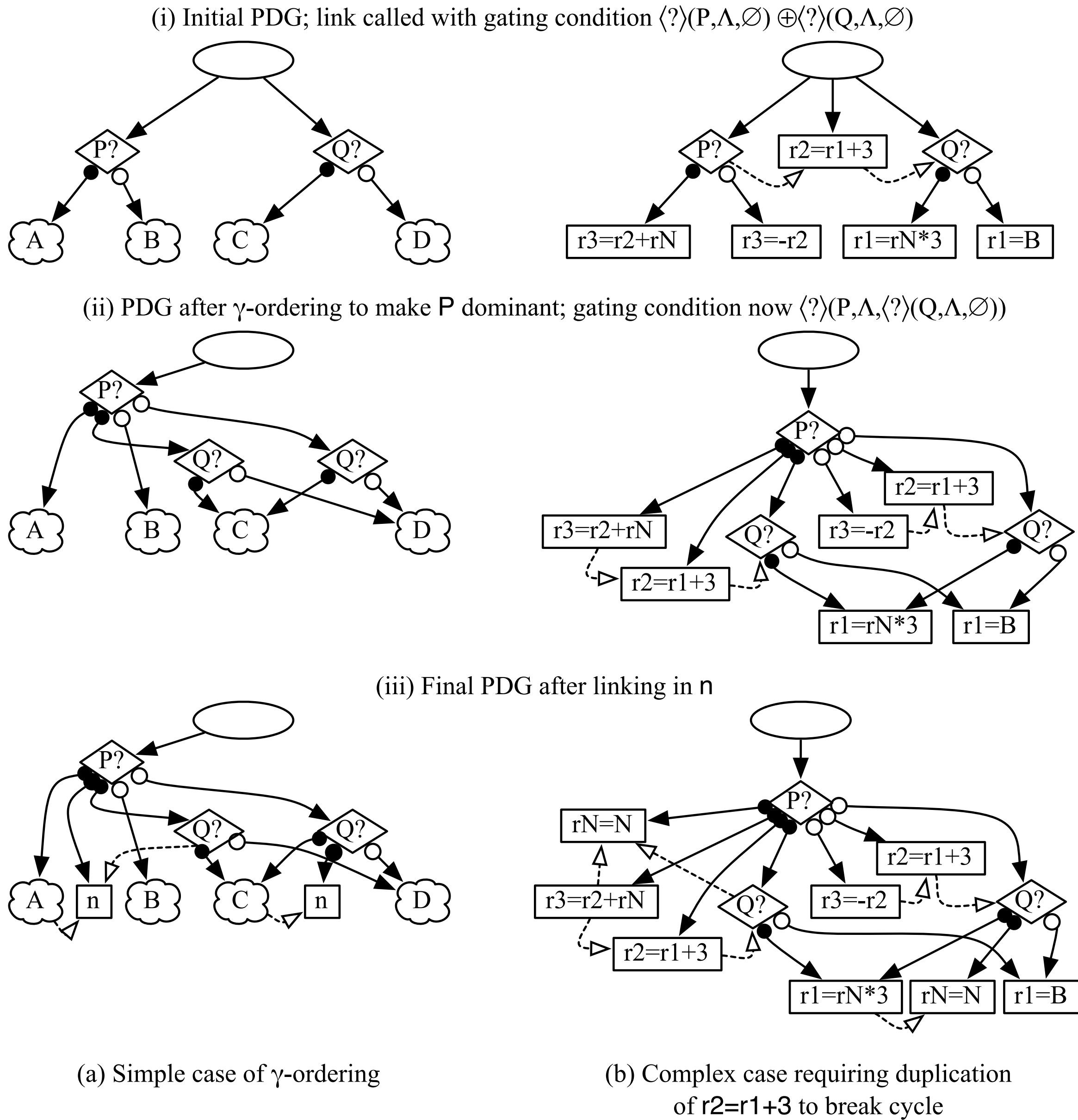


Figure 3.10: Two examples of the γ-ordering transformation. (In both cases, P is selected as dominant.)

These modified GCs originate from the true/false edges of γ-nodes, thus:

$$cond(e) = \begin{cases} \langle ? \rangle(g, \Lambda^{\mathsf{r}}, \emptyset), & \text{if } e \text{ is a true edge to a } \gamma\text{-node } g \\ \langle ? \rangle(g, \emptyset, \Lambda^{\mathsf{r}}), & \text{if } e \text{ is a false edge to a } \gamma\text{-node } g \\ & \textit{(where } \mathsf{r} \textit{ is the corresponding result register of } g\textit{)} \\ \Lambda, & \text{otherwise} \end{cases}$$

Recall the `buildPDG` algorithm (Figure 3.7) uses concatenation and union to compute

$$C(v) = C(v) \cup C(u_i) \cdot \mathsf{gc}^{u_i}(v)$$

These operations depend upon the result register rv for the node v which the gating conditions describe—specifically, concatenation $c_1 \cdot c_2$ is extended as follows:

$$\begin{aligned} \Lambda \cdot c &= c \\ \Lambda^{\mathsf{rt}} \cdot c &= c\,[\mathsf{rt}/\mathsf{rv}] \end{aligned}$$

This captures the idea that returning the RHS (c) as a tail value from the LHS (Λ^{rt}), means that any tail value for the RHS ($\Lambda^{\mathsf{rv}} \in c$) is thus a tail value for the LHS (Λ^{rt} after substitution). This means that the optimization $c \cdot \Lambda = c$ is valid only if c contains no tail registers; and $c \cdot \Lambda^{\mathsf{rt}} = c$ only if $\mathsf{rt} = \mathsf{rv}$.

For union $c \cup c'$, the rule $\Lambda \cup c = c \cup \Lambda = \Lambda$ is extended to $\Lambda^{\mathsf{r}} \cup c = c \cup \Lambda^{\mathsf{r}} = \Lambda$, for $c \neq \emptyset$. However, if in any of these cases either side contains any Λs with tail registers, extra statements must be added to the partial PDG: these take the place of the modified PDGs that would be added in by the `link` procedure as above. Specifically, when computing $G^u(n) = c \cup (G^u(v) \cdot G^v(n))$, consider each Λ^{r} in $(G^u(v) \cdot G^v(n))$: `link` would have added an edge from the corresponding part of $P(u)$ to $P(n)\,[\mathsf{r}/\mathsf{rn}]$. To make the value computed by $P(n)$ be available in the same register, a statement node r = rn must be added in that place. (Any Λs without tail registers indicate PDG locations where the value of n is always required in register rn, so no further action is necessary). An example of this is in the next section.

3.5 Worked Examples

On the Exclusive Redundancy of Figure 3.1 The running example is shown again in Figure 3.11 including its postdominator tree. Note that for simplicity we label places with their registers, and assume that the places (registers) rP and rQ are already available.

Translation proceeds by calling `buildPDG` on γ_1 as follows:

1. A PDG fragment $P(\gamma_1)$ is created, initially as shown in Figure 3.12(a).

2. The edges rA $\xrightarrow{T} \gamma_1$ and rB $\xrightarrow{F} \gamma_1$ are processed, setting $C^{\gamma_1}(A) = \langle ? \rangle(\gamma_1, \emptyset, \Lambda^{\mathsf{r1}})$ and $C^{\gamma_1}(\gamma_2) = \langle ? \rangle(\gamma_1, \Lambda^{\mathsf{r1}}, \emptyset)$. (Note the use of r1, the return register of γ_1, because the edges are true and false edges).

3. `buildPDG` recurses on the children of γ_1, i.e. A (here first) and γ_2

   (a) A has no children. Thus `buildPDG`(A) returns $P(A)$ as shown in Figure 3.12(b), and the set of external producers of A, ${}^{\star}D(A) = \emptyset$.

4. `buildPDG` is called on γ_2...

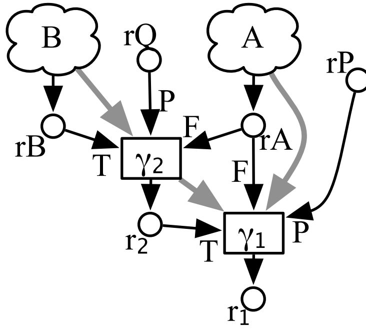


Figure 3.11: Postdominator Tree for Figure 3.1. The gray edge $u \rightarrow v$ indicates that $\mathsf{ipdom}(u) = v$.

(a) PDG fragment $P(\gamma_2)$ is created, initially the same as $P(\gamma_1)$ except testing rQ instead of rP (Figure 3.12(a)$[^{\mathsf{rQ}}/_{\mathsf{rP}}]$).

(b) The edges rB $\xrightarrow{T} \gamma_2$ and rA $\xrightarrow{F} \gamma_2$ are processed, setting $D = \{A, B\}$, $C^{\gamma_2}(B) = \langle ? \rangle(\gamma_2, \Lambda^{\mathsf{r2}}, \emptyset)$ and $C^{\gamma_2}(A) = \langle ? \rangle(\gamma_2, \emptyset, \Lambda^{\mathsf{r2}})$.

(c) γ_2's only child is B; $\mathtt{buildPDG}(B)$ returns $P(B)$ as a group node with child rB = B, much as for node A, and $^\star D(A) = \emptyset$.

(d) $\mathtt{link}(P(\gamma_2), \langle ? \rangle(\gamma_2, \Lambda^{\mathsf{r2}}, \emptyset), P(B))$ is called; recursively, this calls $\mathtt{link}(P(\gamma_2).\mathsf{true}, \Lambda^{\mathsf{r2}}, P(B))$, where $P(\gamma_2).\mathsf{true}$ is the true child of the predicate node for γ_2, and this adds in a modified version of $P(B)$, using r2 instead of rB, resulting in a new $P(\gamma_2)$ as shown in Figure 3.12(c).

(e) Entries for B are removed from D and C^{γ_2}; $\mathtt{buildPDG}(\gamma_2)$ thus returns $P(\gamma_2)$, the set $^\star D(\gamma_2) = \{A\}$, and the gating condition $\mathsf{gc}^{\gamma_2}(A) = \langle ? \rangle(\gamma_2, \emptyset, \Lambda^{\mathsf{r2}})$.

5. The children of γ_1 are topologically sorted, putting γ_2 before A.

6. $P(\gamma_2)$ is then $\mathtt{link}$ed into P with gating condition $\langle ? \rangle(\gamma_1, \Lambda^{\mathsf{r1}}, \emptyset)$; a recursive call to $\mathtt{link}(P(\gamma_1).\mathsf{true}, \Lambda^{\mathsf{r1}}, P(\gamma_2))$ adds in a modified of $P(\gamma_2)$ $[^{\mathsf{r1}}/_{\mathsf{r2}}]$ to produce a new $P(\gamma_1)$ as in Figure 3.12(d).

7. $C^{\gamma_1}(A)$ is updated to include edges $A^\bullet \rightarrow D(\gamma_2)$, by:

$$
\begin{aligned}
C^{\gamma_1}(A) \;\cup= &\;\; (C^{\gamma_1}(\gamma_2) \cdot \mathsf{gc}^{\gamma_2}(A)) \\
= &\;\; \langle ? \rangle(\gamma_1, \emptyset, \Lambda^{\mathsf{r1}}) \cup (\langle ? \rangle(\gamma_1, \Lambda^{\mathsf{r1}}, \emptyset) \cdot \langle ? \rangle(\gamma_2, \emptyset, \Lambda^{\mathsf{r2}})) \\
= &\;\; \langle ? \rangle(\gamma_1, \emptyset, \Lambda^{\mathsf{r1}}) \cup (\langle ? \rangle(\gamma_1, \Lambda^{\mathsf{r1}} \cdot \langle ? \rangle(\gamma_2, \emptyset, \Lambda^{\mathsf{r2}}), \emptyset)) \\
= &\;\; \langle ? \rangle(\gamma_1, \emptyset \cup \langle ? \rangle(\gamma_2, \emptyset, \Lambda^{\mathsf{r1}}), \Lambda^{\mathsf{r1}} \cup \emptyset) \\
= &\;\; \langle ? \rangle(\gamma_1, \langle ? \rangle(\gamma_2, \emptyset, \Lambda^{\mathsf{r1}}), \Lambda^{\mathsf{r1}})
\end{aligned}
$$

8. The set D is updated by adding $^\star D(\gamma_2) = \{A\}$ and removing γ_2 to yield $\{A\}$.

9. $\mathtt{link}$ is called to attach $P(A)$ to $P(\gamma_1)$ with gating condition $\langle ? \rangle(\gamma_1, \langle ? \rangle(\gamma_2, \emptyset, \Lambda^{\mathsf{r1}}), \Lambda^{\mathsf{r1}}$. Recursive calls add edges from $P(\gamma_1)$ to a modified PDG, $P(A)$ $[^{\mathsf{r1}}/_{\mathsf{rA}}]$ (containing a single statement r1 = A), resulting in the final PDG $P(\gamma_1)$ shown in Figure 3.12(e).

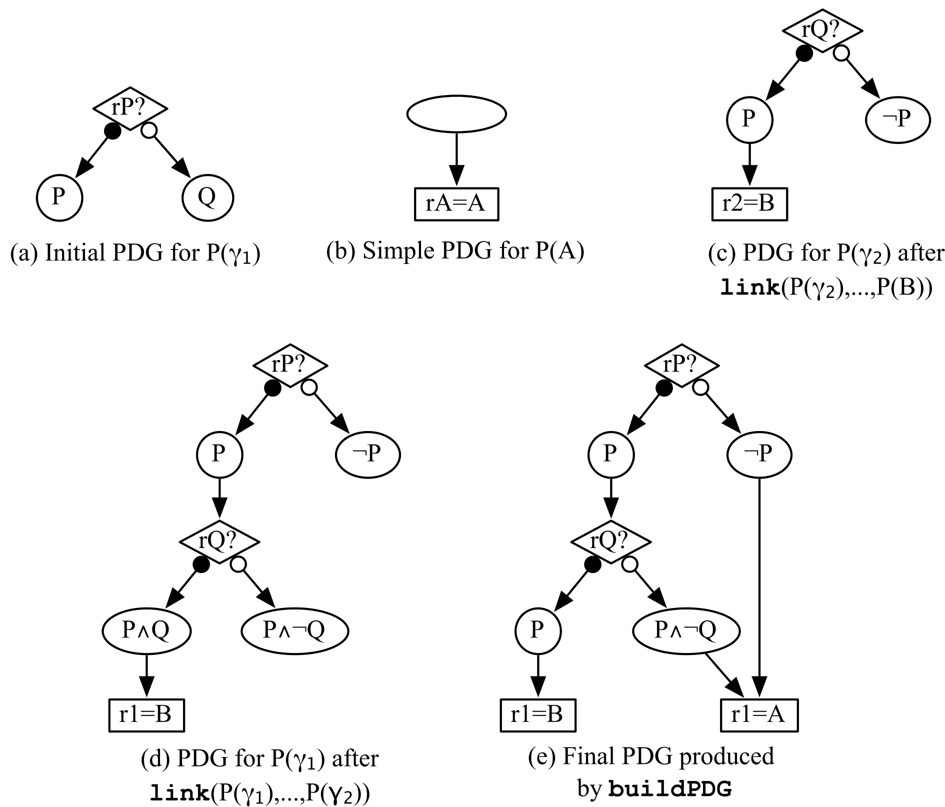


Figure 3.12: Stages in the execution of `buildPDG` on the example of Figure 3.1

10. Set D is updated to include $^{\star}D(A) = \emptyset$ and A is removed, yielding $^{\star}D(\gamma_1) = \emptyset$, which is returned from $\texttt{buildPDG}(\gamma_1)$.

On the Independent Redundancy of Figure 3.2 To illustrate the action of our algorithm on this example, we add a '+' node to combine the results of the two γ-nodes; the VSDG and postdominator tree is shown in Figure 3.11. The algorithm begins with $\texttt{buildPDG}(n^+)$ as follows:

1. An initial PDG $P(n^+)$ is created as in Figure 3.14(a).
2. `buildPDG` recurses on n^+'s children in arbitrary order, here A followed by γ_1 and γ_2...
   (a) $\texttt{buildPDG}(A)$ returns a single statement node $\textsf{rA} = \textsf{A}$ and $^{\star}D(A) = \emptyset$, as in the previous example.
3. $\texttt{buildPDG}(\gamma_1)$ is called as follows...
   (a) An initial $P(\gamma_1)$ is created as in Figure 3.12(a).
   (b) Edges $\texttt{t1} \xrightarrow{T} \gamma_1$ and $\textsf{rB} \xrightarrow{F} \gamma_1$ are processed, yielding $D = \{\texttt{op1}, B\}$, $C^{\gamma_1}(B) = \langle\!\langle ? \rangle\!\rangle(\gamma_1, \Lambda^{\textsf{r1}}, \emptyset)$ and $C^{\gamma_1}(\texttt{op1}) = \langle\!\langle ? \rangle\!\rangle(\gamma_1, \emptyset, \Lambda^{\textsf{r1}})$.
   (c) $\texttt{buildPDG}(B)$ returns a simple $P(B)$ and $^{\star}D(B) = \emptyset$, as in the previous example.

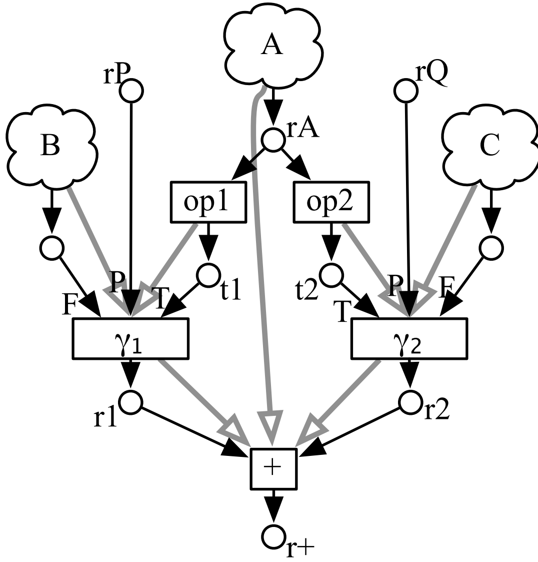


Figure 3.13: Running example of Figure 3.2 as a single VSDG subtree

(d) $\texttt{buildPDG}(\mathsf{op1})$ is called; it has no children, so $P(\mathsf{op1})$ is returned with only child $\mathsf{t}_1 = \mathsf{op1}(\mathsf{rA})$, but the edge $\mathsf{rA} \to \mathsf{op1}$ leads to $^{\star}D(\mathsf{op1}) = \{A\}$ and $\mathsf{gc}^{\mathsf{op1}}(A) = \Lambda$.

(e) $\texttt{buildPDG}(\gamma_1)$ continues by integrating B and $\mathsf{op1}$ (in either order)

(f) Here we process B first. $\texttt{link}(P(\gamma_1), \langle\!?\rangle(\gamma_1, \Lambda^{\mathsf{r1}}, \emptyset), P(B))$ adds an edge from $P(\gamma_1)$.true to a $P(B)\left[{}^{\mathsf{r1}}/_{\mathsf{rB}}\right]$ (producing Figure 3.12(c)$\left[{}^{\mathsf{rP},\mathsf{r1}}/_{\mathsf{rQ},\mathsf{r2}}\right]$). Entries for B are removed from D and $C(\cdot)$.

(g) The $\texttt{link}$ procedure is called for $P(\mathsf{op1})$ with gating condition $\langle\!?\rangle(\gamma_1, \emptyset, \Lambda^{\mathsf{r1}})$. This results in $P(\gamma_1)$ as shown in Figure 3.14(b).

(h) $^{\star}D(\mathsf{op1})$ is incorporated into the value returned for $^{\star}D(\gamma_1)$:

$$\begin{aligned} {}^{\star}D(\gamma_1) &= (D \cup {}^{\star}D(\mathsf{op1}))\backslash\{\mathsf{op1}\} \\ &= (\mathsf{op1}\} \cup \{A\})\backslash\{\mathsf{op1}\} \\ &= \{A\} \end{aligned}$$

(i) $\mathsf{gc}^{\gamma_1}(A)$ is returned as

$$\begin{aligned} \mathsf{gc}^{\gamma_1}(A) &= C^{\gamma_1}(\mathsf{op1}) \cdot \mathsf{gc}^{\mathsf{op1}}(A) \\ &= \langle\!?\rangle(\gamma_1, \Lambda^{\mathsf{r1}}, \emptyset) \cdot \Lambda \\ &= \langle\!?\rangle(\gamma_1, \Lambda, \emptyset) \end{aligned}$$

4. $\texttt{buildPDG}(\gamma_2)$ proceeds in the same way modulo differences in naming, returning $P(\gamma_2)$ as Figure 3.14(b)$\left[{}^{\mathsf{r2},\mathsf{rQ},\mathsf{op2},C}/_{\mathsf{r1},\mathsf{rP},\mathsf{op1},B}\right]$), $^{\star}D(\gamma_2) = \{A\}$ and $\mathsf{gc}^{\gamma_2}(A) = \langle\!?\rangle(\gamma_2, \Lambda, \emptyset)$

5. The children of n^+ are topologically sorted, putting γ_1 and γ_2 before A.

6. $C^{n^+}(\gamma_1) = \Lambda$ is passed to $\texttt{link}(P(n^+), -, P(\gamma_1))$.

7. D is updated by inclusion of $^{\star}D(\gamma_1)$ and removal of γ_1 to yield $D = \{A, \gamma_2\}$. $C^{n^+}(A)$ is updated to $\langle ? \rangle(\gamma_1, \Lambda, \emptyset)$.

8. Similarly $C^{n^+}(\gamma_2) = \Lambda$ is passed to `link` in $P(\gamma_2)$, resulting in the (temporary) $P(n^+)$ of Figure 3.14(c).

9. D is updated by removal of γ_2 to leave just $\{A\}$; $C^{n^+}(A)$ is updated by union with $C^{n^+}(\gamma_2) \cdot \mathtt{gc}^{\gamma_2}(A)$, thus:

$$\begin{aligned} C^{n^+}(A) &= \langle ? \rangle(\gamma_1, \Lambda, \emptyset) \cup (\Lambda \cdot \langle ? \rangle(\gamma_2, \Lambda, \emptyset)) \\ &= \langle ? \rangle(\gamma_1, \Lambda, \emptyset) \cup \langle ? \rangle(\gamma_2, \Lambda, \emptyset) \\ &= \langle ? \rangle(\gamma_1, \Lambda, \emptyset) \oplus \langle ? \rangle(\gamma_2, \Lambda, \emptyset) \end{aligned}$$

10. This GC is then passed to $\mathtt{link}(P(n^+), \langle ? \rangle(\gamma_1, \Lambda, \emptyset) \oplus \langle ? \rangle(\gamma_2, \Lambda, \emptyset), P(A))$....

    - Since this GC contains an $\oplus$, the γ-ordering transformation must be used. Specifically, γ_1 must be ordered with respect to γ_2. Suppose that γ_1 is made dominant; the predicate node for γ_2 (but not its children) is cloned and the copies made children of γ_1.true and γ_1.false as shown in Figure 3.14(d).
    - The gating condition $C^{n^+}(A)$ is recomputed as $\langle ? \rangle(\gamma_1, \Lambda \cup \langle ? \rangle(\gamma_2, \Lambda, \emptyset), \emptyset \cup \langle ? \rangle(\gamma_2, \Lambda, \emptyset))$ to give $\langle ? \rangle(\gamma_1, \Lambda, \langle ? \rangle(\gamma_2, \Lambda, \emptyset))$.
    - Recursive calls to `link` then use this to guide addition of control edges as shown in Figure 3.14(e).

11. Finally `buildPDG`(n^+) returns the PDG of the previous step along with $^{\star}D(n^+) = \emptyset$.

3.6 Chapter Summary

We have seen how VSDGs may be converted into PDGs by a post-order traversal of the dominator tree of the VSDG, using Gating Conditions to guide the application of both γ-ordering and γ-coalescing. These operations result in efficient PDGs suitable for input to existing PDG sequentialization techniques.

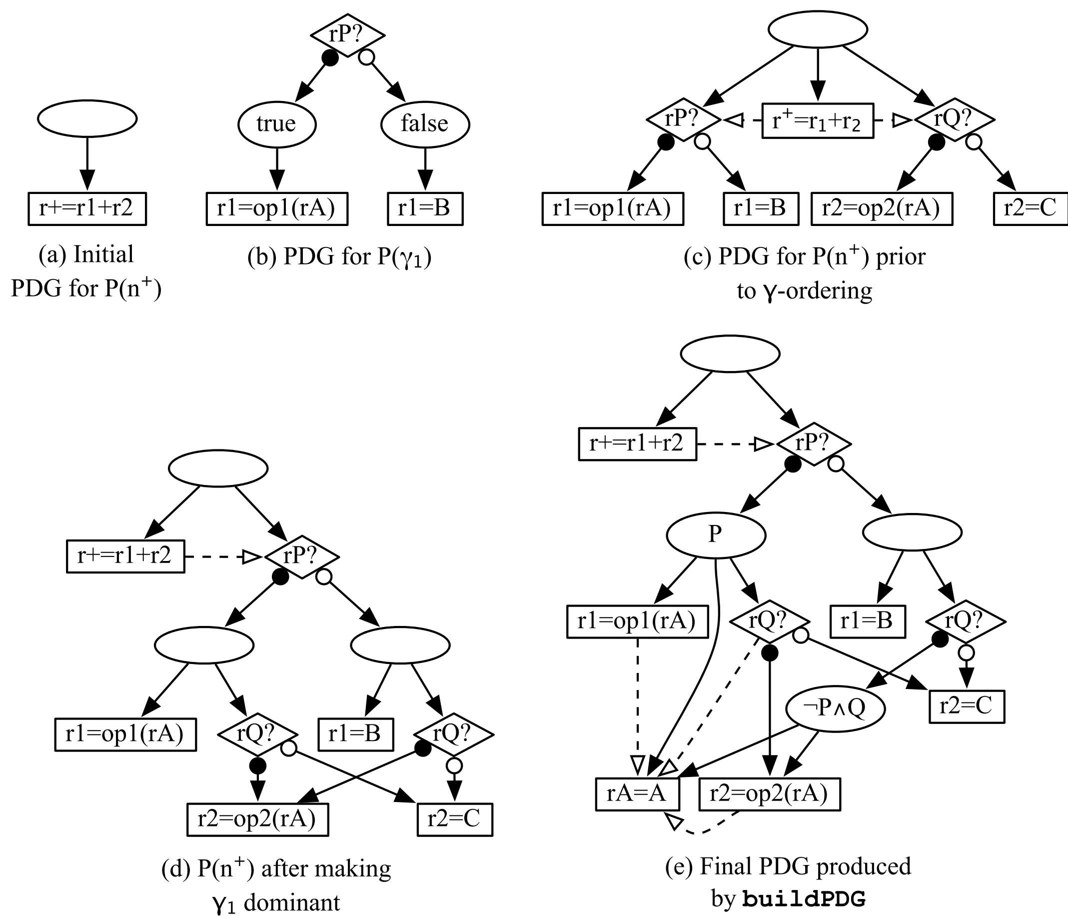


Figure 3.14: Stages in the execution of `buildPDG` on the example of Figure 3.2

CHAPTER 4

PDG Sequentialization

Structure of this Chapter This chapter has two main strands. Firstly, in Section 4.1 we review the effect of PDG sequentialization, including a precise characterization of duplication-freedom taken from existing literature, and in Section 4.2 give an efficient solution for a special case. Secondly, we compare the first two phases of our approach—proceduralization of the VSDG followed by PDG sequentialization—with existing techniques for (i) VSDG sequentialization in Section 4.3, and (ii) classical CFG code motion in Section 4.4.

4.1 Duplication-Freedom

The problem of PDG sequentialization was first considered by Ferrante et al. [FM85, FMS88] and has since received attention from a number of other authors [SAF90, BH92, Ste93, ZSE04] who have made improvements to both efficiency and generality with regards to loops.

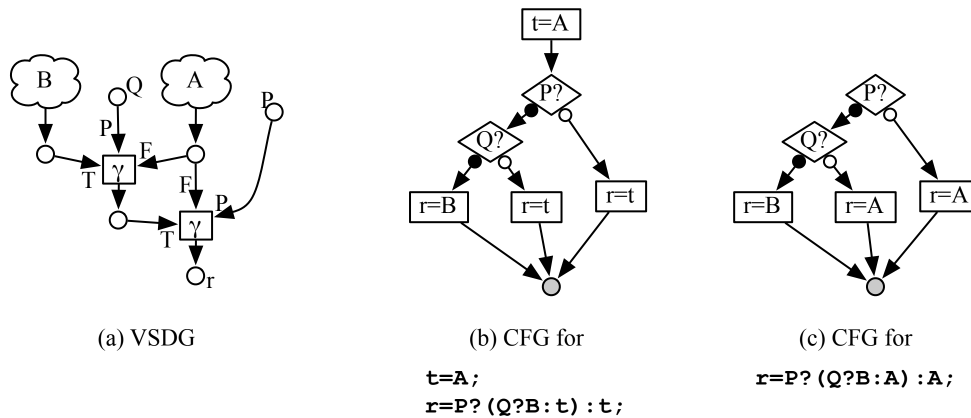
A sequentialization (CFG) in which no PDG node is duplicated is known as *concise*; recall from Section 2.4 that a PDG is described as *duplication-free* when a concise CFG sequentialization exists[1]. The requirements on a PDG for duplication-freedom are precisely characterized by Ferrante et al. [FMS88], as follows:

> A PDG (Program Dependence Graph) is *duplication-free* iff for every group node G, with children $\{C_i\}$, a topological sort of the C_i respecting the data dependence edges exists such that:
>
> $$cc(C_1) \rightarrow \ldots \rightarrow cc(C_i)$$
>
> where $cc(n)$ are the *control conditions* under which node n executes, and $\rightarrow$ is logical implication.

In particular, if all predecessors of each C_i are group nodes—a constraint trivially satisfiable by splitting any edges from predicate nodes (to go via a fresh group node)—the above is equivalent to:

$$preds(C_1) \subseteq \ldots \subseteq preds(C_i)$$

[1] Ball and Horwitz refer to this property as the *feasibility* of the CDG subgraph of the PDG [BH92].

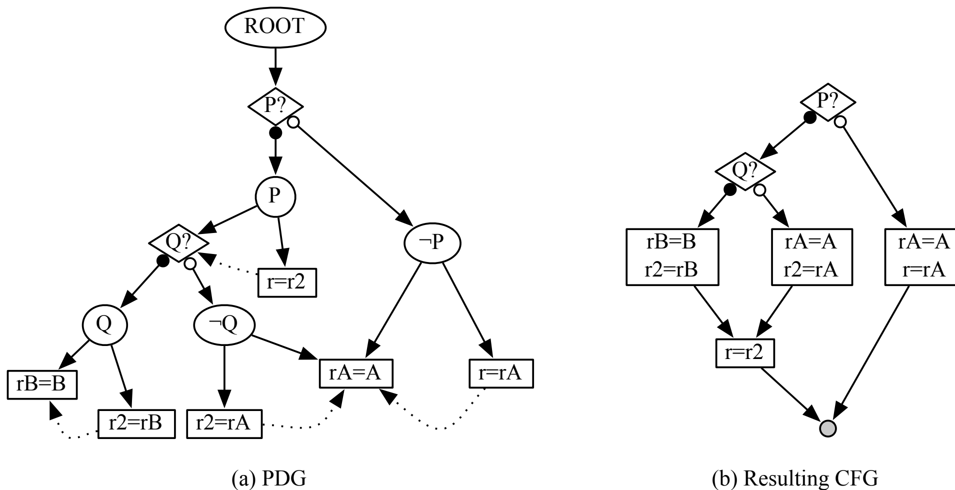


(a) VSDG

(b) CFG for
`t=A;`
`r=P?(Q?B:t):t;`

(c) CFG for
`r=P?(Q?B:A):A;`

Figure 4.1: Two programs—(b) speculating, and (c) duplicating, computation of A—informally known as *exclusive* redundancy (repeated from Figure 3.1)



(a) PDG

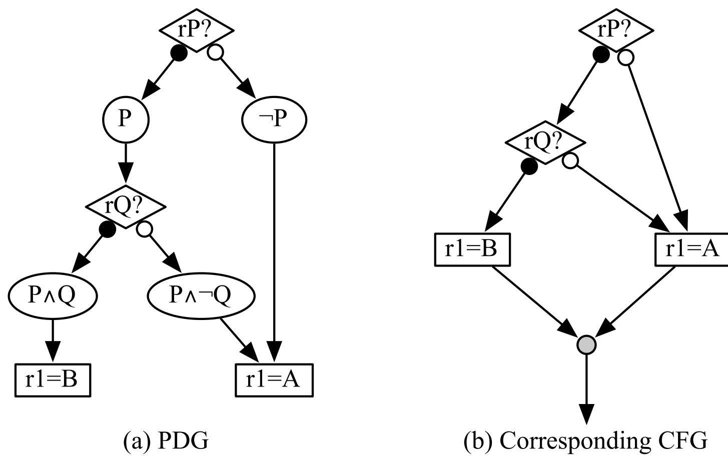
(b) Resulting CFG

Figure 4.2: Naive translation of the VSDG of Figure 4.1 (repeated from Figure 3.3)

When this condition is not satisfied, some nodes must be duplicated in order to produce a satisfying arrangement. In such cases, the challenge is thus to produce the *smallest* possible duplication-free PDG, that is, to find the smallest number (or code size) of nodes that need to be duplicated. We see this as the main task of PDG sequentialization; however, it has not been well-studied, and although conjectured NP-complete, this has not been proven. Ferrante et al. [FMS88] give a number of heuristics, and a number of special cases admit polynomial-time solutions. In Section 4.2 we will study one such, corresponding to VSDGs which are *trees* of γ-nodes returning exactly two values.

4.1.1 Effect on Running Examples

We now consider the effect of PDG sequentialization on the running examples of Chapter 3. Firstly, the PDG produced using the naïve algorithm of Section 3.2.1 on the "exclusive" redundancy of Figure 4.1, as shown in Figure 4.2(a).



(a) PDG (b) Corresponding CFG

Figure 4.3: Result of `buildPDG` on the running example of Figure 4.1

Observe the group nodes $\neg$P and $\neg$Q, and their children (collectively) r $=$ rA, r2 $=$ rA and rA $=$ A. Duplication-freedom requires that for $\neg$P (respectively $\neg$Q) we find an ordering C_1, C_2 of r $=$ rA (respectively r2 $=$ rA) and rA $=$ A which both:

- respects the data dependences between them—i.e. puts rA $=$ A before r2 $=$ rA or r $=$ rA.
- Satisfies $preds(C_1) \subseteq preds(C_2)$, where

  $$preds(\mathsf{r2} = \mathsf{rA}) = \{\neg\mathsf{Q}\}\ ;\ preds(\mathsf{r} = \mathsf{rA}) = \{\neg\mathsf{P}\}\ ;\ preds(\mathsf{rA} = \mathsf{A}) = \{\neg\mathsf{Q}, \neg\mathsf{P}\}$$

  —i.e. puts rA $=$ A after r $=$ rA or r2 $=$ rA.

Thus, duplication-freedom is not satisfied—instead PDG sequentialization must duplicate rA $=$ A, resulting in Figure 4.2(b).

In contrast, our algorithm using tail nodes produced the PDG of Figure 4.3(a), which is already duplication-free, and results in CFG Figure 4.3(b).

On the *independent* redundancy of Figure 4.4, the resulting PDG $P(n^+)$ shown in Figure 4.5 is *not* duplication-free, as can be seen from the group node $\neg P \wedge Q$ (also P) and its two children: The execution conditions of r2 $=$ op2(rA) (respectively r1 $=$ op1(rA)) imply those of rA $=$ A, requiring that rA $=$ A come second, but the data dependence edge requires the opposite ordering. Thus rA $=$ A must be duplicated, producing df-PDG Figure 4.5(b) or CFG (c).

We will compare these results with other techniques for VSDG sequentialization in Section 4.3.

4.2 A Special Case of PDG Sequentialization

In this section we will study a special case of minimising the duplication required to produce a df-PDG from a PDG, corresponding to VSDGs which are *trees* of γ-nodes returning exactly two values. Thus, the PDGs consist of trees of predicate nodes, eventually terminating in *leaves* assigning values to two registers, denoted by r1 and r2. (Although these values may be computed by arbitrary-sized subgraphs in the PDG or VSDG, we require these subgraphs to be disjoint, and hence can consider them as single nodes.)

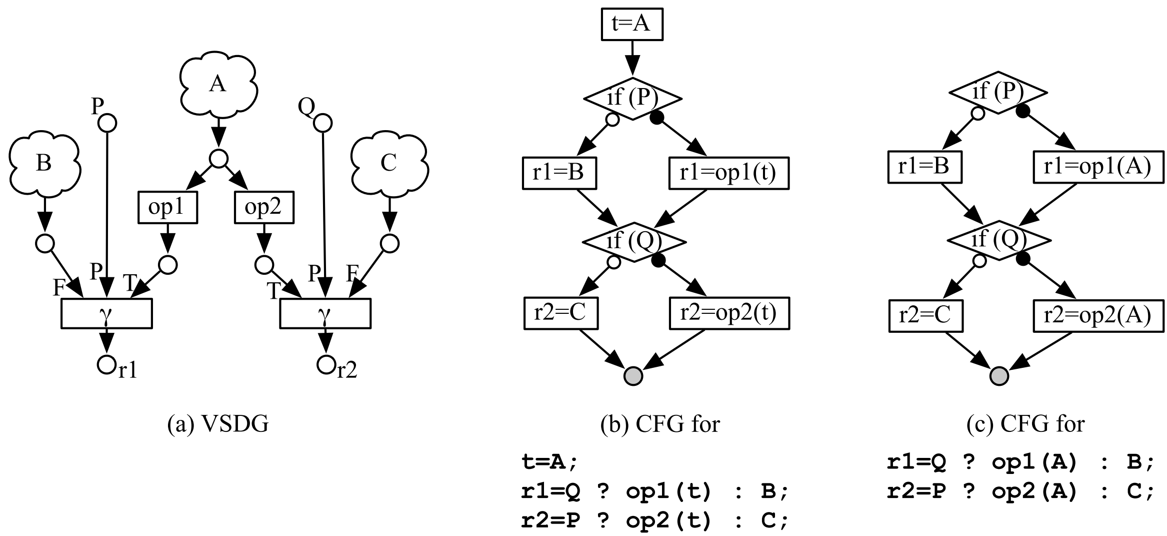


Figure 4.4: Two programs—(b) speculating, and (c) duplicating, computation of A—informally known as *independent* redundancy (repeated from Figure 3.2)

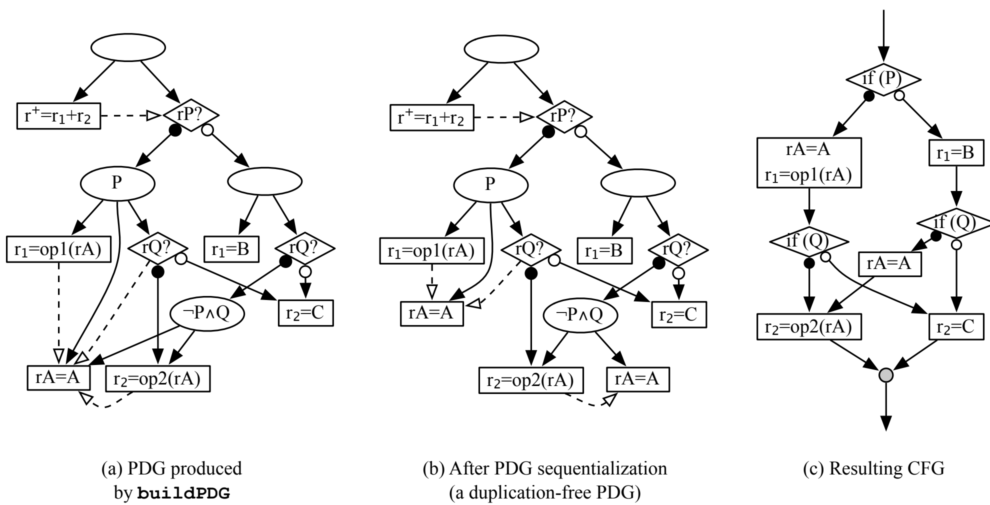


Figure 4.5: Result of our algorithm on running example of Figure 4.4

Our solution proceeds by constructing a form of *sharing graph* from the PDG. This is a bipartite graph G such that a *vertex cover* of the sharing graph identifies a set of PDG nodes which can be duplicated to produce a sequentialization (df-PDG).

First, in Section 4.2.1 we present some graph theoretic background, defining both *bipartite graphs* and vertex coverings. Then, we present our solution in two stages: first Section 4.2.2 presents a solution to a simplified "unweighted" case, and secondly Section 4.2.3 presents the more complex solution to the weighted case, additionally allowing elements to be shared *before* branches.

4.2.1 Bipartite Graphs and Vertex Coverings

Bipartite Graphs An undirected graph $G = (V, E \subset V \times V)$ is called *bipartite* if its vertices can be separated into two disjoint groups A and B such that every edge has exactly one endpoint in each group:

$$V = A \cup B$$
$$A \cap B = \emptyset$$
$$E \subseteq A \times B \cup B \times A$$

For example, the two-node clique is bipartite, but no clique larger than that is.

Vertex Coverings A *vertex covering* of a graph is a subset $V' \subseteq V$ of the vertices, such that every edge is incident on at least one node in that subset:

$$\forall (u, v) \in E.(u \in V') \vee (v \in V')$$

A *minimal* vertex covering is one that is of least size (number of vertices) among all vertex coverings. For general graphs, finding a minimal vertex covering is an NP-complete problem [npc, GJ79]. However, in bipartite graphs $G = (V = A \cup B, E \subseteq A \times B)$, a minimal vertex covering $V' \subseteq V$ can be found in polynomial time, as follows.

Firstly, a *maximal matching* is found. A *matching* $M \subseteq E$ is a subset of the edges such that no vertex appears as endpoint of more than one edge in the matching:

$$(u, u') \in M \wedge (v, v') \in M \to \{u, u'\} \cap \{v, v'\} = \emptyset$$

A matching M is said to be *maximum* if there is no matching strictly containing it ($\nexists M' \supset M$). It is said to be *maximal* if there is no larger matching (i.e. with more edges) at all ($\nexists M'.\|M'\| > \|M\|$). Maximal matchings may be found for arbitrary graphs in polynomial time [Gal86]; however, for bipartite graphs, much simpler algorithms are effective, and these are a standard item of many textbooks [CLRS01].

Secondly, the vertex covering is obtained from the maximal matching $M \subseteq E$ as follows:

- Let S be the smallest set of vertices satisfying the following three conditions:

$$a \in A \wedge \nexists (a, b) \in M \to a \in S$$
$$a \in A \wedge a \in S \wedge b \in B \wedge (a, b) \in E \to b \in S$$
$$b \in B \wedge b \in S \wedge a \in A \wedge (a, b) \in M \to a \in S$$

  (that is, informally, S is the unmatched vertices in A together with those reachable along all edges from A to B and edges from B to A in the matching).

- Let $T = V \backslash S$ be the other nodes.
- Let $K = (S \cap B) \cup (T \cap A)$
- K is a minimal vertex covering

4.2.2 Unweighted Solution

In this section, we will consider a further restricted class of PDGs, composed only of predicate nodes, leaf nodes, and group nodes with exactly two children both of which are leaf nodes. Further, we will assume all leaf nodes are of equal size, that is, the static cost of duplicating any one is the same.

Construction of Sharing Graph For this simple form of PDG, we construct a *simple sharing graph* $G = (V, E)$ in which:

- The nodes V are the leaf statements of the PDG
- The (undirected) edges $E \subseteq V \times V$ represent the pairings, i.e. there is an edge between any two leaves with a common group node parent

Note that this graph is necessarily bipartite, as each leaf assigns to either r1 or r2; if the same value is assigned to r1 in some places and r2 in others, these must be counted as distinct leaves (and duplicated before construction of the sharing graph). Note that this is not a limitation of our technique: no sequentialization can share computation of such a value between leaves assigning to r1 or r2, or both, as the assignment of the value to a different register must follow computation in each case.

For example, see Figure 4.6, where the VSDG and PDG of part (a) results in the simple sharing graph (b).

Interpretation of the Vertex Covering Any minimal vertex covering $V' \subseteq V$ of a simple sharing graph corresponds to a minimal-sized PDG sequentialization as follows. For each group node in the PDG, at least one of its two children must be in the vertex covering. Select c as the child *not* in the covering, if possible; if both children are in the vertex cover, select c arbitrarily (the choice is not important as all leaf statements are the same size). Make a copy c' of c and replace the edge $G \rightarrow c$ with an edge $G \rightarrow c'$. Thus, c' is specific to G and will be executed first; its sibling (in V') may still shared, and will be executed last, after merging the control flow of separate leaves/paths in the CFG.

Each group node now has at most one child shared with any other group node, hence the PDG is duplication-free. The total size is (the number of node copies to do first) plus (the number of nodes to do last); from the above, these are (the number of edges in the bipartite graph) plus (the number of vertices in the covering). Since the former is fixed, the size of the sequentialized PDG is linear in the size of the covering, hence duplication is minimized only by a minimal covering.

For example, Figure 4.6(c) and (d) show two coverings and the PDGs and CFGs corresponding to each.

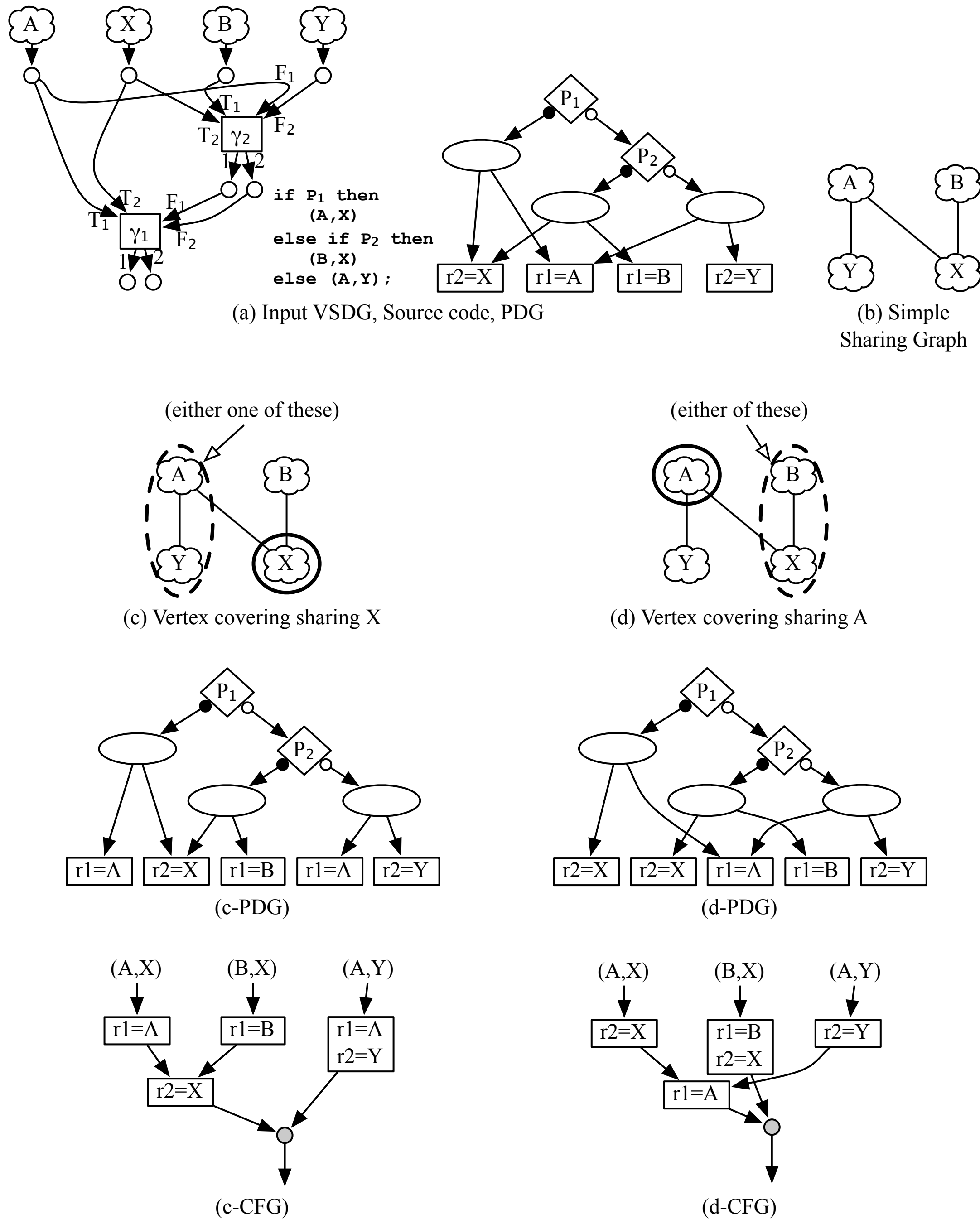


Figure 4.6: Construction and Interpretation of Unweighted Case.

.

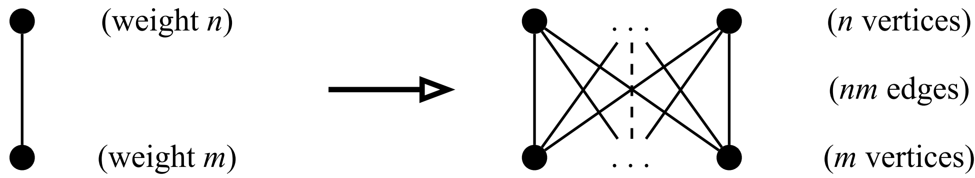


Figure 4.7: Reduction of Weighted Vertex Covering to Vertex Covering

4.2.3 Weights and Measures

In this section we will extend the previous solution to take into account leaf nodes (or subtrees) with different static sizes, and also more general PDGs in which some leaf nodes may selected *before* all predicates have been tested (that is, leaf nodes may be siblings of predicate nodes—see Figure 4.8(a) and (b) for an example). First, we explain how vertex coverings may be computed to take into account *weights* assigned to the nodes of a bipartite graph. Then we present the *sharing conflict graph*, and show how any covering of this identifies a PDG sequentialization (df-PDG or CFG) whose size is linear in the total weight of the covering.

Weighted Vertex Coverings A *Minimal-weight vertex cover* for a graph $G = (V, E)$ and a *weighting function* $w : V \to \mathcal{N}$ is a vertex covering $V' \subseteq V$ which minimises

$$\sum_{v \in V'} w(v)$$

Such a minimal-weight covering can be found by transforming G into a new graph G_w:

$$G_w = \left(\begin{array}{l} \bigcup_{v \in V} \{v_1, \ldots, v_{w(v)}\}, \\ \bigcup_{(u,v) \in E} (\{u_1, \ldots, u_{w(u)}\} \times \{v_1, \ldots, v_{w(v)}\}) \end{array} \right)$$

This transformation is also shown in Figure 4.7. Note in particular that G_w is bipartite iff G is, and further, that $\|G_w\|$, the size of G_w, is bounded by a polynomial on $\|G\|$ (so long as the weights $w(v)$ are bounded likewise). As the figure shows, the edge $(u, v) \in G$ (which forces a cover of G to include either u or v) forces any cover of G_w to include either all the u_i (at cost $w(u)$) or all the v_i (at cost $w(v)$). Hence, a minimal vertex covering of G_w is a minimal-weight vertex covering of G with weighting function w.

Construction Of Sharing Conflict Graph The sharing conflict graph is a weighted bipartite graph $G = (V, E)$ constructed as follows:

- The nodes in the sharing conflict graph correspond one-to-one with the nodes potentially appearing in the resulting sequentialization. That is, for each leaf node or subtree v, of size w, in the input PDG, the sharing conflict graph incorporates:
  - A distinguished vertex v^{share}, with weight w; intuitively, this corresponds to a PDG node which will compute v *last*, shared between multiple control dependence predecessors.

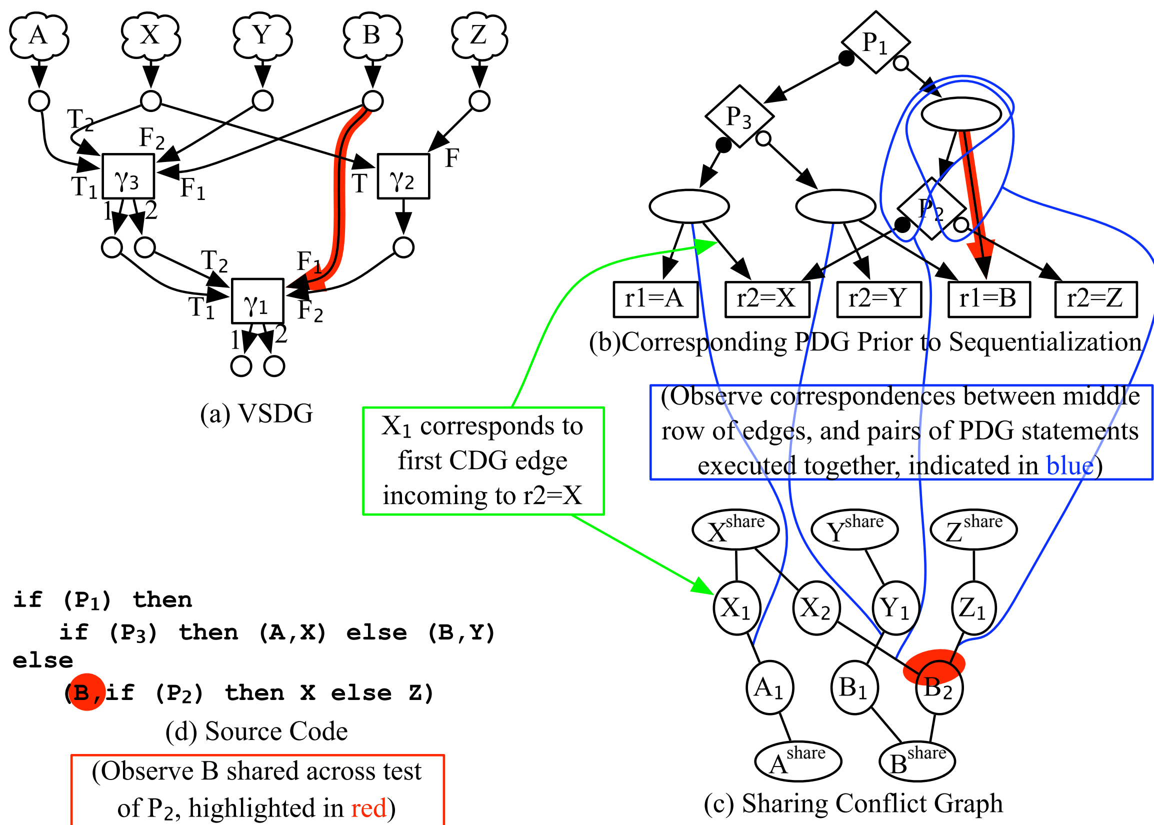


Figure 4.8: Sharing conflict graph corresponds to a PDG.

- A vertex v_j (for fresh j), also with weight w, for each control dependence parent of v; intuitively, this corresponds to a copy of v particular to that parent.

- The edges are constructed to restrict possible vertex coverings, such that any PDG containing only nodes corresponding to those in a vertex covering, is a valid, duplication-free, sequentialization. Specifically, the sharing conflict graph contains:

  - an edge $v^{\text{share}} \leftrightarrow v_j$ for each $v_j \in V$; and
  - an edge $u_j \leftrightarrow v_k$ for each pair of PDG leaf nodes (u, v) which could be executed together, where u_j and v_k are the vertices for the control dependences causing such an execution.

The effect of this procedure, shown in Figure 4.8(c), is to build a chain $u^{\text{share}} \leftrightarrow u_j \leftrightarrow v_k \leftrightarrow v^{\text{share}}$ for each tuple (u, v), with j and k such that whenever one leaf node v may be selected without testing all predicates required to select the other, the corresponding v_j is shared between all the tuples that may result.

Interpretation of Covering The correspondence between the nodes of the sharing conflict graph and copies of the leaf statements or subtrees in the output PDG was explained above. Two examples are shown in Figure 4.9. Note that in each chain $u^{\text{share}} \leftrightarrow u_i \leftrightarrow v_j \leftrightarrow v^{\text{share}}$, the covering must include at least one copy of u (either u^{share} or u_j), *and* at least one copy of

v (either v^{share} or v_j), thus specifying shared or unshared nodes for u and v; moreover, it must include either u_i or v_j—that is, it may not include only u^{share} and v^{share}—thus at most one of the leaf nodes may be shared with other tuples. (In the case where the covering includes both u_j and the corresponding v_k, but neither u^{share} nor v^{share}, there is no tail-sharing.) This ensures the result is a valid (duplication-free) sequentialized PDG.

If u_i, v_j and one or both $^{\text{share}}$ vertices are all included, multiple df-PDGs are possible using any valid subset of the included vertices, but these will all be of equal size.

4.3 Comparison with VSDG Sequentialization Techniques

Only two techniques for sequentialization of the VSDG exist in the literature (indeed, the VSDG was itself developed because of problems in the sequentialization of the earlier VDG [WCES94]):

1. Johnson's algorithm [JM03, Joh04] for combined Register Allocation and Code Motion (RACM)
2. Upton's algorithm [Upt06] for sequentialization of gated data dependence graphs

We compare our approach to each of these in Subsections 4.3.1 and 4.3.2.

4.3.1 Johnson's Algorithm

Johnson and Mycroft sequentialize a VSDG by first inserting *split nodes* based on postdominance by the true or false operands of γ-nodes. This is followed by a combined register allocation algorithm including code motion, but for the purpose of avoiding register spillage rather than reducing node evaluation. Thus, Johnson's technique is more relevant to the discussion of node scheduling techniques in Chapter 6 (in fact we show how it can de adapted into our framework as a separate pass after PDG sequentialization).

However, the VSDG with split nodes specifies an evaluation strategy in the same way as a df-PDG, and so the two can be compared, although the two compiler's aims and criteria for optimality are different: whereas we aim for the smallest df-PDG that is dynamically optimal, Johnson aims purely to reduce code size. This justifies his choice of a call-by-value strategy, speculatively evaluating nodes in order to minimize their number of static occurrences, and his technique tends to produce smaller and somewhat slower code than ours[2]. Thus, it is helpful to recall the maxim "the most normalizing compiler is the most optimizing one" discussed in Chapter 1. In particular we will consider the two running examples of Figure 4.1 and Figure 4.4.

If we consider the two input CFGs of Figure 4.1, we see Johnson's algorithm normalizes both to output CFG (b) (speculating). These is consistent with his "smallest size" goal, but the shortcircuit version of Figure 4.3 produced by our technique is clearly superior: it is just as small, and *also* faster—as fast as any other possibility and the most efficient way the programmer might have written it (if the language used included shortcircuit operators such as C's `||` and `&&`—our technique is also effective on cases which cannot easily be expressed with such operators.). Thus, Johnson's algorithm loses out on (common!) programs written with such operators (producing slower code than naïve CFG techniques).

[2] At least ignoring cache effects.

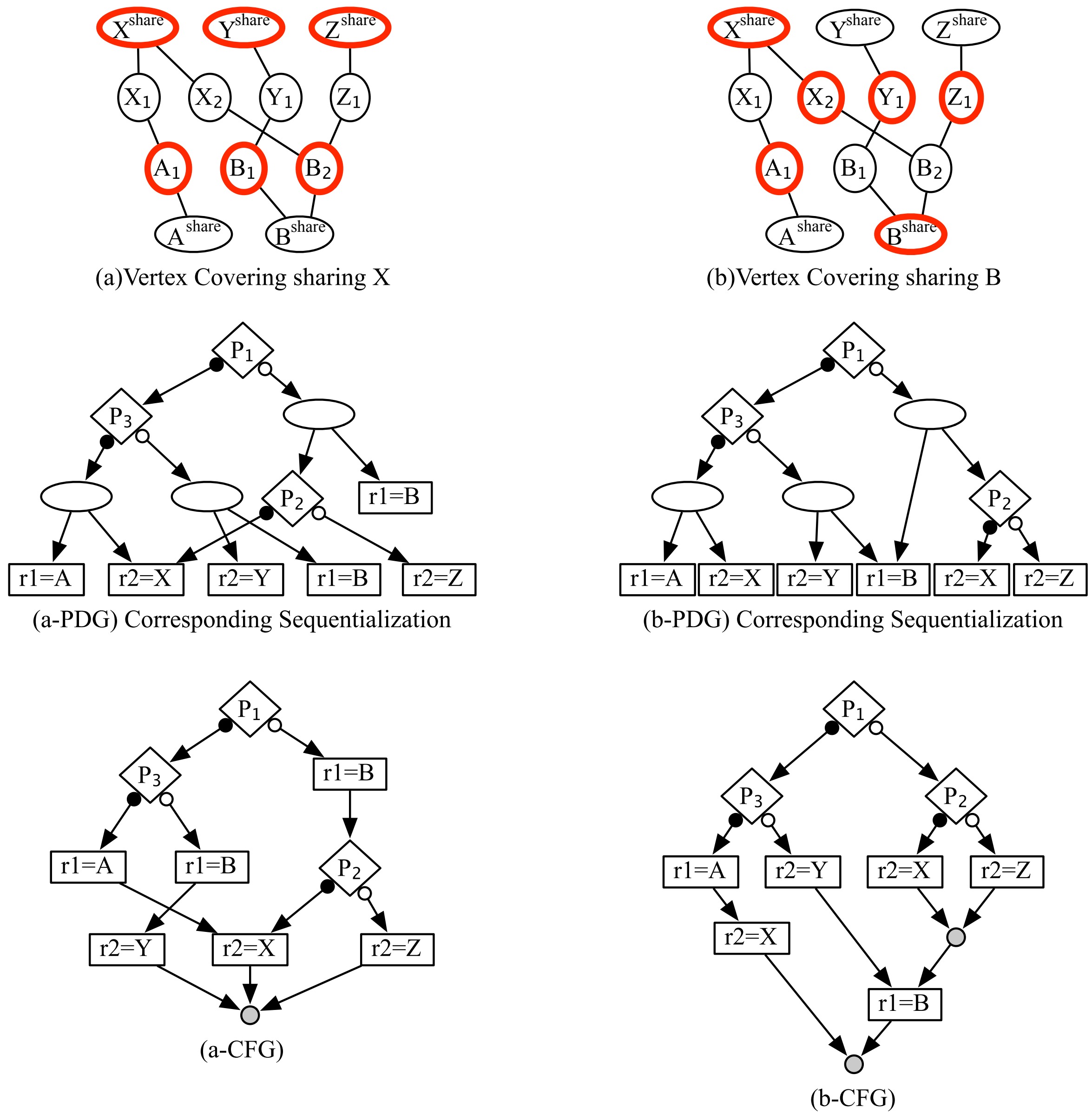
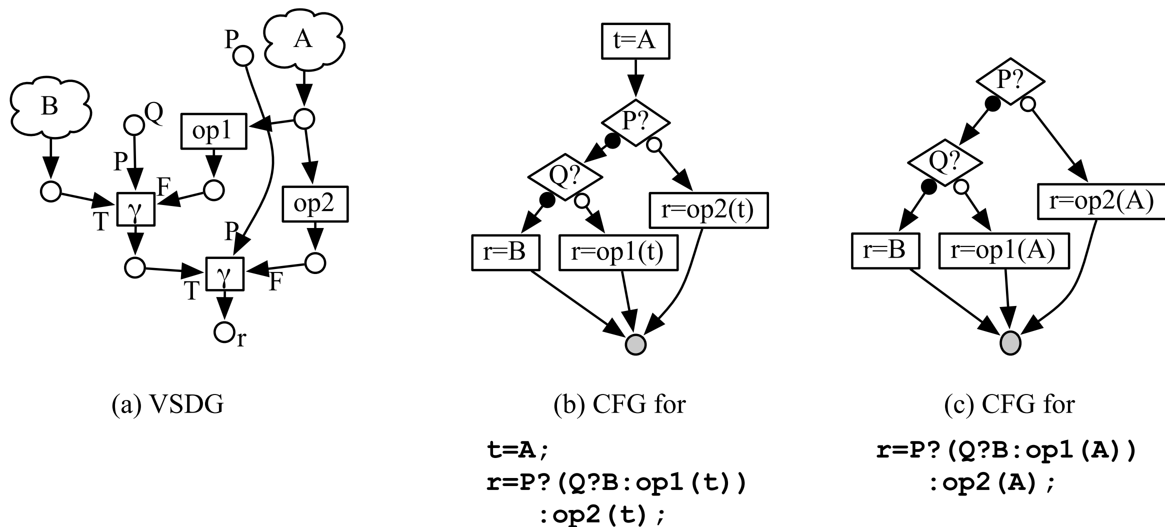


Figure 4.9: Possible minimal-weight coverings, and corresponding sequentialized PDGs, for Figure 4.8.

.



(a) VSDG

(b) CFG for
`t=A;`
`r=P?(Q?B:op1(t))`
`:op2(t);`

(c) CFG for
`r=P?(Q?B:op1(A))`
`:op2(A);`

Figure 4.10: Another *exclusive redundancy*, akin to Figure 4.1, but in which additional operation are performed on the redundant node A

However, when shortcircuit evaluation is not possible due to the presence of extra operations—for example, Figure 4.10—our algorithm produces output CFG (c), whereas Johnson's algorithm continues to normalize both input CFGs to output CFG (b). This is effective normalization: producing the output consistent with the compiler's optimality criteria regardless of the form written by the programmer.

On the two input CFGs of Figure 4.4, we again see Johnson's algorithm normalizing both input CFGs to the smallest output, namely CFG (b). Our algorithm produces CFG Figure 4.5; this is both larger and faster. (However, it is hard to apply arguments of normalization as this CFG cannot be expressed in most programming languages!)

4.3.2 Upton's Algorithm

Upton sequentializes a VSDG by constructing a treelike, and thus trivially duplication-free, dPDG[3]. The dPDG→CFG conversion is accomplished by taking an arbitrary topological sort of the children of each group node, followed by standard register allocation techniques; whilst this seems like a missed opportunity to apply node scheduling techniques, his dPDG construction technique can be compared directly with our proceduralization technique of VSDG→PDG conversion followed by PDG sequentialization. The comparison is particularly pertinent because he also implements a lazy evaluation strategy, and has the same criteria of optimality (speed first, then size) as us.

(However, in a deviation from our approach discussed in Section 3.1, he uses exponential-time BDD-based techniques to solve the boolean equations describing demand conditions precisely, whereas we take the standard approximation that γ-nodes multiplex independently.)

Considering our running examples, on Figure 4.1, Upton's compiler produces output of the form of CFG (c) (duplicating). Although consistent with his "speed first" aim (it is faster

[3]The dPDG is a variant of the PDG with functional nodes rather than imperative statements, with the CDG replaced by a Demand Dependence Graph

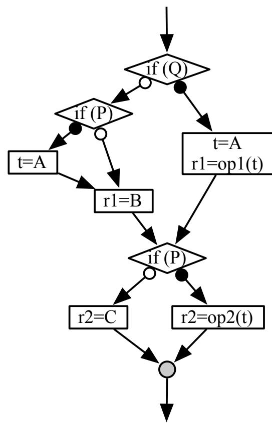


Figure 4.11: Sequentialization of VSDG of Figure 4.4 by Upton's algorithm. (Performs extra tests)

than CFG (b) (speculating)), our result (Figure 4.3) exploiting shortcircuit evaluation is clearly superior—it is just as fast, and also smaller, and static optimality is a secondary goal for Upton as well as us. (Indeed, Upton's technique loses out to naïve CFG techniques on such examples.)

When shortcircuit evaluation is not possible due to the presence of additional operations—for example, Figure 4.10—both our algorithm and Upton's produce output CFG (c); no better possibility exists.

On both of the two input CFGs of Figure 4.4, Upton's algorithm produces the output CFG of Figure 4.11. If we consider Upton's criteria for optimality (given in Section 3.1), this is dynamically better than Figure 4.4(b) or (c): 0–1 evaluations of A as opposed to exactly once or 0–2 times, respectively. While Upton's output is statically larger than either because of the presence of extra test on P, static optimality is a secondary aim. Comparing it with the CFG produced by our algorithm (Figure 4.5), we see both perform the same (optimal) amount of node evaluation, and are statically the same size.

However, our algorithm's result performs fewer runtime tests and branches, making it dynamically more efficient, so we see it as strictly better in this case. (We consider this issue again in Section 4.4.2 below.)

4.4 Comparison with Classical CFG Code Motion

In this section we compare the effectiveness of our algorithm with classical code motion algorithms on the CFG. *Code motion* is a generic term for transformations that move computations between program points (basic blocks), usually with the aim of reducing the number of computations performed. A formal definition is given by Knoop and Ruthing in their work on Partial Redundancy Elimination:

Definition 4.1 due to Knoop and Ruthing [KRS94a]:

Code motion *is a technique for improving the efficiency of a program by avoiding unnecessary recomputations of a value at runtime. This is achieved by replacing the original computations of a program by temporary variables (registers) that are initialized correctly at suitable program points.*

However in common with Bodik, Gupta and Soffa [BGS98b] we see code motion as a transformation mechanism or framework capable of being applied for many purposes, considered further in Section 4.4.4. Nonetheless its application to reducing unused computations (*Partial Dead Code Elimination* or PDCE) and unnecessary recomputations (*Partial Redundancy Elimination* or PRE) forms an important part of the optimization stage of most CFG compilers.

In a VSDG framework we may consider more general *code placement* optimizations instead—specifically, including the construction of a set of program points as well as placing computations at them. However, these may be applied to the same purposes. To paraphrase Weise et al. [WCES94], such code placement optimizations are decided upon when the control dependence graph (CDG) is constructed; since CDG construction, and implementation, is one of the main challenges in the conversion from VSDG to df-PDG, we can see code placement as one of the primary optimizations performed.

Hence, we see code motion as directly comparable in effect to df-PDG construction; at this juncture, the representation specifies both the size of the output, and the amount of node evaluation and predicate testing that will be performed, in an abstract (machine-independent) 3-address code form.

Moreover, the definitions of optimality given by Knoop and Rüthing for both PRE [KRS94a] and PDCE [KRS94b] correspond closely to Upton's definition of the optimality of VSDG sequentialization (given in Section 3.1). Ruthing et al. [KRS94b] write:

> "Let $\mathbf{P}(\mathbf{s},\mathbf{e})$ be the set of all paths from CFG entry to exit node, let $\mathcal{AP}$ be the set of all assignment patterns `x:=e`, and let $\mathcal{G}$ be the universe of programs that can be produced *[see below]*.
>
> 1. Let $G', G'' \in \mathcal{G}$. Then G' is better[4] than G'' if and only if
>
> $$\forall p \in \mathbf{P}(\mathbf{s},\mathbf{e}).\forall \alpha \in \mathcal{AP}.\alpha\#(p_{G'}) \leq \alpha\#(p_{G''})$$
>
> where $\alpha\#(p_{G'})$ and $\alpha\#(p_{G''})$ denote the number of occurrences of the assignment pattern α on p in G' and G'', respectively. [A footnote states: "Remember that the branching structure is preserved. Hence, starting from a path in G, we can easily identify corresponding paths in G' and G''."]
>
> 2. $G^* \in \mathcal{G}$ is *optimal* if and only if G^* is better than any other program in $\mathcal{G}$."

Critically, exactly the same metric is applied for both PRE and PDCE; the difference between them is $\mathcal{G}$, the set of possible transformations considered:

For PDCE: Let $\mathcal{G}$ be the universe of programs that can be produced by any sequence of admissible assignment sinkings and dead code eliminations to the original program G.

For PRE: Let $\mathcal{CM}_{\text{Adm}}$ be the universe of admissible (safe and correct) code motion transformations.

The latter rests on Definition 4.1 above, of code motion being a transformation solely for reducing unnecessary recomputation; however we can see both as instances of the same transformational framework of moving computations between CFG program points.

[4] Both papers point out that "at least as good as" would be a more accurate, but uglier, phrasing.

Thus, Upton's criteria of not performing redundant computation subsumes both of these, with computation of partially-dead expressions and (partially or totally) redundant computation corresponding to speculative and repeated evaluation, and being ruled out by PDCE and PRE respectively. However, this correspondence also highlights a number of differences, discussed in subsections 4.4.1 to 4.4.4.

4.4.1 Suitable Program Points

These definitions of PRE and PDCE optimality interpret "suitable program points" in Definition 4.1 as meaning basic blocks existing in the input CFG[5], and consider only how many evaluations might be avoided given *the set of paths in the existing CFG*. Under that constraint, Knoop and Ruthing [KRS94a] identify and reach the point of optimality, i.e. where no further redundancy can be eliminated. However, elimination of redundancy is restricted because good locations for placing computations do not necessarily exist: Bodik and Gupta [BG97] show that 30-50% of (partial) redundancy in programs cannot be eliminated by classical code motion alone. In such cases, the CFG—both as input and as output—must specify one or more locations, good or otherwise.

In contrast, in a VSDG framework, the previously existing set of program points (i.e. CFG locations) is deliberately discarded, and optimality concerns what is possible with *any set of paths* that might be created. Thus, a key challenge of sequentialization is to create a "good" set of program points, and paths between them, which allow *all* redundant computation to be avoided. Such paths may not have existed in the input CFG.

4.4.2 Extra Tests and Branches

A key result of the *creation* of a set of program points in a VSDG framework is that the dynamic overhead of tests and branches used to control flow along the created paths may vary substantially between CFGs. We saw this in Section 4.3.2: Upton's VSDG-based compiler can produce output performing *multiple* tests and branches on the predicate of a single γ-node. In contrast, in classical code motion the number of tests is fixed as it is a property of the input CFG (which is unchanged), and we chose in Section 3.1 an explicit policy of dynamically performing only one test per γ-node. Hence, although Upton's definition of optimality parallels the CFG definitions, we see the lack of account of this overhead as a significant shortcoming of his technique. (However, due to Upton's use of exponential time BDD techniques, his algorithm is also capable of *combining* tests for multiple γ-nodes with the same predicate; we consider this further in Chapter 7).

4.4.3 Program Points and Sequentialization Phases

A program point in the CFG embodies two properties that we see as distinct:

1. The set of conditions under which control flow reaches that point in the CFG—that is, an encapsulation of *whether* to execute statements;

2. An ordering, relative to the various other computations (single statements or otherwise)

[5] More precisely, existing in the input CFG after augmentation with an extra node at the midpoint of any *critical edge*—an edge from a node with multiple successors to a node with multiple predecessors. Some authors instead place computations directly on CFG *edges* [DS93].

and program points executed under the same conditions—that is, an encapsulation of *when* to execute statements.

In our framework, the choice of *ordering* is performed entirely by the node scheduling phase, considered in Chapter 6. However, the definitions of optimality we have seen—for PRE, PDCE and VSDG sequentialization—do not depend on ordering, and thus such optimality is decided entirely by df-PDG construction.

Another difference is that while many interpretations of code motion refer to motion *between* basic blocks, they perform local *intra-block* transformations (such as deleting multiple occurrences of the same expression) as a preliminary step, passing onto the subsequent *inter-block* stage only information about upward- and downward-exposed occurrences. Further, CFG algorithms typically consider "an expression" as being the RHS of a single 3-address-code instruction, and thus cannot as presented deal effectively with large expressions spread over more than one basic block, e.g. involving conditional expressions.

Contrastingly, in the VSDG, the uniformity of expressions (Section 2.1.1) means that motion *within* basic blocks is presented in the same way as motion *between* them, and expressions spanning multiple basic blocks are treated the same way as others, irrespective of size.

4.4.4 Lifetime Minimization

An additional consideration for CFG code motion algorithms has been the positioning of computations to reduce variable lifetimes—Knoop and Steffen's OCM [KRS94a] succeeds in producing the shortest lifetimes such that optimal elimination of redundancy is satisfied. This approach is continued by many other authors. We see it as simply applying the same transformational framework, but for a different purpose to that of Definition 4.1 above, even if lifetime minimization is taken as only a tertiary aim after dynamic and static optimality.

It also forms a good example of how phase-order problems in the CFG are often addressed (Section 2.4.1): code motion often leads to an undesirable increase in register pressure, so code motion algorithms are modified to attempt to "help" register allocation as much as possible subject to optimality in their own domain.

In our framework, since lifetime minimization depends only on instruction *ordering* (Section 4.4.3), this is performed entirely by the node scheduling phase. We argue this is a better separation of concerns, as the node scheduling phase can consider actual register pressure resulting from placement of all expressions together (Johnson [JM03, Joh04] has already shown that the choice of ordering has a great effect here.) In contrast, the CFG approach of minimizing variable lifetimes is but a proxy for the actual problem, in that where register pressure is not excessive it may simply be wasted effort, and where spilling still occurs, recomputation might be cheaper than storing to memory and reloading.

Further Optimizations Similarly, we argue that many optimizations sometimes considered separate are really more general applications of some form of code motion-like transformation: these include instruction scheduling, where the distinction is already somewhat blurry due to global scheduling techniques (such as superblock scheduling [HMC+93]) moving instructions between basic blocks; combined register allocation techniques, where even local algorithms sensitive to register pressure might insert recomputations to avoid spilling; and many others.

The attempt to exploit the unconstrained nature of code placement in the VSDG can be seen as a unifying theme of this thesis.

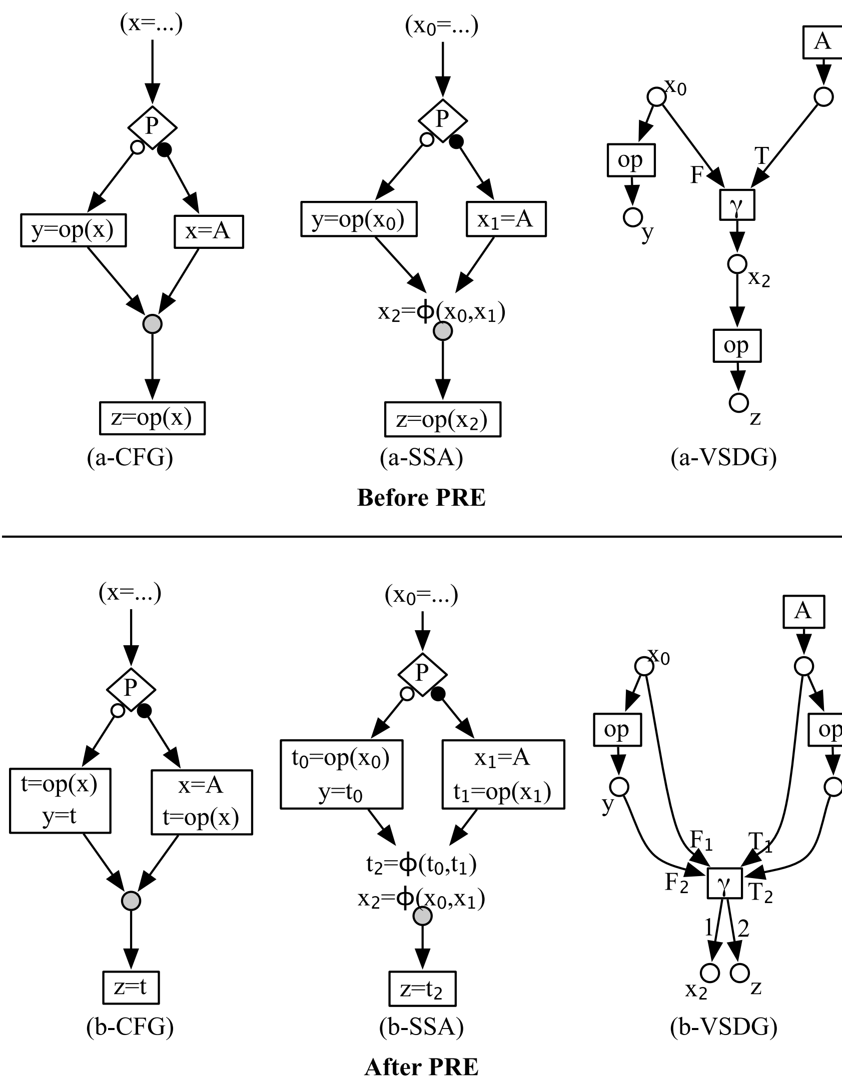


Figure 4.12: Partial Redundancy Elimination on SSA form must deal with textually different expressions, but is handled by a simple node distribution transformation in the VSDG

4.4.5 Variable Naming and Textually Identical Expressions

In the CFG, where there are multiple assignments to the same variable, textually identical 3-address instructions at different locations might be computing entirely different *values*. Contrastingly, in SSA form (and hence also the VSDG), use of variable names (nodes) to identify specific values means that that unrelated expressions have different textual representations (are different nodes).

However, when a variable is assigned to on only one arm of a conditional, following the merge point the value in that variable *may* be the same as before, even though in SSA form it now has a different name. This is shown in Figure 4.12(a). These issues with variable naming cause significant complications when performing code motion directly on SSA form: the SSAPRE algorithm [CCK+97] must trace the flow of values through ϕ-nodes. (The result is shown in Figure 4.12(b).)

In the VSDG, redundancy in these examples is not handled by our proceduralization technique. However, using the VDG, Weise et al. [WCES94] show how a node distribution transformation captures these examples much more simply than on the CFG (in SSA form or otherwise): operations after the merge point are "pushed" upwards through γ-nodes until they can be hash-consed with existing nodes. This effect is shown in Figure 4.12(a-VSDG) and (b-VSDG): note the new *component* (Section 2.6.2) of the γ-node, producing the same value for z but with fewer computations on some paths.

CFG techniques additionally suffer from complications due to the interactions between expressions and assignments—known as *second order effects* [KRS95]. Specifically, motion of expressions is often blocked by assignments to expression operands; if motion of the assignments (i.e. `x:=e` for original program variables `x`) is allowed, rather than just computations into introduced temporaries (i.e. `t:=e`, leaving `x:=t` alone), substantially more optimization is possible than otherwise [KRS95]. However, the VSDG makes assignment transparent, so the same level of optimization is naturally achieved without considering such interactions. We conjecture a similar outcome could be achieved in SSA form by interleaving rounds of SSAPRE [CCK+97] with copy propagation.

4.4.6 Optimal Code Motion on Running Examples

The *Lazy Code Motion* algorithm of Knoop, Ruthing and Steffen [KRS94a] is perhaps the *de facto* standard, and develops on the *Busy Code Motion* algorithm of Morel and Renvoise [MR79]. This algorithm deals with the elimination of partial redundancies—computations of an expression which is available on some but not all predecessors—but they separately apply similar techniques to the elimination of partially dead expressions [KRS94b]. Here it is useful to consider again the running examples of Figures 4.1 and 4.4.

In the first of these, OCM [KRS94a] converts or normalizes both input CFGs to output CFG (c), the same as Upton's algorithm[6]; similarly, both CFGs of Figure 4.10 are normalized to output CFG (c), which is as good as the VSDG.

However, on the independent redundancy of Figure 4.4, these techniques are not capable of converting between any of the CFGs shown, and certainly would not reach a CFG as efficient as our result of Figure 4.5 as this requires the creation of new program points. (A development of PRE known as *Speculative PRE* [SHK04] is capable of converting between Figure 4.4(a) and (b) when given runtime profiling information which showed one form to involve fewer total evaluations than another. This possibility is discussed in Chapter 7 along with competing algorithms which restructure the CFG.)

4.4.7 Other Algorithms

A large number of other techniques for performing code motion directly on the CFG have been described in the literature.

Chow et al. develop an algorithm equivalent to Knoop's OCM operating directly on SSA form [CCK+97].

Briggs and Cooper [BC94] describe a number of *enabling transformations* which expose additional opportunities for PRE by global value numbering [RWZ88] and expression reasso-

[6] It is possible that Figure 4.1(c) could be converted to the shortcircuit version of Figure 4.3 by the CFG cross-jumping optimization, but this is only opportunistic.

ciation. Although their technique can worsen execution times by sometimes sinking code into loops, we believe similar techniques could be straightforwardly applied to the VSDG if considered desirable.

In further work, Ruthing, Knoop and Steffen [RKS00] consider code size as an additional criterion, on top of speed and variable lifetimes, for controlling the application of code motion transformations. Specifically, they consider searching for the *smallest* transformation satisfying computational optimality, or the fastest solution among the smallest transformations at least preserving program performance. This allows some partial redundancy to remain in the program if this allows a smaller solution. On the VSDG, our proceduralization technique always searches for the smallest transformation satisfying computational optimality, but the idea of performance preservation suggests space-conscious heuristics for controlling proceduralization as well as PRE by node distribution (covered in Section 4.4.5).

CHAPTER 5

Intermediate Representations and Sharing

Previously, we compared the PDG with the VSDG, observing that the VSDG's non-specification of evaluation strategy (encoded in the PDG's Control Dependence sub-Graph) made it both more abstract and more normalizing—desirable properties for an intermediate representation. Implicit in this comparison, but never directly stated, was the idea that the VSDG was "better" than the PDG. This suggestion requires that anything that can be expressed in the PDG can also be expressed in the VSDG—but in fact we have already seen in Figure 3.10(iii) (in Section 3.4.1) examples where this is not the case.

This Chapter focusses on refining the VSDG to accommodate these structures, and studies the issues underpinning this, which relate to the *sharing* of computations. Specifically,

- Section 5.1 exhibits and explains exactly what structures can be expressed in the PDG but not the VSDG;
- Section 5.2 shows how *all* specification of sharing can be separated out from the VSDG into explicit *equivalence classes* or *partitions*, leaving behind a special case of *Unshared VSDG* (u-VSDG).
  - Section 5.3 shows how such a partitioning identifies a unique PDG.
  - This yields new insights into how loop optimizations fit into a VSDG compiler, discussed in Section 5.4.
- Lastly, Section 5.5 defines a variant of the VSDG called the *Regionalized VSDG* (RVSDG), which is equivalent to the PDG, easing use of VSDG optimizations and techniques after proceduralization.

Overview of Representations This Chapter defines several new data structures which are new ways of representing the same information as existing IRs. These are shown in Figure 5.1. Specifically, the u-VSDG is a special case of and equivalent to the VSDG; and both the RVSDG, and the u-VSDG with explicit partitions, are equivalent to the PDG. (By equivalent, we mean

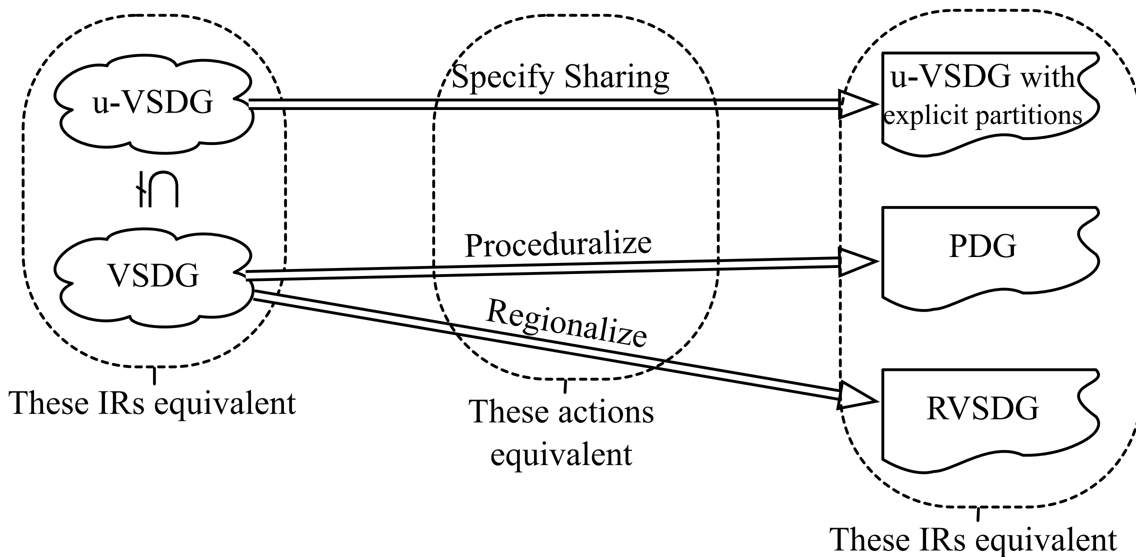


Figure 5.1: Overview of Intermediate Representations and their Relationships

as specifications of evaluation behaviour, i.e. they have the same trace semantics as discussed in Section 2.7.1.)

5.1 Shared Operators: the PDG & VSDG, Part 2

In our review of the PDG and VSDG (Section 2.3), we observed that the VSDG describes the *values* (and states) occurring in the program, and the operations required to produce them. Operators explicitly identify their operands, with the statements required to perform said operations being somewhat implicit. In contrast, the PDG describes the program explicitly in terms of the statements performing those operations. Operands are specified in terms of *variables*, with the actual values present in said variables at the point when the statement is executed being somewhat implicit.

This difference allows a PDG operation (statement) node, called a *shared operator*, to perform a single operation on one of multiple different arguments—where there is no unique node or statement producing the argument value. Such PDGs are exemplified by that of Figure 5.2(a), in which the shared operator $\mathtt{r1} = \mathtt{op}(\mathtt{t1})$ operates on either X or Y (potentially expressions). The choice between X and Y is made earlier in the program; there is nothing akin to a γ-node in the VSDG which can be identified as being the source of the value in t1. Yet a single copy of op is all that is required in the CFG too (Figure 5.2(a-CFG)).

Even in SSA form, with a ϕ-function of form $\mathtt{t1} = \phi(\mathsf{rX}, \mathsf{rY})$ (as shown in Figure 5.2(a-SSA)), there is no equivalent to a γ-node: although the ϕ-function identifies the variables which might be operated on, it is not a function in the strict mathematical sense, merely a pseudo-function which can exist only after a control-flow merge. The choice between its operands is made at an unidentified point earlier in the program, and the ϕ will need to be implemented by *two* register moves *attached to its predecessors*.

PDGs with operator sharing are useful, as they represent programs statically smaller than others producing the same results, such as the two larger programs shown in Figure 5.2(b) and

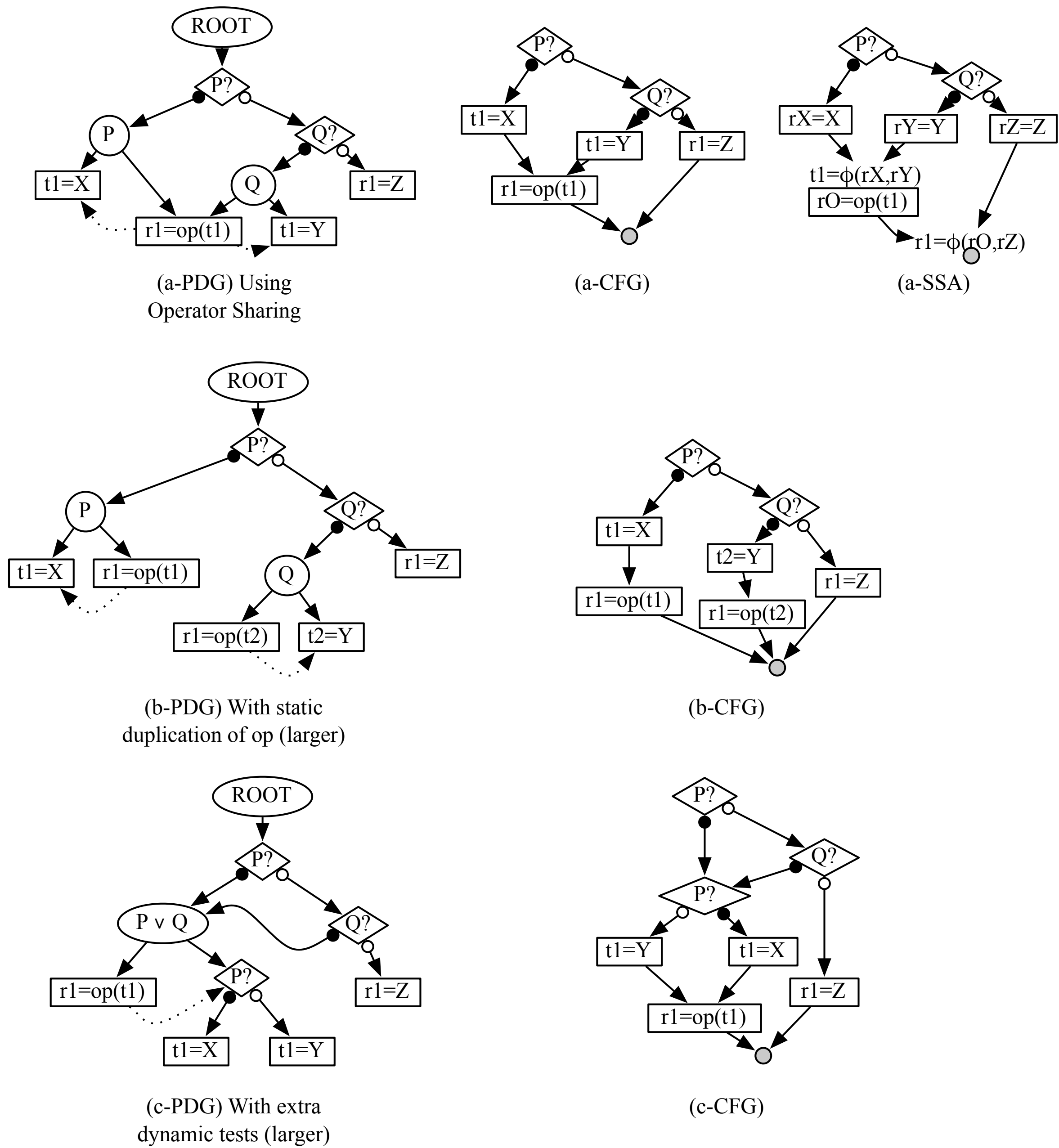


Figure 5.2: Operator sharing and the alternatives

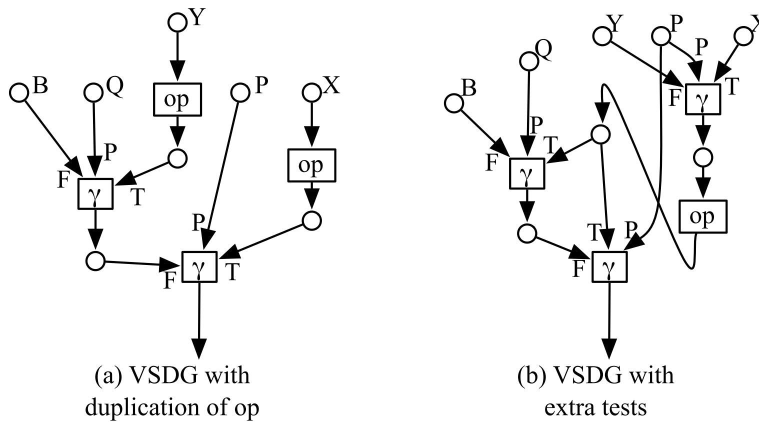


Figure 5.3: Attempts to represent Figure 5.2(a) as a VSDG

(c). Yet there is no way to specify the structure in Figure 5.2(a) using the VSDG as presented so far (both in Chapter 2, and by previous authors [Joh04, Upt06, WCES94]), and this is a drawback of these formulations. Only the larger PDGs 5.2(b) and (c) have corresponding VSDGs, namely Figure 5.3(a) and (b), respectively. Note how the extra γ-node in Figure 5.3(b) is effectively a translation of the SSA ϕ-function $\mathsf{t1} = \phi(\mathsf{rX}, \mathsf{rY})$ shown in Figure 5.2(a-SSA), yet sequentializes to (c-CFG) which performs extra runtime tests[1].

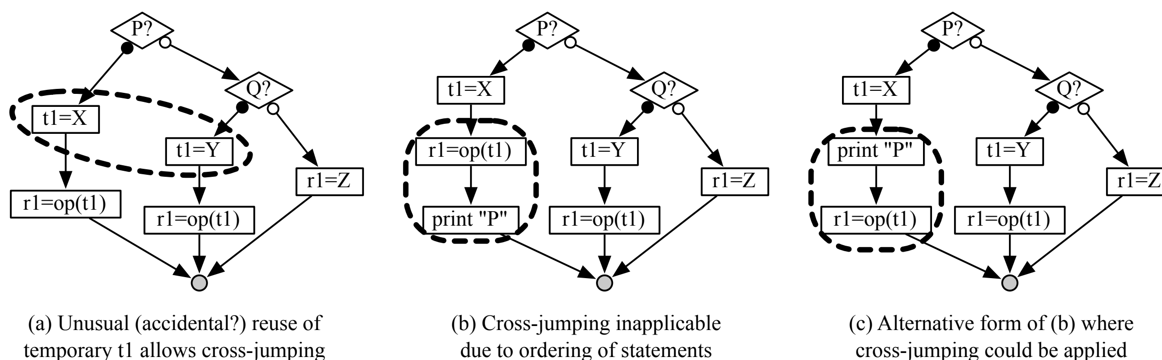
We can see the reason for this in the Haskell translation of the VSDG introduced in Section 2.2.2. Every variable defined by this translation is of ground (non-function) type. For example, applying the scheme to VSDG 5.3(a), with duplication of op, we have:

```
let ny=Y in
let nx=X in
let nz=Z in
let oy=op(ny) in
let op=op(nx) in
let gq=if Q then oy else nz in
let gp=if P then ox else gq in
gp;
```

Whereas if we consider what a translation, of a VSDG in which op was somehow shared, would look like, it would have to be something like:

```
let ny=Y in
let nx=X in
let nz=Z in
let o(arg)=op(arg) in //equiv: let o=\arg -> op(arg) in
let gq=if Q then o(y) else nz in
let gp=if P then o(x) else gq in
gp;
```

[1] Alternatively, sequentialization might produce Figure 5.2(b-CFG) even from this VSDG, if the extra γ-node were duplicated—by the PDG sequentialization algorithm of Ferrante et al. (e.g. on a more complicated example) or by deliberate application of the splitting techniques of Chapter 7.



(a) Unusual (accidental?) reuse of temporary t1 allows cross-jumping

(b) Cross-jumping inapplicable due to ordering of statements

(c) Alternative form of (b) where cross-jumping could be applied

Figure 5.4: The CFG cross-jumping optimization is highly opportunistic

and clearly this cannot be produced from any VSDG: the variable o *is of function type*. Correspondingly, the hypothetical single shared instance of op is being used as a function, i.e. *is not a value*—and the VSDG describes programs in terms of values. So it is no surprise that this structure cannot be represented in the VSDG.

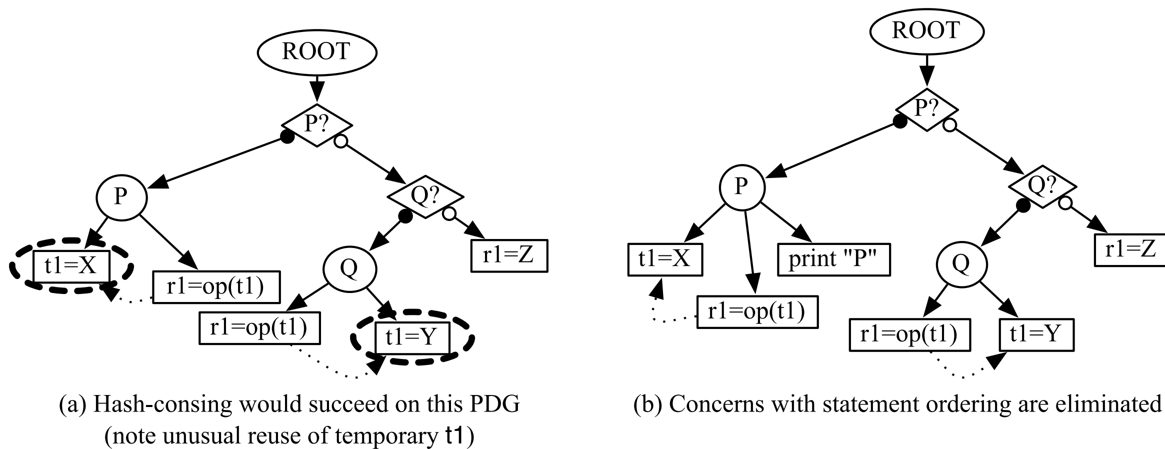
This problem was encountered by Weise et al. in their original approach using the VDG [WCES94]. Construction of a VDG from a CFG such as that of Figure 5.2(b) would proceed in two stages. First, λ-nodes (effectively, separate VDGs) and explicit call nodes would be used in order to share op by making a function call with return address passed as an extra parameter. Second, the λ-γ transform would combine all call sites together into a single one by using extra γ-nodes (as we have seen); the called λ would then be inlined, producing the VSDG of Figure 5.3(b). However, no technique for sequentializing such structures was suggested; we posit that either extra tests or duplication of op would result.

We will address this problem with an explicit sharing relation which represents both sharing of operations (transitions) and pre-colouring of values (places).

5.1.1 Production of Shared Operators

It is worth considering at this point how such PDGs might be produced—or rather, how in practice these are a PDG "feature" unlikely to be exploited by compilers. There is no easy way to express the resulting CFG structure (Figure 5.2(a-CFG)) in, for example, the C or Java languages (it requires the use of `goto` statements or complicated arrangements of labelled `breaks`); it is likely to be produced only by a compiler optimization. One way would be by application of the cross-jumping optimization on a CFG exemplified by Figure 5.4(a): the CFG contains two identical nodes, both labelled with the same statement $\mathrm{r1} = \mathrm{op(t1)}$ and both with the same successor; thus, they can be combined to give the desired CFG 5.2(b). However, this is an opportunistic *peephole optimization*, for two reasons:

- It depends on the *argument* to both statements already being in the same register. Most compilers use the *policy* of using a fresh virtual register each time a new register is required (a *requirement* in SSA form, even for user variables). Such would result in two *different* statements using different argument registers, as shown in Figure 5.2(b-CFG), and cross-jumping could *not* optimize this CFG into the desired form of Figure 5.2(a-CFG). Thus, most compilers perform cross-jumping *after* register allocation (i.e. on statements



(a) Hash-consing would succeed on this PDG (note unusual reuse of temporary `t1`)

(b) Concerns with statement ordering are eliminated

Figure 5.5: Cross-jumping on the PDG also seems only opportunistic

expressed using physical registers), making success dependent on the whims of the register allocator.

- It requires both of the identical statements $\mathtt{r1} = \mathtt{op(t1)}$ to be ordered after any other (not otherwise significant) statements that may be present. For example Figure 5.4(b) and (c) show two equivalent CFGs with different instruction orderings; the peephole optimization would succeed on CFG (c) but not (b).

Optimization directly on the PDG is more promising than on the CFG but still leaves much to be desired. Problems of ordering CFG statements are eliminated: there is only one PDG corresponding to both CFG Figure 5.4(b) and (c), shown in Figure 5.5(b). This could be optimized to share $\mathtt{r1} = \mathtt{op(t1)}$ by *hash-consing* identical statement nodes. However the same issues of register assignment occur as in the CFG: hash-consing would not be possible on the PDG of Figure 5.2(b-PDG), although it would on PDG 5.5(a) (equivalent to the CFG of Figure 5.4(a)) with its serendipitous reuse of argument register `t1`. One might conjecture a more aggressive PDG transformation of combining identical operations with mutually-exclusive control dependences by automatically coalescing their argument registers—perhaps whenever both could execute last among their siblings—but to the author's knowledge no such technique has been presented; thus, the optimization of PDGs to introduce such sharing seems limited to opportunistic techniques in a similar fashion to CFGs. We argue this ability to express, as PDGs, programs that are unlikely to be produced, from PDGs, makes the PDG less suited to being the main IR, and better as an intermediate *target* for translation from another IR.

5.2 An Explicit Specification of Sharing

In Section 2.7.3 we described the *hash-consing* and *node cloning* transformations on the VSDG (shown in Figure 2.10). Recall that these preserve the semantics (trace and observable) of the original VSDG.

Now consider a VSDG to which a maximal degree of node cloning has been applied[2]: every

[2]This first requires infinitely flattening any loops, producing an infinite net; these additional complications, and their benefits, are considered in Section 5.4.

transition node has been cloned, to make a distinct copy for every use[3], and no node has more than one outgoing edge. (That is, $s^\bullet$ and $t^{\bullet\bullet}$ are always singletons[4].)

We call such an entirely treelike VSDG an *Unshared VSDG* (u-VSDG), and see it as making explicit the maximum set of *dynamic computations* which might be performed by any output program. We now show how to explicitly specify how these dynamic computations are to be shared between the static instructions in the output program that is selected; this will include *operator sharing* as in Section 5.1 above.

Representation of sharing will be as an explicit partition, depicted graphically by *sharing edges* and interpreted as their reflexive symmetric transitive closure. The partition is an equivalence relation $\leftrightsquigarrow \subseteq S \times S \cup T \times T$ which relates u-VSDG nodes which will share the same static representation. Specifically, $t_1 \leftrightsquigarrow t_2$ means the two dynamic computations are performed by the same static instruction; $s_1 \leftrightsquigarrow s_2$ when the two values are stored in the same register (thus, such edges act as *coalescing* constraints on the register allocator). We write $[n]$ for the equivalence class of n, that is $\{n' \mid n \leftrightsquigarrow n'\}$; observe $n \leftrightsquigarrow n' \Leftrightarrow [n] = [n']$. Further, we write $\wr N \int = \bigcup_{n \in N} [n]$ to lift equivalence classes over sets. An example is given in Figure 5.6 below.

Two distinct kinds of sharing, which we call *reuse sharing* and *tail sharing*, are important, explained below; intuitively, these represent sharing of computations with *identical histories* or *identical futures*, respectively, as shown in Figure 5.6. Each kind of sharing is a partition in its own right, with all sharing being the transitively-closed union of these:

- $\overset{\mathrm{r}}{\leftrightsquigarrow}$ indicates reuse-sharing, with equivalence classes denoted by $[n]^{\mathrm{r}}$, and depicted with blue sharing edges;
- $\overset{\mathrm{t}}{\leftrightsquigarrow}$ indicates tail-sharing, with equivalence classes written $[n]^{\mathrm{t}}$ and depicted with red sharing edges.

Well-Formedness Conditions The two kinds of sharing are required to be disjoint, in that two nodes are either reuse-shared, tail-shared, or neither:

$$n_1 \overset{\mathrm{t}}{\leftrightsquigarrow} n_2 \wedge n_1 \overset{\mathrm{r}}{\leftrightsquigarrow} n_2 \Leftrightarrow n_1 = n_2$$

They must also commute, as follows:

$$n_1 \overset{\mathrm{r}}{\leftrightsquigarrow} n_2 \overset{\mathrm{t}}{\leftrightsquigarrow} n_3 \Longrightarrow \exists n_2'. n_1 \overset{\mathrm{t}}{\leftrightsquigarrow} n_2' \overset{\mathrm{r}}{\leftrightsquigarrow} n_3$$

This produces a "grid pattern" of sharing, with tail-sharing edges along one axis and reuse-sharing along the other; thus, all sharing can be formulated row-wise or column-wise:

$$[n] = \wr [n]^{\mathrm{r}} \int^{\mathrm{t}} = \wr [n]^{\mathrm{t}} \int^{\mathrm{r}}$$

For transitions to be shared, we also require piecewise sharing of their operands and results:

$$t_1 \leftrightsquigarrow t_2 \Rightarrow {}^\bullet t_1 \leftrightsquigarrow {}^\bullet t_2 \wedge t_1{}^\bullet \leftrightsquigarrow t_2{}^\bullet$$

[3]To simplify the presentation we consider only stateless V(S)DGs here.

[4]For transitions t with multiple results (e.g. integer division), $t^\bullet$ can be considered a singleton by removing the unused results and using labels (e.g. quo, rem) to identify which result $t^\bullet$ represents. This was done for complex nodes in Section 2.6.5.

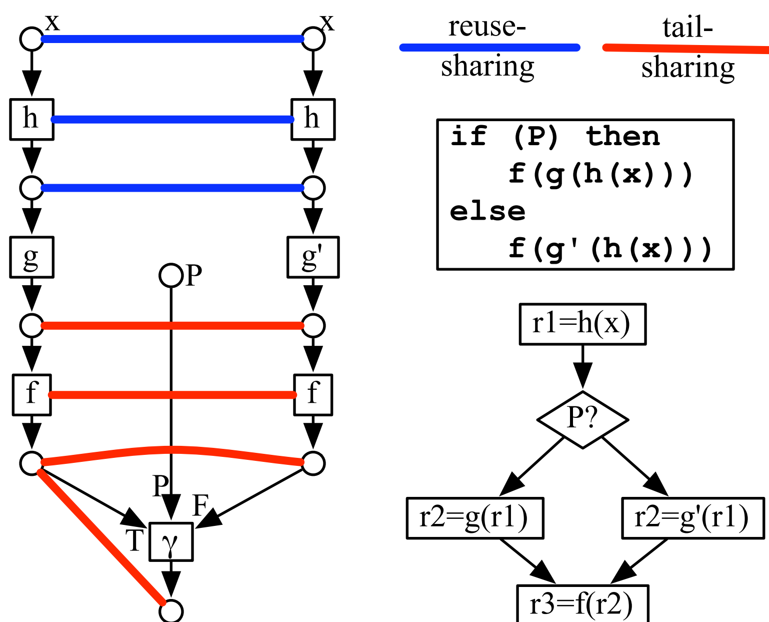


Figure 5.6: Simple example of tail- and reuse-sharing.

(This restriction is the same for both types of sharing, and ensures that the static instruction, which explicitly specifies argument and result registers, is the same for both transitions[5]; however, the two types have different effects on semantics, discussed in Section 5.3.)

We also impose further conditions for places, according to the kind of sharing, given below.

Reuse-Sharing As mentioned above, reuse-sharing captures sharing based on the past: of computations whose *operands* are the same, and of the places they produce as *these store identical values*. Thus, reuse-sharing corresponds to prior notions of hash-consing on the VSDG.

This leads to the following well-formedness condition on reuse edges between places:

> Two places may be reuse-shared *only if* they are produced by reuse-shared transitions:
>
> $$s_1 \overset{r}{\leftrightsquigarrow} s_2 \Rightarrow {}^{\circ}s_1 \overset{r}{\leftrightsquigarrow} {}^{\circ}s_2$$

Tail-Sharing Conversely, tail-sharing captures sharing based on the future: of computations whose *continuation* is the same, or of places with *the same consumers* (uses). (Continuations were discussed in Section 2.4.3; recall in particular that prior to PDG sequentialization, an operation's continuation includes only its uses, not other operations in the total order that phase imposes.) Thus, tail-sharing captures operator sharing in the PDG.

Note that an operation's continuation does not include the operation itself[6]. Thus, if two transitions have an identical continuation, this may include neither transition, and so execution

[5]In fact, we must ensure that the PDG *subtree* corresponding to each is identical—but for operation nodes, the subtree is just the single PDG statement node. However, for γ-nodes, the subtree consists of a PDG predicate node *and its children*, and this leads to additional constraints. We defer discussion of this issue until the semantics in Section 5.3.

[6]Even for loops, as each iteration contains distinct instances—considered in Section 5.4.

of the computations (also liveness of tail-shared places) must be mutually exclusive. Thus, the transitions or places must be at different *leaves* of a γ-tree (defined in Section 2.6.2). This leads to the following two well-formedness conditions on tail-sharing edges between places:

- For any γ-node g, each true or false operand $s \in {}^\bullet g$ is tail-shared with the corresponding result $s' \in g^\bullet$. (That is, recalling the view of γ-nodes as tuples of *components* $(s^{\mathsf{true}}, s^{\mathsf{false}}, s^{\mathsf{res}})$ as defined in Section 2.6.2, within each component $s^{\mathsf{true}} \overset{\mathsf{t}}{\leftrightsquigarrow} s^{\mathsf{false}} \overset{\mathsf{t}}{\leftrightsquigarrow} s^{\mathsf{res}}$.) This ensures that no register move instruction `r=t` or `r=f` must be output unless it is explicitly represented by a `MOV` transition.

- In other cases, places may be tail-shared *if and only if* they are consumed by identical tail-shared operations:

$$s_1 \overset{\mathsf{t}}{\leftrightsquigarrow} s_2 \Rightarrow {s_1}^\bullet \overset{\mathsf{t}}{\leftrightsquigarrow} {s_2}^\bullet$$

An additional complication arises when considering γ-nodes. Non-γ transitions are strict in all their arguments, and their continuation is thus the usage of the value(s) produced by the transitions. However, we make γ-nodes non-strict, and thus they cause additional operations to be executed according to the value of the γ-node predicate[7]. Thus, the continuation of (the test-and-branch corresponding to) the γ-node must *first* include execution of any such control-dependent operations, *before* moving onto the usage of the value(s) produced by the γ-node. This will lead to an additional well-formedness constraint on tail-sharing γ-nodes, which we defer until after the semantics given in Section 5.3.

5.3 A Semantics of Sharing Edges

In this section we will give meaning to the sharing partitions by defining a simple mapping from u-VSDG to PDG[8] —that is, incorporation of the sharing edges allows us to specify the same restrictions on evaluation behaviour as the PDG. We also use this equivalence to define the extra well-formedness constraint on tail-sharing between γ-nodes, mentioned in the previous paragraph.

Elements of the PDG Each equivalence class $[t]$ will correspond to a single static PDG node $N(t)$; each class $[s]$ to a single virtual register v_s. Thus, we insist that the number of equivalence classes be finite (that is, we only define the meaning of u-VSDGs with finite partitionings); the number of nodes may be finite or (as in the case of loops) infinite.

For a γ-node g, $N(g)$ is a predicate node testing v_s, where $s \in {}^\bullet g$ is g's predicate operand; for any other node t, let op be the operation performed, $\{s_1, \ldots, s_i\} = {}^\bullet t$ be its (value) operands, and $\{r_1, \ldots, r_j\} = t^\bullet$ be its value results; then, $N(t)$ is a statement node executing $(\mathsf{v}_{r_1}, \ldots, \mathsf{v}_{r_j}) := \mathsf{op}(\mathsf{v}_{s_1}, \ldots, \mathsf{v}_{s_i})$

[7]Even γ-nodes multiplexing between values present already, equivalent to v_r `= mux( P?`v^{true}`:`v^{false} `);`. For these, explicit v_g`=`v_g^{true}`;` or v_g`=`v_g^{false}`;` operations are introduced (by Chapter 3's analysis of tail registers in Section 3.4.2); on architectures such as ARM or Alpha, such statements might be implemented via conditional moves rather than control flow. An alternative scheme would be to treat *all* γ-nodes as being multiplexers; this would make them strict as other operations and avoid extra well-formedness conditions, but would be inefficient in most cases.

[8]The mapping is largely reversible, but PDG's may reuse temporaries in ways not expressible in the u-VSDG

Control Dependences The control dependences (CDG parents) of each PDG node $N(t)$ are given as follows.

Recall from the commuting restriction in Section 5.2 that $[t] = \wr [t_1]^{\mathrm{r}}, \ldots, [t_j]^{\mathrm{r}} \wr^{\mathrm{t}}$. For each such $[t_i]^{\mathrm{r}}$, $1 \leq i \leq j$, proceed forwards along the u-VSDG's dependence edges, and identify $[g_i]^{\mathrm{r}}$ as the first reuse-partition of γ-nodes encountered which satisfies both:

- Each $t' \in [t_i]^{\mathrm{r}}$ reaches a true operand of a $g' \in [g_i]^{\mathrm{r}}$ *or* each $t' \in [t_i]^{\mathrm{r}}$ reaches a false operand of a $g' \in [g_i]^{\mathrm{r}}$; *and*
- For *every* $[g_k]^{\mathrm{r}} \overset{\mathrm{t}}{\leftrightsquigarrow} [g_i]^{\mathrm{r}}$, there must be some $[t_i']^{\mathrm{r}} \overset{\mathrm{t}}{\leftrightsquigarrow} [t_i]^{\mathrm{r}}$ that reaches $[g_k]^{\mathrm{r}}$ in the same way (i.e. by its true or false operand). We refer to this as the *same-dependants* rule; intuitively, it requires that if g_i is tail-shared, t must be tail-shared *at least as much*).

A CDG edge is then added from the predicate node $N(g_i)$ to $N(t)$, labelled with true or false respectively. (If no such reuse-partition is found, a CDG edge is added from the PDG's root node.)

Data Dependences The data dependences of each $N(t)$ are obtained by taking the producer transitions of t's operands, and for each such $t' \in {}^{\circ\bullet}t$, tracing back up the CDG from $N(t')$ until a sibling n of $N(t)$ is reached; a data dependence edge from $N(t)$ to n is then added to the PDG.

Comparing Tail- and Reuse-Sharing For transitions, we noted that both types of sharing had the same well-formedness condition, whereas for places, the well-formedness conditions differed. In an interesting duality, we see that for places, both types of sharing have the same semantics—merely coalescing the corresponding virtual registers—whereas for transitions, the *semantics* differ in terms of the control dependences which we assign to the $N(t)$. Specifically:

For Reuse-sharing $N(t)$ receives a single control-dependence predecessor. The single static instruction subsumes all the $t' \in [t]^{\mathrm{r}}$ by computing its result *early*, before any such t' would execute, and storing its result in a register which preserves it for each consumer (use) to read (the register is live until every consumer has been, or definitely will not be, executed).

For Tail-sharing $N(t)$ receives multiple control-dependence predecessors. The single instruction subsumes all the $t' \in [t]^{\mathrm{t}}$ by computing its result *late*, when any such t' would have executed, and then executing the continuation common to all.

Thus, for $[t] = \wr [t_1]^{\mathrm{r}}, \ldots, [t_j]^{\mathrm{r}} \wr^{\mathrm{t}}$, the PDG node $N(t)$ has $\|[t]^{\mathrm{t}}\| = j$ control dependence predecessors, with each CDG edge corresponding to a reuse-partition $[t_i]^{\mathrm{r}}$ for $1 \leq i \leq j$.

Extra Well-Formedness Condition on Sharing Edges Note the effect of the same-dependants rule above on tail-shared γ-nodes $g_1 \overset{\mathrm{t}}{\leftrightsquigarrow} g_2$, as follows. The rule forces any $N(t)$ which would have been control dependent on only one $N(g_i)$ to instead be control-dependent on some common ancestor of $N(g_1) = N(g_2)$, and thus be *speculated*. (This ensures that the PDG *subtrees* descended from the $N(g_i)$ are identical, including all CDG children, hence allowing only a single static subtree to exist.) This speculation is the source of the extra well-formedness constraint required. Intuitively, (trees of) γ-nodes allow multiple computations to exist which

write into the same virtual register; at runtime the γ-tree will decide which such computation should execute. However, the speculation means that sometimes such computations can execute *before* the γ-tree has decided they are needed, and we must ensure that *at most one* such computation speculatively writes into any register. Thus:

> For each PDG group node G, with children $\{N(t_i)\}$, identify the set $R \subset \{N(t_i)\}$ for which the control dependence edge $G \rightarrow N(t_i)$ was added from G (rather than from some $N(g)$ descending from G, where g is a γ-node) because of the *same-dependants* rule (i.e. because $g \overset{t}{\leftrightsquigarrow} g'$ for g' which would *not* cause t_i to be executed).
>
> We require that the sets of result places $\wr t^\bullet \int$ are disjoint for each child $N(t) \in R$. (That is, no virtual register is speculatively written to by more than one PDG subtree.)

We note this definition is deficient in that the restriction is defined 'externally', by reference to the PDG produced; an 'internal' reformulation in terms of only the structure of the u-VSDG is left to future work.

5.3.1 An Alternative View of Proceduralization

In Section 2.4.1 we identified *proceduralization* as the stage of converting a parallel, functional, program expressed as a VSDG into a parallel, imperative, program expressed as a PDG—this requires specifying what runtime tests will control evaluation of what nodes (thus fixing an evaluation strategy), and specifying how values will be passed between virtual registers.

We have seen the u-VSDG is merely a special case of VSDG, to which the trace semantics of the VSDG (defined in Section 2.7.3) can be applied (ignoring any sharing partitions[9]), and a u-VSDG with sharing partitions identifies a PDG. Moreover, the (dependence) edges of the u-VSDG are the same in both cases[10]; thus, the challenge for the proceduralizer is to introduce as many sharing edges as possible, in order to produce the same result (i.e. the observable semantics of the computation tree) but at a smaller cost (fewer dynamic evaluations and/or smaller code size).

In some cases the proceduralization phase must choose between placing tail-sharing and reuse-sharing edges between the same nodes (specifically, where the nodes have both the same past *and* the same future); these can lead to different effects, as follows.

- Reuse-sharing yields a single control dependence, which can cause redundant (speculative) evaluation.
- Tail-sharing yields multiple control dependences, avoiding redundant evaluations; however, this can cause duplication during PDG-sequentialization[11].

This is a classical space-vs-time issue, exemplified by Figure 5.7: u-VSDG (b) shows a proceduralization in which both A and X are speculated, whereas u-VSDG (c) shows a combination of reuse- and tail-sharing.

[9] These might be useful to record shortcircuit evaluation in the source code, for example; exhaustive detection of opportunities for tail-sharing constitutes a form of *procedural abstraction* [SHKN76] and is hence likely to be expensive (see Johnson [Joh04] for an algorithm on the VSDG).

[10] Modulo changes due to γ-ordering.

[11] Although we think of reuse-sharing as never leading to a size increase, this is not quite the case either, due to the effect of register-pressure-sensitive node cloning in the node scheduling phase, in Chapter 6.

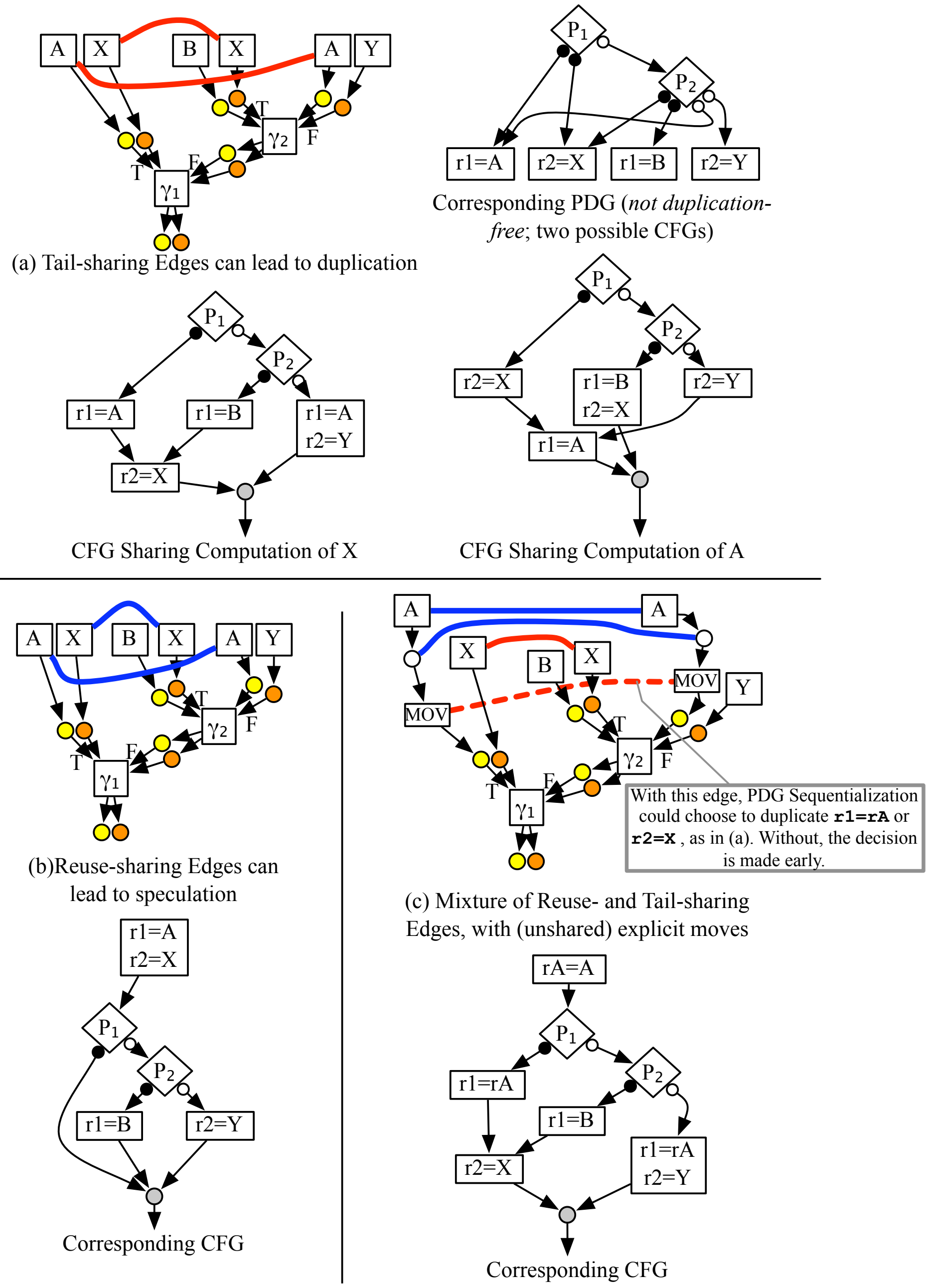


Figure 5.7: Effect of different sharing edges in proceduralizations of `if` (P_1) `then` (A, X) `else if` (P_2) `then` (B, X) `else` (A, Y). Yellow result places are all tail-shared together as `r1`; similarly orange as `r2` (neither of these occur in any of A, B, X, Y).

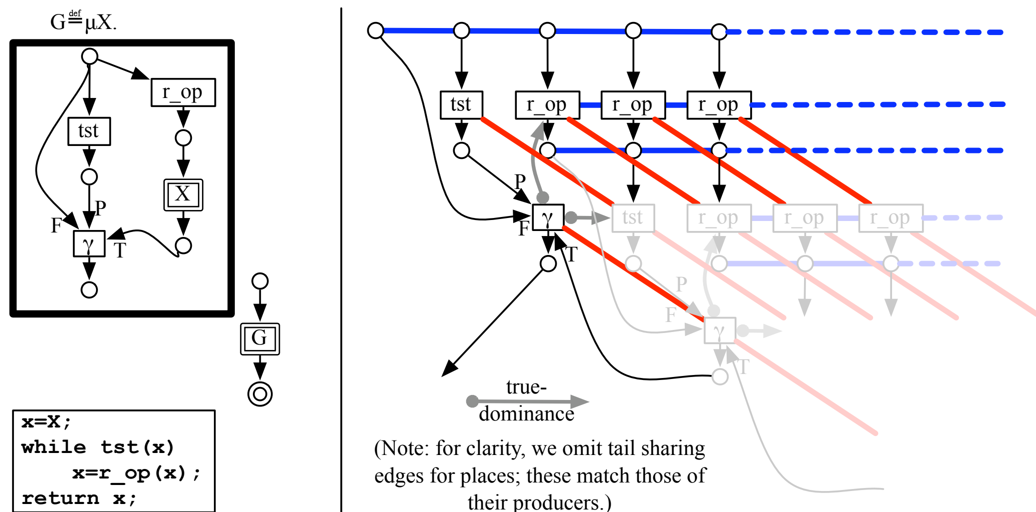


Figure 5.8: Sharing partitions for a simple loop. (Note the notation of a gray edge $g\bullet\!\!\rightarrow t$ to indicate when the `true` operand places of γ-node g together postdominate $[t]^{r}$. Although not an edge in the u-VSDG, this idea is key to establishing control dependence when converting to a PDG, described in Section 5.3.)

5.4 Loops

Viewing proceduralization as the specification of the sharing relation yields new insights as to the effect of proceduralization on loops. A loop is represented in the u-VSDG as an infinitely unfolded tree, as shown in Figure 5.8: note the *double* infinity of `r_op`s resulting from both the treelikeness of the u-VSDG (with a row of dynamic computations representing the work of an early iteration, being reuse-shared between their uses as inputs to all later iterations) and the unfolding of the loop (with the "extra" work performed by each iteration being tail-shared with that of the previous).

The sharing edges present in such an infinitely unrolled structure now specify many common loop optimizations, as follows:

Loop Unpeeling Loop unpeeling is an optimization whereby a static copy of the loop body is made outside the loop, forming the first iteration. That is, it changes `while(P) do C;` into `if(P) {C; while(P) do C;}`. In the u-VSDG, if the operations forming the first iteration are not tail-shared with those of later iterations (merely reuse-shared along a row), then the first iteration will be in its own partition, and separate code will be produced for it, before the repeating loop body is entered.

Loop Unrolling If the loop body nodes are tail-shared with their equivalents at some fixed distance of > 1 iteration, this represents an unrolled version of the loop. For example, Figure 5.9(b) shows a version of Figure 5.8 which has been unrolled by tail-sharing each operation with its equivalent at a distance of two iterations.

Software Pipelining Can be performed by *partial* unpeeling, i.e. by tail-sharing only *some* operations of the initial iteration(s); the *same dependents* rule leads to these operations and their equivalents in later iterations being computed earlier, with the unshared ones

outside the loop. For example, Figure 5.9(a) shows a different pipelining of the loop of Figure 5.8 in which one copy of the repeated operation `r_op` is speculated. Note that deciding on a level of software pipelining, including both unpeeling (speculation outside the loop) and unrolling (e.g. resulting from modulo scheduling), is equivalent to picking two cuts across the u-VSDG that are *isomorphic*: the infinite subgraph above (preceding) each cut must be of exactly the same (regular) form.

Loop-Invariant Code Motion If copies of an invariant operation `inv` are tail-shared, rather than reuse-shared, across iterations, the corresponding computation (there is still only one *statically*, i.e. [`inv`]) will be placed in the loop and performed again in each iteration[12]; if copies are reuse-shared, they will be computed once only. This is shown in Figure 5.10. The same also applies after unrolling—a computation could be entirely tail-shared (computed once per old iteration); reuse-shared within an iteration but tail-shared across iterations (computed once per iteration); entirely reuse-shared (computed once); or some hybrid.

5.4.1 Proceduralization and Loops

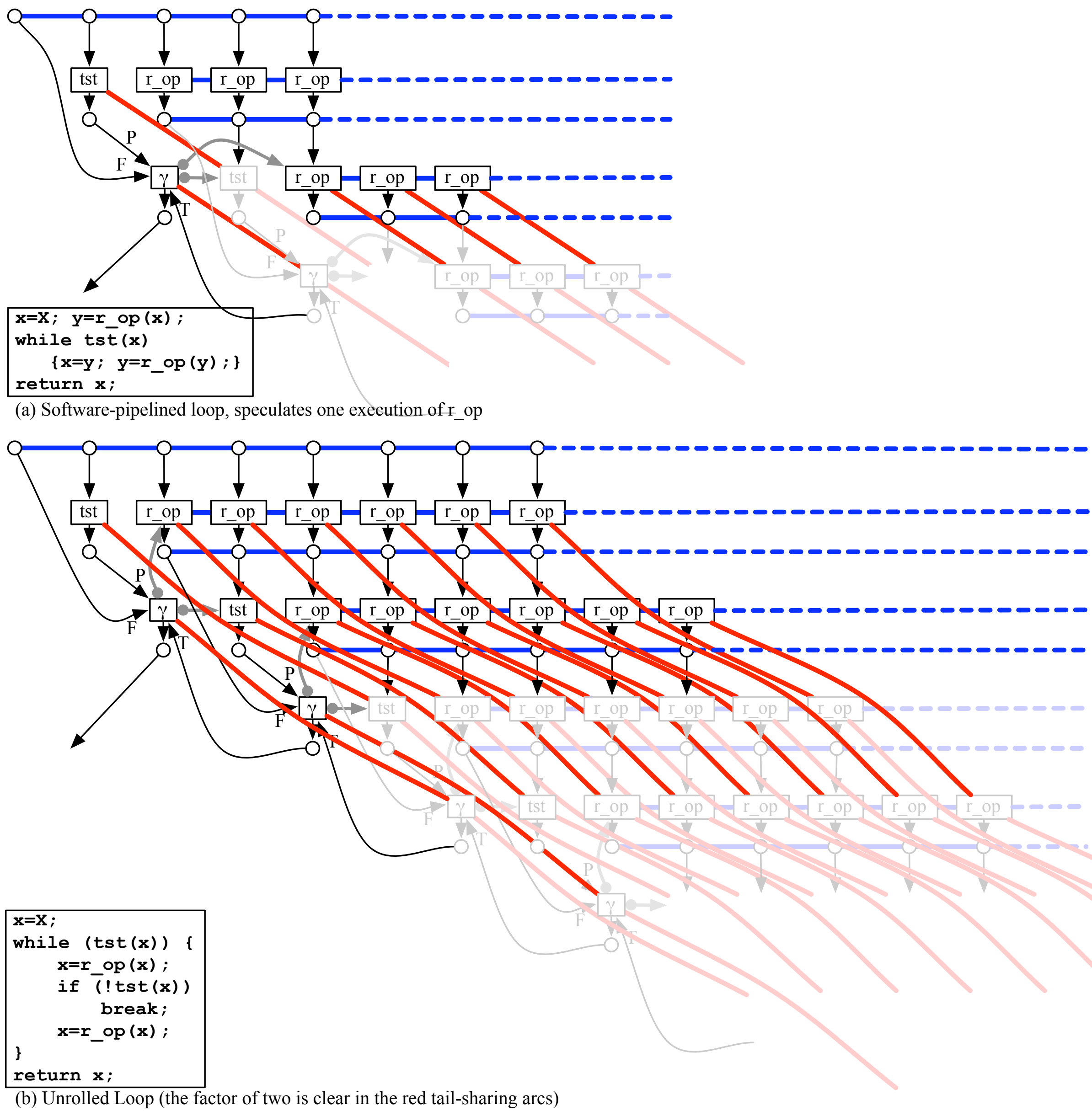
The preceding examples show that these "loop optimizations" are the same transformations as we apply to non-loop nodes, merely applied to infinite (regular) trees. Thus, the same idea of optimality of proceduralization—a machine-independent notion of the smallest program among those that perform minimum evaluation—can be applied, resulting in a fully-rolled loop with all invariants lifted and a pipelining/partial peeling which provides a safe place to compute invariants but otherwise duplicates a minimum of nodes. However, the difference between looping and linear or branchy code is that abandoning this notion of "optimality", in favour of machine-dependent targets, is perhaps more accepted practice for loops—but the VSDG gives us an alternative perspective on relaxing these optimality requirements in which loop and non-loop code are treated more uniformly. This is addressed more fully in the *node scheduling* phase in Chapter 6.

5.5 The RVSDG: a Workable PDG Alternative

The previous section has demonstrated the advantages of the u-VSDG as a unifying framework for proceduralization supporting many optimizations. However working with infinite data structures, and even finite ones after node cloning (which can lead to an exponential size increase), makes implementation awkward. Hence, in this section we give a variant on the VSDG, called the *Regionalized VSDG* or RVSDG, which is logically equivalent to the PDG or u-VSDG with partitions. Specifically, the RVSDG avoids the exponential and/or infinite blowup associated with reuse-sharing (instead represented by hash-consing) and tail-sharing between loop iterations (instead bringing the μ- operator inside the object language), and is based on the following principles:

- Each RVSDG node will represent one u-VSDG reuse-partition.

[12]Each (singleton) reuse-partition is now dominated by a different γ-node, but the pattern of dominance respects tail-sharing as enforced by the same-dependants rule (pg.102)



(a) Software-pipelined loop, speculates one execution of r_op

(b) Unrolled Loop (the factor of two is clear in the red tail-sharing arcs)

Figure 5.9: Loop Optimizations are specified by the sharing relation on the u-VSDG

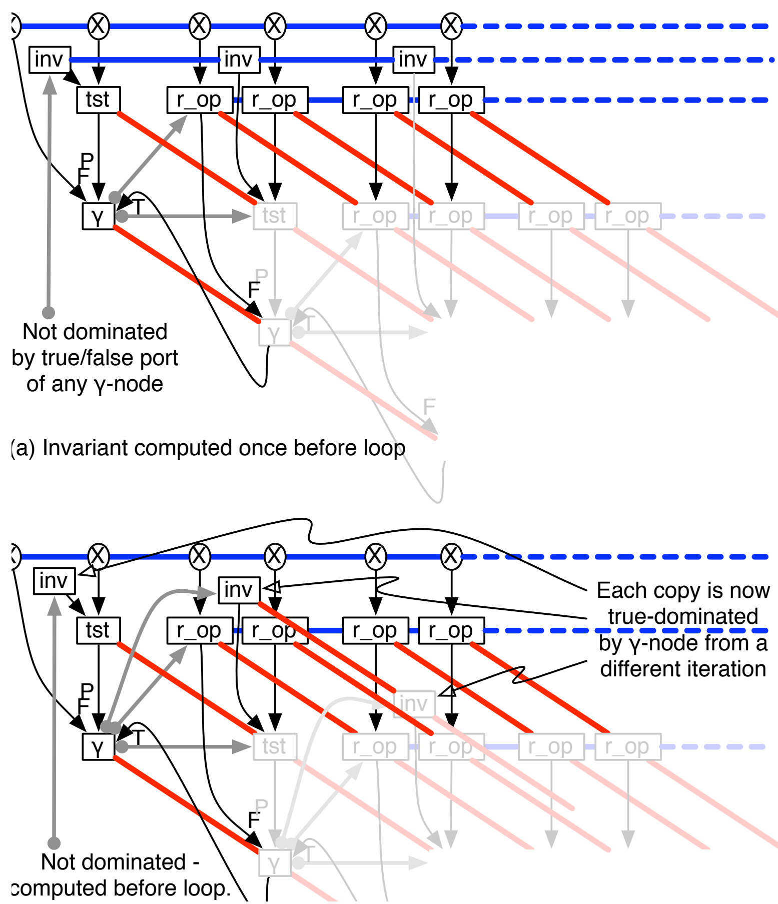


Figure 5.10: Loop Invariant Code Motion in the u-VSDG. For clarity, the single space resulting from each transition is omitted.

- Loops will be represented by `iter` nodes, which explicitly identify an enclosing (ancestor) net. This makes the location of the μ explicit as a cut, which acts as a synchronization point and becomes the loop header.
- The nodes will be divided up into sufficient hierarchy to indicate control dependence, with the control dependence of each reuse-partition explicitly represented by the containing (parent) net of each node.
- Tail-sharing will continue to be expressed by explicit partitions, but we will also allow for more general *coalescing* partitions, used for modelling operations requiring usage of particular registers, and for register allocation (in Chapter 6).

Formally, an RVSDG is a hierarchical petri-net $(S, T, E, H, S_{\text{in}}, S_{\text{out}})$ with places S (state or value), transitions T (potentially including compound nodes) satisfying $S \cap T = \emptyset$, labelled *dependence edges* $E \subseteq S \times T \cup T \times S$, *sharing partition* $H \subseteq S^{\uparrow} \times S^{\uparrow} \cup T^{\uparrow} \times T^{\uparrow}$ (this is explained below—the $^{\uparrow}$s refer to the *fully-flattened* petri-net), and input and output places $S_{\text{in}} \cup S_{\text{out}} \subseteq S$.

Dependence edges are as in the VSDG, as are operation and compound nodes, and input and output places. However, we make extra well-formedness conditions on γ-nodes, which force a finer-grain partitioning into hierarchical nets, and additionally use a system of *names* as follows. We presume there is a set of names $\mathcal{N}$, which are in 1-1 correspondence with nets contained in the hierarchy. In particular we will identify these names with loop body names, used in `iter` nodes (below), and the names of *top-level* nets we also identify with function names (used in CALL operation nodes, Section 2.6.2). We discuss these new features in turn.

5.5.1 Strict Nets and γ-Nets

The RVSDG has γ-nodes identical to those of the VSDG, that is, a single γ-node may select any number of values simultaneously. However, we require a particular structure of hierarchical nets around each γ-node, which has the effect of identifying code which need be executed only if the γ-node's predicate takes the appropriate value.

Specifically, we require that any net containing a γ-node g contains exactly two other transitions, t^{true} and t^{false} ($g \in T \Rightarrow T = \{g, t^{\mathsf{true}}, t^{\mathsf{false}}\}$), and that these are piecewise connected as follows:

$$\begin{gathered} S_{\text{in}} = {}^{\bullet}t^{\mathsf{true}} = {}^{\bullet}t^{\mathsf{false}} \wedge \\ g.\mathsf{pred} \in S_{\text{in}} \wedge \\ g.\mathsf{true} = t^{\mathsf{true}\bullet} \wedge g.\mathsf{false} = t^{\mathsf{false}\bullet} \wedge \\ g^{\bullet} = S_{\text{out}} \end{gathered}$$

We call such a net a γ-net; other nets are *strict*.

Note that if multiple nodes are conditional on the same γ-node, this forces them to be contained inside a single complex node, introducing an additional level of hierarchy.

Thus, the net within the hierarchy which contains a node, completely specifies the conditions under which the node is evaluated.

Graphically, we tend to use a simplified representation of this structure of nested regions, as shown in Figure 5.11.

Example Our solution to the "independent" redundancy, given in PDG form in Figure 3.14(e), is shown as an RVSDG in Figure 5.12.

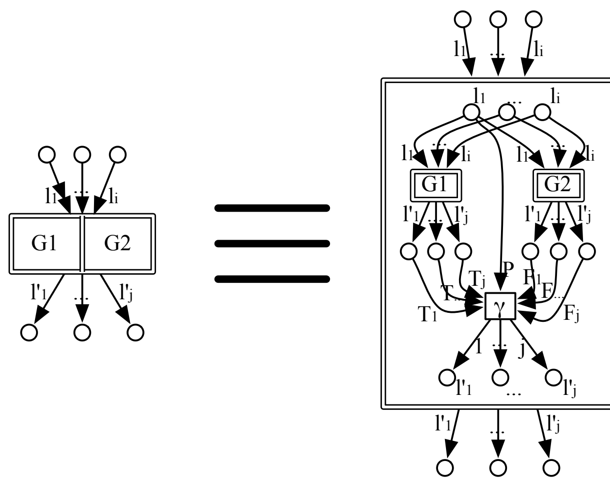


Figure 5.11: Simplified graphical notation for γ-nets: we draw the LHS as shorthand for the RHS. Note the double arrowhead is part of the shorthand notation on the LHS: it identifies which operand is used as predicate on the RHS.

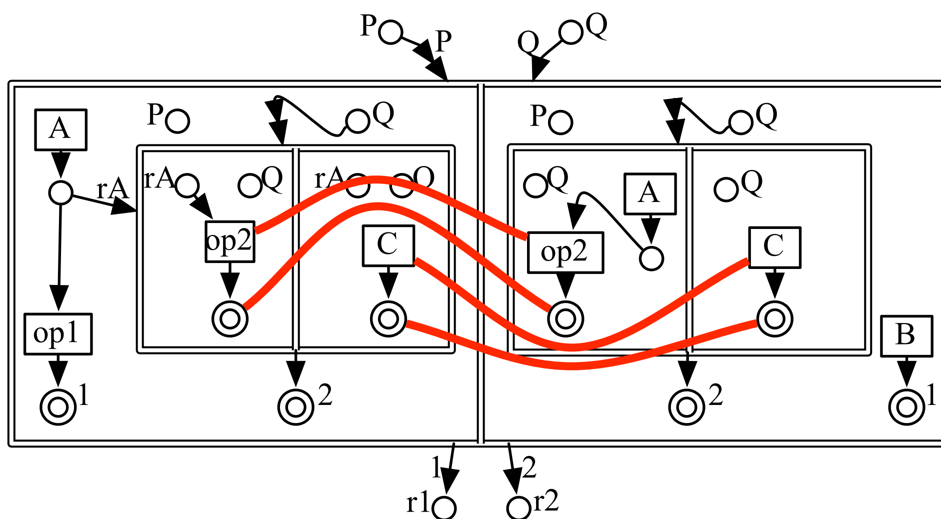


Figure 5.12: The dynamically optimal solution of Figure 3.14(e) as an RVSDG

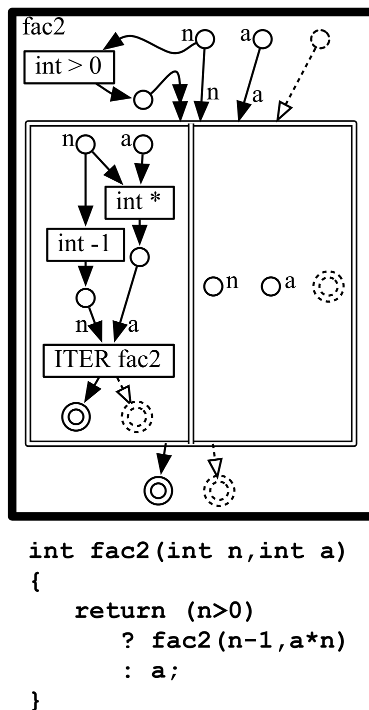


Figure 5.13: Representation of a loop in the RVSDG, using an `iter` node. (Equivalent VSDG in Figure 2.4).

5.5.2 Loops

All RVSDGs are finite, with loops instead being represented using `iter` transitions, as follows.

Each `iter` node t has a loop body name which must identify a net G hierarchically containing t. Further, t must have operands and results of *sort* matching the arguments and results of G, and indeed, these must also have matching labels (Section 2.6.4), though these are sometimes left implicit. Graphically, we show the correspondence between `iter` nodes and loop bodies by writing their names, as in Figure 5.13.

The action of the `iter` node can be seen as performing a recursive call to the named loop. However, well-formedness restrictions in Section 5.5.5 ensure that this recursive call can be implemented as a tail-call, without requiring the use of a stack; thus, execution of an `iter` node is by making an unconditional jump (without link) to the entry point of the named net, which acts as the loop header.

5.5.3 Sharing

The sharing partition in H is an equivalence relation much as in the u-VSDG (Section 5.2) for flattened graphs, again depicted as sharing edges and interpreted as their reflexive symmetric transitive closure. However, on hierarchical nets, we see the partition as being a property of the entire hierarchy, rather than of individual net instances, relating nodes in the *fully-flattened* graph. The relation for any contained or flattened graph $G' = (S', T', E', H')$ is merely that given by *restricting* the partition to only nodes within $S' \cup T'$ and simultaneously *augmenting* it to identify complex nodes whose contained graphs are shared elementwise.

The edges come in two types; these are different from the u-VSDG, in order to support requirements on the usage of particular registers (below):

Coalescing Edges run only between places, and represent a partition written with $\leftrightsquigarrow$ and $[\cdot]$, drawn with green edges. This identifies places which will be assigned the same register, regardless of use—they may identify any places that do not *have to be* simultaneously live—and thus acts merely as a constraint on the register allocator. Such partitions may be labelled with physical registers, provided the same register is not used as label more than once.

Tail-sharing Edges are as in the u-VSDG, written $\overset{\mathrm{t}}{\leftrightsquigarrow}$ or $[\cdot]^{\mathrm{t}}$ and drawn with red edges. Between places, tail-sharing edges are a subset of coalescing edges (that is, $\forall s \in S.[s]^{\mathrm{t}} \subseteq [s]$), identifying only places with the same future uses; unlike coalescing edges, they may also identify transitions.

Well-formedness constraints for both of these are given in Section 5.5.5 below.

5.5.4 Explicit Representation of Register Moves

The RVSDG explicitly represents how registers are used and how values are moved between them, in order to model non-uniform instructions and platform-specific calling conventions, as well as clashes between multiple uses of the same virtual register (e.g. `arg0`).

To this end, every net instance is seen as including one `MOV` per result to move or save the result into an appropriate register (perhaps virtual, or e.g. that required by the target architecture).

When flattening a complex node, this register move becomes represented as an explicit `MOV` transition, as follows. Suppose $t\langle(S', T', E', S'_{\text{in}}, S'_{\text{out}})\rangle \in G = (S, T, E)$ is flattened to make a new net G''. Arguments nodes in S'_{in} are *quotiented* with the corresponding operands ${}^{\bullet}t$ as per the VSDG. However, the result nodes S'_{out} are included in the flattened graph along with the $t^{\bullet}$ as separate nodes, and connected via an explicit register move instruction for each $s \in t^{\bullet}$:

$$s'^{l} \in S'_{\text{out}} \wedge t \xrightarrow{l} s \in E \Rightarrow s' \rightarrow \mathtt{MOV} \rightarrow s \in G''$$

Coalescing edges allow the RVSDG to specify that some register moves t *must* be elided, namely whenever $s \overset{\mathrm{r}}{\leftrightsquigarrow} s'$ for $\{s\} = {}^{\bullet}t$ and $\{s'\} = t^{\bullet}$. Elision of other register moves is left to the register allocator, e.g. by a preference graph mechanism or SSA techniques [HGG05, Hacng]. (Thus, an alternative model would be to see every γ-node as containing two `MOV`s per component unless these are elided.)

Platform calling conventions can now be incorporated as follows.

Firstly, for purposes of making the moves explicit, any net representing an entire function can be considered as being contained in a complex node whose operands and results are coalesced in a way representing the platform's conventions. (This is illustrated in Figure 5.14.)

Secondly, requiring each argument to (or result from) every CALL node to be coalesced with the corresponding argument (resp. result) of the top-level net, identifies where extra `MOV`'s are required (in order to satisfy the well-formedness constraints on coalescing edges, Section 5.5.5), as illustrated in Figure 5.15.

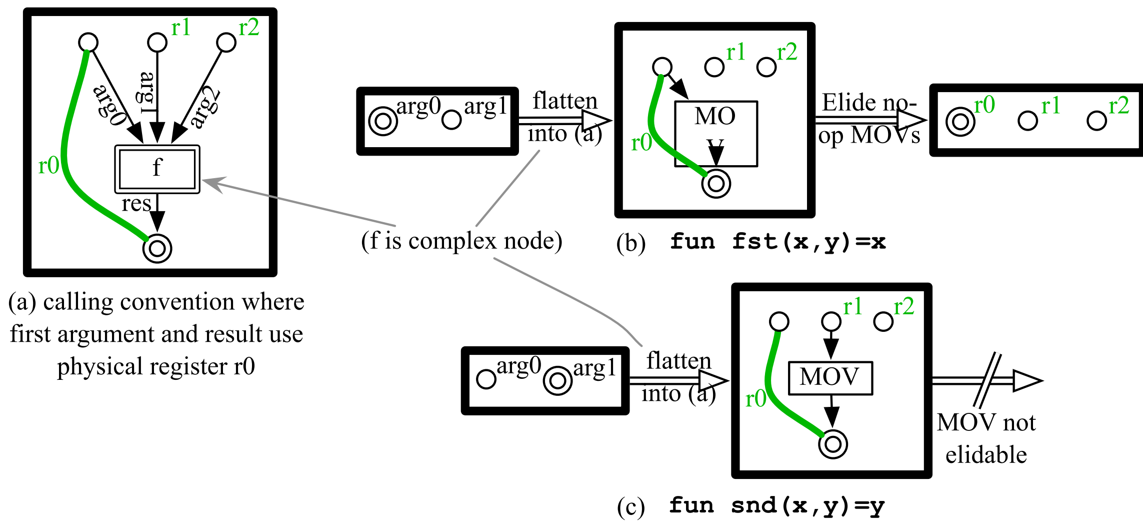


Figure 5.14: Making explicit the `MOV`s implied by procedure calling standards

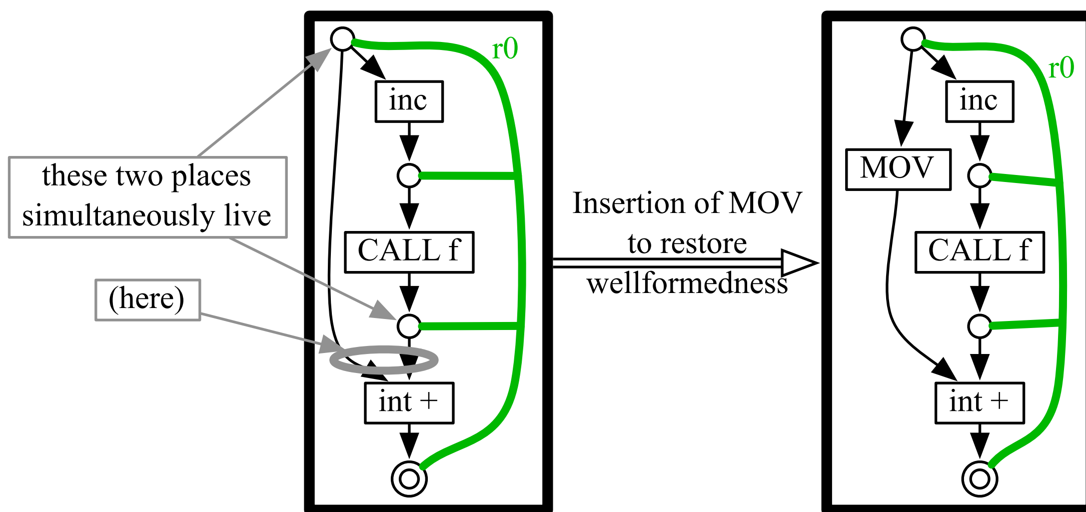


Figure 5.15: Coalescing required by procedure calling standards for `fun g(x)=x+f(x+1);`

5.5.5 Well-Formedness Conditions

We must impose a number of restrictions on RVSDGs, described below. A key concept is that of the *operand group*:

Operand Groups Intuitively, an *operand group* is a set of places whose values are demanded *at the same time*. Thus, the set of operands $^\bullet t$ to any strict transition t (i.e., any t that is not a γ-node) is an operand group, as is the set of true operands to any γ-node g, and likewise the false operands. Let $\Theta \subseteq \mathcal{P}(S)$ be the set of operand groups in a net, and let $\theta \in \Theta$ range over them.

Having defined operand groups, the restrictions are as follows:

States and Values, Acyclicity, Node Arity These are the same as the VSDG

On Coalescing We require a node scheduling of the RVSDG to exist in which no two coalesced places are simultaneously live. Node Scheduling is covered in Chapter 6, but the following constraint ensures this:

$$\forall s_1 \leftrightsquigarrow s_2 . before(s_1, s_2) \vee before(s_2, s_1)$$

where $before(s_1, s_2)$ is true if it is possible to execute all consumers of s_1 before production of s_2:

$$\begin{aligned} before(s_1, s_2) = \quad & s_1 \notin S_{\text{out}} \wedge \\ & \nexists(\theta \in \Theta). s_1 \in \theta \Rightarrow {}^\circ s_2 \to^* \theta \end{aligned}$$

On Tail-Sharing We make three requirements; these are equivalent to those on tail-sharing partitions on the u-VSDG, but the last replaces the commuting (grid-pattern) restriction of Section 5.2 due to the use of hash-consing instead of reuse-sharing edges:

- For any γ-node g, each true or false operand $s \in {}^\bullet g$ is tail-shared with the corresponding result $s' \in g^\bullet$. (That is, recalling the view of γ-nodes as tuples of *components* $(s^{\mathsf{true}}, s^{\mathsf{false}}, s^{\mathsf{res}})$ as defined in Section 2.6.2, within each component $s^{\mathsf{true}} \overset{\mathsf{t}}{\leftrightsquigarrow} s^{\mathsf{false}} \overset{\mathsf{t}}{\leftrightsquigarrow} s^{\mathsf{res}}$.) This may result in violating the above constraint on coalescing partitions; in such cases, well-formedness may be restored by insertion of extra `MOV`'s.
- In other cases, places may be tail-shared *if and only if* all the consumers of each are elementwise tail-shared:
$$s_1 \overset{\mathsf{t}}{\leftrightsquigarrow} s_2 \Rightarrow s_1{}^\bullet \overset{\mathsf{t}}{\leftrightsquigarrow} s_2{}^\bullet$$
- For tail-sharing partitions of transitions, the respective operands and results must be piecewise tail-shared (however recall that tail-sharing is reflexive: any node is considered shared with itself):

$$t_1 \overset{\mathsf{t}}{\leftrightsquigarrow} t_2 \Rightarrow t_1{}^\bullet \overset{\mathsf{t}}{\leftrightsquigarrow} t_2{}^\bullet \wedge {}^\bullet t_1 \overset{\mathsf{t}}{\leftrightsquigarrow} {}^\bullet t_2$$

On Compound Nodes We require each net in the hierarchy to have distinct nodes, and additionally require the sharing and coalescing partitions to be exactly those given by applying the following procedure to the partitions for the fully flattened graph:

- Whenever flattening causes places $s_1 \in G$ and $s_2 \in G'$ (where $t\langle G\rangle \in G'$) to be *quotiented* to become node $s \in G''$, then $\{s_1, s_2, s\} \subseteq [s]^{\mathrm{t}}$.
- Compound nodes are tail-shared whenever their contents are (piece-wise):

$$S \overset{*t}{\leftrightsquigarrow} S' \wedge S_{\text{in}} \overset{*t}{\leftrightsquigarrow} S'_{\text{in}} \wedge S_{\text{out}} \overset{*t}{\leftrightsquigarrow} S'_{\text{out}} \wedge T \overset{*t}{\leftrightsquigarrow} T' \Rightarrow$$
$$t\langle(S,T,E,S_{\text{in}},S_{\text{out}})\rangle \overset{t}{\leftrightsquigarrow} t'\langle(S',T',E',S'_{\text{in}},S'_{\text{out}})\rangle$$

- The sharing edges $H \subseteq S \times S \cup T \times T$ for each net instance G are then restricted to the places S and transitions T in G.

On Loops We make three requirements on any `iter` node t, *naming* a petri-net $G = (S, T, E, H, S_{\text{in}}, S_{\text{out}})$:

- The node must be within the net it names; that is, it must satisfy $t \Subset G$ where

$$t \Subset G \Leftrightarrow t \in G \vee t'\langle G'\rangle \in G \wedge t \Subset G'$$

- Its operands are coalesced piecewise with G's arguments: ${}^{\bullet}t \leftrightsquigarrow S_{\text{in}}$
- Its results are tail-shared piecewise with those of G: $t^{\bullet} \overset{\mathrm{t}}{\leftrightsquigarrow} S_{\text{out}}$

Note that while the requirements on loops are new in the RVSDG, in practice VSDGs constructed from iterative source codes will satisfy similar restrictions (albeit meta-syntactic ones, on the use of μ-bound variables).

Duplication-Freedom In order to use the RVSDG in place of the PDG, in particular to allow PDG sequentialization to be performed on the RVSDG (as discussed in Section 2.4 and Chapter 4), it is helpful to reformulate the condition of duplication-freedom on the RVSDG. Here the concept of operand groups (above) is again helpful. From the well-formedness conditions above, observe first that for any two tail-shared transitions, their results are necessarily tail-shared:

$$t_1 \overset{\mathrm{t}}{\leftrightsquigarrow} t_2 \Rightarrow {t_1}^{\bullet} \overset{\mathrm{t}}{\leftrightsquigarrow} {t_2}^{\bullet}$$

and secondly that if *any* place in an operand group is tail-shared with a place in another such group, the entire group must be piecewise tail-shared:

$$\theta_1 \ni s_1 \overset{\mathrm{t}}{\leftrightsquigarrow} s_2 \in \theta_2 \Rightarrow \theta_1 \overset{\mathrm{t}}{\leftrightsquigarrow} \theta_2$$

Thus, for any operand group θ written to by a transition t (i.e. $\theta \cap t^{\bullet} \neq \emptyset$), all transitions $t' \overset{\mathrm{t}}{\leftrightsquigarrow} t$ tail-shared with t must write to the corresponding element of an operand group $\theta' \overset{\mathrm{t}}{\leftrightsquigarrow} \theta$. In other words, any operand group containing a result of a transition t is tail-shared at least as much as t.

Intuitively, an operand group corresponds to a set of PDG nodes which must be executed together, i.e. a group node. This allows us to define a condition equivalent to duplication-freedom, as follows:

For a transition t, let $\mathcal{O}(t) \subseteq \Theta$ be the set of operand groups written to by t, i.e.

$$\mathcal{O}(t) = \{\theta \in \Theta \mid t^{\bullet} \cap {}^{\bullet}\theta \neq \emptyset\}$$

Duplication-freedom requires that for any operand group $\theta \in \Theta$, it must be possible to order the transitions $\{t_1, \ldots, t_j\} = {}^{\bullet}\theta$ producing its elements, such that

$$\mathcal{O}(t_1) \subseteq \ldots \subseteq \mathcal{O}(t_j)$$

An alternative formulation is that every pair of producing transitions must write to partitions of operand groups related by $\subseteq$, that is:

$$\forall \theta \in \Theta. \forall t_1, t_2 \in {}^{\bullet}\theta. \mathcal{O}(t_1) \subseteq \mathcal{O}(t_2) \vee \mathcal{O}(t_2) \subseteq \mathcal{O}(t_1)$$

5.5.6 Semantics

Informally, the trace semantics of the RVSDG are that evaluating a strict (resp. γ-) net executes each of its transitions (resp. the γ-node and exactly one of t^{true} and t^{false}) exactly once. Moreover, any compound nodes are evaluated without interleaving evaluation of contained nodes with outside nodes (so all operands to a compound node must be evaluated before any contained transition). Evaluation *order* is still partly unspecified (it must respect dependence edges), but extra constraints may be imposed on evaluation order by introducing extra levels of hierarchy. `iter` nodes are evaluated atomically much as compound nodes, by evaluating all inputs (this synchronizes evaluation across the loop header) and then making a tail call (unconditional jump without link) to the entry point of the named net.

More formally, we can define an equivalence between the RVSDG and PDG, as follows. Each partition of places $[s]$ corresponds to a virtual register v_s (or a physical register, if the partition was so labelled), each transition t to a CDG edge, and each partition of transitions $[t]^{\mathsf{t}} = \{t_1, \ldots, t_j\}$ to a PDG subtree $N(t_1) = \ldots = N(t_j)$. Each strict net, containing transitions $\{t_1, \ldots, t_j\}$, becomes a PDG group node with CDG edges to each $n \in \{N(t_1), \ldots, N(t_j)\}$; each γ-net becomes a PDG predicate node, with true/false CDG edges to $t^{\mathsf{true}}/t^{\mathsf{false}}$. We can see this as turning the γ-nodes "upside down": whereas γ-nodes are drawn as a *merge* of their operands, *consuming* the values after their production, in the PDG instead the *split* of control-flow, before operand evaluation, is represented. (This represents how the operands would be conditionally evaluated!).

5.5.7 Performing γ-Ordering on the RVSDG

Recall from Chapter 3 that one of the key operations we performed when converting VSDGs to PDGs was the *γ-ordering* transformation (Section 3.4.1), by which two intuitively parallel γ-nodes were ordered into a tree. This was used to efficiently proceduralize VSDGs containing *independent redundancy* nodes (exemplified by Figure 3.2), as the branches of the tree precisely identified the executions under which the nodes were (or weren't) demanded.

In Chapter 3, γ-ordering was performed on the PDG, rather than the VSDG. Although not made clear at the time, this was because of issues relating to the VSDG's lack of operator sharing (Section 5.1). Specifically, nodes on paths from the subsidiary to the dominant γ-node lead to cycles which in the PDG were fixed by making them into shared operators, thus avoiding duplication. Secondly, PDGs produced after γ-ordering (for example Figure 4.5(a), showing the result of the `buildPDG` algorithm on Figure 3.2) require duplication during PDG sequentialization, after which some nodes (in that example, $\mathsf{r}_2 = \mathsf{op2}(\mathsf{rA})$) act as shared operators (shown in Figure 4.5(b) and (c)).

However, the RVSDGs tail-sharing edges allow the γ-ordering transformation to be performed directly on the RVSDG, as follows. (Examples, corresponding to those of Chapter 3 on the PDG, are shown in Figure 5.16.)

Let $G = (S, T, E, \ldots)$ be an RVSDG containing at least two complex nodes ($T \supseteq \{t_{\mathrm{d}}\langle G_{\mathrm{d}}\rangle, t_{\mathrm{s}}\langle G_{\mathrm{s}}\rangle\}$), each containing a γ-net. (Recall from Section 5.5 that each γ-net contains exactly three transitions: a γ-node g_{d} or g_{s}, a transition $t_{\mathrm{d}}^{\mathsf{true}}$ or $t_{\mathrm{s}}^{\mathsf{true}}$, and another $t_{\mathrm{d}}^{\mathsf{false}}$ or $t_{\mathrm{s}}^{\mathsf{false}}$.) Suppose g_{d} and g_{s} must be γ-ordered, because both G_{d} and G_{s} conditionally demand a value from some independent redundancy node t^{ir}, and g_{d} is chosen to be dominant (g_{s} subsidiary).

Without loss of generality, let $t_{\mathrm{d}}^{\mathsf{true}}$ and $t_{\mathrm{d}}^{\mathsf{false}}$ be complex nodes (if they are not, they can trivially be made so), containing nets $G_{\mathrm{d}}^{\mathsf{true}}$ and $G_{\mathrm{d}}^{\mathsf{false}}$. Transformation proceeds by adding to g_{d} an extra *component* (defined in Section 2.6.2) for each component of g_{s}, selecting between corresponding extra results added to $t_{\mathrm{d}}^{\mathsf{true}}$ and $t_{\mathrm{d}}^{\mathsf{false}}$ (Recall the components of g_{d} correspond 1-to-1 with the results of t_{d}). All edges outgoing from of t_{s} are then rerouted to instead leave the corresponding new results of t_{d}.

The subsidiary complex node t_{s} is then cloned to make two copies; one is placed in $G_{\mathrm{d}}^{\mathsf{true}}$ and the other in $G_{\mathrm{d}}^{\mathsf{false}}$ (operands of t_{s} can be acquired by routing the necessary places through as extra operands to t_{d}, $G_{\mathrm{d}}^{\mathsf{true}}$ and $G_{\mathrm{d}}^{\mathsf{false}}$), with their result places being returned from the respective nets. Importantly (and unlike in the VSDG), the two $t_{\mathrm{s}}^{\mathsf{true}}$ and the two $t_{\mathrm{s}}^{\mathsf{false}}$ may be tail-shared, thus avoiding static duplication of nodes other than g_{s} itself. (The g_{s}'s may not be tail-shared, as one will have an additional copy of n control-dependent on it, described next.)

Finally, the independent redundancy node t^{ir} is itself cloned, and moved into $G_{\mathrm{d}}^{\mathsf{true}}$ and $G_{\mathrm{d}}^{\mathsf{false}}$; however, whereas with respect to G, t^{ir} was an independent redundancy with a gating condition[13] $\langle ? \rangle(g_{\mathrm{d}}, c_{\mathrm{d}}^{\mathrm{t}}, c_{\mathrm{d}}^{\mathrm{f}}) \oplus \langle ? \rangle(g_{\mathrm{s}}, c_{\mathrm{s}}^{\mathrm{t}}, c_{\mathrm{s}}^{\mathrm{f}})$, the two copies now have gating conditions $c_{\mathrm{d}}^{\mathrm{t}} \cup \langle ? \rangle(g_{\mathrm{s}}, c_{\mathrm{s}}^{\mathrm{t}}, c_{\mathrm{s}}^{\mathrm{f}})$ and similarly for $c_{\mathrm{d}}^{\mathrm{f}}$; this simplification eventually allows one copy of t^{ir} to be further moved into a subregion of the appropriate copy of t_{s}.

Fixing Cycles As in the PDG, transitions $t \in G$ on paths $t_{\mathrm{s}}{}^{\bullet} \xrightarrow{*} t_{\mathrm{d}}$ must be cloned along with t_{s}, with their final edges $\rightarrow t_{\mathrm{d}}$ redirected to the appropriate nodes in $G_{\mathrm{d}}^{\mathsf{true}}$ and $G_{\mathrm{d}}^{\mathsf{false}}$. Intuitively, this duplication is necessary, as such nodes correspond to later uses by G_{d} of values computed earlier by G_{s}, yet by making g_{d} dominant, one is testing and branching on the predicate of g_{d} first. However, transitions $t' \in G$ on paths $t_{\mathrm{d}}{}^{\bullet} \xrightarrow{*} t_{\mathrm{s}}$ may be handled without duplication—specifically, although the t' must be cloned and moved into $G_{\mathrm{d}}^{\mathsf{true}}$ and $G_{\mathrm{d}}^{\mathsf{false}}$ as predecessors of the two copies made of t_{s}, the copies of the t' may be *tail-shared* as they are used by identical computations (even though their histories are different, with one copy depending on the nodes originally in $G_{\mathrm{d}}^{\mathsf{true}}$ and one on those from $G_{\mathrm{d}}^{\mathsf{false}}$. Contrast with the t earlier, where the history of the copies was the same, but the future different: one copy would be used by the nodes originally in $G_{\mathrm{d}}^{\mathsf{true}}$ and the other by those from $G_{\mathrm{d}}^{\mathsf{false}}$, and so tail-sharing was not possible).

5.5.8 Chapter Summary

In this Chapter, we have seen how previous formulations of the VSDG fail to represent the sharing of operators satisfactorily. We have seen how sharing can be explicitly specified by an equivalence relation on the nodes of an Unshared VSDG (u-VSDG), and how this technique both captures operator sharing and unifies the VSDG and PDG, providing a new view on proce-

[13] Gating conditions were described in Section 3.3.2, and used in Section 3.4.1 to guide application of γ-ordering; in particular, the $\oplus$ constructor precisely identifies independent redundancies.

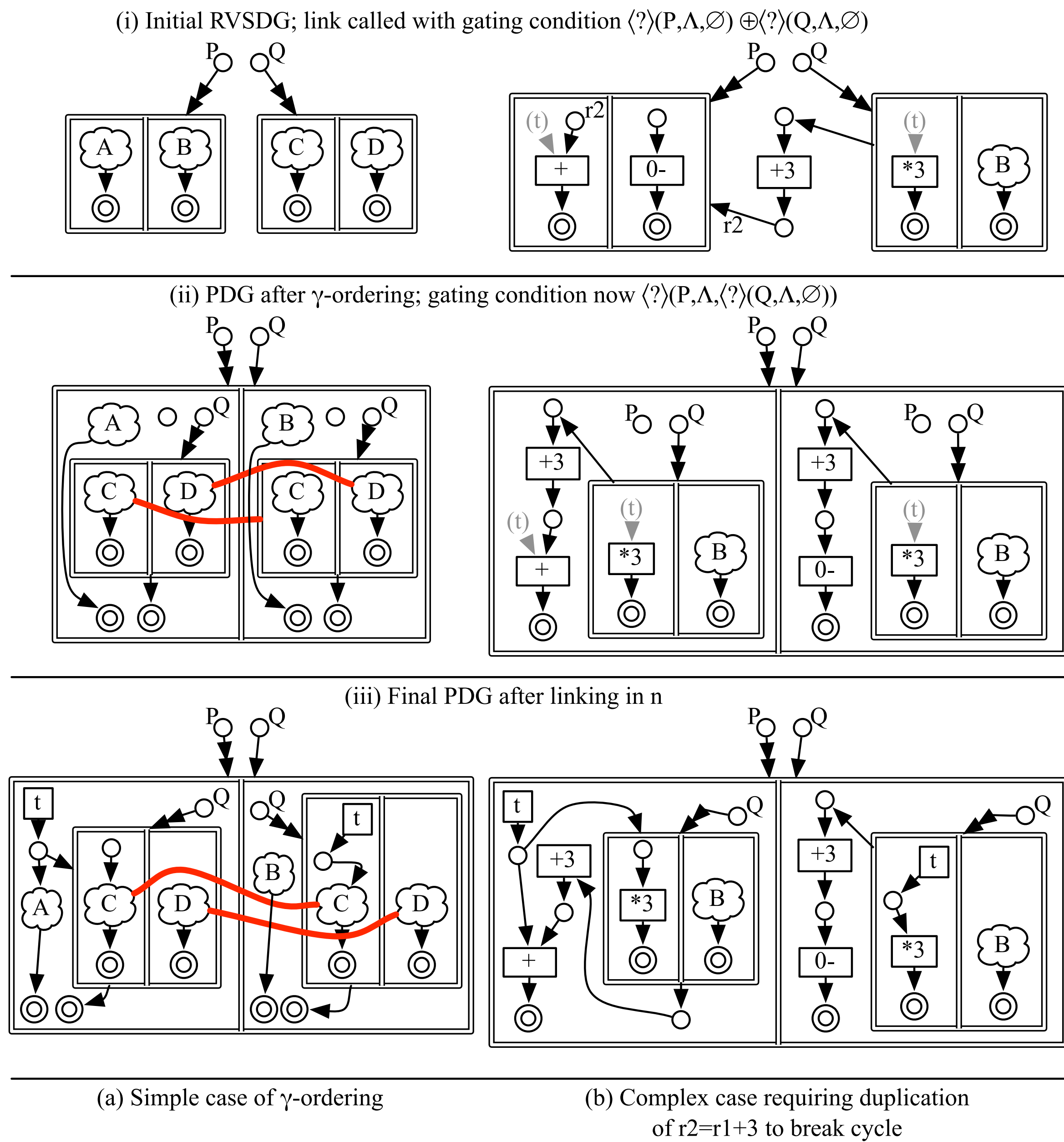


Figure 5.16: The γ-ordering transformation reformulated on the RVSDG (PDG version shown in Figure 3.10)

duralization. Further, we have seen how many loop optimizations are unified with optimizations on straight-line code in this framework. Lastly, we have recast the PDG in the form of a new data structure, the Regionalized VSDG (RVSDG).

CHAPTER 6

Node Scheduling

Node Scheduling is the conversion of imperative code from parallel to serial form, that is, from df-PDG (or df-RVSDG, as defined in Section 5.5) to CFG. Specifically, this includes both putting a total ordering on the children of each group node, paralleling the *instruction scheduling* optimization on the CFG, and the allocation of physical registers.

However, unlike CFG techniques, node scheduling operates uniformly on both single instructions and larger program fragments containing their own control flow. In contrast, CFG techniques become substantially more complex if they are to move things between basic blocks and/or more than a single instruction at a time [HMC+93, Swe92, Fis81, BR91].

This chapter describes how a wide range of techniques for register allocation and instruction scheduling fit into our framework for Node Scheduling, with a particular focus on Johnson's algorithm for combined Register Allocation and Code Motion (RACM) [JM03, Joh04]. We describe his original formulation in Section 6.1, and rework the algorithm onto the RVSDG in Section 6.2. Section 6.3 discusses simple extensions to Johnson's algorithm, and Section 6.4 considers the scope for a wider range of techniques. Lastly, Section 6.5 discusses issues of phase-ordering (first introduced in Section 2.4.1) raised by node scheduling coming as a distinct phase after df-PDG construction.

6.1 Johnson's Algorithm

Johnson's algorithm was developed as working directly on the VSDG, without the explicit hierarchy of the RVSDG or (df-)PDG. While this approach was successful in exploiting the VSDG's lack of ordering constraints, there are problems with the exact structures used, which we discuss throughout, and in particular in Sections 6.1.1 to 6.1.3.

Johnson also used an earlier version of the VSDG, which differed from ours in three ways relevant here:

1. The representation of loops, which used θ-nodes;

2. Transitions and their result places were combined as single nodes using a system of *ports* [Joh04], rather than our Petri-net formulation;

3. Edges were drawn in the opposite direction (from value consumers to value producers).

For consistency, we will continue to use the notation of Petri-nets in the presentation in this thesis, including drawing edges from producers to consumers. Hence, transitions t will often treated together with their results $t^\bullet$ as single entities, with only edges $S \times T$ considered.

A rough outline of Johnson's algorithm is as follows:

1. Partition the nodes into a series of *cuts*—so that edges only go from higher cuts to lower ones, never in the opposite direction—based on their maximum Depth From Root (DFR), i.e. from the return node. (This means we will visit the nodes in breadth-first order).

2. Visit the cuts in order from the return node upwards through the VSDG (that is, in the reverse order to execution), and for each cut in turn

   (a) Calculate the liveness width of the cut (that is, the number of distinct registers required to store the values needed as inputs)

   (b) While the liveness width is greater than the number of available registers, apply the following transformations:

      i. Raise nodes into higher cuts, by adding serializing edges.

      ii. Clone nodes, such that their values are recomputed rather than being kept in registers

      iii. Add load/store nodes (memory operations), thus spilling values to memory.

   Johnson uses the analogy of a snowplough—pushing excess snow forwards and depositing it where there is little snow—to describe this process; *"the goal is to even out the peaks and the troughs"* [Joh04, JM03].

3. At all program points, the number of values to be stored is now less than the number of registers; thus, a physical register can be allocated to each output port without conflict. This completes the process.

We now consider each stage in more detail.

Cuts Based on Depth-from-Root The DFR $\mathcal{D}(t)$ of a transition t is the length of the *longest* (acyclic[1]) path to the return node. Places are grouped with their producers, that is $\mathcal{D}(s) = \mathcal{D}(^\circ s)$—this ensures that each transition t or place s is in a cut which will be evaluated before any of its consumers $t^{\bullet\bullet}$ or $s^\bullet$.

DFR can be computed by a depth-first traversal *backwards* along the edges (both state and value) of the VSDG. The set of places of depth d, $\{s \mid \mathcal{D}(s) = d\}$, is written S_d, and similarly T_d for places; generalizing this notation we also write $S_{\leq d}$, $T_{>d}$, etc.

The algorithm visits the cuts in order, and does not return to or mutate earlier cuts (those closer to the return node) after it has seen later cuts. Thus, when considering the action of the algorithm on a particular cut, we write d for the depth of that cut, consisting of nodes $S_d \cup T_d$.

[1] Johnson's θ-node loop representation meant the VSDG could contain cycles (satisfying particular constraints); these were excluded when calculating DFR.

Liveness Width This is the number of distinct values live at entry to a cut—including values used (perhaps exclusively) in later cuts, closer to the return node. In the absence of γ-nodes, liveness width can be computed as

$$W_{in}(d) = \|S_{>d} \cap {}^{\bullet}T_{\leq d}\|$$

counting only value operands.

However the presence of γ-nodes in the VSDG complicates this somewhat. In the absence of explicit regions, let $R^{\mathsf{true}}(g)$ be the subgraph of the VSDG which is postdominated by the true operands of a γ-node g, and let $R^{\mathsf{false}}(g)$ be likewise for the false operands. In any execution, it will be necessary to store operands for only the nodes in $R^{\mathsf{true}}(g)$ *or* $R^{\mathsf{false}}(g)$ (we refer to any such nodes as being *predicated* on g), not both. Thus, quoting Johnson's thesis (we write N_d for his S_d, as he did not distinguish between transitions and operands):

> To compute the maximal $W_{in}(d)$ we compute the maximum width (at depth d) of $N_{\leq d} \cap R^{\mathsf{true}}(g)$ and $N_{\leq d} \cap R^{\mathsf{false}}(g)$. This computation is recursively applied to all γ-nodes, g, where $N_{\leq d} \cap R^{\mathsf{true}}(g) \neq \emptyset$ or $N_{\leq d} \cap R^{\mathsf{false}}(g) \neq \emptyset$. Thus for a set of nested γ-nodes we compute the maximal (i.e. safe) value of $W_{in}(d)$.

Available Registers We write W_{max} for the number of available registers; this is the number of physical registers but less any required for e.g. holding temporary values loaded from memory, etc.

Transforming the VSDG Transformations are applied both to reduce liveness width to fit in W_{max}, and to bring the operations in each cut down to fit the issue width[2] of the instruction set architecture (ISA). The transformations follow; each is applied at all possible candidate sites before the next transformation is considered:

Node Raising In order to reduce liveness width, additional *serializing edges* are added—these express additional constraints on ordering in a similar manner to state edges[3]. Raising proceeds by first selecting a *base node* $t_b \in T_d$ (see below), and then, while beneficial nodes remain, a transition $t \in T_d$ giving greatest benefit is arbitrarily selected and serializing edges $t^{\bullet} \dashrightarrow t_b$ added. This has the effect of pushing t up into the next higher cut (and perhaps altering other higher—not-yet-visited—cuts).

The benefit derives from that *some* operands to the raised node t (namely, those unique to t and not used by other nodes in the same or lower cuts—i.e. ${}^{\bullet}t \backslash {}^{\bullet}T_{\leq d}$) are removed from $W_{in}(d)$. The results $t^{\bullet}$ are added to $W_{in}(d)$, so the benefit is thus the number of inputs

[2]The number of instructions which appear to execute simultaneously. For example, on x86 or PowerPC, this is a single instruction at a time; as a consequence, the VSDG of Figure 6.1, in which each cut has a liveness width of two, cannot be scheduled in less than *three* registers without spilling. Hence each cut must be transformed until it contains only a single instruction. Contrastingly, in architectures where multiple instructions appear to be executed at once (e.g. VLIW or EPIC architectures), Figure 6.1 could be scheduled with only two registers; however, other restrictions on what instructions may be combined into a single word/bundle must be taken into account.

[3]We require state edges to satisfy a requirement of *linearity*, discussed in Chapter 2 and formalized in Appendix B. This means they cannot specify the *partial order* requirement that (A and C) come *respectively* before (B and D) without specifying a *total* ordering of all four operations. However, in Johnson's VSDG there was no such requirement, and he used state edges to express arbitrary ordering constraints. Hence, it is easiest to see serializing edges as a third kind of edge besides state and value.

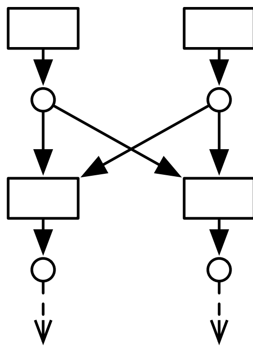


Figure 6.1: Even when liveness width is less than W_{max}, architectural issue width may make register allocation impossible.

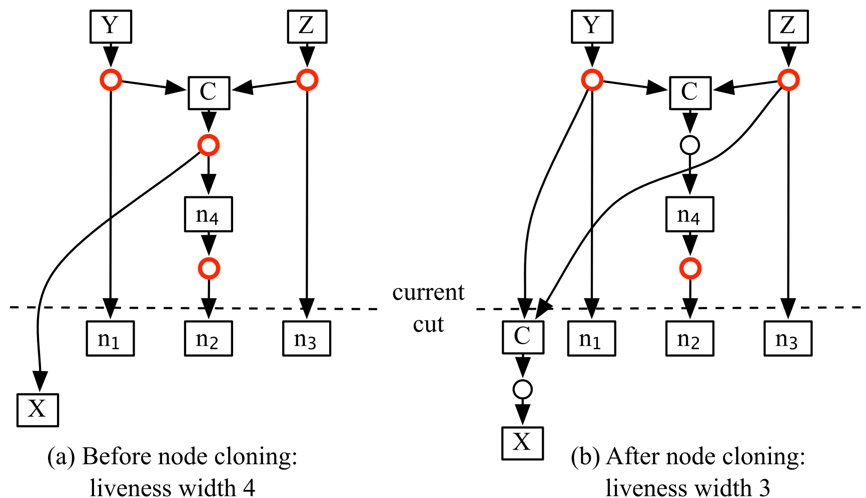


Figure 6.2: Effect of node cloning in reducing liveness width. The places live at entry to the current cut are indicated in red.

unique to t less the number of its results. The base node is chosen as a node which would have the least benefit.

Node Cloning If liveness width cannot be sufficiently reduced by adding serializing edges, the next transformation used is node cloning. A stateless transition t with multiple consumers $t^{\bullet\bullet}$ is removed from above the current cut, and replaced by a copy for each consumer (copies which end up in the same cut can later be recombined); serializing edges are used to ensure such copies are not added below the current cut. As Figure 6.2 shows, this can reduce liveness width by making the node's value dead at entry to the cut (and recomputing it in the current cut); the greatest benefit occurs then the cloned node was used by node(s) below the current cut (node X in diagram) and itself uses other values already live for the current cut (nodes Y, Z in diagram). Selection proceeds by finding the set of candidate nodes whose values are passed through the current cut (i.e. $\{t \in T_{>d} \mid t^{\bullet} \cap {}^{\bullet}T_d = \emptyset \wedge t^{\bullet} \cap {}^{\bullet}T_{<d} \neq \emptyset\}$), and picking one of those which generates the greatest

reduction in liveness width.

Spilling If all previous attempts to reduce liveness width have failed, lastly values are spilled—we refer to their outgoing edges being *tunnelled* through memory. Since the aim is to decrease the liveness width at entry to the current cut, only values $s \in S'$ passed through the current cut are considered: that is, values from higher cuts which are used in lower cuts but not in the cut under consideration ($S' = S_{>d} \cap {}^\bullet T_{<d} \setminus {}^\bullet T_d$). A store transition is added as consumer of s (this node will end up in a higher cut by node raising), and a load transition added to a lower cut (this is guaranteed to succeed since lower cuts already fit in W_{max}). Selection of the s (from candidates S') is decided by a heuristic; Johnson uses the static count of memory operations required (potentially greater for values modified in a loop) divided by the *lifetime* of the tunnelled value, i.e. $\mathcal{D}(s) - max(\{\mathcal{D}(t) \mid t \in s^\bullet \cap T_{<d}\})$.

In the next three subsections we consider some subtleties not mentioned by Johnson.

6.1.1 Atomicity of γ-Regions

In the final serialized output (a CFG), nodes whose execution should be conditional on the predicate of a γ-node (i.e. from $R^{\textsf{true}}(g)$ or $R^{\textsf{false}}(g)$) cannot be interleaved with other nodes[4]: the former nodes must be placed in the arms of an `if(P) then {...} else {...}` whereas the latter must be placed before the `if` or after the merge. Johnson's algorithm handles this subtlety [Joh] by:

- A *split node* ς_g is selected[5] for each γ-node g—either the node computing g's predicate node is used, or an artificial no-op node is added.
- Serializing edges are added from ς_g to all nodes in the $R^{\textsf{true}}(g)$ subgraph.
- A null (no-op) transition t_g, with no value dependencies or results, is added to the VSDG, and serializing edges added to it from all nodes in $R^{\textsf{true}}(g)$
- Further serializing edges are added from t_g to all nodes in $R^{\textsf{false}}(g)$, and from those to g.
- The entire $\varsigma_g \dots t_g \dots g$ complex, which results in an `if(P) then {...} else {...}` sub-CFG, is then treated as atomic for code-motion operations[6].

Note that this complicates the calculation of maximal depth-from-root (DFR) above, as nodes predicated on γ-nodes are not in fact interleaved with other nodes. Furthermore, when liveness width is exceeded in some $R^{\textsf{true}}$ or $R^{\textsf{false}}$ subgraph, a decision must be made as to whether to perform node raising *within* the subgraph, or *without* it—i.e. to attempt to evaluate the whole $\varsigma_g \dots g$ complex at some point where there are fewer registers taken up by external values. It is unclear exactly what happens in Johnson's solution, but it seems reasonable that the usual heuristics (number of unique inputs less outputs, etc.) could be used both inside and outside the region simultaneously, and then transformations applied within and/or without in the order suggested by the union of the metric.

[4]Unless predicated execution was used—discussed in Section 6.3.4.

[5]Johnson referred to the split node by σ_g; we reserve this notation for *state* places.

[6]There is no possibility of cross-jumping into this code as operator sharing (Chapter 5.1) is not expressible in Johnson's VSDG.

6.1.2 Node Raising and Speculation

Node raising is capable of introducing speculation (causing a node to be evaluated under certain circumstances in which it previously would not). Specifically, given a transition t predicated on a γ-node g, adding a serializing edge $t^{\bullet} \dashrightarrow t'$ to any transition t' *outside* $R^{\mathsf{true}}(g)$ or $R^{\mathsf{false}}(g)$ (for example, to the γ-node's predicate), then t itself is moved out of $R^{\mathsf{true}}(g)$ or $R^{\mathsf{false}}(g)$ and *speculatively* evaluated before the `if` corresponding to g. This is shown in Figure 6.3.

However, Johnson left it unclear whether this transformation is performed: the split nodes ς_g, whose use was described in Section 6.1.1 above, might have provided an ideal target for such serializing edges, but the cycles $\varsigma_g \leftrightarrows t$ created might have prevented this (the split node reaches every predicated node because of the extra serializing edges added in Section 6.1.1).

6.1.3 Node Cloning and Dominance

Individual nodes created by the node cloning transformation may be postdominated by the true or false operands of a γ-node g where the original was not, as in Figure 6.4. That is, the clones might be members of $R^{\mathsf{true}}(g)$ or $R^{\mathsf{false}}(g)$, and their execution would then be predicated upon g. (Such clones are not merged together even if at the same DFR.)

6.2 Reformulating Johnson's Algorithm on the RVSDG

In this section, we will consider the issues that arise when using Johnson's algorithm to perform register allocation on the Regionalized VSDG (recall this structure was defined in Section 5.5 as a rationalization of the PDG—however, being closer to Johnson's original VSDG, the RVSDG is the more natural representation).

The main issues are the adaptation to hierarchy and tail-sharing.

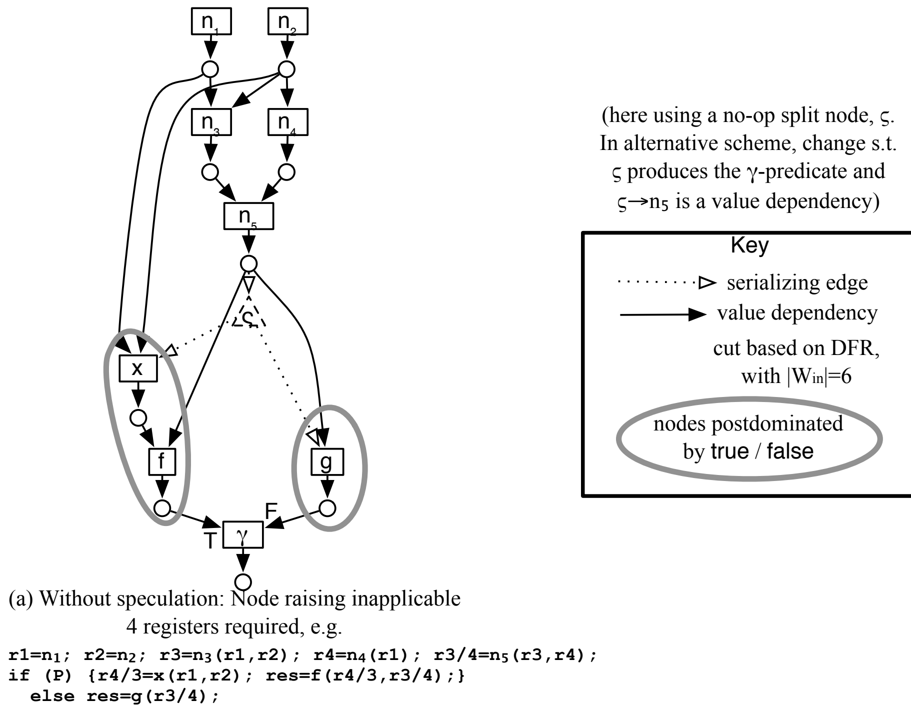
6.2.1 Hierarchy

Many of the subtleties of Johnson's algorithm—particularly, the calculation of liveness width (page 122), and the atomicity of γ-regions (Section 6.1.1)—are caused by the VSDG's not specifying an evaluation strategy: instead, Johnson computes the transitions to be executed conditionally on γ-nodes (i.e. $R^{\mathsf{true}}(g)$ and $R^{\mathsf{false}}(g)$) according to postdominance. We can see this as performing regionalization of proceduralization somewhat implicitly, using an eager (speculative) strategy (justified by Johnson's goal of small code size—this was discussed in Section 4.3.1).

Thus, much of the effect of using the RVSDG is to make explicit book-keeping operations, to do with nodes being predicated, that were happening anyway. We argue that this improves on the original VSDG formulation by making clear and enforcing these subtleties from the outset.

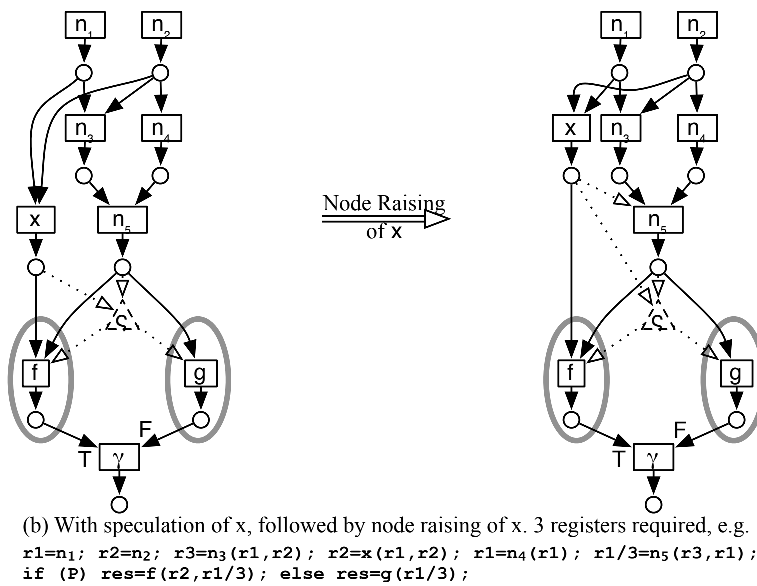
Specifically, we see Johnson's algorithm being applied across the hierarchy one contained net at a time, with complex nodes $t\langle G'\rangle \in G$ being treated as single nodes (forcing their contents G' to be executed without interleaving execution of the containing graph G) and the contained net G' being processed recursively.

Thus, we associate a *maximum internal liveness width* $w_{int}(t)$ with each complex node $t\langle G'\rangle$, being the greatest number of registers required as temporaries any point within G'. An additional check, that there are sufficient registers to evaluate all nodes in a cut even when its inputs fit into W_{max}, is then made. (However, such checks are required anyway due to ISA issue



(a) Without speculation: Node raising inapplicable
4 registers required, e.g.

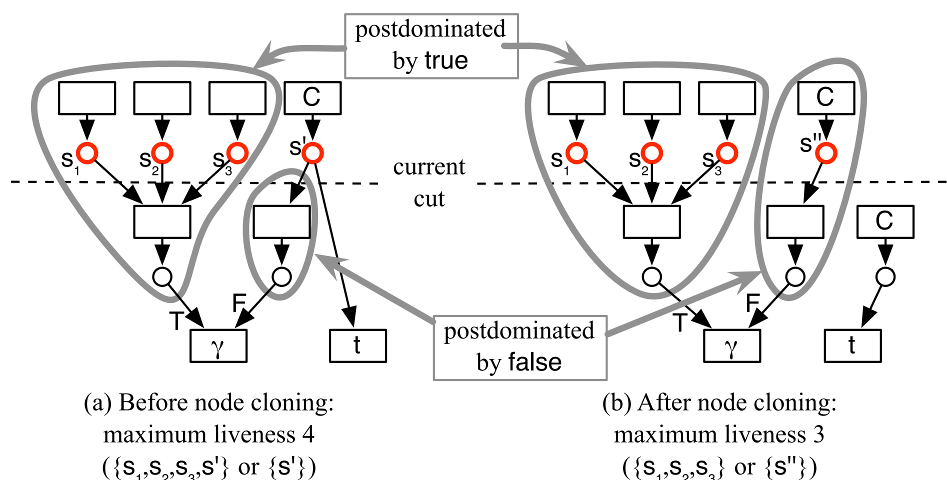
```
r1=n1; r2=n2; r3=n3(r1,r2); r4=n4(r1); r3/4=n5(r3,r4);
if (P) {r4/3=x(r1,r2); res=f(r4/3,r3/4);}
  else res=g(r3/4);
```



(b) With speculation of x, followed by node raising of x. 3 registers required, e.g.

```
r1=n1; r2=n2; r3=n3(r1,r2); r2=x(r1,r2); r1=n4(r1); r1/3=n5(r3,r1);
if (P) res=f(r2,r1/3); else res=g(r1/3);
```

Figure 6.3: Addition of serializing edges in Johnson's algorithm can cause nodes to be speculated



(a) Before node cloning: maximum liveness 4 ($\{s_1,s_2,s_3,s'\}$ or $\{s'\}$)

(b) After node cloning: maximum liveness 3 ($\{s_1,s_2,s_3\}$ or $\{s''\}$)

Figure 6.4: Node cloning can make nodes be predicated when the originals were not. (Places live at entry to the current cut indicated in red.)

width—see footnote[2] on page 123.)

An additional book-keeping operation is required when the *node cloning* transformation leads to individual clones being postdominated (in the flattened graph) by the true or false operands of some γ-node where the original was not, as discussed in Section 6.1.3. Such clones should be *explicitly* moved into the corresponding true or false transitions (recall each γ-node g is in a net containing only three transitions $\{g, t^{\text{true}}, t^{\text{false}}\}$—$t^{\text{true}}$ and t^{false} can first be made into complex nodes if necessary).

6.2.2 Tail-Sharing Regions

Further issues arise because of the representation in the RVSDG of more complicated control flow structures than in the VSDG—specifically, the merging of control flow caused by tail-sharing.

Recall that after PDG sequentialization, a *partial* order (specified by the properties OrderFixed or OrderArbitrary, discussed in Section 2.4) is imposed on the children of each PDG group node, i.e. the transitions in each net instance in an RVSDG. Intuitively, transitions which are left unordered by this partial order (i.e. OrderArbitrary) are so because they all tail-shared *the same amount*, and thus, filling out the ordering may be left to the node scheduling phase.

We somewhat loosely refer to a group of such nodes as a *tail-sharing region* or just a *region*; although it would be possible to group the nodes together explicitly in complex nodes, this has not been a requirement. However, we consider the true and false transitions of γ-nodes as being regions, and describe a region as tail-shared with another if all of its elements are (piecewise) tail-shared, in the same way as for complex nodes.

Moreover, we lift the concept of partitions over regions (i.e. a partition of regions is a maximal set of regions which are tail-shared). The key consideration is that every region in such a tail-sharing partition must be scheduled identically, even though it appears in a different place in the γ-tree. Thus, heuristics must somehow select a single transformation amongst the different ones which might otherwise be chosen in each case. One possibility is to calculate the DFR (maximum depth from root) for each *partition* of nodes, such that every node in the partition

has the greatest DFR that would have been assigned to any: this allows all the elements to be considered at the same time, and the heuristic results from each to be combined.

When PDG sequentialization forces an ordering of nodes (i.e. OrderFixed), because one transition t_1 is tail-shared more than another t_2—and thus t_1 must execute after t_2 for control flow to merge back in—these ordering constraints can be expressed by addition of extra serializing edges (that is, an edge $t_2{}^\bullet \dashrightarrow t_1$ can be added to force t_2 to execute first). This causes serializing edges to be added from every node in a region to every node in its successor region (i.e. that into which control flow will merge), and this prevents the node raising transformation from raising nodes past the boundaries of these regions. (This would raise complications considered in Section 6.3.3.)

Lastly, observe that values from regions ordered earlier will be passed through later (more-shared) regions, right through to the γ-tree itself. This results in some interactions with the spilling and node cloning transformations, as follows.

Spilling Such passed-through values are good candidates for spilling, as the memory access instructions can be statically shared, reducing the code size overhead. (Additionally, observe that values used only in highly-shared (late-executed) groups can also be spilt cheaply, as only one static load is required for all tail-shared copies; the store can be done at each leaf or speculatively.)

Node Cloning Such passed-through values also look like good candidates for node cloning, as this transformation normally picks nodes, from arbitrary higher cuts, whose results are passed through the current cut but are not used in it. However, it is not possible to clone a node from a less-tail-shared region into a more-tail-shared one (unless some form of procedural abstraction was used; we do not consider this in this thesis), so this transformation must be prevented unless an identical node exists in *all* predecessor regions.

6.2.3 Register Allocation

Another difference however is in the handling of register allocation. In Section 5.5 we defined *coalescing partitions* on the RVSDG, stating these could be assigned physical registers. In particular we allowed places to be coalesced without necessarily choosing a target physical register and so long as *some* non-conflicting scheduling existed. This contrasts with Johnson's system in which places are coalesced only by assigning them the same physical register, and only if they would not conflict in *any* possible scheduling (i.e. cut), with serializing edges used to restrict the set of possible schedulings.

However this does not pose a problem: in the algorithm as presented, serializing edges were used mainly to reduce the *number* of registers required, rather than to eliminate conflicts involving a single register. Thus, any coalescing of places and labelling with registers required to satisfy non-orthogonal instructions or procedure calling standards can be performed *first*, with the addition of serializing edges to bring the number of simultaneously live values within W_{max} coming afterwards. Lastly, the remaining places can be assigned registers by Johnson's system; any further coalescing this entails in the RVSDG will not violate well-formedness because the places will not conflict in the ordering already selected by the serializing edges.

6.3 Simple Extensions

6.3.1 Heuristics for Speed over Space

Johnson's heuristics for selecting the nodes to clone, raise, or spill were decided upon in order to optimize code size, rather than speed. If speed is instead the priority, there is much scope for change:

- Johnson's algorithm applies node raising first, as its "cost" in code size is essentially zero. However, in terms of execution speed, it may have an effect on instruction latencies (considered in Section 6.3.2), and further, may cause nodes to be speculated (Section 6.1.2), which is undesirable in this setting.

- Conversely, node cloning *always* increases code size, but may not always increase execution time. Specifically, if the clones are postdominated by the true and false operands of γ-nodes (Figure 6.1.3), it may be possible that no more than one will execute—indeed, if the proceduralization were not optimal in Upton's sense (Section 3.1) and already speculated the node, execution time may be *decreased*. This should be taken into account when evaluating the cost of node cloning, as it may make it cheaper than node raising.

- It may be desirable to clone even complex nodes containing γ-nodes. Johnson's algorithm will always choose to spill the result of a compound γ-node (at a cost of one load/store pair) rather than clone it[7]), except in the case of multiplexers selecting values computed already. However, when optimizing for speed, recomputation may be cheaper, and the heuristics may wish to take this into account.

6.3.2 Instruction Latencies

CFG compilers often include an instruction scheduling phase which attempts to avoid pipeline stalls due to instruction latencies. However, this optimization has been particularly antagonistic with register allocation, as the effect of instruction scheduling is often to "spread out" computations which use each others values, and this tends to increase register pressure. Hence many often complicated algorithms exist combining the two phases [Bra94, NP95, BGS98a, Tou02]. On the VSDG, it is simple to extend Johnson's algorithm to take instruction latencies into account, as follows. For any slow instruction t (one whose results may take multiple CPU cycles to compute), extra no-op transitions (which merely pass their operands through to their results) are added inbetween $t^\bullet$ and their consumers $succt^\bullet$. These naturally increase the DFR of the slow instruction, but do not generate instructions in the output (they can be seen as `MOV`s whose operands and results are coalesced, but whose elision is delayed until after node scheduling).

An alternative is to modify the computation of DFR to allow each node to have a "thickness" (i.e. the amount added to depth varies according to the node passed through); this might allow more flexibility in e.g. modelling the latency of memory accesses (which can be seen as "as long as possible").

[7]Including the predicated nodes—cloning only the root γ-node g would greatly increase liveness width *everywhere*, by ending the predication on any single γ-node of the nodes in $R^{\mathsf{true}}(g)$ and $R^{\mathsf{false}}(g)$.

6.3.3 Movement Between Regions

We defined the concept of *regions* in Section 6.2.2; they identify groups of nodes within which node scheduling may operate freely, but at whose boundaries node scheduling is prevented by control flow structure—specifically, control flow merges due to tail-sharing. However, we also see regions as including the true and false sub-nets of γ-nets, where the boundary is caused by the control flow *split*.

However, both register allocation and instruction scheduling could be performed more effectively if it were possible to move nodes between regions, specifically by raising a node to *before* the beginning of its region. This would allow additional node raising transformations, which may be preferable to node cloning or spilling. We see three distinct cases:

1. Raising a node out of a γ-net subregion. This causes the node to be speculated—discussed in Section 6.1.2.

2. Raising a node out of a tail-shared region. This entails a *space* cost of (statically, not dynamically) duplicating the raised node into each preceding (less-shared) region, but may avoid the *time* cost which would result from node cloning.

3. Raising a node out of a loop region. This entails statically duplicating the raised node in the enclosing region, and moving the node round to the "end" of the loop, but allows the node scheduling phase to perform *software pipelining* (discussed below).

A further hint is the *nondeterministic* algorithm developed by Johnson to define the scope of combined Register Allocation and Code Motion (RACM) techniques. This was a general framework, encompassing both the "snowplough" algorithm we have seen as well as Chaitin/Chow-style register allocation [Cha82, CH84]. Specifically, the framework functioned by "undoing" code motion optimizations performed by preceding stages; in Johnson's case, the only code motion operation performed was hash-consing at construction time (in terms of reducing code size, this encompasses CSE, VBE, and (sparse) PRE [RKS00]). However, in our compiler architecture, the preceding phases of proceduralization and PDG sequentialization can perform a greater range of code motion transformations, and by extension node scheduling should in principle be able to undo these—which requires moving nodes between regions.

Software Pipelining Our optimality criteria for proceduralization results in all loops having zero degree of pipelining, in order to produce the smallest size code. However case 3 above captures the transformation necessary for the node scheduling phase to perform software pipelining in a way uniform with instruction scheduling (discussed in Section 6.3.2). This would allow use or adaptation of of the same heuristics as for straight line code, or application of other algorithms such as modulo scheduling [RG81, Ram92] or integer linear programming [Tou05].

6.3.4 Predicated Execution

Another simple extension is the use of predicated execution for instructions in γ-nets, rather than testing and branching control flow. This could allow the interleaving of predicated and other nodes so long as the predicated nodes do not include any nested γ-nodes[8].

[8]Even this might be possible on architectures with multiple (independent) condition bits—for example, Intel's Itanium [Sha99], with its 64 *predicate registers*. This intriguing possibility is left for future work.

6.4 Alternative Approaches

The VSDG also supports a wide variety of other approaches, which we discuss here. Our intention is merely to show the ease of manipulation provided by the VSDG, and that this enables consideration of issues difficult to address in a CFG-based system (doing so there would entail constructing a global representation of the dataflow and dependencies, i.e. pretty much a VSDG!), so we only sketch some possibilities.

Local Lookahead Generally, all of Johnson's heuristics can be seen as having only a lookahead of one—that is, they do not consider the cumulative effect of repeated transformations. An obvious example is in calculating the benefit of node raising: two nodes may have no benefit individually, but may reduce liveness if *both* are raised. Another is choosing between repeated application of node raising and cloning over subexpressions: the former replaces in the current cut the inputs of the raised node with its results, whereas the latter removes its results but adds its inputs.

This lack of lookahead is particularly problematic when combined with the inability to backtrack to previously-allocated cuts; Figure 6.5 shows an example where a previously-visited cut "commits" the algorithm to having too many values held live over higher cuts.

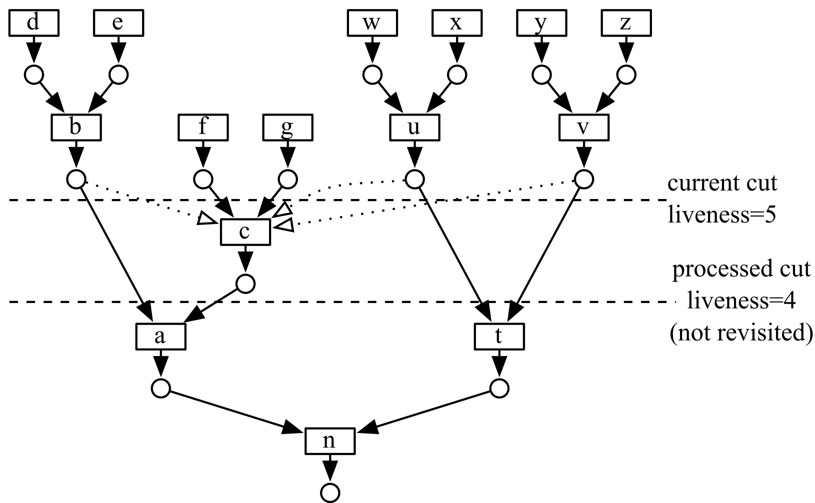
Depth-First Traversal Consider the action of Johnson's algorithm on a binary tree, whose essential characteristic is having large disjoint expressions which are combined after computation of each. The DFR-based approach interleaves the nodes of the two expressions, and at higher DFRs, early cuts (which have already been processed and are not revisited) can lead to many values being live; this can lead to potentially many values being spilled partway through both expressions. A small example (in which only one value is spilled) is shown in Figure 6.5.

An alternative is to use a *depth-first* traversal, which tends to clump together the instructions within each subexpression, with fewer values being held live (but for longer) while other subexpressions are computed. This is shown in Figure 6.6, being the result of traversing the same VSDG as shown in Figure 6.5 using the depth-first technique–observe spilling is avoided. Further, the longer lifetimes before the values need be loaded again give better timing characteristics when spilling does occur.

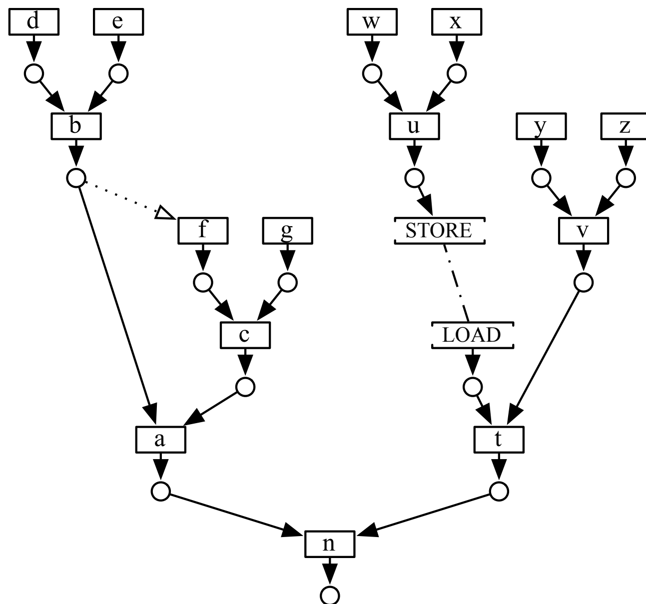
(However, the interleaving of operations which tends to result from the breadth-first algorithm gives better timing characteristcs when values are not spilled—particularly when trying to fill pipeline slots due to instruction latencies, discussed in Section 6.3.2).

An Adaptive Hybrid Thus, a more effective algorithm might be one which intelligently switched between depth-first and breadth-first traversal techniques according to register pressure, using breadth-first where possible but switching to depth-first to avoid spilling.

Graph Shape It seems intuitive that heuristics examining the *shape* of the graph, perhaps attempting to find good boundaries for subexpressions which share little with the rest of the graph, would be useful for guiding such a technique. (Or indeed as better heuristics for Johnson's transformations). However, such analyses are left for future work.



(a) The second cut cannot be allocated with four registers



(b) Final result after spilling and further node raising

Figure 6.5: An expression of binary tree shape for which Johnson's algorithm unnecessarily spills values to memory.

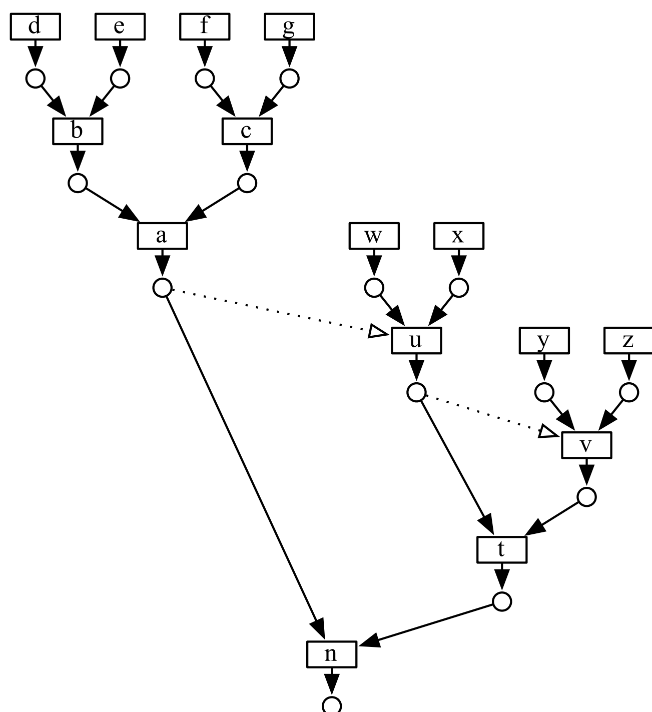


Figure 6.6: The same binary-tree-shape expression as Figure 6.5, here allocated by a depth-first traversal without spilling.

6.5 The Phase-Order Problem Revisited

The phase-order problem was discussed in Section 2.4.1, where we broke sequentialization apart into three distinct phases: proceduralization (aka regionalization), PDG sequentialization, and node scheduling. In this chapter, we have seen how node scheduling can be performed by an existing algorithm for combined Register Allocation and Code Motion (RACM).

Conceptually, we still find this break neat: proceduralization and node scheduling seem to differ in terms of algorithm characteristics (intervals and dominance vs. individual nodes), have separate concerns (platform-independent vs. -dependent attributes), and a well-defined boundary (a duplication-free PDG or RVSDG).

However, *any* node scheduling phase capable of node cloning—or especially, cross-region node raising (Section 6.3.3)—after proceduralization runs contrary to the principle of semantic refinement introduced in Section 2.7.1, and suggests the separation of concerns is not complete and the phase-order problem is not solved. (Intuitively, node cloning changes the evaluation strategy from call-by-need or -value to call-by-name, changing the trace semantics of the input RVSDG.)

The break also introduces problems in practice. Many of these relate to the specification, in the input RVSDG, of the way the regions in the program are structured in a tree (or DAG—as only one static copy exists for each tail-sharing partition of regions):

1. The tree affects the constraints with which node scheduling must deal, namely register pressure and instruction latencies, because of the *ordering* of the regions decided upon by preceding phases (values computed by earlier regions are passed through and held live over later ones). Specifically:

γ-ordering Given an *independent* redundancy, where two γ-nodes both *may* demand the value of some node n, the values returned by the γ-node which is made dominant are passed through computation of the values for the subsidiary γ-node(s).

Ordering of Tail Values Where two tail-groups are shared incomparably (neither is shared strictly more than the other), PDG Sequentialization must duplicate one group (making it strictly less shared); it is then executed earlier and its value held live until after evaluation of the later group.

2. The transformations performed in constructing the tree may have offered better solutions for the problems faced by node scheduling—specifically, the duplication they perform is different from the node cloning transformation and may have offered additional opportunities for reducing register pressure and dealing with instruction latencies.

3. Even if earlier stages performed the (likely NP-complete) searches for "optimal" solutions to the problems they faced, this optimality might be destroyed by subsequent node scheduling, especially if register pressure is high. (Besides the searches involved in γ-ordering and PDG sequentialization, above, we can also include proceduralization more generally here if this included searching for evaluation strategies which best trade speculation against code size or number of tests and branches).

4. The structure of the tree affects the costs and possibilities of moving nodes between regions (discussed in Section 6.3.3).

5. Johnson's framework suggests node scheduling proceed by "undoing" code placement optimizations (discussed in Section 6.3.3), but many of the optimizations our proceduralization phase performs cannot be undone without changing the tree, and it is not clear how this can be done effectively and incrementally *after* tree construction.

We can see a parallel here with superblock scheduling [HMC$^+$93] on the CFG. This is a more advanced instruction scheduling technique, capable of moving operations past the beginning or end of a basic block (and thus adding a copy to each successor or predecessor, respectively, although care must be taken where the successor (predecessor) has other predecessors (successors)). However, the technique retains the CFGs "two-level" system of instructions which label basic blocks—that is, superblock scheduling cannot change the basic blocks or restructure the program.

It might appear that the RVSDG resolves this problem: the VSDG's uniformity of expressions (Section 2.1.1) is retained, and hence node scheduling is capable of rearranging basic blocks by motion of entire expression subtrees relative to each other. However it only does so by moving expressions around the existing region tree, and the tree itself is not changed[9]. Whilst this constitutes an increase in flexibility compared to the CFG, the parallel suggests that a superior system would allow building a tree of regions to be combined with the placing of nodes in it.

The possibility of **predicated execution** also suggests such a system: whilst we have seen this as part of node scheduling, it allows an (almost) "free" retesting of a predicate, and thus the building of *better* region trees (PDG sequentializations) than would otherwise be possible.

[9]The relationship between RVSDG regions and SESE regions is substantially complicated by the former's `iter` nodes, but ignoring this issue for the sake of discussion, the Program Structure Tree [JPP94]—the structure of the program in terms of SESE regions—is not altered by node scheduling.

6.5.1 Combining Proceduralization and Node Scheduling

The lack of ordering restrictions in the RVSDG allow it to support incremental algorithms in which all the transformations we have seen to be interleaved.

Beginning with an input VSDG, first a "book-keeping" stage flattens all non-μ-bound complex nodes, applies hash-consing as fully as possible, and then puts each γ-node into its own net, with trivial or no-op true/false transitions, within a complex node. The result of this stage is a well-formed RVSDG, but one in which *all* nodes would be evaluated in every execution.

In the second stage, this RVSDG is incrementally refined by (nondeterministic) application of a number of operations, which change the regionalization, add serializing edges, or annotate places with registers, until eventually a unique CFG is specified. The operations are as follows[10]:

1. Assign a physical register to a coalescing partition (so long as that register is not assigned to any other partition).
2. Clone a node, redirecting some of the incoming arcs to the new copy
3. Tunnel values through memory by introducing store/load spill nodes
4. Predicate a node, i.e. move it into a γ-node subregion where the node's only consumers are already in that subregion
5. Speculate a node, i.e. move it out of a γ-node subregion into the containing region. (For stateless nodes only; discussed in Section 6.1.2).
6. Add a coalescing edge between two places, so long as there exist orderings in which the two are not simultaneously live.
7. Turn a coalescing edge into a tail-sharing edge, or add a tail-sharing edge between two identical transitions, so long as the sharing in the resulting RVSDG can still be implemented with a single program counter (see below).
8. Reroute one or more edges ($s \rightarrow \ldots$) from a place s to go via some γ-node g (by adding to g a new component c with s as both true and false operand, and edges $g \xrightarrow{c} \ldots$), so long as s which does not reach $g.\mathsf{predicate}$.
9. Add a serializing edge to order one node before another.

Tail-sharing edges must satisfy the standard RVSDG well-formedness conditions imposed upon them by Section 5.5.5, *and*, in order to include the intermediate step of PDG sequentialization, the extra requirement therein for the RVSDG to correspond to a *duplication-free* PDG.

[10]Note that a number of these relate to Johnson's nondeterministic algorithm, discussed in Section 6.3.3; specifically, operations (1-6) parallel or generalize operations in his framework, but (7) has no equivalent, and (8) is a significant extension on his tupling of γ-nodes (discussed below).

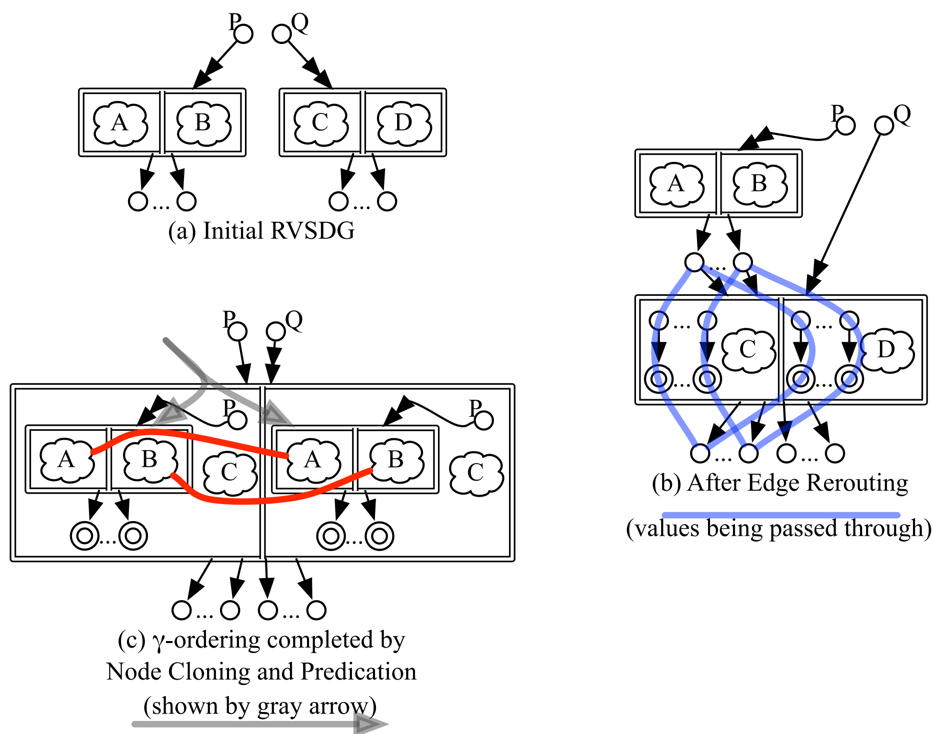


Figure 6.7: The edge rerouting transform serves to perform γ-ordering.

Rerouting of Edges serves two purposes. Firstly, it allows the γ-ordering transformation (Section 3.4.1) to be performed by first rerouting edges, then cloning the node for each of its successors (now exactly two—true and false), and lastly predicating each copy into the respective true or false contained graph, as shown in Figure 6.7. Any cycles hereby created can be removed by further rerouting, and/or cloning and predication, as discussed in Section 5.5.7 (of course, note that any tail-sharing edges introduced must satisfy the same restrictions as above).

Secondly, it allows the **tupling of γ-nodes**. This is discussed in Chapter 7, but the essence is to order two γ-nodes *with the same predicate* under each other; this allows each cloned γ to be collapsed to either its true or false sub-net and potentially *enables* many other optimizations.

Towards a Deterministic Algorithm It remains to be seen how to construct a deterministic algorithm which effectively interleaves all these operations, and this is not surprising: one advantage of dividing the compilation problem into small portions, or “phases”, is that (achieving good results for) each phase may become more tractable when considered in isolation. This is exactly what we did for proceduralization and node scheduling, and optimality for proceduralization seemed (relatively) straightforward to reach; however, as we have discussed, this seeming advantage may in fact be illusory. The quality (by whatever metric) of the final output, e.g. machine code, is what really matters, rather than the quality considering only some factors of intermediate stages along the way.

One possibility is to retain our previous separation into phases, and perform proceduralization according to the *worst* register pressure of any node scheduling, a technique similar to that of Touati [Tou05]. However we see this as both unnecessarily pessimistic and achieving only a

partial separation of concerns. Another possibility may be to interleave proceduralization and node scheduling operations in a recursive traversal of the regions of the prototypical RVSDG; however, it is not clear how predicated execution (which allows interleaving of nodes inside a region with those outside it) would fit into this strategy. Development of such approaches is left to future work; the architecture presented already in this thesis offers numerous opportunities for profitably combining optimizations including many which have been seen as antagonistic.

6.6 Chapter Summary

We have seen how the *node scheduling* phase supports a wide variety of techniques for register allocation, instruction scheduling, and code motion, including a previous VSDG algorithm by Johnson, as a separate phase after proceduralization and PDG sequentialization. We also saw how this scheme still had some vestigial phase-order problems, but that these were minor compared to CFG compilers, and might be handled by a number of schemes (left for future work).

CHAPTER 7

Splitting

Chapters 2 to 6 have developed an architecture for VSDG sequentialization, which operates by progressively refining the trace semantics of the input VSDG until a CFG is reached. This is appropriate for a compiler back-end operating on a VSDG passed to it by a preceding stage on in-place optimization, according to the conventional structure of an optimizing compiler discussed in Chapter 1 (and shown in Figure 7.1).

This Chapter reconsiders the boundary between in-place optimization and sequentialization. Some optimizations do not seem to fit anywhere into this scheme, and this chapter studies in particular the *splitting* transformation, which offers opportunities for enhancing many aspects of a compiler, in particular the number of tests and branches in the output.

Section 7.1 defines the splitting transformation and considers its use as an in-place optimization on the VSDG. Section 7.2 describes how it can also be used on the RVSDG to perform a range of other optimizations which do not make sense in a VSDG context. This raises issues of phase-ordering, which are discussed in Section 7.3 along with some possibilities for how they might be overcome. Section 7.4 considers the application of splitting in more detail, including how it generalizes the NP-complete problem of boolean minimization, and Sections 7.4.1 to 7.4.4 give a range of possible criteria for optimality. Finally Section 7.5 shows how a wide range of existing techniques are related to splitting and might be used as guiding heuristics.

7.1 Splitting, *à la* VSDG

Firstly, we consider the splitting transformation on the VSDG as an in-place optimization changing the program's trace semantics prior to sequentialization. (This is consistent with our architecture of semantic refinement in Section 2.7.1.) In this context, the essence of the transformation is as shown in Figure 7.1:

- Let $\mathfrak{t}$ be a transition and $\mathfrak{g}$ be a γ-node;
- Make a fresh copy of $\mathfrak{t}$, and call this $\mathfrak{t}'$.

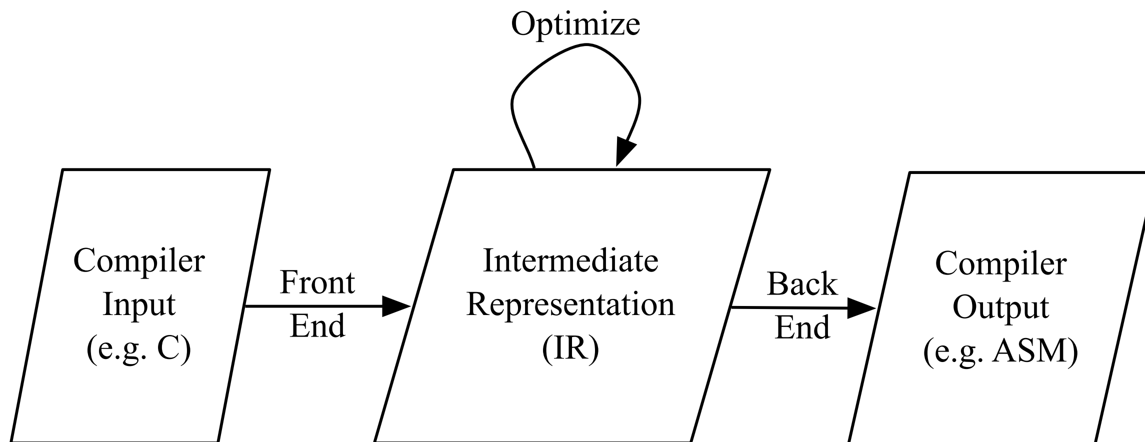


Figure 7.1: Structure of an optimizing compiler (repeated from Figure 1.1.) For the VSDG, the main task of the back end is *sequentialization* of the VSDG into a CFG of machine instructions with physical registers.

- For each result $s \in \mathfrak{t}^\bullet$ and corresponding $s' \in \mathfrak{t}'^\bullet$:
  - Add to $\mathfrak{g}$ a new *component* c (Section 2.6.2) choosing between s and s' (that is, the true and false operands of c are s and s', respectively);
  - Redirect all the original edges to s to the result place of c.
- Where there was a path from $\mathfrak{t}$ to $\mathfrak{g}$ or vice versa, this results in cycles; these can be fixed by duplicating nodes on the path and/or changing uses of the results of $\mathfrak{g}$ into uses of its corresponding true or false operands.

We say $\mathfrak{t}$ is *split across* $\mathfrak{g}$.

A special case of splitting, shown in Figure 7.1(b), is the *node distribution* transformation, where $\mathfrak{t}$ operates directly on $\mathfrak{g}$'s results ($^\bullet\mathfrak{t} \cap \mathfrak{g}^\bullet \neq \emptyset$); here, fixing cycles results in $\mathfrak{t}$ (resp. $\mathfrak{t}'$) operating directly on the corresponding true (resp. false) operand of $\mathfrak{g}$. We can see $\mathfrak{t}$ as being "pulled" upwards through the γ-node $\mathfrak{g}$, and say that $\mathfrak{t}$ is *distributed across* $\mathfrak{g}$.

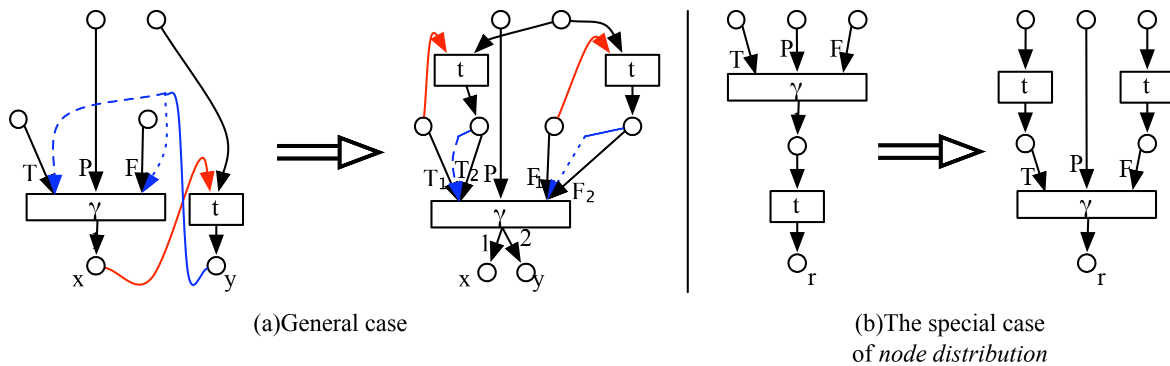
This operation serves two main purposes, which we consider in Sections 7.1.1 and 7.1.2.

7.1.1 Enabling Optimizations

The duplication of $\mathfrak{t}$ allows each copy to be specialized to either true or false, providing opportunities for a wide range of other optimizations: for example, constant propagation, algebraic strength reduction, devirtualization (e.g. due to pre-existence [DA99], or exact type information). These potential benefits accrue from two distinct effects:

1. Knowledge of a particular value for an argument, rather than a merge (i.e. γ-node) of several values
2. Knowledge of control environment

The first of these occurs only in the case of node distribution, where replacement of $\mathfrak{g}^\bullet$ by one or both of the true or false operands to $\mathfrak{g}$ enables optimization. Such cases can be captured



(a)General case

(b)The special case of *node distribution*

Figure 7.2: The *Splitting* transformation on the VSDG. (Note the red and blue arrows, potentially representing chains of nodes, are shown for completeness; they may not both exist in the same VSDG because of acyclicity.)

by expressing transformations as local unconditional rewrite rules. For example, Figure 7.7(a) shows it enabling a constant propagation transformation.

The second effect occurs where knowledge enabling optimization of t can be gleaned from the predicate of g; for example, Figure 7.3(a). This can be captured by *conditional* rewrite rules, paralleling the Sparse Conditional Constant Propagation (SCCP) optimization on the CFG [WZ91].

Sometimes, the two effects must be combined to enable an optimization, as in Figure 7.3(b).

Tracking Dataflow to Enable Optimizations. Whilst it is straightforward to see that knowledge of the predicate $x = 0$ would enable optimization of $x + 1 = 1$ to true—$x + 1$ would first be optimized to 1, and then $1 = 1$ to true—and a reasonable model of predicates would allow knowledge that $x > 5$ to enable rewriting of $x > 0$ to true or $x < 6$ to false, gaining the full benefit of such knowledge is not quite as easy. Firstly, information must be propagated through nodes: $x > 5$ does not enable optimization of $x - 1$ in any way, but still implies $(x - 1) > 4$, despite neither $x - 1$ nor 4 appearing in $x > 5$. Secondly, a further annoyance is that $x > 5$ does not imply $x + 1 > 6$ (because of overflow), unless you have that $x < \texttt{MAX_INT}$ too.

Speculation and Safety Note that the VSDG's semantics allow "speculative" evaluation of each copy of t even if the predicate takes the opposite value. Thus, it might appear that specialization according to knowledge of the predicate is unsafe. However, as long as such specialization does not introduce the possibility of exceptions or nontermination, this is not the case: there is no problem in speculatively computing a value whose *correctness* is only guaranteed if the value is used.

7.1.2 Transformations on γ-Nodes

Besides t being an arithmetic operation, the VSDG's uniformity of expressions (Section 2.1.1) means that t could equally be another γ-node to be split or distributed across g. On the VSDG,

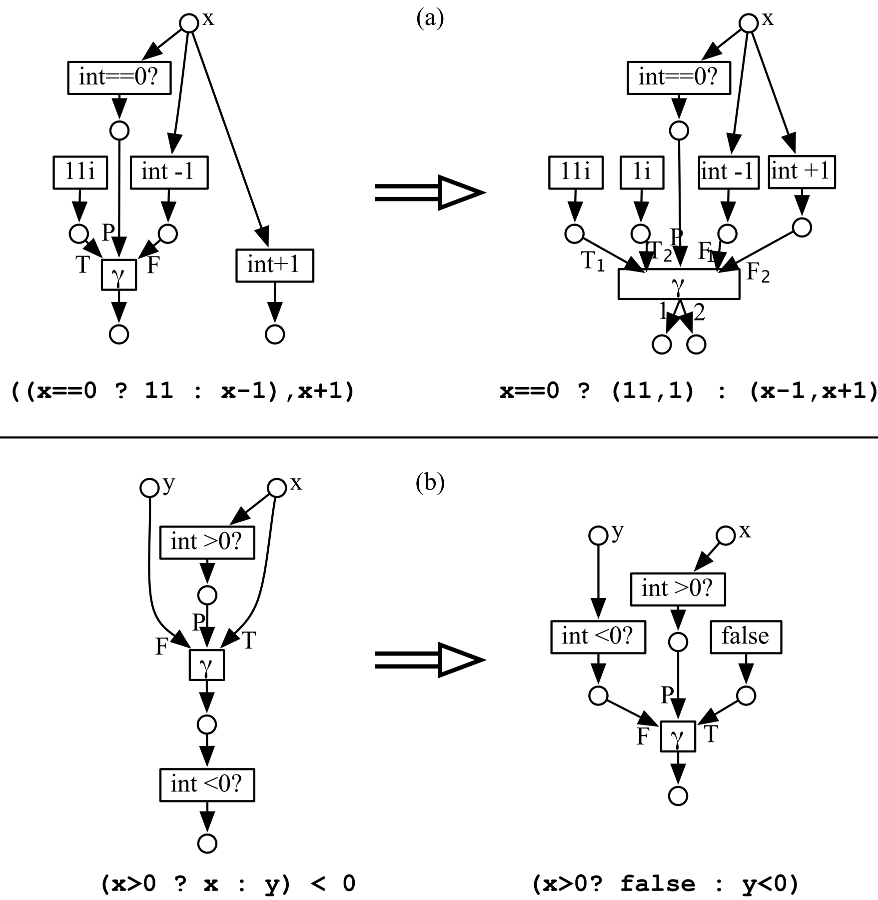


Figure 7.3: Splitting enables optimization due to knowledge of the γ-node predicate (in (b), *combined* with knowledge of argument)

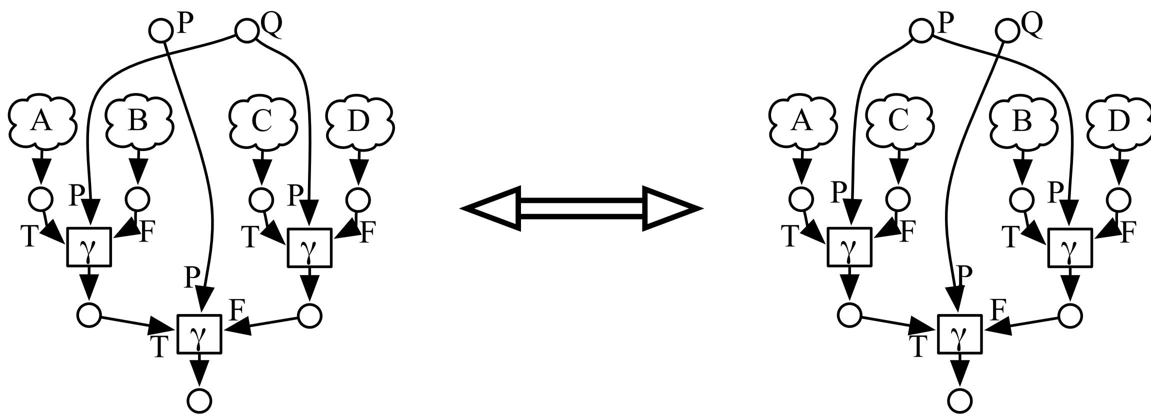


Figure 7.4: Reordering trees of γ-nodes by Splitting

this encompasses a range of useful optimizations, specifically:

Tupling γ-nodes Where $\mathfrak{t}$ and $\mathfrak{g}$ are both γ-nodes with the same predicate, specialization of $\mathfrak{t}$ to true or false allows it to be collapsed down to the respective operand[1]. This effectively combines into a single γ-node all the components of both $\mathfrak{t}$ and $\mathfrak{g}$ and allows a single reduction (or corresponding control flow merge) to select values for all of them.

Reordering γ-nodes Where $\mathfrak{t} \in \mathfrak{g}^{\bullet\bullet}$ is *distributed* across $\mathfrak{g}$, if $\mathfrak{t}$ is a γ-node then this has the effect of switching the two γ-nodes round, such that their predicates are tested in a different order. This is shown in Figure 7.4.

Tests and Branches Both of the above forms of transformation may change the number of (dynamic) tests and branches which will be performed on some or all computation paths. (Specifically, tupling can only reduce this number, whereas reordering can both reduce and increase it.) We have already mentioned in Sections 3.1 and Section 4.4.2 that it is desirable to reduce the number of tests and branches in the output, and that Upton's criteria for the optimality of a sequentialization does not take this into account; such transformations offer a number of possibilities for enhancing his criteria to include this aspect. These are considered in Sections Section 7.4.3 and 7.4.4 below.

7.2 Splitting in the RVSDG

It is also possible to perform the splitting transformation on the *RVSDG*—that is, at some point during sequentialization by semantic refinement. In this context, splitting explicitly moves the two copies of $\mathfrak{t}$ into the true and false subregions of $\mathfrak{g}$'s γ-net, as shown in Figure 7.5.

Such application generalizes two operations we have seen already:

1. The node distribution performed by the PRE algorithm of Weise et al. [WCES94], shown in Figure 4.12(a-VSDG) and (b-VSDG) (discussed in Section 4.4.5).

[1]This is another case of optimization according to knowledge gleaned from the predicate of $\mathfrak{g}$—that is, the predicate of $\mathfrak{g}$ confers exact knowledge of the value of the predicate operand to $\mathfrak{t}$.

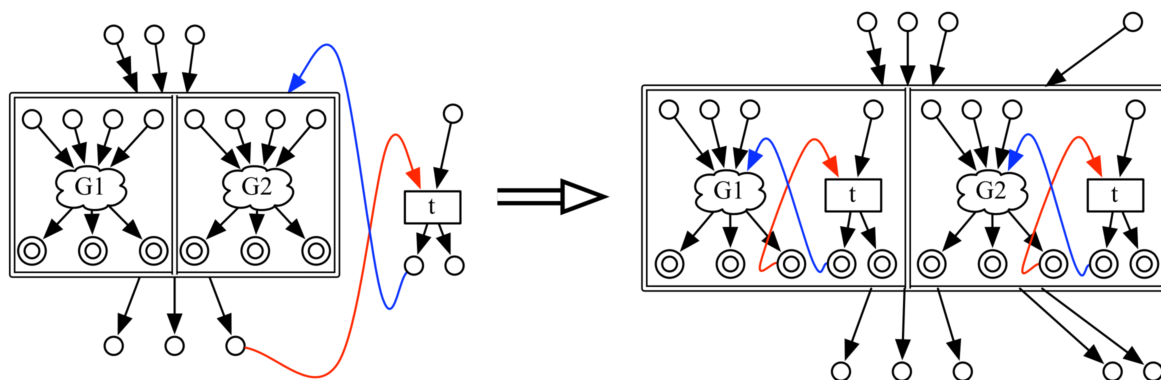


Figure 7.5: The *Splitting* transformation on the RVSDG. (Note the red and blue arrows, potentially representing chains of nodes, are shown for completeness; they may not both exist in the same RVSDG because of acyclicity.)

2. The γ-ordering transformation used in Section 3.4.1 to deal with independent redundancies (as shown in Figure 3.2), reformulated on the RVSDG in Section 5.5.7.

Key to both of these that the two copies of $\mathfrak{t}$ will now only be performed if $\mathfrak{g}$'s predicate takes the appropriate value. In the case of PRE, this allows one copy (say $\mathfrak{t}'$) to be optimized by hash-consing it with an equivalent node already existing *in the appropriate subregion* (true or false). We can see $\mathfrak{t}'$ as being specialized to the availability or precomputation of its own value. In the case of γ-ordering, extension of the same idea allows each copy to be specialized to the precomputation (or otherwise) of the independent redundancy node.

Clearly, these operations and ideas are not appropriate to the VSDG: they deal with whether expressions are computed or not, and this is a matter of evaluation strategy.

7.3 Splitting: a Cross-Phase Concern

In the previous sections we have seen both how splitting can be used to enable a variety of in-place optimizations on the VSDG, and how it is also useful on the RVSDG for optimizations specific to a regionalization.

This might suggest performing *two* passes of splitting, at distinct stages of compilation. However, in fact it is likely that *all* applications of splitting will wish to take into account the evaluation strategy selected during regionalization: if $\mathfrak{t}$ and $\mathfrak{g}$ will not be executed during the same program run, splitting $\mathfrak{t}$ across $\mathfrak{g}$ will introduce a test of $\mathfrak{g}$'s predicate into program runs previously demanding only the value of $\mathfrak{t}$. This is generally not desirable—especially if the aim of the splitting transformation was to *reduce* the number of dynamic tests and branches (Section 7.1.2).

(The RVSDG's regions capture this criterion: if the least common ancestor (LCA) to $\mathfrak{t}$ and $\mathfrak{g}$ is a γ-net, one will be executed only if true, and the other false; if the LCA is a strict net, both may be evaluated.)

This suggests that what we have seen as "in-place" optimization on the VSDG needs to be combined with sequentialization in some way.

Splitting as Part of Proceduralization Recall from Chapter 5 that proceduralization can be seen as the specification of sharing partitions on the nodes of a special case of *Unshared VSDG* (u-VSDG). A simple extension of this viewpoint naturally incorporates some selection of splitting transformations into proceduralization, as follows. Define a *Distributed VSDG* or d-VSDG as an u-VSDG in which *node distribution* has been applied to every (non-γ-node) computation t across any γ-nodes $g \in {}^{\circ\bullet}t$ as much as possible. By taking an appropriate view of trace semantics[2], we can see such a d-VSDG has the same trace semantics as the original VSDG or any which can be obtained from it by reversing the application of node distribution.

Now consider the effect of tail-sharing partitions, using as example the transformed VSDG of Figure 7.7(b) (on page 148—this is a d-VSDG as defined above):

- Where the two `int+` nodes are in distinct partitions, this specifies an RVSDG containing two `int+` nodes and executing one or the other *before* the merge;
- Where the two `int+` nodes are part of the same partition, this specifies an RVSDG containing a single node which is executed *after* the merge.

Further, observe that this special case of tail-sharing—where the operation is always required—*never* results in duplication during PDG Sequentialization, whereas the general case of tail-sharing can. This completes the parallel between tail- and reuse-sharing mentioned in Section 5.3.1: the equivalent special case of reuse-sharing, where the operation is always required, *never* results in speculative evaluation, whereas the general case can.

Further Details of how to resolve this phase-ordering problem are not considered in this thesis, but briefly we see two possibilities:

1. The in-place optimization stage "looks ahead" to the later sequentialization stage, with some representation in the VSDG of the RVSDG or CFG that might be constructed from it
2. The optimization and sequentialization phases are combined, so that all in-place optimizations can be moved to appropriate points in sequentialization.

Phase-Ordering in Sequentialization Any application of splitting on the RVSDG prior to a node scheduling stage also raises phase-order problems: by forcing some computations to be done together, splitting affects register pressure and instruction latency issues. This is shown in Figure 7.6. Thus, we would like node scheduling to be able to choose where splitting was applied. To this end, Johnson included a restricted form of tupling (where the γ-nodes were not reachable from each other—capable of transforming Figure 7.6(a) into (b) or (c), only) in his nondeterministic framework for RACM algorithms (discussed in Section 6.3.3). However, his "snowplough" algorithm (Section 6.1) never performed this operation.

[2]We discussed the VSDG's trace semantics in Section 2.7.3, leaving the correspondence between reduction steps and program operations informal. Thus, here we consider traces as being sequences of only ALU instructions, corresponding to reductions of arithmetic transitions, omitting control-flow merges (these are not machine operations, even if in some views they might correspond to γ-node reductions) and conditional branches (as the VSDG does not contain corresponding split nodes).

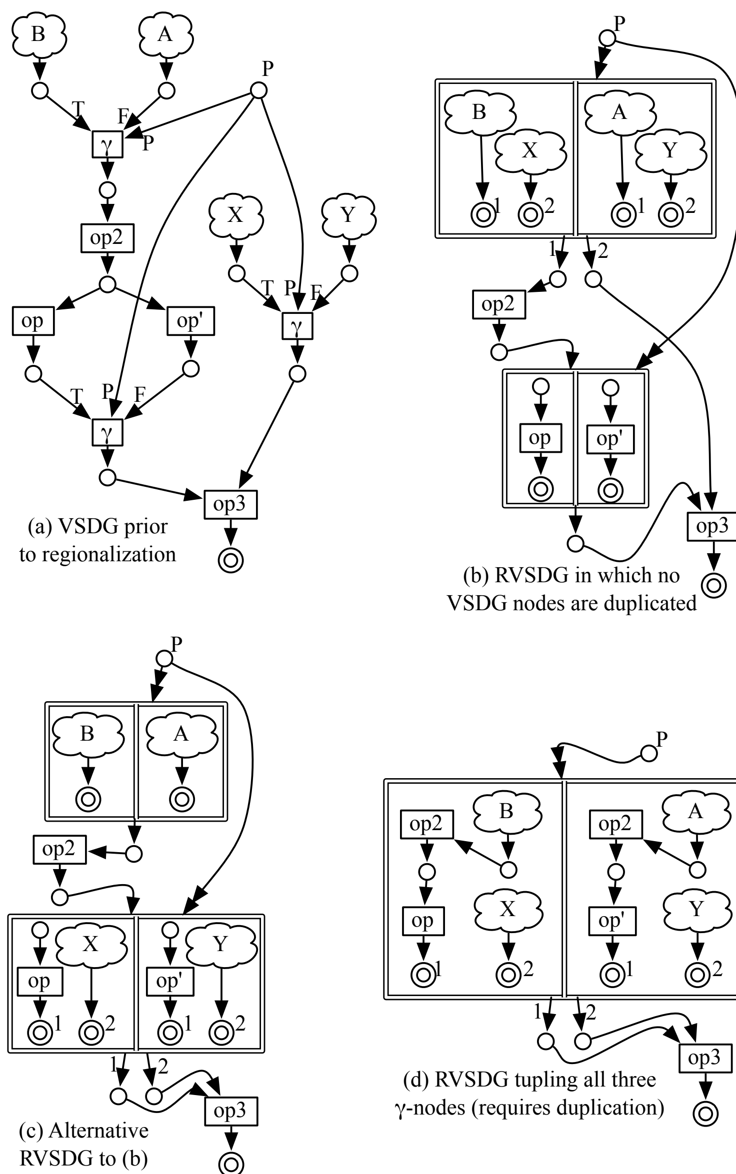


Figure 7.6: A VSDG containing two γ-nodes, and three corresponding but semantically-different RVSDGs. (d) has two copies of `op2`; A must be scheduled in amongst the sequence X`; op2; op;` and B in amongst Y`; op2; op';`. Contrastingly, in (b) and (c), the decision as to whether to schedule A (resp. B) together with X or `op` (resp. Y or `op'`) has already been made.

7.4 Optimality Criteria for Splitting

In Section 7.1 and 7.2 we saw a range of uses of splitting on the VSDG, and also the RVSDG. This section considers the problem of deciding *which* splitting transformations should be performed.

For Enabling Optimizations (Section 7.1.1), and including the tupling of γ-nodes (Section 7.1.2), the use of splitting generally increases code size (by duplication), but does not increase the length of any dynamic path, and may reduce it if other optimizations are enabled. Thus, the application of splitting represents a classical space-time tradeoff, and a number of possible criteria of (sub-)"optimality" may be defined by prioritizing dynamic benefits and static costs, as follows:

Optimal Merge Placement, captures the point at which all possible benefit has been derived from *node distribution* transformations, only (discussed in Section 7.4.1);

Control Flow Preservation captures the point of maximum benefit without changing the control flow of the program (Section 7.4.2);

Limited Optimal Splitting achieves the same level of elimination of redundancy for predicate tests as classical PRE does for computations, but does so while treating branching uniformly with computation (discussed in Section 7.4.3), and

Exhaustive Optimal Splitting extends this approach and unifies it with *complete* PRE (discussed in Section 7.4.4).

For Reordering γ-Nodes (Section 7.1.2), the application of splitting is not a classical tradeoff: some reorderings benefit *both* space and time. (However, others are incomparable even on time alone!) Reordering can also be desirable for normalization purposes. However consider the problem of finding the best tree of γ-nodes (in terms of both static size, and number of runtime tests) to select between just two values: such a tree can be seen as a Free Binary Decision Diagram [SW01] selecting between true and false, and finding the best tree is equivalent to finding the best variable ordering. Alternatively, consider a multidimensional array whose elements are the two values and whose axes are the predicates tested by the γ-nodes; this is clearly a Karnaugh map, and finding the best tree is equivalent to the boolean minimization problem. Both of these are well-known to be NP-complete.

General cases can be further complicated by different leaves returning different functions of a common argument, and the use of tail-sharing. (Tail-sharing allows a single static computation of each value *iff* that value is always returned from the tree without modification; the tree must still decide to which computation a tail-call should be made.) However, reordering can still produce benefits, such as bringing together unspecializable instances of the duplicated nodes: this allows them to be shared *after* the merge of values as shown in Figure 7.7(b).

For γ-Ordering (on the RVSDG, Section 7.2), the NP-completeness (of finding the best tree to decide whether or not to evaluate the independent redundancy) is assured by the same argument.

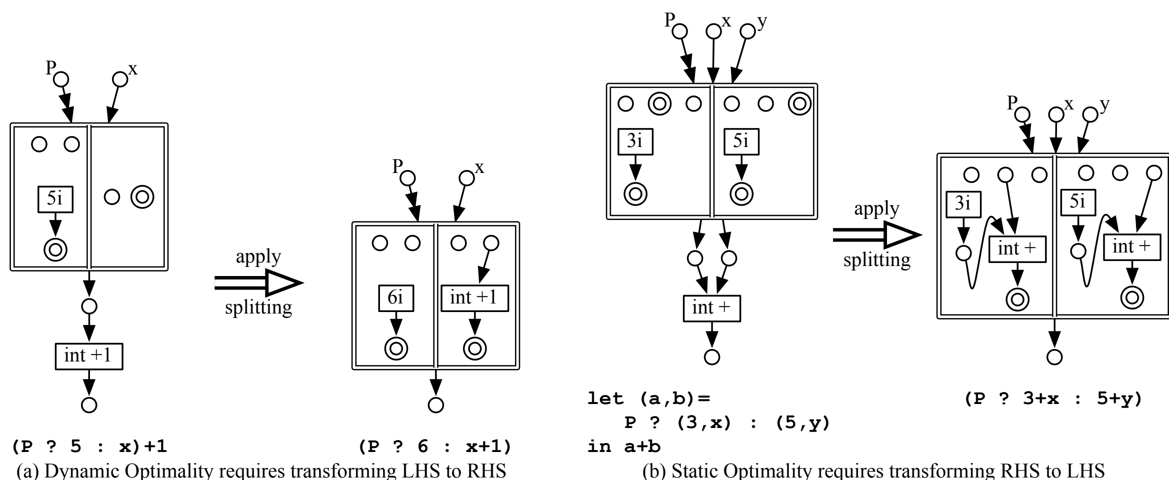


Figure 7.7: Static and Dynamic Optimality of Merge Placement

7.4.1 Optimal Merge Placement

Firstly, we consider *only* applying the node distribution transformation. Similar metrics of *static* and *dynamic* optimality apply here as for the problem of VSDG sequentialization (Section 3.1) and are shown in Figure 7.7:

- A VSDG has *dynamically optimal merge placement* if there is no result $s \in t^\bullet$ which is passed to t' by a chain of γ-nodes, such that an unconditional rewrite rule could be applied if the γ-nodes did not intervene (i.e. if $s \in {}^\bullet t'$). For example, Figure 7.7(a) shows a VSDG which is not dynamically optimal, and a transformed version which is.

- A VSDG has *statically optimal merge placement* if there is no γ-node g whose true and false transitions both return results from operations of the same type. For example, Figure 7.7(b) shows a VSDG which is not statically optimal, and a transformed version which is.

However, note the RVSDG's specification of execution conditions is important here. A transition $t \in G$ where G is (hierarchically) contained in a net G', cannot be split across a complex γ-node $t_g\langle \ldots \rangle \in G'$ without first *speculating* t—that is, moving t outside G and into G'. This would cause the copies of t to be executed when the original was not (even after splitting), as shown in Figure 7.8(a) and (b). As (c) shows, this can be prevented by first duplicating t_g and predicating the copies on the γ-nodes controlling execution of t—that is, by making t_g subsidiary by γ-ordering—however this means that control flow is not preserved, and exponential growth may result. (A particular case of this is an operation t with two *different* γ-node operands, which can be optimized only in *one* case out of the four: pulling t through one γ-node then leads to the situation in Figure 7.8(a)).

7.4.2 Control Flow Preservation

An alternative criteria for optimality is thus, "the best solution that does not require modifying the control-flow structure of the program", as encoded by the γ-nodes. That is, γ-nodes may

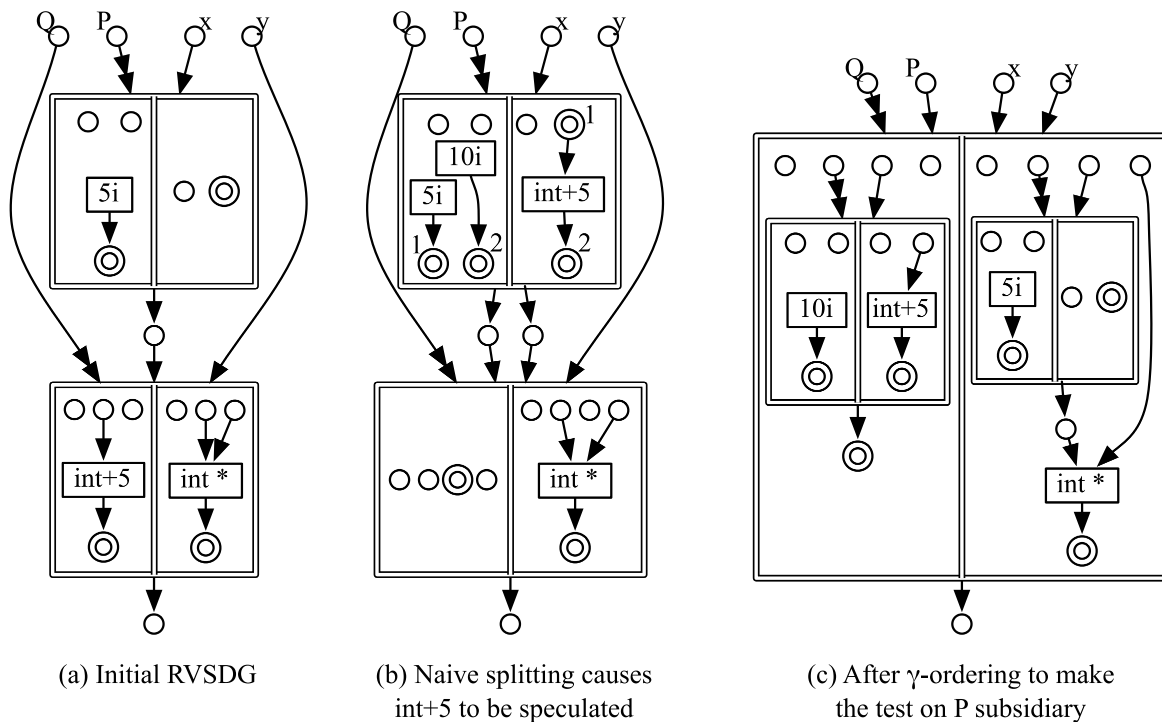


Figure 7.8: Optimal Merge Placement can lead to Exponential Growth

not be duplicated, but within this restriction, any node t (potentially complex, so long as it does not enclose a γ-node) may be split over any γ-node executable in the same program run.

For example, both dynamic and static optimality would be obtained in Figure 7.7, and splitting would also occur in the case of Figure 7.3. However many transformations are ruled out, including node distribution of one γ-node across another (Figure 7.1.2), and distribution of a transition t across a γ-node executed more frequently than t (as this requires first duplicating the γ-node, as in Figure 7.8).

Further, some transformations allowed by this criteria can lead to γ-nodes with other γ-nodes as predicates, and eliminating these by the rule for conditional predicates (Section 3.2.3) would require changing the control flow. For example, if P implies Q then Q can be rewritten into $\gamma^Q = \gamma(P, \mathsf{true}, Q)$, and a node $\gamma(\gamma^Q, A, B)$ would normally be transformed into $\gamma(P, A, \gamma(Q, A, B))$. (However, this would not lead to code expansion *if* the two copies of A can be tail-shared.)

Unlike the other definitions here, control flow preservation ensures sub-exponential (specifically, quadratic) growth: for each of the $O(n)$ non-γ nodes in the RVSDG, at most one copy can be created for each of the $O(n)$ γ-nodes. However note that it does not necessarily lead to the smallest program: reordering γ-trees can allow code size to be reduced, as discussed in Section 7.4 above.

7.4.3 Limited Optimal Splitting

If we remove the restriction of preserving control-flow and allow γ-nodes to be duplicated as other nodes, an intuitive idea of optimality is that each transition t should be split over any

γ-node g which would enable optimization of t if g "must be tested"—i.e. if every execution of t entails execution of g in the same program run[3]. (This is the case when either the LCA region of t and g is the *parent* of the complex node containing g or all intervening ancestors of g are strict nets.)

Because of the VSDG's uniformity of expressions, this *includes* the case where t is itself a complex node testing either the same predicate as g or some related one (i.e. implied by it), as in such cases at least one copy of t can be collapsed down to either its true or false sub-net. This is a nice example of the contrast with the CFG's two-level structure of nodes and statements: the same criteria can be applied to both computations and branching, merely by changing primitive operation nodes into complex (γ-net) nodes.

Thus, this definition enhances Upton's definition (given in Section 3.1) of the optimality of a VSDG sequentialization by giving precise restrictions on the number of runtime tests and branches a sequentialization may perform—and perhaps even completes it[4].

For non-γ-nodes t, this means that t must only be split in cases producing *two* copies of t—there is no requirement to pull t up to the leaves of a *tree* of γ-nodes if they would enable optimization. However, if t is also a γ-node (i.e. the enabled "rewrite" is tupling of two γ-nodes with the same predicate, or predicates related by implication), note that the rewriting works both ways: if γ_1 can be collapsed according to knowledge of the predicate of γ_2, then γ_2 can equally be collapsed according to knowledge of the predicate of γ_1. (Even if the predicates aren't the same, we have $P \Rightarrow Q \equiv \neg Q \Rightarrow \neg P$.)

Thus, limited optimal splitting says, if a strict net G exists as parent of one γ-net (it *must* test P), and ancestor of another (it *may* test Q), then the two γ-nets need to be combined, eliminating the redundant testing. An interesting comparison, shown in Figure 7.4.3, is that this removes the same amount of redundancy in testing-and-branching, as classical PRE on the Control Flow Graph does for assignment: observe that if the SESE regions (indicated by dotted outlines on the CFG) performing testing-and-branching were instead single nodes, classical PRE would remove the redundancy. However, the result can be exponentially bigger than the input source code, in the same way as for Optimal Merge Placement (shown in Figure 7.8 and discussed in Section 7.4.1).

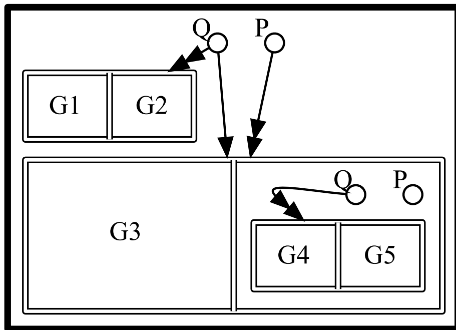
7.4.4 Exhaustive Optimal Splitting

Another, stronger, intuitive criteria is that each transition t should be split over any γ-node g which would enable optimization of t if g "may be tested"—i.e. whenever *any* program run exists in which both execute (clearly, the splitting of t over g should only happen in those program runs involving both!).

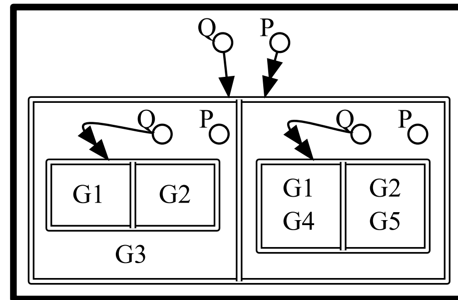
This means that splitting must be applied even if g is deeply nested within some γ-tree; in such cases, t (along with ancestral regions specific to t and not g) must be split across the γ-net ancestors of g until g is reached. In the case of splitting γ-nodes, this means that every time the LCA of two γ-nets (with the same or related predicates) is a strict net, they must be combined. This is a more symmetric requirement than Limited Optimal Splitting (in the previous section):

[3] A weaker criterion would be to only require splitting if, additionally, execution of g implies execution of t; this is akin to control flow preservation but does allow the elimination of repeated (branch-and-merge) structures.

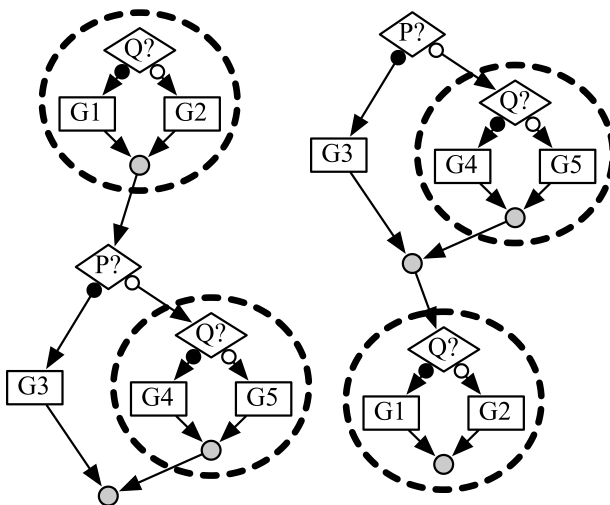
[4] We discussed in Section 4.4.2 that the flaw in Upton's definition was that it took no account of the dynamic number of tests and branches. Of course, even the restriction of Limited Optimal Splitting considers only tests of predicates in the source code, and there are further possibilities of removing tests entirely, and introducing new tests, not considered in this thesis.



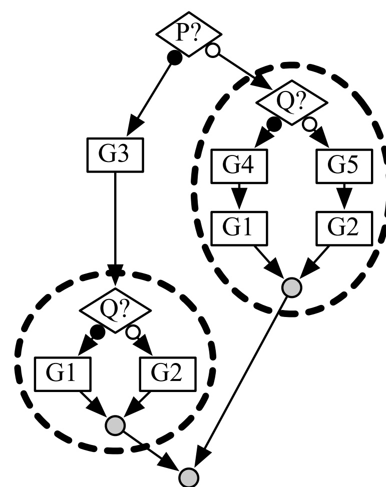
(a-VSDG) Repeated testing and branching on Q fails to satisfy even limited optimal splitting



(b-VSDG) After transformation to make P dominant; satisfies optimal splitting



(a-CFG) Two possible CFGs; note the redundancy of the second branch on Q



(b-CFG) One corresponding CFG (other orderings may be possible)

Figure 7.9: Limited Optimal Splitting corresponds to Classical PRE (see text pg. 150). (We omit: an alternative to (b) in which the test on Q is dominant, also satisfying limited optimal splitting; many alternative orderings in (b-CFG); and many possible tail-sharings of nodes in (b), both CFG and VSDG)

the intuitive criteria above implies that γ-node t must be split across g if and only if g must be split across t. (It does not specify which ordering to choose!)

As shown in Figure 7.4.4, Exhaustively Optimal Splitting thus removes *all* potential redundancy in testing and branching, paralleling the requirement of Upton's optimality criteria for removing *all* redundant node evaluation (i.e. the greatest degree of redundancy that could be eliminated by *any* set of program paths); thus, this is the intuitively cleanest way to complete his definition. However we conjecture that the resulting code growth will be too large for it to be practical.

7.5 Relation to and Application of Existing Techniques

In this section we will consider how a range of existing, mainly CFG-based, techniques relate to the transformations outlined above.

Whereas the VSDG makes a wide space of optimizations easy to reach, such that the point of optimality is hard to identify (or even define), many of these CFG techniques define and give polynomial-time algorithms for "optimal" solutions to particular space-time tradeoffs, thus suggesting clear parallels that could be applied to the VSDG.

7.5.1 Message Splitting

Chambers and Ungar developed a CFG-based splitting transformation which delays a control flow merge and duplicates its would-be successors, as a technique for optimizing object-oriented programs—long a challenge for compiler researchers, as the short method bodies and frequent virtual method calls mean that traditional optimizations such as redundancy elimination and instruction scheduling have only very short blocks of code to operate on, and whilst in procedural languages inlining could be used to ameliorate this, indirect calls make inlining very difficult.[5]

The splitting transformation is particularly useful on object-oriented programs because of the interaction with guarded inlining[6]: each call site results in a merge point where the inlined body and backup call combine together, and calls (to the same receiver object) following the merge can be distributed to before the merge, allowing further inlining without a second guard/test. This leads to *extended message splitting* [CU90, CU91]—the specialization of an entire method, protected by a single guard.

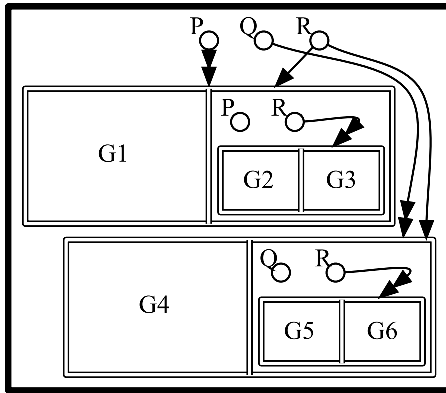
Chambers and Ungar consider a range of controlling heuristics, significantly including:

- *Eager Splitting*—duplicating the CFG nodes after any non-loop merge point onto both predecessors; loop bodies can be similarly split until a fixpoint is reached (as duplicating everything after a loop merge point indefinitely would lead to an *unbounded* increase in code size!)

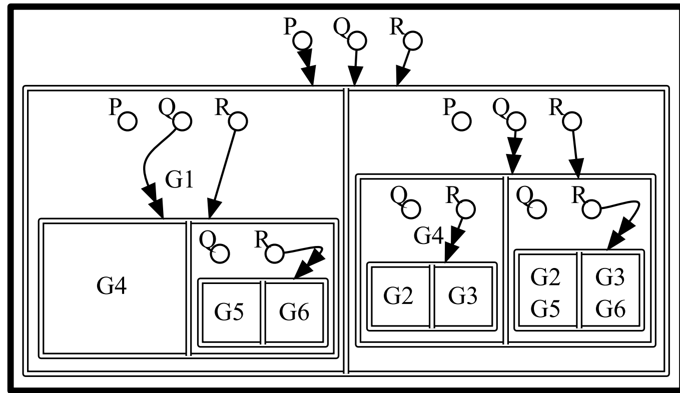
- *Reluctant Splitting* duplicates nodes after a merge point up to the point of a second call to the same object (which can then be devirtualized), subject to the number of nodes

[5]Additionally polymorphic call sites are difficult for CPU branch predictors to handle correctly, although predictors using global history information to exploit branch correlation may avoid this problem in the situations considered here.

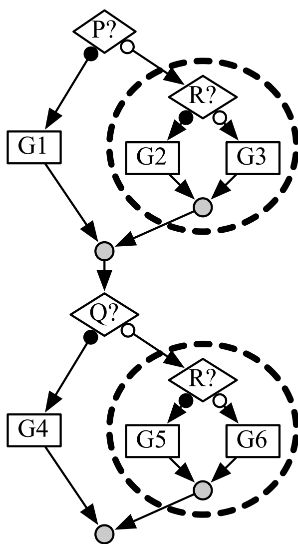
[6]Where a virtual call is replaced by a test of the receiver's class and a choice between an inlined copy of the method body for that class and a "backup" virtual dispatch. This can be done by static heuristics such as receiver class prediction [DS84, CUL89, CU89], or by runtime profiling, e.g. using observed call-graph edge frequencies [HU94, GDGC95].



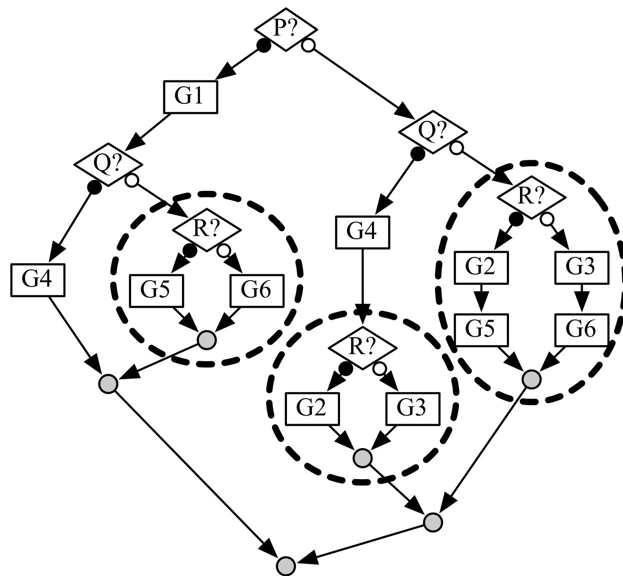
(a-VSDG) Satisfies Limited but not Exhaustive Optimal Splitting



(b-VSDG) Equivalent, satisfying exhaustive optimal splitting



(a-CFG)



(b-CFG)

Figure 7.10: Exhaustive Optimal Splitting eliminates *all* redundancy in branching, as Chapter 3 does for node evaluation; compare (a) with Figure 3.2(a).

duplicated being less than a constant.

Comparison to splitting on the VSDG yields a number of insights:

- In the CFG, the operation to be duplicated (the second call) does not necessarily have any *value* dependence on a result of the merge (the receiver): the information enabling optimization results from the *predicate* (e.g. a class test). Thus, their splitting does not necessarily correspond to node distribution.
- However, they only consider duplicating operations over *earlier* predicate tests, i.e. in the forwards dataflow direction. Thus, their approach seems akin to our *Dynamically Optimal Merge Placement*.
- However, in the specialized context of devirtualization (only), this is not a drawback, as the interactions between virtual method calls (on the same object) are symmetric: each generates a predicate over which later calls may be split, and benefits from duplication and specialization across any test on that predicate generated by an earlier call.
- Their estimate of duplication used by reluctant splitting is somewhat unreliable, due to CFG phase-ordering problems (solved by use of the RVSDG) with code motion operations (specifically PRE or VBE). This is discussed below in Section 7.5.5.

The same transformation was also used by Arnold and Ryder [AHR02], who developed an alternative heuristic using feedback-directed profiling: given some space budget, repeatedly duplicate the CFG node following whichever merge point is most-frequently-executed, until the budget is exhausted or the frequency of execution falls below some constant. Whilst in some sense this is more general (optimizations other than devirtualization/guard-elimination are enabled), it does not measure whether any optimization is actually enabled, and is limited by the overspecification of ordering in the CFG: operations can only be split over predicates and merge points that happen to be ordered before them.

7.5.2 Cost-Optimal Code Motion

Hailperin [Hai98] used the machinery of classical PRE as a framework for enabling other transformations, placing computations (at nodes of the original CFG) so as to minimize their cost measured by a user-supplied metric. Loosely, whereas Lazy Code Motion first moves computations as early as possible, and then delays them as far as possible without reintroducing redundancy (specifically, it does not delay computations past control-flow merges), Cost-Optimal Code Motion moves computations as early as possible, but then delays them only as far as is possible without increasing their cost.

However, in order to ensure an "optimal" solution found by the LCM algorithm exists, Hailperin places two constraints on the cost metrics that may be used:

- When control flow paths merge, the cost of an operation must remain the same or increase if it is placed after the merge rather than before.
- When control flow paths do not merge—this includes the case of a control-flow split—the cost of any operation must remain the same.

(Both constraints apply only in the absence of assignments to the operands or uses of the result.)

The effect of these criteria is that it is not generally possible to exploit knowledge of the predicate governing a control-flow split; instead, only optimizations based on forwards dataflow information (which becomes less accurate following control-flow merges) may be used. Thus, Hailperin's framework can perform *only* node distribution transformations; further, being based on classical code motion, it cannot change the control-flow structure of the program. Hence, we see it as limited to the *intersection* of Optimal Merge Placement (defined in Section 7.4.1) and Control Flow Preservation (Section 7.4.2).

7.5.3 Speculative Code Motion

Scholz, Horspool and Knoop [SHK04] consider the use of runtime profiling information to further reduce the number of dynamic executions. They use the same classical framework of placing computations at existing CFG nodes, but abandon the previous notion of *conservatism*—that is, they allow computations to be introduced onto (i.e. speculated on) paths on which they did not previously occur. Their algorithm finds the code motion transformation minimizing a weighted sum of static size and dynamic executions, allowing the two to be traded against each other. This suggests VSDG heuristics for allowing speculation in both node distribution PRE (Section 4.4.5) and the choice of evaluation strategy in proceduralization (Section 3.1), although it is not clear how to take into account the extra benefit from changing the control flow structure that speculation allows in the RVSDG.

Another potential application for frequency profiling is the reordering of tests, mentioned in Sections 7.1.2 and 7.4. Many possible trees are statically incomparable even only on their time complexity, for example the choice between the following two expressions:

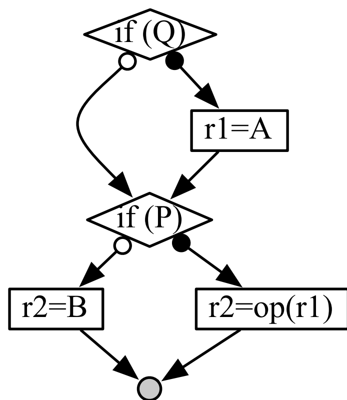
```
if P then A else if Q then A else B

if Q then A else if P then A else B
```

but testing the most-frequently-true predicate would be most efficient. This may be particularly effective in reordering operations which share code for handling exceptional failure cases. Similarly if P implies Q then optimal splitting requires tupling some γ-nodes testing P and Q; dynamically, it is more efficient to make P dominant if P holds more often than $\neg Q$, or Q dominant if the reverse. Such applications are left for future work.

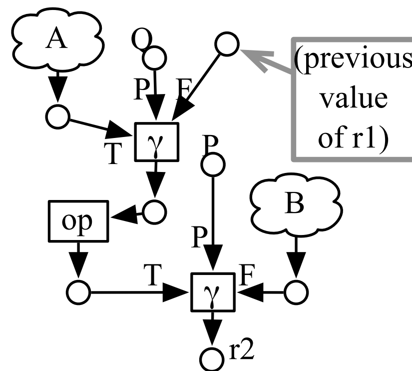
7.5.4 Use of the PDG

Feigen et al.'s *Revival Transformation* [FKCX94] performs PDCE on the Program Dependence Graph by detaching a partially-dead expression (more precisely, a PDG subgraph, thus potentially spanning multiple basic blocks) from the CDG and reattaching or "reviving" it in a new location. Duplication-freedom is assured as the subgraph is attached to only a single parent group node. Significantly, this technique is capable of reordering branches (shown in Figure 7.11), but it cannot rearrange multiple uses of the detached expression to bring them together, nor duplicate the expression (as classical code motion can [KRS94b]) when this would allow a dynamically better solution. Moreover, it performs only PDCE and not PRE (for example, neither CFG Figure 4.4(b) or (c) is transformed into our solution of Figure 4.5).

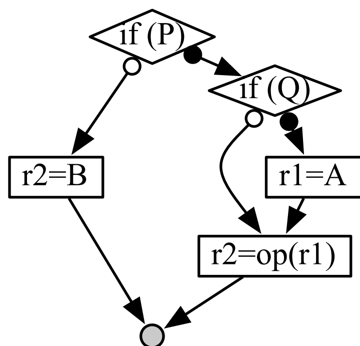


(a) CFG for

```
if (Q) r1=A;
r2=(P ? op(r1) : B);
```



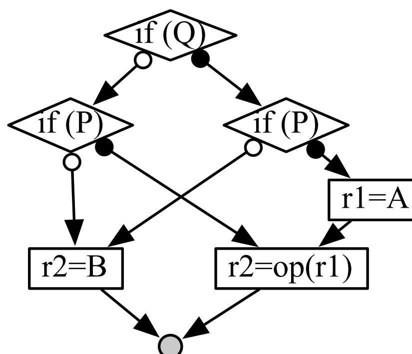
(b) VSDG



(c) CFG for

```
if (P){
   if (Q) r1=A;
   r2=op(r1);
} else r2=B;
```

produced from VSDG
or by Feigen et al



(d) CFG produced by
Bodik and Gupta

Figure 7.11: Comparison of VSDG, PDG, and CFG-restructuring techniques on a PDCE example similar to the “independent” redundancy of Figure 4.4.

7.5.5 Restructuring the CFG

Steffen [Ste96] achieves *total* PDCE by restructuring the CFG. However this is achieved by introducing nondeterminism and backtracking, requiring a substantial overhead in additional control flow.

Mueller and Whalley [MW95] consider applying splitting to the CFG—which they term *path separation*—in order to remove conditional branches. Specifically, they identify a latest CFG node n where the result of some future branch b is known (for example, n is the true or false successor of another branch with the same predicate, and there are no intervening assignments to predicate operands), such that b's outcome is not known at n's successor (which must be a control flow merge). They then duplicate (split) all nodes on the path between n and b, thus hoisting them above the merge; the copy made of b is then statically known to have only a single successor.

This technique is applicable even when the path from n to b has other branches on it (i.e. when b does not postdominate n), and thus is similar to our *exhaustive optimal splitting* (although they do not consider enabling other optimizations). However, such cases highlight the intrinsic limitation of such CFG splitting techniques (in which we include those discussed in Section 7.5.1): the CFG's total ordering (of both branches and statements) means that *all* nodes on the path from the merge to the node to split (b above) must be split (i.e. duplicated) in the same way first. This is shown in Figure 7.12, with the intervening nodes performing an unrelated test, branch and merge on a second predicate Q, before the desired node—a second test on P—is reached: Figure 7.12(a-CFG) shows the input CFG, and (b-CFG) the CFG after application of splitting. Observe that the result performs the same operations in the same sequence as the original; transformation is limited to specializing copies of existing nodes to better knowledge of the paths leading to them.

In contrast, the VSDG removes the artificial ordering and duplicates only those operations which *must* come between the merge and the node to split. Thus, Figure 7.12(c-VSDG) shows the result (so long as *no more than one* of the red and blue dependency arcs exists; if both exist, the solution of (b-CFG) cannot be improved upon), and (c-CFG) the two possible CFGs.

Bodík and Gupta [BG97] apply a program slicing transformation to execute partially dead expressions *conditionally*, only if their results will be used, and achieve complete PDCE on acyclic code. This is at the cost of introducing extra conditional branches, which are subsequently eliminated by application of Mueller and Whalley's CFG splitting technique above [MW95].

In further work, Bodik, Gupta and Soffa [BGS98b] apply the same CFG splitting techniques as an *enabling transformation* for subsequent code motion, and—to our knowledge uniquely—achieve *complete* PDCE without introducing extra runtime tests. Their approach works by identifying *code-motion-preventing* (CMP) *regions*—where the expression is partially-but-not-fully both-anticipatable-and-available—which block code motion by classical techniques; separating out paths where the expression is available from those where it is not allows code motion to proceed. Further, they show how selective application guided by runtime profiling techniques can be used to trade the resulting code growth against speculative evaluation.

In an insightful parallel with the VSDG, CMP regions correspond to the *independent redundancies* of Chapter 3, where naïve treatment of the two γ-nodes resulted in this same branch-merge-branch structure[7]. Their use of CFG splitting to separate out the *paths* where

[7] This structure is shown in Figure 3.4(b) for redundant node evaluation, and Figure 7.4.4(a) for redundant

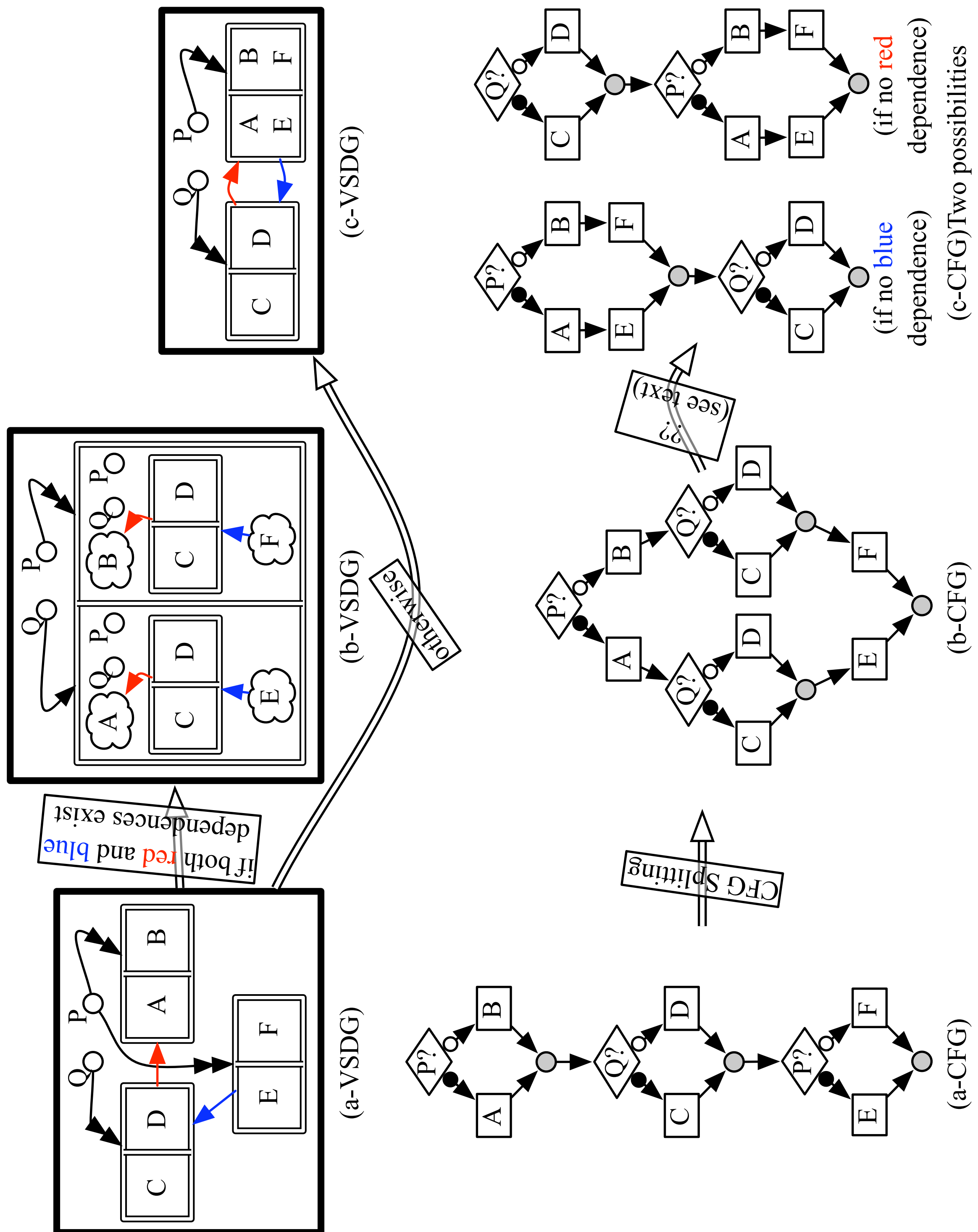


Figure 7.12: Splitting on the CFG and VSDG: A Comparison (see Section 7.5.5)

the expression is available or not parallels our use of the γ-ordering transformation to separate out the corresponding *conditions*. *However*, the conditions capture only the essential essence of whether the computation is available, whereas the paths include both unrelated nodes and branches which provide no useful information, yet which CFG splitting must nevertheless duplicate. Figure 7.11(d) shows another example where the inability to reorder branches leads to an unnecessarily large and complex CFG.

Clearly, the two CFGs of Figure 7.12(c-CFG) can be produced by transforming (b-CFG); perhaps by applying either the Very Busy Expressions (VBE) optimization (subsumed by PRE) or some global restructuring form of instruction-scheduling (akin to full-brown node scheduling on the VSDG, but for example possibly based on SESE-regions) followed by cross-jumping. However, we are not aware of any CFG-based solution combining these two usually-distinct phases, and it seems likely there would be substantial phase-ordering problems: leaving the recombination to a later phase after splitting means that the "costs" (in terms of code size) used to guide the splitting operations will be overestimates or at best inaccurate.

We conclude that restructuring the CFG is a difficult problem; it seems both simpler and more effective to "throw away" the existing CFG structure and build a new one from scratch as VSDG techniques do.

In fact the situation with the CFG is equivalent to operating on a special case of VSDG, in which the (linearly-typed) state dependence chain is used to record all branching and merging of control flow[8], thus totally ordering the branches. Thus, any of these CFG techniques could be applied to the VSDG as a guiding *heuristic*—that is, under the *assumption* of preserving the original CFG's branch (and perhaps statement) ordering. However, the VSDG's usual flexibility in ordering makes this assumption seem somewhat artificial, and suggests possibilities for relaxing the constraint partially or on a fine-grain basis. We leave this as a topic for future work.

branching; in both cases the CMP region is the branch node `Q?` plus the preceding join.

[8]Shortcircuit evaluation in the CFG can be handled by duplication of nodes until later in the compilation process.

CHAPTER 8

Conclusion

The VSDG is a promising basis for the construction of new optimising compilers whose time has now come. Previous obstacles and objections to the use of the VSDG have been comprehensively addressed in this thesis:

Lack of Understanding as to what the VSDG is, what its key features and differences are, are addressed by

- Comparison and relation with the Program Dependence Graph (PDG), and unification of the two using an explicit sharing partition (in Chapter 5).
- The relationship between the VSDG and functional (specifically, lazy) programming languages;
- The importance of continuations and higher-order programming in sequentializing the VSDG.

Sequentialization has been a long-standing problem for the VSDG. Specifically:

- We have addressed the question of how the VSDG may be used as the basis of a compiler, by presenting (in Chapter 2) a framework for sequentialization based upon semantic refinement. The VSDG$\rightarrow$CFG sequentialization process was thus broken into phases using the PDG as midpoint. Further, we have shown how a wide variety of divergent CFG optimizations such as code motion and instruction scheduling fit into this framework.
- Previous approaches have not succeeded in reconstructing effective control flow from the VSDG, losing out to naïve CFG compilers in terms of size and/or speed on common cases such as shortcircuit boolean expressions. We have addressed this issue (in Chapter 3) with a new technique which optimizes the control flow structure of branch-intensive code more thoroughly and systematically than existing techniques. Thus, the maxim of the VSDG: "the best way to optimize control flow is to throw it away".

This thesis has further demonstrated advantages offered by the VSDG in:

Improving Notions of Optimality Many CFG optimizations select transformations which are optimal only among small subsets of all possible transformations. That is, they solve the problem of how best to apply some particular optimization or transformation framework in isolation, and considering only some characteristics of the output. This subset is usually limited to transformations that leave the structure of the CFG unchanged or change it only in particular ways. By throwing away the CFG entirely, VSDG-based approaches make it easier to consider larger universes of transformations.

Raising the Level Of Abstraction Upton [Upt06] argues that we should use the VSDG because it makes implementing an entire compiler *easier*, but this has not proved sufficient incentive due to the ready availability of implementations of and infrastructure the current generation of CFG techniques. However, we have argued that the VSDG's higher level and more abstract representation pushes outwards one of the main limits to optimization—the ingenuity of the human designers of optimization algorithms—and suggests new *targets* for optimization. (Specifically including the reordering of branches, considered in Chapter 7.)

Solving Phase-order Problems Our breakdown of sequentialization into phases of proceduralization (VSDG→PDG) and node scheduling (PDG→CFG) achieves a better separation of concerns than CFG-based approaches with many distinct optimization phases and allows many previously-antagonistic optimizations to be profitably combined. (For example, register allocation, code motion, and instruction scheduling—considered in Chapter 6.)

We have also seen the CFG as a special case of VSDG in which extra state edges are used to restrict the possible orderings, specifically with respect to branches. However, many of the benefits of the VSDG seem to stem from its flexibility in ordering, and this suggests that analyses which remove unnecessary ordering constraints—for example, alias and pointer analyses or the use of linear types, allowing the state to be broken into many small portions—offer benefits in making more optimization opportunities available and/or affordable.

Lastly, the differences between the CFG and VSDG have also demonstrated the principle, suggested in the Introduction (Chapter 1), that more abstraction is better; and consequently for Intermediate Representations, more normalizing IRs are better— **iff** it is possible to select the best output among those corresponding to a given Intermediate Representation form.

8.1 Open Questions

We end with a number of issues that we have not addressed, but that this thesis highlights as being of particular interest or importance for future work:

Optimal Sequentialization of the PDG Although Ferrante et al. have given a necessary and sufficient condition for the existence of a CFG, corresponding to a PDG, without any duplication, the question of how to minimise the duplication where this condition does not hold is still open. In particular, this challenge has not been proven NP-complete nor has a polynomial-time algorithm been presented.

Guiding Heuristics for γ-ordering during proceduralization; splitting, in particular reordering of γ-trees; and node scheduling—both for the various transformations performed by Johnson and more aggressive techniques (discussed in Chapter 6).

Integrating Proceduralization and Node Scheduling on a fine-grain basis, in order to tackle issues of phase-ordering remaining in our architecture.

Quantitative Practical Evaluation of a VSDG-based compiler. We see this in terms of two other questions:

1. How many optimizations would need to be included in a VSDG compiler to make it competitive with existing CFG compilers (which have substantial investments in optimization technology);
2. How performance—of both the compiler and the generated code—would compare between the two techniques iff *all* the same CFG optimizations were implemented on the VSDG. As discussed in Chapter 1, the advantages of the VSDG as Intermediate Representation are likely to increase as increasing effort is put into optimization.

Redeveloping More CFG Optimizations on the VSDG and in our sequentialization framework, to exploit the advantages of both.

APPENDIX A

Glossary of Terms

CDG Control Dependence Graph. Subgraph of a PDG consisting of just nodes and control dependence edges. These allow a notion of "all these statements are to be executed in some order", thus encoding an evaluation strategy but retaining some parallelism.

CFG Control Flow Graph. IR in which nodes are labelled with statements and edges explicitly represent the flow of control from one node to the next.

Construction conversion of a program from source code into an IR

DDG Data Dependence Graph. Subgraph of a PDG consisting of the nodes and data dependence edges, used to restrict possible orderings of execution.

Destruction opposite of construction: conversion of an IR into machine code. For VSDG and PDG, this is performed by sequentialization to a CFG; for CFG, conversion to machine code involves laying out the basic blocks into a single list of machine instructions (not considered in this thesis).

df-PDG duplication-free PDG

df-RVSDG duplication-free RVSDG

Duplication-freedom quality of a PDG or RVSDG, that its statement and predicate nodes (arithmetic and γ-node transitions) can be arranged into a CFG without further duplicating them. That is, indicates its control flow can be implemented with a single program counter.

IR intermediate representation. A structure for representing a program in a compiler, allowing operations of construction, optimization and destruction. This thesis uses three distinct IRs which contain different levels of specification of instruction ordering, namely the VSDG, PDG and CFG.

Node Scheduling Conversion of a df-PDG to a CFG. Parallels the instruction scheduling optimization on CFGs, also including register allocation. Covered in Chapter 6.

Optimization Generically, any change to a program in order to improve its efficiency (in space or time). Also refers to a stage in compilation between construction and destruction where the representation of the program is changed whilst remaining in a particular IR.

PDG Program Dependence Graph. IR containing statement nodes, predicate nodes and group nodes, control dependence edges and data dependence edges. See CDG, DDG. Used in this thesis as a midpoint between VSDG and CFG. (Defined Section 2.3.)

PDG Sequentialization Conventionally seen as the conversion of a PDG into a CFG. However in our architecture we use PDG Sequentialization only to convert from a PDG into the special case of a df-PDG, and leave df-PDG→CFG conversion to a node scheduling phase.

Proceduralization conversion of a VSDG into a PDG

RVSDG Regionalized VSDG. Variant of VSDG equivalent to PDG by explicitly arranging nodes into a hierarchy of regions to specify when evaluation of nodes is conditional on γ-node predicates. Also represents loops using `iter` nodes. (Defined Section 5.5.)

Regionalization Equivalent to proceduralization, but specifically referring to the production of an RVSDG by arranging the VSDG's nodes into regions.

Reuse-sharing Where a single static computation computes a result once, which is then used by multiple consumers.

Sequentialization generically, conversion of a VSDG or PDG into a CFG. See VSDG Sequentialization; PDG Sequentialization.

Tail-sharing Where a single static computation computes one of several possible different dynamic values, because the subsequent use or continuation of each possible value is the same.

u-VSDG unshared VSDG. Special case of VSDG to which a maximum degree of node cloning (Section 2.7.3) has been applied (no transition has more than one consumer). Used in Chapter 5 to unify the PDG and VSDG with an explicit specification of tail-sharing and reuse-sharing.

VSDG Value State Dependence Graph. IR representing a program functionally in terms of values and dependences between them, without representation of evaluation strategy.

VSDG Sequentialization Conversion of a VSDG into a CFG. Performed in this thesis by proceduralization followed by PDG Sequentialization followed by node scheduling.

APPENDIX B

State Edges and Haskell

In this Appendix we will show an encoding of the VSDG's *state edges* into a lazy functional language, using Haskell's *state monad*. This leads neatly to a rigorous definition of the well-formedness conditions whose static enforcement guarantees the dynamic property of state (that every state produced is consumed exactly once) mentioned in Chapter 2 (Section 2.6.7). Lastly we overview some ways in which the VSDG's state edges can be generalized to model a number of other considerations.

B.1 The State Monad

Operations with side effects are often modelled in Haskell using the state monad. States are represented as functional values of some type `state` (e.g. a type parameter to the monad[1]), and a *stateful* value of type T is a function from `state` to $T \times$ `state`—i.e. a function that takes the initial state in which the value is computed and returns not only the value but also the state resulting afterwards. State monads have been widely written about [has] and we refer the reader elsewhere for further details [Mog91, JW93].

In what follows, we assume the representation of states is some abstract type `state` corresponding to a state place in the VSDG, and that operations such as reads and writes, operating on `states`, are provided.

Stateful operations are composed together by use of the monadic *bind* operator, >>=:

```
(>>=): stateful a -> (a -> stateful b) -> stateful b
let x >>= f =
    \init -> let (val,st)=x init in (f val) st;
```

In particular, >>= (and it's cousin >>) ensure that the state existing inbetween evaluation of `x` and `f` is (only) used exactly once. This is the same dynamic requirement as the VSDG makes on states, and thus >> is the basis of the translation.

[1]A special case is the `IO` monad, in which states are machine states, i.e. potentially external to the Haskell VM. The representation of state is abstracted away and a primitive `unsafePerformIO : IO t -> t` exists to allow values to be read out of the monad, but this does not behave as a pure function as it may e.g. perform disk operations.

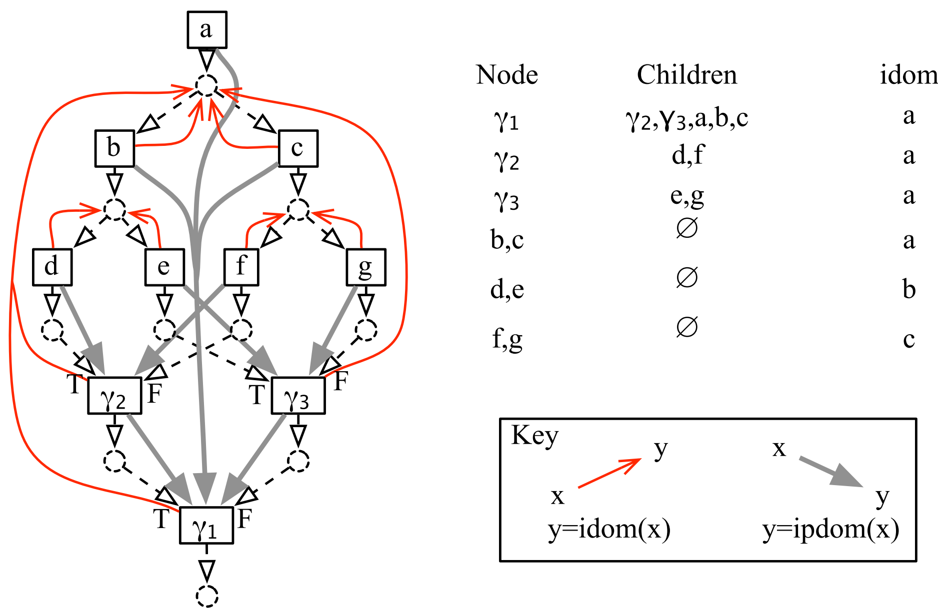


Figure B.1: A complex example in which states are still used linearly; encoded into Haskell on page 168

B.2 Encoding Just the State Edges

First we explain a simplified version of the encoding, in which we consider *just* the state subgraph—that is, state edges and stateful nodes only. Further, we will let the representation in Haskell of each transition a or b be simply `OPER 'a'` or `OPER 'b'`, where a or b is a unique name given to each node[2] and assume that the predicate of any γ-node γ_i is simply available in a global variable `Pi` (as the state subgraph does not include predicates or the edges to them). Nonetheless, this simplified encoding—suitable for e.g. representing the state as a list of actions performed—demonstrates key issues as to how state splits and merges are handled. Values are added in Section B.3 to make a complete encoding.

Straight line code is handled as in `OPER 'a' >> OPER 'b' >> ...`, beginning with the predecessor of the function header, and terminating after the function's result place with `return ()`. This yields an expression for the whole function of type `stateful unit`.

Encoding of γ-nodes is more difficult and will be explained with the example of Figure B.1. The idea is to output a `let` defining a variable of type `stateful unit` for each node (recall all these nodes are state-producing), including γ-nodes—these produce `if` statements gathering other definitions together. The definitions use >> internally as necessary. Finally, when the last γ-node—that is, the merge corresponding to the split—is encountered, this is output as the body of the `let`-expr, and the enclosing chain of >>'s continues.

Thus, for Figure B.1, we get:

```
((OPER 'a') >>
    let b=(OPER 'b') in
    let c=(OPER 'c') in
    let g2=(
```

[2]In particular, we write `gi` for a γ-node γ_i.

```
            let d=(b>>OPER 'd') in
            let f=(c>>OPER 'f') in
            if P2 then d else f
    ) in
    let g3=(
        let e=(b>>OPER 'e') in
            let g=(c>>OPER 'g') in
            if P3 then d else f
    ) in
    if P1 then g2 else g3
)
```

Algorithmically, this can be achieved by a traversal of the postdominator tree, in a similar fashion to the proceduralization algorithm of Chapter 3. We write the traversal algorithm as $encState(t)$, for a transition t; an informal description is given below.

Key to the algorithm is the immediate *dominator* $\mathsf{idom}(t)$ of each transition t, as this is the last operation (in terms of the *flow of execution* in the resulting program) that either *must have executed* when t does, or *must be executed*: execution of $u = \mathsf{idom}(t)$ is performed by the call to $encState(\mathsf{ipdom}(u))$.

Thus, there are three main cases when $encState$ visits a transition t, with children $C = \{t' \mid \mathsf{ipdom}(t') = t\}$ and immediate dominator $u = \mathsf{idom}(t)$:

1. $u \in C$. In this case, $encState(t)$ must make u happen itself, so the result is `(` $encState(u)$ >> $doLetExprs(C \backslash u)$ $code(t)$ `)`, where $doLetExprs$ simply outputs `let` t'=$encState(n')$ `in` for each node t' in its argument set, in topological sort order (this was discussed in Chapter 3).

2. $u = \mathsf{idom}(\mathsf{ipdom}(t))$. That is, u is unchanged from t's parent (this ensures $u \notin C$). Thus, $encState(t)$ leaves it to the enclosing $encState(\mathsf{ipdom}(t))$ to execute u (perhaps by leaving it to a further enclosing call), and the result is just `(`$doLetExprs(C)$ $code(t)$`)`

3. Otherwise, $u \neq \mathsf{idom}(\mathsf{ipdom}(t)) \wedge u \notin C$. The former means u is not already guaranteed to have executed; the latter means $encState(t)$ should not output the code for u as some dynamically enclosing call $encState(t'')$ (for t'' $\mathtt{pdom}^+$ t) will have instead. Instead, u is identified by name: such a variable will be in scope because $encState(t'')$ will have visited u first (because of the topological sort), and defined a variable for u (because a variable is defined for every node apart from in case 1, which cannot have applied because u is not the immediate dominator of t's parent). Thus, the result is `(` u >> $doLetExprs(C)$ $code(n)$ `)`.

Non-γ-nodes have a single predecessor which is their immediate dominator, and only possible child. If there are no branches, each node immediately postdominates said predecessor, and case 1 applies with $C \backslash p = \emptyset$; thus $doLetExprs$ produces no code, and we end up with the simple straight-line behaviour above.

However, where there are γ-nodes, transitions t which are control-flow splits are postdominated only by the corresponding γ-node merges, and their successors t' thus have no children. Hence case 3 applies, and $encState$ identifies $\mathsf{idom}(t') = t$ by name—for example, in Figure B.2 node b has no postdominator children, thus resulting in `(a >> OPER 'b')`, whereas node c postdominates $\mathsf{idom}(c) = b$, resulting in `((a >> OPER 'b') >> OPER 'c')`.

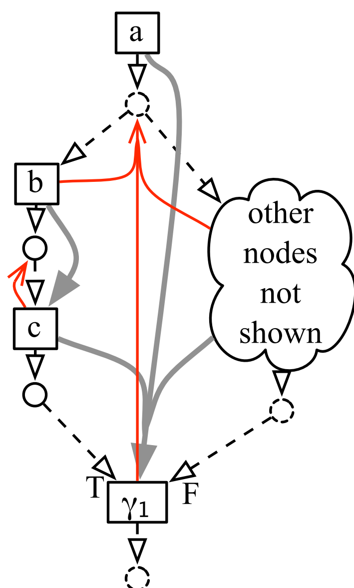


Figure B.2: Simple branching and merging of states—see discussion on page 169

We now consider execution on the example of Figure B.1. Traversal begins at node γ_1, where case 1 applies; the other children are b, c, γ_2 and γ_3, with b and c coming first. Thus `(`$encState(a)$` >> let b=`$encState(b)$` in let c=`$encState(c)$` in let g2=`$encState(\gamma_2)$` in let g3=`$encState(\gamma_3)$` in if P1 then g2 else g3)`.

At node b, case 2 applies, and $C = \emptyset$, resulting in just `OPER 'b'`. Similarly for c.

At node γ_2, case 2 applies again, but there are two children, d and f; thus, we get `(let d=`$encState(d)$` in let f=`$encState(f)$` in if P2 then d else f)`. Similarly for γ_3 with e and g instead of d and f respectively.

At node d, case 3 applies: d's parent is γ_2, and $\mathsf{idom}(\gamma_2) = a \neq \mathsf{idom}(d) = b$. That is, the last operation which definitely has to be executed is b; hence we get `(b >> OPER 'd')` (d has no children). Similarly for node e, and for f and g with $\mathsf{idom}(f) = \mathsf{idom}(g) = c$.

B.3 Adding Values

The key to adding values to the translation is to find an equivalent to the principle of immediate dominance for non-stateful transitions t. Specifically, for each such t, we identify the *earliest* state in which t can be executed, called the *state environment* and written $ste(t)$. Evaluation of t may actually be delayed and occur in some later state (because of lazy evaluation), but any e.g. operands to t which $ste(t)$ produces (perhaps by reading them from memory) can be preserved, as they are (functional) *values*, inside the VSDG. The function $ste(t)$ is defined below in Section B.5; for now, we merely assume it is uniquely defined for each non-stateful node.

The main idea is to change *every variable* t in the state-only Haskell translation of Section B.2 into a tuple of all the values $V_t = \{s \mid ste(^\circ s) = t\}$. That is, whereas previously every variable t was of type `stateful unit`—recall `unit` is the empty tuple!—now each t will have type `stateful (`s_1`*...*`s_{k_t}`)`, where $s_1 \ldots s_{k_t}$ are the types of the places in V_t.

The entire tuple is wrapped in the state monad, so >>= must be used to unwrap the `stateful` tuple and allows access to its elements:

```
x >>= \(n1,n2,n3) -> (* code using node values n1,n2,n3 *)

(x :: stateful (t1*t2*t3)) >>= ((t1*t2*t3) -> stateful T)
                                                          :: stateful T
```

Thus, only operations sequentially composed after t using >>= may access values produced by transitions depending on t's state.

Each stateful transition t is translated as the code for t, followed by >>= to access its results, and then the set V_t translated as in the value-only translation of Chapter 2 (Section 2.2.2); after all $\overrightarrow{s} = V_t$ are defined by `let`-exprs, the body is then `return` $\overrightarrow{s}$. For example, the following is the encoding of a read operation n with a single dependent operation adding one to the value read:

```
let n=READ(addr) >>= \val -> let a=val+1 in return (val,a)
```

For stateful operations which do not produce a value (e.g. writes to memory), $V_t = \emptyset$ (nodes cannot have value dependencies on a node not producing a value!); hence, the type of `t` will be `stateful unit` as in the state-only translation of Section B.2, and the same `let t=`*(code for t)* form as in Section B.2 can be used instead of `let t=`*(code for t)* `>>= \() -> return ()`.

Example Consider the VSDG of Figure B.3. Translation of just the state subgraph yields:

```
(READ) >>
    let e=(WRITE) in
    let d=(READ >> WRITE) in
    if (P) then e else d;
```

Adding values as well yields:

```
let f=6 in (
        (READ >>= \a -> let h=odd(a) in return (a,h))
    >>= \(a,h) ->
        let e=(WRITE(a)) in
        let d=(
                (READ >>=\b -> let c=a+b in return (b,c))
                >>= \(b,c) ->
                WRITE(c) >> return b
        ) in
        if h then (e >> return f)
             else (d >>= \b -> return b)
);
```

This example shows two subtleties.

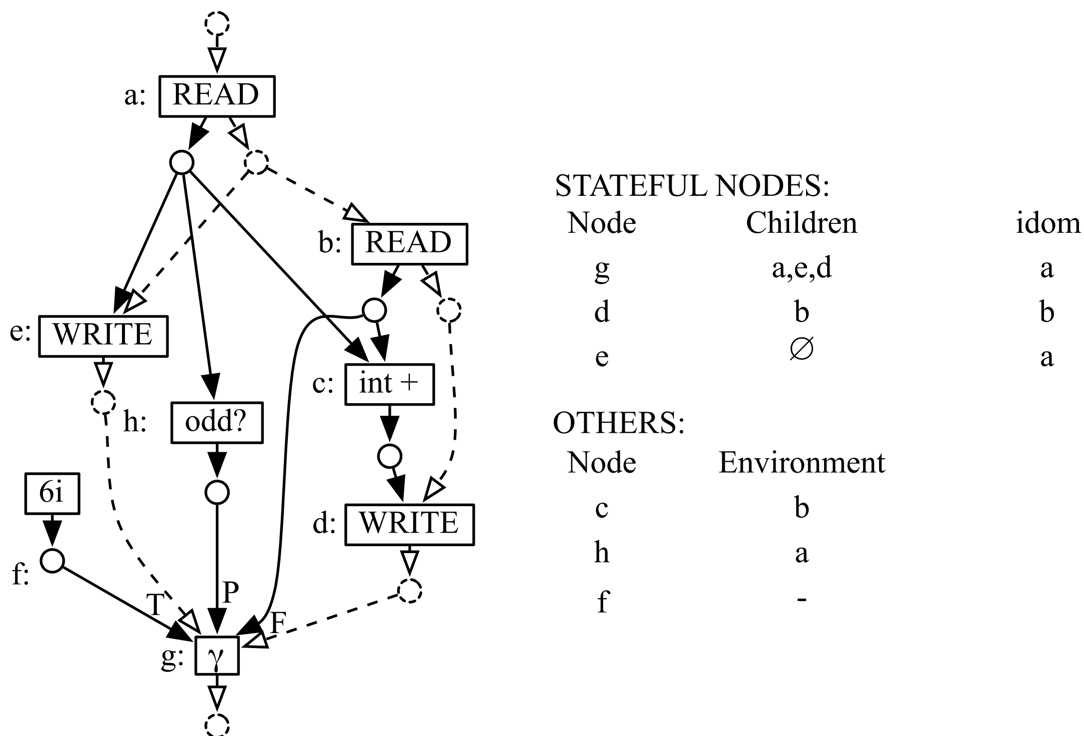


Figure B.3: Example VSDG; see translation into Haskell on page 171

- Firstly, in the treatment of γ-nodes. The code for a γ-node is `if` (p) `then` t `else` f, for predicate p, and t and f being code for the true and false cases. These are most easily produced by pretending the existence of "pseudo-nodes" which pass the values and states for true and false, respectively, through themselves—that is, they gather together into one node all the edges leaving the true or false ports. Thus, the false pseudo-node for g gathers together edges to b and d; its dominator in the state subgraph is d, so it outputs `d >>=` and unpacks the tuple of values returned by d, then returns value b.

- Secondly, the tuple of values `return`ed by a stateful node n is not actually $V_n = \{n' \mid ste(n') = n\}$; these are merely the nodes defined inside the block for n (mostly as `let`-exprs). The `return` statement differs in two respects. Firstly, it need gather together only those nodes that are *actually required* by later code, i.e. which have incoming value edges from outside V_n. Secondly, it must also return any *earlier* values (i.e. unpacked from a previous tuple by >>=, rather than defined by the block for n) that are subsequently required. This second point is demonstrated by the code for `let d=` above, returning the *value* from node b.

B.4 Even the State Monad is Too Expressive

The state monad has been widely criticised for "oversequentializing" operations, that is, forcing even independent operations (for example, reads and writes to memory locations known to be different) into an order. Whilst in our translation this applies only to stateful operations—as value nodes are defined via lazy `let`s—the restriction seems the same as that we deliberately

apply to state edges in the VSDG (that is, in any execution, all stateful operations are totally ordered! We consider more flexible extensions to the VSDG in Section B.6).

However, in fact the state monad is *too* expressive. Consider the following code, which seems similar to that produced by the state-only translation of Section B.2:

```
let a=OPER 'a' in
let b=OPER 'b' in
if (P) then b else a>>b;
```

There is no VSDG which would produce this code! Upon closer inspection we see the reason for this is that the operation `b` is executed in *two* distinct states: the state existing at the beginning of the program fragment, and the state existing after execution of `a`. No single node or variable defines the state upon which `b` operates. That is, `b` is being used *as a function*: it's type is `state -> (unit*state)`, and it is being applied to two distinct operands; this is the same restriction as discussed in Chapter 6 for values (it merely looks different because application of `stateful` operations to states is hidden by >>).

Thus, the sequence of operations must instead be expressed as:

```
(let a=OPER 'a' in
if (P) then return () else a) >> OPER 'b';
```

(we can consider `return ()` as the identity operation on states—it performs no action).

This restriction can also be expressed syntactically: a `stateful` variable defined by a `let`-expr cannot be used on the RHS of >> or >>=.

Such variables *can* be used on the LHS; but the RHS can only be actual code, rather than a variable—the problem being that variables allow to refer to the code from multiple places, whereas the inlined code is written in only one.

B.5 Well-Formedness Conditions

The example in Section B.3 reveals three issues which are the basis of the well-formedness requirements.

- Firstly, the definition of ste, the state environment for each node. We define $ste(t)$, for *any* transition t (the encoding only uses it for non-stateful nodes), as follows:

  Let $V(t)$ be the set of stateful transitions from which n is reachable along paths not going through any stateful transitions. (This requirement is specifically to avoid going *through* γ-nodes; the next step would remove nodes reaching t through non-γ-nodes, but not true/false predecessors of stateful γ-nodes.)

  Let $V'(t)$ be those nodes in $V(t)$ which are not dominators in the state subgraph of other nodes in $V(t)$. (That is, remove from $V(t)$ any node whose execution is necessarily implied by the execution of another.)

  $ste(t)$ is now the unique node in $V'(t)$. The first well-formedness condition is thus that $|V'(t)| = 1$. (For nodes t that produce both state and value(s), $ste(t)$ will be equal to t's unique state predecessor.)

  For γ-nodes, we modify the requirement to apply separately to the true and false places (value(s) and state, considered together, but true separate from false and from predicate—this is the same as the *pseudo-nodes* mentioned in a subtlety in the previous section); it

does *not* apply to the γ-node as a whole. The second requirement then deals with the predicate input, omitted from both true and false:

- Secondly, it must be possible to compute the *predicate* s_p of any γ-node g *before* execution splits. That is, assuming $^{\circ}s_p$ is not stateful, $ste(^{\circ}s_p)$ must dominate g in the state subgraph. However, if $^{\circ}s_p$ is itself stateful, then $^{\circ}s_p$ must itself dominate g in the state subgraph.

B.6 Derestricting State Edges

In this section, we will consider a number of ways in which the VSDG's state edges can be made more flexible.

As mentioned in Section B.4, the VSDGs state edges put a *total ordering* on stateful operations. However, clearly memory operations which do not alias can be reordered with respect to each other[3]. Thus, one possibility would be to allow the state to be "split into chunks", with access to each chunk following similar restrictions on linearity, but operations on different chunks could have state edges which bypassed the other. This would require significant extensions, however, as merging and/or splitting chunks would break basic assumptions on the number of state edges from even ordinary nodes; and "safety" restrictions would depend significantly on alias analysis. Particularly useful however would be to allow chunks of state to be split off somewhat dynamically, allowing modelling of accesses to separate dynamically-allocated arrays, for example.

However, some operations do not seem to obey linearity restrictions at all. Under certain memory semantics, multiple reads to potentially-aliasing locations may be reordered (past each other, but not past writes). This could be modelled by an extension to the VSDG using *quasi-linear types* [Kob99], as in the PacLang language [ESM04].

A third possibility is to model termination of loops and function calls separately from memory modifications. This would allow reads to be reordered past loops or function calls which did not write to the same address—a form of speculation if reads are brought earlier; they could also be speculated more generally.

Lastly, Johnson [JM03] allows state edges to be inserted *arbitrarily* between operations, so long as they do not form cycles; this allows their use as *serializing edges* in node scheduling to express arbitrary ordering decisions, discussed in Chapter 6.

B.7 A Note on Exceptions

We can also consider *generalizing* state edges to model other issues besides memory state and termination. In particular, one possible scheme is to make every operation which might throw an exception (a Potentially-Excepting Operation or PEO) stateful. The state edges then capture the order in which these operations must be executed so that the correct exception behaviour is produced.

We prefer a different scheme, whereby the *tests* as to whether an exception should be thrown are made explicit as γ-nodes (e.g. testing for zero divisors). Exception handlers are then included in the original VSDG. The advantage of this scheme is that it allows the techniques of

[3]Barring subtleties with memory protection—if any memory access might raise a protection fault, reordering even non-aliased operations writes could result in states occuring that should have been impossible.

Chapter 7 to reorder exception tests (e.g. according to the frequency at which they fail); however techniques for performing these tests "for free" using hardware exception support have not yet been presented, and additional issues are raised about *speculating* computations (e.g. divisions) before they are known to be safe. We also argue it is simpler: if exceptions are represented by state edges, some additional mechanism is then required to identify the handler whenever a PEO does raise an exception, including allowing the handler to *resume* execution of the original VSDG.

Bibliography

[AHR02] Matthew Arnold, Michael Hind, and Barbara G. Ryder, *Online feedback-directed optimization of Java*, OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (New York, NY, USA), ACM Press, 2002, pp. 111–129.

[Ana99] C. S. Ananian, *The Static Single Information form*, Tech. Report MIT-LCS-TR-801, Massachusetts Institute of Technology, September 1999.

[App98] Andrew W. Appel, *SSA is functional programming*, ACM SIGPLAN Notices **33** (1998), no. 4, 17–20.

[AU77] Alfred V. Aho and Jeffrey D. Ullman, *Principles of Compiler Design (Addison-Wesley series in computer science and information processing)*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1977.

[BC94] Preston Briggs and Keith D. Cooper, *Effective partial redundancy elimination*, SIGPLAN Not. **29** (1994), no. 6, 159–170.

[BG97] Rastislav Bodik and Rajiv Gupta, *Partial Dead Code Elimination using Slicing Transformations*, SIGPLAN Conference on Programming Language Design and Implementation, 1997, pp. 159–170.

[BGS98a] David A. Berson, Rajiv Gupta, and Mary Lou Soffa, *Integrated Instruction Scheduling and Register Allocation Techniques*, Languages and Compilers for Parallel Computing, 1998, pp. 247–262.

[BGS98b] Rastislav Bodík, Rajiv Gupta, and Mary Lou Soffa, *Complete removal of redundant expressions*, PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation (New York, NY, USA), ACM Press, 1998, pp. 1–14.

[BH92] Thomas J. Ball and Susan Horwitz, *Constructing Control Flow From Control Dependence*, Tech. Report CS-TR-1992-1091, University of Wisconsin-Madison, 1992.

[BH93] Samuel Bates and Susan Horwitz, *Incremental program testing using program dependence graphs*, POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (New York, NY, USA), ACM Press, 1993, pp. 384–396.

[BHRB89] W. Baxter and III H. R. Bauer, *The program dependence graph and vectorization*, POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (New York, NY, USA), ACM Press, 1989, pp. 1–11.

[Bin99] David Binkley, *Computing amorphous program slices using dependence graphs*, SAC '99: Proceedings of the 1999 ACM symposium on Applied computing (New York, NY, USA), ACM Press, 1999, pp. 519–525.

[BR91] David Bernstein and Michael Rodeh, *Global instruction scheduling for superscalar machines*, PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation (New York, NY, USA), ACM Press, 1991, pp. 241–255.

[Bra94] T. Brasier, *FRIGG: A New Approach to Combining Register Assignment and Instruction Scheduling*, Master's thesis, Michigan Technological University, 1994.

[CCK+97] Fred Chow, Sun Chan, Robert Kennedy, Shin-Ming Liu, Raymond Lo, and Peng Tu, *A new algorithm for partial redundancy elimination based on SSA form*, PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation (New York, NY, USA), ACM Press, 1997, pp. 273–286.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck, *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*, ACM Transactions on Programming Languages and Systems **13** (1991), no. 4, 451–490.

[CH84] Frederick Chow and John Hennessy, *Register allocation by priority-based coloring*, SIGPLAN '84: Proceedings of the 1984 SIGPLAN symposium on Compiler construction (New York, NY, USA), ACM Press, 1984, pp. 222–232.

[Cha82] G. J. Chaitin, *Register allocation & spilling via graph coloring*, SIGPLAN '82: Proceedings of the 1982 SIGPLAN symposium on Compiler construction (New York, NY, USA), ACM Press, 1982, pp. 98–105.

[CKZ03] M. Chakravarty, P. Keller, and P. Zadarnowski, *A functional perspective on SSA optimisation algorithms*, Tech. Report 0217, University of New South Wales, 2003.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to Algorithms*, 2nd ed., The MIT Press, 2001.

[CU89] C. Chambers and D. Ungar, *Customization: optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language*, PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation (New York, NY, USA), ACM Press, 1989, pp. 146–160.

[CU90] Craig Chambers and David Ungar, *Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs*, SIGPLAN Conference on Programming Language Design and Implementation, 1990, pp. 150–164.

[CU91] ______, *Making pure object-oriented languages practical*, OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications (New York, NY, USA), ACM Press, 1991, pp. 1–15.

[CUL89] C. Chambers, D. Ungar, and E. Lee, *An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes*, OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications (New York, NY, USA), ACM Press, 1989, pp. 49–70.

[DA99] David Detlefs and Ole Agesen, *Inlining of Virtual Methods*, Lecture Notes in Computer Science **1628** (1999), 258–277.

[DS84] L. Peter Deutsch and Allan M. Schiffman, *Efficient implementation of the Smalltalk-80 system*, Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages (Salt Lake City, Utah), 1984, pp. 297–302.

[DS93] Karl-Heinz Drechsler and Manfred P. Stadel, *A variation of Knoop, Rüthing, and Steffen's Lazy Code Motion*, SIGPLAN Not. **28** (1993), no. 5, 29–38.

[Enn04] Robert J. Ennals, *Adaptive evaluation of non-strict programs*, Ph.D. thesis, University of Cambridge Computer Laboratory, Cambridge, UK, 2004.

[ESM04] Robert Ennals, Richard Sharp, and Alan Mycroft, *Linear Types for Packet Processing.*, ESOP, 2004, pp. 204–218.

[Fis81] Joseph A. Fisher, *Trace Scheduling: A Technique for Global Microcode Compaction.*, IEEE Trans. Computers **30** (1981), no. 7, 478–490.

[FKCX94] Lawrence Feigen, David Klappholz, Robert Casazza, and Xing Xue, *The revival transformation*, POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages (New York, NY, USA), ACM Press, 1994, pp. 421–434.

[FM85] Jeanne Ferrante and Mary Mace, *On linearizing parallel code*, POPL '85: Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (New York, NY, USA), ACM Press, 1985, pp. 179–190.

[FMS88] J. Ferrante, M. Mace, and B. Simons, *Generating sequential code from parallel code*, ICS '88: Proceedings of the 2nd international conference on Supercomputing (New York, NY, USA), ACM Press, 1988, pp. 582–592.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren, *The program dependence graph and its use in optimization*, ACM Trans. Program. Lang. Syst. **9** (1987), no. 3, 319–349.

[Gal86] Zvi Galil, *Efficient algorithms for finding maximum matching in graphs*, ACM Comput. Surv. **18** (1986), no. 1, 23–38.

[GDGC95] David Grove, Jeffrey Dean, Charles Garrett, and Craig Chambers, *Profile-guided receiver class prediction*, OOPSLA '95: Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications (New York, NY, USA), ACM Press, 1995, pp. 108–123.

[Gie95] Jurgen Giesl, *Termination Analysis for Functional Programs using Term Orderings*, Static Analysis Symposium, 1995, pp. 154–171.

[GJ79] Michael R. Garey and David S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman & Co., New York, NY, USA, 1979.

[Hacng] Sebastian Hack, *Register Allocation of Programs in SSA Form*, Ph.D. thesis, UniversitŁt Karlsruhe, Forthcoming.

[Hai98] Max Hailperin, *Cost-optimal code motion*, ACM Transactions on Programming Languages and Systems **20** (1998), no. 6, 1297–1322.

[Har85] D Harel, *A linear algorithm for finding dominators in flow graphs and related problems*, STOC '85: Proceedings of the seventeenth annual ACM symposium on Theory of computing (New York, NY, USA), ACM Press, 1985, pp. 185–194.

[has] *Monad—HaskellWiki*, `http://www.haskell.org/haskellwiki/Monad`.

[HGG05] Sebastian Hack, Daniel Grund, and Gerhard Goos, *Towards Register Allocation for Programs in SSA-form*, Tech. report, UniversitŁt Karlsruhe, September 2005.

[HMC+93] Wen-Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery, *The superblock: an effective technique for VLIW and superscalar compilation*, J. Supercomput. **7** (1993), no. 1-2, 229–248.

[HRB88] Susan Horwitz, Thomas Reps, and David Binkley, *Interprocedural slicing using dependence graphs*, Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation (Atlanta, GA), vol. 23, June 1988, pp. 35–46.

[HU94] Urs Hölzle and David Ungar, *Optimizing dynamically-dispatched calls with run-time type feedback*, PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation (New York, NY, USA), ACM Press, 1994, pp. 326–336.

[JM03] Neil Johnson and Alan Mycroft, *Combined Code Motion and Register Allocation using the Value State Dependence Graph*, Proc. 12th International Conference on Compiler Construction (CC 2003) (Warsaw, Poland) (Görel Hedin, ed.), Lecture

Notes in Computer Science, vol. LNCS 2622, Springer-Verlag, April 2003, pp. 1–16.

[Joh] Neil E. Johnson, Private Communication.

[Joh04] ______, *Code size optimization for embedded processors*, Tech. Report UCAM-CL-TR-607, University of Cambridge, Computer Laboratory, November 2004.

[JPP94] Richard Johnson, David Pearson, and Keshav Pingali, *The Program Structure Tree: Computing Control Regions in Linear Time*, SIGPLAN Conference on Programming Language Design and Implementation, 1994, pp. 171–185.

[JW93] Simon L. Peyton Jones and Philip Wadler, *Imperative functional programming*, POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (New York, NY, USA), ACM Press, 1993, pp. 71–84.

[Kob99] Naoki Kobayashi, *Quasi-Linear Types*, Conference Record of POPL 99: The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas (New York, NY), 1999, pp. 29–42.

[KRS94a] Jens Knoop, Oliver Rüthing, and Bernhard Steffen, *Optimal Code Motion: Theory and Practice*, ACM Transactions on Programming Languages and Systems **16** (1994), no. 4, 1117–1155.

[KRS94b] ______, *Partial Dead Code Elimination*, SIGPLAN Conference on Programming Language Design and Implementation, 1994, pp. 147–158.

[KRS95] ______, *The power of assignment motion*, PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation (New York, NY, USA), ACM Press, 1995, pp. 233–245.

[LT79] Thomas Lengauer and Robert Endre Tarjan, *A fast algorithm for finding dominators in a flowgraph*, ACM Trans. Program. Lang. Syst. **1** (1979), no. 1, 121–141.

[Mog91] Eugenio Moggi, *Notions of Computation and Monads*, Information and Computation **93** (1991), no. 1, 55–92.

[MR79] E. Morel and C. Renvoise, *Global optimization by suppression of partial redundancies*, Commun. ACM **22** (1979), no. 2, 96–103.

[MW95] Frank Mueller and David B. Whalley, *Avoiding conditional branches by code replication*, SIGPLAN Not. **30** (1995), no. 6, 56–66.

[Myc80] Alan Mycroft, *The theory and practice of transforming call-by-need into call-by-value*, Proc. 4th International Symposium on Programming, vol. 83, Springer-Verlag, 1980.

[NP95] Cindy Norris and Lori L. Pollock, *An experimental study of several cooperative register allocation and instruction scheduling strategies*, MICRO 28: Proceedings of the 28th annual international symposium on Microarchitecture (Los Alamitos, CA, USA), IEEE Computer Society Press, 1995, pp. 169–179.

[npc] *A Compendium of NP Optimization Problems*, http://www.nada.kth.se/~viggo/wwwcompendium/.

[OO84] Karl J. Ottenstein and Linda M. Ottenstein, *The program dependence graph in a software development environment*, SDE 1: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments (New York, NY, USA), ACM Press, 1984, pp. 177–184.

[Pet81] J. L. Peterson, *Petri Net Theory and the Modelling of Systems*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[Ram92] S. Ramakrishnan, *Software pipelining in PA-RISC compilers*, Hewlett Packard Journal (1992), 39–45.

[RG81] B. R. Rau and C. D. Glaeser, *Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing*, MICRO 14: Proceedings of the 14th annual workshop on Microprogramming (Piscataway, NJ, USA), IEEE Press, 1981, pp. 183–198.

[RKS00] Oliver Rüthing, Jens Knoop, and Bernhard Steffen, *Sparse code motion*, POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (New York, NY, USA), ACM Press, 2000, pp. 170–183.

[RWZ88] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, *Global value numbers and redundant computations*, Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 1988, pp. 12–27.

[SAF90] B. Simons, D. Alpern, and J. Ferrante, *A foundation for sequentializing parallel code*, SPAA '90: Proceedings of the second annual ACM symposium on Parallel algorithms and architectures (New York, NY, USA), ACM Press, 1990, pp. 350–359.

[Sha99] H. Sharangpani, *Intel Itanium processor microarchitecture overview*, Proceedings of the 1999 Microprocessor Forum, 1999.

[SHK04] Bernhard Scholz, Nigel Horspool, and Jens Knoop, *Optimizing for space and time usage with speculative partial redundancy elimination*, SIGPLAN Not. **39** (2004), no. 7, 221–230.

[SHKN76] T. A. Standish, A. Harriman, D. Kibler, and J. M. Neighbors, *The Irvine Program Transformation Catalogue*, Tech. report, Dept. of Computer and Information Sciences, U.C. Irvine, January 1976.

[Sin05] Jeremy Singer, *Static program analysis based on virtual register renaming*, Ph.D. thesis, University of Cambridge Computer Laboratory, Cambridge, UK, March 2005.

[Ste93] Bjarne Steensgaard, *Sequentializing Program Dependence Graphs for Irreducible Programs*, Tech. Report MSR-TR-93-14, Microsoft Research, Redmond, WA, 1993.

[Ste96] Bernhard Steffen, *Property-Oriented Expansion*, SAS '96: Proceedings of the Third International Symposium on Static Analysis (London, UK), Springer-Verlag, 1996, pp. 22–41.

[SW01] Detlef Sieling and Ingo Wegener, *A Comparison of Free BDDs and Transformed BDDs*, Form. Methods Syst. Des. **19** (2001), no. 3, 223–236.

[Swe92] Philip Hamilton Sweany, *Inter-block code motion without copies*, Ph.D. thesis, Colorado State University, Fort Collins, CO, USA, 1992.

[Tou02] Sid Touati, *Register Pressure in Instruction Level Parallelisme*, Ph.D. thesis, Université de Versailles, France, June 2002.

[Tou05] Sid-Ahmed-Ali Touati, *Register saturation in instruction level parallelism*, Int. J. Parallel Program. **33** (2005), no. 4, 393–449.

[TP95] Peng Tu and David A. Padua, *Efficient Building and Placing of Gating Functions*, SIGPLAN Conference on Programming Language Design and Implementation, 1995, pp. 47–55.

[Upt06] Eben Upton, *Compiling with Data Dependence Graphs*, Ph.D. thesis, University of Cambridge Computer Laboratory, Cambridge, UK, 2006.

[Wad71] Christopher P. Wadsworth, *Semantics and pragmatics of the lambda-calculus*, Ph.D. thesis, Oxford University, 1971.

[Wal00] Larry Wall, *Programming Perl*, O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.

[WCES94] Daniel Weise, Roger F. Crew, Michael D. Ernst, and Bjarne Steensgaard, *Value dependence graphs: Representation without taxation*, Proceedings of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Portland, OR), January 1994, pp. 297–310.

[Wul72] William A. Wulf, *A case against the GOTO*, ACM '72: Proceedings of the ACM annual conference (New York, NY, USA), ACM Press, 1972, pp. 791–797.

[WZ91] Mark N. Wegman and F. Kenneth Zadeck, *Constant propagation with conditional branches*, ACM Trans. Program. Lang. Syst. **13** (1991), no. 2, 181–210.

[ZSE04] Jia Zeng, Cristian Soviani, and Stephen A. Edwards, *Generating fast code from concurrent program dependence graphs*, LCTES '04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems (New York, NY, USA), ACM Press, 2004, pp. 175–181.