Lots of authors

Static Single Assignment Book

Friday 6th May, 2011
11:55

Springer

Contents

Part I Vanilla SSA

1 Introduction — (*J. Singer*) 5
1.1 Definition of SSA
1.2 Informal Semantics of SSA
1.3 Comparison with Classical Data Flow Analysis
1.4 SSA in Context
1.4.1 Historical Context
1.4.2 Current Usage
1.4.3 SSA for High-Level Languages
1.5 About the Rest of this Book
1.5.1 Benefits of SSA
1.5.2 Fallacies about SSA

2 Properties and flavours — (*P. Brisk*) 15
2.1 Preliminaries
2.2 Dominance
2.3 Post-dominance
2.4 Join Sets and Dominance Frontiers
2.5 Iterated Join Sets and Dominance Frontiers
2.6 Split Sets and (Iterated) Post-dominance Frontiers
2.7 Philip's comment
2.8 SSA Form, Def-Use, and Use-Def Chains
2.9 Dead Code Elimination Under SSA
2.10 SSA with dominance property
2.11 Minimal SSA Form
2.12 Semi-pruned SSA Form
2.13 Liveness and Liveness Analysis
2.14 Pruned SSA Form
2.15 Live Range Insersection and Interference
2.16 Philip's comments
2.17 Conventional and Transformed SSA Form

3 Classical construction/update/destruction algorithm — (*J. Singer, F. Rastello*) 27
3.1 Simple programming language model
3.2 Construction
3.3 Turning general SSA into conventional/pruned/dominance property
3.4 Destruction — (*F. Rastello*)

4 Alternative SSA construction algorithms — (*D. Das, U. Ramakrishna, V. Sreedhar*) 33
4.1 Introduction
4.2 Basic Algorithm
4.3 Placing ϕ-functions using DJ graphs
4.3.1 Key Observation
4.3.2 Main Algorithm
4.4 Placing ϕ-functions using Merge Sets
4.4.1 Merge Set and Merge Relation
4.4.2 Iterative Merge Set Computation
4.5 Computing Iterated Dominance Frontier Using Loop Nesting Forests
4.5.1 Loop nesting forest
4.5.2 Main Algorithm
4.6 Summary

5 SSA destruction for machine code — (*F. Rastello*) 49

6 SSA Reconstruction — (*S. Hack*) 51
6.1 Introduction
6.1.1 Live-Range Splitting
6.1.2 Jump Threading
6.2 General Considerations
6.3 Reconstruction based on the Dominance Frontier
6.4 Search-based Reconstruction
6.5 Conclusions

7 Semantics — (*L. Beringer*) 59
7.1 Functional interpretations
7.1.1 Low-level functional program representations
7.1.2 Functional construction of SSA
7.1.3 Program analyses
7.2 Data flow representation
7.3 Pointers to the literature

Part II Extensions

8 Introduction — (*V. Sarkar*) 87
8.1 TODO

9 Array SSA Form — (*Vivek Sarkar, Kathleen Knobe, Stephen Fink*) . . . 89
9.1 Introduction
9.2 Array SSA Form
9.2.1 Full Array SSA Form
9.2.2 Partial Array SSA Form
9.3 Array Lattice
9.4 Sparse Constant Propagation for Scalars and Array Elements
9.5 Beyond Constant Indices
9.6 Extension to Object References: Redundant Load and Dead Store Elimination in Strongly Typed Languages
9.6.1 Analysis Framework
9.6.2 Scalar Replacement Algorithms
9.7 Summary

10 Hashed SSA form: HSSA — (*M. Mantione, F. Chow*) 117
10.1 Introduction
10.2 SSA and aliasing: μ and χ functions
10.3 Introducing "zero versions" to limit the explosion of the number of variables
10.4 SSA and indirect memory operations: indirect variables
10.5 From *indirect variables to* virtual variables
10.6 GVN and indirect memory operations: HSSA
10.7 Building HSSA
10.8 Using HSSA

11 Extended ssa numbering — (*???*) 127
11.1 TODO

12 Static Single Information Form — (*A. Tavares*) 129

13 Psi-SSA Form — (*F. de Ferrière*) 131
13.1 Overview
13.2 Definition
13.3 Construction
13.4 SSA algorithms
13.5 Psi-SSA algorithms
13.6 Out of Psi-SSA
13.6.1 Psi-normalize
13.6.2 Psi-congruence
13.6.3 Phi-congruence
13.6.4 Improvements to the out of Psi-SSA algorithm

14 Graphs and gating functions — (*J. Stanier*) 145
14.1 Introduction
14.2 Data Flow Graphs
14.3 The SSA Graph
14.3.1 Finding induction variables with the SSA Graph
14.4 Program Dependence Graph
14.4.1 Detecting parallelism with the PDG
14.5 Gating functions and GSA
14.5.1 Backwards symbolic analysis with GSA
14.6 Value State Dependence Graph
14.6.1 Definition of the VSDG
14.6.2 Nodes
14.6.3 γ-Nodes
14.6.4 θ-Nodes
14.6.5 State Nodes
14.6.6 Dead node elimination with the VSDG
14.7 Summary

Part III Analysis

15 Introduction — (*A. Ertl*) 165
15.1 benefits of SSA
15.2 properties: suffixed representation can use dense vector, vs DF needs (x,PP) needs hashtable
15.3 phi node construction precomputes DF merge
15.4 dominance properties imply that simple forward analysis works
15.5 Flow insensitive somewhat improved by SSA
15.6 building defuse chains: for each SSA variable can get pointer to next var
15.7 improves SSA DF-propagation

16 Propagating Information using SSA— (*F. Brandner, D. Novillo*) 167
16.1 Overview
16.2 Preliminaries
16.2.1 Property Space
16.2.2 Program Representation
16.2.3 Transfer Functions
16.2.4 Solving Data Flow Problems
16.3 Propagation Using the SSA Form
16.3.1 Program Representation
16.3.2 Sparse Data Flow Propagation
16.3.3 Limitations
16.4 Examples
16.4.1 Copy Propagation
16.4.2 Value Range Propagation

16.5 Beyond Sparse Data Propagation
16.6 Further Reading

17 Liveness — (*B. Boissinot, F. Rastello*) 187
17.1 TODO
17.2 SSA tells nothing about liveness
17.3 Computing liveness sets
17.4 Liveness check

18 Alias analysis — (*???*) 189
18.1 TODO

19 Loop tree — (*S. Pop*) 191
19.1 CFG and Loop Tree can be discovered from the SSA
19.1.1 An SSA representation without the CFG
19.1.2 Discovering natural loop structures on the SSA
19.1.3 Improving the SSA pretty printer for loops
19.2 Analysis of Induction Variables
19.2.1 Stride detection
19.2.2 Translation to chains of recurrences
19.2.3 Instantiation of symbols and region parameters
19.2.4 Number of iterations and computation of the end of loop value
19.3 Conclusion

20 Redundancy Elimination — (*F. Chow*) 201
20.1 Introduction
20.2 Why PRE and SSA are related
20.3 How SSAPRE Works
20.3.1 The Safety Criterion
20.3.2 The Computational Optimality Criterion
20.3.3 The Lifetime Optimality Criterion
20.4 Speculative PRE
20.5 Register Promotion via PRE
20.5.1 Register Promotion as Placement Optimization
20.5.2 Load Placement Optimization
20.5.3 Store Placement Optimization
20.6 Redundancy via the Semantic Approach
20.6.1 Value Numbering
20.6.2 Redundancy Elimination via Value Numbering
20.7 Conclusion

21 Typestate analysis — (*???*) 221
21.1 TODO

Part IV Machine dependent optimizations and codegen

22 Introduction — (*M. A. Ertl*) 227

23 Register Allocation— (*F. Bouchez, S. Hack*) 231
23.1 Introduction
23.1.1 Non-SSA Register Allocators
23.1.2 SSA form to the rescue of register allocation
23.2 Spilling
23.2.1 Spilling under the SSA Form
23.2.2 Finding Spilling Candidates
23.2.3 Maxlive and colorability of graphs
23.3 The problem of spilling
23.3.1 Lowering Maxlive
23.3.2 Spilling is a difficult problem
23.3.3 SSA does not help for spilling
23.3.4 Spilling under SSA
23.4 Coloring and coalescing
23.4.1 Greedy coloring scheme
23.4.2 SSA graphs are colorable with a greedy scheme
23.4.3 A light tree-scan coloring algorithm
23.4.4 Coalescing under SSA form
23.5 Practical and advanced discussions
23.5.1 Handling registers constraints
23.5.2 Out-of-SSA and critical edge splitting
23.5.3 More repairing on ϕ-functions

24 Instruction Selection — (*D. Ebner, A. Krall, B. Scholz*) 251
24.1 Introduction
24.1.1 RTL-Based Instruction Selection
24.1.2 Tree Pattern Matching
24.2 Motivation
24.3 Partitioned Boolean Quadratic Programming
24.3.1 Problem Definition
24.3.2 Heuristic Algorithm
24.3.3 Branch & Bound
24.4 Code Selection for Tree Patterns
24.5 Extensions and Generalizations
24.5.1 Code Selection for DAG Patterns
24.5.2 Chain Rule Placement

25 Automatic Parallelization and Sequencialization — (*S. Pop*) 275

26 If-Conversion — (*C. Bruel*) 277
26.1 Overview/Motivations
26.1.1 Introduction
26.1.2 Architectural requirements
26.2 Classical If-Conversion process

26.3 SSA If-Conversion
26.3.1 SSA operations on basic blocks
26.3.2 SSA representation of conditional instructions
26.3.3 SSA promotion
26.4 Hyperblock formation in SSA
26.4.1 SSA Iterative if-conversion
26.4.2 Tail Duplication
26.4.3 Profitability
26.5 Conclusion

27 Hardware Compilation using SSA — (*Pedro C. Diniz, Philip Brisk*) . . 297
27.1 Brief History and Overview
27.2 Why use SSA for Hardware Compilation?
27.3 Mapping a Control Flow Graph to Hardware
27.3.1 Basic Block Mapping
27.3.2 Basic Control-Flow-Graph Mapping
27.3.3 Control-Flow-Graph Mapping using SSA
27.3.4 ϕ-Function and Multiplexer Optimizations
27.3.5 Implications of using SSA-form in Floor-planing
27.4 Existing SSA-based Hardware Compilation Efforts

28 php experience report — (*P. Biggar*) 313
28.1 Abstract
28.2 Motivation
28.3 SSA form in “traditional” languages
28.3.1 PHP
28.3.2 Whole-program analysis
28.3.3 Analysis results
28.3.4 Object handlers
28.3.5 Building def-use chains
28.3.6 HSSA
28.3.7 Implications

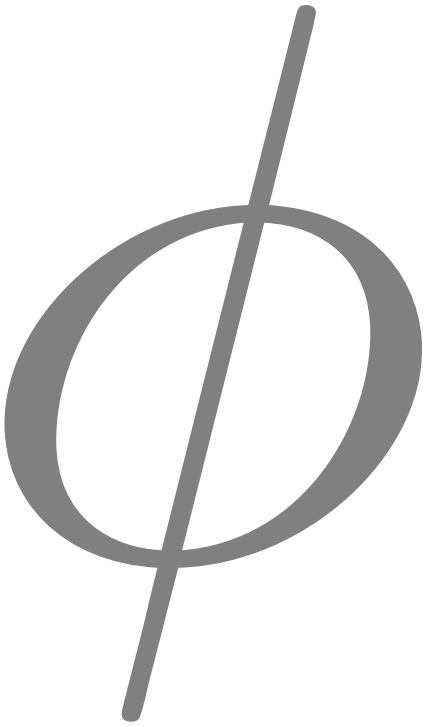
Foreword

Author: Zadeck

Preface

TODO: Roadmap - for students, compiler engineers, etc

TODO: Pre-requisites for reading this book (Tiger book, dragon book, muchnick, etc)?



Part I
Vanilla SSA

Progress: 61%

Progress: 61%

TODO: (tutorial style - not too heavy)

CHAPTER 1

Introduction

J. Singer

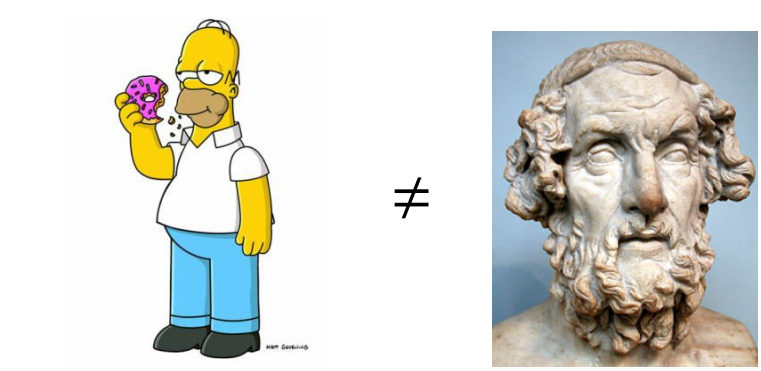
Progress: 80%

Minor polishing in progress

In computer programming, as in real life, names are useful handles for concrete entities. The key message of this book is that having *unique names* for *distinct entities* reduces uncertainty and imprecision.

For example, consider overhearing a conversation about 'Homer.' Without any more contextual clues, you cannot disambiguate between Homer Simpson and Homer the classical Greek poet; or indeed, any other people called Homer that you may know. As soon as the conversation mentions Springfield (rather than Smyrna), you are fairly sure that the Simpsons television series (rather than Greek poetry) is the subject. On the other hand, if everyone had a *unique* name, then there would be no possibility of confusing 20th century American cartoon characters with ancient Greek literary figures.

TODO: Are we allowed to have a cartoon picture such as this one here?



This book is about the *static single assignment form* (SSA), which is a naming convention for storage locations (variables) in low-level representations of computer programs. The term *static* indicates that SSA is particularly relevant for the analysis

of program code that occurs inside a compiler, prior to execution. The term *single* refers to the uniqueness property of variable names that SSA imposes. As illustrated above, this enables a greater degree of precision. The term *assignment* means variable definitions. For instance, in the code

```
x = y+1;
```

the variable x is being assigned the value of expression $(y + 1)$. This is a definition, or assignment statement, for x. A compiler engineer would interpret the above assignment statement to mean that the lvalue of x (i.e., the memory address labelled as x) should be modified to store the value $(y + 1)$.

1.1 Definition of SSA

The seminal paper on SSA [**?**] gives an informal prose definition as follows:

> *A program is defined to be in SSA form if each variable is a target of exactly one assignment statement in the program text.*

This is the simplest, least constrained, definition of SSA. However there are various, more specialized, varieties of SSA, which impose further constraints on programs. Such constraints may relate to graph-theoretic properties of variable definitions and uses, or the encapsulation of specific control-flow or data-flow information. Each distinct SSA variety has specific characteristics. Basic varieties of SSA are discussed in Chapter 2. Part II *FIXME - forward ref* of this book presents more complex extensions.

One important property that holds for all varieties of SSA, including the simplest definition above, is *referential transparency*. Since there is only a single definition for each variable in the program text, a variable's value is *independent of its position* in the program. We may refine our knowledge about a particular variable based on branching conditions, e.g. we know the value of x in the `then` branch following a statement such as:

```
if (x==0)
```

however the *underlying value* of x does not change at this `if` statement. Programs written in pure functional languages are referentially transparent.

Referentially transparent programs are more amenable to formal methods and mathematical reasoning, since the meaning of an expression depends only on the meaning of its subexpressions and not on the order of evaluation or side-effects of other expressions.

For a referentially opaque program, consider the following code fragment.

```
x := 1;
y := x + 1;
x := 2;
z := x + 1;
```

A naive (and incorrect) analysis may assume that the values of y and z are equal, since they have identical definitions of $(x + 1)$. However the value of variable x depends on whether the current code position is before or after the second definition of x, i.e. variable values depend on their *context*.

When a compiler transforms this program fragment to SSA code, it becomes referentially transparent. The translation process involves renaming to eliminate multiple assignment statements for the same variable. Now it is apparent that y and z are equal if and only if $x1$ and $x2$ are equal.

```
x1 := 1;
y  := x1 + 1;
x2 := 2;
z  := x2 + 1;
```

1.2 Informal Semantics of SSA

In the previous section, we saw how straightline sequences of code can be transformed to SSA by simple renaming of variable definitions. The *target* of the definition is the variable being defined, on the left-hand side of the assignment statement. In SSA, each definition target must be a unique variable name. On the other hand, variable names can be used multiple times on the right-hand side of any assignment statements, as *source* variables for definitions. Throughout this book, renaming is generally performed by adding integer subscripts to original variable names. In general this is an unimportant implementation feature, although it can prove useful for compiler debugging purposes.

The *ϕ-function* is the most important SSA concept to grasp. It is a special statement, known as a *pseudo-assignment* function. Appel [**?**] calls it a 'notational fiction.' [1] The purpose of a ϕ-function is to merge values from different incoming paths, at control-flow merge points.

Consider the following example:

```
x := input();
if (x == 42)
    y := 1;
else
    y := x*2;
print(y);
```

There is a distinct definition of y in each branch of the `if` statement. So multiple definitions of y reach the `print` statement at the control-flow merge point. When a compiler transforms this program to SSA, the multiple definitions of y are renamed

[1] Kenneth Zadeck reports that ϕ-functions were originally known as *phoney*-functions, during the development of SSA at IBM Research. Although this was an in-house joke, it did serve as the basis for the eventual name.

as $y1$ and $y2$. However the `print` statement could use either variable, dependent on the outcome of the `if` conditional test. A ϕ-function introduces a new variable $y3$, which takes the value of either $y1$ or $y2$. Thus the SSA version of the program is:

x = input();
if (x == 42)
    $y_1 = 1$;
else
    $y_2 = x \times 2$;
$y_3 = \phi(y_1, y_2)$;
print(y_3);

In terms of their position, ϕ-functions are generally placed at control-flow merge points, i.e. at the heads of basic blocks that have multiple predecessors in control-flow graphs. A ϕ-function at block b has n parameters if there are n incoming control-flow paths to b. The behaviour of the ϕ-function is to select dynamically the value of the parameter associated with the actually executed control-flow path into b. This parameter value is assigned to the fresh variable name, on the left-hand-side of the ϕ-function. Such pseudo-functions are required to maintain the SSA property of unique variable definitions, in the presence of branching control flow.

It is important to note that, if there are multiple ϕ-functions at the head of a basic block, then these are executed simultaneously, *not* sequentially. This distinction becomes important if the target of a ϕ-function is the same as the source of another ϕ-function, perhaps after optimizations such as copy propagation (see Section 16.4.1). When ϕ-functions are eliminated in the SSA destruction phase, they are replaced by copy operations, as described in Section 3.4.

Strictly speaking, ϕ-functions are not directly executable in software, since the dynamic control-flow path leading to the ϕ-function is not explicitly encoded as an input to ϕ-function. This is tolerable, since ϕ-functions are only used during static analysis of the program. They are removed before any program interpretation or execution takes place. There are various executable extensions of ϕ-functions, such as γ-functions (Section 14.5), which take an extra parameter to specify the index of the source parameter whose value should be assigned to the target variable.

We present one further example in this section, to illustrate how the `while` loop control-flow structure appears in SSA. Here is the non-SSA version of the program:

```
x := 0;
y := 0;
while (x < 10) {
  y := y + x;
  x := x + 1;
}
print(y)
```

The SSA code below has two ϕ-functions in the compound loop header statement, that merge incoming definitions from before the loop for first iteration, and from the loop body for subsequent iterations.

```
x1 := 0;
```

```
y1 := 0;
while ({x2 := \phi(x1, x3);
        y2 := \phi(y1, y3);
        x2 < 10
      }) {
  y3 := y2 + x2;
  x3 := x2 + 1;
}
print(y2)
```

It is important to note that the static single assignment property does not prevent multiple assignments to a variable during program execution. For instance, in the SSA code fragment above, variables y_3 and x_3 in the loop body are defined with fresh values at each loop iteration.

1.3 Comparison with Classical Data Flow Analysis

Data flow analysis collects information about programs in order to make optimizing code transformations. During actual program execution, information flows between variables. Static analysis captures this behaviour by propagating *abstract* information, or data flow facts, using an operational representation of the program such as the control flow graph (CFG). This is the approach used in classical data flow analysis.

Often, data flow information can be propagated more efficiently using a *functional*, or *sparse*, representation of the program such as SSA. When a program is translated into SSA form, variables are renamed at definition points. For certain data flow problems (e.g. constant propagation) this is exactly the set of program points where data flow facts may change. Thus it is possible to associate data flow facts directly with variable names, rather than maintaining a vector of data flow facts indexed over all variables, at each program point. For other data flow problems, properties may change at points that are not variable definitions. These problems can be accommodated in a sparse analysis framework by inserting additional pseudo-definition functions at appropriate points to induce additional variable renaming. See Chapter 12 for one such example.

The rest of this section presents a trivial example data flow analysis: **non-zero value analysis**. For each variable in a program, the aim is to determine statically whether that variable can contain a zero integer value at runtime. Figure 1.1 shows the lattice of data flow facts. The analysis will assign an abstract value from this lattice to each variable at each point in the program text.

Figure 1.2(a) shows an example program CFG. The full control flow structure of the program is given, however it only shows definition statements relating to variable x. Figure 1.2(b) shows the SSA version of this example program. Full details of the

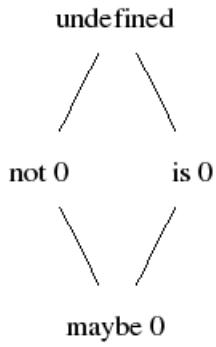


Fig. 1.1 Lattice of data flow facts for non-zero value analysis

SSA construction algorithm are given in Chapter 3. For now, it is sufficient to see that:

1. Integer subscripts have been used to rename variable x from the original program.
2. We have assumed an implicit definition of x_0 at the entry point of the program.
3. A ϕ-function has been inserted at the appropriate control-flow merge point where multiple reaching definitions of x converged in the original program.

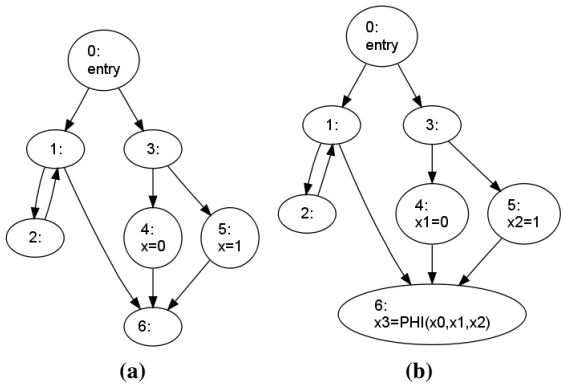


Fig. 1.2 Example control flow graph for non-zero value analysis, only showing relevant definition statements for variable x, in both (a) non-SSA and (b) SSA form.

With classical data flow analysis on the CFG in Figure 1.2(a), we would compute information about variable x for each of the seven nodes in the CFG, using suitable data flow equations. This might give results such as those shown in Table 1.1.

CFG node	abstract value of x
0	undefined
1	undefined
2	undefined
3	undefined
4	is 0
5	not 0
6	maybe 0

Table 1.1 Results of the CFG data flow analysis

Using SSA-based data flow analysis on Figure 1.2(b), we compute information about each variable based on a simple analysis of its definition statement. This gives us four data flow facts, one for each SSA version of variable x, as shown in Table 1.2.

SSA variable	abstract value
x_0	undefined
x_1	is 0
x_2	not 0
x_3	maybe 0

Table 1.2 Results of the SSA data flow analysis

This illustrates some key advantages of the SSA-based analysis.

1. Data flow information *propagates directly* from definition statements to uses, via the def-use links implicit in the SSA naming scheme. In contrast, the classical data flow framework propagates information throughout the program, including points such as node 2 where the information about x does not change, or is not relevant.
2. The results of the SSA data flow analysis are *more succinct*. There are four data flow facts (one for each SSA version of x) versus seven facts for the non-SSA program (one fact about variable x per CFG node).

Part III *FIXME - forward ref* of this textbook gives a comprehensive treatment of SSA-based data flow analysis.

1.4 SSA in Context

1.4.1 Historical Context

Throughout the 1980s, as optimizing compiler techology became more mature, various intermediate representations (IRs) were proposed to encapsulate data dependence in a way that enabled fast and accurate data flow analysis. The motivation behind the design of such IRs was the exposure of direct links between variable definitions and uses, known as *def-use chains*, enabling efficient propagation of data-flow information. Example IRs include the program dependence graph [**?**] and program dependence web [**?**]. Chapter 14 gives further details on dependence graph style IRs.

Static single assignment form was one such IR, which was developed at IBM Research, and announced publicly in several research papers in the late 1980s [**?**, **?**, **?**]. SSA rapidly acquired popularity due to its intuitive nature and straightforward construction algorithm. The SSA property gives a standardized shape for variable def-use chains, which simplifies data flow analysis techniques.

1.4.2 Current Usage

The majority of current commercial and open-source compilers use SSA as a key intermediate representation for program analysis. SSA is generally enabled with the `-O` flag in ahead-of-time compilers (since SSA construction is only required for optimization). Similarly for just-in-time compilers, only the *hot* methods will be recompiled with SSA-based optimizations.

Recent optimizing compiler infrastructures, e.g. LLVM, use SSA from the ground up. Other compilers, e.g. GCC, began development before SSA was well-characterized or widely known. In such cases, SSA support can be back-ported into the original optimization framework. For instance, GCC uses SSA as of version 4.0 [**?**, **?**].

1.4.3 SSA for High-Level Languages

So far, we have presented SSA as a useful feature for compiler-based analysis of low-level programs. It is interesting to note that some high-level languages enforce the SSA property. The SISAL language is defined in such a way that programs automatically have referential transparency, since multiple assignments are not permitted to variables. Other languages allow the SSA property to be applied on a per-variable basis, using special annotations like `final` in Java, or `const` and `readonly` in C#.

The main motivation for allowing the programmer to enforce SSA in an explicit manner in high-level programs is that *immutability simplifies concurrent programming*. Read-only data can be shared freely between multiple threads, without any data dependence problems. This is becoming an increasingly important issue, with the trend of multi- and many-core processors.

High-level functional languages claim referential transparency as one of the cornerstones of their programming paradigm. Thus functional programming supports the SSA property implicitly. Chapter **??** explains the dualities between SSA and functional programming.

1.5 About the Rest of this Book

In this chapter, we have introduced the notion of SSA. The rest of this book presents various aspects of SSA, from the pragmatic perspective of compiler engineers and code analysts. The ultimate goals of this book are:

1. To demonstrate clearly the *benefits* of SSA-based analysis.
2. To dispell the *fallacies* that prevent people from using SSA.

This section gives pointers to later parts of the book that deal with specific topics.

1.5.1 Benefits of SSA

SSA imposes a strict discipline on variable naming in programs, so that each variable has a unique definition. Fresh variable names are introduced at assignment statements, and control-flow merge points. This serves to simplify the structure of variable *def-use* relationships and *live ranges*, which underpin data flow analysis. Parts II and III of this book focus on data flow analysis using SSA.

There are three major advantages that SSA enables:

Program runtime benefit Certain compiler optimizations can be more effective when operating on programs in SSA form. These include the class of *control-flow insensitive analyses*, e.g. [**?**].

Compile time benefit Certain compiler optimizations can be more efficient when operating on SSA programs, since referential transparency means that data flow information can be associated directly with variables, rather than with variables at each program point. We have illustrated this simply with the non-zero value analysis in Section 1.3.

Compiler development benefit Program analyses and transformations can be easier to express in SSA. This means that compiler engineers can be more productive, in writing new compiler passes, and debugging existing passes. For example, the *dead code elimination* pass in GCC 4.x, which relies on an underlying

SSA-based intermediate representation, takes only 40% as many lines of code as the equivalent pass in GCC 3.x, which does not use SSA. The SSA version of the pass is simpler, since it relies on the general-purpose, factored-out, data-flow propagation engine.

1.5.2 Fallacies about SSA

Some people believe that SSA is too cumbersome to be an effective program representation. This book aims to convince the reader that this concern is unnecessary, given the application of suitable techniques. The table below presents some common myths about SSA, and references in this book containing material to dispell these myths.

Myth	*Reference*
SSA greatly increases the overall number of program variables	Chapter 2 FIXME.
Inserted ϕ-functions are difficult to eliminate, since they introduce many copy operations	Chapters 3 and 4 FIXME.
It is difficult to maintain and update the SSA property when optimizing transformations are applied to the code.	Chapter 6 FIXME. Part IV.

CHAPTER 2

Properties and flavours

P. Brisk

Progress: 60%

Structural reordering in progress

2.1 Preliminaries

Any operation of the form $v \leftarrow \ldots$ that assigns a value to variable v is called a *definition* of v. Any operation of the form $\ldots \leftarrow v$ that reads a value of v is called a *use* of v. One of the key properties of SSA Form is that each variable is defined only once; however, renaming each variable alone is insufficient, and ϕ-functions are needed to merge the disparate redefinitions of each variable. The remaining sections of this chapter provide the theoretical foundations that describe precisely how ϕ-functions are inserted, and then derives important properties of the SSA Form that result.

2.2 Dominance

The concepts of *dominance*, introduced in this section, and *post-dominance*, introduced in the following section, form the foundations of a collection of compiler techniques called *control flow analysis*. Intuitively dominance and post-dominance respectively tell us which basic blocks are *guaranteed* to execute before or after others. Historically, this information has been used to identify hierarchies of nested loops. Although this problem may seem trivial, given the prevalence of programming constructs to express loops, such as *for*, *while* and *do-while*, the problem be-

comes significantly more complicated when trying to account for *irreducible loops*, which can be constructed using arbitrary *goto* statements [**?**]. Our interest, however, is SSA Form rather than loop analysis; as we shall see, extensions to these basic definitions, in the sections that follow, are necessary to understand the criteria for instantiating ϕ-functions during the construction of SSA Form.

In a CFG, basic block n_1 *dominates* basic block n_2 if every path in the CFG from the entry point to n_2 includes n_1. By convention, every basic block in a CFG dominates itself. Basic block n_1 *strictly dominates* n_2 if n_1 dominates n_2 and $n_1 \neq n_2$. We use the symbols $n_1 domn_2$ and $n_1 sdomn_2$ to denote dominance and strict dominance respectively.

The *immediate dominator* of a basic block n_2, denoted by $idom(n_2)$, is the basic block n_1 such that $n_1 sdomn_2$, and there does not exist any basic block n_3 such that $n_1 sdomn_3 sdomn_2$. Intuitively, this means that n_1 is the closest basic block to n_2 that strictly dominates n_2. Every basic block other than the entry point has exactly one immediate dominator.

The *dominator tree* is a data structure that links every basic block to its immediate dominator. The entry point of the CFG, itself a basic block, is the root of the dominator tree. If N is the set of basic blocks in the CFG and r represents the entry point, then the dominator tree is the graph $T = (N, E_T)$, where $E_T = \{(n, idom(n)) | n \in N \wedge n \neq r\}$. In other words, each edge in the dominator tree points from each basic block, other than r, to its immediate dominator.

2.3 Post-dominance

Post-dominance is similar in principle to dominance, as we will see below. In fact, all of the post-dominance information described below can be constructed using the basic methods required to compute dominance information. It suffices to reverse every edge in the CFG, and then swap the roles of the entry and exit node. The dominance information computed for the resulting CFG is wholly equivalent to the post-dominance information for the original CFG.

In a CFG, basic block n_2 *post-dominates* basic block n_1 if every path in the CFG from n_1 to the exit point includes n_2. By convention, every basic block in the CFG post-dominates itself. Basic block n_2 *strictly post-dominates* n_1 if n_2 post-dominates n_1 and $n_2 \neq n_1$. We use the symbols $n_2 pdomn_1$ and $n_2 spdomn_1$ to denote post-dominance and strict post-dominance respectively.

The *immediate post-dominator* of a basic block n_1, denoted by $ipdom(n_1)$, is the basic block n_2 such that $n_2 spdomn_1$, and there does not exist any basic block n_3 such that $n_2 spdomn_3 spdomn_1$. Intuitively, this means that n_2 is the closest basic block to n_1 that strictly post-dominates n_1. Every basic block other than the exit point has exactly one immediate post-dominator.

The *post-dominator tree* is a data structure that links every basic block to its immediate post-dominator. The exit point of the CFG is the root of the post-dominator

tree. If N is the set of basic blocks in the CFG, and t is the unique exit point, then the post-dominator tree is the graph $P = (N, E_P)$, where $E_P = \{(n, ipdom(n)), n \in N \wedge n \neq t\}$. In other words, each edge in the post-dominator tree points from each basic block, other than t, to its immediate post-dominator.

2.4 Join Sets and Dominance Frontiers

Let n_1 and n_2 be distinct basic blocks in a CFG. A basic block n_3, which may or may not be distinct from n_1 or n_2, is a *join node* of n_1 and n_2 if there exist at least two non-empty paths, i.e., paths containing at least one CFG edge, from n_1 to n_3 and from n_2 to n_3, respectively, such that n_3 is the only basic block that occurs on both of the paths. In other words, the two paths converge at n_3 and no other CFG node. Given a set S of basic blocks, n_3 is a join node of S if it is the join node of at least two basic blocks in S. The set of join nodes of set S is denoted by the set $\mathcal{J}(S)$.

Intuitively, a join set corresponds to the placement of ϕ-functions. In other words, if n_1 and n_2 are basic blocks that both contain the definition of a variable v, then we ought to instantiate ϕ-functions for v at every basic block in $\mathcal{J}(n_1, n_2)$. Generalizing this statement, if S is the set of basic blocks containing definitions of v, then ϕ-functions should be instantiated in every basic block block in $\mathcal{J}(S)$.

That being said, we still require an efficient method to compute join sets. In principle, we may require join sets for every subset of basic blocks in the CFG, although in principle, we are likely to require fewer. For this reason, we will turn, briefly, to an alternative structure based on dominance, rather than paths in a graph.

The *dominance frontier* of a basic block n_1 in a CFG, denoted $DF(n_1)$, is the set of basic blocks X, such that n_1 *does not* strictly dominate X, but *does* dominate at least one predecessor block of X. The dominance frontier of a set S of basic blocks, denoted $DF(S)$, is the union of the dominance frontiers of the elements of S.

The following section will make a connection between join sets and dominance frontiers.

2.5 Iterated Join Sets and Dominance Frontiers

Let S be a set of basic blocks, and let $\mathcal{J}^0(S) = \mathcal{J}(S)$. Now, let us recursively define $\mathcal{J}^i(S) = \mathcal{J}(S \cup \mathcal{J}^{i\setminus 1}(S))$. By construction, it is straightforward to see that $\mathcal{J}^i(S) \supseteq \mathcal{J}^{i-1}(S)$ for all i, as there is no process to facilitate vertex removal. If we compute these sets iteratively, we will eventually converge to a situation where $\mathcal{J}^i(S) = \mathcal{J}^{i-1}(S)$ in a finite number of steps, as each CFG contains a finite number of basic blocks. The resulting set, denoted by $\mathcal{J}^+(S)$, is called the *iterated join set* of S.

We can do something similar with dominance frontiers. Given a set S of basic blocks, let $DF^0(S) = DF(S)$, and let $DF^i(S) = DF(S \cup DF^{i-1}(S))$. Once again, we can iteratively compute $DF^i(S)$ from $DF^{i-1}(S)$, stopping when the two sets are equal. The resulting set, denoted by $DF^+(S)$ is called the *iterated dominance frontier* of S.

There is, in fact, a connection between iterated join sets and iterated dominance frontiers. Let X be any set of CFG nodes that contains the CFG entry node. Cytron et al. [**?**] proved that $\mathcal{J}^+(X) = DF^+(X)$. Weiss [**?**] proved a slightly stronger version, that $\mathcal{J}(X) = DF^+(X)$. Weiss also conjectured that $\mathcal{J}^+(S) = \mathcal{J}(S)$ for any set of CFG nodes S, and this conjecture was formally proven by Wolfe [**?**].

For reasons that we will discuss in greater detail below, each variable in an SSA Form program has an *implicit* definition at the CFG entry point. Based on the results above, either the join set or iterated dominance frontier could be used, in principle to instantiate ϕ-functions. Cytron et al. [**?**] compute iterated dominance frontiers using a worklist algorithm. As noted by Wolfe [**?**], join and iterated join sets are too complicated to compute directly, and no algorithms exist that call for their explicit use, outside of any context where iterated dominance frontiers suffice.

Intuitively, dominance frontiers are not enough, which is why iterated dominance frontiers are needed. Ignoring the subtle differences between dominance frontiers and join sets, consider a variable v defined in basic blocks n_1 and n_2. Initially, we will instantiate a $\phi-$function for v in some basic block $n_3 \in DF(n_1, n_2)$; however, before renaming, this ϕ-function itself is a new definition of v, so we will need to instantiate additional ϕ-functions in $DF(n_3)$ as well. Taken ad-infinitum, this line of reasoning is precisely why we use iterated dominance frontiers instead of dominance frontiers.

2.6 Split Sets and (Iterated) Post-dominance Frontiers

We have already noted the existence of a duality between the dominance and post-dominance relations. The *dual* of a join set is called a *split set*: given two distinct basic blocks n_2 and n_3, basic block n_1 belongs to their split set, denoted $\S(n_1, n_2)$ if there are non-empty paths from n_3 to n_1 and n_2 respectively, such that the only basic block common to both paths is n_3. Clearly, if all of the CFG edges are reversed, n_3 would belong to the join set of n_1 and n_2. Based on Wolfe's [**?**] proof, there is no need to define an *iterated split set*, as it is equivalent to the split set.

Similarly, the dominance fronter can be generalized to the *post-dominance frontier* by replacing the dominance with the post dominance relation in the definition. For a set of basic blocks, S, we let $PDF(S)$ denote the post-dominance frontier of S. Furthermore, we can define an *iterated post-dominance frontier* in a manner that is exactly similar to the iterated dominance frontier. The iterated post-dominance frontier of a set S of vertices is denoted $PDF^+(S)$.

2.7 Philip's comment

From the outline, I budgeted 2 pages for the text of the preceding sections, and 2 pages for illustrations.

2.8 SSA Form, Def-Use, and Use-Def Chains

A procedure is defined to be in SSA Form if it satisfies two properties. Firstly, each variable can only be defined once. This property is easily achieved: assuming that varible v is defined k times in the procedure, rename each of the definites to $v_1, v_2, \cdots, v_k$ respectively. The second property is that each use of v must now be renamed to correspond to precisely one definition. Without loss of generality, if there is a use of v in a basic block that belongs to the join set of two (or more) definitions v_i and v_j, then the second property is implicitly unsatisfied: renaming this use of v to be a use of v_i or v_j will not maintain correct program semantics. In order to rectify the situation, a ϕ-function must be instantiated to merge the definitions of v_i and v_j into a new variable v_l, and the use of v is replaced with a use of v_l.

Note: insert a figure here to illustrate.

Two important data structures that are used in compilers are *Definition-Use (DU) Chains* and *Use-Definition (UD) Chains*. A definition D of variable v reaches a use U of v if there is a path in the CFG from D to U that does not include another definition of v. A DU Chain connects D to all uses of v that are reachable from D; a UD Chain connects a use U of v to all of the definition of v from which U is reachable.

Under SSA Form, as mentioned above, each variable is defined once and each use corresponds to a single definition. Therefore, there is exactly one DU Chain per variable (the definition points to all of the uses), and one UD Chain per use, which points to the single definition. Thus, the UD chains are explicit, and do not need to be represented.

Note: Insert a figure here to illustrate what happens to DU and UD chains by the conversion to SSA Form.

2.9 Dead Code Elimination Under SSA

Note: In an email on October 19, 2009, Fabrice suggested that I include this section. If I choose to write it and include it here, I will base the discussion on The SSA DCE algorithm from Robert Morgan's textbook, which was used in MachSUIF (among other places). Alternativly, I could use the DCE algorithm from the classic Cytron et al. 1991 paper.

I have not yet written this section, because I am not sure whether an introductory chapter is the appropriate place to insert a discussion of DCE, or whether it is better moved to a chapter on SSA-based optimizations.

2.10 SSA with dominance property

A procedure is defined to be *strict* if every variable is defined before it is used along every path from the entry to exit point [**?**]; otherwise, it is *non-strict*. Some languages, such as Java, impose strictness as part of the language definition; others, such as C/C++, impose no such restrictions.

It is possible to define SSA Form in a manner that ensures strictness, even if the original procedure itself is non-strict; it is also possible to define SSA Form in a way that does not ensure strictness. The designer of an SSA-based compiler must determine which definition is most appropriate.

Any existing SSA construction algorithm can ensure strictness: it suffices to add a pseudo-definition of each variable at the entry point of the procedure. This pseudo-definition does not affect the live range of the variable: it is only used to guide the placement of ϕ-functions. Any SSA construction algorithm that uses iterated dominance frontiers, as we will shortly show, uses this implicit definition.

Let v_0 denote the pseudo-definition corresponding to variable v in a pre-SSA procedure. Any instruction that uses v_0 indicates a likely programmer error. The instruction will use a non-initialized variable; in the general case, this leads to unpredictable program behavior whenever this instruction is executed. Similarly, a ϕ-function may also contain v_0 as a parameter. If the corresponding path is taken into the basic block containing the ϕ-function, then an uninitialized variable (denoted by the pseudo- definition) will be copied to the variable defined by the ϕ-function. As these programs are still legal, depending on the language definition, the best that a compiler can do is to present a warning to the user in order to alert him or her of the situation.

Let S_v be the set of basic blocks containing definitions of variable v, and let n_0 denote the entry point of the CFG. Conceptually, non-strict SSA Form can be constructed by placing ϕ-functions at the entry points of each basic block belonging to $J(S_v) = J^+(S_v)$, i.e., using join sets. Using iterated dominance frontiers, in contrast, ensures strict SSA Form, since $DF^+(S_v) = J(S_v \cup n_0)$.

As there are no known efficient algorithms to compute join sets, the most straightforward way to compute non-strict SSA Form is to first built strict SSA Form and then remove ϕ-functions that are otherwise extraneous. Consider, for example, a ϕ-function that defines variable v_j: one of the parameters is another definition of v, denoted by v_i, while *all* of the remaining parameters are v_0: the implicit definition. It is possible, in this case, to eliminate this ϕ-function by replacing all uses of v_j with uses of v_i instead. Although this destroys the dominance property, it does reduce the number of ϕ-functions in the program.

Note: Insert an example here, as described in an email from Fabrice: 'example of a double diamond with a def and used on the left and b def and used on the right: no interference'. I also have some decent examples from a paper published at IWLS 2007 (no official proceedings).

There are certainly situations where one would want the dominance property as well. Most of these situations exploit properties relating to liveness and interference, which are briefly outlined in the following sections of this chapter, and are described in much greater detail in subsequence chapters.

It is also important to note that many program transformations may cause strict SSA Form to become non-strict. Examples of such transformations include copy propagation and spilling. Techniques to convert non-strict to strict SSA Form without explicitly destroying and rebuilding SSA Form will be discussed in Section ...

(requires coordination with Sebastian). (possibly put an example here showing how copy propagation destroys strictness property; spilling works too).

2.11 Minimal SSA Form

Minimal SSA Form is computed in a straightforward manner, given the preceding discussion. Let S_v be the set of basic blocks containing definitions of variable v. Then ϕ-functions for v are instantiated in every basic block in $DF^+(S_v)$. This is followed by a pass over the program that renames each variable so that it has a unique definition, and renames each use to correspond precisely to one definition. Detailed pseudocode for these procedures can be found in ...

As we will soon see, Minimal SSA Form is hardly minimal, as other algorithms that insert considerably fewer ϕ-functions have been developed; however, Minimal SSA Form does insert fewer ϕ-functions compared to the naive alternative, which is to instantiate a ϕ-function for each variable at the entry point of each basic block in the program that has more than one predecessor.

2.12 Semi-pruned SSA Form

Semi-pruned SSA From, introduced by Briggs et al. [**?**], was based on the observation that many variables are never used outside of the basic blocks where they are defined. Consequently, there is no need to instantiate any ϕ-functions for any of them. Specifically, if every use of variable v occurs after a definition of v in the same basic block, then no ϕ-functions are necessary, although each definition and use must still be renamed. All of these variables are filtered out, and the algorithm to construct Minimal SSA Form is then applied to the remaining variables. For many applications, the vast majority of variables are filtered, which significantly reduces the number of ϕ-functions that are instantiated, and avoids the correspond-

ing increase in the size of the symbol table to accommmodate variables that would otherwise be defined by ϕ-functions.

2.13 Liveness and Liveness Analysis

Now, we turn our attention to variables and their lifetimes. By exploiting this information, we can further reduce the number of ϕ-functions instantiated during the conversion to SSA Form.

Variable v is *live* at some point p in a program if:

1. There is a path from some definition of v to p, and
2. There is a use of v reachable from p, meaning that there is a path from from p to the use of v that does not include a definition of v.

The *live range* of a variable is the set of points at which it is live. A variable must reside in a storage location, in a register, in memory or both, throughout its live range.

Liveness Analysis is the process of computing the live range of each variable in a procedure. The bulk of the work is performed by an iterative data flow analysis that computes either the *Live-In* or the *Live-Out* sets of each basic blocks, i.e., the set of variables that are live at the entry or exit of each basic block in the program. Given this information, a linear traversal of each basic block yields the exact set of variables that are live at each program point. In general, the sets of program points that constitute each variable's live range are not computed.

Here, we will show how to compute the Live-Out set of each basic block. $LIVEOUT[n]$ will denote the Live-Out set of basic block n. Each of the sets is represented as a bit-vector, with one bit corresponding to each variable. If V is the set of variables in the procedure, then each bit-vector contains $|V|$ bits. To simplify notation, we will associate each variable v_i with its index i, so $LIVEOUT[n][i] = 1$ if v_i is live at the exit point of basic block n, and $LIVEOUT[n][i] = 0$ otherwise.

To perform liveness analysis, we need two other sets for each basic block. $UEVAR[n]$ is defined to be the set of *upwards exposed* variables in basic block n, i.e., those variables that are used in n, but do not have a definition preceding them in n. Clearly, the variables belonging to $UEVAR[n]$ are live at the entry point of n; Liveness analysis is a backward dataflow analysis, as variable lifetimes are propagated backwards from $UEVAR[n]$ and into the predecessors of n.

$VARKILL[n]$ contains the set of variables that are defined in basic block n. As liveness analysis is a backward data flow analysis, the backward propagation of liveness information ends with the definition of a variable. The $VARKILL$ sets effectively stop the backward propagation of variables that are live at their definition points.

Initially, the sets $LIVEOUT[n]$ are empty for each basic block, and a linear traversal of each basic block yields the sets $UEVAR[n]$ and $VARKILL[n]$. Once the

initial sets are established, the following data flow equations are repeatedly computed for each basic block n; this is called an *iteration*. The process repeats until an entire iteration fails to change any of the $LIVEOUT$ sets. At this point, stability is reached, and the information contained in the $LIVEOUT$ sets is wholly accurate.

[Insert liveness equations here]

The results of liveness analysis are generally used in three contexts: the construction of Pruned SSA Form, as described in the following section; register allocation; and the translation out of SSA Form.

2.14 Pruned SSA Form

Minimal SSA Form does not take variable liveness information into account when instantiating ϕ-functions. Semi-pruned SSA recognizes that many variables are never live across the boundaries of the basic blocks that contain their definition, and does not instantiate any ϕ-functions for these variables. Under both Minimal and Semi-pruned SSA Form, the ϕ-functions that are instantiated are done so on the basis of iterated dominance frontiers alone, and do not account for any additional liveness information about the corresponding variable. Therefore, ϕ-functions are often instantiated for variables at the entry point of basic blocks where the variable is not actually live. *Pruned SSA Form*, in contrast, requires that ϕ-functions are only instantiated for a variable v at the entry point of basic blocks where v is live [**?**].

There are two possible methods to construct Pruned SSA Form. The direct method is to perform liveness analysis upfront, and check whether a variable is live at the entry point of a basic block before instantiating a ϕ-function for that variable. This approach is straightforward, but requires performing liveness analysis upfront.

Briggs et al. [**?**], who introduced Semi-pruned SSA Form, compared the runtimes of constructing Minimal SSA, Semi-pruned SSA, and the direct method of constructing Pruned SSA Form. Minimal SSA Form instantiated an exorbitant number of ϕ-functions, and, as a consequence, it had the slowest runtime. Semi-pruned SSA Form was the fastest to construct, because Pruned SSA Form required liveness analysis in advance. Although Semi-pruned SSA Form instantiated significantly more ϕ-functions than Pruned SSA Form, the overhead of liveness analysis was significantly greater than the overhead of instantiating the extra ϕ-functions.

It is also possible to construction Pruned SSA by first building Semi-Pruned SSA and then using a simplified dead code elimination algorithm on ϕ-functions to convert the resulting program into Pruned SSA Form; in fact, a more algorithm that eliminates dead code as well as dead ϕ functions also suffices. To the best of our knowledge, such an algorithm has not been formally published; however, Choi et al. [**?**] suggested that such an algorithm may exist, and the method has even been patented [I'm not sure how to cite this]. As long as the runtime of the dead code elimination step plus the cost of instantiating the extra ϕ-functions, which are later removed, is cheaper than performing liveness analysis, this latter method will run faster.

2.15 Live Range Insersection and Interference

Liveness information also plays an important role in register allocation. Any variable must reside in a storage location: a register, in memory, or both, at each point when it is live; otherwise, its value will be lost. The register allocator, therefore, must partition the variables between register and memory at each point in the program, and make sure that at most one variable is allocated to each register at any point.

The live ranges of two variables *intersect* if there is at least one point in a procedure where both variables are live. In general, two variables whose live ranges intersect cannot share the same storage location; however, there is one exception: Chaitin et al. [**?**, **?**] suggested that two variables whose live ranges overlap, but are known to always hold the same value, can share the same register; unfortunately, testing whether or not two computations produce the same result is undecideable, as it is a generalization of the *Halting Problem*. Therefore, Chaitin's more general notion of live range *interference* is often simplified to live range intersection, or intersection with provisions to account for constant values.

Many register allocators build an auxiliary data structure called an *interference graph* to assist with the allocation process. An interference graph is denoted by the tuple $G = (V, E, A)$, where V is a set of vertices, one for each variable in the procedure, E is a set of *interference edges*, and A is a set of *affinity edges*. An interference edge (v_i, v_j) is placed between every pair of variables v_i and v_j that interfere with one another; in principle, any definition of interference that at least includes intersection can be used. An affinity edge (v_k, v_l) is placed between two variables that are repectively defined and used by a copy operation, i.e., $v_k \leftarrow v_l$ or $v_l \leftarrow v_k$; in this case, assigning v_k and v_l to the same register eliminates the copy operation. More details on this process will be provided in section ...

An SSA Form program that is also strict satisfies a very important property: the definition point of each variable dominates each use [**?**]; from there, it is fairly straightforward to show that the definition point not only dominates each use, but the entire live range of the resulting variable. In other words, each live range is a subtree of the dominator tree. The resulting interference graph, therefore, is the intersection graph of a set of subtrees of a tree, which is precisely the definition of the class of *chordal* graphs. Chordal graphs are significant because several problems that are NP-complete for general graphs have efficient polynomial-time solutions for chordal graphs, including graph coloring, which is often used as a model for register allocation. This theory will be presented in much greater detail in future chapters of this book.

2.16 Philip's comments

Detailed example (hopefully, corresponding to the procedure illustrated in the preceding section.

Here, I anticipate 1 page of text and 1 page for the illustration.

2.17 Conventional and Transformed SSA Form

The conversion to SSA form replaces each variable v in the pre-SSA program with a set of variables $v_1, v_2, \ldots, v_k$, which perfectly partition v. At every point in the procedure where v is live, *exactly* one variable v_i is also live; and none of the v_i are live at any point where v is not.

Based on this observation, we can partition the variables in a program that has been converted to SSA Form into congruence classes. We say that x and y are *ϕ-related* to one another if they are referenced by the same ϕ-function, i.e.: either both x and y are parameters of the ϕ-function, or, without loss of generality, x is a parameter, and y is defined by the ϕ-function. This relation is

1. *reflexive*: x is ϕ-related to x;
2. *symmetric*: x is ϕ-related to y if and only if y is ϕ-related to x; and
3. *transitive*: if x is ϕ-related to y and y is ϕ-related to z, then x is ϕ-related to z.

Therefore, this notion of ϕ-relationship is itself and equivalence relation, meaning that the transitive closure of the relation partitions the variables defined locally in the procedure into equivalence classes. By the reflexive property, each variable that is not involved in a ϕ-function belongs to a singleton class. Let ϕ-*CongruenceClassx* denote the equivalence class containing variable x.

The conversion to SSA Form, as described above, ensures that all varibles v_i introduced for pre-SSA variable v belong to the same ϕ- congruence class. The program can be immediately translated out of SSA Form by replacing each definition or use of v_i with v instead, and deleting the ϕ-functions. This simplistic reverse transformation is not always possible after SSA-based program transformations and optimizations such as dead code elimination, copy folding and propagation, and various forms of code motion are applied [**?**].

In particular, transformations applied to an SSA Form program may merge the ϕ-congruence classes of distinct variables u_i and v_j, which respectively correspond to pre-SSA variables u and v respectively. Moreover, u_i and v_j may interfere, making it impossible to replace all of the variables in a single ϕ-congruence class with a single variable.

Note: A figure showing an example here would be very nice.

For example, suppose that the lifetimes of u_i and v_j overlap, and we replace all of the variables in their ϕ-congruence class with a new variable x during the translation

out of SSA Form. Then when the program executes, there will be a point where x is live, and it will contain one value: either the value corresponding to u_i or v_j, but not both. This is incorrect, as both values are required, because they will be used at some point later in the program as per the definition of liveness. As a consequence, the semantics of the program have not been preserved during the translation out of SSA Form.

Sreedhar et al. [**?**] defines a program to be in *Conventional SSA (CSSA)* if it is possible to replace all occurrences of variables in a ϕ-congruence class with a single variable without affecting program correctness; in other words, none of the variables in a ϕ-congruence class interfere with one another under CSSA Form. The basic SSA construction algorithms all build CSSA Form; however, CSSA Form is not preserved after numerous transformations and optimizations are applied. A program is defined to be *Transformed SSA (TSSA)* Form if at least two variables in a ϕ-congruence class interfere. The conversion from TSSA to CSSA Form is an important and required step in order to translate out of SSA Form. Efficient and effective algorithms to translate out of SSA Form will be presented in Sections ...

CHAPTER 3

Classical construction/update/destruction algorithm *J. Singer*

F. Rastello

Progress: 70%

Structural reordering in progress

This chapter describes the standard algorithms for SSA *construction* and *destruction*.

SSA construction refers to the process of translating a non-SSA program into one that satisfies the SSA constraints. In general, this transformation occurs as one of the earliest phases in the middle-end of an optimizing compiler, when the program has been converted to three-address intermediate code. SSA destruction is sometimes called out-of-SSA translation. This phase takes place in an optimizing compiler after all SSA optimizations have been performed, and prior to code generation. (However note that there are specialized code generation techniques that work directly on SSA-based intermediate representations. *FIXME - forward ref*)

The algorithms presented in this chapter are simple and pragmatic. They were originally reported in the seminal SSA research papers [**?**, **?**]. For all of these reasons, they are widely implemented in current compilers. Note that more efficient, albeit more complex, alternative algorithms have been devised. These are described further in Chapters 4 and 5.

3.1 Simple programming language model

Throughout this chapter, we assume that a 'program' is represented as an intraprocedural control flow graph (CFG) with a single entry node r, from which every other node is reachable. *FIXME - precise definition of CFG in appendix?* Data flow occurs

through definitions and uses of global variables, with no pointer indirection. (Part II of this textbook contains chapters dealing with SSA extensions for high-level features like pointer-based aliasing, arrays, compound data structures, concurrency, etc.) We further assume that all global variables have an implicit definition at r.

3.2 Construction

We need the notion of CFG, r the entry node. Need the notion of dominance, and (iterated)-dominance-frontier. Need the notion of reaching definition. Give pointers to relevant appendix definitions.

The original construction algorithm for SSA form [**?**] divides into two distinct phases.

1. **ϕ-function insertion** ensures that any use of a given variable v is reached by exactly one definition of v. In other words, this first phase of ϕ-function insertion performs *live-range splitting*. The resulting live-ranges have the property of having a single definition, which occurs at the beginning of each live range.
2. **variable renaming** assigns a unique variable name to each live-range. This second phase rewrites variable names in program statements such that the program text contains only one definition of each variable, and every use refers to its corresponding unique reaching definition.

As already outlined in Chapter 2, there are different flavors of SSA with distinct properties. In this chapter, we focus on the *minimal* SSA form. Construction of minimal SSA requires the insertion of ϕ-functions at *join nodes*, i.e. nodes in the CFG that can be reached by two or more distinct definitions of the same variable using disjoint paths. Given the assumption that each variable has a *pseudo-definition* at the root r of the CFG, the set of join nodes for a variable definition corresponds exactly to the iterated dominance frontier of that definition. *FIXME - this previous sentence needs to be tidied up.*

We end up with a straightforward algorithm that places ϕ-functions on a per-variable basis. For a given variable v, we place ϕ-functions at the iterated dominance frontier $DF^{+}(S)$ of the set of definition points S for v. If we assume, not unreasonably, that the dominance frontier of each CFG node is precomputed, then the ϕ-functions can be inserted iteratively using a worklist of definition points, and flags (to avoid multiple insertions). This gives the following pseudo-code for the ϕ-function insertion algorithm:

Once ϕ-functions have been inserted using this algorithm, the program may still contain several definitions per variable, however there is a static single definition statement that reaches each use. Moreover, with the following semantics for ϕ-functions where its variable uses are considered to be on the corresponding predecessor basic blocks, then use-def chains are aligned with the CFG dominance tree. In other word the unique definition that reaches each use dominates that use.

Warning: package “algorithmic” is no longer used. Please use algorithm2e instead.

```
[1]
\STATE $W \leftarrow \{ \}$ : set of basic blocks
\STATE $I \leftarrow \{ \}$ : set of basic blocks
\FORALL{$v$ : variable names in original program}
  \FORALL{$d$ : definition statements of variable $v$}
    \STATE \textbf{let} $B$ be the basic block containing $d$
    \STATE $W \leftarrow W \cup \{ B \}$
    \STATE $I \leftarrow I \cup \{ B \}$
  \ENDFOR
  \WHILE{$W \neq \{\}$}
    \STATE remove a basic block $X$ from $W$
    \FORALL{$Y : \mbox{basic block} \in \mathrm{DomFrontier}(X)$}
      \IF{$Y$ does not contain a $\phi$-function for $v$}
        \STATE add $v \leftarrow \phi(...)$ at start of $Y$
        \IF{$Y \notin I$}
          \STATE $W \leftarrow W \cup \{ Y \}$
          \STATE $I \leftarrow I \cup \{ Y \}$
        \ENDIF
      \ENDIF
    \ENDFOR
  \ENDWHILE
\ENDFOR
```

Fig. 3.1 Standard algorithm for inserting ϕ-functions

To obtain a code with a static single assignment per variable, it is necessary to perform variable renaming. Because of the dominance property outlined above, renaming can be easily achieved using a DFS traversal of the dominance tree. During the traversal, for each variable v, it is necessary to remember its nearest dominating definition. This gives the following pseudo-code for the variable renaming algorithm:

```
proc renaming:
// rename variable def and use to have one definition per variable
  foreach v: Variable
    let v.reachingDef:=undef
  foreach B: basicBlock in DFS preorder of the dominance tree
    foreach inst: instruction from top to bottom of B
      foreach o: a use operand of inst
        let v the variable used in o
        v.update_reachingDef(o)
        replace v by v.reachingDef in o
      foreach o: a def operand of inst
        let v the variable used in o
        v.update_reachingDef(o)
        create new variable v'
        replace v by v' in o
        v'.reachingDef:=v.reachingDef
```

```
    foreach phi: phi instruction in a successor of B
      foreach o: a use operand of phi
        let v the variable used in o
        v.update_reachingDef(o)
        replace v by v.reachingDef in o

proc v.update_reachingDef(o):
// simply go up along the reachingDef chain until dominance is fulfilled
  let rd:=v.reachingDef
  while not (rd==undef || def(rd) dominates o)
    rd:=rd.reachingDef
  v.reachingDef:=rd
```

As detailed and explained further *(in Philip's chapter?)*, this simple algorithm constructs programs in a SSA form that is *minimal* (with the meaning given by Cytron et al.). The SSA code could be *pruned* (some dead ϕ-functions are unnecessarily inserted). It is *conventional* SSA (the transformation that renames back all ϕ-related variables into a unique representative one, and removes all ϕ-functions is a correct SSA-destruction algorithm). Finally, such SSA has the *dominance* property (each variable use is dominated by its unique definition).

3.3 Turning general SSA into conventional/pruned/dominance property

Discussion regarding relationship between minimal and pruned SSA, etc. *phi*-function removal can give pruned SSA but break dominance properties, etc. Keep this section simple, and don't duplicate material from Philip Brisk in previous chapter. Ask Philip who may have some materials for the "turning into SSA with dominance property" part.

3.4 Destruction — (*F. Rastello*)

SSA form is a simple and efficient way of having sparse representation of some program information used by some code analysis and optimization. Once we are done with SSA based optimizations, or at least before code generation, one have to get rid of ϕ-functions that are not machine instructions. This phase is called SSA destruction. Destructing a freshly constructed SSA code is easy. One just have to first rename back into a unique representative variable all ϕ-related variables (called SSA-webs) that had initially the same name. Then each ϕ-function have all its operands identical and can be removed to coalesce the related live-ranges. Finding SSA-webs

can be performed efficiently using the classical union-find algorithm for merging sets:

```
proc find_webs
// find the ssa web of each variable
foreach variable v:
  ssaweb(v) := {v}
foreach instruction of the form "a=phi(a1,...,an)"
  foreach ai union(ssaweb(a),ssaweb(ai))
```

An SSA form is said to be conventional if for each ssaweb, all its variables can be safely renamed into a unique representative one. In other words each SSA web should be interference free. While a freshly constructed SSA code is conventional, this may not be the case after optimizations such as copy folding have been performed.

The simplest (but not the more efficient) way for destructing non conventional SSA form is to split all (critical) edge, and then replace ϕ-functions by parallel copies on predecessor blocks. By splitting an edge say (s,d), we mean replacing it by an edge from s to a freshly created basic block and by another edge from this new basic block to d.

Provide a simple but not optimal pseudo code for parallel copy sequentialization. Provide the pseudo-code for ϕ-function replacement.

We should outline that this technique has several drawbacks: first because of specific architectural constraints, region boundaries, or exception handling code, the compiler might not allow to split a given edge; second, the obtained code contains many additional temporary-to-temporary copies. In theory, coping with these copies is the role of the coalescer during the register allocation phase. But only decoupled register allocators (such as the one detailed in Chapter **??**) with memory and time consuming coalescing heuristics can handle the removal of these copies efficiently. Coalescing can also, with less efforts, be handled prior to the register allocation phase. As opposed to a (so called conservative) coalescer during register allocation, this so called aggressive coalescing would not cope with the interference graph colorability. Still the insertion of copies itself might take a substantial amount of time and might not be suitable for dynamic compilation. The goal of Chapter **??** is to cope both with non splittable edges and difficulties related with SSA destruction at machine code level, but also aggressive coalescing in the context of resource constrained compilation.

CHAPTER 4

Alternative SSA construction algorithms

D. Das
U. Ramakrishna
V. Sreedhar

Progress: 80%

Minor polishing in progress

4.1 Introduction

The placement of ϕ-functions is an important step in the construction of Static Single Assignment (SSA) form [**?**]. In SSA form every variable is assigned only once. At control flow merge points ϕ-functions are added to ensure that every use of a variable has exactly one definition. A ϕ-function is of the form $x_n = \phi(x_0, x_1, x_2, \ldots, x_{n-1})$, where x_i's $(i = 0 \ldots n-1)$ are a set of variables with static single assignment. Consider the example control flow graph where we have shown definitions and uses for a variable x. The SSA form for the variable x is illustrated in Figure 4.1(a). Notice ϕ-functions have been introduced at certain join (also called merge) nodes in the control flow graph. In the rest of this chapter we will present three different approaches for placing ϕ-functions at appropriate join nodes. Recall that SSA construction falls into two phases of ϕ-placement and variable renaming. Here we present different approaches for the first phase for minimal SSA form. We first present Sreedhar and Gao's algorithm for placing ϕ-functions. Sreedhar and Gao's algorithm uses DJ graph representation of a CFG. Then we will discuss the notion of merge set and merge relation, and present Das and Ramakrishna's algorithm for placing ϕ-functions based on merge relation and using DJ graph. Finally we describe another approach for placing ϕ-functions based on loop nesting forest proposed by Ramalingam.

4.2 Basic Algorithm

The original algorithm for placing ϕ-functions was based on first computing the dominance frontier (DF) set for the given control flow graph. The dominance frontier $DF(x)$ of a node x is the set of all nodes z such that x dominates a predecessor of z, without strictly dominating z. For example, $DF(8) = \{6, 8\}$ in Figure 4.1(b). A straight forward algorithm for computing DF for each node takes $O(N^2)$ (where N is the number of nodes in the CFG) since the size of the full DF set in the worst case can be $O(N^2)$. Cytron et al.'s algorithm for the placement of ϕ-functions consists of computing iterated dominance frontier (IDF) for a set of all definition points (or nodes where variables are defined). Let N_α be the set of nodes where variable x are defined. Given that the dominance frontier for a set of nodes is just the union of the DF of each node, we can compute $IDF(N_\alpha)$ as a limit of the following recurrence equation (where S is initially N_α)

$$IDF_1(S) = DF(S)$$
$$IDF_{i+1}(S) = DF(S \cup IDF_i(S))$$

A ϕ-function is then placed at each join node in the $IDF(N_\alpha)$ set. Although Cytron et al.'s algorithm works very well in practice, in the worst case the time complexity of ϕ-placement algorithm for a single variable is quadratic in the number of nodes in the original control flow graph.

4.3 Placing ϕ-functions using DJ graphs

Sreedhar and Gao proposed the first linear time algorithm for computing the IDF set without the need for explicitly computing the full DF set. Sreedhar and Gao's original algorithm was implemented using the DJ graph representation of CFG. A DJ graph is a CFG augmented with dominator tree edges. An edge $x \rightarrow y$, of the CFG, which is not a dominator-tree edge, becomes a join(J) edge, referred as $x \xrightarrow{J} y$. The DJ graph for the example CFG is also shown in Figure 4.1(b). The CFG edge $10 \rightarrow 8$, which is not a dominator-tree edge becomes the J edge, $10 \xrightarrow{J} 8$. Rather than explicitly computing the DF set, Sreedhar and Gao's algorithm uses a DJ graph to compute the $IDF(N_\alpha)$ on-the-fly. Even though the time complexity of Sreedhar and Gao's algorithm is linear in the size of the DJ graph, in practice it sometimes performs worse than the Cytron et al. algorithm. The main reason for this is that in practice the size of the DF set is linear in the number of nodes in the CFG and sometimes smaller than the size of the DJ graph.

4.3.1 Key Observation

Now let us try to understand how to compute the DF for a single node using DJ graphs. Consider the DJ graph shown in Figure 4.1(b). To compute $DF(8)$ we simply walk down the dominator (D) tree edges from node 8 and identify all join (J) edges $x \xrightarrow{J} y$ such the $y.level \leq 8.level$, where *level* of a node is the depth of the node from the root of the dominator tree. The *level* of each node is also shown in Figure 4.1(b). For our example the J edges that satisfy this condition are $9 \xrightarrow{J} 6$ and $10 \xrightarrow{J} 8$. Therefore $DF(8) = \{6, 8\}$. To generalize the example, we can compute the DF of a node x using

$$\boxed{DF(x) = \{y | z \in SubTree(x) \ \wedge \ z \xrightarrow{J} y \ \wedge \ y.level \leq x.level\}}$$

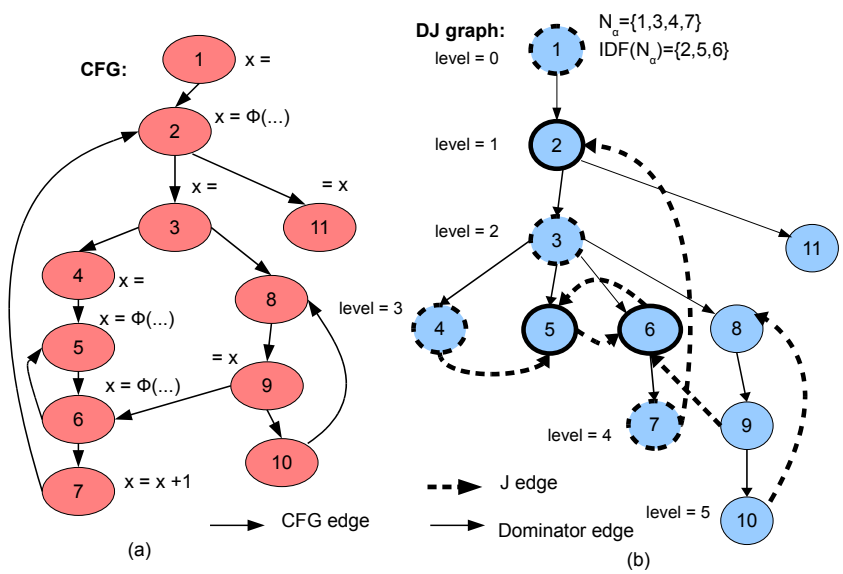


Fig. 4.1 A Motivating Example (a) CFG (b) DJ graph

Now we can extend the above idea to compute the IDF for a set of nodes, and hence the set of ϕ-functions. This algorithm does not precompute DF. Given a set of initial nodes N_α to compute the relevant set of ϕ-functions, a key observation can be made. Let y be an ancestor node of a node x on the dominator tree. If $DF(x)$ has already been computed before the computation of $DF(y)$, $DF(x)$ need not be recomputed when computing $DF(y)$. This is because nodes reachable from $Subtree(x)$ are already in $DF(x)$. However, the reverse may not be true, and therefore the order of the computation of DF is crucial.

To illustrate the key observation consider the example DJ graph in Figure 4.1(b), and let us compute $IDF(\{3,8\})$. Now supposing we start with node 3 and compute $DF(3)$. The resulting DF set is $DF(3) = \{2\}$. Now supposing we next compute the DF set for node 8, and the resulting set is $DF(8) = \{6,8\}$. Notice here that we had already visited node 8 and its subtree as part of visiting node 3. We can avoid such duplicate visits by ordering the computation of DF set so that we first compute $DF(8)$ and then during the computation of $DF(3)$ we avoid visiting the sub-tree of node 8, and use the result $DF(8)$ that was previously computed.

Thus, to compute $IDF(x,y)$ where x is an ancestor of y in the DJ graph one needs to compute:

$$\begin{aligned} IDF(x,y) = \{w | w \in DF(y) \wedge w.level \leq x.level\} \cup \\ \{w | t \in Subtree(x) \setminus Subtree(y) \wedge t \xrightarrow{J} w \wedge w.level \leq x.level\} \end{aligned}$$

Hence, from the equation, $IDF(3,8) = \{w | w \in DF(8) \wedge w.level \leq 3.level\} \cup \{w | t \in Subtree(3) \setminus Subtree(8) \wedge t \xrightarrow{J} w \wedge w.level \leq 3.level\}$, where the first part of the right hand side i.e. $\{w | w \in DF(8) \wedge w.level \leq 3.level\}$ is computed first to avoid duplicated effort. Since neither $6.level$ nor $8.level$ is lesser than $3.level$ none of the nodes in $DF(8)$ appear in $IDF(3,8)$. The nodes that appear in $Subtree(3) \setminus Subtree(8)$ are $4,5,6,7$. From these only $7 \xrightarrow{J} 2$ has a target node 2 such that $2.level \leq 3.level$. Hence $IDF(3,8)$ will include only node 2 without duplicate visits to node 8.

In the next section we present the Sreedhar and Gao's algorithm based on the above key observation.

4.3.2 Main Algorithm

In this section we present Sreedhar and Gao's algorithm. Let $x.level$ be the depth of the node from the root node with $root.level = 0$. To ensure that the nodes are processed according to the first key observation we use a simple array of sets *OrderedBucket*, and two functions defined over the array of sets: (1) InsertNode(n) that inserts the node n in the set $OrderedBucket[n.level]$, and (2) GetNode() that returns a node from the *OrderedBucket* with highest level number. The complete algorithm is shown in Figure 4.2.

First we insert all nodes in N_α in the *OrderedBucket*. The nodes are processed in a bottom-up fashion over the dominator tree from highest node level to least node level (step 5). The procedure Visit(x) essentially walks down the DJ graph and identifies candidate J edges whose destination node are in the IDF set (step 14). Notice that at step 16 a node is inserted in the *OrderedBucket* if it was never inserted before. Finally at step 24 we continue to process the nodes in the sub-tree by visiting over the D edges. When the algorithm terminates the set IDF will contain the IDF set for the initial N_α.

Input: A DJ graph representation of a program and N_α. **Output:** The set $IDF(N_\alpha)$.

```
Procedure IDFMain(Set N_α) {
1:    IDF = {};
2:    foreach ( node x ∈ N_α) do
3:       InsertNode(x);
4:    end for
5:    while ((z = GetNode()) ≠ NULL)
6:       croot = z;
7:       z.visited = true ;
8:       Visit(z);
9:    end while
}
Procedure Visit(x) {
10:   foreach (node y ∈ Succ(x)) do
11:      if (x → y is a J edge) then
12:         if (y.level ≤ croot.level) then
13:            if (y ∉ IDF) then
14:               IDF = IDF ∪ y;
15:               if (y ∉ N_α) then
16:                  InsertNode(y);
17:               endif
18:            end if
19:         end if
20:      else // visit D edges
21:         if (y.visited == false) then
22:            y.visited = true;
23:            // if(y.boundary == false)
24:               Visit(y);
25:            // end if
26:         end if
27:      end if
28:   end for
}
```

Fig. 4.2 Sreedhar and Gao's algorithm for computing IDF set.

In Figure 4.3, some of the phases of the algorithm are depicted for clarity. The *OrderedBucket* is populated with the nodes $1, 3, 4$ and 7 corresponding to $N_\alpha = \{1, 3, 4, 7\}$. The nodes are placed in the buckets corresponding to the levels at which they appear. Hence, node 1 which appears at level 0 is in the zero-th bucket, node 3 is in the second bucket and so on. Since the nodes are processed bottom-up, the first node that is visited is node 7. The successor of node 7 is node 2 and since there exists a J edge $7 \xrightarrow{J} 2$, and the IDF set is empty, the IDF set is updated to hold node 2 according to step 14 of the Visit procedure. In addition, InsertNode(2) is invoked and node 2 is placed in the second bucket. The next node visited is node 4. The successor of node 4 which is node 5 has an incoming J edge $4 \xrightarrow{J} 5$ which results in the new $IDF = \{2, 5\}$. The final IDF set converges to $\{2, 5, 6\}$ when node 5 is visited. Subsequent visits of other nodes do not add anything to the IDF set. An interesting case arises when node 3 is visited. Node 3 finally causes nodes 9 and 10 also to be visited. However, when node 10 is visited, its successor node which

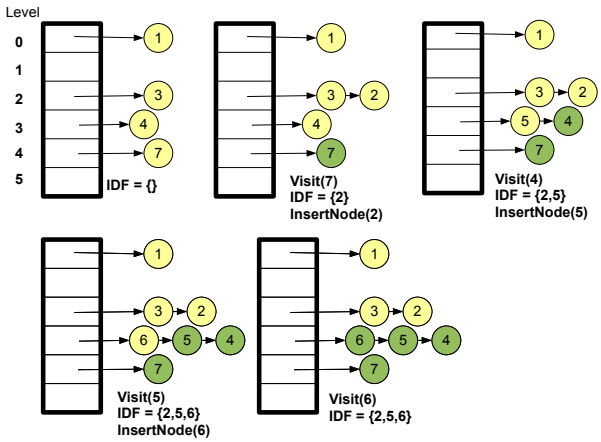


Fig. 4.3 Phases of Sreedhar and Gao's algorithm for $N_\alpha = \{1, 3, 4, 7\}$.

is node 8 and which also corresponds to the J edge $10 \xrightarrow{J} 8$, does not result in an update of the IDF set as the level of node 8 is higher than that of node 3.

Cytron's approach for ϕ-placement involves a fully eager approach of constructing the entire DF graph. On the other hand Sreedhar and Gao's approach is a fully lazy approach as it constructs DF on-the-fly only when a query is encountered. Pingali and Bilardi [**?**] suggested a middle-ground by combining both the approaches. They proposed a new representation called ADT(Augmented Dominator Tree). The ADT representation can be thought as a DJ graph, where the DF sets are pre-computed for certain nodes called "boundary nodes" using an eager approach. For the rest of the nodes, termed "interior nodes", the DF needs to be computed on-the-fly as in the Sreedhar-Gao algorithm. The nodes which act as "boundary nodes" are detected in a separate pass. A factor β is used to determine the partitioning of the nodes of a CFG into boundary or interior nodes by dividing the CFG into zones. β is a number that represents space/query-time tradeoff. $\beta << 1$ denotes a fully eager approach where storage requirement for DF is maximum but query-time is faster while $\beta >> 1$ denotes a fully lazy approach where storage requirement is zero but query is slower.

Given the ADT of a control flow graph, it is straight forward to modify Sreedhar and Gao's algorithm for computing ϕ-functions in linear time. The only modification that is needed is to ensure that we need not visit all the nodes of a sub-tree rooted at a node y when y is a boundary node whose DF set is already known. This change is reflected at step 23 of Figure 4.2, where a subtree rooted at y is visited or not visited based on whether it is a boundary node or not.

4.4 Placing ϕ-functions using Merge Sets

In the previous section we described ϕ-function placement algorithm in terms of iterated dominance frontier. There is another way to approach the problem of ϕ-function placement using the concept of merge relation and merge set. In this section we will first introduce the notion of merge set and merge relation and then show how merge relation is related to DF relation. We will then show how to compute M (merge) graph using DJ graph and then use M graph for placing ϕ-functions.

4.4.1 Merge Set and Merge Relation

Let us define first the notion of a *join* set $J(S)$ for a given set of nodes S in a control flow graph.[1] Consider two nodes u and v and distinct paths from $u \xrightarrow{+} w$ and $v \xrightarrow{+} w$, where w is some node in the CFG. If the two paths meet only at w then w is in the join set of the nodes $\{u, v\}$. For instance, consider nodes 1 and 9 in Figure 4.1(a). The paths $1 \rightarrow 2 \rightarrow 3 \rightarrow 8$ and $9 \rightarrow 10 \rightarrow 8$ meet at 8 for the first time and so $\{8\} \in J(\{1, 9\})$.

Now let us define *merge* relation as a relation $v = M(u)$ that holds between two nodes u and v whenever $v \in J(\{root, u\})$. We insert a ϕ-function at v for a variable that is assigned only at *root* and u. One can show that $J(S) = \cup_{u \in S} M(u)$ where $S \subseteq V$. Also, for any node $u \in V$, $v \in M(u)$ if and only if there is a path $u \xrightarrow{+} v$ that does not contain $idom(v)$. This relationship between dominance and merge can be conveniently encoded using DJ graph and used for placing ϕ-functions. First we will construct DF (Dominance Frontier) graph using DJ graph. For each J edge $u \rightarrow v$ in the DJ graph insert a new $w \xrightarrow{J} v$ where $w = idom(u)$ and w does not strictly dominate v. Another way to look at this is to insert a new J edge $w \rightarrow v$ if $w = idom(u)$ and $w.level \geq v.level$.[2] We repeatedly insert new J edges in a bottom-up fashion over the DJ graph until no more J edges can be inserted. The resulting DJ graph is a "fully cached" DJ graph. We will call the fully cached DJ graph as the DF graph as the $DF(x) = \{t | x \xrightarrow{J} t\}$.

Next we can compute the M relation from the DF graph. Recall that we insert an edge from a node u to a node v if $v \in M(u)$. Using DF graph we compute the M graph as transitive closure using only the DF edges of the DJ graph. Now given the M graph of a CFG we can place ϕ-functions to a set $S \subseteq V$ at the neighbouring nodes of the nodes in S. In Figure 4.4, $M(4) = \{2, 5, 6\}$ as the nodes 2,5 and 6 are the only nodes reachable from node 4 following the DF edges. Similarly, for node 8, $M(8) = \{2, 5, 6, 8\}$ due to these nodes being reachable from node 8 via the DF edges.

[1] In English 'join' and 'merge' are synonyms, but unfortunately in the literature, due to lack of better terms, these two synonyms are used to mean distinct but related concepts.

[2] Recall that *w.level* is the depth of *w* from *root* of the dominator tree.

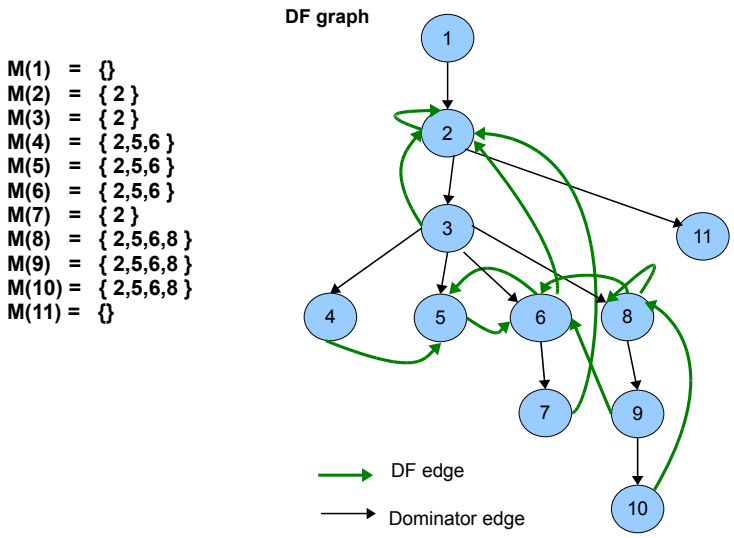


Fig. 4.4 DF graph and M sets

The algorithms [] for placing ϕ-functions based on the construction of DF graph or M relation in the worst case can be quadratic. The merge relation is due to Pingali and Bilardi []. The DF graph and the M sets for the running example are given in Figure 4.4.

4.4.2 Iterative Merge Set Computation

In this section we describe a method to iteratively compute the *merge* relation using a dataflow formulation. In the previous section we saw that the *merge* relation can be computed using a transitive closure of the DF graph, which in turn can be computed from the DJ graph. In the algorithm proposed by Das and Ramakrishna [**?**, **?**,] explicit DF graph construction or the transitive closure formation are not necessary. Instead, the same result can be achieved by formulating the *merge* relation as a dataflow problem and solving it iteratively. For several applications, this approach has been found to be a fast and effective method to construct $M(x)$ for each node x and the corresponding ϕ-function placement using the Mx) sets.

Consider a J edge $u \xrightarrow{J} v$. Also consider all nodes w s.t. w dominates u and $w.level \geq v.level$. For every w we can set up the dataflow equation as $M(w) = M(w) \cup M(v) \cup \{v\}$. The set of dataflow equations for each node n in the DJ graph can be solved iteratively using a top-down pass over the DJ graph. To check whether

multiple-passes are required over the DJ graph before a fixed-point is reached for the dataflow equations, we devise an "inconsistency condition" stated as follows.

Inconsistency Condition:

For a J edge, $u \xrightarrow{J} v$, if u does not satisfy $M(u) \supseteq M(v)$, then the node u is said to be inconsistent.

The algorithm described in the next section is directly based on the method of building up the $M(x)$ sets of the nodes as each J edge is encountered in an iterative fashion by traversing the DJ graph top-down. If no node is found to be **inconsistent** after a single top-down pass, all the nodes are supposed to have reached fixed-point solutions. If some node is found to be inconsistent, multiple passes are required till fixed-point solutions can be reached.

4.4.2.1 TopDownMergeSetComputation

The first and direct variant of the approach laid out above is termed TDMSC-I. This variant works by scanning the DJ graph in a top-down fashion as shown in step 7 of Figure 4.5. All $M(x)$ sets are set to the null set before the initial pass of TDMSC-I. The $M(x)$ sets computed in a previous pass are carried over if a subsequent pass is required.

The DJ graph is visited level-by-level. During this process, for each node n encountered, if there is an **incoming** J edge $s \xrightarrow{J} n$ to n as in step 9, then a separate bottom-up pass starts at step 15. This bottom-up pass traverses all nodes w s.t. w dominates s and $w.level \geq s.level$, updating the $M(w)$ values using the aforementioned dataflow equation. Step 20 is used for inconsistency check. *RequireAnotherPass* is set to true only if a fixed point is not reached and the inconsistency check succeeds for some node.

There are some subtleties in the algorithm that should be noted. Step 20 of the algorithm visits incoming edges to *lnode* only as *lnode* is at the same level as n, which is the current level of inspection and the incoming edges to *lnode*'s posterity are at a level greater than that of node n and are unvisited yet.

Here, we will briefly walk through TDMSC-I using the DJ graph of Figure 4.1(b). Moving top-down over the graph, the first J edge encountered is $7 \xrightarrow{J} 2$. As a result, a bottom-up climbing of the nodes happen, starting at node 7 and ending at node 2 and the merge sets of these nodes are updated so that $M(7) = M(6) = M(3) = M(2) = \{2\}$. The next J edge to be visited can be any of $4 \xrightarrow{J} 5$, $5 \xrightarrow{J} 6$ or $6 \xrightarrow{J} 5$ at $level = 3$. Assume it is $5 \xrightarrow{J} 6$. This results in $M(5) = M(5) \cup M(6) \cup \{6\} = \{2, 6\}$. Now, let $6 \xrightarrow{J} 5$ be visited. Hence, $M(6) = M(6) \cup M(5) \cup \{5\} = \{2, 5, 6\}$. At this point, the **inconsistency check** comes into picture for the edge $6 \xrightarrow{J} 5$ as $5 \xrightarrow{J} 6$ is another J edge that is already visited and is an incoming edge of node 6. Checking for $M(5) \supseteq M(6)$ fails, implying that the $M(5)$ needs to be computed again. This is done in a separate pass as suggested by the *RequireAnotherPass* value of true. In a

Input: A DJ graph representation of a program. **Output:** The merge sets for the nodes.

```
Procedure TDMSCMain {
    ∀node x ∈ DJ graph set M(x) = {}
    done = false;
    while ( ! done ) do
        done = TDMSC-I(DJ graph);
    end while
}
Procedure TDMSC-I(DJ graph) {
    RequireAnotherPass = false;
    while (in B(readth) F(irst) S(earch) order of DJ graph) do
        n = Next Node in BFS list
        for (all incoming edges to n) do
            Let e = s -J-> n , be an incoming J edge
            if (e not marked visited) then
                Mark e as visited
                tmp = s;
                lnode = NULL;
                while (level(tmp) ≥ level(n)) do
                    M(tmp) = M(tmp) ∪ M(n) ∪ {n};
                    lnode = tmp;
                    tmp = parent(tmp); //dominator tree parent
                end while
                for (all incoming edges to lnode) do
                    Let e' = s' -J-> lnode, be an incoming J edge
                    if (e' visited) then
                        if (M(s') ⊉ M(lnode)) then //Check inconsistency
                            RequireAnotherPass = true;
                        end if
                    end if
                end for
            end if
        end for
    end while
    return RequireAnotherPass;
}
```

Fig. 4.5 Top Down Merge Set Computation algorithm for computing Merge sets.

second iterative pass, the J edges are visited in the same order. Now, when $5 \xrightarrow{J} 6$ is visited, $M(5) = M(5) \cup M(6) \cup \{6\} = \{2, 5, 6\}$ as the current $M(5) = \{2, 6\}$ and $M(6) = \{2, 5, 6\}$. On a subsequent visit of $6 \xrightarrow{J} 5$, $M(6)$ is also set to $\{2, 5, 6\}$. The inconsistency does not appear any more and the algorithm proceeds to handle the edges $4 \xrightarrow{J} 5$, $9 \xrightarrow{J} 6$ and $10 \xrightarrow{J} 8$ which have also been visited in the earlier pass. TDMSC-I is invoked repeatedly by a different function which calls it in a loop till *RequireAnotherPass* is returned as $false$ as shown in the procedure TDMSCMain.

4.4.2.2 TDMSC-II

TDMSC-II is an improvement to algorithm TDMSC-I. This improvement is fueled by the observation that for an inconsistent node u, the merge sets of all nodes w s.t. w dominates u and $w.level \geq u.level$, can be locally corrected for some special cases. This heuristic works very well for certain class of problems – especially for CFGs with DF graphs having cycles consisting of a few edges. This eliminates extra passes as an inconsistent node is made consistent immediately on being detected. For details of this algorithm refer to [].

4.4.2.3 Final ϕ-function placement using Merge Sets

Once the $Merge$ relation is computed for the entire CFG, placing ϕ is a straightforward application of the $M(x)$ values for a given N_α, as shown in Figure 4.6.

Input: A CFG with Merge sets computed and N_α. **Output:** Blocks augmented by ϕ-functions for the given N_α.

```
Procedure φ-placement for N_α {
1:    for (∀n ∈ N_α) do
2:        for (∀n' ∈ M(n)) do
3:            Add a φ for n in n' if not placed already
4:        end for
5:    end for
}
```

Fig. 4.6 ϕ-placement using $Merge$ sets

4.5 Computing Iterated Dominance Frontier Using Loop Nesting Forests

This section illustrates the use of **loop nesting forests** for construction of the iterated dominance frontier (IDF) of a set of vertices in a CFG, which may contain both reducible as well as irreducible loops.

4.5.1 Loop nesting forest

Loop nesting forest is a data structure that respresents the loops in a CFG and the containment relation between them. Loop nesting forest has been defined formally in [tobequoted]. However, while there is a fairly accepted notion of loop nesting for-

est in a reducible graph [**?**], there is less agreement about their definition in arbitrary graphs. Steensgard [**?**], Sreedhar [**?**] and Havlak [**?**] all provide different definitions which have merits under different situations. For the example shown in Figure 4.7(a) the loops with backedges $11 \rightarrow 9$ and $12 \rightarrow 2$ are both reducible loops. The corresponding loop nesting forest shown in Figure 4.7(b), consists of two loops with their header nodes being 2 and 9. The loop with header node 2 contains the loop with header node 9.

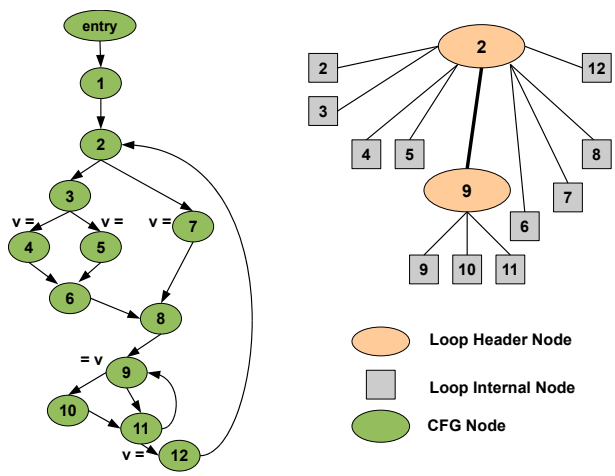


Fig. 4.7 An example (a) CFG and (b) IDF Computation Using Loop Nesting Forest

4.5.2 Main Algorithm

Let us say that a definition node d "reaches" another node u if there is a path in the graph from d to u which does not contain any definition node (for variable v) other than d. If at least two definitions reach a node u, then u belongs to $IDF(X)$ where X consists of these definition nodes. This suggests the algorithm in Figure 4.8. According to the algorithm, for a given N_α, we can compute $IDF(N_\alpha)$ as follows:

- Initialize IDF to empty set
- Using topological order compute the subset of $IDF(N_\alpha)$ that can reach a node using forward dataflow
- Add a node to IDF if it is reachable from multiple nodes

For Figure 4.7, the acyclic version of the graph G termed G_{ac} is formed by dropping the backedges $11 \rightarrow 9$ and $12 \rightarrow 2$. Also, $entry(G) = \{entry\}$ where $entry$ is a specially designated node that is root of the CFG. For the definitions of v in nodes $4, 5, 7$ and 12 in Figure 4.7, the subsequent nodes where multiple definitions reach

turn out to be at 6 and 8. For node 6 any one of two definitions in nodes 4 or 5 can reach. For node 8, it can be either the definition from node 7 or one of 4 or 6. Note that the backedges do not exist in the acyclic graph and hence node 2 is not part of the IDF set. We will see later how the IDF set for the entire graph is computed by factoring in the contribution of the backedges.

Input: An acyclic CFG G_{ac}, X: subset of $V(G)$. **Output:** The set $IDF(X)$.
Procedure ComputeIteratedDominanceFrontier(G_{ac}:Acyclic graph,X: subset of $V(G)$) {
1: $IDF = \{\}$;
2: $\forall$ *node* of $V(G)$, $UniqueReachingDefs(node) = \{\ \}$;
3: **for** $\forall u$ of $V(G) - entry(G)$ in topological sort order **do** {
4: $ReachingDefs = \{\}$;
5: **for** $\forall$ predecessor v of u **do** {
6: **if** $v \in (\ IDF \cup X \cup entry(G)\ )$ **then**
7: $ReachingDefs = ReachingDefs \cup \{v\}$;
8: **else**
9: $ReachingDefs = ReachingDefs \cup UniqueReachingDefs(v)$;
10: **end if**
11: **end for**
12: **if** ($\|ReachingDefs\| == 1$) **then**
13: $UniqueReachingDefs(u)$ = only element of $ReachingDefs$;
14: **else**
15: $IDF = IDF \cup \{u\}$;
16: **end if**
17: **end for**
18: **return** IDF;
}

Fig. 4.8 Ramalingam's algorithm for computing the IDF of an acyclic graph.

We will walk through some of the steps of this algorithm for $IDF(\{4, 5, 7, 12\})$. We will start at vertex 4. For this vertex, $predecessors = \{3\}$. But as $3 \notin IDF \cup X \cup \{entry\}$, $ReachingDefs = UniqueReachingDefs(3)$ according to step 9. Hence, $ReachingDefs = \{entry\}$. As $\|ReachingDefs\|$ equals 1, $UniqueReachingDefs(4) = \{entry\}$. The same logic applies for vertices 5 and 7, and so, $UniqueReachingDefs(5) = UniqueReachingDefs(7) = \{entry\}$. For vertex 6, $predecessors = \{4, 5\}$. Since both vertices 4 and 5 belong to X, according to step 7, $ReachingDefs = \{4, 5\}$. As $\|ReachingDefs\| \neq 1$, according to step 15, $IDF = \{6\}$. Following similar arguments, $IDF = \{6, 8\}$, when node 8 is visited. When the rest of the vertices are visited, the IDF set remains unchanged.

How can the algorithm to find IDF for acyclic graphs be extended to handle reducible graphs? A reducible graph can be decomposed into an acyclic graph and a set of backedges. The contribution of backedges to the iterated dominance frontier can be identified by using the loop nesting forest. If a vertex u is contained in a loop then $IDF(u)$ will contain the loop header. For any vertex u, let $HLC(u)$ denote the set of loop headers of the loops containing u. Given a set of vertices X, it turns out that $IDF(X) = HLC(X) \cup IDF_{ac}(X \cup HLC(X))$ where $IDF(X)$ and IDF_{ac} denote the IDF over the original graph G and the acyclic graph G_{ac} respectively. Revert-

ing back to Figure 4.7 we see that in order to find the IDF for the nodes where the variable v is defined, we need to evaluate $IDF(\{4,5,7,12\})$. Firstly, we would need to evaluate $IDF_{ac}(\{4,5,7,12\} \cup HLC(\{4,5,7,12\}))$. $HLC(\{4,5,7,12\}) = \{2\}$ as all these nodes are contained in a single loop with header 2. Hence, we would need to find $IDF_{ac}(\{2,4,5,7,12\})$ which turns out to be the set $\{6,8\}$. Finally, $IDF(\{4,5,7,12\}) = HLC(\{4,5,7,12\}) \cup \{6,8\} = \{2,6,8\}$.

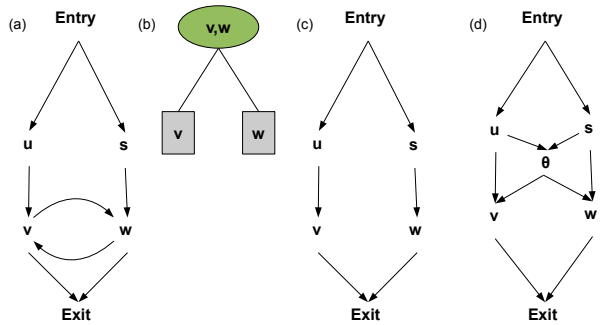


Fig. 4.9 (a) An irreducible graph (b) The Loop Nesting Forest (c) The acyclic subgraph (c) Transformed graph

Now that we have shown how IDF can be computed for graphs with reducible loops using loop nesting forests, we will briefly touch upon how graphs containing irreducible loops can be handled. The insight behind the implementation is to transform the irreducible loop in such a way that an acyclic graph is created from the loop without changing the dominance properties of the nodes. Referring to Figure 4.9, we see that the irreducible loop comprising of nodes v and w in (a) can be transformed to the acyclic graph in (c) by removing the edges between nodes v and w that create the irreducible loop. We can now create a new dummy node θ and add edges from the predecessor of all the header nodes of the cycles of the irreducible graph to θ, as well as add edges from θ to all the header nodes. This results in additional edges from u and s which are the predecessors of the header nodes v and w respectively to θ. In addition, extra edges are added from θ to the header nodes v and w.The transformed graph is shown in (d). Following this transformation, computing IDF for the nodes in the original irreducible graph translates to computing IDF for the transformed graph. Since this graph is acyclic, the algorithm depicted in Figure 4.8 can be applied by noting the loop nesting forest structure as shown in (b). The crucial observation that allows this transformation to create an equivalent acyclic graph is the fact that the dominator tree of the transformed graph remains identical to the original graph containing an irreducible cycle. One of the drawbacks

of the transformation is the possible explosion in the number of dummy nodes and edges for graphs with many irreducible cycles as a unique dummy node needs to be created for each irreducible loop. However, such cases may be rare for practical applications.

4.6 Summary

This chapter provides an overview of some advanced construction algorithms for SSA. These include the first proposed linear-time algorithm by Sreedhar using the concept of DJ graphs, the merge relation by Pingali and Bilardi, a DJ graph and merge set based algorithm by Das and Ramakrishna and finally an algorithm by Ramalingam based on the concept of loop nesting forests. Though all these algorithms claim to be better than the original CFR algorithm, they are difficult to compare due to the unavailability of these algorithms in a common compiler framework. However, some of these algorithms are known to perform better than CFR for several well-known benchmarks.

CHAPTER 5

SSA destruction for machine code

F. Rastello

Progress: 20%

Material gathering in progress

Describes the context: cannot necessarily split critical edges, some variables have renaming constraints, some instructions have renaming constraints that we want to handle during SSA destruction (some may be handled during register allocation).

Renaming constraints are handled using copies around constrained instructions. Go to conventional-SSA using Sreedhar's I technique. Outline the limitation of this technique when dealing with condional jump instruction that define a variable itself; when dealing with renaming constraints for variables that interfere...

Define our "ultimate" notion of interference using value. Build the interference graph (provide the simplified pseudo-code). Do coalescing. Refer to CGO'09 paper for issues concerning JIT compilation. Remove ϕ-functions and perform renaming. Provide a more sophisticated pseudo-code (than for 3.4) for parallel copies sequentialisation (that can be performed before or after coalescing and ϕ removal.

CHAPTER 6

SSA Reconstruction

S. Hack

Progress: 60%

Structural reordering in progress

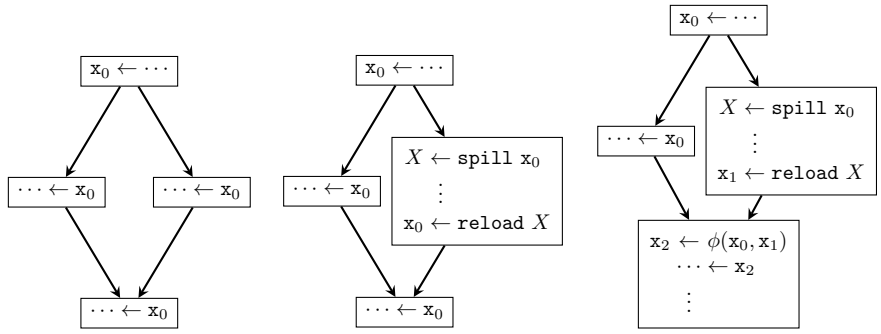
6.1 Introduction

Some optimizations break the single-assignment property of the SSA form by inserting additional definitions for a single SSA value. A common example is live-range splitting by inserting copy operations or inserting spill and reload code during register allocation. Other optimizations, such as loop unrolling or jump threading might duplicate code, thus adding additional variable definitions, *and* modify the control flow of the program. Let us first go through two examples before we present algorithms to properly repair SSA.

6.1.1 Live-Range Splitting

In Figure 6.1b, our spilling pass decided to spill a part of the live range of the variable x_0 in the right block. Therefore, it inserted a store and a load instruction. This is indicated by assigning to the memory location **X**. The load however is a second definition of x_0, hence SSA is violated and has to be reconstructed as shown in Figure 6.1c. Furthermore, This figure shows that maintaining SSA also involves placing ϕ-functions.

Such program modifications are done by many optimizations. Not surprisingly, maintaining SSA is often one of the more complicated and error-prone parts in such



(a) Original program **(b)** Spilling x_0, SSA violated **(c)** SSA reconstructed

Fig. 6.1 Adding a second definition

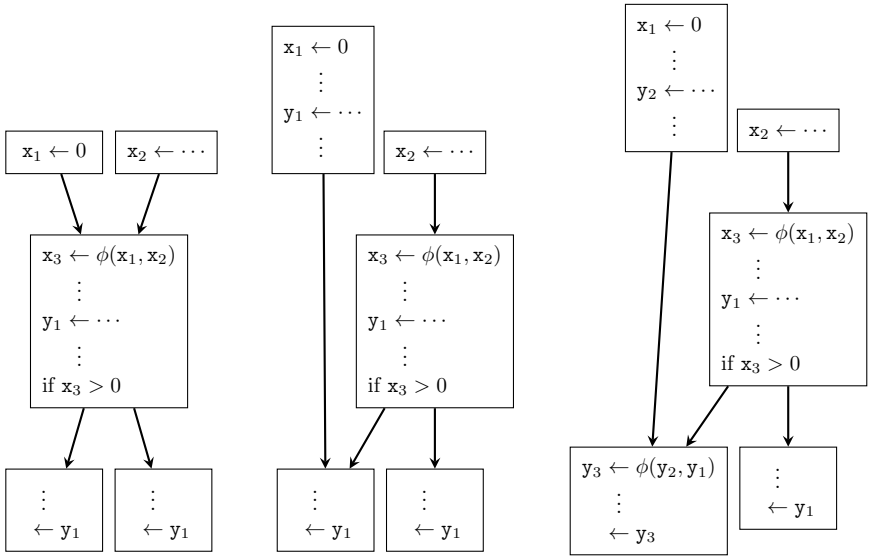
optimizations; owing to the insertion of additional ϕ-functions and the correct redirection of the variable's uses.

6.1.2 Jump Threading

Jump threading is a transformation performed by many popular compilers such as GCC and LLVM. Jump threading is applied to enable more expressive optimizations. Consider following situation: A block contains a conditional branch that depends on some variable `x`. In the example shown in Figure 6.2, the conditional branch tests, if $\mathtt{x} > 0$. Assume that the block containing that conditional branch has multiple predecessors and `x` can be proven constant for one of the predecessors. In the example below, this is shown by the assignment $\mathtt{x}_1 \leftarrow 0$. Jump threading now partially evaluates the conditional branch by directly making the corresponding successor of the branch block a successor of the *predecessor* of the branch. To this end, the code of the branch block is duplicated and attached to the predecessor. This also duplicates potential definitions of variables in the branch block. Hence, SSA is destroyed for those variables and has to be reconstructed. In contrast to the examples above, the control flow has changed however which poses an additional challenge for an efficient SSA reconstruction algorithm.

6.2 General Considerations

In the following, we will discuss two algorithms. The first is an adoption of the classical dominance-frontier based algorithm. The second performs a search from the variables' uses to the definition and places ϕ-functions on demand at appropri-



(a) Original program **(b)** Jump threaded, SSA broken **(c)** SSA repaired

Fig. 6.2 Jump Threading

ate places. In contrast to the first, the second algorithm might not yield minimal SSA form (in Cytron's sense). However, it does not need to update its internal data structures when the CFG is modified.

We consider following scenario: The program is represented as a control-flow graph (CFG) and is in SSA form. For the sake of simplicity, we assume that each instruction in the program only writes to a single variable. Due to the single-assignment property of the SSA form, we can then identify the program point of the instruction and the variable. An optimization/transformation now violates SSA by inserting additional definitions for an existing SSA variable, like in the examples above. The original variable and the additional definitions can be seen as a single non-SSA variable that has multiple definitions and uses.

In the following, v will be such a non-SSA variable. D is a set of SSA variables being the definitions of v. A use of a variable is a pair consisting of a program point (a variable) and an integer denoting the index of the operand at the using instruction.

Both algorithms which we are going to present share the same driver routine (Algorithm 1). First, every basic block `b` is equipped with a list `b.defs` that contains all instructions in the block which define one of the variables in D. This list is sorted according to the schedule of the instructions in the block from back to front. Hence, the latest definition is the first in the list.

Then, all uses of the variables in D are scanned. For every program point ℓ where a variable in D is used, search locally in ℓ's block for a definition of variable in D.

By scanning the block's list from back to front, we find the latest such use if one exists. If the variable has no definition in the block, we have to find the definition that reaches this block from *outside*. Here we have to differentiate whether the user is a ϕ-function or not. If it is a ϕ-function, the block of the use is the corresponding predecessor block. We use two functions `find_def_from_begin` and `find_def_from_end` that find the reaching definition at the beginning and end of a block, respectively. As can seen from Algorithm 1, `find_def_from_end` can be expressed in terms of `find_def_from_begin`. `find_def_from_end` additionally considers definitions in the block, while `find_def_from_begin` does. Our two approaches only differ in the implementation of the function `find_def_from_begin`. The differences are described in the next two sections.

```
proc ssa_reconstruct(set of var D):
  for d in D:
    b = d.block
    insert d in b.defs according to schedule
    # latest definition is first in list

  for each use u of D:
    v = get_used_var(u)
    b = v.block
    d = None

    # search for a local definition in the block
    for e in b.defs:
      if v == e and is_later(u, e):
        d = e
        break

    # no local definition was found, search in the predecessors
    if d == None:
      # if the user is a phi we have to start the search
      # at the end the corresponding predecessor block
      if u.is_phi():
        d = find_def_from_end(u.block.pred(u.index), ...)
      else:
        d = find_def_from_begin(b, ...)

    rewrite use at u to d

proc find_def_from_end(block b, ...):
  if not b.defs.empty():
    return b.defs[0]
  return find_def_from_begin(b, ...)
```

Algorithm 1: SSA Reconstruction Driver

6.3 Reconstruction based on the Dominance Frontier

This algorithm, proposed by Sastry and Ju [**?**], follows the same principles as the classical SSA construction algorithm by Cytron et al. [**?**]. We compute the iterated dominance frontier (IDF) of D (cf. Appel's book [**?**] for an algorithm on how to compute the IDF). This set is a sound over-approximation of the set where ϕ-functions must be placed (it might contain blocks where a ϕ-function would be dead). Then, we search for each use the corresponding reaching definition. This search starts at the block of u. If that block is in the IDF of D a ϕ-function needs to be placed at its entrance. The operands of that ϕ-function are then (recursively) searched in the predecessors of the block. If the block is not in the IDF, the search continues in the block's immediate dominator. This is because in SSA, every use of a variable must be dominated by its definition[1]. If the block is not in the IDF, the reaching definition is the same for all predecessors and hence for the immediate dominator of this block. Note that by rewiring the uses of several variables, some variables defined by ϕ-functions may not be used anymore. A dead code elimination pass after SSA reconstruction will remove these. Algorithm 2 shows this procedure in pseudo-code.

```
proc find_def_from_begin(block b, set of blocks F):
  if b in F:
    d = new_phi(b)
    i = 0
    for p in b.preds:
      o = find_def_from_end(p, F)
      set i-th operand of d to o
      i = i + 1
  else:
    d = find_def_from_end(b.idom, F)
  b.def = d
  return d
```

Algorithm 2: SSA Reconstruction based on Dominance Frontiers

6.4 Search-based Reconstruction

The second algorithm we present here is similar to the construction algorithm that Click describes in his thesis [**?**]. Although his algorithm is designed to construct SSA from the abstract syntax tree, it also works well on control flow graphs. Its major advantage over the algorithm presented in the last section is that it does neither

[1] The definition of an operand of a ϕ-function has to dominate the according predecessor block.

require dominance information nor dominance frontiers. Thus it is well suited to be used in transformations that change the control flow graph. Its disadvantage is that potentially more blocks have to visited during the reconstruction. The principle idea is to start a search from every use to find the corresponding definition inserting ϕ-functions on the fly while caching the found definitions at the basic blocks. This is similar to the implementation of a data-flow analysis, that places the ϕ-functions. As in the last section, we only consider the reconstruction for a single variable. If multiple variables have to be reconstructed, the algorithm can be applied to each variable separately.

We perform a backward depth-first search in the CFG to collect the reaching definitions at each block. To mark a block as visited, the reaching definition of that block is inserted into the `defs` list of that block. If the CFG is a DAG, all predecessors of a block can be visited before the block itself is processed (post-order traversal). Hence, all reaching definitions at a block `b` can be computed before we decide whether to place a ϕ-function in `b` or not. If more than one definition reaches the block, we need to place a ϕ-function.

If the CFG has loops, there are blocks for which not all reaching definitions can be computed before we can decide whether a ϕ-function has to be placed. Recursively computing the reaching definitions for a block `b` can end up at `b` itself. To avoid infinite recursion, we create a ϕ-function without operands in the block before descending to the predecessors. Hence, if a variable has no definition in a loop, the ϕ-function placed in the header eventually reaches itself (and can later be eliminated). When we return to `b` we decide whether a ϕ-function has to be placed in `b` by looking at the reaching definition for every predecessor. If the set of reaching definitions is a subset of $\{a, x\}$ where x is the ϕ-function inserted at `b`, then ϕ-function is not necessary and we can propagate a further downwards. Otherwise, we place a ϕ-function.

```
proc find_def_from_begin(block b):
  phi = make_phi()
  if b.defs.empty():
    b.defs = [ phi ]

  reaching_defs = []
  for p in b.preds:
    reaching_defs += find_def_from_end(p)
  if phi_necessary(reaching_defs):
    set_arguments(phi, reaching_defs)
    d = phi
  else:
    reaching_defs.remove(phi)
    d = reaching_defs[0]
  return d
```

Algorithm 3: Search-based SSA Reconstruction

6.5 Conclusions

Some optimizations, such as loop unrolling or live-range splitting destroy the single-assignment property of the SSA form. In this chapter we presented two generic algorithms to reconstruct SSA. The algorithms are independent of the transformation that violated SSA and can be used as a black box: For every variable for which SSA was violated, a routine is called that restores SSA. The presented algorithms differ in the prerequisites and their runtime behavior:

1. The first is based on the iterated dominance frontiers like the classical SSA construction algorithm by Cytron et al. [**?**]. Hence, it is less suited for optimizations that also change the flow of control since that would require recomputing the iterated dominance frontiers. On the other hand, by using the iterated dominance frontiers, the algorithm can find the reaching definitions quickly by scanning the dominance tree upwards.
2. The second algorithm does not depend on additional analysis information such as iterated dominance frontiers or the dominance tree. Thus, it is well suited for transformations that change the CFG because no information needs to be recomputed. On the other hand, it might find the reaching definitions slower than the first one because they are searched by a depth-first search in the CFG.

Both approaches construct *pruned* SSA (TODO: cite other chapter), i.e. no ϕ-function is dead. One can also show that the first approach produces minimal SSA in the sense of Cytron et al. [**?**] whereas the second approach might create superfluous ϕ-functions in the case of irregular control flow.

CHAPTER 7

Semantics

L. Beringer

Progress: 60%

Structural reordering in progress

In this chapter we discuss alternative representations that highlight aspects of control and data flow structure that lie at the heart of the SSA discipline. The development of such representations is motivated by the following considerations.

- Firstly, the reformulation of the SSA discipline in other formalisms complements the intuitive meaning of "unimplementable" ϕ-instructions and makes syntactic conditions and semantic invariants that are implicit in the definition of SSA more explicit. The introduction of SSA itself was motivated by a similar goal: to represent aspects of program structure, namely the def-use relationships, explicitly in syntax, by enforcing a particular naming discipline. In a similar way, the functional representations treated in this chapter immediately enforce invariants such as "all ϕ-functions in a block must be of the same arity", "the variables assigned to by the ϕ-functions in a block must be distinct", "ϕ-functions are only allowed to occur at the beginning of a basic block", or "each use of a variable should be dominated by its (unique) definition". Consequently, less code is required that validates the well-structuredness of programs between compiler phases, and the robustness and maintainability of compiler frameworks are increased.
- Secondly, alternative representations provide formal criteria with respect to which SSA-based code transformations may be proven correct. For example, Glesner [**?**] uses a representation of SSA in terms of abstract state machines to prove the correctness of a code generation transformation, while Chakravarty et al. [**?**] prove the correctness of a functional representation of Wegmann and Zadeck's SSA-based sparse conditional constant propagation algorithm [**?**]. In the same spirit, these representations provide a formal basis for comparing variants of SSA – such as the variants discussed elsewhere in this book –, for translating between these variants, and for constructing and destructing SSA.

- Thirdly, they facilitate the implementation of interpreters operating at SSA level. This enables the compiler developer to experimentally validate SSA-based analyses and transformations at their genuine language level, prior to SSA destruction.
- Finally, alternative formalisms provide conceptual insight into for the appeal of SSA by relating its core ideas to concepts from other areas of compiler and programming language research.

We first outline representations in functional programming languages. Pioneered by O'Donnell, Kelsey, and Appel [**?**, **?**, **?**], these representations are based on the correspondences between the control flow structure of SSA and a functional programming discipline called continuation-passing-style (short: CPS), and between the constraints on def-use-relationships imposed by ϕ-nodes and the notion of static scope. We examine various aspects of these correspondences in detail, outline how the construction and destruction of SSA can be mirrored by matching operations on functional programs, and indicate how the correspondence may be extended to program analysis frameworks.

We then consider a representation of SSA programs as sets of mutually recursive equations $x_i = e_i$ that stresses data flow aspects of SSA and was originally introduced by Pop [**?**]. Our discussion focuses on some principal underlying this representation, preparing for a more in-depth discussion of advanced topics in Chapter **??**.

We conclude by discussing some pointers to the literature.

7.1 Functional interpretations

Our first model of SSA is provided by functional programming languages and exploits the observation that core properties of SSA have direct counterparts in the world of functional programming. We first describe the functional representations *continuation-passing* and *direct style* and then outline the relationhip to SSA by discussing functional counterparts to the construction and destruction of SSA code.

Like the remainder of the book, our discussion concerns code in a single procedure.

7.1.1 Low-level functional program representations

Functional languages represent a procedure by a declaration

$$\texttt{function}\ f(x_0, \ldots, x_n) = e$$

where the syntactic category of expressions e conflates the notions of expressions and commands of imperative languages.

Variable assignment versus name binding

A language construct provided by almost all functional languages is the let-binding

$$\texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end}.$$

The effect of this expression is to evaluate e_1 and bind the resulting value to variable x for the duration of the evaluation of e_2. The code affected by this binding, e_2, is called the *static scope* of x and is easily syntactically identifiable. In the following, we occasionally indicate scopes by code-enclosing boxes, indicating the variables that are in scope using subscripts.

In contrast to an assignment in an imperative language, a let-binding for variable x hides any previous value bound to x for the duration of evaluating e_2 but does not permanently overwrite it. Thus, bindings are treated in a stack-like fashion, and boxes in our code excerpts are properly nested. For example, in code

$$\begin{array}{l}\texttt{let}\ v = 3\ \texttt{in}\\ \quad\boxed{\begin{array}{l}\texttt{let}\ y = (\texttt{let}\ v = 2 * v\ \texttt{in}\ \boxed{4 * v}_{v}\ \texttt{end})\\ \texttt{in}\ \boxed{y * v}_{v,y}\ \texttt{end}\end{array}}_{v}\\ \texttt{end}\end{array} \tag{7.1}$$

the inner binding of v to value $2 * 3 = 6$ shadows the outer binding of v to value 3 precisely for the duration of the evaluation of the expression $4 * v$. Once this evaluation has terminated (resulting in the binding of y to 24), the binding of v to 3 comes back into force, yielding the overall result of 72.

The concepts of binding and static scope ensure that functional programs enjoy the characteristic feature of SSA, namely the fact that each use of a variable is uniquely associated with a point of definition. Indeed, the point of definition for a use of x is given by the *nearest enclosing binding of x*. Occurrences of variables in an expression that are not enclosed by a binding are called *free*. A well-formed procedure declaration contains all free variables of its body amongst its formal parameters. Thus, the notion of scope makes explicit a crucial invariant of SSA that is often left implicit: each use of a variable should be dominated by its (unique) definition.

In contrast to SSA, functional languages achieve the association of definitions to uses without imposing the global uniqueness of variables, as witnessed by the duplicate binding occurrences for v in the above code. As a consequence of this decoupling, functional languages enjoy a strong notion of referential transparency: the choice of x as the variable holding the result of e_1 depends only on the free variables of e_2. In fact, code (7.1) is equivalent to the fragments

$$\begin{array}{l}\texttt{let}\ v = 3\ \texttt{in}\\ \quad\boxed{\begin{array}{l}\texttt{let}\ y = (\texttt{let}\ z = 2 * v\ \texttt{in}\ \boxed{4 * z}_{v,z}\ \texttt{end})\\ \texttt{in}\ \boxed{y * v}_{v,y}\ \texttt{end}\end{array}}_{v}\\ \texttt{end}\end{array} \tag{7.2}$$

and

$$\begin{array}{l}\texttt{let}\ v = 3\ \texttt{in}\\ \quad\boxed{\begin{array}{l}\texttt{let}\ y = (\texttt{let}\ y = 2 * v\ \texttt{in}\ \boxed{4 * y}_{v,y}\ \texttt{end})\\ \texttt{in}\ \boxed{y * v}_{v,y}\ \texttt{end}\end{array}}_{v}\\ \texttt{end}\end{array} \tag{7.3}$$

that are obtained if we rename the bound variable v in a process called *α-renaming*. Code (7.3) illustrates that additional bindings of x in e_1 do not clash with the outer binding of x in $\texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end}$ as the evaluation of e_1 precedes the binding of its result to x.

In order to illustrate that the choice of a let-bound variable depends on the free variables of e_2, let us consider possible renamings for the variable y in (7.1). Since the scope of y, namely the expression $y * v$, contains the variable v free, we cannot replace y by v but only by some other variable, say y'.

Program analyses for functional languages are typically compatible with α-renaming in that they behave equivalently for fragments that differ only in their choice of bound variables, and program transformations rename bound variables whenever necessary.

A consequence of referential transparency, and thus a property typically enjoyed by functional languages, is *compositional equational reasoning*: the meaning of a piece of code is only dependent on the meaning of its subexpressions. Hence, languages with referential transparency allow one to replace a subexpression by some semantically equivalent phrase without altering the meaning of the surrounding code. Since semantic preservation is a core requirement of program transformations, the suitability of SSA for formulating and implementing such transformations can be explained by the proximity of SSA to functional languages.

O'Donnell [**?**] and Kelsey [**?**] observed that the correspondence between let-bindings and points of variable definition in assignments extends to other aspects of program structure, in particular to code in *continuation-passing-style* (CPS), a program representation routinely used in compilers for functional languages [**?**, **?**].

Control flow: continuations

Satisfying a roughly similar purpose as return addresses or function pointers in imperative languages, a continuation specifies how the execution should proceed once the evaluation of the current code fragment has terminated. Syntactically, continuations are expressions that may occur in functional position (i.e. are typically applied

to argument expressions), as is the case for the variable k in

$$\begin{array}{l}\texttt{let}\ v = 3\ \texttt{in}\\ \quad \texttt{let}\ y = (\texttt{let}\ v = 2 * v\ \texttt{in}\ 4 * v\ \texttt{end})\\ \quad \texttt{in}\ k(y * v)\ \texttt{end}\\ \texttt{end}\end{array}. \tag{7.4}$$

In effect, k represents any function that may be applied to the result of expression (7.1).

Surrounding code may specify the conrecte continuation by binding k to a suitable expression, as in

$$\begin{array}{l}\texttt{let}\ k = \lambda\, x.\, 2 * x\\ \texttt{in}\ \texttt{let}\ v = 3\ \texttt{in}\\ \qquad \texttt{let}\ y = (\texttt{let}\ z = 2 * v\ \texttt{in}\ 4 * z\ \texttt{end})\\ \qquad \texttt{in}\ k(y * v)\ \texttt{end}\\ \quad \texttt{end}\\ \texttt{end}\end{array}$$

or wrap fragment (7.4) in a function definition with formal argument k and construct the continuation in the calling code:

$$\begin{array}{l}\texttt{function}\ f(k) =\\ \quad \texttt{let}\ v = 3\ \texttt{in}\\ \qquad \texttt{let}\ y = (\texttt{let}\ z = 2 * v\ \texttt{in}\ 4 * z\ \texttt{end})\\ \qquad \texttt{in}\ k(y * v)\ \texttt{end}\\ \quad \texttt{end}\\ \texttt{in}\ \texttt{let}\ k = \lambda\, x.\, 2 * x\ \texttt{in}\ f(k)\ \texttt{end}\\ \texttt{end}.\end{array} \tag{7.5}$$

In both cases, the λ-term $\lambda\, x.\, 2 * x$ stipulates that the result of f should be multiplied by 2.

Typically, the caller of f is itself parametric in *its* continuation, as in

$$\begin{array}{l}\texttt{function}\ g(k) =\\ \quad \texttt{let}\ k' = \lambda\, x.\, k(2 * x)\ \texttt{in}\ f(k')\ \texttt{end}.\end{array} \tag{7.6}$$

where f is invoked with a newly constructed continuation k' that applies the multiplication by 2 to its formal argument x (which at runtime will hold the result of f) before passing the resulting value on as an argument to the outer continuation k. In a similar way, the function

```
function h(y, k) =
  let x = 4 in
    let k' = λz. k(z * x)
    in if y > 0
      then let z = y * 2 in k'(z) end
      else let z = 3 in k'(z) end
    end
  end
```
(7.7)

constructs from k a continuation k' that is invoked (with different arguments) in each branche of the conditional. In effect, the sharing of k' amounts to the definition of a control flow merge point, as indicated by the CFG corresponding to h in Figure 7.1 (left).

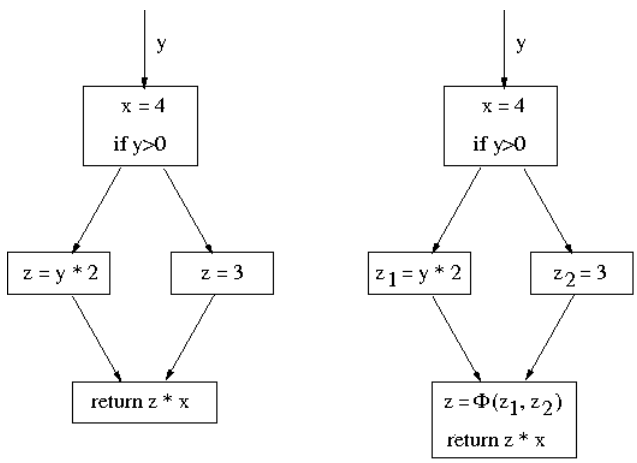


Fig. 7.1 Control flow graph for code (7.7) (left), and SSA representation (right)

The SSA form of this CFG is shown in Figure 7.1 on the right. If we apply similar renamings of z to z_1 and z_2 in the two branches of (7.7), we obtain

```
function h(y, k) =
  let x = 4 in
    let k' = λz. k(z * x)
    in if y > 0
      then let z1 = y * 2 in k'(z1) end
      else let z2 = 3 in k'(z2) end
    end
  end
```
(7.8)

We observe that the role of the formal parameter z of continuation k' is exactly that of a ϕ-function: to unify the arguments stemming from various calls sites by binding

them to a common name for the duration of the ensuing code fragment – in this case just the return expression. As expected from the above understanding of scope and dominance, the scopes of the bindings for z_1 and z_2 coincide with the dominance regions of the identically named imperative variables: both terminate at the point of function invocation / jump to the control flow merge point.

The fact that transforming (7.7) into (7.8) only involves the referentially transparent process of α-renaming indicates that program (7.7) already contains the essential structural properties that SSA distills from an imperative program.

Programs in CPS equip *all* functions declarations which continuation arguments. By interspersing ordinary code with continuation-forming expressions as shown above, they model the flow of control exclusively by communicating, constructing, and invoking continuations.

Before examining further aspects of the relationship between CPS and SSA, we discuss a close relative of CPS, the so-called *direct-style* representation.

Control flow: direct style

An alternative to the explicit passing of continuation terms via additional function arguments is to represent code as a set of locally named tail-recursive functions. Appel [**?**] popularized the correspondence to SSA of this representation, which is called *direct style* [**?**].

In direct style, code (7.7) may be represented as

$$
\begin{array}{l}
\texttt{function}\ h(y) = \\
\quad \boxed{\begin{array}{l}
\texttt{let}\ x = 4\ \texttt{in} \\
\quad \boxed{\begin{array}{l}
\texttt{function}\ h'(z) = \boxed{z * x}_{x,y,z,h,h'} \\
\texttt{in}\ \boxed{\begin{array}{l}
\texttt{if}\ y > 0 \\
\texttt{then let}\ z = y * 2\ \texttt{in}\ \boxed{h'(z)}_{x,y,z,h,h'}\ \texttt{end} \\
\texttt{else let}\ z = 3\ \texttt{in}\ \boxed{h'(z)}_{x,y,z,h,h'}\ \texttt{end}
\end{array}}_{x,y,h,h'} \\
\texttt{end}
\end{array}}_{x,y,h} \\
\texttt{end}
\end{array}}_{y,h}
\end{array}
\tag{7.9}
$$

where the local function h' plays a similar role as the continuation k' and is jointly called from both branches. In contrast to the CPS representation the body of h' returns its result directly rather than by passing it on as an argument to some continuation. Neither the declaration of h nor that of h' contain additional continuation parameters.

A stricter format is obtained if the granularity of local functions is required to be that of basic blocks:

```
function h(y) =
  let x = 4 in
    function h'(z) = z * x
    in if y > 0
       then function h1() = let z = y * 2 in h'(z) end
            in h1() end
       else function h2() = let z = 3 in h'(z) end
            in h2() end
    end
  end
```
(7.10)

Now, function invocations correspond precisely to jumps, reflecting more directly the CFG from Figure 7.1. Both, CPS and direct style are compatible with the strict notion of basic blocks as well as more relaxed ones where, for example, functions that have only a single invocation site are inlined (extended basic blocks). At the other expreme, both representation also admit the explicit naming of all control flow points, i.e. the introduction of one local function or continuation per instruction. The questions whether CPS or direct style should be preferred, and what the appropriate granularity level of functions is, have received considerable attention, with no clear consensus being established. In our discussion below, we employ the arguably easier-to-read direct style, although the gist of the discussion applies equally well to CPS.

Independent of the granularity level of local functions, the process of moving from the CFG to the SSA form is again captured by suitably α-renaming the bindings of z in h_1 and h_2:

```
function h(y) =
  let x = 4 in
    function h'(z) = z * x
    in if y > 0
       then function h1() = let z1 = y * 2 in h'(z1) end
            in h1() end
       else function h2() = let z2 = 3 in h'(z2) end
            in h2() end
    end
  end
```
(7.11)

Again, the role of the formal parameter z of the control flow merge point function h' is identical to that of a ϕ-function. In accordance with the fact that the basic blocks representing the arms of the conditional do not contain ϕ-functions, the local functions h_1 and h_2 have empty parameter lists – the free occurrence of y in the body of h_1 is bound at the top level by the formal argument of h.

For both direct style and CPS the correspondence to SSA is most pronounced for code in *let-normal-form*: each intermediate result must be explicitly named by a variable, and function arguments must be names or constants. Syntactically, let-normal-form isolates basic instructions in a separate category of primitive terms a

and then requires let-bindings to be of the form $\texttt{let}\ x = a\ \texttt{in}\ e\ \texttt{end}$. In particular, neither jumps (conditional or unconditional) nor let-bindings are primitive. The let-normalized form of (7.2),

$$\begin{array}{l}\texttt{let}\ v = 3\ \texttt{in}\\ \quad\boxed{\begin{array}{l}\texttt{let}\ z = 2 * v\ \texttt{in}\\ \quad\boxed{\texttt{let}\ y = 4 * z\ \texttt{in}\ \boxed{y * v}_{v,y,z}\ \texttt{end}}_{v,z}\\ \texttt{end}\end{array}}_{v}\\ \texttt{end}\end{array} \tag{7.12}$$

is obtained by pulling the let-binding for z from the e_1-position to the outside of the binding for y. In general, the transformation repeatedly rewrites

$$\begin{array}{l}\texttt{let}\ x = \texttt{let}\ y = e_1\ \texttt{in}\ \boxed{e_2}_y\ \texttt{end}\\ \texttt{in}\ \boxed{e_3}_x\\ \texttt{end}\end{array} \quad \text{into} \quad \begin{array}{l}\texttt{let}\ y = e_1\\ \texttt{in}\ \boxed{\texttt{let}\ x = e_2\ \texttt{in}\ \boxed{e_3}_{x,y}\ \texttt{end}}_y\\ \texttt{end},\end{array}$$

subject to the side condition that y not be free in e_3.

Programs in let-normal form thus do not contain let-bindings in the e_1-position of outer let-expressions. The stack discipline in which let-bindings are managed is simplified as scopes are nested inside each other[1]. While still enjoying referential transparency, let-normal code is in closer correspondence to imperative code than nonnormalized code as the chain of nested let-bindings directly reflects the sequence of statements in a basic block, interspersed occasionally by the definition of continuations or local functions.

Summarizing our discussion up to this point, Table 7.1 collects some correspondences between functional and imperative/SSA concepts.

Functional concept	Imperative/SSA concept
variable binding in let	assignment (point of definition)
α-renaming	variable renaming
unique association of binding occurrences to uses	unique association of defs to uses
formal parameter of continuation/local function	ϕ-function (point of definition)
lexical scope of bound variable	dominance region

Table 7.1 Correspondence pairs between functional form and SSA (part I)

[1] Provided the continuation/function definitions are closed, this means that bindings can be implemented destructively, i.e. as assignments/updates.

7.1.2 Functional construction of SSA

The relationship between SSA and functional languages is extended by the correspondences shown in Table 7.2. We discuss some of these aspects by considering the translation into SSA, using the program in Figure 7.2 as a running example.

Functional concept	Imperative/SSA concept
subterm relationship	control flow successor relationship
arity of function f_i	number of ϕ-functions at beginning of b_i
distinctness of formal parameters of f_i	distinctness of LHS-variables in the ϕ-block of b_i
number of call sites of function f_i	arity of ϕ-functions in block b_i
parameter lifting/dropping	addition/removal of ϕ-function
block floating/sinking	reordering according to dominator tree structure
potential nesting structure	dominator tree
nesting level	maximal level index in dominator tree

Table 7.2 Correspondence pairs between functional form and SSA: program structure

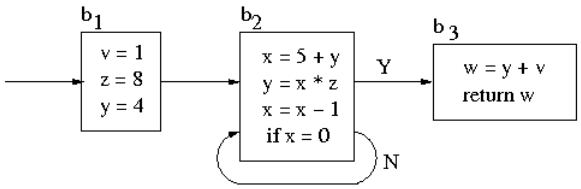


Fig. 7.2 Functional construction of SSA: running example

Initial construction using liveness analysis

A simple way to represent this program in let-normalized direct style is to introduce one function f_i for each basic block b_i. The body of each f_i arises by introducing one let-binding for each assignment and converting jumps into function calls. In order to determine the formal parameters of these functions we perform a liveness analysis. For each basic block b_i, we choose an arbitrary enumeration of its live-in variables. We then use this enumeration as the list of formal parameters in the declaration of the function f_i, and also as the list of actual arguments in calls to f_i. We organize all function definitions in a single block of mutually tail-recursive functions at the top level:

$$
\begin{array}{l}
\texttt{function}\ f_1() = \\
\quad\quad \texttt{let}\ v = 1\ \texttt{in}\ \texttt{let}\ z = 8\ \texttt{in}\ \texttt{let}\ y = 4\ \texttt{in}\ f_2(v, z, y)\ \texttt{end end end} \\
\texttt{and}\ f_2(v, z, y) = \texttt{let}\ x = 5 + y\ \texttt{in}\ \texttt{let}\ y = x * z\ \texttt{in}\ \texttt{let}\ x = x - 1\ \texttt{in} \\
\quad\quad\quad\quad \texttt{if}\ x = 0\ \texttt{then}\ f_3(y, v)\ \texttt{else}\ f_2(v, z, y)\ \texttt{end end end} \\
\texttt{and}\ f_3(y, v) = \texttt{let}\ w = y + v\ \texttt{in}\ w\ \texttt{end} \\
\texttt{in}\ f_1()\ \texttt{end}
\end{array}
\tag{7.13}
$$

The resulting program has the following properties:

- all function declarations are *closed*: the free variables of their bodies are contained in their formal parameter lists[2];
- variable names are not unique, but the unique association of definitions to uses is satisfied;
- each subterm e_2 of a let-binding $\texttt{let}\ x = e_1\ \texttt{in}\ e_2\ \texttt{end}$ corresponds to the control flow successor of the assignment to x.

If desired, we may α-rename to make names globally unique. As all function declarations are closed, the renamings are independent. The resulting program

$$
\begin{array}{l}
\texttt{function}\ f_1() = \\
\quad\quad \texttt{let}\ v_1 = 1\ \texttt{in}\ \texttt{let}\ z_1 = 8\ \texttt{in}\ \texttt{let}\ y_1 = 4\ \texttt{in}\ f_2(v_1, z_1, y_1)\ \texttt{end end end} \\
\texttt{and}\ f_2(v_2, z_2, y_2) = \texttt{let}\ x_1 = 5 + y_2\ \texttt{in}\ \texttt{let}\ y_3 = x_1 * z_2\ \texttt{in}\ \texttt{let}\ x_2 = x_1 - 1\ \texttt{in} \\
\quad\quad\quad\quad \texttt{if}\ x_2 = 0\ \texttt{then}\ f_3(y_3, v_2)\ \texttt{else}\ f_2(v_2, z_2, y_3)\ \texttt{end end end} \\
\texttt{and}\ f_3(y_4, v_3) = \texttt{let}\ w_1 = y_4 + v_3\ \texttt{in}\ w_1\ \texttt{end} \\
\texttt{in}\ f_1()\ \texttt{end}
\end{array}
\tag{7.14}
$$

is an SSA-program in disguise (cf. Figure 7.3): each formal parameter of a function f_i is the target of one ϕ-function for the corresponding block b_i. The arguments of these ϕ-functions are the arguments in the corresponding positions in the calls to f_i. As the number of arguments in each call to f_i coincides with the number of f_i's formal parameters, the ϕ-functions in b_i are all of the same arity, namely the number of call sites to f_i. In order to coordinate the relative positioning of the arguments of the ϕ-functions, we choose an arbitrary enumeration of these call sites.

Under this perspective, the above construction of parameter lists amounts to equipping each b_i with ϕ-functions for all its live-in variables, with subsequent renaming of the variables. Thus, the above method corresponds to the construction of *pruned* (but not minimal) SSA – see Chapter 2.

While resulting in a legal SSA program, the construction clearly introduces more ϕ-functions than necessary. Each superfluous ϕ-function corresponds to the situation where all call sites to some function f_i pass identical arguments. The technique for eliminating such arguments is called *λ-dropping* [**?**], the inverse of the more widely known transformation *λ-lifting* [**?**].

[2] Apart from the function identifiers f_i which can always be chosen distinct from the variables

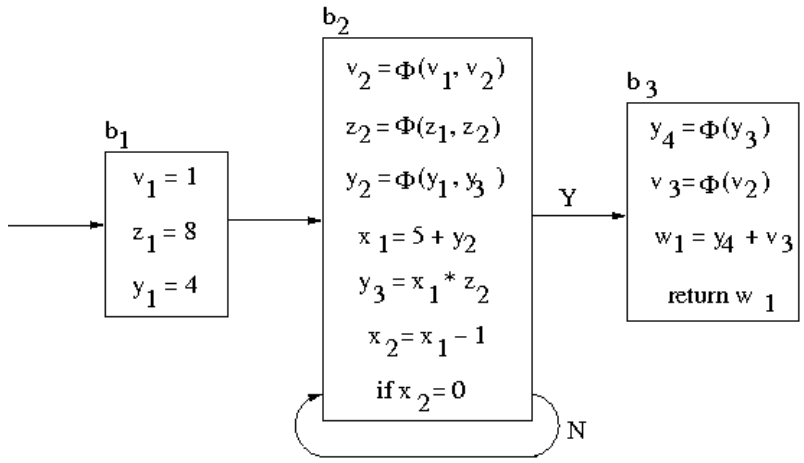


Fig. 7.3 Pruned (non-minimal) SSA form

λ-dropping

λ-dropping may be performed before or after variable names are made distinct, but for our purpose, the former option is more instructive. The transformation consists of two phases, *block sinking* and *parameter dropping*.

Block sinking analyzes the static call structure to identify which function definitions may be moved inside each other. Whenever our set of function declarations contains definitions $f(x_1, \ldots, x_n) = e_f$ and $g(y_1, \ldots, y_m) = e_g$ where $f \neq g$ such that all calls to f occur in e_f or e_g, we can move the declaration for f into that of g. In our example, f_3 is only invoked from within f_2, and f_2 is only called in the bodies of f_2 and f_1. We may thus move the definition of f_3 into that of f_2, and the latter one into f_1.

Several options exist as to where f should be placed in its host function. The first option is to place f at the beginning of g, by rewriting to

$$\texttt{function}\ g(y_1, \ldots, y_m) = \begin{array}{l}\texttt{function}\ f(x_1, \ldots, x_n) = e_f \\ \texttt{in}\ e_g\ \texttt{end}\end{array}$$

in general and to

```
function f1() =
  function f2(v, z, y) =
    function f3(y, v) = let w = y + v in w end
    in let x = 5 + y in let y = x * z in let x = x − 1 in
       if x = 0 then f3(y, v) else f2(v, z, y) end end end
    end
  in let v = 1 in let z = 8 in let y = 4 in f2(v, z, y) end end end
in f1() end
```
(7.15)

in the case of our example program. Note that since the declaration of f is closed, moving it into the scope of g's formal parameters $y_1, \ldots, y_m$ (and also into the scope of g itself) is harmless.

A preferable alternative is to insert f near the end of its host function, in the vicinity of its calls:

```
function f1() =
  let v = 1 in let z = 8 in let y = 4
  in function f2(v, z, y) =
       let x = 5 + y in let y = x * z in let x = x − 1
       in if x = 0
          then function f3(y, v) = let w = y + v in w end
               in f3(y, v) end
          else f2(v, z, y)
       end end end
     in f2(v, z, y) end
  end end end
in f1() end
```
(7.16)

This brings the declaration of f additionally into the scope of let-bindings in e_g, but functional transparency is again respected due to the fact that the declaration is closed.

The second phase, parameter dropping, removes superfluous parameters based on the syntactic scope structure: a parameter x may be removed from the declaration of and calls to f if

- the scope in force for x at the declaration of f coincides with the scope in force for x at each call site to f outside f's declaration;
- the scope in force for x at any recursive call to f is the one associated with the formal parameter x in f's declaration.

In (7.16), these conditions sanction the removal of both parameters from the non-recursive function f_3. The scope applicable for v at the site of declaration of f_3 and also at its call site is the one rooted at the formal parameter v of f_2. In case of y, the common scope is the one rooted at the let-binding for y in the body of f_2. We thus obtain

```
function f1() =
  let v = 1 in let z = 8 in let y = 4
  in function f2(v, z, y) =
       let x = 5 + y in let y = x * z in let x = x − 1
        in if x = 0
           then function f3() = let w = y + v in w end
                in f3() end
           else f2(v, z, y)
       end end end
     in f2(v, z, y) end
  end end end
in f1() end
```
(7.17)

Considering the recursive function f_2 next we observe that the recursive call is in the scope of the let-binding for y in f_2's body, preventing us from removing y. In contrast, neither v nor z have binding occurrences in the body of f_2. The scopes applicable at the external call site to f_2 coincide with those applicable at its site of declaration and are given by the scopes rooted in the let-bindings for v and z. Thus, parameters v and z may be removed from f_2, yielding

```
function f1() =
  let v = 1 in let z = 8 in let y = 4
  in function f2(y) =
       let x = 5 + y in let y = x * z in let x = x − 1
        in if x = 0
           then function f3() = let w = y + v in w end
                in f3() end
           else f2(y)
       end end end
     in f2(y) end
  end end end
in f1() end
```
(7.18)

as the result of parameter dropping and hence λ-lifting. Interpreting the uniquely-renamed variant of (7.18) back in SSA yields the desired minimal code with a single ϕ-function, for variable y at the beginning of block b_2 – see Figure 7.4. The reason that this ϕ-function can't be eliminated (the fact that y is redefined in the loop) is precisely the reason why y survives λ-dropping.

Given this understanding of parameter dropping we can also see why code (7.16) is preferable to code (7.15): the placement of function declarations in the vicinity of their calls potentially enables the dropping of further parameters, namely those that are let-bound in the host function's body.

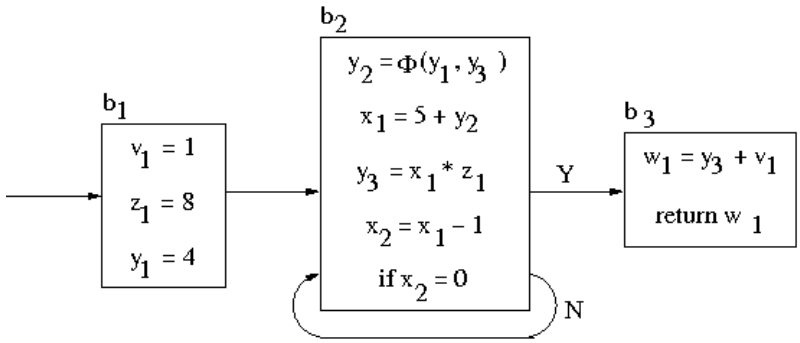


Fig. 7.4 SSA code after λ-dropping

Nesting, dominance, loop-closure

Analyzing whether function definitions may be nested inside one another is tantamount to analyzing the imperative dominance structure: function f_i may be moved inside f_j exactly if all non-recursive calls to f_i come from within f_j exactly if all paths from the initial program point to block b_i traverse b_j exactly if b_j dominates b_i. This observation is merely the extension to function identifiers of our earlier remark that lexical scope coincides with the dominance region, and that points of definition/binding occurrences should dominate the uses. Indeed, functional languages do not distinguish between code and data when aspects of binding and use of variables are concerned, as witnesses by our use of the let-binding construct for binding code-representing expressions to the variables k in our syntax for CPS. The fact that they also treat functions like data (as "first class citizens") when argument and result passing are concerned makes functional languages particularly suitable for the extension of (SSA-based) program analyses to inter-procedural analyses[3].

The *optimal* nesting structure is thus given by the dominator tree: the maximal level at which a function may occur is its level (counting from the root) in the dominator tree.

The choice as to where functions are placed corresponds to variants of SSA. For example, loop-closed SSA form [**?**, **?**] requires the insertion of ϕ-nodes for all variables that are modified in a loop. This enables one to merge copies of these variables that arise when the loop is unrolled. In the functional setting, this discipline amounts to a small modification of block-sinking: functions f that are called from within a *recursive* function g are placed at the *same* level as g rather than *inside* g. Returning to our running example, we place f_3 at the same level as f_2, in contrast to code (7.16). The placement of both functions relative to f_1 is left unaltered.

[3] This sentence should probably be moved to the subsection on type systems, once that has been written.

```
function f1() =
  let v = 1 in let z = 8 in let y = 4
  in function f3(y, v) = let w = y + v in w end
     and f2(v, z, y) =
        let x = 5 + y in let y = x * z in let x = x - 1
        in if x = 0 then f3(y, v) else f2(v, z, y)
        end end end
     in f2(v, z, y) end
  end end end
in f1() end
```
(7.19)

As a consequence, any parameter of f that is rebound in g cannot be dropped. In the example, y is not deleted from the parameter list of f_3, as the declaration of f_3 is not any longer in the scope of the binding for y that applies at the call to f_3. In contrast, v may still be dropped from f_3, and v and z may be dropped from f_2:

```
function f1() =
  let v = 1 in let z = 8 in let y = 4
  in function f3(y) = let w = y + v in w end
     and f2(y) =
        let x = 5 + y in let y = x * z in let x = x - 1
        in if x = 0 then f3(y) else f2(y)
        end end end
     in f2(y) end
  end end end
in f1() end
```
(7.20)

If we unroll f_2 in this loop-closed form by replacing the recursive call to f_2 by one to its copy f_4 – nesting f_4 inside f_2 allows us to immediately parameter-drop y – we obtain

```
function f1() =
  let v = 1 in let z = 8 in let y = 4
  in function f3(y) = let w = y + v in w end
     and f2(y) =
        let x = 5 + y in let y = x * z in let x = x - 1
        in if x = 0
           then f3(y)
           else function f4() =
                   let x = 5 + y in let y = x * z in let x = x - 1
                   in if x = 0 then f3(y) else f2(y)
                   end end end
                in f4() and
        end end end
     in f2(y) end
  end end end
in f1() end
```
(7.21)

This code exhibits the same sharing as the loop-unrolled SSA program shown at the top of Figure 7.5. The invocations sites to f_3 correspond to the control flow arcs with

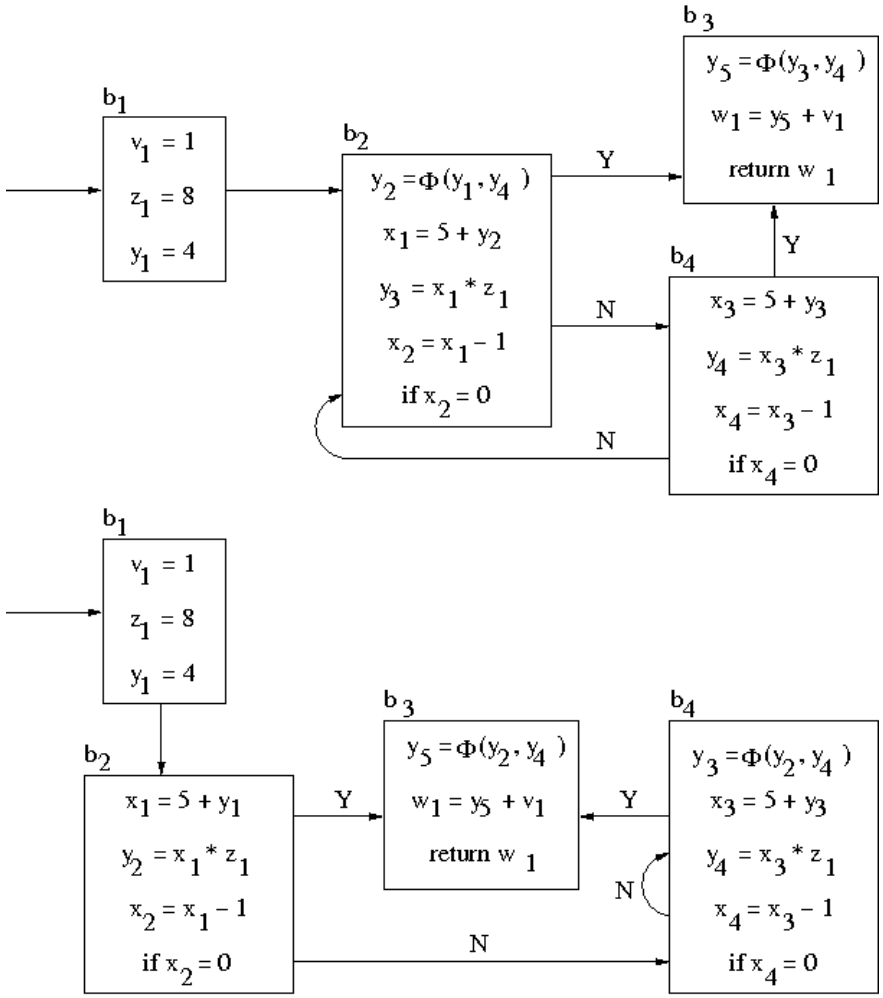


Fig. 7.5 Two outcomes of loop-unrolling, corresponding to programs (7.21) and (7.22)

target b_3, each passing the appropriate value to the "loop closing" parameter y of f_3.

The lower half of Figure 7.5 shows an alternative outcome of loop unrolling. Here, the resulting loop encompasses only b_4. We obtain corresponding functional code if – starting again from (7.20) – we first place the initial copy of f_2 (again named f_4) at the same nesting level as f_2 and replace the call to f_2 inside f_4 by a recursive call to f_4. The only invocation of f_2 that remains is that in f_1, whereas two invocation sites exist for f_4. We then perform λ-dropping, which moves the

definition f_4 inside that of f_2 and also drops the parameter y from the declaration of f_2:

```
function f1() =
  let v = 1 in let z = 8 in let y = 4
  in function f3(y) = let w = y + v in w end
     and f2() =
       let x = 5 + y in let y = x * z in let x = x - 1
       in if x = 0
          then f3(y)
          else function f4(y) =
                 let x = 5 + y in let y = x * z in let x = x - 1
                 in if x = 0 then f3(y) else f4(y)
                 end end end
               in f4() and
       end end end
     in f2() end
  end end end
in f1() end
```
(7.22)

Destruction of SSA

The above example code excerpts where variables are not made distinct exhibit a further pattern: the argument list of any call coincides with the list of formal parameters of the invoked function. This discipline is not enjoyed by functional programs in general, and is often destroyed by optimizaing program transformations. However, programs that do obey this discipline can be immediately converted to imperative non-SSA form. Thus, the task of SSA destruction amounts to converting a functional program with arbitrary argument lists into one where argument lists and formal parameter lists coincide for each function. This is achieved by introducing additional let-bindings of the form `let` $x = y$ `in` e `end`: for example, a call $f(v, z, y)$ where f is declared as `function` $f(x, y, z) = e$ may be converted to

`let` $x = y$ `in let` $y = z$ `in let` $z = x$ `in let` $x = a$ `in` $f(x, y, z)$ `end end end end`,

in correspondence to the move instructions introduced in imperative formulations of SSA destruction (see Chapters 3.4 and [**?**]). Beringer [**?**] calls the appropriate transformation on a functional representation GNF-conversion – note that the target language is only syntactically a functional language as it is not immune against α-renaming – and presents a simple local algorithm that considers each call site individually. A single additional variable suffices for performing all necessary insertions of let-bindings, in line with the results of [**?**]. Rideau et al. [**?**] present an in-depth study of this conversion problem, including a verified implementation in the proof asistant Coq.

7.1.3 Program analyses

[4] The correspondence between liveness and free occurrences of names manifests itself in the striking structural similarity between the liveness-equation for assignments

$$\mathsf{LV}([x := a]^i) = \mathsf{Use}(a) \cup (\mathsf{LV}(succ(i)) \setminus \mathsf{Defs}(i))$$

(where i denotes the assignment's program point) and the clause for let-bindings in the definition of free variables,

$$\mathsf{FV}(\texttt{let}\ x = a\ \texttt{in}\ e\ \texttt{end}) = \mathsf{Use}(a) \cup (\mathsf{FV}(e) \setminus \{x\}).$$

In fact, the *least* solution to the liveness equations cannot only be used to determine the formal parameters of the local functions but in fact assigns each program point i (even intermediate ones) exactly the free non-functional variables of the subexpression[5] corresponding to i.
[6]

Program analyses for functional languages are typically formulated as *type systems*. Table 7.3 collects some correspondence pairs that relate concepts from type systems to notions from dataflow analysis frameworks. A typical type judgement

Functional concept	Imperative/SSA concept
free variable	live-in variable (least solution)
type systems	dataflow frameworks
typing context	scope-aware symbol table
typing rules	dataflow (in-)equations/transfer functions
subtype relationship	merge operator
typing type inference	fixed point iteration
derivations	solutions to dataflow equations
polymorphism/intersection types	polyvariance

Table 7.3 Correspondence pairs between functional form and SSA: program analyses

$\Gamma \vdash e : \tau$ associates a type τ to an expression e, based on typing assumptions in context Γ. Usually, contexts track the types of (at least) the free names of e, similarly to a symbol table in an imperative analysis. Thus, almost any type system is an extension of the concept of free variables, turning the above relationship between liveness

[4] This section still needs polishing/reformulation

[5] even if variables not globally unique? Add example!

[6] In principle, the parameter lists can be constructed from any solution to the liveness-*in*equations. These arise by replacing = with $\supseteq$ in the dataflow equations. Using inequations rather than equations allows functions to have more formal parameters than strictly necessary. Requiring all parameter lists to be chosen according to the *same* solution prevents (ill-defined) functions whose bodies contain free variables that are not amongst the formal parameters. In particular, including all variables in all parameter lists constitutes a solution to the inequations (and legal functional and SSA programs) but not necessarily a solution to the equations.

and free variables into an instance of the given analogies. The distinctness of formal parameters, the distinctness of function names in function declaration blocks, and similar syntactic restrictions, may be easily enforced by equipping the corresponding typing rules with additional side-conditions, and are in any case enforced by many functional languages as part of the language definition.

A major benefit of SSA for dataflow analyses is the avoidance of variables that have several unrelated uses but happen to be identically named. Even in the absence of globally unique names, this property is enjoyed by type systems, as the adaptation of type contexts in the rule for let-bindings is compatible with referential transparency.

Imperative analysis frameworks employ transfer functions for relating the information associated with adjacent program points. In accordance with the correspondence between the control flow successor relation and the subterm relationship, this role is in type systems played by syntax-directed typing rules. Merge operators at control flow merge points correspond to appropriate notions of subtyping.

The correspondent to fixed point algorithms for obtaining dataflow solutions is type inference. Both tasks proceed algorithmically in a structurally equal fashion, along the control flow successor-/predecessor relationship or sub-/superterm relationship. Finally, *solutions* of dataflow analyses arise when all constraints are met – in type systems, the corresponding notion is that of a successful typing derivation.

As many functional languages support high-order functions, type systems are particularly well suited for formulating inter-procedural analyses[7].

7.2 Data flow representation

[8] Our second model of SSA dispenses with the control flow structure entirely, by eliminating any tangible forms of basic blocks or program order. Instead, the model – introduced by Pop [**?**] – emphasizes dataflow aspects, i.e. the flow of values along the def-use-chains. Programs are represented as collections of defining equations,

$$\begin{aligned} x_1 &= e_1 \\ &\vdots \\ x_n &= e_n \end{aligned}$$

one for each variable x_i. Contrary to the functional representation, the right-hand sides e_i of these equations do not refer to *control flow successors* but to *data flow predecessors*, i.e. to the variables that provide the operands necessary for updating x_i. As a consequence, the order in which equations are presented is irrelevant, and execution proceeds completely data-driven.

[7] Maybe the author of the chapter on inter-procedural analyses can briefly take up this point, allowing me to insert a forward-reference here?

[8] This section has to be rewritten.

In order to transform a sequence of assignments into this form we may apply the approach for converting a basic block into SSA: we introduce a new variable for each assignment, and substitute these variables in the right-hand sides of the instructions according to the data flow. The order of assignments may be permuted arbitrarily, so a sequence like $x := 5;\ y := x + z;\ x := y * 3$ may, for example, be represented by the equations

$$\begin{aligned} x_2 &= y_2 + 3 \\ y_1 &= x_1 + z_1 \\ x_1 &= 5 \end{aligned}$$

(variable z is live-in here).

In order to transform loops, the category of right-hand side expressions e is extended by two novel operations, $\texttt{loop}_\ell(e, e')$ and $\texttt{close}_\ell(e, e')$. Both operations resemble ϕ-instructions, but their arity is independent of any control flow structure: e and e' are expressions and ℓ is an index from $\{1, \ldots, N\}$ where N is the number of while-statements in the original program. An equation

$$x = \texttt{loop}_\ell(e, e')$$

roughly corresponds to the occurrence of a ϕ-node for x at the beginning of a loop in SSA, assigning e to x during the first iteration of loop ℓ and assigning e' to x in later iterations. An equation

$$x = \texttt{close}_\ell(e, e')$$

corresponds roughly to a loop-closing ϕ-node for the loop ℓ. Expression e represents the boolean loop condition, and e' represents the value that will be assigned to x when the loop is left, i.e. when e evaluates for the first time to `false`.

For example, the representation of `i = 7; j = 0; while (j < 10) {j = j + i}` (taken from [**?**]) contains the five equations

$$\begin{array}{ll} i_1 = 7 & j_2 = \texttt{loop}_1(j_1, j_3) \\ j_1 = 0 & j_4 = \texttt{close}_1(j_2 < 10, j_2) \\ j_3 = j_2 + i_1 & \end{array}$$

where 1 is the (trivial) index for the single while-command occurring in the program. In effect, j_4 is assigned the value held in j_2 in that iteration for which $j_2 < 10$ is falsified for the first time, i.e. 14.

In order to formally define concepts such as *for the first time*, the representation is equipped with a semantics that employs so-called *iteration space vectors*: for a program with N loops, such a vector consists of an N-tuple of values, with the value at position ℓ representing the number of iterations of loop ℓ. Given such a vector k, the meaning of a right-hand-side expression e is given recursively as follows:

- constant expressions and arithmetic operators have their standard (iteration space vector independent) meaning.
- a variable x is interpreted by evaluating its defining equation at the iteration space vector k.
- an expression $\texttt{loop}_\ell(e_1, e_2)$ evaluates to the value of e_1 at k if k at index ℓ is zero. Otherwise, $\texttt{loop}_\ell(e_1, e_2)$ evaluates to the result of evaluating e_2 at k', where k' is obtained by decrementing index ℓ of k by one.
- an expression $\texttt{close}_\ell(e_1, e_2)$ evaluates to the value of e_2 at the iteration space vector k' that arises by updating k by the least value x such that e_1 for k' is `false`.

The semantics for the entire program is then given in denotational style, by a mapping that associates each variable to the interpretation of its right-hand side, i.e. to the function that given an iteration space vector evaluates the expression on that vector.

Due to the decrementing of the iteration index in the interpretation of $\texttt{loop}_\ell(e_1, e_2)$ expressions, an evaluation of this expression for a particular k only requires further evaluations at vectors that are smaller with respect to the component-wise ordering on tuples, with least element $(0, \ldots, 0)$. In contrast, the minima mentioned in the interpretation of $\texttt{close}_\ell(e_1, e_2)$ do not necessarily exist. In each such case, the interpretation of the equation in question, and thus the corresponding variable, is set to $\perp$, in accordance with the standard treatment of non-termination in denotational semantics [**?**].

In contrast to ϕ-instructions in SSA, the semantics of the equation-based representation thus does not require control-flow information to be maintained, as the choice as to which argument of $\texttt{loop}_\ell(e, e')$ is evaluated is encoded in the dependency on the entry at the appropriate position in the iteration space vector.

The article [**?**] and Pop's dissertation [**?**] contain formal details about the representation, its interpretation, and its formal relationship to a non-SSA language. In particular, these sources explain how a conventional program of assignments and loops may be compositionally converted into a set of equations, in a semantics-preserving way. Source program expressions are uniquely labeled, so that globally unique variable names can be generated by differentiating the original program variables according to the (naturally distinct) labels for assignments. Contrary to the unstructured labels used in [**?**], the authors use a class of labels ("Dewey-like numbers") with the following three properties.

extensibility: this feature is used for generating fresh target variables for ϕ-operations

hierarchical structure according to the subterm relation: this admits a structure-directed translation into the equation-based representation

compatibility with the control flow successor relationship: this feature – in combination with the iteration space vectors – is employed to define a compositional (denotational) semantics of the source language.

In addition to proving a suitable theorem asserting that the translation is semantics-preserving, the authors also give a reading of this result in terms of classical models

of computations by interpreting it as the embedding of the RAM model into the model of partial recursive functions. Similar to out-of-SSA translation, a conversion is defined (and proven correct) that transforms systems of equations into imperative programs. Finally, Pop's dissertation [**?**] describes how a number of program analyses may be phrased in terms of the equational language, including induction variable analysis and other loop optimizations.

7.3 Pointers to the literature

The concept of continuations was introduced multiple times, the earliest discoveries being attributed by Reynolds [**?**] and Wadsworth [**?**] to van Wijngaarden [**?**] and Landin [**?**]. Early uses and studies of CPS include [**?**, **?**].

The relative merits of the various functional representations remain an active area of research, in particular with respect to their integration with program analyses and optimizing transformations, and conversions between these formats. Recent contributions include [**?**, **?**, **?**].

Occasionally, *direct style* refers to the combination of tail-recursive functions and let-normal form. Variations of this discipline include *administrative normal form* (A-normal form, ANF [**?**]), B-form [**?**], and SIL [**?**].

Closely related to continuations and direct-style functional representations are *monadic* languages such as Benton et al.'s MIL [**?**] and Peyton-Jones et al.'s language [**?**]. These partition expressions into a category of *values* and *computations*, similar to the isolation of primitive terms in let-normal form (see also [**?**, **?**]). This allows one to treat side-effects (memory access, IO, exceptions,...) in a uniform way, following Moggi [**?**].

Regarding formally worked-out instantiations of the correspondences for program analyses, Chakravarty et al. present a functional analysis of sparse constant propagation [**?**]. Beringer et al. [**?**] consider data flow equations for liveness and read-only variables, and formally translate their solutions to properties of corresponding typing derivations. Laud et al. [**?**] present a formal correspondence between dataflow analyses and type systems but consider a simple imperative language rather than SSA or a functional representation. The textbook [**?**] presents a unifying perspective on program analysis techniques, including data flow analysis, abstract interpretation, and type systems.

Modern textbooks on programming language semantics and type systems include [**?**, **?**, **?**].

Part II
Extensions

Progress: 34%

Progress: 61%

TODO: Aimed at experts, but with a track for beginners

TODO: Section on Sigma-SSA?

TODO: Section on Dynamic SSA?

CHAPTER 8

Introduction

V. Sarkar

Progress: 0%

Material gathering in progress

8.1 TODO

Author: Sarkar

CHAPTER 9

Array SSA Form

Vivek Sarkar
Kathleen Knobe
Stephen Fink

Progress: 45%

Material gathering in progress

9.1 Introduction

In this chapter, we introduce an Array SSA form that captures precise element-level data flow information for array variables, and coincides with standard SSA form when applied to scalar variables. It can also be applied to structures, objects and other variable types that can be modeled as arrays. As we will see, Array SSA form has a more powerful ϕ function than scalar SSA form, since it can merge values from distinct definitions on an element by element basis. There are several potential uses for Array SSA form in compiler analysis and optimization for uniprocessor and multiprocessor systems. In this chapter, we will use *constant propagation* and *conditional constant propagation* as exemplars of program analyses that can be extended to array variables using Array SSA form, and *redundant load elimination* and *dead store elimination* as exemplars of program optimizations that can be extended to array variables and heap objects using Array SSA form. As with many algorithms based on scalar SSA form, the algorithms presented in this chapter are linear in the size of the Array SSA form representation. Though Array SSA form can be made manifest at run-time (*e.g.,* when enabling parallelization via storage duplication [**?**]), all the algorithms described in this chapter use Array SSA form as a basis for program analysis, which means that the Array SSA form structures can be removed after the program properties of interest have been discovered.

The rest of the chapter is organized as follows. Section 9.2 reviews full Array SSA form for run-time evaluation as in [**?**], and also introduces partial Array SSA form for static analysis. Section 9.3 describes how we extend the constant propagation lattice so that it can efficiently record information about array elements. Section 9.4 presents an extension to the Sparse Constant propagation (SC) algorithm from [**?**] that enables constant propagation through array elements. For simplicity, the algorithm in section 9.4 is restricted to cases in which both the subscript and the value of an array definition are constant. Section 9.5 generalizes the algorithm from section 9.4 so that it can operate on non-constant (symbolic) array subscripts as well *e.g.,* to propagate a def such as $A[m] := 99$ into a use of $A[m]$ even if m is not a constant. Section 9.6 shows how Array SSA form can be extended to support analysis and optimization of object field and array element accesses in strongly typed languages, and section 9.7 contains our conclusions.

9.2 Array SSA Form

The goal of Array SSA form is to provide the same benefits for arrays that traditional SSA provides for scalars. Section 9.2.1 summarizes the *full Array SSA form* introduced in [**?**]. Full Array SSA form provides exact use-def information at run-time for each dynamic read access of an array element. Section 9.2.2 introduces *partial Array SSA form* as a representation for static analysis; partial Array SSA form provides conservative use-def information at compile-time for each static read access of an array element. Throughout this chapter, we assume that all array operations in the input program are expressed as reads and writes of individual array elements. The extension to more complex array operations (*e.g.,* on array sections or whole arrays) has been omitted to simplify the presentation of this chapter.

9.2.1 Full Array SSA Form

As a reminder, the salient properties of traditional scalar SSA form are as follows [**?**]:

1. Each definition is assigned a unique name.
2. At certain points in the program, new names are generated which combine the results from several definitions. This combining is performed by a ϕ function which determines which of several values to use, based on the flow path traversed.
3. Each use refers to exactly one name generated from either of the two rules above.

It is important to note that the ϕ function in traditional SSA form statement, $S_3 := \phi(S_1, S_2)$, is not a pure function of S_1 and S_2 because its value depends on the control flow path taken to reach the statement.

```
if (C) then
   S := ...
else
   S := ...
end if
```

Fig. 9.1 Control Flow with Scalar Definitions

```
@S1 := ( ) ; @S2 := ( )

if (C) then
   S1 := ...
   @S1 := (1)
else
   S2 := ...
   @S2 := (1)
end if
S3 := Φ(S1, @S1, S2, @S2)
@S3 := max(@S1, @S2)
```

Fig. 9.2 After conversion of figure 9.1 to Array SSA form

The full Array SSA form uses Φ operators instead of ϕ functions used by traditional SSA form. The semantics of the Φ operator can be defined as a pure function. This is one respect in which Array SSA form has advantages over traditional SSA form even for scalar variables. @ *variables* (pronounced "at variables") are used to obtain a pure function semantics for the Φ operator. Each ϕ function in traditional SSA form such as $\phi(S_1, S_2)$ is rewritten as $\Phi(S_1, @S_1, S_2, @S_2)$. For each static definition S_k, its @ variable $@S_k$ identifies the most recent *iteration vector* ("timestamp") *at* which S_k was modified by this definition.

The *iteration vector* [**?**, **?**] of a static definition S_k identifies a single iteration in the iteration space of the set of loops that enclose the definition. We do not require that the surrounding loops be structured counted loops or that the surrounding loops be tightly nested. Our only assumption in full Array SSA form is that all loops are single-entry, or equivalently, that the control flow graph is *reducible* [**?**, **?**]. (As we will see in section 9.2.2, this assumption is not necessary for partial Array SSA form.) For single-entry loops, we know that each def executes at most once in a

given iteration of its surrounding loops, hence the iteration vector serves the purpose of a "timestamp".

Let n be the number of loops that enclose a given definition. For convenience, we treat the outermost region of acyclic control flow in a procedure as a dummy outermost loop with a single iteration. Therefore $n \geq 1$ for each definition. A single point in the iteration space is specified by the iteration vector $\mathbf{i} = (i_1, \ldots, i_n)$, which is an n-tuple of iteration numbers one for each enclosing loop. We assume that all @ variables, $@S_k$, are initialized to the empty vector, $@S_k := (\ )$, at the start of program execution. For each real (non-Φ) definition of a renamed scalar, S_k, we assume that a statement of the form $@S_k := \mathbf{i}$ is inserted immediately after definition S_k, where $\mathbf{i}$ is the current iteration vector for all loops that surround S_k. All @ variables are initialized to the empty vector because the empty vector is the identity element for a lexicographic max operation *i.e.,* $\max((\ ), \mathbf{i}) = \mathbf{i}$, for any @ variable value $\mathbf{i}$.

As a simple example, figure 9.2 shows the Array SSA form for the program in figure 9.1. Note that @ variables $@S_1$ and $@S_2$ are explicit arguments of the Φ operator. In this example of acyclic code, there are only two possible values for each @ variable — the empty vector, ( ), and the unit vector[1], (1).

Figure 9.3 shows an example for-loop and its conversion to Array SSA form. Given a Φ operator, $S_3 := \Phi(S_2, @S_2, S_1, @S_1)$, the value of S_3 is given by the following conditional expression (where $\geq$ denotes a lexicographic greater-than-or-equal comparison of iteration vectors):

$$S_3 = \begin{array}{ll} \textbf{if} & @S_2 \geq @S_1 \textbf{ then } S_2 \\ \textbf{else} & S_1 \\ \textbf{end if} & \end{array}$$

Each Φ definition in Array SSA form also has an associated @ variable. Specifically, the statement, $@S_3 := \max(@S_2, @S_1)$, is inserted after the Φ definition, $S_3 := \Phi(S_2, @S_2, S_1, @S_1)$, where max represents a *lexicographic maximum* operation of iteration vector values $@S_2$ and $@S_1$. No initialization is required for an @ variable for a Φ definition because its value is completely determined by other @ variables.

The prior discussion was on full Array SSA form for scalar variables; we now describe full Array SSA form for array variables. Figures 9.4 and 9.5 show an example program with an array variable, and the conversion of the program to Array SSA form as defined in [**?**]. The key differences between Array SSA form for array variables and Array SSA form for scalar variables are as follows:

1. **Renamed array variables:**
   All array variables are renamed so as to satisfy the static single assignment property *i.e.,* each definition of an array element is assigned a unique name.

[1] The astute reader may have observed that the @ variables do not satisfy the static single assignment property because each $@S_k$ variable has two static definitions, one in the initialization and one at the real definition of S_k. However, the initialization def is executed only once at the start of program execution and is treated as a special-case initial value rather than as a separate definition.

Example for-loop:

```
S := ...
for i := 1 to m do
    S := ...
    if (C) then
        S := ...
    end if
end for
```

After conversion to Array SSA form:

```
@S := ( ) ; @S_1 := ( ) ; @S_2 := ( )
S := ...
@S := (1)
for i := 1 to m do
    S_0 := Φ(S_3, @S_3, S, @S)
    @S_0 := max(@S_3, @S)
    S_1 := ...
    @S_1 := (1, i)
    if (C) then
        S_2 := ...
        @S_2 := (1, i)
    end if
    S_3 := Φ(S_2, @S_2, S_1, @S_1)
    @S_3 := max(@S_2, @S_2)
end for
```

Fig. 9.3 A for-loop and its conversion to Array SSA form

```
n1:   A[*] := initial value of A
      i := 1
      C := i < 2
      if C then
n2:       k := 2 * i
          A[k] := i
          print A[k]
      endif
n3:   print A[2]
```

Fig. 9.4 Example program with array variables

```
n1:   @i := ( ) ; @C := ( ) ; @k := ( ) ;
      @A0[*] := ( ) ; @A1[*] := ( )

      A0[*] := initial value of A
      @A0[*] := (1)
      i := 1
      @i := (1)
      C := i < n
      @C := (1)
      if C then
n2:      k := 2 * i
         @k := (1)
         A1[k] := i
         @A1[k] := (1)
         A2 := dΦ(A1, @A1, A0, @A0)
         @A2 := max(@A1, @A0)
         print A2[k]
      endif
n4:   A3 := Φ(A2, @A2, A0, @A0)
      @A3 := max(@A2, @A0)
      print A3[2]
```

Fig. 9.5 Conversion of program in figure 9.4 to Full Array SSA Form

Analogous to traditional scalar SSA form, control Φ operators are introduced to generate new names for merging two or more prior definitions, and to ensure that each use refers to exactly one definition.

2. **Definition Φ's:**
   A *definition* Φ operator is introduced in Array SSA form to deal with preserving ("non-killing") definitions of arrays. Consider A_0 and A_1, two renamed arrays that originated from the same array variable in the source program such that $A_1[k] := \ldots$ is an update of a single array element and A_0 is the prevailing definition at the program point just prior to the definition of A_1. A definition Φ of the form $A_2 := d\Phi(A_1, @A_1, A_0, @A_0)$ is inserted immediately after the definition for A_1 and $@A_1$. (We use the notation $d\Phi$ when we want to distinguish a definition Φ operator from a control Φ operator.) Since definition A_1 only updates one element of A_0, A_2 represents an element-level merge of arrays A_1 and A_0. Definition Φ's did not need to be inserted for definitions of scalar variables because a scalar definition completely kills the old value of the variable.
3. **Array-valued @ variables:**
   One consequence of renaming arrays is that each (renamed) array variable, A_j, in Array SSA form has an associated @ variable, $@A_j$, such that $@A_j$ has the same shape (rank and dimension sizes) as array variable A_j. Each update of a single array element of the form $A_j[k] := \ldots$, is followed by the statement of the form $@A_j[k] := \mathbf{i}$ where $\mathbf{i}$ is the iteration vector for the loops surrounding the

definition of A_j. Thus, an array-valued @ variable, $@A_j$, can record a separate iteration vector for each element that is assigned by definition A_j.

4. **Array-valued Φ operators:**
   Another consequence of renaming arrays is that a Φ operator for array variables must also return an array value. Consider a (control or definition) Φ operator in a statement of the form, $A_2 := \Phi(A_1, @A_1, A_0, @A_0)$. Its semantics can be specified exactly by the following conditional expression for each element, $A_2[j]$, in the result array A_2:

$$A_2[j] = \begin{array}{ll} \textbf{if} & @A_1[j] \geq @A_0[j] \textbf{ then } A_1[j] \\ \textbf{else} & A_0[j] \\ \textbf{end if} & \end{array}$$

   The key extension over the scalar case is that the conditional expression specifies an element-level merge of arrays A_1 and A_0.

9.2.2 Partial Array SSA Form

The previous section described full Array SSA form with @ variables and Φ operators that can be evaluated at run-time. This section introduces *partial Array SSA form* as a representation for static analysis. Partial Array SSA form is a compile-time approximation of full Array SSA form that provides conservative use-def information for each static read access of an array element.

Consider a statement of the form, $A_2 := \Phi(A_1, @A_1, A_0, @A_0)$, that contains a Φ operator. A static analysis will need to approximate the computation of this Φ operator by some data flow transfer function, $\mathcal{L}_\Phi$. The inputs and output of $\mathcal{L}_\Phi$ will be *lattice elements* for scalar/array variables that are compile-time approximations of their run-time values. We use the notation $\mathcal{L}(V)$ to denote the lattice element for a scalar or array variable V. Therefore, the statement, $A_2 := \Phi(A_1, @A_1, A_0, @A_0)$, will (in general) be modeled by the data flow equation $\mathcal{L}(A_2) = \mathcal{L}_\Phi(\mathcal{L}(A_1), \mathcal{L}(@A_1), \mathcal{L}(A_0), \mathcal{L}(@A_0))$.

Our first observation is that there is no extra information provided at compile-time by the @ variables for any static analysis that does not distinguish between reachable code and unreachable code. In such cases, it is sufficient to model $A_2 := \Phi(A_1, @A_1, A_0, @A_0)$ by a data flow equation of the form $\mathcal{L}(A_2) = \mathcal{L}_\phi(\mathcal{L}(A_1), \mathcal{L}(A_0))$ that does not use lattice variables $\mathcal{L}(@A_1)$ and $\mathcal{L}(@A_0)$. For array variables, the only useful information provided by an @ variable, $@A_1$ (say), at compile-time is an indication of which elements were updated by the assignment to array A_1. However, as we will see in section 9.3, this information is also included in the lattice value $\mathcal{L}(A_1)$ for array A_1.

Our second observation is that a static analysis that needs to distinguish between unreachable code and reachable code can do so efficiently by introducing *executable flags* for nodes and edges in the CFG (Control Flow Graph) as in [**?**]. If executable

flags are computed in the data flow analysis, then the @ variables again do not provide any useful extra information. In fact, if we consider a control Φ statement $A_2 := \Phi(A_1, @A_1, A_0, @A_0)$, with array values A_1 and A_0 carried by incoming CFG edges $e1$ and $e0$ respectively, then the corresponding data flow equation (in the presence of unreachable code elimination) will be $\mathcal{L}(A_2) = \mathcal{L}_\Phi(\mathcal{L}(A_1), X_{e1}, \mathcal{L}(A_0), X_{e0})$ where X_{e1} and X_{e0} are the executable flags for edges $e1$ and $e0$. (Full details can be found in [**?**].)

Since @ variables need not be modeled for the compile-time analyses discussed in this chapter, we drop them and use the ϕ operator, $A_2 := \phi(A_1, A_0)$ instead of $A_2 := \Phi(A_1, @A_1, A_0, @A_0)$ in *partial* Array SSA form. A consequence of dropping @ variables is that partial Array SSA form does not need to deal with iteration vectors, and therefore does not require the control flow graph to be *reducible* as in full Array SSA form. The use of ϕ operators without @ variables brings partial Array SSA form closer to traditional SSA form. The key difference is that partial Array SSA form still contains renamed arrays and definition ϕ operators for updates to array elements.

9.3 Array Lattice

In this section, we describe the lattice representation used to model array values for the analyses in this chapter. Constant propagation for scalar variables has historically been performed by efficient dataflow analysis or abstract interpretation techniques in which values of variables are modeled as *lattice elements* [**?**, **?**]. Given a scalar variable v, the usual approach is to allow the value of its lattice element $\mathcal{L}(v)$ to be $\top$, *Constant* or $\bot$. When $\mathcal{L}(v)$ is *Constant*, the lattice element also contains the value of the constant[2].

Formally, a lattice used for scalar constant propagation consists of:

1. A set of lattice elements: a lattice element for a program variable v is written as $\mathcal{L}(v)$, and denotes $\textsc{set}(\mathcal{L}(v))$ = a set of possible values for variable v.
2. $\top$ ("top") and $\bot$ ("bottom"), two distinguished elements of $\mathcal{L}$. The sets denoted by these lattice elements are $\textsc{set}(\top) = \{\ \}$ (the empty set), and $\textsc{set}(\bot) = \mathcal{U}^v$, where $\mathcal{U}^v$ is the universal set of values for variable v.
3. If $\mathcal{L}(v)$ is a *Constant* lattice element $\textsc{set}(\mathcal{L}(v)) = \{Constant\}$, the singleton set containing a constant.
4. A *join* operator, $\sqcap$, such that for any lattice element e, $e \sqcap \top = e$ and $e \sqcap \bot = \bot$. The $\sqcap$ operator on lattice elements corresponds to the set *union* operation on the sets denoted by lattice elements *i.e.,* if e and f are two lattice elements, then $\textsc{set}(e \sqcap f) = \textsc{set}(e) \cup \textsc{set}(f)$.

[2] An extension that is sometimes employed is to also include lattice elements that can represent a small set of constants or a range of constants[**?**]. This functionality is not addressed in our chapter, but would be a straightforward extension to our framework.

It follows that the $\sqcap$ operator is idempotent, commutative, and associative, and that the lattice is *complete i.e.,* $\sqcap$ is closed on $\mathcal{L}$.

5. A $\sqsupseteq$ operator such that $e \sqsupseteq f$ if and only if $e \sqcap f = f$, and a $\sqsupset$ operator such that $e \sqsupset f$ if and only if $e \sqsupseteq f$ and $e \neq f$.
   The $\sqsupseteq$ and $\sqsupset$ operators on lattice elements correspond to the *inclusive subset* and *proper subset* operations on the sets denoted by lattice elements *i.e.,* $e \sqsupseteq f$ if and only if $\textsc{set}(e) \subseteq \textsc{set}(f)$, and $e \sqsupset f$ if and only if $\textsc{set}(e) \subset \textsc{set}(f)$.
   The $\sqsupset$ operator defines a partial order on lattice elements. If $e \sqsupset f$, we say that e is "above" f and that f is "below" e in the lattice. Hence, $\top$ is above all other lattice elements and $\bot$ is below all other lattice elements.

The *height* H of lattice $\mathcal{L}$ is the length of the largest sequence of lattice elements $e_1, e_2, \ldots, e_H$ such that $e_i \sqsupset e_{i+1}$ for all $1 \leq i < H$. The height of the lattice for scalar constant propagation is 3.

We now describe how lattice elements for array variables are represented in our framework for constant propagation. Let $\mathcal{U}^A_{ind}$ and $\mathcal{U}^A_{elem}$ be the universal set of *index values* and the universal set of array *element values* respectively for an array variable A in Array SSA form. For an array variable, the set denoted by lattice element $\mathcal{L}(A)$ is a subset of $\mathcal{U}^A_{ind} \times \mathcal{U}^A_{elem}$ *i.e.,* a set of index-element pairs. There are three kinds of lattice elements for array variables that are of interest in our framework:

1. $\mathcal{L}(A) = \top \quad \Rightarrow \quad \textsc{set}(\mathcal{L}(A)) = \{\ \}$
   This "top" case indicates that the set of possible index-element pairs that have been identified thus far for A is the empty set, $\{\ \}$.
2. $\mathcal{L}(A) = \langle (i_1, e_1), (i_2, e_2), \ldots \rangle$
   $\Rightarrow \quad \textsc{set}(\mathcal{L}(A)) = \{(i_1, e_1), (i_2, e_2), \ldots\} \ \cup \ (\mathcal{U}^A_{ind} - \{i_1, i_2, \ldots\}) \times \mathcal{U}^A_{elem}$
   In general, the lattice element for this "constant" case is represented by a finite list of index-element pairs, $\langle (i_1, e_1), (i_2, e_2), \ldots \rangle$ where $i_1, i_2, \ldots$ are constant index values, and $e_1, e_2, \ldots$ are constant element values. The constant indices, $i_1, i_2, \ldots$, must represent distinct (non-equal) index values. As in the scalar case, the lattice ordering ($\sqsupset$) for these elements is determined by the subset relationship among the sets that they denote.
   The meaning of this "constant" lattice element is that the current stage of analysis has identified some finite number of constant index-element pairs for array variable A, such that $A[i_1] = e_1$, $A[i_2] = e_2$, etc. All other elements of A are assumed to be non-constant. (Extensions to handle non-constant indices are described in section 9.5.)
3. $\mathcal{L}(A) = \bot \quad \Rightarrow \quad \textsc{set}(\mathcal{L}(A)) = \mathcal{U}^A_{ind} \times \mathcal{U}^A_{elem}$
   This "bottom" case indicates that, according to the approximation in the current stage of analysis, array A may take on any value from the universal set of index-element pairs. Note that $\mathcal{L}(A) = \bot$ is equivalent to an empty list, $\mathcal{L}(A) = \langle\ \rangle$, in case (2) above; they both denote the universal set of index-element pairs.

Regardless of the size of an array, A, the number of index-element pairs in $\mathcal{L}(A)$ is bounded by the number of static assignments to A in the source. For the sake of efficiency, we will further restrict the constant array lattice elements, $\mathcal{L}(A) = \langle (i_1, e_1), (i_2, e_2), \ldots \rangle$, to lists that are bounded in size by some constant,

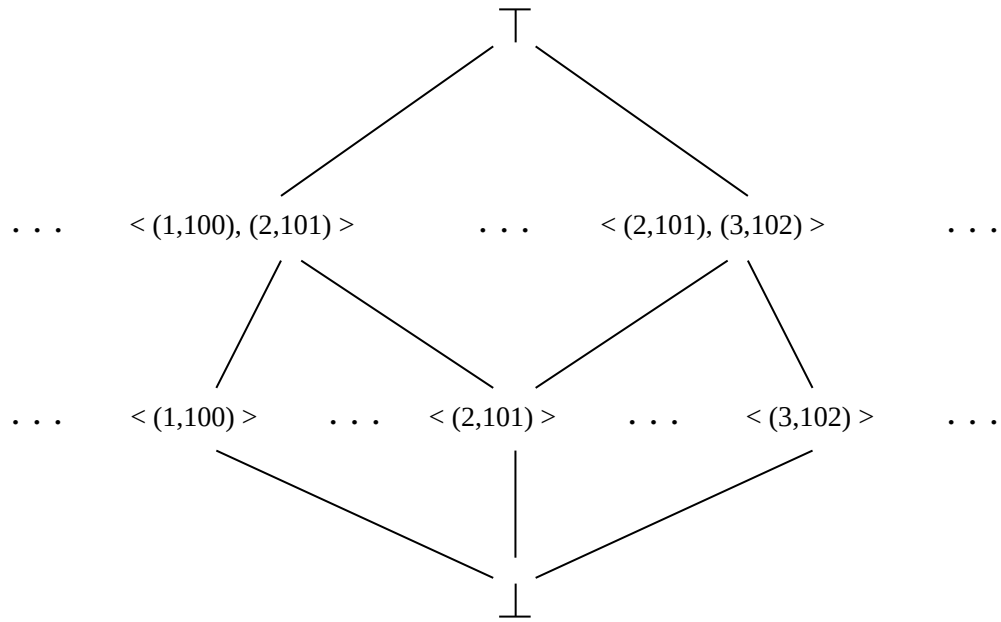


Fig. 9.6 Lattice elements of array values with maximum list size $Z = 2$

$Z \geq 1$. The lattice structure for the $Z = 2$ case is shown in figure 9.6. This lattice has four levels. The second level (just below $\top$) contains all possible lists that contain exactly two constant index-element pairs. The third level (just above $\bot$) contains all possible lists that contain a single constant index-element pair. As mentioned earlier, the lattice ordering is determined by the subset relationship among the sets denoted by the lattice elements. For example, consider two lattice elements $\mathcal{L}_1 = \langle(1,100),(2,101)\rangle$ and $\mathcal{L}_2 = \langle(2,101)\rangle$. The sets denoted by these lattice elements are:

$$\textsc{set}(\mathcal{L}_1) = \{(1,100),(2,101)\} \cup (\mathcal{U}_{ind} - \{1,2\}) \times \mathcal{U}_{elem}$$
$$\textsc{set}(\mathcal{L}_2) = \{(2,101)\} \cup (\mathcal{U}_{ind} - \{2\}) \times \mathcal{U}_{elem}$$

Therefore, $\textsc{set}(\mathcal{L}_1)$ is a proper subset of $\textsc{set}(\mathcal{L}_2)$ and we have $\mathcal{L}_1 \sqsupset \mathcal{L}_2$ *i.e.,* $\mathcal{L}_1$ is above $\mathcal{L}_2$ in the lattice in figure 9.6.

We now describe how array lattice elements are computed for various operations that appear in Array SSA form. We start with the simplest operation *viz.,* a reference (read access) to an array element. Figure 9.7 shows how $\mathcal{L}(A_1[k])$, the lattice element for array reference $A_1[k]$, is computed as a function of $\mathcal{L}(A_1)$ and $\mathcal{L}(k)$, the lattice elements for A_1 and k. We denote this function by $\mathcal{L}_{[\,]}$ *i.e.,* $\mathcal{L}(A_1[k]) = \mathcal{L}_{[\,]}(\mathcal{L}(A_1), \mathcal{L}(k))$. The interesting case in figure 9.7 occurs in the middle cell when neither $\mathcal{L}(A_1)$ nor $\mathcal{L}(k)$ is $\top$ or $\bot$. In this case, $\mathcal{L}(k) = Constant$ and $\mathcal{L}(A_1)$ is a nonempty list of the form, $\langle(i_1, e_1), \ldots\rangle$. For this case, if there exists an index-element pair (i_j, e_j) in $\mathcal{L}(A_1)$ such that i_j is the same as the constant $\mathcal{L}(k)$,

$\mathcal{L}(A_1[k])$	$\mathcal{L}(k) = \top$	$\mathcal{L}(k) = Constant$	$\mathcal{L}(k) = \bot$
$\mathcal{L}(A_1) = \top$	$\top$	$\top$	$\bot$
$\mathcal{L}(A_1) = \langle(i_1, e_1), \ldots\rangle$	$\top$	e_j, if $\exists\ (i_j, e_j) \in \mathcal{L}(A_1)$ with $\mathcal{DS}(i_j, \mathcal{L}(k)) = true$ $\bot$, otherwise	$\bot$
$\mathcal{L}(A_1) = \bot$	$\bot$	$\bot$	$\bot$

Fig. 9.7 Lattice computation for $\mathcal{L}(A_1[k]) = \mathcal{L}_{[\,]}(\mathcal{L}(A_1), \mathcal{L}(k))$, where $A_1[k]$ is an array element read operator

$\mathcal{L}(A_1)$	$\mathcal{L}(i) = \top$	$\mathcal{L}(i) = Constant$	$\mathcal{L}(i) = \bot$
$\mathcal{L}(k) = \top$	$\top$	$\top$	$\bot$
$\mathcal{L}(k) = Constant$	$\top$	$\langle(\mathcal{L}(k), \mathcal{L}(i))\rangle$	$\bot$
$\mathcal{L}(k) = \bot$	$\bot$	$\bot$	$\bot$

Fig. 9.8 Lattice computation for $\mathcal{L}(A_1) = \mathcal{L}_{d[\,]}(\mathcal{L}(k), \mathcal{L}(i))$, where $A_1[k] := i$ is an array element write operator

then the value returned for $\mathcal{L}(A_1[k])$ is the constant e_j. Otherwise, $\bot$ is returned as the value for $\mathcal{L}(A_1[k])$.

The notation $\mathcal{DS}$ in the middle cell in figure 9.7 represents a "definitely-same" binary relation *i.e.,* $\mathcal{DS}(a, b) = true$ if and only if a and b are known to have exactly the same value. If, as in the current discussion, a and b are constants then $\mathcal{DS}(a, b) = true$ if and only if $a = b$. However, we use the $\mathcal{DS}$ notation for generality because later in section 9.5 we show how the lattice modeling of arrays introduced in this section can be extended to symbolic index values.

Next, consider a definition (write access) of an array element, which in general has the form $A_1[k] := i$. Figure 9.8 shows how $\mathcal{L}(A_1)$, the lattice element for the array being written into, is computed as a function of $\mathcal{L}(k)$ and $\mathcal{L}(i)$, the lattice elements for k and i. We denote this function by $\mathcal{L}_{d[\,]}$ *i.e.,* $\mathcal{L}(A_1) = \mathcal{L}_{d[\,]}(\mathcal{L}(k), \mathcal{L}(i))$. As before, the interesting case in figure 9.8 occurs in the middle cell when both $\mathcal{L}(k)$ and $\mathcal{L}(i)$ are constant. For this case, the value returned for $\mathcal{L}(A_1)$ is simply the singleton list, $\langle\ (\mathcal{L}(k), \mathcal{L}(i))\ \rangle$, which contains exactly one constant index-element pair.

Now, we turn our attention to the ϕ functions. Consider a definition ϕ operation of the form, $A_2 := d\phi(A_1, A_0)$. The lattice computation for $\mathcal{L}(A_2) = \mathcal{L}_{d\phi}(\mathcal{L}(A_1), \mathcal{L}(A_0))$ is shown in figure 9.9. Since A_1 corresponds to a definition of a single array element, the list for $\mathcal{L}(A_1)$ can contain at most one pair (see figure 9.8). Therefore, the three cases considered for $\mathcal{L}(A_1)$ in figure 9.9 are $\mathcal{L}(A_1) = \top$, $\mathcal{L}(A_1) = \langle(i', e')\rangle$, and $\mathcal{L}(A_1) = \bot$.

The notation $\textsc{update}((i', e'), \langle(i_1, e_1), \ldots\rangle)$ used in the middle cell in figure 9.9 denotes a special update of the list $\mathcal{L}(A_0) = \langle(i_1, e_1), \ldots\rangle$ with respect to the constant index-element pair (i', e'). UPDATE involves four steps:

1. Compute the list $T = \{ (i_j, e_j) \mid (i_j, e_j) \in \mathcal{L}(A_0) \text{ and } \mathcal{DD}(i', i_j) = \textit{true} \}$. List T contains only those pairs from $\mathcal{L}(A_0)$ that have an index value i_j that is *definitely different* from i'.
   Analogous to $\mathcal{DS}$, $\mathcal{DD}$ denotes a "definitely-different" binary relation *i.e.,* $\mathcal{DS}(a, b) = \textit{true}$ if and only if a and b are known to have distinct (non-equal) values.
2. Insert the pair (i', e') into T to obtain a new list, I.
3. If the size of list I exceeds the threshold size Z, then one of the pairs in I is dropped from the output list so as to satisfy the size constraint. (Since the size of $\mathcal{L}(A_0)$ must have been $\leq Z$, it is sufficient to drop only one pair to satisfy the size constraint.)
4. Return I as the value of $\textsc{update}((i', e'), \langle (i_1, e_1) , \ldots \rangle)$.

$\mathcal{L}(A_2)$	$\mathcal{L}(A_0) = \top$	$\mathcal{L}(A_0) = \langle (i_1, e_1), \ldots \rangle$	$\mathcal{L}(A_0) = \bot$
$\mathcal{L}(A_1) = \top$	$\top$	$\top$	$\top$
$\mathcal{L}(A_1) = \langle (i', e') \rangle$	$\top$	$\textsc{update}((i', e'), \langle (i_1, e_1), \ldots \rangle)$	$\langle (i', e') \rangle$
$\mathcal{L}(A_1) = \bot$	$\bot$	$\bot$	$\bot$

Fig. 9.9 Lattice computation for $\mathcal{L}(A_2) = \mathcal{L}_{d\phi}(\mathcal{L}(A_1), \mathcal{L}(A_0))$ where $A_2 := d\phi(A_1, A_0)$ is a definition ϕ operation

$\mathcal{L}(A_2) = \mathcal{L}(A_1) \sqcap \mathcal{L}(A_0)$	$\mathcal{L}(A_0) = \top$	$\mathcal{L}(A_0) = \langle (i_1, e_1), \ldots \rangle$	$\mathcal{L}(A_0) = \bot$
$\mathcal{L}(A_1) = \top$	$\top$	$\mathcal{L}(A_0)$	$\bot$
$\mathcal{L}(A_1) = \langle (i'_1, e'_1), \ldots \rangle$	$\mathcal{L}(A_1)$	$\mathcal{L}(A_1) \cap \mathcal{L}(A_0)$	$\bot$
$\mathcal{L}(A_1) = \bot$	$\bot$	$\bot$	$\bot$

Fig. 9.10 Lattice computation for $\mathcal{L}(A_2) = \mathcal{L}_{\phi}(\mathcal{L}(A_1), \mathcal{L}(A_0)) = \mathcal{L}(A_1) \sqcap \mathcal{L}(A_0)$, where $A_2 := \phi(A_1, A_0)$ is a control ϕ operation

Finally, consider a control ϕ operation that merges two array values, $A_2 := \phi(A_1, A_0)$. The join operator ($\sqcap$) is used to compute $\mathcal{L}(A_2)$, the lattice element for A_2, as a function of $\mathcal{L}(A_1)$ and $\mathcal{L}(A_0)$, the lattice elements for A_1 and A_0 *i.e.,* $\mathcal{L}(A_2) = \mathcal{L}_{\phi}(\mathcal{L}(A_1), \mathcal{L}(A_0)) = \mathcal{L}(A_1) \sqcap \mathcal{L}(A_0)$. The rules for computing this join operator are shown in figure 9.10, depending on different cases for $\mathcal{L}(A_1)$ and $\mathcal{L}(A_0)$. The notation $\mathcal{L}(A_1) \cap \mathcal{L}(A_0)$ used in the middle cell in figure 9.10 denotes a simple intersection of lists $\mathcal{L}(A_1)$ and $\mathcal{L}(A_0)$ — the result is a list of pairs that appear in both $\mathcal{L}(A_1)$ and $\mathcal{L}(A_0)$.

We conclude this section by discussing the example program in figure 9.11. The partial Array SSA form for this example is shown in figure 9.12, and the data flow equations for this example are shown in figure 9.13. Each assignment statement in the partial Array SSA form (in figure 9.12) results in one data flow equation (in figure 9.13); the numbering S1 through S8 indicates the correspondence. In general, each data flow equation in our framework has a single lattice variable on its LHS

(Left Hand Side). Also, a lattice variable will only appear on the LHS of at most one equation.

The lattice operations ($\mathcal{L}_{\phi}$, $\mathcal{L}_{d\phi}$, $\mathcal{L}_{[\ ]}$, $\mathcal{L}_{d[\ ]}$, $\mathcal{L}_{*}$) in figure 9.13 depend on the operations within the corresponding statements. For example, there are reads of array elements in the RHS of statements S3 and S5, writes of array elements in the LHS of statements S3 and S5, definition ϕ operators in statements S4 and S6, and a control ϕ operator in statement S7. We also incorporate lattice computations for specific arithmetic operators such as lattice function L_* for the multiply operator in statements S3 and S5. Tables for lattice computations such as L_* are straightforward and are not shown.

```
Y[3] := 99
if C then
    D[1] := Y[3] * 2
else
    D[1] := Y[I] * 2
endif
Z := D[1]
```

Fig. 9.11 Sparse Constant Propagation Example

```
      Y0 and D0 in effect here.
      ...
S1:   Y1[3] := 99
S2:   Y2 := dφ(Y1, Y0)
      if C then
S3:     D1[1] := Y2[3] * 2
S4:     D2 := dφ(D1, D0)
      else
S5:     D3[1] := Y2[I] * 2
S6:     D4 := dφ(D3, D0)
      endif
S7:   D5 := φ(D2, D4)
S8:   Z := D5[1]
```

Fig. 9.12 Array SSA form for the Sparse Constant Propagation Example

$$
\begin{array}{ll}
\text{S1:} & \mathcal{L}(Y_1) = <(3,99)> \\
\text{S2:} & \mathcal{L}(Y_2) = \mathcal{L}_{d\phi}(\mathcal{L}(Y_1), \mathcal{L}(Y_0)) \\
\text{S3:} & \mathcal{L}(D_1) = \mathcal{L}_{d[\ ]}(\mathcal{L}_{*}(\mathcal{L}_{[\ ]}(\mathcal{L}(Y_2), 3)), 2)) \\
\text{S4:} & \mathcal{L}(D_2) = \mathcal{L}_{d\phi}(\mathcal{L}(D_1), \mathcal{L}(D_0)) \\
\text{S5:} & \mathcal{L}(D_3) = \mathcal{L}_{d[\ ]}(\mathcal{L}_{*}(\mathcal{L}_{[\ ]}(\mathcal{L}(Y_2), \mathcal{L}(I))), 2)) \\
\text{S6:} & \mathcal{L}(D_4) = \mathcal{L}_{d\phi}(\mathcal{L}(D_3), \mathcal{L}(D_0)) \\
\text{S7:} & \mathcal{L}(D_5) = \mathcal{L}_{\phi}(\mathcal{L}(D_2), \mathcal{L}(D_4)) \\
\text{S8:} & \mathcal{L}(Z) = \mathcal{L}_{[\ ]}(\mathcal{L}(D_5), 1)
\end{array}
$$

Fig. 9.13 Data Flow Equations for the Sparse Constant Propagation Example

9.4 Sparse Constant Propagation for Scalars and Array Elements

This section shows how we extend the Sparse Constant propagation (SC) algorithm from [**?**] so as to enable constant propagation through array elements. For simplicity, this algorithm is restricted to cases in which both the subscript and the value of an array definition are constant. A generalization to non-constant subscripts is presented next in section 9.5.

Figure 9.14 contains an outline of the sparse constant propagation algorithm for scalar and array variables. It is similar in structure to the sparse constant propagation algorithm for scalars presented in [**?**]. A major difference is that the data flow equations now include the lattice values for array variables, as described in section 9.3. In addition, the algorithm in figure 9.14 uses a worklist of data flow equations (rather than a worklist of SSA edges as in [**?**]), thus enabling future integration with other analysis algorithms that are based on data flow equations.

Let us consider how the algorithm in figure 9.14 will work on the data flow equations in figure 9.13. Recall that the Array SSA form for this example program (see figure 9.12) includes the propagation of array Y_2 into two control flow successors, and the propagation into array D_5 from two control flow predecessors.

The initialization step first initializes $\mathcal{L}(Y_0)$ and $\mathcal{L}(D_0)$ to $\perp$, since they are both considered to be unknown variables in this example program. Next, *worklist* is intialized to contain equations S1, S2, S3, S4, S5, S6, S8, since they all contain some terms that are different from $\top$ *i.e.,* terms that contain a constant or equal $\perp$.

For the fixpoint iteration step, the heuristic in figure 9.14 causes the equations in *worklist* to be processed in their textual order *i.e.,* S1, S2, S3, ... (This heuristic is similar to the heuristic of processing basic blocks in a topological order of their positions in the control flow graph.) The iteration terminates when no LHS lattice variable changes its value, thus causing *worklist* to become empty. The final values of the lattice variables obtained after the fixpoint iteration step has completed are shown in figure 9.15.

The above lattice values were obtained assuming $\mathcal{L}(I) = \perp$. If, instead, variable I is known to equal 3 *i.e.,* $\mathcal{L}(I) = 3$, then the lattice variables that would be obtained after the fixpoint iteration step has completed are shown in figure 9.16.

```
/* INITIALIZATION */

   for each lattice variable ℒ(v) do
      if it is not possible that ℒ(v) can be recognized as a constant
            at compile-time (e.g., v is a return value from an unknown call)
      then
         ℒ(v) ← ⊥
      else
         ℒ(v) ← ⊤
      end if
   end for

   Intialize worklist ← an empty list
   for each equation E do
      if the RHS of equation E has at least one term that is ≠ ⊤
            (i.e., at least one term that is ⊥ or contains a constant)
      then
         Insert equation E into the worklist
      end if
   end for

/* FIXPOINT ITERATION */

   while worklist is not empty do
      E ← remove any equation from worklist
      /* Heuristic: remove an equation that depends on the smallest number
       of other equations in worklist */
      Recompute ℒ(v), the LHS of equation E, based on the values of E's RHS terms
      if the LHS of equation E has changed then
         for each equation E′ that uses ℒ(v) in its RHS do
            Insert equation E′ into worklist
         end for
      end if
   end while

/* TERMINATION */

   For each Array SSA variable v such that ℒ(v) contains a constant, transform the
   program to replace each use of v by a constant, if profitable to do so
```

Fig. 9.14 Algorithm for Sparse Constant propagation (SC) for scalar and array variables

S1: $\mathcal{L}(Y_1) = \langle\ (3, 99)\ \rangle$
S2: $\mathcal{L}(Y_2) = \langle\ (3, 99)\ \rangle$
S3: $\mathcal{L}(D_1) = \langle\ (1, 198)\ \rangle$
S4: $\mathcal{L}(D_2) = \langle\ (1, 198)\ \rangle$
S5: $\mathcal{L}(D_3) = \bot$
S6: $\mathcal{L}(D_4) = \bot$
S7: $\mathcal{L}(D_5) = \bot$
S8: $\mathcal{L}(Z) = \bot$

Fig. 9.15 Solution to data flow equations from figure 9.13, assuming I is unknown

S1: $\mathcal{L}(Y_1) = \langle\ (3, 99)\ \rangle$
S2: $\mathcal{L}(Y_2) = \langle\ (3, 99)\ \rangle$
S3: $\mathcal{L}(D_1) = \langle\ (1, 198)\ \rangle$
S4: $\mathcal{L}(D_2) = \langle\ (1, 198)\ \rangle$
S5: $\mathcal{L}(D_3) = \langle\ (1, 198)\ \rangle$
S6: $\mathcal{L}(D_4) = \langle\ (1, 198)\ \rangle$
S7: $\mathcal{L}(D_5) = \langle\ (1, 198)\ \rangle$
S8: $\mathcal{L}(Z) = 198$

Fig. 9.16 Solution to data flow equations from figure 9.13, assuming I is known to be = 3

In either case ($\mathcal{L}(I) = \perp$ or $\mathcal{L}(I) = 3$), the resulting constants revealed by the algorithm can be used in whatever analyses or transformations the compiler considers to be profitable to perform. This is the job of the termination step in figure 9.14.

9.5 Beyond Constant Indices

```
k := 2
do i := ...
    ...
    a[i] := k * 5
    ... := a[i]
enddo
```

Fig. 9.17 Example of Constant Propagation through Non-constant Index

In this section we address constant propagation through *non-constant array subscripts*, as a generalization of the algorithm for constant subscripts described in section 9.4. As an example, consider the program fragment in figure 9.17. In the loop in figure 9.17, we see that the read access of $a[i]$ will have a constant value ($k * 5 = 10$), even though the index/subscript value i is not a constant. We would like to extend the framework from sections 9.3 and 9.4 to be able to recognize the read of $a[i]$ as constant in such programs. There are two key extensions that need to be considered for non-constant (symbolic) subscript values:

- For constants, C_1 and C_2, $\mathcal{DS}(C_1, C_2) \neq \mathcal{DD}(C_1, C_2)$. However, for two symbols, S_1 and S_2, it is possible that both $\mathcal{DS}(S_1, S_2)$ and $\mathcal{DD}(S_1, S_2)$ are FALSE, that is, we don't know if they are the same or different.

- For constants, C_1 and C_2, the values for $\mathcal{DS}(C_1, C_2)$ and $\mathcal{DD}(C_1, C_2)$ can be computed by inspection. For symbolic indices, however, some program analysis is necessary to compute the $\mathcal{DS}$ and $\mathcal{DD}$ relations.

We now discuss the compile-time computation of $\mathcal{DS}$ and $\mathcal{DD}$ for symbolic indices. Observe that, given index values I_1 and I_2, only one of the following three cases is possible:

```
Case 1: DS(I1, I2) = FALSE; DD(I1, I2) = FALSE
Case 2: DS(I1, I2) = TRUE; DD(I1, I2) = FALSE
Case 3: DS(I1, I2) = FALSE; DD(I1, I2) = TRUE
```

The first case is the most conservative solution. In the absence of any other knowledge, it is always correct to state that $\mathcal{DS}(I_1, I_2) = \mathit{false}$ and $\mathcal{DD}(I_1, I_2) = \mathit{false}$.

The problem of determining if two symbolic index values are the same is equivalent to the classical problem of *global value numbering* [**?**, **?**]. If two indices i and j have the same value number, then $\mathcal{DS}(i, j)$ must $= \mathit{true}$. The problem of computing $\mathcal{DD}$ is more complex. Note that $\mathcal{DD}$, unlike $\mathcal{DS}$, is not an equivalence relation because $\mathcal{DD}$ is not transitive. If $\mathcal{DD}(A, B) = \mathit{true}$ and $\mathcal{DD}(B, C) = \mathit{true}$, it does not imply that $\mathcal{DD}(A, C) = \mathit{true}$. However, we can leverage past work on array dependence analysis [**?**] to identify cases for which $\mathcal{DD}$ can be evaluated to *true*. For example, it is clear that $\mathcal{DD}(i, i + 1) = \mathit{true}$, and that $\mathcal{DD}(i, 0) = \mathit{true}$ if i is a loop index variable that is known to be ≥ 1.

Let us consider how the $\mathcal{DS}$ and $\mathcal{DD}$ relations for symbolic index values are used by our constant propagation algorithms. Note that the specification of how $\mathcal{DS}$ and $\mathcal{DD}$ are used is a separate issue from teh precision of the $\mathcal{DS}$ and $\mathcal{DD}$ values. We now describe how the lattice and the lattice operations presented in section 9.3 can be extended to deal with non-constant subscripts.

First, consider the lattice itself. The $\top$ and $\bot$ lattice elements retain the same meaning as in section 9.3 *viz.,* $\textsc{set}(\top) = \{\ \}$ and $\textsc{set}(\bot) = \mathcal{U}^A_{ind} \times \mathcal{U}^A_{elem}$. Each element in the lattice is a list of index-value pairs where the value is still required to be constant but the index may be symbolic — the index is represented by its value number.

We now revisit the processing of an array element read of $A_1[k]$ and the processing of an array element write of $A_1[k]$. These operations were presented in section 9.3 (figures 9.7 and 9.8) for constant indices. The versions for non-constant indices appear in figure 9.18 and figure 9.19. For the read operation in figure 9.18, if there exists a pair (i_j, e_j) such that $\mathcal{DS}(i_j, \textsc{valnum}(k)) = \mathit{true}$ (*i.e.,* i_j and k have the same value number), then the result is e_j. Otherwise, the result is $\top$ or $\bot$ as specified in figure 9.18. For the write operation in figure 9.19, if the value of the right-hand-side, i, is a constant, the result is the singleton list $\langle(\textsc{valnum}(k), \mathcal{L}(i))\rangle$. Otherwise, the result is $\top$ or $\bot$ as specified in figure 9.19.

Let us now consider the propagation of lattice values through $d\phi$ operators. The only extension required relative to figure 9.9 is that the $\mathcal{DD}$ relation used in performing the $\textsc{update}$ operation should be able to determine when $\mathcal{DD}(i', i_j) = \mathit{true}$ if

$\mathcal{L}(A_1[k])$	$\mathcal{L}(k) = \top$	$\mathcal{L}(k) = \textsc{valnum}(k)$	$\mathcal{L}(k) = \bot$
$\mathcal{L}(A_1) = \top$	$\top$	$\top$	$\bot$
$\mathcal{L}(A_1) = \langle(i_1, e_1), \ldots\rangle$	$\top$	e_j, if $\exists\,(i_j, e_j) \in \mathcal{L}(A_1)$ with $\mathcal{DS}(i_j, \textsc{valnum}(k)) = \mathit{true}$ $\bot$, otherwise	$\bot$
$\mathcal{L}(A_1) = \bot$	$\bot$	$\bot$	$\bot$

Fig. 9.18 Lattice computation for $\mathcal{L}(A_1[k]) = \mathcal{L}_{[\,]}(\mathcal{L}(A_1), \mathcal{L}(k))$, where $A_1[k]$ is an array element read operator. If $\mathcal{L}(k) = \textsc{valnum}(k)$, the lattice value of index k is a value number that represents a constant or a symbolic value.

$\mathcal{L}(A_1)$	$\mathcal{L}(i) = \top$	$\mathcal{L}(i) = \mathit{Constant}$	$\mathcal{L}(i) = \bot$
$\mathcal{L}(k) = \top$	$\top$	$\top$	$\bot$
$\mathcal{L}(k) = \textsc{valnum}(k)$	$\top$	$\langle(\textsc{valnum}(k), \mathcal{L}(i))\rangle$	$\bot$
$\mathcal{L}(k) = \bot$	$\bot$	$\bot$	$\bot$

Fig. 9.19 Lattice computation for $\mathcal{L}(A_1) = \mathcal{L}_{d[\,]}(\mathcal{L}(k), \mathcal{L}(i))$, where $A_1[k] := i$ is an array element write operator. If $\mathcal{L}(k) = \textsc{valnum}(k)$, the lattice value of index k is a value number that represents a constant or a symbolic value.

i' and i_j are symbolic value numbers rather than constants. (If no symbolic information is available for i' and i_j, then it is always safe to return $\mathcal{DD}(i', i_j) = \mathit{false}$.)

Note that the UPDATE operation (section 9.3, page 99) always returns a non-empty list, even if it uses the most conservative $\mathcal{DD} = false$ approach. Further, the list will always contain the pair (i', e'), so long as the heuristic for dropping a pair to obey the $\leq Z$ size limit chooses a pair other than (i', e') to drop. This is sufficient to handle the propagation of the constant through the symbolic index in the example in figure 9.17. More precise computations of the $\mathcal{DD}$ relation will lead to more precise (longer) lists, and hence lead to discovery of more constants for more complicated programs with symbolic subscripts.

9.6 Extension to Object References: Redundant Load and Dead Store Elimination in Strongly Typed Languages

In this section, we present a simple, unified approach for the analysis and optimization of object field and array element accesses in strongly typed languages, that works in the presence of object references/pointers. We show how SSA-based program analyses developed for scalars and arrays can be extended to operate on object references in a strongly typed language like Java. This extension models object references as indices into hypothetical *heap arrays*. We then present two new sparse analysis algorithms using the heap array representation; one identifies redundant loads, and the other identifies dead stores. Using strong typing to help disambiguation, these algorithms are more efficient than equivalent analyses for weakly typed

languages. Using the results of these algorithms, we can perform scalar replacement transformations to change operations on object fields and array elements into operations on scalar variables.

9.6.1 Analysis Framework

In this section, we describe a unified representation called *Extended Array SSA* form, which can be used to perform sparse dataflow analysis of values through scalars, array elements, and object references. First, we introduce a formalism called *Heap Arrays* which allows us to represent object references with the same representation used to represent named arrays [**?**]. Then, we show how to use the Extended Array SSA representation and global value numbering to disambiguate pointers with the same framework used to analyze array indices.

9.6.1.1 Heap Arrays

The partitioning of memory locations into heap arrays is analogous to the partitioning of memory locations using type-based alias analysis [**?**]. The main difference is that our approach also performs a flow-sensitive analysis of element-level accesses to the heap arrays. We model accesses to object fields as follows. For each field x, we introduce a hypothetical one-dimensional heap array, $\mathcal{H}^x$. Heap array $\mathcal{H}^x$ consolidates all instances of field x present in the heap. Heap arrays are indexed by object references. Thus, a GETFIELD of $p.x$ is modeled as a read of element $\mathcal{H}^x[p]$, and a PUTFIELD of $q.x$ is modeled as a write of element $\mathcal{H}^x[q]$. The use of distinct heap arrays for distinct fields leverages the fact that accesses to distinct fields must be directed to distinct memory locations in a strongly typed language. Note that field x is considered to be the same field for objects of types C_1 and C_2, if C_2 is a subtype of C_1.

Recall that arrays in an object-oriented language like Java are also allocated, so both an object reference and an integer subscript are necessary for accessing an array element. Such arrays can be modeled as *two-dimensional* heap arrays, with one dimension indexed by the object reference as in heap arrays for fields, and the second dimension indexed by the integer subscript. Further details on how array objects are handled can be found in [**?**].

Having modeled object and array references as accesses to named arrays, we can rename heap arrays and scalar variables to build an extended version of Array SSA form [**?**]. First, we rename heap arrays so that each renamed heap array has a unique static definition. This includes renaming of the dummy definition inserted at the start block to capture the unknown initial value of the heap array.

We insert three kinds of ϕ functions to obtain an *extended* Array SSA form that we use for data flow analyses[3]:

1. A *control* ϕ from scalar SSA form [**?**].
2. A *definition* ϕ ($d\phi$) from Array SSA form [**?**, **?**].
3. A *use* ϕ ($u\phi$) function creates a new name whenever a statement reads a heap array element. $u\phi$ functions represent the extension in "extended" Array SSA form.

The main purpose of the $u\phi$ function is to link together load instructions for the same heap array in control flow order. Intuitively, the $u\phi$ function creates a new SSA variable name, with which a sparse dataflow analysis can associate a lattice variable. We present one dataflow algorithm that uses this information for redundant load identification later in the chapter. Other algorithms (eg. constant propagation) will not require a new name at each use, in which case the $u\phi$ function can be ignored.

9.6.1.2 *Definitely-Same and Definitely-Different Analyses for Heap Array Indices*

In this section, we show how the heap arrays of Extended Array SSA form reduce questions of pointer analysis questions regarding array indices. In particular, we show how global value numbering and allocation site information can be used to efficiently compute *definitely-same* ($\mathcal{DS}$) and *definitely-different* ($\mathcal{DD}$) information for heap array indices.

As an example, consider the following Java source code fragment annotated with heap array accesses:

```
r = p ;
q = new Type1 ;
p.y = ... ;      // H^y[p] := ...
q.y = ... ;      // H^y[q] := ...
... = r.y ;      // ... := H^y[r]
```

One analysis goal is to identify the redundant load of `r.y`, enabling the compiler to replace it with a use of scalar temporary that captures the value stored into `p.y`. We need to establish two facts to perform this transformation: 1) object references p and r are identical (definitely same) in all program executions, and 2) object references q and r are distinct (definitely different) in all program executions.

As before, we use the notation $\mathcal{V}(i)$ to denote the value number of SSA variable i. Therefore, if $\mathcal{V}(i) = \mathcal{V}(j)$, then $\mathcal{DS}(i, j) = true$. For the code fragment above, the statement, `p = r`, ensures that p and r are given the same value number (*i.e.,* $\mathcal{V}(p) = \mathcal{V}(r)$), so that $\mathcal{DS}(p, r) = true$.

[3] The extended Array SSA form can also be viewed as a sparse data flow evaluation graph [**?**] for a heap array.

The problem of computing $\mathcal{DD}$ for object references is more complex than value numbering, and relates to the classical work on pointer alias analysis. We outline a simple approach below, which can be replaced by more sophisticated techniques that may be available. We rely on two observations related to allocation-site information:

1. Object references that contain the results of distinct allocation-sites must be different.
2. An object reference containing the result of an allocation-site must be different from any object reference that occurs at a program point that dominates the allocation site in the control flow graph. (As a special case, this implies that the result of an allocation site must be distinct from all object references that are method parameters.)

For example, in the above code fragement, the presence of the allocation site in `q = new Type1` ensures that $\mathcal{DD}(p, q) = \mathit{true}$.

In the remainder of the chapter, we will use the notation $\mathcal{V}(\mathbf{k})$ to represent a vector of value numbers, $(\mathcal{V}(k_1), \ldots)$, given a vector index $\mathbf{k} = (k_1, \ldots)$. Thus, $\mathcal{DS}(\mathbf{j}, \mathbf{k})$ is *true* if and only if vectors $\mathbf{j}$ and $\mathbf{k}$ have the same size, and their corresponding elements are definitely-same *i.e.,* $\mathcal{DS}(j_i, k_i) = \mathit{true}$ for all i. Analogously, $\mathcal{DD}(\mathbf{j}, \mathbf{k})$ is *true* if and only if vectors $\mathbf{j}$ and $\mathbf{k}$ have the same size, and at least one pair of elements is definitely-different *i.e.,* $\mathcal{DD}(j_i, k_i) = \mathit{true}$ for some i.

9.6.2 Scalar Replacement Algorithms

In this section, we introduce two new analyses based on Extended Array SSA form. These two analyses form the backbone of *scalar replacement* transformations, which replace accesses to memory by uses of scalar temporaries. First, we present an analysis to identify fully redundant loads. Then, we present an analysis to identify dead stores.

Figure 9.20 illustrates three different cases of scalar replacement for object fields. All three cases can be identified by the algorithms presented in this chapter. (Similar examples can be constructed for scalar replacement of array elements.) For the original program in figure 9.20(a), introducing a scalar temporary `T1` for the store (def) of `p.x` can enable the load (use) of `p.x` to be eliminated *i.e.,* to be replaced by a use of `T1`. Figure 9.20(b) contains an example in which a scalar temporary (`T2`) is introduced for the first load of `p.x`, thus enabling the second load of `p.x` to be eliminated *i.e.,* replaced by `T2`. Finally, figure 9.20(c) contains an example in which the first store of `p.x` can be eliminated because it is known to be dead (redundant); no scalar temporary needs to be introduced in this case.

Original program:

```
  p  := new Type1
  q  := new Type1
  . . .
 p.x := ...
 q.x := ...
...  := p.x
```

After redundant load elimination:

```
  p  := new Type1
  q  := new Type1
  . . .
  T1 := ...
 p.x := T1
 q.x := ...
...  := T1
```

(a)

Original program:

```
  p  := new Type1
  q  := new Type1
  . . .
...  := p.x
 q.x := ...
...  := p.x
```

After redundant load elimination:

```
  p  := new Type1
  q  := new Type1
  . . .
  T2 := p.x
...  := T2
 q.x := ...
...  := T2
```

(b)

Original program:

```
  p  := new Type1
  q  := new Type1
  r  := p
  . . .
 p.x := ...
 q.x := ...
 r.x := ...
```

After dead store elimination:

```
  p  := new Type1
  q  := new Type1
  r  := p
  . . .
 q.x := ...
 r.x := ...
```

(c)

Fig. 9.20 Examples of scalar replacement

9.6.2.1 Redundant Load Elimination

Figure 9.21 outlines our algorithm for identifying uses (loads) of heap array elements that are redundant with respect to prior defs and uses of the same heap array. The algorithm's main analysis is *index propagation*, which identifies the set of indices that are *available* at a specific def/use A_i of heap array A.

Index propagation is a dataflow problem, which computes a lattice value $\mathcal{L}(\mathcal{H})$ for each heap variable $\mathcal{H}$ in the Array SSA form. This lattice value $\mathcal{L}(\mathcal{H})$ is a set of value number vectors $\{\mathbf{i_1}, \ldots\}$, such that a load of $\mathcal{H}[\mathbf{i}]$ is *available* if $\mathcal{V}(\mathbf{i}) \in \mathcal{L}(\mathcal{H})$. Note that the lattice element does not include the value of $\mathcal{H}[\mathbf{i}]$ (as in constant propagation), just teh fact that it is available. Figures 9.22, 9.23 and 9.24 give the lattice computations which define the index propagation solution. The notation $\textsc{update}(\mathbf{i'}, \langle \mathbf{i_1}, \ldots \rangle)$ used in the middle cell in figure 9.22 denotes a special update of the list $\mathcal{L}(A_0) = \langle \mathbf{i_1}, \ldots \rangle$ with respect to index $\mathbf{i'}$. UPDATE involves four steps:

1. Compute the list $T = \{\ \mathbf{i_j} \mid \mathbf{i_j} \in \mathcal{L}(A_0)$ and $\mathcal{DD}(\mathbf{i'}, \mathbf{i_j}) = \mathit{true}\ \}$. List T contains only those indices from $\mathcal{L}(A_0)$ that are *definitely different* from $\mathbf{i'}$.
2. Insert $\mathbf{i'}$ into T to obtain a new list, I.
3. If the size of list I exceeds the threshold size Z, then one of the indices in I is dropped from the output list so as to satisfy the size constraint. (Since the size of $\mathcal{L}(A_0)$ must have been $\leq Z$, it is sufficient to drop only one index to satisfy the size constraint.)
4. Return I as the value of $\textsc{update}(\mathbf{i'}, \langle \mathbf{i_1}, \ldots \rangle)$.

Input: Intermediate code for method being optimized, augmented with the $\mathcal{DS}$ and $\mathcal{DD}$ relations defined in Section 9.6.1.2.

Output: Transformed intermediate code after performing scalar replacement.

Algorithm:

1. **Build extended Array SSA form for each heap array.**
   Build Array SSA form, inserting control ϕ, $d\phi$ and $u\phi$ functions as outlined in Section 9.6.1.1, and renaming of all heap array definitions and uses.
   As part of this step, we annotate each call instruction with dummy defs and uses of each heap array for which a def or a use can reach the call instruction. If interprocedural analysis is possible, the call instruction's heap array defs and uses can be derived from a simple flow-insensitive summary of the called method.
2. **Perform index propagation.**
   a. Walk through the extended Array SSA intermediate representation, and for each ϕ, $d\phi$, or $u\phi$ statement, create a dataflow equation with the appropriate operator as listed in Figures 9.22, 9.23 or 9.24.
   b. Solve the system of dataflow equations by iterating to a fixed point.

   After index propagation, the lattice value of each heap array, A_i, is $\mathcal{L}(A_i) = \{ \mathcal{V}(\mathbf{k}) \mid \text{location } A[\mathbf{k}] \text{ is "available" at def } A_i \text{ (and all uses of } A_i) \}$.
3. **Scalar replacement analysis.**
   a. Compute $UseRepSet = \{ \text{ use } A_j[\mathbf{x}] \mid \exists\ \mathcal{V}(\mathbf{x}) \in \mathcal{L}(A_j) \}$ *i.e.,* use $A_j[\mathbf{x}]$ is placed in *UseRepSet* if and only if location $A[\mathbf{x}]$ is available at the def of A_j and hence at the use of $A_j[\mathbf{x}]$. (Note that A_j uniquely identifies a use, since all uses are renamed in extended Array SSA form.)
   b. Compute $DefRepSet = \{ \text{ def } A_i[\mathbf{k}] \mid \exists \text{ use } A_j[\mathbf{x}] \in UseRepSet \text{ with } \mathcal{V}(\mathbf{x}) = \mathcal{V}(\mathbf{k}) \}$ *i.e.,* def $A_i[\mathbf{k}]$ is placed in *DefRepSet* if and only if a use $A_j[\mathbf{x}]$ was placed in *UseRepSet* with $\mathcal{V}(\mathbf{x}) = \mathcal{V}(\mathbf{k})$.
4. **Scalar replacement transformation.**
   Apply scalar replacement actions selected in step 3 above to the *original* program and obtain the transformed program.

Fig. 9.21 Overview of Redundant Load Elimination algorithm.

$\mathcal{L}(A_2)$	$\mathcal{L}(A_0) = \top$	$\mathcal{L}(A_0) = \langle(\mathbf{i_1}), \ldots\rangle$	$\mathcal{L}(A_0) = \bot$
$\mathcal{L}(A_1) = \top$	$\top$	$\top$	$\top$
$\mathcal{L}(A_1) = \langle(\mathbf{i'})\rangle$	$\top$	$\textsc{update}((\mathbf{i'}), \langle(\mathbf{i_1}), \ldots\rangle)$	$\langle(\mathbf{i'})\rangle$
$\mathcal{L}(A_1) = \bot$	$\bot$	$\bot$	$\bot$

Fig. 9.22 Lattice computation for $\mathcal{L}(A_2) = \mathcal{L}_{d\phi}(\mathcal{L}(A_1), \mathcal{L}(A_0))$ where $A_2 := d\phi(A_1, A_0)$ is a definition ϕ operation

$\mathcal{L}(A_2)$	$\mathcal{L}(A_0) = \top$	$\mathcal{L}(A_0) = \langle(\mathbf{i_1}), \ldots\rangle$	$\mathcal{L}(A_0) = \bot$
$\mathcal{L}(A_1) = \top$	$\top$	$\top$	$\top$
$\mathcal{L}(A_1) = \langle(\mathbf{i'})\rangle$	$\top$	$\mathcal{L}(A_1) \cup \mathcal{L}(A_0)$	$\mathcal{L}(A_1)$
$\mathcal{L}(A_1) = \bot$	$\bot$	$\bot$	$\bot$

Fig. 9.23 Lattice computation for $\mathcal{L}(A_2) = \mathcal{L}_{u\phi}(\mathcal{L}(A_1), \mathcal{L}(A_0))$ where $A_2 := u\phi(A_1, A_0)$ is a use ϕ operation

$\mathcal{L}(A_2) = \mathcal{L}(A_1) \sqcap \mathcal{L}(A_0)$	$\mathcal{L}(A_0) = \top$	$\mathcal{L}(A_0) = \langle(\mathbf{i_1}), \ldots\rangle$	$\mathcal{L}(A_0) = \bot$
$\mathcal{L}(A_1) = \top$	$\top$	$\mathcal{L}(A_0)$	$\bot$
$\mathcal{L}(A_1) = \langle(\mathbf{i'_1}), \ldots\rangle$	$\mathcal{L}(A_1)$	$\mathcal{L}(A_1) \cap \mathcal{L}(A_0)$	$\bot$
$\mathcal{L}(A_1) = \bot$	$\bot$	$\bot$	$\bot$

Fig. 9.24 Lattice computation for $\mathcal{L}(A_2) = \mathcal{L}_\phi(\mathcal{L}(A_1), \mathcal{L}(A_0)) = \mathcal{L}(A_1) \sqcap \mathcal{L}(A_0)$, where $A_2 := \phi(A_1, A_0)$ is a control ϕ operation

(a) Extended Partial Array SSA form:	(b) After index propagation:	(c) Scalar replacement actions selected:	(d) After transforming original program:

`p := new Type1`
`q := new Type1`
`. . .`
$\mathcal{H}_1^x[p]$ `:= ...`
$\mathcal{H}_2^x$ `:=` $d\phi(\mathcal{H}_1^x, \mathcal{H}_0^x)$
$\mathcal{H}_3^x[q]$ `:= ...`
$\mathcal{H}_4^x$ `:=` $d\phi(\mathcal{H}_3^x, \mathcal{H}_2^x)$
`... :=` $\mathcal{H}_4^x[p]$
$\mathcal{H}_5^x$ `:=` $u\phi(\mathcal{H}_4^x, \mathcal{H}_3^x)$
	$\mathcal{L}(\mathcal{H}_0^x) = \{\ \}$
	$\mathcal{L}(\mathcal{H}_1^x) = \{\mathcal{V}(p)\}$
	$\mathcal{L}(\mathcal{H}_2^x) = \{\mathcal{V}(p)\}$
	$\mathcal{L}(\mathcal{H}_3^x) = \{\mathcal{V}(q)\}$
	$\mathcal{L}(\mathcal{H}_4^x) = \{\mathcal{V}(p), \mathcal{V}(q)\}$
	$\mathcal{L}(\mathcal{H}_5^x) = \{\mathcal{V}(p), \mathcal{V}(q)\}$
		$UseRepSet = \{\mathcal{H}_4^x[p]\}$
		$DefRepSet = \{\mathcal{H}_1^x[p]\}$
			`p := new Type1`
			`q := new Type1`
			`. . .`
			$A_temp_{\mathcal{V}(p)}$ `:= ...`
			`p.x :=` $A_temp_{\mathcal{V}(p)}$
			`q.x := ...`
			`... :=` $A_temp_{\mathcal{V}(p)}$

Fig. 9.25 Trace of load elimination algorithm from figure 9.21 for program in figure 9.20(a)

After index propagation, the algorithm selects an array use (load), $A_j[\mathbf{x}]$, for scalar replacement if and only if index propagation determines that an index with value number $\mathcal{V}(\mathbf{x})$ is available at the def of A_j. If so, the use is included in *UseRepSet* the set of uses selected for scalar replacement. Finally, an array def, $A_i[\mathbf{k}]$, is selected for scalar replacement if and only if some use $A_j[\mathbf{x}]$ was placed in *UseRepSet* such that $\mathcal{V}(\mathbf{x}) = \mathcal{V}(\mathbf{k})$. All such defs are included in *DefRepSet* the set of defs selected for scalar replacement.

Figure 9.25 illustrates a trace of this load elimination algorithm for the example program in figure 9.20(a). Figure 9.25(a) shows the partial Array SSA form computed for this example program. The results of index propagation are shown in figure 9.25(b). These results depend on definitely-different analysis establishing that $\mathcal{V}(p) \neq \mathcal{V}(q)$ by using allocation site information as described in Section 9.6.1.2. Figure 9.25(c) shows the scalar replacement actions derived from the results of index propagation, and Figure 9.25(d) shows the transformed code after performing these scalar replacement actions. The load of `p.x` has thus been eliminated in the transformed code, and replaced by a use of the scalar temporary, $A_temp_{\mathcal{V}(p)}$.

We conclude this section with a brief discussion of the impact of the Java Memory Model (JMM). It has been observed that redundant load elimination can be an illegal transformation for multithreaded programs with data races written for a memory model, such as the JMM, that includes the memory coherence assumption [**?**]. If necessary, our algorithms can be modified to obey memory coherence by simply treating each $u\phi$ function as a $d\phi$ function *i.e.,* by treating each array use also as an array def.

9.6.2.2 Dead Store Elimination

In this section, we show how our Array SSA framework can be used to identify redundant (dead) stores of array elements. Dead store elimination is related to load elimination, because scalar replacement can convert non-redundant stores into redundant stores. For example, consider the program in Figure 9.20(a). If it contained an additional store of `p.x` at the bottom, the first store of `p.x` will become redundant after scalar replacement. The program after scalar replacement will then be similar to the program shown in Figure 9.20(c) as an example of dead store elimination.

Our algorithm for dead store elimination is based on a backward propagation of *DEAD* sets. As in load elimination, the propagation is *sparse i.e.,* it goes through ϕ nodes in the Array SSA form rather than basic blocks in a control flow graph. However, $u\phi$ functions are not used in dead store elimination, since the ordering of uses is not relevant to identifying a dead store. Without $u\phi$ functions, it is possible for multiple uses to access the same heap array name. Hence, we use the notation $\langle A, s\rangle$ to refer to a specific use of heap array A in statement (instruction) s. We also use the term "non-ϕ" to refer to any def or use that does not appear in a (control or definition) ϕ statement *i.e.,* a def or use from the original program.

Consider a ϕ *def* A_i, a ϕ or non-ϕ *use* $\langle A_j, s\rangle$, and a real (non-ϕ) def A_k in Array SSA form. We define the following four sets:

$$
\begin{aligned}
DEAD_{def}(A_i) &= \{\ \mathcal{V}(x) \mid \text{element } x \text{ of array } A \text{ is dead at } \phi \text{ def } A_i\ \} \\
DEAD_{use}(\langle A_j, s\rangle) &= \{\ \mathcal{V}(x) \mid \text{element } x \text{ of array } A \text{ is dead at non-}\phi \text{ use of } A_j \text{ in statement } s\ \} \\
KILL(A_k) &= \{\ \mathcal{V}(x) \mid \text{element } x \text{ of array } A \text{ is killed by non-}\phi \text{ def of } A_k\ \} \\
LIVE(A_i) &= \{\ \mathcal{V}(x) \mid \exists \text{ a non-}\phi \text{ use } A_i[x] \text{ of } \phi \text{ def } A_i\ \}
\end{aligned}
$$

The *KILL* and *LIVE* sets are local sets *i.e.,* they can be computed immediately without propagation of data flow information. If A_i "escapes" from the procedure (*i.e.,* definition A_i is exposed on procedure exit), then we must conservatively set $LIVE(A_i) = \mathcal{U}^A_{ind}$, the universal set of index value numbers for array A. Note that in Java, every instruction that can potentially throw an uncaught exception must be treated as a procedure exit, although this property can be relaxed with some interprocedural analysis.

The data flow equations used to compute the $DEAD_{def}$ and $DEAD_{use}$ sets are given in Figure 9.26. The goal of our analysis is to find the maximal $DEAD_{def}$ and $DEAD_{use}$ sets that satisfy these equations. Hence our algorithm will initialize each $DEAD_{def}$ and $DEAD_{use}$ set to $= \mathcal{U}^A_{ind}$ (for renamed arrays derived from original array A), and then iterate on the equations till a fixpoint is obtained. After *DEAD* sets have been computed, we can determine if a real (non-ϕ) definition is redundant quite simply as follows. Consider a real definition, $A_1[j] := \ldots$, followed by a definition ϕ statement, $A_2 := d\phi(A_1, A_0)$. Then, if $\mathcal{V}(j) \in DEAD(A_2)$, then def (store) A_1 is redundant and can be eliminated.

1. **Propagation from the LHS to the RHS of a control ϕ:**
   Consider a control ϕ statement s of the form, $A_2 := \phi(A_1, A_0)$. In this case, the uses, $\langle A_1, s\rangle$ and $\langle A_0, s\rangle$, must both come from ϕ defs, and the propagation of $DEAD_{def}(A_2)$ to the RHS is a simple copy *i.e.,* $DEAD_{use}(\langle A_1, s\rangle) = DEAD_{def}(A_2)$ and $DEAD_{use}(\langle A_0, s\rangle) = DEAD_{def}(A_2)$.
2. **Propagation from the LHS to the RHS of a definition ϕ:**
   Consider a $d\phi$ statement s of the form $A_2 := d\phi(A_1, A_0)$. In this case use $\langle A_1, s\rangle$ must come from a real (non-ϕ) definition, and use $\langle A_0, s\rangle$ must come from a ϕ definition. The propagation of $DEAD_{def}(A_2)$ and $KILL(A_1)$ to $DEAD_{use}(\langle A_0, s\rangle)$ is given by the equation, $DEAD_{use}(\langle A_0, s\rangle) = KILL(A_1) \cup DEAD_{def}(A_2)$.
3. **Propagation to the LHS of a ϕ statement from uses in other statements:**
   Consider a definition or control ϕ statement of the form $A_i := \phi(\ldots)$. The value of $DEAD_{def}(A_i)$ is obtained by intersecting the $DEAD_{use}$ sets of all uses of A_i, and subtracting out all value numbers that are not definitely different from every element of $LIVE(A_i)$. This set is specified by the following equation:

$$DEAD_{def}(A_i) = \left(\bigcap_{s \text{ is a } \phi \text{ use of } A_i} DEAD_{use}(\langle A_i, s\rangle) \right) - \{v | \exists w \in LIVE(A_i) s.t. \neg DD(v, w)\}$$

Fig. 9.26 Data flow equations for $DEAD_{def}$ and $DEAD_{use}$ sets

9.7 Summary

Static single assignment (SSA) form for scalars has been a significant advance. It has simplified the way we think about scalar variables. It has simplified the design of some optimizations and has made other optimizations more effective. A direct application of classical SSA form to arrays, would view an array as a single object. But the kinds of analyses that sophisticated compilers need to perform on arrays, for example those that drive loop parallelization, are at the element level. In this chapter, we introduced an Array SSA form that captures precise element-level data flow information for array variables. It is general and simple, and coincides with standard SSA form when applied to scalar variables. It can also be used for structures and other variable types that can be modeled as arrays. We presented efficient algorithms based on Array SSA form that perform constant propagation and conditional constant propagation through both scalar and array references. These algorithms use an extension of the classical constant propagation lattice so that it can efficiently record information about array elements. The original sparse conditional constant propagation algorithm in [**?**] dealt with control flow and data flow separately by maintaining two distinct work lists. The algorithm presented in this chapter is conceptually simpler because it uses a single set of data flow equations instead.

We also presented a unified framework to analyze object-field and array-element references for programs written in strongly-typed languages such as Java and Modula-3. Our solution models object references as heap arrays, and uses global value numbering and allocation site information to determine if two object references are known to be same or different. We presented algorithms to identify fully

redundant loads and dead stores, based on sparse propagation in an extended Array SSA form.

There are many possible directions for future research based on this work. One direction is to extend the value numbering and definitely-different analyses mentioned in section 9.5 so that they can be combined with conditional constant propagation rather than performed as a pre-pass. Further, it would be interesting to gain an understanding of when adding lattice elements for @ variables could lead to extra precision in program analysis compared to using executable flags. An ultimate goal is to combine conditional constant and type propagation, value numbering, partial redundancy elimination, and scalar replacement analyses with a single framework that can analyze heap accesses as effectively as scalar operations.

CHAPTER 10

Hashed SSA form: HSSA

M. Mantione
F. Chow

Progress: 53%

Review in progress

10.1 Introduction

Hashed SSA (or in short HSSA), as described in [Chow...], is an SSA extension that can effectively represent how aliasing relations affect a program in SSA form. It works equally well for aliasing among local variables and, more generally, for indirect load and store operations on arbitrary memory locations. This allows the application of all common SSA based optimizations uniformly both on local variables and on external memory areas.

It should be noted, however, that HSSA is a technique useful for representing aliasing effects, but not for detecting aliasing. For this purpose, a separate alias analysis pass must be performed, and the effectiveness of HSSA will be influenced by the accuracy of this analysis.

The following sections explain how HSSA works. Initially, given aliasing information, we will see how to represent them in SSA form for scalar variables. Then we will introduce a technique that reduces the overhead of the above representation, avoiding an explosion in the number of SSA versions for aliased variables. Subsequently we will represent indirect operations on external memory areas as operations on ”virtual variables” in SSA form, which will be handled uniformly with scalar (local) variables. Finally we will apply global value numbering (GVN) to all of the above, obtaining the Hashed SSA form.

10.2 SSA and aliasing: μ and χ functions

Aliasing is the situation when, inside a compilation unit, one single storage location (that contains a value) can be potentially accessed through different program ”variables”. This can happen in one of the four following ways:

- First, when two or more storage locations partially overlap. This, for instance, happens with the C ”union” construct, where different parts of a program can access the same storage location under different names.
- Second, when a local variable is referred by a pointer used in an indirect operation. In this case the two ways in which the variable can be accessed are *directly*, through the variable name, and *indirectly*, through its address stored in another variable.
- Third, when the address of a local variable is passed to a function, which in turn can then access the variable indirectly.
- Finally, variables that are not local can obviously be accessed by different functions, and in this case every function call can potentially access every global variable, unless the compiler uses global optimizations techniques where every function is analyzed before the actual compilation takes place.

The real problem with aliasing is that these different accesses to the same program variable are difficult to predict. Only in the first case (explicitly overlapping locations) the compiler has full knowledge of when each access takes place. In all the other cases (indirect accesses through the address of the variable) the situation becomes more complex, because the access depends on the address that is effectively stored in the variable used in the indirect operation. This is a problem because every optimization pass is concerned with the actual value that is stored in every variable, and when those values are used. If variables can be accessed in unpredictable program points, the only safe option for the compiler is to handle them as ”volatile” and avoid performing optimizations on them, which is not desirable.

Intuitively, in presence of aliasing the compiler could try to track the values of variable addresses inside other variables (and this is exactly what HSSA does), but the formalization of this process is not trivial. The first thing that is needed is a way to model the *effects* of aliasing on a program in SSA form. To do this, assuming that we have already performed alias analysis, we must formally define the effects of indirect definitions and uses of variables. Particularly, each definition can be a *”MustDef”* operation in the direct case, or a *”MayDef”* operation in the indirect case. We will represent MayDef operations with the χ operator. Similarly, uses can be *”MustUse”* or *”MayUse”* operations (respectively in the direct and indirect case), and we will represent MayUse operations with the μ operator. Obviously the argument of the μ operator is the potentially used variable. Less obviously, the argument to the χ operator operator is the assigned variable itself. This makes so that liveness information is naturally correct: the χ operator only *potentially* modifies the variable, so the original value could ”flow through” it. By making the χ operator

use the original value, a store can be considered dead only if it and all its associated χs are not marked live.

The use of μ and χ operations does not alter the complexity of transforming a program in SSA form. All that is necessary is a pre-pass that inserts them before or after the instructions that involve aliasing. Particularly, μ operations must be inserted immediately *before* the involved statement or expression, and χ operations immediately *after* it. This distinction allows us to model call effects correctly: the called function appears to potentially use the values of variables before the call, and the potentially modified values appear after the call.

This can be clarified with an example (FIXME: turn the example into a picture). Consider the following code fragment in C syntax:

```
i = 2; if (j) {f();} else {a = *p; *p = 3;} return i;
```

and assume that p might alias i, and that function f might indirectly use i but not alter it. The code, after the μ and χ insertion pass, becomes the following:

```
i = 2; if (j) {$\mu$(i); f();} else {*p = 3; i = $\chi$(i);} return i;
```

The same code in SSA form becomes:

```
i1 = 2; if (j1) {$\mu$(i1); f();} else {*p = 3; i2 = $\chi$(i1);} i3 = $\phi$(i1,i2); return
```

where it is clear that the SSA renaming and ϕ insertion must be applied in the usual way, handling μ and χ operators as every other expression and statement.

10.3 Introducing "zero versions" to limit the explosion of the number of variables

While it is true that μ and χ insertion does not alter the complexity of SSA construction, applying it to a production compiler as described in the previous section would make working with code in SSA form terribly inefficient. This is because χ operators cause an explosion in the number of variable values, inducing the insertion of new ϕ functions which in turn create new variable versions. In practice the resulting IR would be needlessly large, not in terms of instructions, but in number of distinct variable versions. The biggest issue is that the SSA versions introduced by χ operators are mostly useless: since χ definitions are by nature uncertain, the actual value of a variable after a χ definition is unknown: it could be its original value, or it could be the one indirectly assigned by the χ, but any optimization that deals with variable values will not be able to operate on that.

Intuitively, the solution to this problem is to factor all these useless variable versions together, so that no space is wasted to distinguish among them. We assign number 0 to this special variable version, and call it "*zero version*".

The formal definition of zero version relies on the concept of "*real occurrence*", which is an occurrence of a variable in the original program. Therefore, in SSA form

variable occurrences in μ, χ and ϕ operators are not "real occurrences". The idea is that variable versions that have no real occurrence do not influence the program output (because once the program is converted back from SSA form these variables are removed from the code). Since they do not directly appear in the code, and their value is unknown, distinguishing among them is almost pointless. A direct definition of zero versions is that they are versions of variable that have no real occurrence, and whose value comes from at least one χ operator (optionally through ϕ functions). An equivalent, recursive definition is the following:

- The result of a χ has zero version if it has no real occurrence.
- If the operand of a ϕ has zero version, the ϕ result has zero version if it has no real occurrence.

To detect zero versions, we associate a "HasRealOcc" flag to each variable version, setting it to true whenever a real occurrence is met in the code (this can be done while constructing the SSA form). Moreover, we associate to each original (unversioned) program variable a list "NonZeroPhiList", initially empty. The detection algorithm is the following:

```
* For each version of each variable:
      + if HasRealOcc is false and the version is defined by a $\chi$, set version to ze
      + if HasRealOcc is false and the version is defined by a $\phi$, examine the $\phi
            - if all of them have HasRealOcc true, set the $\phi$ HasRealOcc to true
            - if one or more has version zero, set $\phi$ version to zero
            - otherwise, add $\phi$ to NonZeroPhiList of its variable
* For each program variable, iterate until NonZeroPhiList no longer changes, and for eac
      o if all of them have HasRealOcc true, set the $\phi$ HasRealOcc to true
      o if one or more has version zero, set $\phi$ version to zero
      o if any of the two above conditions is met, remove the $\phi$ from the list
```

The time spent in the first iteration grows linearly with the number of variable versions, which in turn is proportional to the number of definitions and therefore to the code size. On the other hand, the upper bound of each final iteration on a NonZeroPhiList is the longest chain of contiguous ϕ assignments in the program. All in all, zero version detection in the presence of μ and χ functions does not change the complexity of SSA construction in a significant way, while the corresponding reduction in the number of variable versions is definitely desirable.

This loss of information has almost no consequences on the effectiveness of subsequent optimization passes. Since variables with zero versions have no known value, not being able to distinguish them does not affect optimizations that operate on values. On the other hand, when performing DEADCE zero versions must be assumed live. This is necessary because we cannot know their actual liveness status (all their uses appear as uses of generic zero versions, so they are no longer distinct). However also this has no practical effect, because zero versions have no real definitions (they have no real occurrence), so there is no real statement that would be eliminated by the DEADCE pass if we could detect that a zero version occurrence is dead. There is only one case in DEADCE where the information loss is evident.

A zero version can be used in a μ and defined in a χ, which in turn can have a real occurrence as argument. If the μ is eliminated by some optimization, the real occurrence used in the χ becomes dead but we cannot detect it (because the χ will not be eliminated). According to [Chow...] this case is sufficiently rare in real world code that using zero versions is anyway convenient.

10.4 SSA and indirect memory operations: indirect variables

The techniques described in the previous sections (μ and χ operators introduction and using zero version for SSA variables with no real occurrences or meaningful values) only apply to ”regular“ variables in a compilation unit, and not to arbitrary memory locations accessed indirectly. So, using them we can apply SSA based optimizations to variables also when they are affected by aliasing, but memory access operations are still excluded from the SSA form.

This situation is far from ideal, because code written in current mainstream imperative languages (like C++, Java or C#) typically contains many operations on data stored in global memory areas. Every heap allocated object is referred through pointers, and typically to access individual object fields offsets are added to the pointer values. Code that works with those objects will usually access the fields directly inside expressions and statements, and the result is that a lot of operations are performed on values stored in indirectly accessed global memory locations. It is also worth noting that operations on array element suffer from the same problem.

The situation is not ideal because conventional SSA techniques only handle local variables without aliasing. By annotating the code with μ and χ operators we can deal with aliasing in local variables, but we are still unable to optimize memory accesses (indirect loads and stores) and all the operations that use values that come from them.

The purpose of HSSA is to handle direct and indirect operations uniformly, and be able to apply all SSA based optimizations on them. To do this, in the general case, we assume that the code intermediate representation supports a *dereference* operator, which performs an indirect load (memory read) from the given address. This operator can be placed in expressions on both the left and right side of the assignment operator, and we will represent it with the usual ”*” C language operator. We will then, for simplicity, represent indirect stores with the usual combination of dereference and assignment operators, like in C. Some examples of this notation are:

- *p: access memory at address p.
- *(p+1): access memory at address p+1 (like to access an object field at offset 1).
- **p: double indirection.
- *p = expression: indirect store.

An intuitive approach to deal with indirect operations in SSA form would be to consider, in each indirect operation, the referred location as an ”indirect variable”. In this approach we would first put in SSA form all direct variables. At this point each occurrence of the ”*” operator would have as argument an address expression, with all its variables in SSA form (in the simplest case this expression would be a single variable). We can then consider each ”(*expression)” construct as an indirect variable, because it represents the location referred by the ”*” operator, and apply the SSA construction algorithm again on these ”variables”. For instance, the code ”i = *p; p = p + 1; j = *p;”, when put in SSA form, becomes ”i1 = *p1; p2 = p1 + 1; j1 = *p;”. Applying SSA to indirect variables we obtain the notation ”i1 = (*p1)1; p2 = p1 + 1; j1 = (*p2)2;”. In this notation it is evident that (*p1)1 and (*p2)2 have different values, while, for the properties of SSA code, identical expressions will have the same value. If the code contains multiple indirections (like ”*(*p + 1)”), the renaming process must be run separately (and sequentially) for each subsequent level of indirection.

10.5 From *indirect variables to* virtual variables

Indirect variables, as described in the previous section, are intuitive but impractical. The biggest problem they have, from a practical point of view, is that they require the application of the SSA renaming algorithm multiple times. But they also raise the issue of dealing with aliasing among indirect variables in a sound way, just like we do with direct variables using μ and χ operators: the algorithm we showed for placing them only works for direct variables.

To deal with this, we introduce the concept of ”*virtual variable*”. The virtual variable of an indirect variable is an imaginary scalar variable that has identical aliasing characteristics as the indirect variable. It represent the location (or better the set of potential locations) referred by the indirect memory operation of the indirect variable. Contrast this with the indirect operation itself, which is the actual code that is emitted by the compiler (the dereferencing operation).

The key part of the definition of ”virtual variable” is that it must have identical aliasing characteristics as its indirect variable. In this sense, while an indirect variable intuitively *is* the referred memory location, its corresponding virtual variable is a ”formal placeholder” for the memory location, with the same aliasing properties. As a placeholder, we handle it like every other scalar variable, and particularly μ and χ operators are inserted in he code in the same pass and according to the same rules. Therefore, each occurrence of an indirect variable is annotated with the μ and χ of its virtual variable. As an added bonus, to improve the accuracy of aliasing information, alias analysis can take into account their address expressions to deduce that two indirect variables do not alias each other (because this means that also their virtual variables will not alias each other). For example, it is clear that *p and *(p + 1) do not alias each other.

SSA renaming on virtual variables works as usual, and in the same pass as with scalar variables. In this context, use-def relationships of virtual variables now represent use-def relationships of indirect variables. Note how this approach effectively solves the SSA renaming problem for indirect variables: their SSA version is the version numbers of their virtual variable (unless the address expression itself has changed version, in which case we need to generate a new number). It is the fact that virtual variables have the same aliasing properties as indirect variables that allow us to perform one single, uniform SSA renaming pass, with μ and χ operators automatically imposing new version numbers as a consequence of indirect operations. Also the zero versioning technique can be applied unaltered to virtual variables.

10.6 GVN and indirect memory operations: HSSA

In the previous sections we sketched the foundations of a framework for dealing with aliasing and indirect operations in SSA form: we identified the effects of aliasing on local variables, introduced μ and χ operators to handle them, applied zero versioning to keep the number of SSA versions acceptable, considered indirect variables as a way to handle arbitrary memory accesses and defined virtual variables as a concrete (and feasible) way to apply SSA also to them.

However, in practice in HSSA we do not renumber indirect variables. Instead, we apply *Global Value Numbering* (FIXME: Find a bibliographic reference for it) (GVN) to both scalar and virtual variables, handling all of them uniformly. In the resulting code representation value numbers have the same role of SSA versions, which can effectively be discarded.

In this sense virtual variables are just an aid to apply aliasing effects to indirect operations, be aware of liveness and perform SSA renaming (with any of the regular SSA renaming algorithms). They have no real counterpart in the program code, and after GVN has been applied they are not considered anymore.

A program in HSSA form is a sequence of statements (assignments and procedure calls), where the expressions cannot have side effects and are composed by five kinds of nodes:

- Three kinds of leaf nodes: constants, addresses and variables.
- Operand nodes.
- Indirect variables (”*ivar*”), which are half operand and half variable nodes.

Ivar nodes correspond to indirect operations. They have a double role because as operations they behave as operands in the IR (they are not leaf nodes), but as ”variables” they represent a memory location that holds a value (identified by its value number). We call them *indirect* variables to distinguish them from the original program scalar variables, which are leaf nodes. But note that ivar nodes, considered as code, are operations and not variables: the compiler back end, when processing them, will emit indirect memory accesses to the address passed as their operand (in practice they correspond to the ”*” operator in the C programming language). Their

value number is needed because through it they can participate to all value based optimizations (like constant folding and propagation, redundancy elimination, register promotion, dead code elimination...). It is in this way that HSSA extends all SSA based optimizations to indirect operations, even if they were originally designed to be applied to ”plain” program variables.

10.7 Building HSSA

2 pages

The HSSA construction algorithm.

We now present the HSSA construction algorithm. It is straightforward, because it is a simple composition of μ and χ insertion, zero versioning and virtual variable introduction (described in previous sections), together with regular SSA renaming and GVN application.

```
SSA form construction
    * Perform alias analysis and assign virtual variables to indirect variables
    * Insert $\mu$ and $\chi$ operands for scalar and virtual variables
    * Insert $\phi$ operands (considering both regular and $\chi$ store operations)
    * Perform SSA renaming on all scalar and virtual variables
```

At the end of this step we have code in plain SSA form. The use of μ and χ operands guarantees that SSA versions are correct also in case of aliasing. Moreover, indirect operations are ”annotated” with virtual variables, and also virtual variables have SSA version numbers. However, note how virtual variables are sort of ”artificial” in the code and will not contribute to the final code generation pass, because what really matters are the indirect operations themselves.

```
Detecting zero versions
    * In one single pass:
          - perform DEADCE (also on $\chi$ and $\phi$ stores)
          - perform steps 1 and 2 of the zero version detection (up to setting HasRealOcc f
    * Perform the remaining passes of the zero version algorithm
```

At the end of this step the code has exactly the same structure as before, but the number of unique SSA versions had diminished because of the application of zero versions.

```
GVN application
    * Perform a pre-order traversal of the dominator tree, applying GVN to the whole code (
          - expressions are processed bottom up, reusing existing expression nodes and using
          - two ivar nodes have the same value number if these conditions are both true:
                - their address expressions have the same value number, and
                - their virtual variables have the same versions, or are separated by defin
```

The application of global value numbering is straightforward, as described in [FIXME: find a suitable reference, or maybe write a side bar briefly describing GVN]. As a result of this, each node in the code representation has a proper value number, and nodes with the same number are guaranteed to produce the same value (or hold it in the case of variables). The crucial issue is that the code must be traversed following the dominator tree in pre-order. This is important because when generating value numbers we must be sure that all the involved definitions already have a value number. Since in SSA form all definitions dominate their use, a dominator tree pre-order traversal satisfies this requirement.

Note that after this step virtual variables are not needed anymore, and can be discarded from the code representation: the information they convey about aliasing of indirect variables has already been used to generate correct value numbers for ivar nodes.

```
Linking definitions
    * The left side of each assignment (direct and indirect, real, $\phi$ and $\chi$) is up
    * Also all $\phi$, $\mu$ and $\chi$ operands are updated to point to the corresponding
```

At the end of this step the HSSA form is complete, and every value in the program code is represented by a reference to a node in the HSSA value table.

10.8 Using HSSA

1 page

As seen in the previous sections, HSSA is an internal code representation that applies SSA to indirect memory operations and builds a global value number table, valid also for values computed by indirect operations.

This representation, once built, is particularly memory efficient because expressions are shared among use points (they are represented as HSSA table nodes). In fact the original program code can be discarded, keeping only a list of statements pointing to the shared expressions nodes.

All optimizations conceived to be applied on scalar variables work ”out of the box” on indirect locations: of course the implementation must be adapted to use the HSSA table, but their algorithm (and computational complexity) is the same even when dealing with values accessed indirectly.

Indirect operation (ivar) nodes are both variables and expressions, and benefit from the optimizations applied to both kinds of nodes. Particularly, it is relatively easy to implement ”register promotion” (the paper authors call it ”indirect removal”).

Of course the effectiveness of optimizations applied to indirect operations depends on the quality of alias analysis: if the analysis is poor the compiler will be forced to ”play safe”, and in practice the values will have ”zero version” most of the time, so few or no optimizations will be applied to them. On the other hand, for

all indirect operations where the alias analyzer determines that there are no interferences caused by aliasing all optimizations can happen naturally, like in the scalar case.

(explain the tradeoff of ”factored HSSA”, which is also explained in the gcc chapter...)

Author: Mantione

CHAPTER 11

Extended ssa numbering *???*

Progress: 0%

Material gathering in progress

11.1 TODO

CHAPTER 12

Static Single Information Form

A. Tavares

Progress: 20%

Material gathering in progress

Talk about extended-SSA, SSI that both correspond to do additional live-range splitting.

CHAPTER 13

Psi-SSA Form

F. de Ferrière

Progress: 50%

Review in progress

13.1 Overview

In the SSA representation, each definition of a variable is given a unique name, and new pseudo definitions are introduced on ϕ instructions to merge values coming from different control-flow paths. In this representation, each definition is an unconditional definition, and the value of a variable is the value of the expression on the unique assignment to this variable. This essential property of the SSA representation does not any longer hold when definitions may be conditionally executed. When the definition for a variable is a predicated operation, the value of the variable will or will not be modified depending on the value of a guard register. As a result, the value of the variable after the predicated operation is either the value of the expression on the assignment if the predicate is true, or the value the variable had before this operation if the predicate is false. We need a way to express these conditional definitions whilst keeping the static single assignment property.

13.2 Definition

1 page

The use of predicated operations allows to remove control-flow instructions and have instead straight line code. The compiler can perform such a transforma-

tion, which is called an if-conversion optimization [**?**, **?**]. A simple example of if-conversion is given in figure 13.1. We use the notation `p? <exp>` to say that `<exp>` is executed only if the predicate `p` is TRUE.

```
if(p)
   a = op1;        p? a = op1;
else
   b = op2;        p̄? b = op2;
x = Phi(a, b)         x = Psi(p?a, p̄?b)
```

Fig. 13.1 ψ-SSA representation

The SSA representation introduces ϕ operations at control-flow merge points. Each argument of a ϕ operation flows from a different incoming edge.

The ψ-SSA representation adds ψ operations. ψ operations are for predicated definitions what ϕ operations are for definitions on different control-flow edges. A ψ operation merges values that are defined under different predicates, and defines a single variable to represent these different values. A ψ operation is equivalent to a ϕ operation on which all the incoming edges would have been merged into a single execution path. Each argument of a ψ operation is now defined on a different predicate.

In figure 13.1, variables `a` and `b` were originally the same variable. On the left-hand side, the SSA construction renamed the two definitions of this unique variable into two different names, and introduced a new variable `x` defined by a ϕ operation to merge the two values coming from the different control-flow paths. On the right-hand side, an if-conversion algorithm transformed this code to remove the control-flow edges. It introduced predicated operations for the definitions of the variables `a` and `b` and turned the ϕ operation into a ψ operation. Each argument of the ψ operation is defined by a predicated operation. The intersection of the domain of the two predicates is empty and the value of the ψ operation is given by one or the other of its arguments, depending on the value of the predicate.

The ψ operations can also represent cases where variables are defined on predicates that are computed from independent conditions. This is illustrated in figure 13.2, where the predicates `p` and `q` are independent. During the SSA construction a unique variable was renamed into the variables `a`, `b` and `c` and the variables `x` and `y` were introduced to merge values coming from different control-flow paths. In the non-predicated code, there is a control-dependency between `x` and `c`, which means the definition of `c` must be executed after the value for `x` has been computed. In the predicated form of this example, there are no longer any control dependencies between the definitions of `a`, `b` and `c`. A compiler transformation can now freely move these definitions independently of each other, which may allow more optimizations to be performed on this code. However, the semantics of the original code requires that the definition of `c` occurs after the definitions of `a` and `b`. The order of the arguments in a ψ operation gives information on the original order of the definitions. We take the convention that the order of the arguments in a ψ operation is, from

left to right, equal to the original order of their definitions, from top to bottom, in the control-flow dominance tree of the program in a non-SSA representation. This information is needed to maintain the correct semantics of the code during transformations of the ψ-SSA representation and to revert the code back to a non ψ-SSA representation.

```
if(p)
   a = 1;          p? a = 1;
else
   b = −1;         p̄? b = −1;
x = Phi(a, b)         x = Psi(p?a, p̄?b)
if(q)
   c = 0;          q? c = 0;
y = Phi(x, c)         y = Psi(p?a, p̄?b, q?c)
```

Fig. 13.2 ψ-SSA with non-disjoint predicates

13.3 Construction

0.5 page

The construction of the ψ-SSA representation is a small modification on the standard algorithm to built an SSA representation.

Only the SSA renaming part of the algorithm needs to be modified. During the SSA renaming phase, basic blocks are processed in their dominance order, and operations in each basic block are scanned from top to bottom. On an operation, for each predicated definition of a variable, a new ψ instruction must be inserted just after the operation. For the definition of a variable `x` under predicate `p`, the ψ operation will take the form `x = Psi(p?`x_1`, p?`x`)`, where x_1 is the current renaming of `x` before the definition, and `p` is the predicate used on the definition of x_1. Once this instruction is inserted, the normal renaming of the operation proceeds, renaming x into a new name x_2. When the renaming of the operation is completed, the algorithm continues on the next instruction, which will be a ψ operation if there was a predicated definition. The first argument of the ψ operation is already renamed and thus is not modified. The second argument is just renamed into the current renaming for `x` which is x_2. On the definition of the ψ operation, the variable `x` is given a new name x_3 which becomes the renaming for further references to the `x` variable.

ψ operations can also be introduced in an SSA representation by applying an if-conversion transformation, such as the one that is described in **??**. Local transformations on control-flow patterns can also require to replace ϕ operations by ψ operations.

13.4 SSA algorithms

1 page

With this definition of the ψ-SSA representation, conditional definitions on predicated code are now replaced by unconditional definitions on ψ operations. Usual algorithms that perform optimizations or transformations on the SSA representation can now be easily adapted to the ψ-SSA representation, without compromising the efficiency of the transformations performed. Actually, within the ψ-SSA representation, predicated definitions behave exactly the same as non predicated ones for optimizations on the SSA representation. Only the ψ operations have to be treated in a specific way. As an example, the constant propagation algorithm described in [**?**] can be easily adapted to the ψ-SSA representation. In this algorithm, the only modification is that ψ operations have to be handled with the same rules as the ϕ operations. Other algorithms such as dead code elimination [**?**], global value numbering [**?**], partial redundancy elimination [**?**], and induction variable analysis [**?**] are examples of algorithm that can easily be adapted to this representation.

13.5 Psi-SSA algorithms

3 page

In addition to standard algorithms that can be applied to ψ operations and predicated code, a number of additional transformations can be performed on the ψ operations : ψ-inlining, ψ-reduction and ψ-projection.

ψ-inlining will recursively replace in a ψ operation an argument that is defined on another ψ operation by the arguments of this other ψ operation.

ψ-reduction will remove from a ψ operation an argument whose value will always be overridden by arguments on its right in the argument list, because the domain of the predicate associated with this argument is included in the union of the domains of the predicates associated with the arguments on its right.

ψ-projection will create from a ψ operation new ψ operations for uses in operations guarded by different predicates. Each new ψ operation is created as the projection on a given predicate of the original ψ operation. In this new ψ operation, arguments whose associated predicate has a domain that is disjoint with the domain of the predicate on which the projection is performed actually contribute no value to the ψ operation and thus are removed.

The ψ-SSA representation can also be used on a partially predicated architecture, where only a subset of the instruction set supports a predicate operand. The only impact of partial predication on the ψ-SSA representation is that when a ψ operation is created as a replacement for a ϕ operation, during if-conversion for example, some of its arguments may be defined by operations that cannot be predicated. In this case, the only constraint is that these non-predicated arguments can be safely speculated,

which means executed under some conditions on which they would not have been executed otherwise. Although these definitions are speculated, their values are only meaningful under a given predicate that must be kept in the ψ operation.

```
if(p)
   a = ADD i, 1;              a = ADD i, 1;
else
   b = ADD i, 2;              b = ADD i, 2;
x = Phi(a, b)                 x = Psi(p?a, p̄?b)
```

a) before if – conversion **b) Psi operation**

Fig. 13.3 Psi-SSA for partial predication

Figure 13.3 shows an example where some code with control-flow edges was transformed into a linear sequence of instructions. In this example, the ADD operation cannot be predicated. The information represented in the ϕ operation by the control-flow edges is now present in the ψ operation by means of predicates.

However, even if the ADD operation can be predicated, it can be profitable to perform a predicate promotion optimization to reduce the number of computed predicate registers. Using the representation in figure 13.3 b), there can be one predicate associated with the definition of a variable, and there will be one predicate associated with the use of the variable in a ψ operation. The predicate associated with an argument in a ψ operation can be promoted, without changing the semantics of the ψ operation. By predicate promotion, we mean that a predicate can be replaced by a predicate with a larger predicate domain. This promotion must obey the two following conditions so that the semantics of the ψ operation after the transformation is valid and unchanged.

- **Condition 1** For an argument in a ψ operation, the domain of the predicate used on the definition of this argument must contain the domain of the new predicate associated with this argument.
  for the instructions
  p? x = ...
  y = Psi(..., q?x, ...)
  then
  $q \subseteq p$
- **Condition 2** For an argument in a ψ operation, the domain of the new predicate associated with it can be extended up to include the domains of the predicates associated with arguments in the ψ operation that were defined after the definition for this argument in the original program.

for an instruction
$\mathtt{y} = \mathtt{Psi}(\mathtt{p}_1?\mathtt{x}_1, \mathtt{p}_2?\mathtt{x}_2, ..., \mathtt{p}_i?\mathtt{x}_i, ..., \mathtt{p}_n?\mathtt{x}_n)$
transformed to
$\mathtt{y} = \mathtt{Psi}(\mathtt{p}_1?\mathtt{x}_1, \mathtt{p}_2?\mathtt{x}_2, ..., \mathtt{p}'_i?\mathtt{x}_i, ..., \mathtt{p}_n?\mathtt{x}_n)$
then
$\mathtt{p}'_i \subseteq \bigcup_{k=i}^{n} \mathtt{p}_k$

This ψ-predicate promotion transformation allows to reduce the number of predicates that need to be computed, and to reduce the dependencies between predicate computations and conditional operations. In fact, the first argument of a ψ operation can usually be promoted under the TRUE predicate, provided that speculation can be applied. Also, when disjoint conditions are computed, one of them can be promoted to include the other conditions, usually reducing the dependency height of the predicated expressions. The ψ-predicate promotion transformation can be applied during an if-conversion algorithm for example. A side effect of this transformation is that it may increase the number of copy instructions to be generated during the out of ψ-SSA phase, because of more live-range interference between arguments in a ψ operation, as will be explained next.

13.6 Out of Psi-SSA

3 page

The out-of-SSA phase reverts an SSA representation into a non-SSA representation. This phase must be adapted to the ψ-SSA representation. The algorithm we present here is derived from the out of SSA algorithm from Sreedhar et al. [**?**].

This algorithm uses ψ congruence classes to create a conventional ψ-SSA representation. We define the conventional ψ-SSA (*ψ-CSSA*) form in a similar way to the Sreedhar definition of the conventional SSA (*CSSA*) form. The congruence relation is extended to the ψ operations. Two variables x and y are in a ψ congruence relation if they are referenced in the same ϕ or ψ function, or if there exists a variable z such that x is in a ψ congruence relation with z and y is in a ψ congruence relation with z. Then we define a ψ congruence class as the transitive closure of the ψ congruence relation. The property of the ψ-CSSA form is that the renaming into a single variable of all variables that belong to the same ψ congruence class, and the removal of the ψ and ϕ operations, results in a program with the same semantics as the original program.

Now, look at figure 13.4 to examine the transformations that must be performed to convert a program from a ψ-SSA form into a program in ψ-CSSA form.

Looking at the first example, the dominance order of the definitions for the variables a and b differs from their order from left to right in the ψ operation. Such code may appear after a code motion algorithm has moved the definitions for a and b relatively to each other. We said that the semantics of a ψ operation is dependent on the order of its arguments, and that the order of the arguments in a ψ operation

```
p? b = ...
   a = ...

   x = Psi(1?a, p?b)
```

Psi – SSA form

```
p? b = ...
   a = ...
p? c = b
   x = Psi(1?a, p?c)
```

Psi – CSSA form

```
p? b = ...
   x = ...
p? x = b
```

non – SSA form

```
   a = ...

p? b = ...
q? c = ...
   x = Psi(1?a, p?b)
   y = Psi(1?a, q?c)
```

Psi – SSA form

```
   a = ...
   d = a
p? b = ...
q? c = ...
   x = Psi(1?a, p?b)
   y = Psi(1?d, q?c)
```

Psi – CSSA form

```
   x = ...
   y = x
p? x = ...
q? y = ...
```

non – SSA form

Fig. 13.4 ψ-SSA and ψ-CSSA forms

is the order of their definitions in the dominance tree in the original program. In this example the renaming of the variables a, b and x into a single variable will not preserve the semantics of the original program. The order in which the definitions of the variables a, b and x occur must be corrected. This is done through the introduction of the variable c that is defined as a copy of the variable b, and is inserted after the definition of a. Now, the renaming of the variables a, c and x into a single variable will result in the correct semantics.

In the second example, the renaming of the variables a, b, c, x and y into a single variable will not give the correct semantics. In fact, the value of a used in the second ψ operation would be overridden by the definition of b before the definition of the variable c. Such code will occur after copy folding has been applied on a ψ-SSA representation. We see that the value of a has to be preserved before the definition of b, resulting in the code given for the ψ-CSSA representation. Now, the variables a, b and x can be renamed into a single variable, and the variables d, c and y will be renamed in another variable, resulting in a program in a non-SSA form with the correct semantics.

We will now present an algorithm that will transform a program from a ψ-SSA form into its ψ-CSSA form. This algorithm is made of three parts.

- **ψ-normalize** This part will put all ψ operations in what we call a *normalized* form.
- **ψ-congruence** This part will grow ψ-congruence classes from ψ operations, and will introduce repair code where needed.
- **ϕ-congruence** This part will extend the ψ-congruence classes with ϕ operations. This part is very similar to the Sreedhar algorithm.

We detail now the implementation of each of these three parts.

13.6.1 Psi-normalize

We define the notion of *normalized-ψ*. When ψ operations are created during the construction of the ψ-SSA representation, they are naturally built in their normalized form. The normalized form of a ψ operation has two characteristics:

- The predicate associated with each argument in a normalized-ψ operation is equal to the predicate used on the unique definition of this argument.
- The order of the arguments in a normalized-ψ operation is, from left to right, equal to the order of their definitions, from top to bottom, in the control-flow dominance tree.

When transformations are applied to the ψ-SSA representation, predicated definitions may be moved relatively to each others. Operation speculation and copy folding may enlarge the domain of the predicate used on the definition of a variable. These transformations may cause some ψ operations to be in a non-normalized form.

PSI-normalize implementation.

A dominator tree must be available for the control-flow graph to lookup the dominance relation between basic blocks. The dominance relation between two operations in a same basic block will be given by their relative positions in the basic block.

Each ψ operation is processed independently. An analysis of the ψ operations in a top down traversal of the dominator tree reduces the amount of repair code that is inserted during this pass. We only detail the algorithm for such a traversal.

For a ψ operation, the argument list is processed from left to right. For each argument arg_i, the predicate associated with this argument in the ψ operation and the predicate used on the definition of this argument are compared. If they are not equal, a new variable is introduced and is initialized just below the definition for arg_i with a copy of arg_i. This definition is predicated with the predicate associated with arg_i in the ψ operation. Then, arg_i is replaced by this new variable in the ψ operation.

Then, we consider the dominance order of the definition for arg_i, with the definition of the next argument in the ψ argument list, arg_{i+1}. When arg_{i+1} is defined on a ψ operation, we recursively look for the definition of the first argument of this ψ operation, until a non-ψ operation is found. Now, if the definition we found for arg_{i+1} dominates the definition for arg_i, repair code is needed. A new variable is created for this repair. This variable is initialized with a copy of arg_{i+1}, guarded by the predicate associated with this argument in the ψ operation. This copy operation is inserted at the lowest point, either after the definition of arg_i or arg_{i+1}[1]. Then, arg_{i+1} is replaced in the ψ operation by this new variable.

[1] When arg_{i+1} is defined by a ψ operation, its definition may appear after the definition for arg_i, although the non-ψ definition for arg_{i+1} appears before the definition for arg_i.

The algorithm continues with the argument arg_{i+1}, until all arguments of the ψ operation are processed. When all arguments are processed, the ψ is in its normalized form. When all ψ operations are processed, the function will contain only normalized-ψ operations.

The top-down traversal of the dominator tree will ensure that when a variable in a ψ operation is defined by another ψ operation, this ψ operation has already been analyzed and put in its normalized form. Thus the definition of its first variable already dominates the definitions for the other arguments of the ψ operation.

13.6.2 Psi-congruence

In this pass, we repair the ψ operations when variables cannot be put into the same congruence class, because their live ranges interfere. In the same way as Sreedhar gives a definition of the liveness on the ϕ operation, we first give a definition for the liveness on ψ operations. With this definition of liveness, an interference graph is built.

Liveness and interferences in Psi-SSA.

We have already seen that in some cases, repair code is needed so that the arguments and definition of a ψ operation can be renamed into a single name. We first give a definition of the liveness on ψ operations such that these cases can be easily and accurately detected by observing that live-ranges for variables in a ψ operation overlap.

Consider the code in figure 13.5. The ψ operation has been replaced by explicit `select` operations on each predicated definition. In this example, there is no relation between predicates `p` and `q`. Each of these `select` operations makes an explicit use of the variable immediately to its left in the argument list of the original ψ operation. We can see that a renaming of the variables `a`, `b`, `c` and `x` into a single representative name will still compute the same value for the variable `x`. Note that this transformation can only be performed on normalized ψ operations, since the definition of an argument must be dominated by the definition of the argument immediately to its left in the argument list of the ψ operation. Using this equivalent representation for the ψ operation, we now give a definition of the liveness for the ψ operations.

Definition *We say that the point of use of an argument in a normalized ψ operation occurs at the point of definition of the argument immediately to its right in the argument list of the ψ operation. For the last argument of the ψ operation, the point of use occurs at the ψ operation itself.*

Given this definition of liveness on ψ operations, and using the definition of liveness for ϕ operations given by Sreedhar, a traditional liveness analysis can be run.

```
   a = op1                a = op1
p? b = op2                b = p ? op2 : a
q? c = op3                c = q ? op3 : b
   x = Psi(1?a, p?b, q?c) x = c
```

a) Psi-SSA form **b) select form**

Fig. 13.5 ψ and select operations equivalence

Then an interference graph can be built to collect the interferences between variables involved in ψ or ϕ operations.

Repairing interferences on ψ operations.

We now present an algorithm that creates congruence classes with ψ operations such that there are no interference between two variables in the same congruence class.

First, the congruence classes are initialized such that each variable in the ψ-SSA representation belongs to its own congruence class. Then, ψ operations are processed one at a time, in no specific order. Two arguments of a ψ operation interfere if at least one variable from the congruence class of the first argument interferes with at least one variable from the congruence class of the second argument. When there is an interference, the two ψ arguments are marked as needing a repair. When all pairs of arguments of the ψ operation are analyzed, repair code is inserted. For each argument in the ψ operation that needs a repair, a new variable is introduced. This new variable is initialized with a predicated copy of the argument's variable. The copy operation is inserted just below the definition of the argument's variable, predicated with the predicate associated with the argument in the ψ operation.

Once a ψ operation has been processed, the interference graph must be updated, so that other ψ operations are correctly handled. Interferences for the newly introduced variables must be added to the interference graph. Conservatively, we can say that each new variable interferes with all the variables that the original variable interfered with, except those variables that are now in its congruence class. Also, conservatively, we can say that the original variable interferes with the new variable in order to avoid a merge of a later ψ or ϕ operation of the two congruence classes these two variables belong to. The conservative update of the interference graph may increase the number of copies generated during the conversion to the ψ-CSSA form.

Consider the code in figure 13.6 to see how this algorithm works. The definition of liveness on the ψ operation will create a live-range for variable `a` that extends down to the definition of `b`, but not further down. Thus, the variable `a` does not interfere with the variables `b`, `c` or `x`. The live-range for variable `b` extends down to its use in the definition of variable `d`. This live-range creates an interference with the variables `c` and `x`. Thus variables `b`, `c` and `x` cannot be put into the same congruence class. These variables are renamed respectively into variables `e`, `f` and `g`

and initialized with predicated copies. These copies are inserted respectively after the definitions for b, c and x. Variables a, e, f and g can now be put into the same congruence class, and will be renamed later into a unique representative name.

```
p? a = ...                  p? a = ...
q? b = ...                  q? b = ...
                            q? e = b
r? c = ...                  r? c = ...
                            r? f = c
   x = Psi(p?a, q?b, r?c)     g = Psi(p?a, q?e, r?f)
                              x = g
s? d = b + 1                s? d = b + 1
```

Fig. 13.6 Elimination of ψ live-interference

13.6.3 Phi-congruence

When all ψ operations are processed, the congruence classes built from ψ operations are extended to include the variables in ϕ operations. In this part, the algorithm from Sreedhar is used, with a few modifications.

The first modification is that the congruence classes must not be initialized at the beginning of this process. They have already been initialized at the beginning of the ψ-congruence step, and were extended during the processing of ψ operations. These congruence classes will be extended now with ϕ operations during this step.

The second modification is that the live-analysis run for this part must also take into account the special liveness rule on the ψ operations. The reason for this is that for any two variables in the same congruence class, any interference, either on a ψ or on a ϕ operation, will not preserve the correct semantics if the variables are renamed into a representative name.

All other parts of the out-of-SSA algorithm from Sreedhar are unchanged, and in particular, any of the three algorithms described for the conversion into a CSSA form can be used.

We have described a complete algorithm to convert a ψ-SSA representation into a ψ-CSSA representation. The final step to convert the code into a non-SSA form is a simple renaming of all the variables in the same congruence class into a representative name. The ψ and ϕ operations are then removed.

We now present some improvements that can be added so as to reduce the number of copies inserted by this algorithm.

13.6.4 Improvements to the out of Psi-SSA algorithm

Non-normalized ψ operations with disjoint predicates.

When two arguments in a ψ operation do not have their definitions correctly ordered, the ψ operation is not normalized. We presented an algorithm to restore the normalized property by adding a new predicated definition of a new variable. However, if we know that the predicate domains of the two arguments are actually disjoint, the semantics of the ψ operation is independent on their relative order. So, instead of adding repair code, these two arguments can simply be reordered in the ψ operation itself, to restore the normalized property.

Interference with disjoint predicates.

When the live-ranges of two variables overlap, an interference is added for these two variables in the interference graph. If the definitions for these variables are predicated definitions, their live-ranges are only valid under a specific predicate domain. These domains are the domains of the predicates used on the definitions of the variables. Then, if these domains are disjoint, then although the live-range overlap, they are on disjoint conditions and thus they do not create an interference in the interference graph. Removing this interference from the interference graph will avoid the need to add repair code when live-ranges on disjoint predicates overlap.

Repair interference on the left argument only.

When an interference is detected between two arguments in a ψ operation, only the argument on the left actually needs a repair. The reason is that, since the ψ operations are normalized, the definition of an argument is always dominated by the definition of an argument on its left. Thus adding a copy for the argument on the right will not remove the interference. However, the copy must now be put just before the definition of the next argument in the ψ operation, or just before the ψ operation if this is the last argument.

Interference with the result of a ψ operation.

When the live-range for an argument of a ψ operation overlaps with the live-range of the variable defined by the ψ operation, this interference can be ignored. Actually, there are two cases to consider:

- If the argument is not the last one in the ψ operation, and its live-range overlaps with the live-range of the definition of the ψ operation, then this live-range

also overlaps with the live-range of the last argument. Thus this interference will already be detected and repaired.

- If the argument is the last one of the ψ operation, then the value of the ψ operation is the value of this last argument, and this argument and the definition will be renamed into the same variable out of the SSA representation. Thus, there is no need to introduce a copy here.

CHAPTER 14

Graphs and gating functions

J. Stanier

Progress: 70%

Structural reordering in progress

14.1 Introduction

Many compilers represent the input program as some form of graph in order to aid analysis and transformation. A cornucopia of program graphs have been presented in the literature and implemented in real compilers. Therefore it comes as no surprise that a number of program graphs use SSA concepts as the core principle of their representation. These range from very literal translations of SSA into graph form to more abstract graphs which are implicitly SSA. We aim to introduce a selection of program graphs which use SSA concepts, and examine how they may be useful to a compiler writer.

One of the seminal graph representations is the Control Flow Graph (CFG), which was introduced by Allen to explicitly represent possible control paths in a program. Traditionally, the CFG is used to convert a program into SSA form. Additionally, representing a program in this way makes a number of operations simpler to perform, such as identifying loops, discovering irreducibility and performing interval analysis techniques.

The CFG models control flow, but many graphs model *data flow*. This is useful as a large number of compiler optimizations are based on data flow. The graphs we consider in this section are all data flow graphs, representing the data dependencies in a program. We will look at a number of different SSA-based graph representations. These range from those which are a very literal translation of SSA into a graph form to those which are more abstract in nature. An introduction to each graph will be given, along with diagrams to show how sample programs look when translated

into that particular graph. Additionally, we will touch on the literature describing a usage of a given graph with the application that it was used for.

14.2 Data Flow Graphs

The data flow graph (DFG) is also a directed graph $G = (V, E)$ except edges E represent the flow of data from the result of one operation to the input of another. An instruction executes once all of its input data values have been consumed. When an instruction executes it produces a new data value which is propagated to other connected instructions.

Whereas the CFG imposes a total ordering on instructions, the DFG has no such concept, nor does the DFG contain whole program information. Thus, target code cannot be generated directly from the DFG. The DFG can be seen as a companion to the CFG, and they can be generated alongside each other. With access to both graphs, optimisations such as dead code elimination, constant folding and common subexpression elimination can be performed effectively. However, keeping both graphs updated during optimisation can be costly and complicated.

14.3 The SSA Graph

We begin our exploration with a graph that is very similar to SSA: the SSA Graph. Many different variations exist in the literature, so we take ours from Cooper et al. An SSA Graph consists of vertices which represent operations (such as `add` and `load`) or ϕ-functions, and directed edges connect uses to definitions of values. The edges to a vertex represent the arguments required for that operation, and the edge from a vertex represents the propagation of that operation's result after it has been computed. This graph is therefore a *demand-based* representation. In order to compute a vertex, we must first *demand* the results of the operands and then perform the operation indicated on that vertex. The SSA Graph can be constructed from a program in SSA form by *explicitly* adding use-definition chains. We present some sample code in Figure 14.1 which is then translated into an SSA Graph. Note that there are no explicit nodes for variables in the graph. Instead, an operator node can be seen as the "location" of the value stored in a variable. We have annotated operators with variable names to show the correspondence between the operations in the graph and in the SSA form program.

The textual representation of SSA is much easier for a human to read. However, the primary benefit of representing the input program in this form is that the compiler writer is able to apply a wide array of graph-based optimizations by using standard graph traversal and transformation techniques. It is possible to augment the SSA Graph to model memory dependencies. This is achieved by adding additional

```
begin: a0 = 0;
       i0 = 0;
loop:  a1 = φ(a0,a2);
       i1 = φ(i0,i2);
       a2 = a1 * i1;
       i2 = i1 + 1;
       if i2 < 100 goto loop;
end:   a3 = φ(a1,a2);
       i3 = φ(i1,i2);
       print(a3 + i3);
```

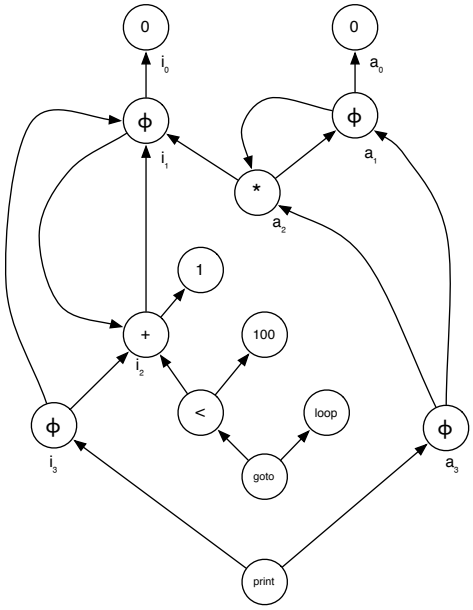


Fig. 14.1 Some SSA code translated into an SSA graph.

state edges that enforce an order of interpretation. These edges are extensively used in the Value State Dependence Graph, which we will look at later, after we touch on the concept of gating functions.

In the literature, the SSA Graph has been used to detect a variety of induction variables in loops, also for performing instruction selection techniques, operator strength reduction, rematerialization, and has been combined with an extended SSA language to aid compilation in a parallelizing compiler. The reader should note that the exact specification of what constitutes an SSA Graph changes from paper to paper. The essence of the IR has been presented here, as each author tends to make small modifications for their particular implementation.

14.3.1 Finding induction variables with the SSA Graph

Given that a program is represented as an SSA Graph, the task of finding induction variables is simplified. A *basic induction variable* is a variable that appears only in the form:

where `k` is a constant or loop invariant. Wolfe made the observation that each basic linear induction variable will belong to a nontrivial strongly connected region

```
i = 10
loop
     ...
     i = i + k
     ...
endloop
```

Fig. 14.2 `i` is a basic induction variable.

(SCR) in the SSA graph. There exist linear algorithms for discovering SCRs. Each such SCR must conform to the following constraints:

- The SCR contains only one ϕ-function at the header of the loop.
- The SCR contains only addition and subtraction operators, and the right operand of the subtraction is not part of the SCR (no `i = n − i` assignments).
- The other operand of each addition or subtraction is loop invariant.

This technique can be expanded to detect a variety of other classes of induction variables, such as wrap-around variables, non-linear induction variables and nested induction variables.

14.4 Program Dependence Graph

The Program Dependence Graph (PDG) represents both control and data dependencies together in one graph. The PDG was developed to aid optimisations requiring reordering of instructions and graph rewriting for parallelism, as the strict ordering of the CFG is relaxed and complemented by the presence of data dependence information. The PDG is a directed graph $G = (V, E)$ where nodes V are statements, predicate expressions or region nodes, and edges E represent either control or data dependencies. Thus, the set of all edges E has two distinct subsets: the control dependence subgraph E_C and the data dependence subgraph E_D. E_C can be cyclic if a loop is present in the program, since a loop in the PDG is defined by a control back edge forming an SCR. E_D is always acyclic, and can be seen as a series of data dependency DAGs for each basic block, which are then connected together based on the data flow through the program. Similar to the CFG, a PDG also has two nodes `ENTRY` and `EXIT`, through which control flow enters and exits the program respectively.

Statement nodes represent instructions in the program. Predicate nodes test a conditional statement and have true and false edges to represent the choice taken on evaluation of the predicate. Region nodes group all nodes with the same control dependencies together, and order them into a hierarchy. If the control dependence for a region node is satisfied, then it follows that all of its children can be executed. Thus, if a region node has three different control-independent statements

as immediate children, then these could potentially be executed in parallel. Diagrammatically, rectangular nodes represent statements, diamond nodes predicates, and circular nodes are region nodes. Filled edges represent control dependence, and dashed edges represent data dependence. Loops in the PDG are represented by back edges in the control dependence subgraph. We show example code translated into a PDG in Figure 14.3.

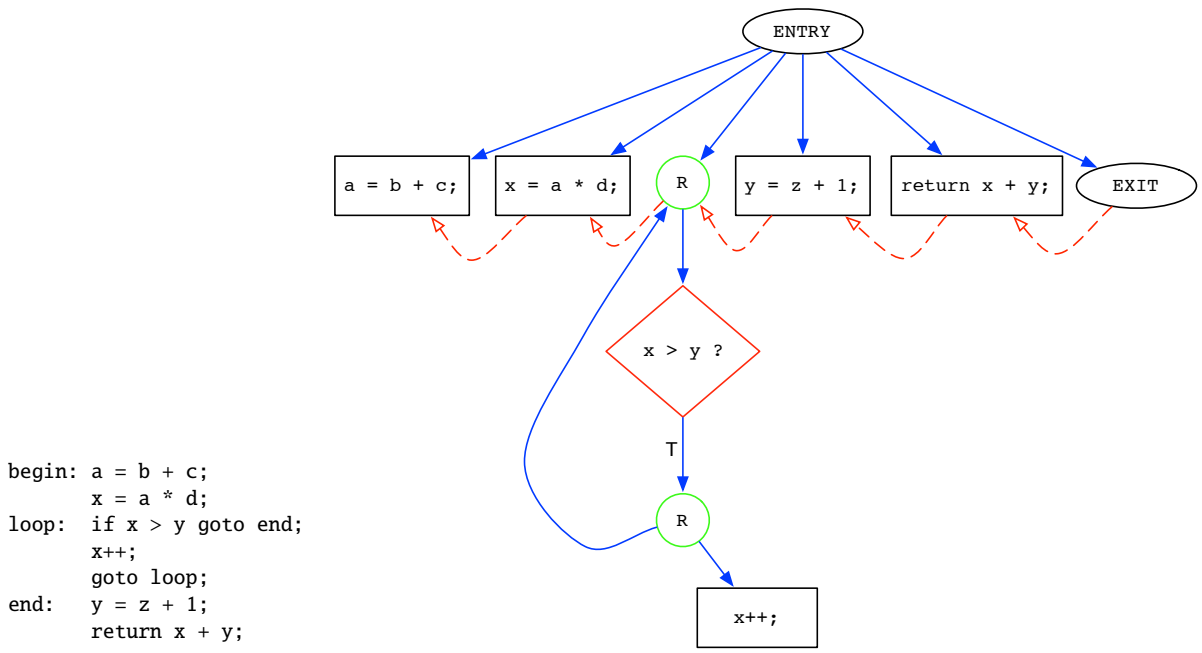


Fig. 14.3 Some code translated into a PDG.

Construction of the PDG is tackled in two steps from the CFG: construction of the control dependence subgraph and construction of the data dependence subgraph. Ferrante et al. construct the control dependence subgraph in $O(N^2)$ time. To compute the control dependence subgraph, the postdominator tree is constructed for the procedure. Then, the `ENTRY` node is added with one edge labelled T pointing to the CFG entry node, and another labelled F going to the CFG exit node. Then, let S consist of all edges (A, B) in the CFG such that B is not an ancestor of A in the postdominator tree. Each of these edges has an associated label T or F. Then, each edge in S is considered in turn. Given (A, B), the postdominator tree is traversed backwards from B until we reach A's parent, marking all nodes visited as control dependent on A.

Next, region nodes are added to the PDG. Each region node summarizes the set of control conditions and groups all nodes with the same set of control conditions

together. Region nodes are also inserted so that predicate nodes will only have two successors, thus hierarchically ordering the control dependencies. To begin with, an unpruned PDG is created by checking, for each node of the CFG, which control region or control dependence node it belongs to. This is done by traversing the postdominator tree in postorder, and using a hash table to map sets of control dependencies to region nodes. For each node N visited in the postdominator tree, the hash table is checked for an existing region node with the same set CD of control dependencies. If none exists, a new region node R is created with these control dependencies and entered into the hash table. R is made to be the only control dependence predecessor of N. Next, the intersection INT of CD is computed for each immediate child of N in the postdominator tree. If $INT = CD$ then the corresponding dependencies are deleted from the child and replaced with a single dependence on the child's control predecessor. Then, a pass over the graph is made to make sure that each predicate node has a unique successor for each truth value. If more than one exists, a region node is created and inserted as the successor of that particular predicate node's T or F edge.

The data dependence subgraph can be constructed after aliasing, procedure calls and side effects are analysed in the program. DAGs are constructed for each basic block during the initial parse of the source program. Each upwards exposed use in a block has a corresponding DAG leaf node as usual; these leaves are called *merge nodes*. Data flow analysis is then performed to compute the set of reaching definitions for each basic block, with the additional assumption that every variable is initially assigned the value "undefined" at program entry. Finally, the individual DAGs are connected to one another using the results of the data flow computation: edges are added from definition nodes to the corresponding merge nodes that may be reached.

Harrold et al. construct the PDG during parsing. Ferrante et al. considered the problem of generating linear code from a PDG, showing that generating the *minimal size* CFG from a PDG is an NP-Complete problem. Further work has improved on the efficiency and generality of generating sequential code. The PDG's structure has been exploited for generating code for vectorisation, and has also been used in order to perform accurate program slicing and testing.

14.4.1 Detecting parallelism with the PDG

The structure of the PDG allows for parallelism to be detected easily. In order to discover potential parallelism from the CFG, the if-conversion transformation must first be performed. However, the PDG can *directly* be used to detect parallelism. Any node in the PDG that is not contained in an SCR consisting of *both* control and data dependencies can be parallelized. In the example in Figure 14.4 we have a program containing a loop consisting of multiple assignment statements, and the corresponding PDG. Here, since the statements `d = e + f` and `g = h + i` (outlined in bold) in the loop do not form an SCR, so they are parallelizable. The statements `a = b`

`+ c` and `i = i + a` are not, since the assignment to `i` is data dependent on the assignment to `a`.

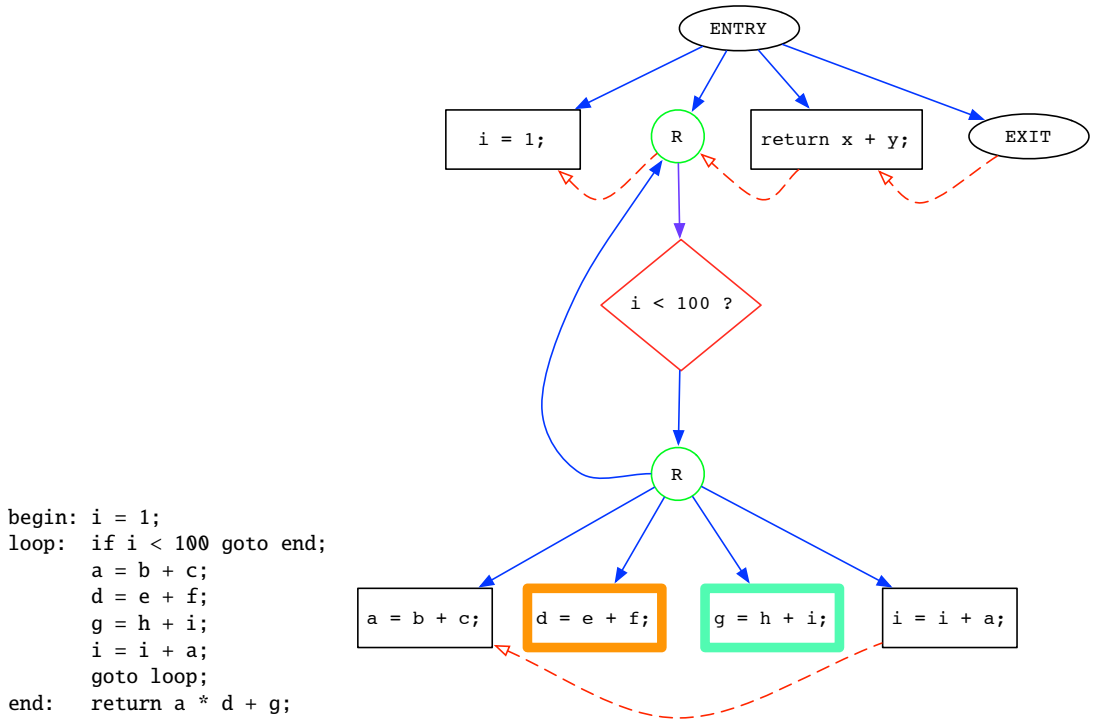


Fig. 14.4 Detecting parallelism with the PDG. The statements `d = e + f` and `g = h + i` (outlined in bold) are parallelizable.

14.5 Gating functions and GSA

In SSA form, ϕ-functions are used to identify points where variable definitions converge. However, they cannot be directly *interpreted*, as they do not specify the condition which determines which of the variable definitions to choose. By this logic, we cannot directly interpret the SSA Graph. Being able to interpret our IR is a useful property as it gives the compiler writer more information when implementing optimizations, and also reduces the complexity of performing code generation. Gated Single Assignment form (GSA; sometimes called Gated SSA) is an extension of SSA with *gating functions*. These gating functions are directly interpretable ver-

sions of ϕ-nodes, and replace ϕ-nodes in the representation. There are three forms of gating function and we take our definition from Tu and Padua:

- The γ function explicitly represents the condition which determines which ϕ value to select. A γ function is of the form $\gamma(P, V_1, V_2)$ where P is a predicate, and V_1 and V_2 are the values to be selected if the predicate evaluates to true or false respectively. This can be read simply as *if-then-else*.
- The μ function is inserted at loop headers to select the initial and loop carried values. A μ function is of the form $\mu(V_{init}, V_{iter})$, where V_{init} is the initial input value for the loop, and V_{iter} is the iterative input. We replace ϕ-functions at loop headers with μ functions.
- The η function determines the value of a variable when a loop terminates. An η function is of the form $\eta(P, V_{final})$ where P is a predicate and V_{final} is the definition reaching beyond the loop.

It is easiest to understand these gating functions by means of an example. Figure 14.5 shows how our earlier code in Figure 14.1 translates into GSA form. Here, we can see the use of both μ and η gating functions. At the header of our sample loop the ϕ-functions have been replaced by μ functions which determine between the initial and iterative values of `a` and `i`. After the loop has finished executing, the two η functions propagate the correct value from the corresponding μ function.

These gating functions are important as the concept will form components of the Value State Dependence Graph later. GSA has seen a number of uses in the literature. The original usage of GSA was by Ballance et al. as an intermediate stage in the construction of the Program Dependence Web IR. Havlak presented an algorithm for construction of a simpler version of GSA—Thinned GSA—which is constructed from a CFG in SSA form. GSA has been used for a number of analyses and transformations based on data flow.

Tu and Padua presented an algorithm that constructs SSA and GSA simultaneously in a single process. Construction of GSA proceeds from the CFG. It uses the concept of *path expressions*. A path expression P of type (u, v) is a simple regular expression over E such that every string in $\sigma(P)$ is a path from node u to node v (where $\sigma(P)$ represents the string generated by the regular expression P). These path expressions are represented as *gating functions*:

- The symbol Λ represents an unconditional edge.
- The symbol $\emptyset$ represents an edge not taken at a branch node.
- A γ expression $\gamma(P, e_1, e_2, \ldots, e_n)$ where only one e_i is Λ and all the other e's are $\emptyset$ represents the i's edge from an n-way branch statement with condition P.

Then, these gating functions can be simplified along gating paths. The full algorithm for construction and simplification of gating functions can be found in the Tu and Padua paper in the summary.

By using gating functions it becomes possible to construct IRs based solely on data dependencies. These IRs are sparse in nature compared to the CFG, making them good for analysis and transformation. This is also a more attractive proposition than generating and maintaining both a CFG and DFG, which can be complex and

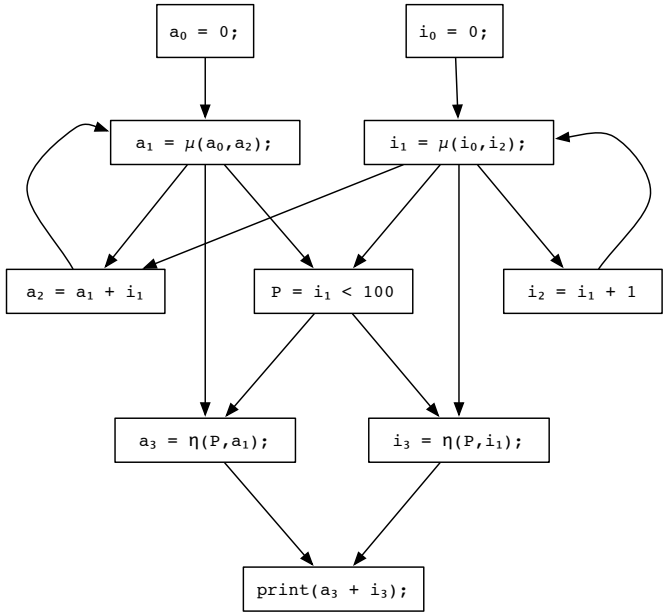


Fig. 14.5 A graph representation of our sample code in GSA form.

prone to human error. One approach has been to combine both of these into one representation, as is done in the PDG. Alternatively, we can utilize gating functions along with a data flow graph for an effective way of representing whole program information using data flow information.

14.5.1 Backwards symbolic analysis with GSA

GSA is useful for performing symbolic analysis. Traditionally, symbolic analysis is performed by forwards propagation of expressions through a program. However, complete forward substitution is expensive and can result in a large quantity of unused information and complicated expressions. Tu and Padua showed how *backwards*, demand-driven substitution can be performed using GSA which only substitutes *needed* information. Consider the following program, taken from the authors:

```
R: JMAX = Expr
S: if(P) then J = JMAX - 1
         else J = JMAX
T: assert(J ≤ JMAX)
```

Fig. 14.6 A program on which to perform symbolic analysis.

If forwards substitution were to be used in order to determine whether the assertion is correct, then the symbolic value of J must be discovered, starting at the top of the program in statement R. Forward propagation through this program results in statement T being $assert((if\ \mathtt{P}\ then\ \mathtt{Expr} - 1\ else\ \mathtt{Expr}) \leq \mathtt{Expr})$, thus the *assert* statement evaluates to true. In real, non-trivial programs, these expressions can get unnecessarily long and complicated.

Using GSA instead allows for backwards, demand-driven substitution. The program above has the following GSA form:

```
R: JMAX1 = Expr
S: if(P) then J1 = JMAX1 - 1
         else J2 = JMAX1
   J3 = γ(P,J1,J2)
T: assert(J3 ≤ JMAX1)
```

Fig. 14.7 Figure 14.6 in GSA form.

Using this backwards substitution technique, we start at statement T, and follow the SSA links of the variables from $\mathtt{J}_3$. This allows for skipping of any intermediate statements that do not affect variables in T. Thus the substitution steps are:

$$\begin{aligned} \mathtt{J}_3 &= \gamma(\mathtt{P},\mathtt{J}_1,\mathtt{J}_2) \\ &= \gamma(\mathtt{P},\mathtt{JMAX}_1 - 1,\mathtt{JMAX}_1) \end{aligned}$$

Fig. 14.8 Substitution steps in backwards symbolic analysis.

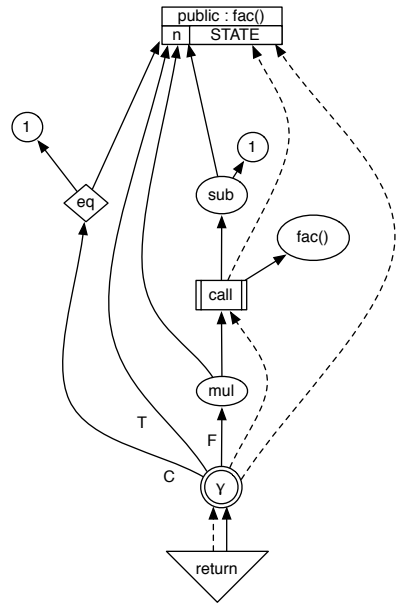
The backwards substitution then stops because enough information has been found, avoiding the redundant substitution of $\mathtt{JMAX}_1$ by Expr. In non-trivial programs this can greatly reduce the number of redundant substitutions, making symbolic analysis significantly cheaper.

14.6 Value State Dependence Graph

The gating functions defined in the previous section were used in the development of a sparse data flow graph IR called the Value State Dependence Graph (VSDG). The VSDG is a directed graph consisting of operation nodes, loop and merge nodes together with value and state dependency edges. Cycles are permitted but must satisfy various restrictions. A VSDG represents a single procedure: this matches the classical CFG.

An example VSDG is shown in Figure 14.9. In (a) we have the original C source for a recursive factorial function. The corresponding VSDG (b) shows both value and state edges and a selection of nodes.

```
int fac(int n) {
    int result;
    if(n == 1)
       result = n;
    else
       result = n * fac(n - 1);
    return result;
}
```



(a) (b)

Fig. 14.9 A recursive factorial function, whose VSDG illustrates the key graph components—value dependency edges (solid lines), state dependency edges (dashed lines), a `const` node, a `call` node, a γ-node, a conditional node,and the function entry and exit nodes.

14.6.1 Definition of the VSDG

A VSDG is a labelled directed graph $G = (N, E_V, E_S, \ell, N_0, N_\infty)$ consisting of nodes N (with unique entry node N_0 and exit node N_∞), value dependency edges $E_V \subseteq N \times N$, state dependency edges $E_S \subseteq N \times N$. The labelling function ℓ associates each node with an operator.

The VSDG corresponds to a reducible program, e.g. there are no cycles in the VSDG except those mediated by θ (loop) nodes.

Value dependency (E_V) indicates the flow of values between nodes. State dependency (E_S) represents two things; the first is essential sequential dependency required by the original program, e.g. a given `load` instruction may be required to follow a given `store` instruction without being re-ordered, and a `return` node in general must wait for an earlier loop to terminate even though there might be no value-dependency between the loop and the `return` node. The second purpose is that state dependency edges can be added incrementally until the VSDG corresponds to a unique CFG. Such state dependency edges are called *serializing* edges.

The VSDG is implicitly represented in SSA form: a given operator node, n, will have zero or more E_V-consumers using its value. Note that, in implementation terms,

a single register can hold the produced value for consumption at all consumers; it is therefore useful to talk about the idea of an output *port* for n being allocated a specific register, r, to abbreviate the idea of r being used for each edge (n_1, n_2) where $n_2 \in succ(n_1)$.

14.6.2 Nodes

There are four main classes of VSDG nodes: value nodes (representing pure arithmetic), γ-nodes (conditionals), θ-nodes (loops), and state nodes (side-effects). The majority of nodes in a VSDG generate a value based on some computation (add, subtract, etc) applied to their dependent values (constant nodes, which have no dependent nodes, are a special case).

14.6.3 γ-Nodes

The γ-node is similar to the γ gating function in being dependent on a control predicate, rather than the control-independent nature of SSA ϕ-functions.

A γ-node $\gamma(C, T, F)$ evaluates the condition dependency C, and returns the value of T if C is true, otherwise F.

We generally treat γ-nodes as single-valued nodes (contrast θ-nodes, which are treated as tuples), with the effect that two separate γ-nodes with the same condition can be later combined into a tuple using a single test. Figure 14.10 illustrates two γ-nodes that can be combined in this way. Here, we use a pair of values (2-tuple) of values for the T and F ports. We also see how two syntactically different programs can map to the same structure in the VSDG.

```
a) if (P)
x = 2, y = 3;
else
x = 4, y = 5;

b) if (P) x = 2; else x = 4;
...
if (P) y = 3; else y = 5;
```

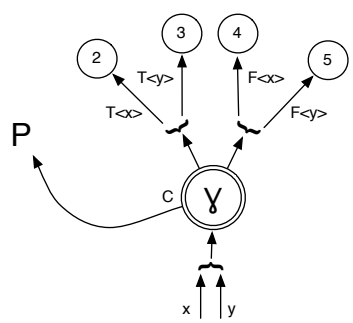


Fig. 14.10 Two different code schemes (a) & (b) map to the same γ-node structure.

14.6.4 θ-Nodes

The θ-node models the iterative behaviour of loops, modelling loop state with the notion of an *internal value* which may be updated on each iteration of the loop. It has five specific ports which represent dependencies at various stages of computation.

A θ-node $\theta(C, I, R, L, X)$ sets its internal value to initial value I then, while condition value C holds true, sets L to the current internal value and updates the internal value with the repeat value R. When C evaluates to false computation ceases and the last internal value is returned through the X port.

A loop which updates k variables will have: a single condition port C, initial-value ports $I_1, \ldots, I_k$, loop iteration ports $L_1, \ldots, L_k$, loop return ports $R_1, \ldots, R_k$, and loop exit ports $X_1, \ldots, X_k$. The example in Figure 14.11 also shows a pair (2-tuple) of values being used for I, R, L, X, one for each loop-variant value.

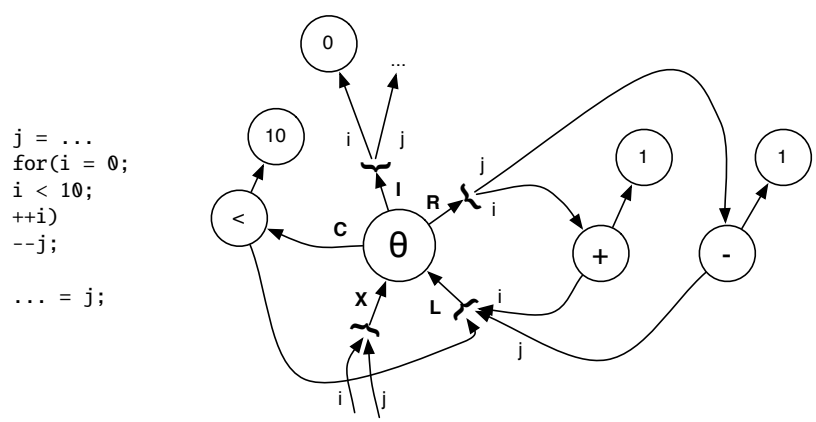


Fig. 14.11 An example showing a `for` loop. Evaluating the **X** port triggers it to evaluate the **I** value (outputting the value on the **L** port). While **C** evaluates to true, it evaluates the **R** value (which in this case also uses the θ-node's **L** value). When **C** is false, it returns the final internal value through the **X** port. As `i` is not used after the loop there is is no dependency on the `i` port of **X**.

The θ-node directly implements pre-test loops (`while`, `for`); post-test loops (`do...while`, `repeat...until`) are synthesised from a pre-test loop preceded by a duplicate of the loop body. At first this may seem to cause unnecessary duplication of code, but it has two important benefits: (*i*) it exposes the first loop body iteration to optimization in post-test loops (cf. loop-peeling), and (*ii*) it normalizes all loops to one loop structure, which both reduces the cost of optimization, and increases the likelihood of two schematically-dissimilar loops being isomorphic in the VSDG.

14.6.5 State Nodes

Loads and stores compute a value and state. The `call` node takes both the name of the function to call and a list of arguments, and returns a list of results; it is treated as a state node as the function body may read or update state.

We maintain the simplicity of the VSDG by imposing the restriction that *all* functions have *one* return node (the exit node N_∞), which returns at least one result (which will be a state value in the case of `void` functions). To ensure that function calls and definitions are able to be allocated registers easily, we suppose that the number of arguments to, and results from, a function is smaller than the number of physical registers—further arguments can be passed via a stack as usual.

Note also that the VSDG neither forces loop invariant code into nor out-of loop bodies, but rather allows later phases to determine, by adding serializing edges, such placement of loop invariant nodes for later phases.

14.6.6 Dead node elimination with the VSDG

By representing a program as a VSDG, many optimisations become trivial. For example, consider dead node elimination (Figure 14.12). This combines both dead code elimination and unreachable code elimination. Dead code generates VSDG nodes for which there is no value or state dependency path from the `return` node, i.e. the result of the function does not in any way depend on the results of the dead nodes. Unreachable code generates VSDG nodes that are either dead, or become dead after some other optimisation. Thus, a *dead node* is a node that is not post-dominated by the exit node N_∞. To perform dead node elimination, only two passes are required over the VSDG resulting in linear runtime complexity: one pass to identify all of the live nodes, and a second pass to delete the unmarked (i.e. dead) nodes. It is safe because all nodes which are deleted are guaranteed never to be reachable from the `return` node.

14.7 Summary

A compiler's intermediate representation can be a graph, and many different graphs exist in the literature. We can represent the control flow of a program as a Control Flow Graph (CFG) [**?**], where straight-line instructions are contained within basic blocks and edges show where the flow of control may be transferred to once leaving that block. A CFG is traditionally used to convert a program to SSA form [**?**]. We can also represent programs as a type of Data Flow Graph (DFG) [**?**, **?**], and SSA can be represented in this way as an SSA Graph [**?**]. An example was given that used the SSA Graph to detect a variety of induction variables in loops [**?**, **?**]. It has also

Input: A VSDG $G(N, E_V, E_S, N_\infty)$ with zero or more dead nodes.
Output: A VSDG with no dead nodes.

```
Procedure DNE(G) {
1:      WalkAndMark(N∞, G);
2:      DeleteMarked(G);
}
Procedure WalkAndMark(n, G) {
1:      if n is marked then finish;
2:      mark n;
3:      foreach (node m ∈ N ∧ (n, m) ∈ (E_V ∪ E_S)) do
4:          WalkAndMark(m);
}
Procedure DeleteMarked(G) {
1:      foreach (node n ∈ N) do
2:          if n is unmarked then delete(n);
}
```

Fig. 14.12 Dead node elimination on the VSDG.

been used for performing instruction selection techniques [**?**, **?**], operator strength reduction [**?**], rematerialization [**?**], and has been combined with an extended SSA language to aid compilation in a parallelizing compiler [**?**].

The Program Dependence Graph (PDG) [**?**] represents control and data dependencies in one graph. The PDG has been used for program slicing [**?**], testing [**?**], and widely for parallelization [**?**, **?**, **?**, **?**]. We showed an example of how the PDG directly exposes parallel code.

Gating functions can be used to create directly interpretable ϕ-functions. These are used in Gated Single Assignment Form [**?**, **?**, **?**]. An example was given of how to perform backwards demand-driven symbolic analysis using GSA [**?**].

We then described the Value State Dependence Graph (VSDG) [**?**], which is an improvement on a previous, unmentioned graph, the Value Dependence Graph [**?**]. It uses the concept of gating functions, data dependencies and state to model a program. We gave an example of how to perform dead node elimination on the VSDG. Detailed semantics of the VSDG are available [**?**], as well as semantics of a related IR: the Gated Data Dependence Graph [**?**]. Further study has taken place on the problem of generating code from the VSDG [**?**, **?**, **?**], and it has also been used to perform a combined register allocation and code motion algorithm [**?**].

Part III
Analysis

Progress: 27%

Progress: 61%

TODO: some throwaway remarks that never found a place

- what structural invariants does ssa provide for analysis (majumdar? ask palsberg to ask him)
- available expressions become available values
- split the analyses between algorithms that need kill calculation and those that dont?
- bidirectional (but dont discuss ssi)

CHAPTER 15

Introduction

A. Ertl

Progress: 0%

Material gathering in progress

. . . . . .

15.1 benefits of SSA

. . . . . .

15.2 properties: suffixed representation can use dense vector, vs DF needs (x,PP) needs hashtable

. . . . . .

15.3 phi node construction precomputes DF merge

. . . . . .

15.4 dominance properties imply that simple forward analysis works

. . . . . .

15.5 Flow insensitive somewhat improved by SSA

. . . . . .

15.6 building defuse chains: for each SSA variable can get pointer to next var

. . . . . .

15.7 improves SSA DF-propagation

CHAPTER 16

Propagating Information using SSA

F. Brandner
D. Novillo

Progress: 50%

Review in progress

16.1 Overview

A central task of compilers is to *optimize* a given input program such that the resulting code is more efficient in terms of execution time, code size, or some other metric of interest. However, in order to perform optimizations and program transformations typically some form of *program analysis* is required in order to determine if a given transformation is applicable, to estimate its profitability, or guarantee correctness.

Data flow analysis [**?**] is a simple, yet powerful, approach to program analysis that is utilized by many compiler frameworks and program analysis tools today. We will introduce the basic concepts of traditional data flow analysis in this chapter and show that *static single assignment* form (SSA) facilitates the design and implementation of equivalent analyses, while leveraging properties of programs in SSA form allows to reduce the compilation time and memory consumption.

Traditionally, data flow analysis is performed on a *control flow graph* (CFG) representation of the input program, where nodes in the graph represent operations and edges the possible flow of program execution. Information on certain *program properties* is propagated among the nodes along the control flow edges until the computed information at the nodes stabilizes, i.e., no *new* information can be devised from the program.

The *propagation engine* presented in the following sections is an extension of the well known approach by Wegman and Zadeck for *sparse conditional constant*

propagation [**?**] (also known as SSA-CCP). Instead of using the CFG they represent the input program as an *SSA graph* [**?**]. Operations are again represented as nodes in this graph, however, the edges represent *data dependencies* instead of control flow. This representation allows a selective propagation of program properties among data dependent graph nodes only. As before, the processing stops when the information at the graph's nodes stabilizes. The basic algorithm is not limited to constant propagation and can also be applied to solve a large class of other data flow problems efficiently [**?**]. However, not all data flow analyses can be modeled. We will thus investigate the limitations of the SSA-based approach and briefly discuss problems that cannot be modeled.

The remainder of this chapter is organized as follows. First, the basic concepts of (traditional) data flow analysis are presented in Section 16.2. This will provide the theoretical foundation and background for the discussion of the SSA-based propagation engine in Section 16.3. We then provide examples of data flow analyses that can be realized efficiently by the aforementioned engine, namely copy propagation in Section 16.4.1 and value range propagation in Section 16.4.2. Finally, some examples of data flow problems are given that cannot be modeled using the generic propagation engine but still benefit from SSA properties in Section 16.5. We conclude and provide links for further reading in Section 16.6.

16.2 Preliminaries

Data flow analysis is at the heart of many compiler transformations and optimizations, but also finds application in a broad spectrum of analysis and verification tasks in program analysis tools such as program checkers, profiling tools, and timing analysis tools. This section gives a brief introduction to the basics of data flow analysis, due to space considerations, we cannot cover this topic in full depth.

As noted before, data flow analysis allows to derive information of certain interesting program properties that may help to optimize the program. Typical examples of interesting properties are: The set of *live* variables at a given program point, the constant value a variable may take, or the set of program points that are reachable at run-time. Liveness information is critical, for example, during register allocation, while the two latter properties help in simplifying computations and avoiding useless calculations as well as dead code.

The analysis results are gathered from the input program by propagating information among its operations along all possible execution paths. The propagation is typically performed iteratively until the computed results stabilize. Formally, a data flow problem can be specified using a *monotone framework* that consists of:

- a *complete lattice* representing the property space,
- a *flow graph* resembling the control flow of the input program,
- and a set of *transfer functions* modeling the effect of individual operations on the property space.

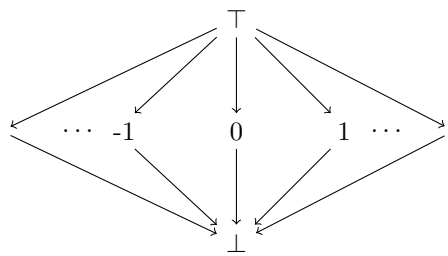


Fig. 16.1 Example of a lattice, which is commonly used to represent whether a variable is known to hold a constant value at a given program point.

16.2.1 Property Space

A key concept for data flow analysis is the representation of the property space via *partially ordered sets*. A partially ordered set $(L, \sqsubseteq)$ is a set L that is equipped with a reflexive, transitive, and anti-symmetric relation $\sqsubseteq\colon L \times L \rightarrow \{true, false\}$. Based on the relation an *upper bound* for subsets of L can be defined: $l \in L$ is the upper bound for a subset Y of L if $\forall l' \in Y\colon l' \sqsubseteq l$. Furthermore, $l \in L$ is the *least upper bound* if it is an upper bound and for any other upper bound l_o of Y: $l \sqsubseteq l_0$. Lower bounds and greatest lower bounds are defined analogously.

A particularly interesting class of partially ordered sets are *complete lattices*, where all subsets have a least upper bound as well as a greatest lower bound. These bounds are unique and are denoted by $\bigsqcup$ and $\bigsqcap$ respectively. In the context of program analysis the former is often referred to as the *join operator*, while the latter is termed the *meet operator*. Complete lattices have two distinguished elements, the *least element* and the *greatest element*, often denoted by $\bot$ and $\top$ respectively.

An *ascending chain* is a totally ordered subset $\{l_1, \ldots, l_n\}$ of a complete lattice, where for all $i, j \in \{1, \ldots, n\}, i < j\colon l_i \sqsubseteq l_j$. A chain is said to *stabilize* if there exists an $i_0 \in \mathbb{N}$, where $\forall i \in \mathbb{N}, i > i_0\colon l_i = l_{i_0}$; i_0 is then called the *length* of the chain.

Consider, for example, the lattice presented in Figure 16.1 that is commonly used to represent whether a given variable is known to hold a specific constant value at a given program point at run-time. The analysis and the corresponding optimization are often uniformly referred to as *constant propagation*. For this lattice, $\top$ indicates that no specific information on the variable's value is available. The symbol $\bot$, on the other hand, indicates that the analysis was not able to prove that the variable holds the same constant value in all cases. The other elements in the lattice denote that the variable is known to hold the respective value. The $\sqsubseteq$ relation is represented by the arrows in the figure. Clearly, all chains for this lattice are at most of length three ($\bot \sqsubseteq c \sqsubseteq \top, c \in \mathbb{N}$).

16.2.2 Program Representation

Individual functions of the input program are represented as separate flow graphs of the form $G = (V, E)$, where the nodes represent operations, or instructions, at a program point and edges denote the possible flow of control at run-time. The graph contains two distinguished nodes, the *start* node s and the *exit* node t. These two nodes are special in that the former is the only node without any predecessors and the latter is the only node without any successors.

During the analysis, data flow information is propagated from one node to another adjacent node along the respective graph edge. Every node is associated with data flow information using two sets, *in* and *out*. If the target node has a single incoming edge only, the propagation corresponds to a simple copy from the source node's *out* set to the target node's *in* set. However, in the more general case of multiple incoming edges, we need to combine the information from all those edges. This is accomplished by applying the join or meet operator of the lattice that represents the property space. For example, consider the lattice from Figure 16.1 and the flow graph shown in Figure 16.2. We can clearly say that, after executing operation 1 on the top left side, variable x holds the constant value of 4, as indicated by the *out* annotation $(x, 4)$ below. The same applies for operation 2 on the right, with the slight difference that the variable holds the value of 5. Combining this information yields $4 \sqcap 5 = \bot$, i.e., x does not hold a constant value at program point 3.

In many cases it is helpful to reverse the flow graph to propagate information, i.e., reverse the direction of the graph's edges. Analyses relying on this reversed flow graph are termed *backward analyses*, while those using the regular flow graph are called *forward analyses*. As shown before, computing whether a variable holds a certain constant value is a forward problem, while determining whether a computation is dead, i.e., not used, is a typical backward problem.

16.2.3 Transfer Functions

Aside from the control flow, the effect of the program's operations need to be accounted for during analysis. However, in contrast to the actual execution, we are only interested in an abstract model of the operations' effects with respect to the

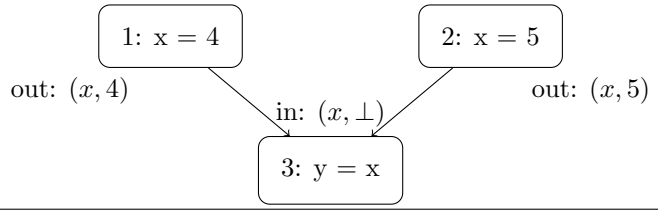


Fig. 16.2 Combining data flow information using the join operator.

data flow information. Operations are thus mapped to a set of abstract *transfer functions* of the form $f_{op}\colon L \to L$ that transform the information available from the *in* set of the respective flow graph nodes and store the result in the corresponding *out* set.

Continuing the example from before, a simple set of transfer functions can be defined, as shown in Table 16.1. Assignments to variables invalidate the information that is available on that particular variable, i.e., information is removed from the input set. However, in the case of a copy operation or the assignment of a constant value, new information is obtained, i.e., is appended to the input set. Care has to be taken that previous information is properly invalidated. Other operations not modifying any variable are associated with the identity function, i.e., all the information is preserved.

16.2.4 Solving Data Flow Problems

Putting all those elements together, i.e., a complete lattice, a flow graph, and a set of transfer functions, yields an instance of a monotone framework. Which, in fact, describes a set of *data flow equations* that need to be solved in order to retrieve the analysis result. A very popular and intuitive way of solving these equations is to compute the *maximal fixed point* (MFP) using an iterative work list algorithm. The work list contains operations that have to be reevaluated by first combining the information form all their predecessors in the flow graph, then applying their transfer function, and finally propagating the obtained information to all successors by appending them to the work list. The algorithm terminates when the data flow information stabilizes and the work list becomes empty [**?**].

It is obvious that a single flow edge can be appended several times to the work list in the course of the analysis. It may even happen that an infinite feedback loop prevents the algorithm from terminating. We are thus interested in bounding the number a given edge is processed. Recalling the definition of chains from Section 16.2.1, the *height* of a lattice is defined by the length of its longest chain. We can ensure termination for lattices fulfilling the *ascending chain condition*, which ensures that the lattice has finite height. Given a lattice with finite height h and a flow graph $G = (V, E)$ it is easy to see that the MFP solution can be computed in $O(|E| \cdot h)$ time, or, due to the fact that $|E| \leq |V|^2$, in $O(|V|^2 \cdot h)$. Note that the height of the lattice often depends on properties of the input program, which might ultimately

Operation	Transfer Function
x = c, $c \in \mathbb{N}$	$f_{x=c}(L) = L \setminus \{(x, c')\|(x, c') \in L\} \cup \{(x, c)\}$
x = y	$f_{x=y}(L) = L \setminus \{(x, c')\|(x, c') \in L\} \cup \{(x, c)\|(y, c) \in L\}$
x = ...	$f_{x=\ldots}(L) = L \setminus \{(x, c')\|(x, c') \in L\}$
...	$f_{\ldots}(L) = L$

Table 16.1 A simplified set of transfer functions for constant propagation analysis.

yield bounds worse than cubic in the number of graph nodes. Furthermore, every node of the flow graph is associated with an *in* and an *out* set, in order to store data flow information and propagate preliminary analysis results to the neighbors in the graph. The sets often contain information that is unrelated to the particular nodes in question in order to ensure that the information is properly propagated to all relevant program points.

For reducible flow graphs the order in which operations are processed by the work list algorithm can be optimized [**?**, **?**, **?**], allowing to derive tighter complexity bounds. However, relying on reducibility is problematic, because flow graphs are often *not* reducible even for proper structured languages, e.g., reversed flow graphs of backward problems can be, and in fact almost always are, irreducible even for programs with reducible flow graphs. Furthermore, experiments have shown that the tighter bounds not necessarily lead to improved compilation times [**?**].

Apart from computing a fixed point, data flow equations can also be solved using a more powerful approach called the *meet over all paths* (MOP) solution. Even though more powerful, computing the MOP solution is often harder to compute or even undecidable [**?**]. Consequently, the MFP solution is preferred in practice.

16.3 Propagation Using the SSA Form

SSA form allows to solve a large class of data flow problems more efficiently than the standard fixed point solution presented previously. The basic idea is to directly propagate information computed at the unique definition of a variable to all its uses. Intermediate program points on regular control flow paths from the definition to those uses are skipped, reducing memory consumption and compilation time.

16.3.1 Program Representation

Programs in SSA form exhibit, aside from the regular operations of the original program, special operations called ϕ-operations, which are placed at join points of the original CFG – see Part I of this book for more information. In the following, we assume that possible many ϕ-operations are associated with the corresponding CFG nodes at those join points.

Reference for SSA graphs: Point directly to a chapter in the book?

Data flow analysis under SSA form relies on a specialized program representation based on *SSA graphs* [**?**], which allows to simplify the propagation of data flow information. The nodes of an SSA graph correspond to the operations of the program. In particular the program's ϕ-operations are represented by dedicated nodes in the SSA graph. The edges of the graph connect the unique definition of a variable with all its uses, i.e., an edge represents a true data dependence among two nodes.

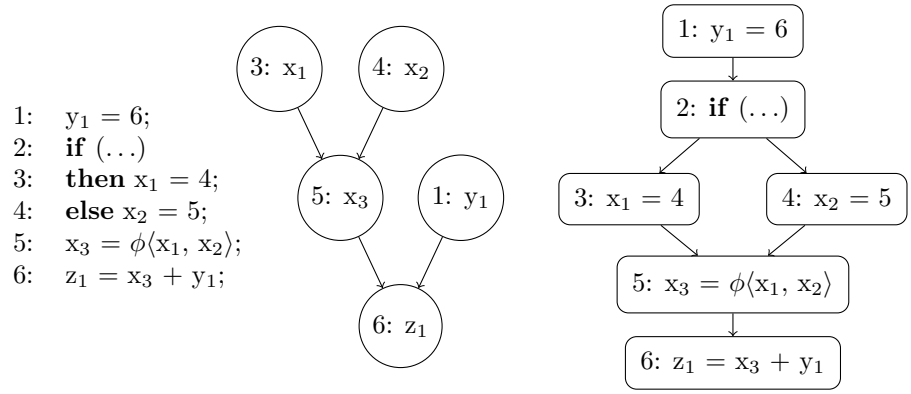


Fig. 16.3 Example program and its SSA graph.

An important property of SSA graphs is that they also capture, besides the data dependencies, the *relevant* join points of the program's CFG. A join point is relevant for the analysis, whenever the value of two or more definitions may reach a use by passing through that join. Due to the properties of SSA form, it is ensured that a ϕ-operation is placed at the join point and that the use has been properly updated to refer to the respective ϕ-operation. Other join points can safely be ignored, because uses are guaranteed to receive the same value on all paths due to the dominance property.

Consider for example, the code excerpt shown in Figure 16.3, along with the corresponding SSA graph and CFG. Assume we are interested in propagating information from the assignment of variable y_1 at the beginning of the code excerpt down to its unique use at the end. Using the traditional CFG representation this requires the propagation to pass though several intermediate program points. These program points are concerned with computations of the variables x_1 through x_3 only and are thus irrelevant for the computation at hand. The SSA graph representation, on the other hand, allows to propagate the desired information directly without any intermediate steps. At the same time, we also find that the control flow join following the **if** is properly represented by the ϕ-operation defining the variable x_3 in the SSA graph.

Even though the SSA graph captures data dependencies and the relevant join points in the CFG, it lacks information on other *control dependencies*. However, analysis results can often be improved significantly by considering the additional information that is available from the control dependencies in the program's CFG. As an example consider once more the code excerpt shown in Figure 16.3. Assume that the condition associated with the **if** operation is known to be false for all possible program executions. Consequently, the ϕ-operation will select the value of x_2 in all cases, which is known to be of constant value 5. However, due to the shortcom-

1. Initially, every edge in the CFG is marked not executable and the *FlowWorkList* is seeded with the outgoing edges of the flow graph's *start* node. The *SSAWorkList* is empty.
2. Remove the top element of either of the two work lists.
3. If the element is a flow edge that is marked to be executable, do nothing, otherwise proceed as follows:
   - Mark the edge as executable.
   - Visit every ϕ-operation associated with the edge's target node.
   - When the target node was reached the first time via the *FlowWorkList* visit its operation.
   - When the target node has a single outgoing non-executable edge append that edge to the *FlowWorkList*.
4. If the element is an edge from the SSA graph, process the target operation as follows:
   a. When the target operation is a ϕ-operation visit that ϕ-operation.
   b. For regular target operations, examine the corresponding executable flag of the incoming edges of its corresponding flow graph node. If any of those edges is marked executable visit the operation, otherwise do nothing.
5. Continue with step 2 until both work lists become empty.

Algorithm 4: Sparse Data Flow Propagation

ings of the SSA graph this information cannot be derived. It is thus important to use both graphs during data flow analysis in order to obtain the best possible results.

16.3.2 Sparse Data Flow Propagation

Similar to monotone frameworks for traditional data flow analysis, frameworks for *sparse data flow propagation* under SSA form can be defined. As before, such a framework consist of: (1) a complete lattice, (2) a set of transfer functions, (3) a flow graph, and, additionally, (4) an SSA graph. We again seek a maximal fixed point solution (MFP) using an iterative work list algorithm. However, in contrast to the algorithm described before, data flow information is not propagated along the edges of the flow graph but along the edges of the SSA graph. For regular uses the propagation is straightforward due to the fact that every use receives its value from a unique definition. Special care has to be taken for ϕ-operations, which select a value among their operands depending on the incoming flow edges. The data flow information of the incoming operands has to be combined using the meet or join operator of the lattice. As data flow information is propagated along SSA edges that have a single source, is is sufficient to store the data flow information with the node. The *in* and *out* sets used by the traditional approach – see Section 16.2.2 – are obsolete, since ϕ-operations already provide the required buffering. Furthermore, the flow graph is used in order to track which operations are not reachable under any program execution and thus can be ignored safely during the computation of the fixed point solution.

The algorithm processes two work lists, the *FlowWorkList* containing edges of the flow graph and the *SSAWorkList*, which consists of edges from the SSA graph.

1. Compute the operation's data flow information:
   a. If the operation is a ϕ-operation, combine the data flow information from all its operands where the corresponding flow edge is marked executable.
   b. In the case of conditional branches, update the operation's data flow information. Determine which of the outgoing flow edges are reachable from the corresponding flow graph node by examining the branch's condition(s) and append the respective non-executable edges to the *FlowWorkList*.
   c. For regular operations, update the corresponding data flow information by applying its transfer function.
2. Whenever the data flow information changes append all outgoing edges of the corresponding SSA graph node to the *SSAWorkList*.

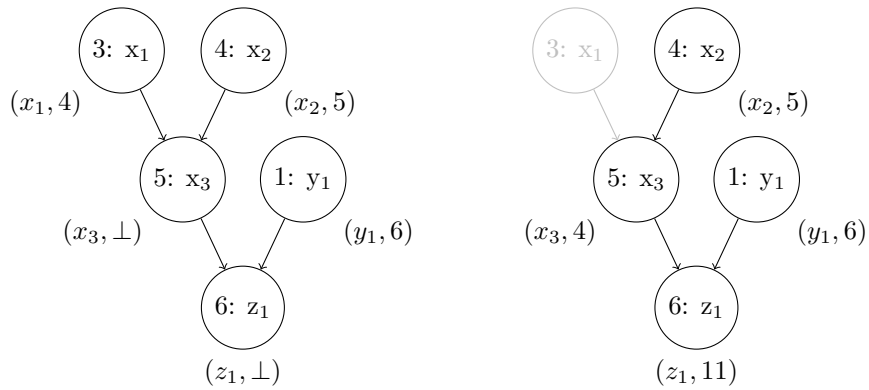
Algorithm 5: Visiting an Operation

It proceeds by removing the top element of either of those lists and processing the respective edge as indicated by Algorithm 4. Throughout the algorithm operations of the program may be visited in order to update the work lists and propagate information as shown by Algorithm 5. We will highlight the most relevant parts of the algorithms in the following paragraphs and explain some important aspects of their interaction.

In step 3 of the main algorithm, flow edges are processed that were encountered to be executable for the first time in the course of the analysis. Whenever such a flow edge is processed, all ϕ-operations of its target node need to be reevaluated due to the fact that Algorithm 5a discarded the respective operands of the ϕ-operations so far – because the flow edge was not yet marked executable. Similarly, the operation of the target node has to be evaluated when the target node is encountered to be executable for the first time, i.e., the currently processed edge is the first of its incoming edges that is marked executable. Note that this is only required the *first* time the node is encountered to be executable, due to the processing of operations in Step 4b, which thereafter triggers the reevaluation automatically when necessary.

Regular operations as well as ϕ-operations are visited by Algorithm 5 when the corresponding flow graph node has become executable or whenever the data flow information of one of their predecessors in the SSA graph changed. ϕ-operations combine the information from multiple flow paths using the usual meet operator, but only considering those operands where the associated edge is marked executable. Conditional branches are handled by examining its conditions based on the data flow information computed so far. Depending on whether those conditions are satisfiable or not, flow edges are appended to the *FlowWorkList* in order to ensure that all reachable operations are considered during the analysis. Finally, all regular operations are processed by applying the relevant transfer function and possibly propagating the updated information to all uses by appending the respective SSA graph edges to the *SSAWorkList*.

As an example, consider the program shown in Figure 16.3 and the constant propagation problem. First, assume that the condition of the **if** cannot be statically evaluated, we thus have to assume that all its successors in the CFG are reachable. Consequently, all flow edges in the program will eventually be marked executable.



(a) All code reachable. (b) With unreachable code.

Fig. 16.4 Sparse data flow propagation using SSA graphs.

This will trigger the evaluation of the constant assignments to the variables x_1, x_2, and y_1. The transfer functions immediately yield that the variables are all constant holding the value 4, 5, and 6 respectively. This new information will trigger the reevaluation of the ϕ-operation of variable x_3. As both of its operands are reachable the combined information yields $3 \sqcap 5 = \bot$. Finally, also the assignment to variable z_1 is reevaluated, but the analysis shows that its value is not a constant as depicted by Figure 16.4a. If, however, the condition of the **if** is known to be false for all possible program executions a more precise result can be computed, as shown in Figure 16.4b. Neither the flow edge leading to the assignment of variable x_1 is marked executable nor its outgoing edge leading to the ϕ-operation of variable x_3. Consequently, the reevaluation of the ϕ-operation considers the data flow information of its second operand x_2 only, which is known to be constant. This finally enables to analysis to show that the assignment to variable z_1 is, in fact, constant as well.

During the course of the propagation algorithm, every edge of the SSA graph is processed at least once, whenever the operation corresponding to its definition is found to be executable. Afterward, an edge can be revisited several times depending on the height h of the lattice representing the analysis' property space. Edges of the flow graph, on the other hand, are processed at most once. This leads to an upper bound in execution time of $O(|E_{SSA}| \cdot h + |E_{FG}|)$, where E_{SSA} and E_{FG} represent the edges of the SSA graph and the flow graph respectively – see [**?**]. The size of the SSA graph increases with respect to the original non-SSA program. Measurements indicate that this growth is linear [**?**], yielding a bound that is comparable to the bound of traditional data flow analysis. However, in practice the SSA-based propagation engine outperforms the traditional approach. This is due to the direct propagation from the definition of a variable to its uses, without the costly intermediate steps that have to be performed on the CFG. The overhead is also reduced in terms of memory consumption. Instead of the *in* and *out* sets capturing the complete

property space on every program point in the program, it is sufficient to associate every node in the SSA graph with the data flow information of the corresponding variable. This leads to considerable savings in practice.

16.3.3 Limitations

Unfortunately, the presented approach also has its drawbacks in terms of general applicability. The problem arises from two sources: (1) the exclusive propagation of information between data-dependent operations and (2) the semantics and placement of ϕ operations. The former issue prohibits the modeling of data flow problems that propagate information to program points that are not directly related to either a definition or a use of a variable, while the latter prohibits the modeling of backward problems.

Consider, for example, the well known problem of available expressions [**?**] that often occurs in the context of redundancy elimination. An expression is available at a given program point when the expression is computed and not modified thereafter on all paths leading to that program point. In particular, this might include program points that are independent from the expression and its operands, i.e., neither defines nor uses any of its operands. However, the SSA graph does not cover those program points as it propagates information directly from definitions to uses without any intermediate steps.

Furthermore, data flow analysis using SSA graphs is limited to forward problems. Due to the structure of the SSA graph it is not possible to simply reverse the edges in the graph as it is done with flow graphs. For one, this would invalidate the nice property of having a single source for incoming edges of a given variable, as variables typically have more than one use. In addition, ϕ-operations are placed at join points with respect to the *forward* control flow and thus do not capture join points in the reversed flow graph. SSA graphs are consequently not suited to model backward problems in general.

16.4 Examples

Even though data flow analysis based on SSA graphs has its limitations, it is still a useful and effective solution for many interesting problems, as will be shown in the following sections.

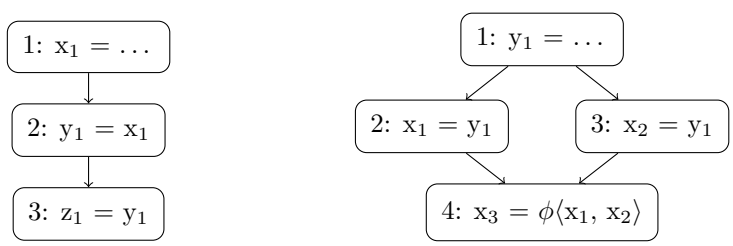
16.4.1 Copy Propagation

Copy propagation in SSA form is, in principle, very simple. Given the assignment $x = y$, all we need to do is traverse all the immediate uses of x and replace them with y, thereby effectively eliminating the original copy operation. However, such an approach will not be able to propagate copies past ϕ-operations, particularly those in loops. A more powerful approach is to split copy propagation into two phases. First, data flow analysis is performed in order to find copy-related variables throughout the program. Followed by a rewrite phase that eliminates spurious copies and renames variables.

The analysis for copy propagation can be described as the problem of propagating the *copy-of value* of variables. Given a sequence of copies as shown in Figure 16.5a. We say that y_1 is a *copy of* x_1 and z_1 is a *copy of* y_1. The problem with this representation is that there is no apparent link from z_1 to x_1. In order to handle transitive copy relations, all transfer functions operate on copy-of values instead of the direct source of the copy. If a variable is not found to be a copy of anything else, its copy-of value is the variable itself. For the example above, this yields that both, y_1 and z_1, are copies of x_1, which in turn is a copy of itself. The lattice of this data flow problem is thus similar to the lattice shown previously for constant propagation. However, the lattice elements correspond to variables of the program instead of integer numbers. The least element of the lattice represents the fact that a variable is a copy of itself.

Similarly, we would like to obtain the result that x_3 is a copy of y_1 for the example depicted in Figure 16.5b. This is accomplished by choosing the meet operator such that a copy relation is propagated whenever the copy-of values of all the ϕ-operation's operands match. So, when visiting the ϕ-operation for x_3, the analysis finds that x_1 and x_2 are both copies of y_1, which allows to propagate the desired result that x_3 is a copy of y_1.

The following example shows a more complex situation where copy relations are obfuscated by loops – see Figure 16.6. Note that the actual visiting order depends on the shape of the CFG and immediate uses, the ordering used here is meant for illustration only. Processing starts at the operation labeled 1, with both work lists empty and the data flow information $\top$ associated with all variables:



(a) Transitive copy relations. (b) Copy relations on ϕ-operations.

Fig. 16.5 Analysis of copy-related variables.

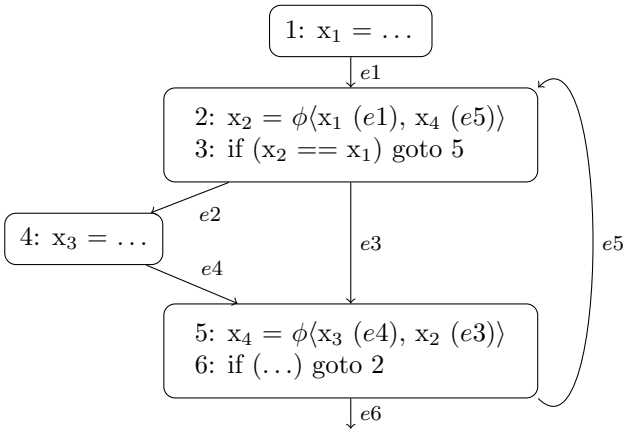


Fig. 16.6 ϕ-operations in loops often obfuscate copy relations.

1. Assuming that the value assigned to variable x_1 is not a copy, the data flow information for this variable is lowered to $\perp$, the SSA edges leading to operations 2 and 3 are appended to the *SSAWorkList*, and the flow graph edge $e1$ is appended to the *FlowWorkList*.
2. Processing the flow edge from the work list causes the edge to be marked executable and the operations labeled 2 and 3 to be visited. Since, edge $e5$ is not yet known to be executable the processing of the ϕ-operation yields a copy relation between x_2 and x_1. This information is utilized in order to determine which outgoing flow graph edges are executable for the conditional branch. Examining the condition shows that only edge $e3$ is reachable and thus needs to be added to the work list.
3. Flow edge $e3$ is processed next and marked executable for the first time. Furthermore, the ϕ-operation labeled 5 is visited. Due to the fact that edge $e4$ is not known the be executable, this allows to discover a copy relation between x_4 and x_1 (via x_2). The condition of the branch labeled 6 cannot be analyzed and thus causes its outgoing flow edges $e5$ and $e6$ to be added to the work list.
4. Now, flow edge $e5$ is processed and marked executable. Since the target operations are already known to be executable, only the ϕ-operation is revisited. However, variables x_1 and x_4 have the same copy-of value x_1, which is identical to the previous result computed in Step 2. Thus, neither of the two work lists is modified.
5. Assuming that the flow edge $e6$ leads to the exit node of the flow graph the algorithm stops after processing the edge without modifications to the data flow information computed so far.

The straightforward implementation of copy propagation, would have needed multiple passes to discover that x_4 is a copy of x_1. But the iterative nature of the propagation along with the ability to discover non-executable code allows to han-

dle even obfuscated copy relations. Moreover, this kind of propagation will only reevaluate the subset of operations affected by newly computed data flow information instead of the complete flow graph once the set of executable operations has been discovered.

16.4.2 Value Range Propagation

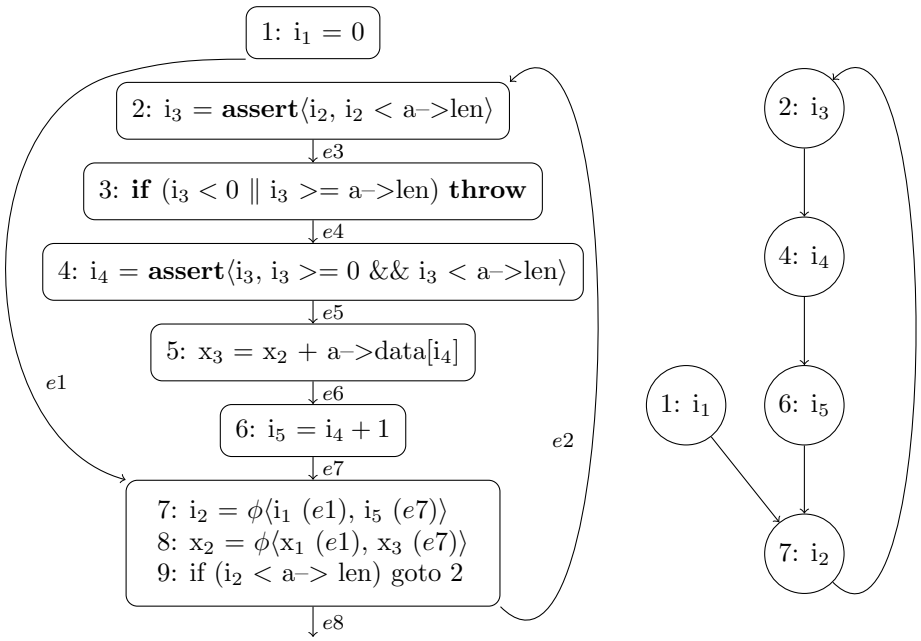
Value range propagation is similar to constant propagation, which served as a running example throughout this chapter. But instead of propagating single constant values, contiguous value ranges are propagated for all variables. For instance, the code in Figure 16.7 is extracted from a typical expansion of bound checking code in languages like Java. Notice how the bounds check at line 9 is redundant as variable *i* is guaranteed to take values in the range [0, a->len - 1]. An important source for range information are the conditions of branch operations, e.g., the condition associated with the **for** loop in the example.

Regular SSA graphs do not allow to take advantage of these conditions, because the range information only applies to the code region that is guarded by the respective branch. Furthermore, certain operations allow to derive range information as a side effect. Typical examples are arithmetic operations with implicit conditions such as the *minimum*, *maximum*, or the *absolute* operations. Trapping operations also provide information in case the execution succeeds, e.g., a pointer is known not to be `NULL` when a memory access succeeds. However, not all program points where this additional range information is available are captured by SSA graphs, which capture variable definitions, their uses, and ϕ-operations only.

The graph is thus preprocessed in order to split the live ranges of variables at appropriate program points whenever range information can be derived. The splitting can be accomplished by inserting artificial *assert* operations into the SSA graph that take an input variable and a condition as operands and return a copy of the input

```
1  struct array {
2     const int len;
3     int *data;
4  };
5
6  void
7  doit(array *a) {
8     for (int i = 0; i < a->len; ++i) {
9        if (i < 0 || i >= a->len)
10          throw;
11       x += a->data[i];
12    }
13 }
```

Fig. 16.7 Useless array bound checking code.



(a) Control flow with assertions. (b) Subset of the SSA graph.

Fig. 16.8 Splitting live ranges using additional *assert* expressions allows to derive more precise value ranges.

variable's value. Note that this *ad hoc* form of live range splitting could be generalized, e.g., by program representations such as *static single information* (SSI) [**?**, **?**] or Extended SSA (*e*-SSA) [**?**] form – also see Chapter **??** of this book.

Reference: Missing reference to live range splitting chapter.

Following this live range splitting, a data flow problem is solved in order to propagate the range information available through the conditions of the assert expressions to the respective uses. The lattice representing the range information is more complex in structure than the previous examples. The lattice elements are divided into two classes, *regular* ranges and *anti* ranges, and consist of a lower and upper bound, where the bounds can be arbitrary symbolic expressions. Regular ranges specify that the actual values of a variable lie within the lower and upper bound, while for anti ranges the opposite is true. The meet operator is defined by combining the bounds such that the resulting range covers both of its arguments, i.e., in the case of regular ranges combining the information of $r_1 = [l_1, u_1]$ and $r_2 = [l_2, u_2]$ yields the range $[min(l_1, l_2), max(u_1, u_2)]$.

However, a naive formulation of the lattice quickly leads to long or even infinite chains. The lattice thus needs to be restricted in order to guarantee termination of the analysis. As an example, consider a loop with an unconditional increment of a counter variable. This could lead to an infinite feedback look where the value range

associated with the counter grows infinitely on every iteration of the work list algorithm. In practice, infinite chains are prevented by choosing the transfer functions such that growth is restricted. For example, the popular open source compiler GCC, restricts the transfer functions of many operations, such as addition or subtraction, to constant singleton ranges only, i.e., ranges that consist of a single element that is known to be constant. Other compiler allow the range to grow within certain limits.

The augmented CFG of the original program from Figure 16.7 is shown in Figure 16.9a. Two additional *assert* expressions were inserted. The first, immediately following the conditional branch of the **for** loop, while the other is placed after the bounds check. For this example we are only interested in the value range of the counter variable i – and the corresponding variables in SSA form. Figure 16.9b thus only shows a subset of the complete SSA graph that is relevant for the counter variable. In the following the major steps of the data flow analysis are discussed:

1. The algorithm starts with the data flow information $\top$ assigned to all variables and by processing the operation labeled 1. The analysis derives the range $[0, 0]$ for variable i_1, which causes the ϕ-operations and the branch labeled 7-9 to be evaluated.
2. Since the flow edge $e7$ is not yet known to be executable the analysis derives the same range, $[0, 0]$, for variable i_2 of ϕ-operation 7.
3. Analyzing the condition of the branch operation 9 yields that both outgoing edges are executable, which triggers the analysis of the operations within the loop.
4. At first, however, examining the conditions of the *assert* expressions does not provide interesting ranges. The analysis merely retains the range $[0, 0]$ for both variables i_3 and i_4.
5. The analysis eventually reaches the increment of the counter variable labeled 6. As mentioned before, the transfer function for such increments has a huge impact on the quality of the analysis results and worst case complexity. For the sake of simplicity, assume that the increment is handled conservatively. The transfer function yields the range $[0, \infty]$ for variable i_5.[1]
6. Due to the updated range information of i_5 and the fact that the flow edge $e7$ is now known to be executable, the ϕ-operation of i_2 has to be reevaluated. Combining the range information computed by step 1 and 5 yields a new range $[0, \infty]$.
7. Note that at this point all flow edges have been marked executable, the following processing steps operate on the sparse SSA graph representation only and thus bypass unrelated computations.
8. First, the *assert* expression for variable i_3 is reevaluated. This time the expression's condition allows to clip the range of the input variable i_2 since the value of a->len is known to be smaller than ∞. The new range for i_3 is thus [0, a->len−1], which triggers the reevaluation of all its uses, in particular the array bounds check labeled with the number 3.

[1] Note that actual compilers have to assume a potential overflow of the counter variable, which further complicates the task of finding a proper transfer function.

9. The second *assert* expression is handled similarly and yields the same range, [0, a->len−1], for variable i_4. Again all of its uses are reevaluated.
10. Finally, the counter increment is reevaluated, however, the range for its variable i_5 does not change and the analysis terminates.

The final result of the analysis is used in order to eliminate array bounds checks, conditional branches, or unreachable code [**?**]. For example, the analysis result computed at Step 8 for the previous example, allows to safely eliminate the array bounds check. Furthermore, range information is valuable during the optimization of data representations, e.g., to reduce the number of bits for certain computations without loss of precision [**?**].

16.5 Beyond Sparse Data Propagation

We have seen that the generic propagation engine is capable to elegantly solve forward problems based on the sparse SSA graph. However, due to limitations of the graph, not all problems can be modeled. This section will highlight some problems where SSA form has proven helpful using specialized algorithms.

Dead Code Elimination

Dead code elimination (DCE) is a very important compiler optimization that eliminates computations that do not have any effect on the program output. This situation may arise from different sources. Code is called *unreachable* when it can never, under any circumstances, be executed at run-time. A common case of unreachable code is often related to conditional branches where the condition always evaluates to either true or false – the respective other case is unreachable, provided no other control path leads to the code. Another form of dead code arises from reachable computations, i.e., it is possible that the computations are carried out at run-time, that are never used by any subsequent computations. Dead or unreachable code can be removed in either case without impairing the program's correctness.

A popular algorithm for DCE, a typical backward data flow analysis, is due to Cytron et al. [**?**]. This algorithm exploits SSA form in order to eliminate both forms of dead code. It proceeds by initially marking all computations of a procedure to be dead. Computations are iteratively marked to be live using a work list algorithm according to the following criteria: (1) computations with observable side-effects such as I/O operations, function calls, and assignments to global or shared memory locations, (2) assignments to variables that are used by other live computations, (3) conditional branches where at least on control-dependent computation has been marked live. When the work list drains, no additional live computations can be discovered. The remaining code marked as dead can safely be eliminate. It is easy to

see that SSA form simplifies the processing of criteria (2) of this algorithm, since the relation between definitions and uses is captured naturally.

Live Variables

Another important problem is the computation of live variables at all program points of a procedure or a subset thereof, which is usually solved using a backward data flow analysis. The approach is applicable to programs in SSA form, however, due to the increased number of variable names due to renaming during SSA construction, a considerable overhead is induced.

Reference: Missing reference to liveness chapter.

Fortunately, the live ranges, i.e., the set of program points where variables are live, offer properties under SSA form that facilitate the computation of liveness information. In fact, several options exist that allow to derive traditional liveness sets, where every program point is annotated with a set of live variables, or to perform on-demand liveness checks for individual variables [**?**]. In either case, the fact that uses of a variables are in any case dominated by its definition is exploited, which allows to prune the set of program point where variables are possibly live. We do not present details of these approaches and the involved algorithms, since Chapter **??** covers this topic in great detail.

16.6 Further Reading

Traditional data flow analysis is well established and well described in numerous papers. The book by Nielsen, Nielsen, and Hankin [**?**] gives an excellent introduction to the theoretical foundations and practical applications.

The sparse propagation engine, as presented in the chapter, is based on the underlying properties of SSA form. Other intermediate representations offer similar properties. *Static Single Information* (SSI) form [**?**] allows both backward and forward problems to be modeled by introducing σ operations, which are placed at program points where data flow information for backward problems needs to be merged [**?**]. Bodík uses an extended SSA form, *e*-SSA, in order to eliminate array bounds checks [**?**]. Ruf [**?**] introduces the *value dependence graph*, which captures both control and data dependencies. Using a set of transformations and simplifications a sparse representation of the input program can be derived which is suited for data flow analysis.

The *sparse evaluation graph* by Choi et.al [**?**] is based on the same basic idea as the approach presented in this chapter, intermediate steps are eliminating by bypassing irrelevant CFG nodes and merging the data flow information only when necessary. Their approach is closely related to the placement of ϕ-operations and similarly relies on the dominance frontier during construction. A similar approach, presented by Johnson and Pingali [**?**], is based on single-entry/single-exit regions.

The resulting graph is usually less sparse, but is also less complex to compute. Ramalingam [**?**] further extends these ideas and introduces the *compact evaluation graph*, which is constructed from the initial CFG using two basic transformations. The approach is superior to the sparse representations by Choi et.al as well as the approach presented by Johnson and Pingali.

The previous approaches derive a sparse graph suited for data flow analysis using graph transformations applied to the CFG. Duesterwald et.al [**?**] instead examine the data flow equations, eliminate redundancies, and apply simplifications to them.

CHAPTER 17

Liveness

B. Boissinot
F. Rastello

Progress: 30%

Material gathering in progress

17.1 TODO

17.2 SSA tells nothing about liveness

Do we really need this? I am not convinced.

Starts with a counter example of the intuition that live-range is "between" the definition and the uses.

Discussion about ϕ-function semantic (this is transverse to the technique).

17.3 Computing liveness sets

Provides Appel's (In the modern compiler book) technique to compute liveness sets under SSA.

17.4 Liveness check

CHAPTER 18

Alias analysis *???*

Progress: 0%

Material gathering in progress

18.1 TODO

CHAPTER 19

Loop tree

S. Pop

Progress: 60%

Structural reordering in progress

This chapter presents an extension of the SSA under which the extraction of the reducible loop tree can be done only on the SSA graph itself. This extension of the SSA representation captures more than the scalar computations: the phi nodes and the scalar assignments encode the structure of the CFG and the strongly connected components of the CFG that are the reducible natural loops. This chapter will present two analysis algorithms: the extraction of the reducible loop tree and the analysis of induction variables based on the SSA representation.

19.1 CFG and Loop Tree can be discovered from the SSA

During the construction of the SSA representation based on a CFG representation, a large part of the CFG information is translated into the SSA representation. As the construction of the SSA has precise rules to place the phi nodes in special points of the CFG (i.e., at the merge of control-flow branches), by identifying patterns of uses and definitions, it is possible to expose the CFG structure from the SSA representation.

Furthermore, it is possible to identify, based on the SSA definitions and uses patterns, higher level constructs inherent to the CFG representation, such as strongly connected components of basic blocks (or natural loops). The induction variable analysis, that we will see in this chapter, is based on the detection of self references in the SSA representation, and on the characterization of these cyclic definitions.

This first section shows that the classical SSA representation is not enough to represent the semantics of the original program. We will see the minimal amount of information that has to be added to the classical SSA representation in order to

represent the loop information in an elegant way: the loop closed SSA form adds an extra variable at the end of a loop for each variable defined in a loop and used after the loop. This is similar to the Gated SSA form presented in Chapter 14.

19.1.1 An SSA representation without the CFG

In the classic definition of the SSA, the CFG provides the skeleton of the program: basic blocks contain assignment statements defining SSA variable names, and the basic blocks with multiple predecessors contain phi nodes. Let's look at what happens when, starting from a classic SSA representation, we remove the CFG.

In order to remove the CFG, a pretty printer function writes the content of basic blocks by traversing the CFG structure (the CFG traversal could be performed in any order: random order, depth-first order, dominator order, etc.). Does the representation, obtained from this pretty printer, contain enough information to enable us to compute the same thing as the original program?

Let's see what happens with an example: supposing that the original program looks like this (in its CFG based SSA representation):

```
bb_1 (preds = {bb_0}, succs = {bb_2})
{
  a = #some computation independent of b
}
bb_2 (preds = {bb_1}, succs = {bb_3})
{
  b = #some computation independent of a
}
bb_3 (preds = {bb_2}, succs = {bb_4})
{
  c = a + b;
}
bb_4 (preds = {bb_3}, succs = {bb_5})
{
  return c;
}
```

after removing the CFG structure, using a random order traversal, we could obtain this:

```
  return c;
  b = #some computation independent of a
  c = a + b;
  a = #some computation independent of b
```

and this SSA code is enough, in the absence of side effects, to recover an order of computation that leads to the same result as in the original program. For example, the evaluation of this sequence of statements would produce the same result:

```
  b = #some computation independent of a
  a = #some computation independent of b
  c = a + b;
  return c;
```

19.1.2 Discovering natural loop structures on the SSA

We will now see how to represent the natural loops in the SSA form by systematically adding extra phi nodes at the end of loops, together with extra information about the loop exit predicate.

Supposing that the original program contains a loop:

```
bb_1 (preds = {bb_0}, succs = {bb_2})
{
  x = 3;
}
bb_2 (preds = {bb_1, bb_3}, succs = {bb_3, bb_4})
{
  i = phi (x, j)
  if (i < N) goto bb_3 else goto bb_4;
}
bb_3 (preds = {bb_2}, succs = {bb_3})
{
  j = i + 1;
}
bb_4 (preds = {bb_2}, succs = {bb_5})
{
  k = phi (i)
}
bb_5 (preds = {bb_4}, succs = {bb_6})
{
  return k;
}
```

Pretty printing, with a random order traversal, we could obtain this SSA code:

```
  x = 3;
  return k;
  i = phi (x, j)
  k = phi (i)
  j = i + 1;
```

We can remark that some information is lost in this pretty printing: the exit condition of the loop disappeared: we will have to record this information in the extension of the SSA representation. The information about the natural loop is still available

under the form of a cyclic definition: by simple substitutions, we can rewrite this SSA code to expose the self reference to "i", as:

```
i = phi (3, i + 1)
k = phi (i)
return k;
```

Thus, we have the definition of the SSA name "i" defined in function of itself. This pattern is characteristic of the existence of a natural loop. We can remark that there are two kinds of phi nodes used in this example:

- loop-ϕ nodes "i = phi (x, j)" have an invariant argument "x" (i.e., the definition does not depend on the values that the phi node takes) and an argument that contains a self reference "j" (i.e., the defining expression "j = i + 1" contains a reference to the same loop-ϕ definition "i"). It is possible to define a canonical SSA form by limiting the number of arguments of loop-ϕ nodes to two.
- close-ϕ nodes "k = phi (i)" merge the values in the end of a loop. They are used to capture the last value of a name defined in a loop. Names defined in a loop can only be used within that loop or in the arguments of a close-ϕ node (that is "closing" the set of uses of the names defined in that loop). In a canonical SSA form it is possible to limit the number of arguments of close-ϕ nodes to one.

19.1.3 Improving the SSA pretty printer for loops

As we have seen in the above example, the exit condition of the loop disappears during the basic pretty printing of the SSA. To capture the semantics of the computation of the loop, we have to specify in the close-ϕ node, which value will be available in the end of the loop. And thus we have to slightly modify the syntax of the close-ϕ nodes to also contain the loop exit condition. This representation is also known under the name of Gated SSA, as presented in Chapter 14. With this extension, the SSA pretty printing of the above example would be:

```
x = 3;
i = loop-phi (x, j)
j = i + 1;
k = close-phi (i >= N, i)
return k;
```

So "k" is defined as the first value of "i" satisfying the loop exit condition, "i >= N". This is a well defined value (in the case of finite loops), as the sequence of values that "i" takes, is defined by the loop-ϕ node and taking the first element of that sequence satisfying the loop exit condition.

In the next section, we will look at an algorithm that translates the SSA representation into a representation of polynomial functions, describing the sequence of values that SSA names take during the execution of a loop.

19.2 Analysis of Induction Variables

The purpose of the induction variables analysis is to provide a characterization of the sequences of values taken by a variable during the execution of a loop. This characterization can be an exact function of the iteration counter of the loop (i.e., the canonical induction variable of a loop starts at zero with a step of one for each iteration of the loop) or an approximation of the values taken during the execution of the loop represented by values in an abstract domain. In this section, we will see a possible characterization of induction variables in terms of scalar sequences. The domain of scalar sequences will be represented by the chains of recurrences: as we will see in more detail in this section, the canonical induction variable will be syntactically represented by the chain of recurrence $\{0, +, 1\}_x$ the initial value of the canonical induction variable is 0, the stride is 1, and the evolution happens in loop number x.

19.2.1 Stride detection

The first step of the induction variables analysis is the detection of the strongly connected components of the SSA. This can be performed by traversing the SSA chains (from a use to its unique definition) and detecting that some definitions are visited twice. From a self referring use-def chain, it is possible to compute the overall effect of one iteration of the loop on the cyclic definition: this is the step of the induction variable. When the step of an induction variable depends on another cyclic definition, one has to further analyze the inner cycle. The analysis of the induction variable ends when all the inner cyclic definitions used for the computation of the step are analyzed. Note that it is possible to construct SSA graphs with strongly connected components that are impossible to characterize with the chains of recurrences. For example, in the following example, "a" does not follow a linear progression:

```
a = loop-phi (0, b)
c = loop-phi (1, d)
b = c + 2
d = a + 3
```

Let's look at an example, presented in Figure 19.1, to see how this algorithm works. The arguments of a phi node are analyzed to determine whether they contain self references or if they are pointing towards the initial value of the induction variable. In this example, (1) represents the edge that points towards the invariant definition. When the argument to be analyzed points towards a longer use-def chain, the full chain is traversed, as shown in (2), until a phi node is reached. In this example, the phi node that is reached in (2) is different to the phi node from which the analysis started, and so in (3) a search starts over the uses that have not yet been analyzed. When the original phi node is found, as in (3), the cyclic def-use

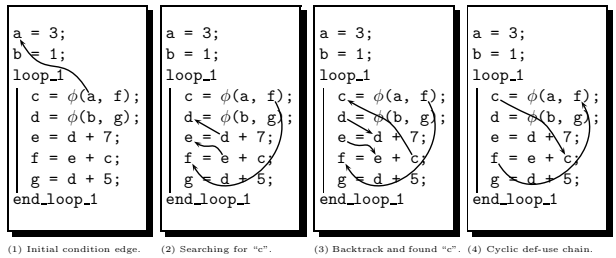


Fig. 19.1 Detection of the cyclic definition.

chain provides the step of the induction variable: in this example, the step is "+ e". Knowing the symbolic expression for the step of the induction variable may not be enough, as we will see next, one has to instantiate all the symbols ("e" in the current example) defined in the varying loop to precisely characterize the induction variable.

19.2.2 Translation to chains of recurrences

Once the cyclic def-use chain has been outlined and the overall loop update expression has been identified, it is possible to translate the sequence of values of the induction variable to a chain of recurrence. The syntax of a polynomial chain of recurrence is: $\{base, +, step\}_x$, where "base" and "step" may be arbitrary expressions or constants, and "x" is the loop in which the sequence is generated. As a chain of recurrence represents the sequence of values taken by a variable during the execution of a loop, the semantics of a chain of recurrence is given by $\{base, +, step\}_x(\ell_x) = base + step * \ell_x$, that is a function of ℓ_x that represents the number of times the body of loop number x has been executed.

When "base" or "step" translates to sequences variating in outer loops, the resulting sequence is represented by a multivariate chain of recurrences. For example $\{\{0, +, 1\}_3, +, 2\}_4$ defines a multivariate chain of recurrence with a step of 1 in loop number 3 and a step of 2 in loop number 4.

When "step" translates into a sequence variating in the same loop, the chain of recurrence represents a polynomial of a higher degree. For example, $\{0, +, \{4, +, 5\}_3\}_3$ represents a polynomial evolution of degree 2 in loop number 3. In this case, the chain of recurrence is also written omitting the extra braces: $\{0, +, 4, +, 5\}_3$. The semantics of the chains of recurrences is defined, using the binomial coefficients $\binom{n}{p}$, by the equation:

$$\{c_0, +, c_1, +, c_2, +, \ldots, +, c_n\}_k(\boldsymbol{\ell}) = \sum_{p=0}^{n} c_p \binom{\ell_k}{p}.$$

with $\boldsymbol{\ell}$ the iteration domain vector (the iteration loop counters for all the loops in which the chain of recurrence variates), and ℓ_k the iteration counter for loop number k. This semantics allows the rewriting rule:

$$\{c_0, +, \{c_1, +, c_2\}_x\}_x = \{c_0, +, c_1, +, c_2\}_x$$

that is very useful in the analysis of induction variables, as it makes it possible to split the analysis into two phases, with a symbolic representation as a partial intermediate result:

- first, the analysis leads to a symbolic form, where the step part "s" is left in a symbolic form, i.e., $\{c_0, +, s\}_x$;
- then, by instantiating the step, i.e., $s = \{c_1, +, c_2\}_x$, the chain of recurrence is that of a higher degree polynomial, i.e., $\{c_0, +, \{c_1, +, c_2\}_x\}_x = \{c_0, +, c_1, +, c_2\}_x$.

19.2.3 Instantiation of symbols and region parameters

The last step of the induction variable analysis consists in the instantiation (or further analysis) of symbolic expressions left from the first step. This includes the analysis of induction variables in outer loops, the analysis of the end value of a loop preceding the analyzed loop, and the replacement of any definitions occurring in sequence before the loop with their defined expression. In some cases, it becomes necessary to leave in a symbolic form every definition outside a given region, and these symbols are then called parameters of the region.

Let's look again at the example of Figure 19.1 to see how the sequence of values of the induction variable "c" is characterized with the chains of recurrences notation. The first step, after the cyclic definition is detected, is the translation of this information into a chain of recurrence: in this example, the initial value (or base of the induction variable) is "a" and the step is "e", and so "c" is represented by a chain of recurrence $\{a, +, e\}_1$ that is variating in loop number 1. The symbols are then instantiated: "a" is trivially replaced by its definition leading to $\{3, +, e\}_1$. The analysis of "e" leads to this chain of recurrence: $\{8, +, 5\}_1$ that is then used in the chain of recurrence of "c", $\{3, +, \{8, +, 5\}_1\}_1$ and that is equivalent to $\{3, +, 8, +, 5\}_1$, a polynomial of degree two:

$$\begin{aligned} F(\ell) &= 3\binom{\ell}{0} + 8\binom{\ell}{1} + 5\binom{\ell}{2} \\ &= \frac{5}{2}\ell^2 + \frac{11}{2}\ell + 3. \end{aligned}$$

19.2.4 Number of iterations and computation of the end of loop value

One of the important properties of loops is their trip count (i.e., the number of times the loop body is executed before the exit condition becomes true). In simple loops, the exit condition of the loop is a comparison of an induction variable against some constant, parameter, or another induction variable. The number of iterations is then computed as the solution of an equation with integer coefficients and integer solutions, also called a Diophantine equation. When one or more coefficients of the Diophantine equation are parameters, the solution is left under a parametric form. The number of iterations can also be an expression variating in an outer loop, in which case, it can be characterized using a chain of recurrence expression.

With an expression representing the number of iterations in a loop, it becomes possible to express the evolution functions of scalar variables variating in outer loops with strides dependent on the value computed in an inner loop: the overall effect of a loop on a scalar variable can be computed as an apply of the number of iterations on the evolution function (of the scalar variable) in the varying loop.

For example, the following code

```
x = 0;
for (i = 0; i < N; i++)
  for (j = 0; j < M; j++)
    x = x + 1;
```

would be written in loop closed SSA form as:

```
  a = 0
  b = loop_1_phi (a, c)
  c = close_2_phi (j < M, d)
  d = loop_2_phi (b, d + 1)
  e = close_1_phi (i < N, b)
  i = loop_1_phi (0, i + 1)
  j = loop_2_phi (0, j + 1)
```

"e" represents the value of variable "x" at the end of the original imperative program. The analysis of scalar evolutions for variable "e" would trigger the analysis of scalar evolutions for all the other variables defined in the loop closed SSA form as follows:

- first, the analysis of variable "e" would trigger the analysis of "i", "N" and "b"
  - the analysis of "i" leads to $i = \{0, +, 1\}_1$
  - "N" is a parameter and is left under its symbolic form
  - the analysis of "b" triggers the analysis of "a" and "c"
    - the analysis of "a" leads to $a = 0$
    - analyzing "c" triggers the analysis of "j", "M", and "d"
      - $j = \{0, +, 1\}_2$
      - "M" is a parameter
      - $d = loop_2_phi(b, d + 1) = \{b, +, 1\}_2$

  - $c = close_2_phi(j < M, d)$ is then computed as the last value of "d" after $loop_2$, i.e., it is the chain of recurrence of "d" applied to the first iteration that does not satisfy $j < M$ and because "j" counts the number of times the body of $loop_2$ has run, that is the number of iterations of $loop_2$, i.e., M. So, to finish the computation of the scalar evolution of "c" we apply "M" to the scalar evolution of "d", leading to $c = \{b, +, 1\}_2(M) = b + M$;
- the scalar evolution analysis of "b" then leads to $b = loop_1_phi(a, c) = loop_1_phi(a, b + M) = \{a, +, M\}_1 = \{0, +, M\}_1$

- and finally the analysis of "e" ends with $e = close_1_phi(i < N, b) = \{0, +, M\}_1(N) = M * N$.

The computation of the end of loop value can also be used for optimization purposes, for example in the case of the constant propagation or the value range propagation after loops. It enables more scalar transformations when the induction variable is used after its defining loop. Another interesting use of the end of loop value is the estimation of the worst case execution time (WCET) where one tries to obtain an upper bound approximation of the time necessary for a program to terminate.

19.3 Conclusion

As we have seen in this chapter, the CFG representation is embedded in the SSA structure, and properties of the CFG like the execution order and the natural loops can be detected by only looking at the SSA definitions and uses. The detection of natural loops is the first step in the analysis of induction variables: after the detection of self referent definitions, it is practical to use the chains of recurrences [**?**, **?**, **?**] to characterize the sequence of values taken by a variable during the execution of a loop. The number of iterations of loops can be computed based on the characterization of induction variables. This paves the way to advanced loop optimizations that need both the number of iterations and induction variable characterizations.

CHAPTER 20

Redundancy Elimination

F. Chow

Progress: 50%

Review in progress

Author: F. Chow

20.1 Introduction

Redundancy elimination is an important category of optimizations performed by modern optimizing compilers. In the course of program execution, certain computations may be repeated multiple times that yield the same results. Such redundant computations can be eliminated by saving the results of the non-redundant computations for reuse at the redundant computations.

There are two types of redundancies: *full* redundancy and *partial* redundancy. A computation is fully redundant if the computation has occurred earlier regardless of the flow of control. The elimination of full redundancy is also called common subexpression elimination, and if applied at the global scope, it is called global common subexpression elimination. A computation is partially redundant if the computation has occurred only along certain paths. Full redundancy can be regarded as a special case of partial redundancy where the computation has occurred regardless of which path is taken.

There are two different methods for deciding whether two computations are the same: the *lexical* method and the *semantic* method. Under the lexical method, two computations are the same if they are written the same way using variables and/or constants before converting to SSA, like $a + 3$. In this case, redundancy can arise only if the variables' values have not changed between the occurrences of the computation. Under the semantic method, two computations are the same if they are the same operation performed on operands that are not necessarily identical by name,

but known to have the same values. For example, $a + b$ and $a + c$ will compute the same result if b and c are known to hold the same value. In this chapter, we are mostly dealing with lexically identified expressions. Our algorithm discussion will focus on the optimization of an expression, like $a + b$, that appears in the program. The compiler will repeat the redundancy elimination algorithm on all the other lexically identified expressions in the program. Section 20.6 will discuss redundancy elimination among expressions that have been semantically proven to yield the same value.

The concept of partial redundancy was first introduced by Morel and Renvoise. Before the technique of partial redundancy elimination was developed, optimizing compilers have been performing global common subexpression elimination and loop invariant code motion in separate global optimization phases. In their seminal work [**?**], Morel and Renvoise showed that global common subexpressions and loop-invariant computations are special cases of partial redundancy that can be subsumed by the single optimization of partial redundancy elimination (PRE). Morel and Renvoise formulated PRE as a code placement problem, in which the best set of insertion points for the expression being optimized is to be determined. Such insertions render some original computations to be fully redundant, so they can be trivially deleted. The PRE algorithm developed by Morel and Renvoise involves bi-directional data flow analysis, which incurs more overhead than uni-directional data flow analysis. In addition, their algorithm does not yield optimal results in certain situations.

An better placement strategy, called lazy code motion (LCM), was later developed by Knoop *et al* [**?**][**?**]. It improved on Morel and Renvoise's results by avoiding unnecessary code movements and by removing the bi-directional nature of the original PRE data flow analysis. The code placement produced by lazy code motion is optimal: the number of computations during execution time cannot be further reduced by *safe* code motion [**?**], and the lifetimes of the temporaries introduced for storing the computed values are minimized. After lazy code motion was introduced, there have been alternative formulations of PRE algorithms that achieve the same optimal results, but differ in the formulation approach and implementation details[**?**][**?**][**?**][**?**].

The above approaches to PRE are all based on encoding program properties in bit vector forms and the iterative solution of data flow equations. Since the bit vector representation uses basic blocks as its granularity, a separate algorithm is needed to detect and suppress local common subexpressions. An SSA-based approach to solve PRE was proposed by Chow *et al* [**?**][**?**]. Their SSAPRE algorithm is an adaptation of LCM to take advantage of the use-def information inherent in SSA. It avoids having to encode data flow information in bit vector form, and eliminates the need for a separate algorithm to suppress local common subexpressionsi. Their algorithm was first to make use of SSA to solve data flow problems for expressions in the program, taking advantage of SSA's sparse representation so that fewer number of steps are needed to propagate data flow information. The SSAPRE algorithm thus brings the many desirable characteristics of SSA-based solution techniques to PRE, and further advance our understanding of this important optimization. Because SS-

APRE operates on a sparse representation, it precludes the need for any alternative formulation to its algorithm.

In this chapter, we only cover the conceptual bases of SSAPRE for the purpose of getting an intuitive understanding of its inner workings. The readers are referred to the original publications for the full algorithm[**?**][**?**].

20.2 Why PRE and SSA are related

Figure 20.1 shows the two most basic forms of partial redundancy. In Figure 20.1(a), $a+b$ is redundant when the right path is taken. In Figure 20.1(b), $a+b$ is redundant whenever the branch-back edge of the loop is taken. Both are examples of *strictly* partial redundancies, in which insertions are required to eliminate the redundancies. In contrast, a full redundancy can be deleted without requiring any insertion.

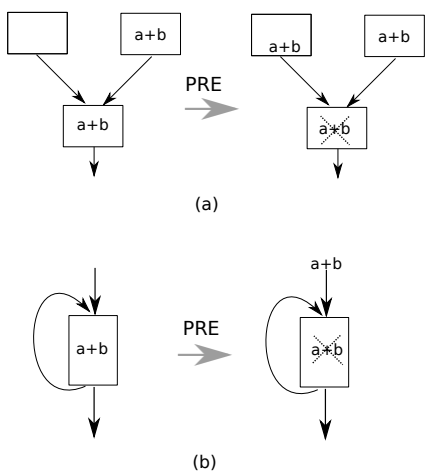


Fig. 20.1 Two basic examples of partial redundancy elimination.

We can visualize the impact on redundancies of a single computation as shown in Figure 20.2. In the region of the control flow graph dominated by the occurrence of $a+b$, any further occurrence of $a+b$ is fully redundant, assuming a and b are not modified. Following the program flow, once we are past the dominance frontiers, any further occurrence of $a+b$ is partially redundant. In constructing SSA form, dominance frontiers are where ϕ's are inserted. Since partial redundancies start at dominance frontiers, it must be related to SSA's ϕ's. In fact, the same sparse approach to modeling the use-def relationships among the occurrences of a program

variable can be used to model the redundancy relationships among the different occurrences of $a + b$.

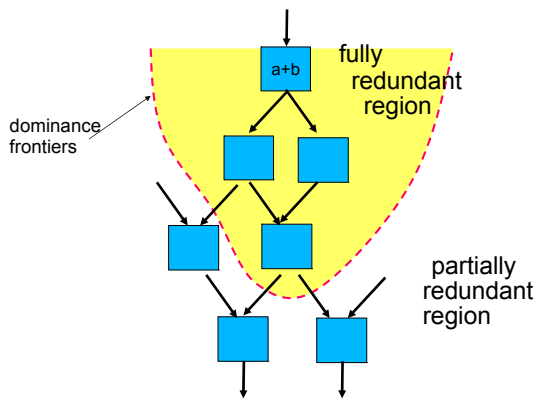


Fig. 20.2 Dominance frontiers are boundaries between fully and partially redundant regions.

We assume the input program is already in SSA form. If an occurrence $a_j + b_j$ is redundant with respect to $a_i + b_i$, our representation has redundancy edges that connect $a_i + b_i$ to $a_j + b_j$. To expose potential partial redundancies, we introduce the operator Φ at the dominance frontiers of the occurrences, which has the effect of factoring the redundancy edges at merge points in the control flow graph.[1] The resulting *factored redundancy graph* (FRG) can be regarded as the SSA form for expressions.

To make the expression SSA form more intuitive, we introduce the hypothetical temporary h, which can be thought of as the temporary that will be used to store the value of the expression for reuse in order to suppress redundant computations. The constructed SSA form for h is not precise, because we have not yet determined where h should be defined or used.

The SSA form for h is constructed in two steps similar to ordinary SSA form: the Φ-Insertion step followed by the Renaming step. In the Φ-Insertion step, we insert Φ's at the dominance frontiers of all the expression occurrences, to ensure that we do not miss any possible placement positions for the purpose of PRE, as in Figure 20.3(a). We also insert Φ's caused by expression alteration. Such Φ's are triggered by the occurrence of ϕ's for any of the operands in the expression. In

[1] Adhering to SSAPRE's convention, we use lower case ϕ's in the SSA form of variables and upper case Φ's in the SSA form for expressions.

Figure 20.3(b), the Φ at block 3 is caused by the ϕ for a in the same block, which in turns reflects the assignment to a in block 2.

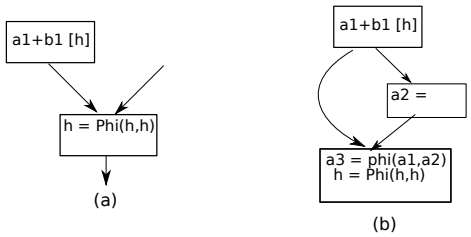


Fig. 20.3 Examples of Φ insertion

The Renaming step assigns SSA versions to h such that occurrences renamed to idential h-versions will compute to the same values. We conduct a pre-order traversal of the dominator tree similar to the renaming step in SSA construction for variables, but with the following modifications. In addition to a renaming stack for each variable, we maintain a renaming stack for the expression. Entries on the expression stack are popped as our dominator tree traversal backtracks past the blocks where the expression originally received the version. Maintaining the variable and expression stacks together allows us to decide efficiently whether two occurrences of an expression should be given the same h-version.

There are three kinds of occurrences of the expression in the program: (a) the occurrences in the original program, which we call *real* occurrences; (b) the Φ's inserted in the Φ-Insertion step; and (c) Φ operands, which are regarded as occurring at the ends of the predecessor blocks of their corresponding edges. During the visitation in Renaming, a Φ's is always given a new version. For a non-Φ, i.e. cases (a) and (c), we check the current version of every variable in the expression (the version on the top of each variable's renaming stack) against the version of the corresponding variable in the occurrence on the top of the expression's renaming stack. If all the variable versions match, we assign it the same version as the top of the expression's renaming stack. If any of the variable versions does not match, for case (a), we assign it a new version, as in the example of Figure 20.4(a); for case (c), we assign the special class $\perp$ to the Φ operand to denote that the value of the expression is unavailable at that point, as in the example of Figure 20.4(b). If a new version is assigned, we push the version on the expression stack.

The FRG captures all the redundancies of $a+b$ in the program. In fact, it contains just the right amount of information for determining the optimal code placement. Because strictly partial redundancies can only occur at the Φ nodes, insertions for PRE only need to be considered at the Φ's.

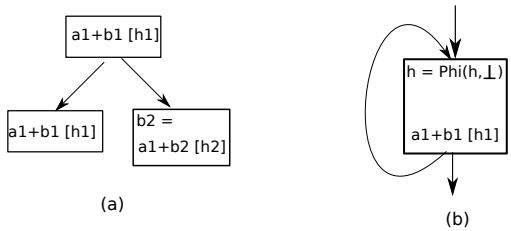


Fig. 20.4 Examples of expression renaming

20.3 How SSAPRE Works

Referring to the expression being optimized as X, we use the term *placement* to denote the set of points in the *optimized* program where X's computation occurs. In contrast, *original computation points* refer to the points in the *original* program where X's computation took place. The objective of SSAPRE is to find a placement that satisfies the following four criteria in this order:

1. Correctness — X is fully available at all the original computation points.
2. Safety — There is no insertion of X on any path that did not originally contain X.
3. Computational optimality —No other safe placement can result in fewer computations of X on any path in the program.
4. Lifetime optimality — Subject to the computational optimality, the life range of the temporary introduced to store X is minimized.

Each occurrence of X at its original computation point can be qualified with exactly one of the following attributes:

- fully redundant
- strictly partially redundant (SPR)
- non-redundant

As a code placement problem, SSAPRE follows the same two-step process used in all PRE algorithsm. The first step determines the best set of insertion points that render as many SPR occurrences fully redundant as possible. The second step deletes fully redundant computations taking into account the effects of the inserted computations. Since the second full redundancy elimination step is trivial and well understood, the challenge lies in the first step for coming up with the best set of insertion points. The first step will tackle the safety, computational optimality and lifetime optimality criteria, while the correctness criterion is delegated to the second step. For the rest of this section, we only focus on the first step for finding the best insertion points, which is driven by the SPR occurrences.

We assume that all *critical edges* in the control flow graph have been removed by inserting empty basic blocks at such edges[2][**?**]. In the SSAPRE approach, insertions are only performed at Φ operands. When we say a Φ is a candidate for insertion, it means we will consider inserting at its operands to render X available at the entry to the basic block containing that Φ. An insertion at a Φ operand means inserting X at the incoming edge corresponding to that Φ operand. In reality, the actual insertion is done at the end of the predecessor block.

20.3.1 The Safety Criterion

As we have pointed out at the end of Section 20.2, insertions only need to be considered at the Φ's. The safety criterion implies that we can only insert at Φ's where X is *downsafe* (fully anticipated). Thus, we perform data flow analysis on the FRG to determine the *downsafe* attribute for Φ's. Data flow analysis can be performed with linear complexity on SSA graphs, which we illustrate with the Downsafety computation.

A Φ is not *downsafe* if there is a control flow path from that Φ along which the expression is not evaluated before program exit or before being altered by redefinition of one of its variables. Except for loops with no exit, this can happen only due to one of the following cases: (a) there is a path to exit or an alteration of the expression along which the Φ result version is not used; or (b) the Φ result version appears as the operand of another Φ that is not *downsafe*. Case (a) represents the initialization for our backward propagation of $\neg downsafe$; all other Φ's are initially marked $downsafe$. The Downsafety propagation is based on case (b). Since a real occurrence of the expression blocks the case (b) propagation, we mark each Φ operand with a flag *has_real_use* when the path to the Φ operand crosses a real occurrence of the same version of X to effect the blocking. Figure 20.5 gives the DownSafety propagation algorithm.

20.3.2 The Computational Optimality Criterion

The best set of Φ's for performing insertion is arrived at via a process of elimination. At this point, we have eliminated the unsafe Φ's based on the safety criterion. We now seek to disqualify more Φ's from insertion consideration by identifying those that we can prove as violating the computational optimality criterion. This is done by performing the CanBeAvail propagation, which is in the forward direction.

The CanBeAvail propagation is derived from the well-understood full availability analysis. We define a Φ *can_be_avail* if and only if inserting there will not violate

[2] A critical edge is one whose tail block has multiple successors and whose head block has multiple predecessors.

```
                    procedure Reset_downsafe(X) {
    if (has_real_use(X) or def(X) is not a Φ)
        return
    f ← def(X)
    if (not downsafe(f))
        return
    downsafe(f) ← false
    for each operand ω of f do
        Reset_downsafe(ω)
                    }
                    procedure DownSafety {
    for each f ∈ {Φ's in the program} do
        if (not downsafe(f))
            for each operand ω of f do
                Reset_downsafe(ω)
                    }
```

Fig. 20.5 Algorithm for DownSafety

computational optimality. In other words, a Φ is $\neg$*can_be_avail* if and only if inserting there violates computational optimality. This can happen only due to one of the following cases: (a) the Φ is not *downsafe* and one of its operands is $\perp$; or (b) the Φ is not *downsafe* and it has an operand that is a $\neg$*can_be_avail*Φ and that operand is not *has_real_use*. Case (a) represents the initialization for our forward propagation of $\neg$*can_be_avail*; all other Φ's are initially marked *can_be_avail*. The CanBeAvail propagation is based on case (b).

After *can_be_avail* has been computed, it is possible to perform insertions at all the *can_be_avail* Φ's. There would be full redundancies created among the insertions themselves, but they would not affect computational optimality because the subsequent full redundancy elimination step will remove any fully redundant inserted or non-inserted computation, leaving the earliest computations as the optimal code placement.[3]

20.3.3 The Lifetime Optimality Criterion

To fulfill lifetime optimality, we perform a second forward propagation called Later that is derived from the well-understood partial availability analysis. The purpose is to disqualify *can_be_avail* Φ's that are partially available based on the original occurrences of X. A Φ is marked *later* if it is not necessary to insert there because a later insertion is possible. We optimistically regard all the *can_be_avail* Φ's to be *later*, except the following cases: (a) the Φ has an operand defined by a real computation; or (b) the Φ has an operand that is a Φ marked not *later*. Case (a) represents

[3] This outcome is referred to as *busy code motion* by Knoop *et al.* to contrast with their lazy code motion.

the initialization for our forward propgation of not *later*; all other *can_be_avail* Φ's are marked *later*. The Later propagation is based on case (b).

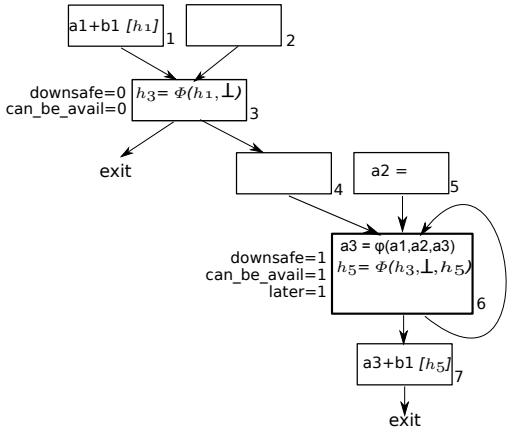


Fig. 20.6 Example to show the need of the *Later* attribute

The final Φ's for performing insertion are the Φ's where *can_be_avail* and $\neg$*later* hold. We call such Φ's *will_be_avail*. At each of these Φ's, insertion is performed at each operand that satisfies either of the following condition:

1. it is $\perp$; or
2. *has_real_use* is false and it is defined by a $\neg$*will_be_avail*Φ.

We illustrate our discussion in this section with the example of Figure 20.6, where the program exhibits partial redundancy that cannot be removed by safe code motion. The two Φ's with their computed data flow attributes are as shown. If insertions were based on *can_be_avail*, $a + b$ would have been inserted at blocks 4 and 5, which would have resulted in unnecessary code motion that increases register pressure. By considering *later*, no insertion is performed, which is optimal under safe PRE for this example.

20.4 Speculative PRE

If we ignore the safety requirement of PRE discussed in Section 20.3, the resulting code motion will involve speculation. In the absence of systems support, speculation should only be applied to expressions that will not cause runtime exceptions or faults, since such exceptions or faults will alter the external behavior of the program. For example, indirect loads from unknown pointer values cannot be speculated unless the systems would mask out the effect of loading from invalid addresses. Speculative code motion suppresses redundancy in some path at the expense of another

path where the computation is added but result is unused. As long as the paths that are burdened with more computations are executed less frequently than the paths where the redundant computations are avoided, a net gain in program performance can be achieved. Thus, speculative code motion should only be performed when there are clues about the relative execution frequencies of the paths involved.

Without profile data, speculative PRE can be conservatively performed by restricting it to loop-invariant computations. Figure 20.7 shows a loop-invariant computation $a+b$ that occurs in a branch inside the loop. This loop-invariant code motion is speculative because, depending on the branch condition inside the loop, it may be executed zero time, while moving it to the loop header causes it to execute one time. This speculative loop-invariant code motion is profitable unless the path inside the loop containing the expression is never taken, which is usually not the case. When performing SSAPRE, marking Φ's located at the start of loop bodies downsafe will effect speculative loop invariant code motion[**?**].

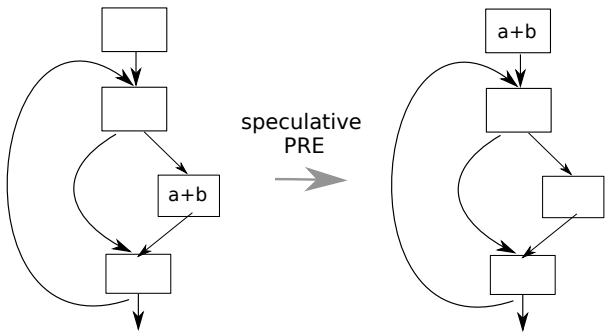


Fig. 20.7 Speculative loop-invariant code motion

To enable more code motion of unsafe operations, Murphy *et al.* modify SSAPRE to use the *fault-safety* property[**?**]. They define *dangerous* computations as operations that may fault but will otherwise have no observable side effects. Such computations include indirect loads and divides. Such dangerous computations are sometimes protected by tests (or guards) placed in the code by the programmers. Compilers for languages like Java also insert runtime checks to ensure faults never occur. When such a test occurs in the program, the dangerous computation is said to be *safety-dependent* on the control flow point that establishes its safety. A dangerous instruction is *fault-safe* at any point in the program where its safety dependence is satisfied.

Murphy *et al.* represent safety dependences as value dependences in the form of the abstract *tau values* described in [**?**]. Each check that succeeds will define a tau value on its fall-through path. During SSAPRE, dangerous computations will have additional *tau* operands attached to them. The tau operands are also variables

in SSA form, so their definitions can be found by following the use-def edges. The compiler inserts the definitions of the taus also with abstract right-hand-side values, like **tauedge**. Because they are abstract, they are omitted in the generated code after the SSAPRE phase. A dangerous computation can be defined to have more than one tau operands, depending on its semantics. When all its tau operands have definitions, it means the computation is fault-safe; otherwise, it is unsafe. By including the tau operands into consideration, speculative PRE automatically honors the fault-safety of dangerous computations when it performs speculative code motion.

As an example, if we replace the expression $a+b$ in Figure 20.7 by a/b, the speculative code motion cannot be performed because if b is 0, the speculative insertion of a/b at the loop header will cause a run-time divide-by-zero fault. In Figure 20.8, the program contains a non-zero test for b. We define a tau operand for a/b in SSAPRE to provide the information whether a non-zero test for b is available. The presense of the non-zero test for b causes the compiler to insert the definition of $tau1$ with the abstract right-hand-side value **tauedge**. Since the divide inside the loop $a_1/b_1, tau_1$ has a tau operand whose definition is visible, speculative SSAPRE will force the Φ at the head of the loop body to be downsafe. When $a_1/b_1, tau_1$ is hoisted out of the loop, it automatically stops at the definition of tau_1 because PRE obeys value dependence.

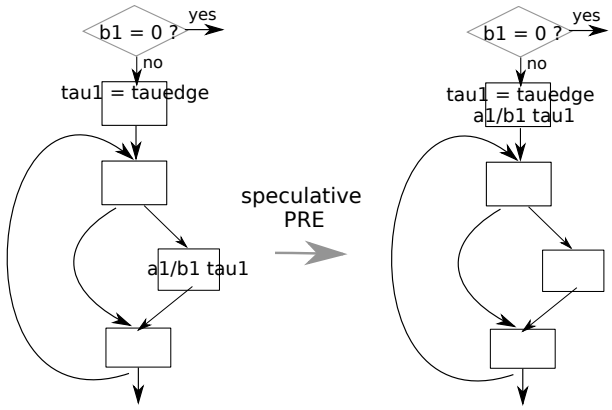


Fig. 20.8 Speculative and fault-safe loop-invariant code motion

When execution profile data are available, it is possible to tailor the use of speculation to maximize run-time performance for the execution that matches the profile. Xue and Cai presented a computationally and lifetime optimal algorithm for speculative PRE based on profile data[**?**]. Their algorithm uses bit-vector-based data flow analysis and applies minimum cut to flow networks formed out of the control flow graph to find the optimal code placement. Zhou et al. applies the minimum cut approach to flow networks formed out of the FRG in the SSAPRE framework to

achieve the same computational and lifetime optimal code motion[**?**]. They showed their sparse approach based on SSA results in smaller flow networks, enabling the optimal code placements to be computed more efficiently.

20.5 Register Promotion via PRE

Register promotion refers to the important task in an optimizing compiler of identifying the data items that are candidates for register allocation in the program. To represent register allocation candidates, compilers commonly use an unlimited number of *pseudo-registers*. Pseudo-registers are also called symbolic registers or virtual registers, to distinguish them from real or physical registers. Pseudo-registers have no alias, and the process of assigning them to real registers involves only renaming them.

20.5.1 Register Promotion as Placement Optimization

Under PRE, the temporaries generated to hold the values of redundant computations are pseudo-registers. Local variables determined by the compiler to have no alias can be trivially renamed to pseudo-registers. All remaining register allocation candidates have to be assigned pseudo-registers through the process of register promotion. Register promotion is also responsible for generating the most efficient code to set up the data objects in pseudo-registers. Targets for register promotion include scalar variables, indirectly accessd memory locations and program constants. Load operations are needed to put them into pseudo-registers before they are used[4], and when there is an assignment, a store operation is needed. Since the goal of register promotion is to obtain the most efficient placements for the loads and stores, register promotion can be modeled as two separate problems: PRE of loads, followed by PRE of stores.

From the point of view of redundancy, loads are like expressions because the later occurrences are the ones to be deleted. For stores, the reverse is true: the earlier stores are the ones to be deleted, as is evident in the examples of Figure 20.9(a) and (b). The PRE of stores, also called *partial dead code elimination*, can thus be treated as the dual of the PRE of loads. Performing PRE of stores thus has the effects of moving stores forward while inserting them as early as possible. Combining the effects of the PRE of loads and stores results in optimal placements of loads and stores while minimizing the live ranges of the pseudo-registers, by virtue of the computational and lifetime optimalities of our PRE algorithm.

[4] Depending on the ISA, some constants may not need to be put in registers, and they should be excluded from register promotion.

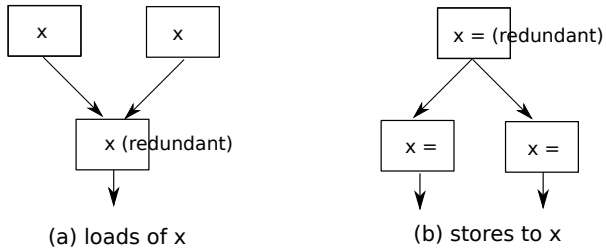


Fig. 20.9 Duality between load and store redundancies

20.5.2 Load Placement Optimization

PRE applies to any computation including loads from memory locations and loads of constants. In program representations, loads can either be indirect through a pointer or direct. Direct loads and constants are leaves in expression trees. When we apply SSAPRE to direct loads, since the hypothetical temporary h can be regarded as the candidate variable itself, the Φ-insertion step and Rename step can be streamlined. In other words, the Φ's are the variable's ϕ's, and the h-versions are the variable's SSA version.

When working on the PRE of memory loads, it is important to also take into account the stores, which we call *l-value* occurrences. A store of the form $x \leftarrow$ `<expr>` can be regarded as being made up of the sequence:

$r \leftarrow$ `<expr>`
$x \leftarrow r$

Because the pseudo-register r contains the current value of x, any subsequent occurrences of the load of x can reuse the value from r, and thus can be regarded as redundant. Figure 20.10 gives examples of loads made redundant by stores.

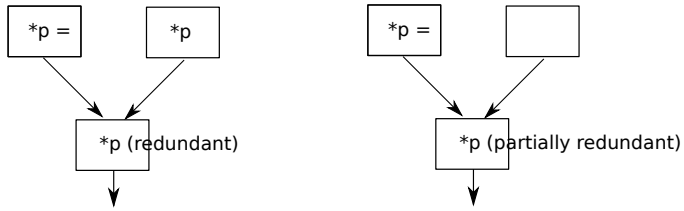


Fig. 20.10 Redundant loads after stores

When we perform the PRE of loads, we thus include the l-value occurrences into consideration. The Φ-insertion step will insert Φ's at the iterated dominance

frontiers of l-value occurrences. In the Rename step, an l-value occurrence is always given a new h-version, because a store is a definition. Any subsequent load renamed to the same h-version is redundant with respect to the store.

We apply the PRE of loads first, followed by the PRE of stores. This ordering is based on the fact that the PRE of loads is not affected by the results of the PRE of stores, but the PRE of loads creates more opportunities for the PRE of stores by deleting loads that would otherwise have blocked the movement of stores. In addition, speculation is required for the PRE of loads and stores in order for register promotion to do a decent job in loops.

The example in Figure 20.11 illustrates what we discuss in this section. During the PRE of loads (LPRE), a = is regarded as an l-value occurrence. The hoisting of the load of a to the loop header does not involve speculation. The occurrence of a = causes r to be updated by splitting the store into the two statements r = followed by $a = r$. In the PRE of stores (SPRE), speculation is needed to sink a = to outside the loop because the store occurs in a branch inside the loop. Without performing LPRE first, the load of a inside the loop would have blocked the sinking of a =.

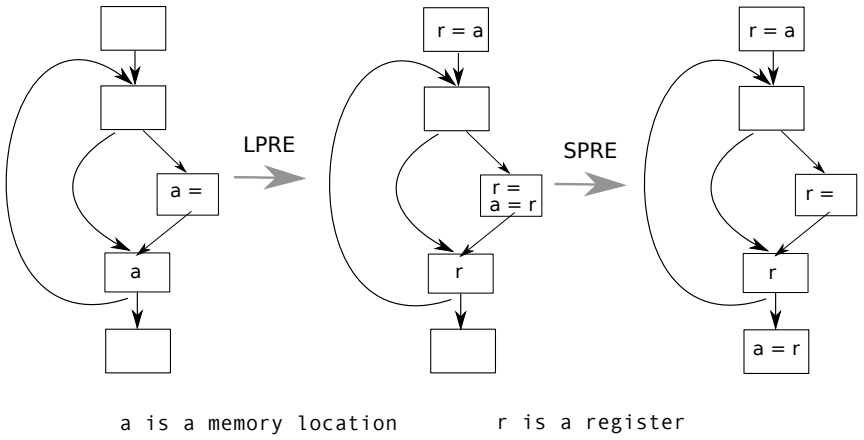


Fig. 20.11 Register promotion via load PRE followed by store PRE

20.5.3 Store Placement Optimization

As mentioned earlier, SPRE is the dual of LPRE. In the presence of store redundancies, the earlier occurrences are redundant. Code motion in SPRE will have the effect of moving stores forward with respect to the control flow graph. Any presence of (aliased) loads have the effect of blocking the movement of stores or rendering the earlier stores non-redundant.

To apply the dual of the SSAPRE algorithm, it is necessary to compute a program representation that is the dual of the SSA form, and we call this the *static single use* form. In SSU, use-def edges are factored at divergence points in the control flow graph. We call this factoring operator Λ. Each use of a variable establishes a new version (we say the load *uses* the version), and every store reaches exactly one load. The Λ is regarded as a use of a new version of the variable. The use post-dominates all the stores of its version. The left-hand-side of the Λ is the multiple definitions, each of which is post-dominated by their single uses.

We call our store PRE algorithm SSUPRE, which is made up of the corresponding steps in SSAPRE. Λ-Insertion and SSU-Rename, construct the SSU form for the variable whose store is being optimized. The data flow analyses consist of Up-Safety to compute the *upsafe* (fully available) attribute, CanBeAnt to compute the *can_be_ant* attribute and Earlier to compute the *earlier* attribute. Though store elimination itself does not require the introduction of temporaries, lifetime optimality still needs to be considered for the temporaries introduced in the LPRE phase which hold the values to the point where the stores are placed. It is desirable not to sink the stores too far down.

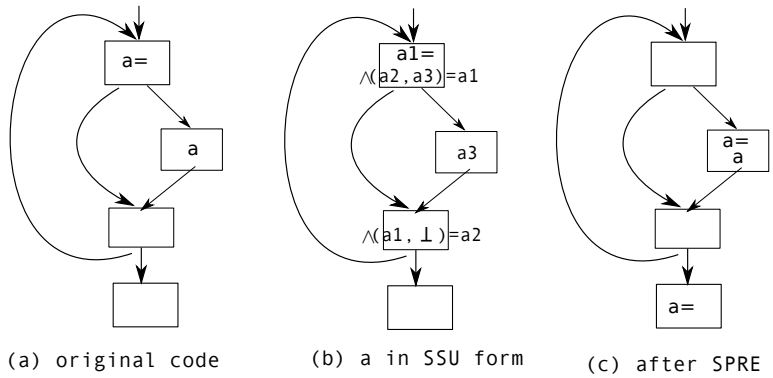


Fig. 20.12 Example of program in SSU form and the result of applying SSUPRE

Figure 20.12 gives an example program with (b) being the SSU representation for the program in (a). (c) shows the result of applying SSUPRE to the code. The store can be sunk to outside the loop only if it is also inserted in the branch inside the loop that contains the load. The optimized code no longer exhibits any store redundancy.

A limitation of the above SSUPRE algorithm is that it does not apply to indirect stores. It may be possible to extend the algorithm to make it work on indirect stores as well.

20.6 Redundancy via the Semantic Approach

The redundant computations stored into a temporary introduced by PRE may be of different values because the same lexically identified expression may yield different results at different points in the program. Since PRE applies only to lexically identified expressions, it is not capable of recognizing redundant computations among lexically different expressions based on their semantics. On the other hand, under the semantic approach, redundant computations can be recognized among computations that are semantically determined to yield the same values.

20.6.1 Value Numbering

Computations are determined to yield the same value using *value numbering* analysis techniques. The term *value number* originated from the hash-based method developed by Cocke and Schwartz for recognizing when two expressions evaluate to the same value within a basic block [**?**]. The value number of an expression tree can be regarded as the index of its hashed entry in the hash table. An expression tree is hashed bottom up starting with the leaf nodes. Each internal node is hashed based on its operator and the value numbers of its operands. Two expressions with the same value number *must* evaluate to the same value at run-time.

When the program has been put into SSA form, value number can be extended to the global scope by assigning a unique value number to each variable version by virtue of the single definition property [**?**]. Additional refinements to hash-based global value numbering algorithms have been proposed by Briggs *et al* [**?**].

A second approach for determining whether two expressions compute the same value that uses the partitioning method instead of hashing. First developed by Alpern *et al.* [**?**], the algorithe partitions all the expressions in the program into congruence classes. Values in the same congruence class are considered as evaluating to identical values. The algorithm is optimistic because when it starts, it puts all expressions based on the same operator into the same congruence class. Given two expressions within the same congruence class, if their operands at the same operand position belong to different congruence classes, the two expressions may compute to different values, and thus should not be in the same congruence class. This is used as the subdivision criterion. As the algorithm iterates, the congruence classes are subdivided into smaller ones while the total number of congruence classes increases. The algorithm terminates when no more subdivision can occur. At this point, an enumeration of the resulting congruence classes can be used as value numbers. The detailed algorithm is shown in Figure 20.13. In the last **for** loop, $\phi \subset (s \cap touched) \subset s$ is the criterion for subdividing class s because the other members of s that do not have x at operand position p potentially compute to a different value. After the partition into n and the new s, if s is not in the worklist (i.e. processed already), the partitioning was already stable with respect to the old s, and we can add either n or the new s to

```
Place all values computed by the same opcode in the same congruence class
worklist ← set of all congruence classes
while worklist ≠ φ
    Select and delete an arbitrary congruence class c from worklist
    for each operand position p of a use of x ∈ c
        touched ← φ
        for each x ∈ c
            Add any expression in the program that uses x in position p to touched
        for each class s such that φ ⊂ (s ∩ touched) ⊂ s
            Create a new class n ← s ∩ tourhed
            s ← s − n
            if s ∈ worklist
                Add n to worklist
            else
                Add smaller of n and s to worklist
```

Fig. 20.13 The partitioning algorithm

the worklist to re-stabilize with respect to that split of s. Choosing the smaller one results in less overhead.

Briggs *et al.* have also proposed additional refinements to the partitioning technique [**?**]. Partition-based algorithms do not supercede, but instead complement the hash-based algorithms. The most powerful value numbering algorithms are in fact based on a combination of the two approaches.

20.6.2 Redundancy Elimination via Value Numbering

So far, we have only talked about finding computations that compute the same value, but have not addressed how to use the results of such analysis to optimize the program. Two computations that compute the same value do not exhibit redundancy if they are not situated on the same execution path. Thus, it is logical to consider performing PRE separately for each value number.

If we consider the temporary t that will be introduced to store the redundant computations under value-number-based PRE, we can see that its value will stay the same throughout its lifetime. If there are ϕ's introduced for the temporary, they will be merging identical values, and we know from experience that such ϕ's are rare. A subset of such ϕ's is expected to come from PRE's insertions, and that implies that insertions introduced by value-number-based PRE are also rare.

Value-number-based PRE also has to deal with the additional issue of *how* to generate an insertion. Because the same value can come from different forms of expressions at different points in the program, it is necessary to determine which form to use at an insertion point. If the insertion point is outside the live range of any

variable version that can compute that value, then the insertion point has to be disqualified. Due to this complexity, and the expectation that strictly partial redundancy is rare among computations that yield the same value, we focus only on eliminating full redundancies among computations that have the same value number[5].

Since full redundancy elimination (FRE) is a subset of PRE, we can adapt the SSAPRE algorithm to work on value-number-identified expressions and remove full redundancies among them. We call this adapted algorithm VNFRE.

In VNFRE, the Φ-Insertion step only needs to consider the iterated dominance frontiers. It is not necessary to consider the variable operands, because as long as the value number is the same, we do not have to consider when any variable operand's value is changed. The Renaming step is also simpler because we also ignore the variable operands. In fact, the only situations where new h-versions are introduced are at program entry (when the renaming stack is empty) and when encountering Φ's. With the FRG constructed, we perform the full availability analysis on the Φ's to identify the expressions that can be deleted.

In practice, with copy propagation, constant folding and SSAPRE having been performed earlier, there are not much optimization opportunities left for VNFRE to cater to. One useful application of VNFRE is in induction variable collescing. Code like the following, with multiple induction variables, could be the result of earlier optimizations like strength reduction:

```
i = 0;
j = 0;
while <cond> {
  i = i + 4;
  j = j + 4;
}
```

Value numbering will determine that at different points in the code, `i` and `j` always have the same value number. Performing VNFRE will yield the net effect of getting rid of the extra induction variables.

20.7 Conclusion

The close relationsihp between PRE and SSA arises because partial redundancies can be exposed by factoring at control flow merge points. The SSAPRE algorithm capitalizes on prior techniques developed for computing and manipulating SSA form. The SSAPRE framework also shows that the concept and techniques of SSA can be made to apply to any program constructs, not just variables. The constructed SSA graphs can then be used to efficiently perform sparse data flow propagation.

[5] This assumes PRE for lexically identified expressions have been applied, which would have removed the redundancies among lexically identified expressions that also have the same value number.

There are additional optimizations that can be implemented using the SSAPRE framework that we have not covered. They include code hoisting, register shrink-wrapping [**?**] and live range shrinking. Moreover, PRE has traditionally provided the context for integrating additional optimizations into its framework. They include operator strength reduction [**?**] and linear function test replacement [**?**]. As a result, PRE has become the most powerful and encompassing optimization framework in modern optimizing compilers.

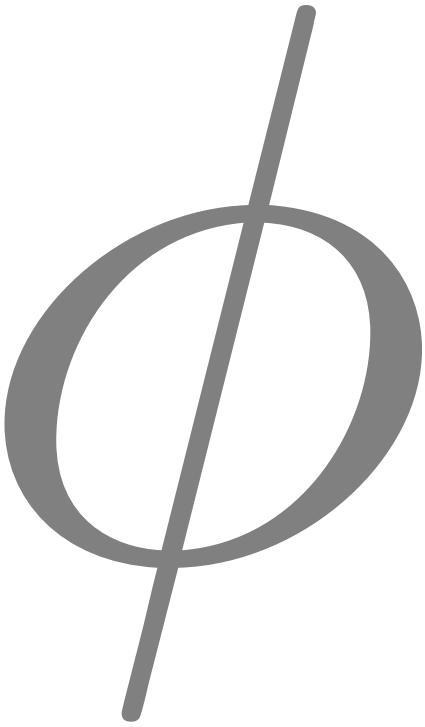
CHAPTER 21

Typestate analysis *???*

Progress: 0%

Material gathering in progress

21.1 TODO



Part IV
Machine dependent optimizations and codegen

Progress: 38%

Progress: 61%

TODO: Style: instructive, no performance results, try for max pages 10 per ssainput

CHAPTER 22

Introduction

M. A. Ertl

Progress: 5%

Material gathering in progress

Code generation transforms the program from a machine-independent intermediate representation (IR) into some representation of the code for the target machine. Traditionally code generation is divided into the following subtasks:

Instruction selection combines the operations of the IR into target machine instructions. Instruction selection aims for selecting a set of instructions with the lowest combined cost; the cost of an instruction is its code size and/or its execution time.

Instruction Scheduling orders the instructions, resulting in a total order; among the many possible orders, instruction scheduling usually aims for one that exploits instruction-level parallelism, i.e., it tries to avoid dependences between instructions that are close together. Another (conflicting) goal is to reduce register pressure.

Register allocation assigns target-machine registers to IR values (aka live ranges or pseudo-registers) and may insert code to spill registers to and refill registers from memory if there are too many values competing for the limited number of registers. The goals in register allocation are to avoid spill code and moves between registers.

The ordering of these phases determines the quality of the resulting code. Usually the phases are performed in the order given above, with another scheduling pass afterwards to schedule spill code; however, there are phase ordering conflicts between these phases as long as the phases are treated separately: e.g., instruction scheduling for instruction-level parallelism generally increases register pressure, while register allocation reduces the exploitable instruction-level parallelism.

Traditionally, instruction selection is performed on IR trees or at most at the basic block level. Plain instruction scheduling is performed at the basic block level, but

there are techniques for larger control structures: Trace scheduling and superblock scheduling for sequences of basic blocks, and software pipelining for simple loops. Register allocation is usually performed at the function level (global register allocation), but in its traditional form this is an NP-complete problem, whereas register allocation for (already-scheduled) basic blocks can be solved in linear time.

SSA form can be useful when extending work from basic blocks (and superblocks and extended basic blocks[1]) to control structures that include control flow joins. In particular, the three phases of code generation interact with SSA form as follows:

Instruction selection: When extending instruction selection to larger control structures, SSA form has the usual benefit of being a clean representation of data flow; in particular, it avoids false dependencies. Such an extension is the topic of Chapter 24.

Instruction scheduling: The input to instruction scheduling is a data dependence graph; these data dependences include data flow dependences (represented in SSA form), but also other dependences such as write-after-read dependences (aka antidependences) and write-after-write (aka output) dependences through memory (not represented in SSA form at the low level useful as input for code generation). These other dependencies also have to be taken into account in instruction selection. They have to be maintained during various transformations on the SSA form. As a (small) benefit, there is then no need to generate the data dependence graph from the sequence of instructions.

SSA-based register allocation requires that the conflict graphs for basic blocks are interval graphs; therefore the code has to be in a total order for SSA-based register allocation (whereas the data flow graph represented in SSA form, and the data dependence graph are only partial orders). Running the scheduler before SSA-based register allocation is therefore necessary if the compiler does not have a totally-ordered IR in addition to SSA form; running the scheduler before register allocation is usually a good idea anyway, because register allocation introduces write-after-read dependences between registers which hinder scheduling.

Apart from these considerations, SSA form does not particularly affect instruction scheduling, so there is no chapter in this book about instruction scheduling.

If-conversion transforms if-statements into predicated code, where each instruction is executed only when some condition holds; the condition is an additional input to the instruction. I.e., if-conversion converts control flow into data flow. One benefit is that this can avoid branch mispredictions (especially if the condition is hard to predict); another benefit is that the applicability of instruction scheduling techniques for limited control structures (such as software pipelining for simple loops) can be extended (for software pipelining, to loops containing ifs). To achieve the latter benefit even when predicated code is not profitable (because the if-statement is too big and too predictable), one can use reverse if-

[1] Both are single-entry multiple-exit control structures; superblocks are limited to sequences of basic blocks, whereas extended basic blocks allow control-flow trees of basic blocks.

conversion after scheduling. Chapter 26 describes an if-conversion algorithm for SSA form.

Register allocation: The classical view of live ranges leads to arbitrary conflict graphs, for which register allocation is equivalent to graph colouring, which is NP-complete. The SSA-based view introduces register renaming at control-flow joins. As a result, in the conflict-graph view we get chordal graphs, which can be coloured in linear time (after spilling, which is still hard). Or, in the direct view, the control flow that register allocation has to consider is an extended basic block, which can be solved through an extension of the linear basic block scheduling algorithm. This topic is discussed in Chapter 23.

CHAPTER 23

Register Allocation

F. Bouchez
S. Hack

Progress: 40%

Material gathering in progress

Register allocation maps the variables of a program to physical memory locations. The compiler determines the location for each variable and each program point. Ideally, as many operations as possible should draw their operands from processor registers without loading them from memory beforehand. Due to the huge latency of the memory hierarchy (even loading from L1 cache takes three to ten times longer than accessing a register), register allocation is one of the most important optimizations in a compiler.

There is only a small number of registers available in a CPU, with usual values ranging from 8 to 128. Hence, the task of register allocation is not only assigning the variables to registers but also deciding which variables should be evicted from registers and when to store and load them from memory (spilling). Furthermore, register allocation has to remove spurious copy operations (copy coalescing) inserted by previous phases in the compilation process, and to deal with allocation restrictions that the instruction set architecture and the runtime system impose (register targeting).

23.1 Introduction

- *based on graph coloring, linear scan*
- *spilling & coloring are dependent*
- *heuristics are used to find a working solution*
- *avoiding spills is more important than coalescing*

- *Fab: It is important to outline that with decoupled reg-alloc, spilling and coloring become much simpler and more efficient.*

Register allocation is usually performed per procedure. A liveness analysis determines for each variable the program points where the variable is live. A variable is live at a program point if there exists a path from this point on which the variable will be used later on and is not overwritten before that use. Hence, storage needs to be allocated for that variable; ideally a register. The set of all program points where a variable is live is called the *live-range* of the variable. The resource conflict of two variables is called *interference* and is usually defined via liveness: Two variables interfere if (and only if) there exists a program point where they are simultaneously live.[1] The number of live variables at a program point is called the *register pressure* at that program point. The maximum register pressure over all program points in a procedure is called the register pressure of that procedure, or "Maxlive."

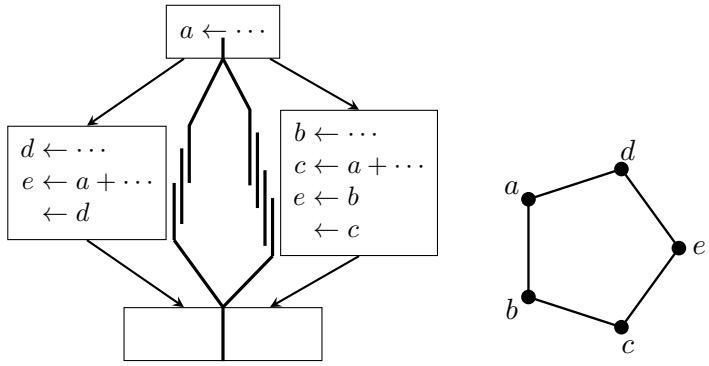
It is helpful to think of interferences as an undirected *interference graph:* The nodes are the variables of the program, and two nodes are connected if they interfere. The set of variables live at some program point form a *clique* in this graph: Any two variables in this set interfere, hence their nodes are all mutually connected. The size of the largest clique in the interference graph is called its *clique number*, denoted ω. A *coloring* of this graph with k colors corresponds to a valid register allocation with k registers. Hence, we will use the terms "register" and "color" interchangeably in this chapter. A k-coloring is a mapping from the nodes of the graph to the first k natural numbers (called colors) such that two neighbouring nodes have different colors. The smallest k for which a coloring exists is called the *chromatic number*, denoted χ. Of course $\chi \geq \omega$ holds for every graph, because the nodes of a clique must all have different colors.

Chaitin et al. [**?**] showed that for every undirected graph, there is a program whose interference graph is this graph. From the NP-completeness of graph coloring then follows the NP-completeness of register allocation. Hence, determining the chromatic number of an interference graph is not possible in polynomial time unless P=NP.

This the major nuisance of classical register allocation: The compiler cannot efficiently determine how many registers are needed. This means one might need more registers than Maxlive, Maxlive being only a lower bound on the chromatic number. The somewhat unintuitive result is that, even if at every program point there are no more than Maxlive variables live, we still might need more than Maxlive registers for a correct register allocation! The causes of this problem are control-flow merges, as can be seen in Figure 23.1.

In that example, the register pressure is at most two at every program point. However, the interference graph cannot be colored with two colors. Its chromatic number

[1] This definition of interference by liveness is an overapproximation. There are refined definitions that create less interferences. However, in this chapter we will restrict ourselves to this definition.



(a) Example program and live-ranges of its variables

(b) Interference graph

Fig. 23.1 Example program and its interference graph

is three. The inequality between Maxlive and the chromatic number is caused by the cycle in the interference graph.[2]

This situation changes if we permit *live-range splitting*. That is, inserting a copy (move) instruction at a program point that creates a new variable. Thus, the value of the variables is allowed to reside in different registers at different times. In Figure 23.1a, assume we split the live-range of e in the left block: we rename it by e' and add a copy $e \leftarrow e'$ at the end of the block. Then, the node e in the graph is split into two nodes, e and e', that do not interfere: this breaks the cycle in the graph, making its chromatic number equal to two, i.e., Maxlive.

In an extreme setting, one could split the live-ranges of all variables after every instruction. The interference graph degenerates into several disconnected components and its chromatic number drops to Maxlive. Of course, such an extreme live-range splitting introduces *a lot* of shuffle code, which degrades the runtime of the compiled program. This interplay between live-range splitting and colorability is the key issue in register allocation.

23.1.1 Non-SSA Register Allocators

As finding a valid k-coloring is NP-complete, assigning registers is performed by some heuristic algorithm. If that algorithm fails to assign a register to a variable, then it either spills this variable to memory, or frees a register by spilling the variable it contains. The problem here is that the spilling decision is made to revive the coloring heuristic, and not because the variable that gets spilled is in itself a good candidate

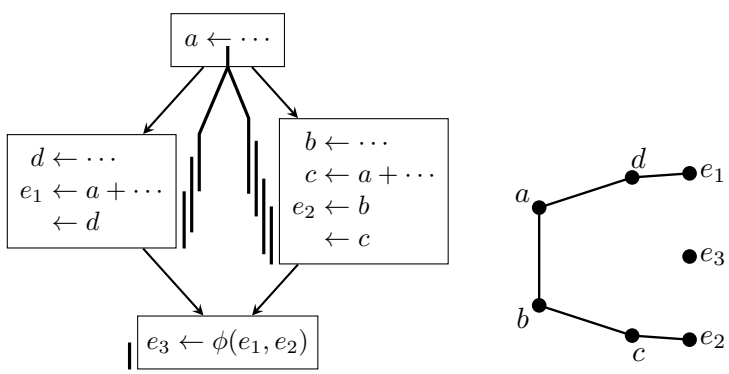
[2] In theory, the gap between the chromatic number and Maxlive can even be arbitrarily large. This can be shown using by applying Chaitin's proof of the NP-completeness of register allocation to Mycielski graphs.

for spilling. Even worse, we might spill a variable because the heuristic is "not good enough," and not because we are actually out of registers.

At the same time, classical algorithms perform copy coalescing (i.e., undoing live-range splitting) during coloring. However, if done aggressively, coalescing may increase the chromatic number of the graph, or make the graph harder to color for the heuristic. Both cases generate additional spills. This is of course unacceptable because we never want to add memory accesses in favor of an eliminated copy. Thus, existing techniques often apply *conservative* coalescing approaches which are guaranteed not to increase the chromatic number of the graph, at the expense of the quality of the coalescing.

23.1.2 SSA form to the rescue of register allocation

The live-ranges in an SSA form program all have a certain property: SSA requires that all uses of a variable are dominated by its definition. Hence, the whole live-range is dominated by the definition of the variable. Dominance, however, induces a tree on the control-flow graph. Thus, the live-ranges of SSA variables are all tree-shaped [**?**, **?**, **?**]. They can branch downwards on the dominance tree but have a single root: the program point where the variable is defined. Hence a situation like in Figure 23.1 can no longer occur: *e* had two "roots" because it was defined twice. Under SSA form, the live-range of *e* is split by a ϕ-function. The argument and result variables of that ϕ-function constitute new live-ranges, giving more freedom to the register allocator since they can be assigned to different registers.



(a) example program in SSA **(b)** Interference graph

Fig. 23.2 SSA version of the example program

This special structure of the live-ranges leads to a special class of interference graphs: Gavril [**?**] showed that the intersection graphs of subtrees are the *chordal graphs*. Chordal graphs can be optimally colored in linear a time with respect to

the number of edges in the graph. Furthermore, they are *perfect*, which means that for every subgraph, the clique number equals the chromatic number. This is a very important property for register allocation because it means that Maxlive equals the chromatic number of the graph;[3] It has even been shown [**?**, **?**] that, for every clique in the interference graph, there is one program point where all the variables of that clique are live. This allows to decouples spilling from coloring: First, lower the register pressure to k everywhere in the program; Then, color the interference graph with k colors in polynomial time.

23.2 Spilling

The term spilling is heavily overloaded: In the graph-coloring setting it often means to spill a node of the interference graph and thus the *entire* live-range of a variable. This implies inserting load instructions in front of every use and store instructions after each definition of the (non-SSA) variables. Other approaches, like some linear scan allocators, are able to keep parts of the live-range of a variable in a register such that multiple uses can potentially reuse the same reload.

23.2.1 Spilling under the SSA Form

As stated above, lowering Maxlive to R ensures that a register allocation with R registers can be found in polynomial time for SSA programs. Thus, spilling should takes place before registers are assigned *and* yield a program in SSA form. For the moment, let us ignore *how* to choose "what" to spill and "where," and concentrate on the implications of spilling in SSA programs. Because many compilers use SSA heavily, it is a reasonable assumption that the program was in SSA before spilling. Hence, we consider spilling as an SSA program transformation that establishes Maxlive $\leq R$ by inserting loads and stores into the program.

23.2.2 Finding Spilling Candidates

How to find program points to place loads and stores such that Maxlive $\leq R$? Let us start with a simplistic setting where the location of loads and stores is not optimized: loads are placed directly in front of uses and stores directly after the definition. Furthermore we consider a *spill everywhere* setting: a variable is either spilled completely or not at all; If spilled, the live-range of a variable then degenerates into

[3] This also implies that chordal graphs do not have "holes" like the interference graph in Figure 23.1. Every cycle is spanned by *chords*.

small intervals: one between the definition and the store, and one between each load and its subsequent use. **TODO:** figure? Flo: if space allows it, yes. Gavril and Yannakakis [**?**] showed that, even in this simplistic setting, it is NP-complete to find the minimum number of nodes to establish Maxlive $\leq R$. Bouchez et al. [**?**] showed that it is even NP-complete to find the minimum number of nodes to spill to decrease Maxlive just by one. Thus, finding spill candidates is not facilitated by SSA.

Moreover, formulations like *spill everywhere* are often not appropriate for practical purposes, and putting the whole live-range to memory is too simplistic. A variable might be spilled because at some program point the pressure is too high; However, if that same variable is later used in a loop where the register pressure is low, spilling everywhere will place a superfluous (and costly!) load in that loop. Spill everywhere approaches try to minimize this behavior by adding costs to variables; These bring in a flow-insensitive algorithm information that a variable reside an frequently executed area. However, such approximations are often too coarse to give good performance. Hence, it is imperative to intelligently split the live-range of the variable according to the *program structure* and spill only parts of it.

More promising spilling techniques are based on Belady's algorithm [**?**]. Belady's algorithm was developed in the context of page replacement in operating systems; When all pages slots are used a new page must be mapped in, the question is: "which page to evict?" Each time this question arise, a different page might be chosen; If an evicted page was brought back, it is not necessarily evicted again. This problem is very similar to register allocation, besides the fact that *putting* a register to memory (evicting a page) also incurs a cost (a store) in the register allocation setting while it does not in the page replacement setting. Belady showed that evicting the page whose next use is farthest in the future leads to the minimum number of page loads. Nevertheless, Farrach and Liberatore [**?**] showed that Belady's algorithm gives good results on straight-line code.

Braun and Hack [**?**] extended Belady's algorithm to general control-flow graphs in two ways. First, they gave a data-flow analysis to compute approximate next use values on the control-flow graph. In this next use calculation, loops are virtually unrolled by a large factor such that code "behind" loop appears far away from code in front of the loop. This forces the eviction algorithm to prefer variables that are not used in loops and to leave parts of live-ranges that are used within loops in registers. Second, they showed how and in which order to process the basic blocks of the control-flow graph in order to determine a reasonable initial register occupation for each block.

23.2.3 Maxlive and colorability of graphs

TODO: seb: do we need this? doing a standard liveness analysis does the job. we can also refer to Appel's liveness algo.

- *definition of Maxlive*
- *show that Maxlive is the minimum number of colors required*

- *scan of the dominance tree sufficient to know Maxlive: pseudo-code*
- *Fab: MAXLIVE minimum if interference graph==intersection graph. Clearly we do not want to go into such details but be careful to use precise and correct terms.*

The register pressure at a program point is the number of variables alive at this point.[4] The greatest register pressure over all a program is called Maxlive, since it is the maximum number of simultaneously alive variables. Obviously, spilling is mandatory if Maxlive is strictly greater than R, the number of registers. Under SSA, the good news is that this is the only case where spilling is required, i.e., if Maxlive $\leq R$, there is no need for spilling. Indeed, Maxlive is the coloring number of the chordal interference graph of the program (**TODO:** ref to chapter 2?). So, we can devise a polynomial test to know whether spilling is necessary or not by computing Maxlive. This can be done by checking on every program point the number of variables that are alive. If liveness information is available on the whole program, the order does not matter. However, in the case of SSA, this is not required as Maxlive can easily be computed without liveness information by performing a scan of the dominance tree, starting from the root. We only need to know which uses are "last-uses." The pseudo-code is given on Figure 23.3. **TODO:** check Maxlive scan needs last uses.

```
function compute-branch (p, live)
  live <- live - { uses in p that are last uses }
  live <- live + { definitions of p }
  maxlive <- max( maxlive, #live )
  foreach child c of p
    compute-branch (c, live)

function compute-maxlive
  maxlive <- 0
  compute-branch (root, {})
```

Fig. 23.3 Pseudo-code for Maxlive

23.3 The problem of spilling

23.3.1 Lowering Maxlive

- *The min number of colors is Maxlive. Variables must be stored in memory to lower the register pressure at points where it is $> R$.*
- *Whenever, Maxlive $\leq R$, we know for sure that spilling more in unnecessary.*

[4] Considering all simultaneously alive variables interfere.

- *Fab: if MAXLIVE≤ R spilling is unecessary under some conditions (we color SSA code, going out of SSA code might require some additionnal spilling. Also without naming constraints). So again be careful about what you say. I think that the idea is to have a register allocator under SSA as simple as possible with a very simple modeling and nice properties. Then we cope with real world issues during the colored-SSA destruction.*

Usually, there are too many variables in a program and Maxlive is greater than R. In that case, we need to lower the register pressure at program points where it exceeds R by spilling, i.e., storing variables in memory instead of registers. Whenever we reach a point where Maxlive $\leq R$, it will be possible to color all SSA variables with the R registers. For now, the difficulty lies in choosing which variables to spill, and where to spill them. Indeed, a careful choice will lead to performance, while making bad decisions can lead to big slowdowns. As accesses to variables in memory require more time that accesses to registers, it is for instance better to avoid spilling variables used inside frequently executed code regions like loops.

TODO: spilling while keeping SSA property

23.3.2 Spilling is a difficult problem

The problem of spilling is a difficult one in the literature, even under one of its simpler forms, the spill everywhere problem, where spilled variables stay so on their entire live-range [**?**]. This spilling is advocated on the first graph-coloring algorithms [**?**] and even in subsequent algorithms [**?**] because it fits better the graph-coloring representation. In these schemes, this amounts to removing nodes from a non R-colorable graph until it becomes R-colorable. Although this node deletion problem is known to be NP-complete, this allows the writing of simple heuristics based on the number of neighbours in the graph and a cost attached to nodes representing the overhead incurred in case they were to be spilled. An added problem is the fact that, for many architectures like RISC, spilled variables cannot be accessed directly from memory but must instead by copied back to the registers, when they are used, or which must be in a register when defined, just before being copied to memory. This creates new variables with very short live-ranges at the definition and use points, which must be accounted for register allocation. This explains why algorithms such as Iterated Register Coalescing must rebuild the interference graph after a phase of spilling and start again, until no more spilling is required.

The actual spilling problem is even more difficult, as variables do not have to be spilled on their entire live-range. Consider a loop with excessive register pressure and a variable that is defined before the loop and used afterwards. Ideally, a compiler would store (spill) the variable in front of the loop and load (reload) the variable after the loop. If the variable was reloaded inside the loop, the reload would be executed in each loop iteration. Another example is a variable that is used in a loop but has already been spilled before the loop. Reloading this variable directly before its use in the loop will cause memory traffic in each loop iteration. Thus, it

is preferable to put the reload in front of the loop. This poses a new problem on top of choosing which variables to spill: now we need also to choose where to spill them, i.e., where to put `load` and `store` instructions. This more general problem where the goal is to minimize the overhead of these added instructions is called the load-store optimization problem and is known to be NP-complete, even for a basic-block [**?**].

23.3.3 SSA does not help for spilling

- *easy spill method: spill everywhere*
- *SSA split points are good for coloring, but not really for spilling (see article spill everywhere under SSA)*
- *other split points to avoid spill everywhere are probably better*
- *Fab: if you need an example to illustrate that SSA does not really help in practice for spilling ask Quentin. It does help a little bit (dynamic programming possible for spill everywhere with few registers; further first can be generalized). But what you want to point it here is the fact that the disadvantages seem more important than advantages.*
- *Fab: the LCTES spill everywhere article does not discuss practical issues (such as the fact that it might insert a load or a store inside a loop...). It is more algorithmical issues.*
- *Seb: Want do we want to say here? I think the title of the section is too negative. Perhaps we should merge the section with the last. I think it would be good to first talk about different definitions for spilling, then give an overview over your results (LCTES paper).*

For the spilling problem, the SSA is not really of much help. Over a blind spill everywhere algorithm on the initial program, it has however the slight advantage of allowing the spilling of SSA variables, i.e., sub-variables of the initial variables. One major disadvantage is the added complexity when spilling variables defined in ϕ-functions **TODO:** explain the problem if not all args are spilled, or if spilled in $\neq$ locations. **TODO:** Seb: complexity? perhaps slight complications :)

Examples by Quentin

```
CSSA :
   a =
   b =
 /       \
c= a   d = b
 \       /
 e = phi(c,d)
 = e
```

After spilling spill :

```
   a =
   b =
 /         \
c= a     d = b
C=st c  D = st d
 \         /
 E = phi(C,D)
 e = ld E
 = e
```

In that case, memory locations C, D and E can be the same. But this does not mean that placement of `store` instruction is the best one.

```
SSA :
   a =
   b =
 /         \
 \         /
 e = phi(a,b)
 = e
```

Spill supposing `stores` are after definition

```
   a =
   A = st a
   b =
   B = st b
/         \
\ <--     / <--
E = phi(A,B)
e = ld E
= e
```

On the two edges pointed by arrows (or on the preceding basic block if they are not critical), additional memory transactions must reconcile the values:

```
tmp = ld X
Y = st tmp
```

This can induce more spill, unforeseen. Without SSA, this problem does not exist: for instance, with spill everywhere, the same memory slot is affected to all definitions of the same variable so no post-treatment is required as it is under SSA.

23.3.4 Spilling under SSA

Still, we give a possibility to perform spilling under SSA

- Hack's algorithm: simplified version?
- à la Belady following dominance tree?
- Fab: The last Algorithm from Sebastian is Belady like. So yes try to simplify it, provide the pseudo code and give the intuitions on how the full version improves it.
- Seb: I think it's ok to briefly outline the approaches that are out there and refer to the papers.

TODO: Ask Sebastian to do it.
[**?**]**TODO:** file braun09cc.pdf in biblio

23.4 Coloring and coalescing

Due to the decoupled register allocation and the use of SSA, the interference graph after spilling has two interesting properties. First, it still is chordal, hence easily colorable in linear time; Second, we know that R colors are sufficient to color it since Maxlive $\leq R$. In this section, we will first present the traditional graph coloring heuristic of Chaitin et al. and show how it successfully color programs under SSA

form. We will then present a lighter coloring algorithm based on the scanning on the dominance tree. Finally, we will then explain how to add *coalescing* to the coloring phase to reduce the number of copies in a program.

23.4.1 Greedy coloring scheme

- *two simplicial nodes*
- *color in the reverse order of simplification*
- *one such order considers subtrees from the root of the tree*
- *no need to construct interference graph*
- *Fab: make it as more intuitive as possible. Avoid as much as possible chordal stuffs which are useless. Greedy coloring and its relationship with the subtree representation is important on the other hand. See with Philip to avoid redundancies between your chapter and his (SSA properties). It is important to outline here that this property that is due to decoupled+SSA enables more aggressive coalescing heuristics. But also simpler (tree-scan).*

In traditional graph coloring register allocation algorithms, the assignment is done by coloring the interference graph with the greedy heuristic of Chaitin et al. [**?**]. This scheme is based on the observation that given R colors—representing the registers—, if a node in the graph has at most $R-1$ neighbours, there will always be one color available for this node whatever colors the remaining nodes have. Such a node node can be *simplified*, i.e., removed from the graph and placed on a stack. This process can be iterated with the remaining nodes, whose degree may have decreased if the simplified node was one of their neighbours. If the graph becomes empty, we know it is possible to color the graph with R colors; Such a coloring can be obtained by assigning colors to nodes in the reverse order of their simplification, i.e., by popping nodes from the stack and assigning them one available color, which is always possible since they have at most $R-1$ colored neighbours. We call this algorithm the *greedy coloring scheme*.

Since it is a coloring heuristic, this scheme can get stuck (whenever all remaining nodes have degree at least R). In that case, we do not know whether the graph is R-colorable or not. In traditional register allocation, this is the trigger for spilling some variables so as to unstuck the simplification process. However, we will see that under the SSA form, after spilling has already been done so that the register pressure is at most R, the greedy coloring scheme can never get stuck.

23.4.2 SSA graphs are colorable with a greedy scheme

One of the interesting properties of SSA form for register allocation is that the live-ranges of variables are subtrees of the dominance tree (see Chapter 2, Section 2.15).

```
Function Simplify(G)
  Data: Undirected graph G = (V, E);
  For all v, degree[v] = #neighbours of v in G,
  k number of colors
stack = {} ;
worklist = {v in V | degree[v] < k} ;
while worklist != {} do
  let v in worklist ;
  foreach w neighbour of v do
    degree[w] = degree[w]-1 ;
    if degree[w] = k - 1 then worklist = worklist U {w}
  push v on stack ;
  worklist = worklist \ {v} ; /* Remove v from G */
if V != {} then Failure ``The graph is not simplifiable''
return stack ;
```

Fig. 23.4 Iskgreedy function

This means the interference graph is chordal and can be represented as the intersection graph of a family of subtrees. We will now prove that if the register pressure is at most R, the interference graph can be colored with R colors using the greedy scheme.

Proof. Let us consider such a subtree representation of the interference graph (see Figure 23.5 TODO) and try to apply the greedy coloring scheme to it. In this representation, a node candidate for simplification corresponds to a live-range which intersect with at most $R-1$ other live-ranges. Consider the "leaf" of a branch of the dominance tree, i.e., the live-range on this branch which starts last, say at point d (the instruction that defines the leaf). If this leaf does not intersect any other live-range, it can be simplified. If it does, then all its neighbour live-ranges start before d—by definition of the leaf—and end after d—because they intersect. So, all neighbours of the leaf are live at d; Since spilling has already been done, we know that the register pressure at d is at most R, hence there is at most $R-1$ intersecting live-ranges: the corresponding node in the interference graph has at most $R-1$ neighbours and can be simplified. Removing a live-range does not change the shape of the graph (the remaining live-ranges are still subtrees of the dominance tree): as long at the graph is not empty, there will still be "leaf" live-ranges, i.e., nodes that can be simplified. Eventually, the simplification process will empty the whole graph, and colors can be assigned in the reverse order of the simplification.

Insert subtree figure HERE.

Fig. 23.5 Subtree

We just proved that if we are under SSA form and the spilling has already been done so that Maxlive $\leq R$, the classical greedy coloring scheme of Chaitin et al. is *guaranteed* to perform register allocation with R without any additional spilling.

This is practical for instance when working on an existing compiler that uses classical register allocation (for instance, the Iterated Register Coalescing of George and Appel [**?**]).

The original simplification scheme of Chaitin et al. does the job, without any modification. Following a call to Is_k_Greedy, it is possible to re-use the stack to assign the colors by making a call to Assign_colors, Figure 23.6. Among the good things, this also means that existing register allocators can be easily modified to handle SSA code and that register allocation can benefit from powerful coalescing strategies such as aggressive or conservative ones. Moreover, the fact that register allocation can be decoupled into a phase of spilling first and then a phase of coloring/coalescing allows the writing of more involved coalescing **TODO:** cite examples.

However, this algorithm still needs the interference graph, an additional structure that takes time to build and uses memory up. We will propose next a lighter coloring algorithm based on the insight given by the greedy coloring scheme.

```
Function Assign_colors(G)
  available = new array of size R and values True
  while stack != {} do
    v = pop stack ;
    for each neighbour w in G
      available[color(w)] = False
    col = 0;
    for each color c from 1 to R
      if available[c]
        col = c
      available[c] = True /* prepare for next round */
    color(v) = col
    add v to G
```

Fig. 23.6 Assign_color function

23.4.3 A light tree-scan coloring algorithm

- *On the dominance tree, possible to scan from top and assign colors to variables as they come.*
- *Biased coloring is used to coalesce variables linked by ϕ-functions.*
- *see next section for more involved coalescing*
- *Fab: I think that you already talked about scan coloring. Ok to talk about biased.*
- *Fab: You can talk about biased coloring that uses the result of an aggressive coalescing (see with Quentin). It is not more complicated and it improves results.*

One of the advantages of SSA is to make things simpler, faster, and use less memory. For the register assignment problem, we have seen that the existing greedy

scheme based on simplification still works; However, it requires constructing and maintaining an interference graph, a structure judged to big and cumbersome to be used in a JIT context, where compilation time and memory prints matter more. We will now present a method to perform a fast register assignment for SSA after spilling which does not need more than the dominance tree and def-use chains.

The algorithm we propose scans the dominance tree, coloring the variables from the root to the leaves in a top-down order. The variables are colored in the order of their definitions. A pseudo-code is given in procedure Tree_Scan, Figure 23.7. Intuitively, this method works because when the scanning arrives at the definition of a variable, the only colored variables are "above" it and since there is at most $R-1$ other variables live at the definition, there is always a free color. We will now prove that this method always work; The key observation is that the order in which variables are considered corresponds to a reverse order of simplification, i.e., a valid order of the nodes on the stack after the greedy simplification scheme.

Proof. Let us consider an ordering $v_1, v_2, \ldots, v_n$ of the nodes of the nodes based on the dominance: if v_i dominates v_j, then v_j appear before v_i in the ordering ($j < i$). We will show that this order correspond to a greedy simplification. Suppose $v_1, \ldots, v_{i-1}$ have been simplified by the greedy scheme (for $i = 1$, this is the initial graph). If v_i dominates any variable, say v_j, then $j < i$ and the node has already been simplified; So, there is no variable defined after v_i on the dominance tree: the live-range of v_i is a "leaf" of the subtree representation (see proof in Section 23.4.2). So, v_i can be chosen for simplification, and by induction the ordering $v_1, \ldots, v_n$ is a valid order of simplification. As we have seen previously in Section 23.4.1, this means the reverse order $v_n, v_{n-1}, \ldots, v_1$ is a valid order for coloring.

It is important to understand why this method does not work in the general non-SSA case. Under SSA, the variables are "split" at join points by the use of ϕ-functions. If this was not the case, we would have live-ranges that spans on multiple branches of the dominance tree, creating cycles in the representation. In that case, coloring a branch would constrain the coloring on leaves on a different branch; Under SSA form, such live-ranges are split and their colorings are independent. See Figure **??** for an example.

The tree-scan coloring algorithm is really fast as it only needs one traversal of the dominance tree. Since the number of colors is fixed and small, it can be implemented as a bit-set to speed-up updates of the `available` array. The pseudo-code for function `choose_color` is deliberately not given yet. A very basic implementation could just scan the `available` array until it finds one that is not taken. We will see in the next section how to bias the choosing of colors so as to perform some coalescing.

```
Tree_Scan(T)

  function assign_color(p, available)
    for each v last use at p
      available[color(v)] = True     /* colors not used anymore */

    for each v defined at p
      c = choose_color(v, available)  /* choose available color */
      available[c] = False
      color(v) = c

    for each child p' of p
      assign_color(p', available)

  assign_color(root(T), [True, True, ..., True])
```

Fig. 23.7 Tree scan coloring algorithm for SSA.

23.4.4 Coalescing under SSA form

The goal of *coalescing* is to minimize the number of register-to-register `move` instructions in the final code. While there may not be so many such "copies" at the high-level (e.g., instruction "`a = b;`" in C)—especially after a phase of copy propagation under SSA (**TODO:** See chapter X)—, many such instructions are added by different compiler phases by the time compilation reaches the register allocation phase. For instance, adding copies is a common way to deal with register constraints (see Section 23.5). An even more obvious and unavoidable reason in our case is the presence of ϕ-functions due to SSA form. Indeed, remember that a ϕ-function represents in fact parallel copies on incoming edges of basic blocks. For instance, if the function is $a \leftarrow \phi(b, c)$, then it means that instructions $a \leftarrow b$ should be executed on the edge coming from the left, and $a \leftarrow c$ on the one from the right. Suppose now that register allocation decided to put a and b in register R_1, but c in register R_2; Then, the copy $a \leftarrow b$ does not need to be executed anymore, but $a \leftarrow c$ still does: the value of variable c will be contained in R_2 and needs to transfered to R_1, which means the final code will contain an instruction "`mov` R_1, R_2" or the like.

In the case of SSA, it is for instance obviously better to assign variables linked by a ϕ-function, to the same register, so as to remove copies between subscripts of the same variable (a_1, a_2, etc.). This is possible if we are under CSSA, but optimizations can break this property and interferences can appear between subscripted variables with the same origin. In order to bypass this problem and also allow for more involved coalescing between SSA variables of different origin, we will define a notion of *affinity*, acting as the converse of the relation of interference and expressing how much two variables "want" to share the same register. We add a metric to this notion, measuring the benefit one could get if the two variables are assigned to the same register. Given two variables a and b, the weight of the affinity between

them is the number of occurrences of copy instructions involving them ($a \leftarrow b$ or $b \leftarrow a$) or ϕ-functions ($a \leftarrow \phi(\ldots, b, \ldots)$ or $b \leftarrow \phi(\ldots, a, \ldots)$). These numbers should of course be weighted as a copy in a loop costs more than one outside a loop for instance. We recommend using using actual execution frequencies based on profiling in possible, and empirically parameters if not (e.g., $\times 10$ for each level of nested loop, $\times 0.5$ if in a conditional branch, etc.).

23.4.4.1 Biased coalescing

Since the number of affinities is sparse compared to the number of possible pairs of variables, we propose to keep for each variable v an "affinity list" where each element is a pair (u, w) meaning that v has an affinity with u of weight w. We propose a version of function `choose_color` that bias the choosing of colors based on these affinity lists, shown on Figure 23.8. The bias is very simple, each choice of color maximizes locally, under the current knowledge, the coalescing for the variable being colored. It is not of course optimal since the general problem of coalescing is NP-complete [**?**], and biased algorithms are known to be easily misguided.

```
global variables
  count = array of size #colors and elements 0

choose_color(v, available)
  for each (u,weight) in affinity_list(v)
    count[color(u)] += weight
  max_weight = -1
  col = 0
  for each color c
    if available[c] == true /* color is not used by a neighbour of v */
      if count[c] > max_weight
        col = c
        max_weight = count[c]
    count[c] = 0 /* prepare array for next call to choose_color */
```

Fig. 23.8 Choosing a color with a bias for coalescing.

23.4.4.2 Aggressive coalescing to improve biased coalescing.

There is another simple and interesting idea to improve biased coalescing. We still consider a JIT context where compilation speed is very important. **TODO:** what kind of aggressive? Ask fab...

23.4.4.3 More involved coalescing strategies

It is beyond the scope of this book to describe in details deeply involved strategies for coalescing. So we will just give a short overview of other coalescing schemes that might be use in a decoupled register allocator.

- Iterated Register Coalescing: conservative coalescing that works on the interference graph
- Brute force strategy, working on all "greedy-k-colorable" graphs

23.5 Practical and advanced discussions

23.5.1 Handling registers constraints

- *ABI contraints*
- *split beforehand solution: many parallel copies, need good biased coloring*
- *repair afterwards solution: biased try to give right color, else, actual copies are added*
- *Fab: For the repair afterward at the time we will publish the book you will be able to cite the current paper we write on tree-scan coalescing. We plan to elaborate on the repairing mechanism and illustrate it in the context of tree-scan.*
- *Fab: For the repair afterward you will build an interference graph with negative affinity weights. Current coalescers do not handle it. A workaround (but costly) for affinity adge (a,b) with weight W¡0 is replace it by an interference edge (a,ab) and an affinity edge (ab,b) of weight -W with ab a dummy node. For iterated reg-alloc (Briggs & George's test) you can rewrite the different rules by doing as if you had such a dummy node (you virtualize it). You can discuss this with Quentin. Of course the idea is not to describe it here but it can be interesting to discuss a little bit the implications.*

In theory, register allocation algorithms are always nicely working with nodes and colors. In practice, however, not all variables or registers are equivalent. Depending on the architecture, some registers might be dedicated to perform memory operations, some instructions expects their operands to reside in particular registers, and in general conventions are defined to simplify for example the writing of libraries, by specifying how parameters to functions are passed and how results are returned. This adds constraints to the register allocation, usually by restricting the coloring possibilities of variables. Some examples are given in Figure .

The problem with such constraints is that they cannot be expressed directly in the register allocation problem. For instance, if variable a and b must absolutely reside in register R_1, it is not possible to pre-color them with color R_1 as the interference graph could not be chordal anymore. Furthermore, a and b maybe interfere, for instance if they are both the first parameter of successive function calls, it that

Context	Constraint	Instruction	Effect on reg. alloc.
x86	Division in registers R_x and R_y	a/b	a in R_x, b in R_y
st200	Memory load cannot use R_z	$a = load(b)$	b cannot be in R_z
ABI	Functions arguments in $R_1, R_2 \ldots$	$a = f(b, c)$	b in R_1, c in R_2

Fig. 23.9 Examples of register constraints.

case putting them in the same register without check would break the program. We propose two solutions to deal with this problem. The first is classic in the literature, and the second is newer and promising.

23.5.1.1 Splitting variables to handle register constraints.

Traditionally, variables involved in constraining operations are split before and after the operation. For instance, if a is involved in a division on an x86, the instructions $a' \leftarrow a$ and $a \leftarrow a'$ are issued before and after the division so that, if a is not in the right register, it can be moved in it for the operation. This however still poses the problem of variable a' which still interfere with every variable alive during the division, and hence is part of the interference graph. Moreover, a is redefined which breaks the SSA. A workaround solution is to split *all* variables alive before and after the operation using a *parallel copy*. New subscripted variables must be defined by the second split to keep the SSA property (and subsequent uses must be changed accordingly). The use of parallel copies assures that the locally created variables, with very short live-ranges (span of only one instruction), are completely disconnected from the interference graph. Their only relation with the other, "normal" variable are affinities that each variable share with the two other parts coming from the same original variable. In that case, the danger is that many copy instructions can be added if the coalescing is not good enough to assign most of the created copies to the same register.

23.5.1.2 Repairing problems afterwards.

Another possibility is to let register allocation do its job, and then intervene to repair the coloring whenever it does not fit the constraints, by adding copies afterwards around mismatches. To minimize the number of conflicts on constraints, it is however important to drive the coloring so that it still gives the right color whenever possible. For example, if variable a must reside in register R_1, it is possible to consider a clique of R nodes otherwise disconnected from the graph, one for each register, and adding R_1 in the affinity list of a. On the contrary, if a cannot be assigned to a register, say R_2, an affinity of *negative weight* is added to the list, representing the added cost it will required if a is still put in register R_2. Existing coalescing algo-

rithms in the literature currently do not support negative weight **TODO:** see Fab's hack.

23.5.2 *Out-of-SSA and critical edge splitting*

- *ϕ-functions are not machine instructions, they are replaced by actual copies in out-of-SSA phase*
- *in our case, colored SSA: Sreedhar not possible*
- *SSA implicitly actual "parallel copies" are placed on edges*
- *problem with some critical edges: abnormal edges, back-edge of loop*
- *Fab: Sreedhar IS possible. It generates local variables not colored. The point is that its coloring might be stucked (Chaitin reduction). So the possible need to split edges...*

As mentioned previously, the ϕ-functions are not actual machine instructions. They must be removed when going "out-of-SSA." In traditional translation out-of-SSA, this happens before register allocation, but in our case, variables are already allocated to memory or registers, and arguments and results of ϕ-functions are not variables anymore but registers.

If the definition and arguments of a ϕ-function have been allocated to the same register, then it can be safely removed. If this is not the case, however, the some `move` instructions must be added to keep the behaviour of the program.

23.5.3 *More repairing on ϕ-functions*

- Biased coloring is fast but not very good at coalescing
- Basic parallel copy motion: move to next edge.
- More involved problem: spill variables
- Better solution: look at biblio (parallel copy motion)
- Fab: Yes make it as simple as possible. Local pcopy motion works in practice. It is useless to go further. The ultimate solution (Sreedhar based) might have been discussed in the previous section

CHAPTER 24

Instruction Selection

D. Ebner
A. Krall
B. Scholz

Progress: 52%

Review in progress

24.1 Introduction

Instruction selection is a transformation step in a compiler that translates a machine-independent intermediate code representation into a low-level intermediate representation or to machine code for a specific target architecture. Instead of hand-crafting an instruction selector for each target architecture, generator tools have been designed and implemented that generate the instruction selector based on a specification of the machine description of the target. The study and implementation of generator tools is mainly motivated by implementing compiler back-ends efficiently and effectively using specifications, and represent a first step towards fully retargetable compilers. Generators for instruction selectors are used in large compiler infrastructures including GCC [**?**] and LLVM [**?**] that target a range of architectures. A possible scenario of a code generator in a compiler is depicted in Figure 24.1. The Intermediate Representation (IR) of an input program is passed on to an optional lowering phase that breaks down instructions and performs other machine dependent transformations such as assuming concrete data sizes of data types. Thereafter, the instruction selection performs the mapping to machine code or lowered IR based on the machine description of the target architecture.

The approaches in literature can be classified based on the scope of translation. A translation unit may be a single statement, a basic block, or a whole procedure of

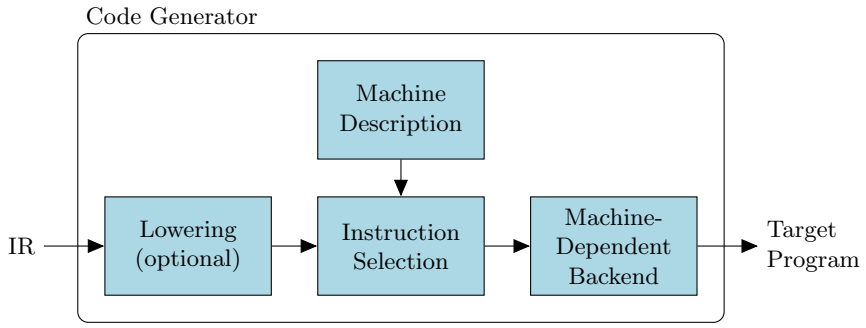


Fig. 24.1 Scenario: An instruction selector translates a compiler's IR to a low-level machine-dependent representation.

the input program. The use of the Static Single Assignment (SSA) form improves the code generation for most instruction selection approaches because it makes def-use relationships explicit. Hence, SSA exposes the data flow of a translation unit and utilizes the code generation process. The approaches in literature can be further classified at which stage of the compilation the machine description is used to instrument a generic instruction selection algorithm: (1) either during compile time or (2) at generation time of the instruction selector by generating a specialized version of the algorithm based on the specification of the machine description. We refer to the generation time of the instruction selection as *compiler compile time*.

In the following we give a brief overview of the most important techniques applied in practice ordered by increasing scope of translation. This includes efficient local techniques such as RTL-based instruction selection and (tree) pattern matching techniques. The latter is discussed in more detail as it lays the foundations for a more general whole-procedure instruction selection techniques based on SSA form discussed in the rest of this chapter.

24.1.1 RTL-Based Instruction Selection

Register Transfer Lists (RTL) are a common machine-independent IR employed in several compiler infrastructure including Zehpyr/VPO [**?**] and GCC [**?**]. An RTL is a simultaneous composition of list of effects. An effect computes the value of an expression and stores it in a location, which is either a single mutable storage cell or an aggregate of mutable cells. The value of an expression can either be constant, fetched from a storage location, or the result of the application of an operator to a list of argument expressions. Even though an RTL is machine-independent, its semantics depend on the particular target architecture.

Code generators in RTL-based compilers are based on the four-step scheme depicted in Algorithm 6. Forward substitution forms more complex expressions by

Warning: package “algorithmic” is no longer used. Please use algorithm2e instead.

```
[1]
\STATE Forward substitution
\STATE Combination of independent effects
\STATE Removal of useless effects
\STATE Validation based on a machine-description
```

Algorithm 6: RTL-based code generation scheme.

substituting sub-expressions with their defining RTL. Likewise, step (2) combines individual RTLs to a single expression. After application of these rules, the results are verified against the machine descriptions. Step (3) simplifies redundant effects that might result from the application of the previous steps. RTL expressions that do not represent a native instruction are discarded in step (4).

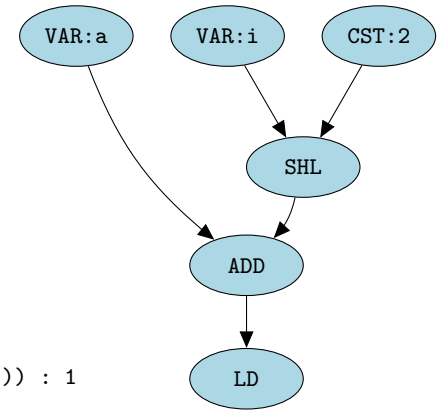
The scope of this approach is limited to simple patterns and constitutes a form of “poor-man’s instruction selection” achieving only locally optimal code. Thus, the code quality of RTL-based compilers such as GCC strongly depends on post-pass RTL-based optimizations such as common subexpression elimination (CSE) that make up for missed opportunities.

24.1.2 Tree Pattern Matching

Tree pattern matching is more sophisticated than RTL-based instruction selection, and it is widely used in modern compiler frameworks. The technique was introduced by Aho and Johnson [**?**]. The unit of translation is a single statement represented in the form of a data flow tree (DFT). The basic idea is to describe the target instruction set using an *ambiguous* cost-annotated graph grammar. The instruction selector seeks for a cost-minimal *cover* of the DFT. Each of the selected rules have an associated semantic action that is used to emit the corresponding machine instruction, either by constructing a new intermediate representation or by rewriting the DFT bottom-up.

An example of a DFT along with a set of rules representing valid ARM instructions is shown in Figure 24.2. Each rule consists of terminal symbols that are shown in upper-case, and nonterminals that are shown in lower-case. Terminal symbols match the corresponding labels of the dataflow trees. Nonterminals are used to chain individual rules together. Rules that translate from one nonterminal to another are called *chain rules*. Nonterminal `s` denotes a distinguished start symbol. Note that there are multiple possibilities to obtain a cover of the data-flow tree using the rule fragment to the left. Each rule has associated costs. The cost of a tree cover is the sum of the costs of the selected rules.

Aho and Johnson [**?**] were the first to propose a dynamic programming algorithm in order to obtain cost-minimal covers. The approach of Aho and Johnson was further refined in [**?**], and tree-pattern matching can be performed in linear time by




```
s   <- reg                  : 0
reg <- imm                  : 1
imm <- IMM                  : 0
reg <- VAR                  : 0
reg <- SHL(reg, reg)        : 1
reg <- SHL(reg, imm)        : 1
reg <- ADD(reg, reg)        : 1
reg <- LD(reg)              : 1
reg <- LD(ADD(reg, reg))    : 1
reg <- LD(ADD(reg, SHL(reg, imm))) : 1
```

Fig. 24.2 Example of a data flow tree and a rule fragment with associated costs.

pre-computing static lookup tables at compiler compile time. Most implementations rely on the following two-pass scheme:

(1) *labeling:* use dynamic programming in order to determine a minimum-cost cover of the given DFT bottom-up.
(2) *reduce*: traverse the DFT in postfix order and execute the semantic actions associated with the chosen rules to obtain a semantically equivalent target program.

One of the more popular existing implementations is the tool `burg` [**?**] that converts a specification in the form of a tree grammar into an optimized tree pattern matcher written in `C`. While `burg` computes costs at compiler compile time and thus requires constant costs, `iburg` [**?**] can handle dynamic costs by shifting the dynamic programming algorithm to instruction selection time. This allows the use of dynamic properties for cost computations, *e.g.*,, concrete values of immediates. The additional flexibility is traded for a small penalty in runtime. More recent techniques [**?**] save the computed states for tree nodes in a lookup table. This approach retains the flexibility of dynamic cost computations at nearly the speed of precomputed tree parsing automata.

Tree-pattern matching approaches are limited in scope to simple statements. DAG matching techniques are an approach to overcome these limitations. However, DAG matching is an NP-complete problem [**?**] in general. A straight forward generalization [**?**] of the algorithm discussed so far is to treat DAGs as if they were trees and make local choices irrespective of shared subgraphs. It turns out that this approach is exact for a specific class of grammars but fails to deliver optimal solutions in the general case. For code generation techniques that exploit aggressively the properties of the target machine, it is useful to increase the scope of instruction selection further to whole procedures at a time. An excellent underlying data-structure for code generation is the SSA graph [**?**] that is an extension of DFTs and represent the data-flow of a procedure in SSA form. In the following, we will demonstrate how to reduce the instruction selection problem in SSA graphs to a

specialized quadratic assignment problem (PBQP) [**?**]. An extension of *patterns* to arbitrary acyclic graph structures which we refer to as DAG grammars is discussed in Section 24.5.1 for details. These extension are useful in the presence of SIMD instructions of CPUs.

As we move from acyclic linear code regions to whole-functions, it becomes less obvious in which basic block, the selected machine instructions should be emitted. For chain rules, the obvious choices are often sub-optimal. We will discuss in Section 24.5.2 a polynomial-time algorithm based on generic network flows that can be applied to generate more efficient code based on a flexible cost model.

24.2 Motivation

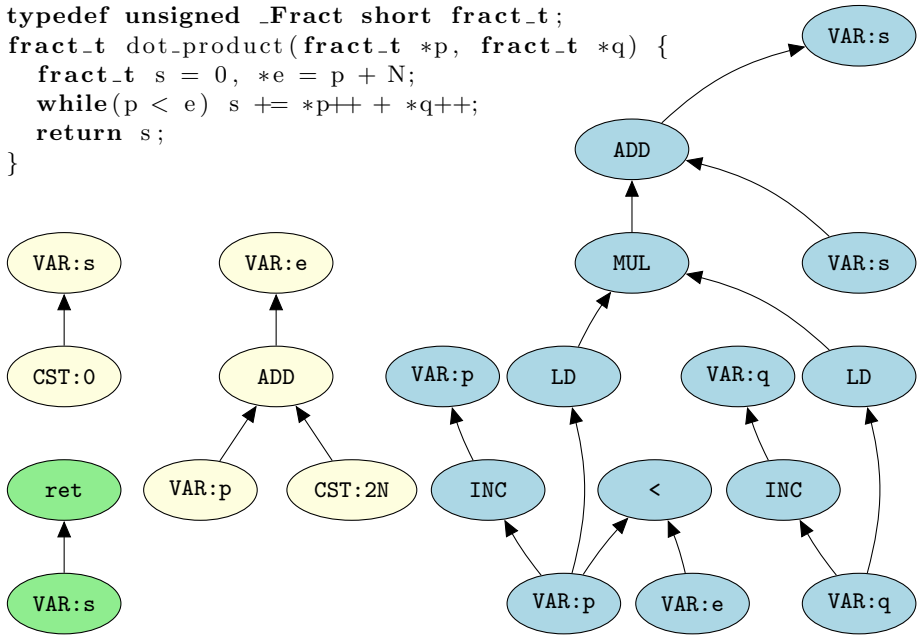


Fig. 24.3 Instruction selection DAGs for a vector dot-product in fixed-point artithmetic.

We demonstrate that DAG based instruction selection is not sufficient to generate optimal code as shown in Figure 24.3. For the example we assume an ARM-like RISC architecture. The code implements a simple vector dot-product using fixed-point arithmetic in Embedded C. The color of DAG nodes depicts the particular basic block they belong to, and we assume the notion of type annotation to distinguish fixed-point data types from regular integer arithmetic.

For sake of simplicity, we neglect the details including numerical precision, loop control code, and we disregard output- and anti-dependencies that have to be respected in order to maintain semantic equivalence. For example, we have to emit the post-increment for variables `p` and `q` *after* the corresponding loads which is neglected.

In our grammar fixed-point values are represented by nonterminal symbol `fp` and the cost function separates fixed-point arithmetic from general integer arithmetic, *i.e.*,, we alter the grammar introduced in Figure 24.2 as shown below:

```
reg <- VAR : is_fixed_point() ? ∞ : 0
fp  <- VAR : is_fixed_point() ? 0 : ∞
```

For fixed-point values most arithmetic and bitwise operations are identical to their integer equivalents. However, some operations have a different semantics, *e.g.*,, multiplying two fixed-point values in format $m.i$ results in a value with $2i$ digits of fraction. The result of the multiplication has to be adjusted by a shift operation to the right. The adjustment can be modeled for an ARM-like architecture by following rule fragment that models double-precision fixed-point values with nonterminal `fp2`:

```
fp2 <- MUL(fp, fp)    : 1   { MUL Rd, Rm, Rs }
fp  <- fp2            : 1   { LSR Rd, Rm, i  }
fp  <- ADD(fp, fp)    : 1   { ADD Rd, Rm, Rs }
fp2 <- ADD(fp2, fp2)  : 1   { ADD Rd, Rm, Rs }
```

We can perform the accumulation for double-precision fixed point values at the same cost as for `fp`. Thus, it would be beneficial to move the necessary shift from the inner loop to the return block, performing the intermediate calculations in the extended format. However, as a DAG based matcher processes basic block by basic block, the information of having values as double-precision cannot be hoisted across basic block boundaries.

To overcome this limitation, we can extend the scope of the matching algorithm to the computational flow of a whole procedure. SSA graphs provide a very suitable vehicle for performing instruction selection for manifold reasons: First, in contrast to classical tree or DAG representations, SSA graphs enable the matching of circular flow stemming from program loops. Second, SSA graphs often arise naturally in modern compilers as intermediate code is already in SSA form. Third, the set of edges fully specifies the set of data dependencies that have to be respected as there are no additional output or anti-dependencies. These additional dependencies significantly complicate or limit DAG based techniques.

The corresponding SSA graph for the example introduced in Figure 24.3 is shown in Figure 24.4. Nodes in the SSA graph represent a single operation while edges describe a flow of data that is produced at the source node and that is consumed at the target node. Incoming edges have an order which reflects the argument order of the operation. In the figure the color of the nodes show to which basic block the operations belong to. An instruction selector operating on the SSA graph is free to propagate nonterminal `fp2` across the ϕ node prior to the return and emits the code for the shift to the right in the return block.

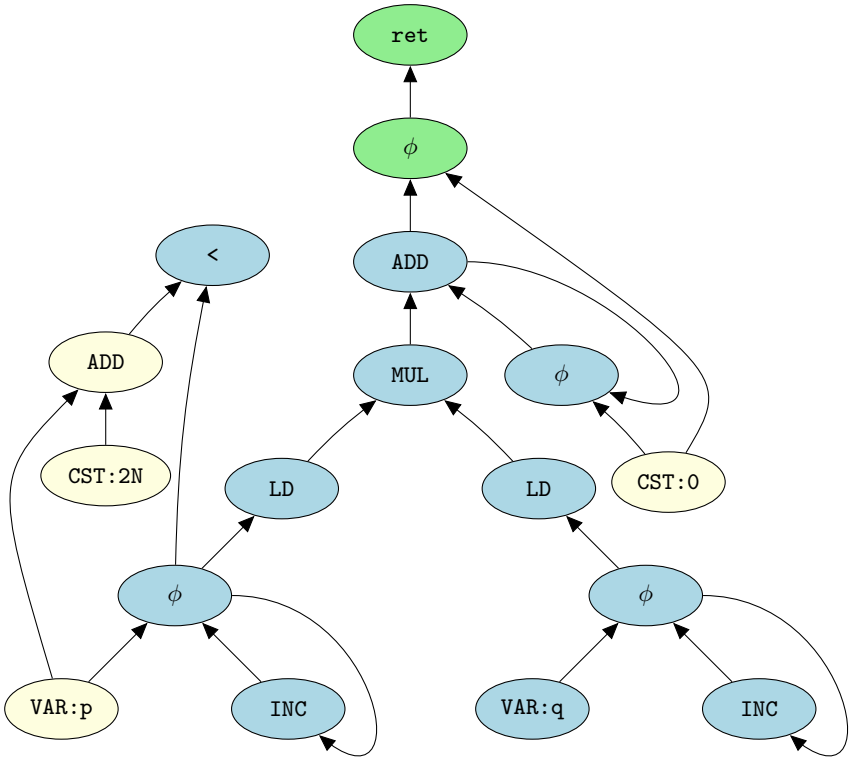


Fig. 24.4 Corresponding SSA graph for the example introduced in Figure 24.3.

As SSA graphs are potentially cyclic, no dynamic programming approach can be employed for instruction selection. To get a handle on code selection for SSA graphs, we will discuss in the following an approach based on a reduction to a quadratic mathematical programming problem (PBQP) [**?**, **?**, **?**] that has been adopted by at least two major embedded system compiler vendors. We continue with a short introduction to PBQP before discussing the main problem formulation and two extensions that allow for more flexible grammars and optimal chain code placement.

24.3 Partitioned Boolean Quadratic Programming

Partitioned Boolean Quadratic Programming (PBQP) is a generalized quadratic assignment problem that has proven to be effective for a wide range of applications in embedded code generation, *e.g.*,, instruction selection, register assignment, address mode selection [**?**], or bank selection for architectures with partitioned memory [**?**].

Instead of problem-specific algorithms, these problems can be modeled in terms of generic PBQPs that are solved using a common solver library. PBQP is flexible enough to model irregularities of embedded architectures that are hard to cope with using traditional heuristic approaches.

24.3.1 Problem Definition

A PBQP is a generalized quadratic assignment problem proposed by Scholz and Eckstein for various sub-problems of embedded code generation [**?**, **?**]. Consider a set of discrete variables $X = \{x_1, \ldots, x_n\}$ and their finite domains $\{\mathbb{D}_1, \ldots, \mathbb{D}_n\}$. A solution of PBQP is a mapping h of each variable to an element in its domain. The quality of a solution is based on the contribution of two sets of terms,

1. for assigning variable x_i to the element d_i in $\mathbb{D}_i$. The quality of the assignment is measured by a *local cost function* $c(x_i, d_i)$,
2. for assigning two related variables x_i and x_j to the elements $d_i \in \mathbb{D}_i$ and $d_j \in \mathbb{D}_j$. We measure the quality of the assignment with a *related cost function* $C(x_i, x_j, d_i, d_j)$,

and the total cost of a solution h is given as,

$$f = \sum_{1 \le i \le n} c(x_i, h(x_i)) + \sum_{1 \le i < j \le n} C\left(x_i, x_j, h(x_i), h(x_j)\right). \tag{24.1}$$

The PBQP problem seeks for an assignment with minimum total costs.

We can alternatively formulate PBQP using matrix calculus: a discrete variable x_i is represented as a boolean vector $\overrightarrow{x_i}$ whose elements are zeros and ones and whose length is determined by the number of elements in its domain $|\mathbb{D}_i|$. Each 0-1 element of $\overrightarrow{x_i}$ corresponds to an element of $\mathbb{D}_i$. An assignment of x_i to d_i is represented as a unit vector whose element for d_i is set to one. Hence, a valid assignment for a variable x_i is modeled by the constraint $\overrightarrow{x_i}^T \overrightarrow{1} = 1$ that restricts vectors $\overrightarrow{x_i}$ such that exactly one vector element is assigned one; all other elements are set to zero.

The related cost function $C(x_i, x_j, d_i, d_j)$ is decomposed for each pair (x_i, x_j). The costs for the pair are represented as matrix C_{ij}. A matrix element corresponds to an assignment (d_i, d_j). Similarly, the local cost function $c(x_i, d_i)$ is mapped to cost vectors $\overrightarrow{c_i}$. We can formulate a PBQP as the following mathematical program:

$$\min f(X) = \sum_{1 \le i \le n} \overrightarrow{x_i}^T \overrightarrow{c_i} + \sum_{1 \le i < j \le n} \overrightarrow{x_i}^T C_{ij} \overrightarrow{x_j} \tag{24.2}$$

$$s.t.\ \forall 1 \le i \le n : \overrightarrow{x_i} \in \{0, 1\}^{|\mathbb{D}_i|} \tag{24.3}$$

$$\forall 1 \le i \le n : \overrightarrow{x_i}^T \overrightarrow{1} = 1 \tag{24.4}$$

A solution satisfying Constraints 24.3 and 24.4 maps each boolean decision vector $\overrightarrow{x_i}$ to a binary vector that contains a single one element. This defines a one-to-one

mapping among decision vectors $\overrightarrow{x_i}$ and elements in their domain $\mathbb{D}_i$. Thus, the domain for the objective function is the cross product of the domains for the individual variables $\mathbb{D}_1 \times \cdots \times \mathbb{D}_n$. The solution to a PBQP is not necessarily unique, *i.e.,*, there are in general multiple solution vectors $\bar{X}$ with the same minimal objective value.

A PBQP problem has an underlying graph structure graph $G = (V, E, C, c)$, which we refer to as a PBQP graph. For each decision vector $\overrightarrow{x_i}$ we have a corresponding node $v_i \in V$ in the graph, and for each cost matrix $C_{i,j}$ that is not the zero matrix, we introduce an edge $e = (v_i, v_j)$. The cost functions c and C map nodes and edges to the original cost vectors and matrices respectively. Note, that due to the properties of quadratic forms, there is an implicit reverse edge $e' = (v_j, v_i)$ for each edge $e = (v_i, v_n)$ and $C(e) = C(e')^T$. A PBQP graph is free of self- and multi-edges. We will present an example later in this chapter in the context of instruction selection.

In general, finding a solution to this minimization problem is NP hard. However, for many practical cases, the PBQP instances are sparse, *i.e.,*, many of the cost matrices $C_{i,j}$ are zero matrices and do not contribute to the overall solution. Thus, optimal or near-optimal solutions can often be found within reasonable time limits. We will discuss two algorithmic approaches for PBQP that have been proven to be efficient in practice for instruction selection problems, *i.e.,*, a polynomial-time heuristic algorithm and a branch-&-bound based algorithm with exponential worst case complexity.

24.3.2 Heuristic Algorithm

In this section, we introduce an algorithm for PBQP proposed by Scholz and Eckstein [**?**, **?**]. For a certain subclass of PBQP, their algorithm produces provably optimal solutions in time $O(nm^3)$, where n is the number of discrete variables and m is the maximal number of elements in their domains, *i.e.,*, $m = \max(|\mathbb{D}_1|, \ldots, |\mathbb{D}_n|)$. For general PBQPs, however, the solution may be sub-optimal.

The algorithm works in three phases as shown in Algorithm 7. First, the PBQP graph is reduced until only nodes of degree 0 are left. For these nodes, a solution can easily be found using a local minimum computation. In a last step, the eliminated nodes are re-inserted in reversed order thereby computing a solution for the original problem instance.

24.3.2.1 Phase I: Reduction.

Eliminating nodes from the PBQP graph is equivalent to the elimination of decision vectors from the original problem. The algorithm removes nodes from the graph until only nodes of degree 0 remain. Nodes to be removed are chosen according to their degree in increasing order. In practice, this can be accomplished efficiently by putting nodes into buckets according to their degree. The algorithm selects a node from the bucket with the smallest index that is non-empty.

Warning: package "algorithmic" is no longer used. Please use algorithm2e instead.

```
\STATE \COMMENT{\textbf{\emph{Phase I: reduction}}}
\WHILE{$\exists v \in V : \deg v > 0$}
\STATE choose vertex $v \in V: 0 < \deg (v) \leq \deg (v') \ \forall v'
\in V : \deg(v') > 0$
\IF{$\deg(v) == 1$}
\STATE \textbf{RI}(v)
\ELSIF{$\deg(v) == 2$}
\STATE \textbf{RII}(v)
\ELSE
\STATE \textbf{RN}(v) \COMMENT{solution may be sub-optimal if RN is applied}
\ENDIF
\STATE remove $v$ from the graph
\ENDWHILE
\STATE
\STATE \COMMENT{\textbf{\emph{Phase II: trivial solution}}}
\FORALL{$v_i \in V$}
\STATE determine solution for $v_i$ by finding the minimum element
in $c_i$
\ENDFOR
\STATE
\STATE \COMMENT{\textbf{\emph{Phase III: back-propagation}}}
\STATE re-insert the remaining nodes in reversed elimination order
until a solution for the original graph is obtained
```

Algorithm 7: PBQP Heuristic

Nodes with degree one or two can be eliminated without loss of optimality using reductions RI and RII respectively. If the algorithm reaches a point where only nodes with degree three or more are left, the problem becomes irreducible and a heuristic is applied, which is called RN. The heuristic chooses a beneficial discrete variable and a good assignment for it by searching for local minima. The obtained solution is guaranteed to be optimal if the reduction RN is not used [**?**].

RI Reduction.

Let v_i be a node with degree one and let v_j denote its only adjacent vertex in the PBQP graph, we can eliminate v_i and the incident edge (v_i, v_j) simply by removing them from the PBQP graph. The cost vector associated with v_j is incremented by the minimum costs over all possible choices of v_i. More formally, the updated cost vector $\overrightarrow{c_j}'$ is given by

$$\overrightarrow{c_j}'(a) = \overrightarrow{c_j}(a) + \min(C_{j,i}(a, :) + \overrightarrow{c_i}).$$

As for the reduction RII, it is easy to show that the optimal objective value for the modified graph corresponds to the optimal value of the original problem. A formal proof is given by Eckstein [**?**].

RII Reduction.

Reduction RII follows the same idea as reduction RI. The operation is applied to nodes v_i with degree two. The two adjacent nodes are denoted by v_j and v_k respectively. Vertex v_i is removed and the minimal costs for v_i depending on choices for v_j and v_k are added to the cost matrix $C_{j,k}$. The new cost matrix $C'_{j,k}$ is defined as follows:

$$C'_{j,k}(a,b) = C_{j,k}(a,b) + \min(C_{j,i}(a,:) + C_{k,i}(b,:) + \overrightarrow{c_i}.$$

RN Reduction.

PBQP graphs that cannot be reduced using reductions RI and RII are called *irreducible*. The smallest example for an irreducible graph is the complete graph with four nodes K_4. One possibility to deal with irreducible graphs is recursive enumeration. However, recursive enumeration has exponential worst-case complexity which makes it an infeasible approach for most applications. Instead, a heuristic called RN is applied that reduces a node using a local minimum computation. The basic idea is to make a decision as if the node and the adjacent vertices are disconnected from the rest of the PBQP graph. While reductions RI and RII can be shown to maintain the optimality of the obtained solution, RN does not and leads to sub-optimal solutions in general. Let v_i denote a vertex of degree three or more, we heuristically select the index with minimal costs from the cost vector $\overrightarrow{c}'$ defined as follows:

$$\overrightarrow{c}'(a) = \overrightarrow{c_i}(a) + \sum_{(v_i,v_j)\in E} \min(C_{i,j}(a,:) + \overrightarrow{c_j}).$$

Let s_i denote such an index with minimal costs in $\overrightarrow{c}'$, it remains to update the cost vectors for adjacent nodes accordingly. For each adjacent node v_j, the new modified cost vector $\overrightarrow{c_j}'$ evaluates to $\overrightarrow{c_j}' = \overrightarrow{c_j} + C_{i,j}(s_i,:)$.

24.3.2.2 Phase II: Trivial Solution.

Upon completion of the reduction phase, there are only nodes with degree 0 left in the PBQP graph. This corresponds to a PBQP in matrix notation where all cost matrices are the zero matrix and the cost function is reduced to

$$\min f(X) = \sum_{1\le i\le n} \overrightarrow{x_i}^T \overrightarrow{c_i}.$$

Since there are no dependencies among the decision variables, the minimization problem is equivalent to the evaluation of the term

$$\sum_{1\le i\le n} \min_{x_i} \overrightarrow{x_i}^T \overrightarrow{c_i}.$$

Thus, we can determine the solution for each decision variable x_i by finding the smallest element in its associated cost vector c_i.

24.3.2.3 Phase III: Back-Propagation.

Warning: package "algorithmic" is no longer used. Please use algorithm2e instead.

```
\FORALL{$v_i \in V$}
\STATE $s_{v_i} = k : c_i(k) = \min(c_i)$
\ENDFOR
\WHILE{$S$ not empty}
\STATE pop node $v_i$ from S
\STATE $\vec{c} = \vec{c_i}$
\FORALL{$(v_j,v_i) \in E$}
\STATE $\vec{c} += {\matrixfont C}_{j,i}(s_{v_j},:)$
\ENDFOR
\STATE $s_{v_i} = k : c(k) = \min(c)$
\ENDWHILE
```

Algorithm 8: Back-Propagation

In the back-propagation phase, nodes are re-inserted in reversed elimination order. At each step, the solution for the newly inserted node can be easily determined as the decision vectors for adjacent nodes are already known; see [**?**] for a formal proof. The algorithm for a PBQP graph $G = (V, E, C, c)$ is shown in pseudo-code in Algorithm 8. We denote with S the stack of eliminated nodes from Phase I.

24.3.3 Branch & Bound

Branch & Bound (BAB) is a very generic and efficient enumeration scheme for combinatorial optimization problems that has been successfully applied to a large set of NP hard optimization problems, *e.g.*,, integer programming, knapsack problem, traveling salesman problem, or maximum satisfiability problem.

The main idea is to arrange the search space such that large parts can be explored only implicitly. At any point in the optimization process, we maintain a pool of yet unexplored subspaces together with the best solution found so far. Initially, there is only one subset representing the whole solution space and the best solution is set to ∞[1]. Subspaces are created dynamically and arranged in a so-called search tree. The key-idea is that large portions of these subspaces can be eliminated by comparing their lower bounds with the best solution found so far.

Any BAB algorithm consists of the following key ingredients:

[1] Without loss of generality, we assume minimization problems within this chapter.

1. *Bounding procedure:* for a given subspace of the solution space, compute a *tight* lower bound on the best solution value that is obtainable within this subspace.
2. *Selection strategy:* select the next live subproblem to be investigated in the search procedure. Popular techniques are breath first search (BFS), depth first search (DFS), and best first search. The latter always selects the subproblem with the lowest bound among the set of live subproblems. An experimental evaluation of the various strategies for QAP can be found in [**?**].
3. *Branching rule:* divide the considered subspace into two or more subspaces to be considered in subsequent iterations of the algorithm.

An advantage of BAB based algorithms that becomes more and more important with the rise of multi- and many-core systems is their ability to scale naturally with the number of cores.

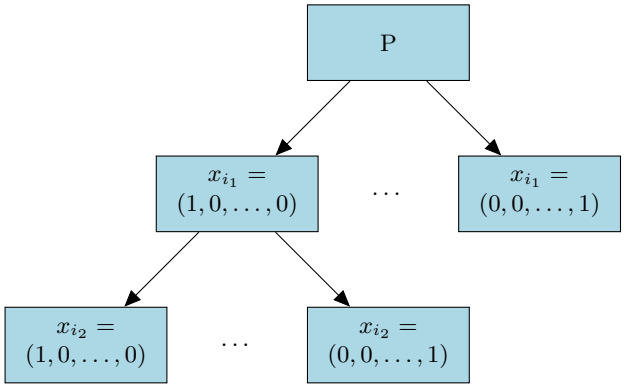


Fig. 24.5 A fragment of the search tree for a BAB based PBQP algorithm.

PBQP allows for a natural mapping to the BAB scheme outline so far [**?**]. As in Section 24.3.2, reductions RI and RII can be applied until a irreducible graph remains. A subspace of the solution space is represented by a node in a search tree as shown in Figure 24.5. A subspace is divided into smaller subspaces by selecting a yet unconsidered decision vector x_i from one of the leaf nodes and creating $|\mathbb{D}_i|$ child nodes, each of which representing one of the possible assignments for x_i. The whole set of partial assignments for a subspace of the solution space can be obtained by walking the search tree back to the root node.

A simple lower bound f_l on the objective function can be obtained using the following term:

$$f_l = \sum_{1 \le i \le n} \min c_i + \sum_{1 \le i < j \le n} \min C_{i,j}.$$

For inner nodes on the search tree, some of the variables x_i are already included in the partial assignment and evaluate to constants that replace the minimum computation and lead to tighter bounds in general. Note that we can derive the new lower

bound after a branching operation in an incremental way using the lower bound of the parent node.

Leaf nodes that are not yet solved are called *live* and usually kept in a priority queue according to their lower bound. Various selection strategies are possible. The *best first search* strategy leads to good results in practice for instruction selection problems.

A whole subspace can be pruned from the search if its lower bound exceeds a global upper bound on the value of the optimal solution. Initially, the global upper bound can be set to ∞. However, faster convergence can be achieved in general by applying a heuristic algorithm such as the method proposed in Section 24.3.2 in a pre-processing step. This will allow the BAB algorithm to prune large subspaces early in the optimization process.

24.4 Code Selection for Tree Patterns

The matching problem for SSA graphs reduces to PBQP [**?**] in a straightforward fashion. In the basic modeling, SSA and PBQP graphs coincide. This means that the number of decision vectors and cost matrices in the PBQP is determined by the number of nodes and edges in the SSA graph respectively. A solution for the PBQP instance induces a complete cost minimal cover of the SSA graph.

As in traditional tree pattern matching, an ambiguous graph grammar consisting of tree patterns with associated costs and semantic actions is used. Input grammars have to be *normalized*. This means that each rule is either a *base rule* or a *chain rule*. A base rule is a production of the form $\mathtt{nt}_0 \leftarrow op(\mathtt{nt}_1, \ldots, \mathtt{nt}_{k_p})$ where $\mathtt{nt}_i$ are non-terminals and *op* is a terminal symbol, *i.e.*,, an operation represented by a node in the SSA graph. A chain-rule is a production of the form $\mathtt{nt}_0 \leftarrow \mathtt{nt}_1$, where $\mathtt{nt}_0$ and $\mathtt{nt}_1$ are non-terminals. A production rule $\mathtt{nt} \leftarrow op_1(\alpha, op_2(\beta), \gamma))$ can be normalized by rewriting the rule into two production rules $\mathtt{nt} \leftarrow op_1(\alpha, \mathtt{nt}', \gamma)$ and $\mathtt{nt}' \leftarrow op_2(\beta)$ where $\mathtt{nt}'$ is a new non-terminal symbol and α, β and γ denote arbitrary pattern fragments. This transformation can be iteratively applied until all production rules are either chain rules or base rules.

To illustrate this transformation, consider the grammar in Figure 24.6, which is a normalized version of the tree grammar introduced in Figure 24.2. Temporary nonterminal symbols `t1`, `t2`, and `t3` are used to decompose larger tree patterns into simple base rules. Each of these base rules spans across a single node in the SSA graph.

The instruction selection problem for SSA graphs is modeled in PBQP as follows. For each node u in the SSA graph, a PBQP variable x_u is introduced. The domain of variable x_u is determined by the subset of base rules whose terminal symbol matches the operation of the SSA node, *e.g.*,, there are three rules (R_4, R_5, R_6) that can be used to cover the shift operation `SHL` in our example. The last rule is the result of automatic normalization of a more complex tree pattern.

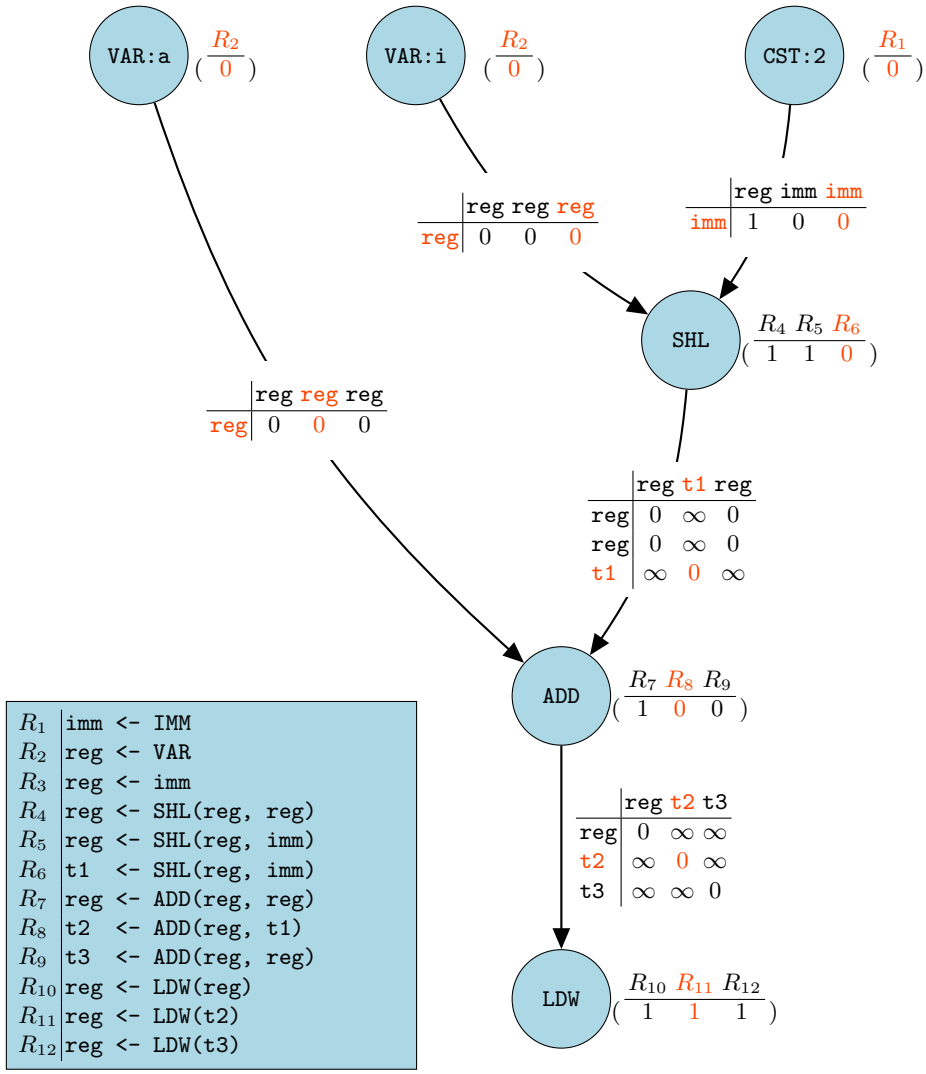


Fig. 24.6 PBQP instance derived from the example shown in Figure 24.2. The grammar has been normalized by introducing additional nonterminals.

The cost vector $\overrightarrow{c_u} = w_u \cdot \langle c(R_1), \ldots, c(R_{k_u}) \rangle$ of variable x_u encodes the local costs for a particular assignment where $c(R_i)$ denotes the associated cost of base rule R_i. Weight w_u is used as a parameter to optimize for various objectives including speed (e.g. w_u is the expected execution frequency of the operation at node u) and space (e.g. the w_u is set to one). In our example, both R_4 and R_5 have associated costs of one. Rule R_6 contributes no local costs as we account for the full costs of a complex tree pattern at the root node. All nodes have the same weight of one, thus the cost vector for the `SHL` node is $\langle 1, 1, 0 \rangle$.

An edge in the SSA graph represents data transfer between the result of an operation u, which is the source of the edge, and the operand v which is the tail of the edge. To ensure consistency among base rules and to account for the costs of chain rules, we impose costs dependent on the selection of variable x_u and variable x_v in the form of a cost matrix C_{uv}. An element in the matrix corresponds to the costs of selecting a specific base rule $r_u \in R_u$ of the result and a specific base rule $r_v \in R_v$ of the operand node. Assume that r_u is $\texttt{nt} \leftarrow op(\ldots)$ and r_v is $\cdots \leftarrow op(\alpha, \texttt{nt}', \beta)$ where `nt'` is the non-terminal of operand v whose value is obtained from the result of node u. There are three possible cases:

1. If the nonterminal `nt` and `nt'` are identical, the corresponding element in matrix C_{uv} is zero, since the result of u is compatible with the operand of node v.
2. If the nonterminals `nt` and $\texttt{nt}'$ differ and there exists a rule $r : \texttt{nt}' \leftarrow \texttt{nt}$ in the transitive closure of all chain-rules, the corresponding element in C_{uv} has the costs of the chain rule, *i.e.,*$w_v \cdot c(r)$.
3. Otherwise, the corresponding element in C_{uv} has infinite costs prohibiting the selection of incompatible base rules.

As an example, consider the edge from `CST:2` to node `SHL` in Figure 24.6. There is a single base rule R_1 with local costs 0 and result nonterminal `imm` for the constant. Base rules R_4, R_5, and R_6 are applicable for the shift, of which the first one expects nonterminal `reg` as its second argument, rules R_5 and R_6 both expect `imm`. Consequently, the corresponding cost matrix accounts for the costs of converting from `reg` to `imm` at index $(1, 1)$ and is zero otherwise.

Highlighted elements in Figure 24.6 show a cost-minimal solution of the PBQP with costs one. A solution of the PBQP directly induces a selection of base and chain rules for the SSA graph. A traversal over the basic blocks using the SSA graph is sufficient to execute the associated semantic rules in order to emit the code.

24.5 Extensions and Generalizations

24.5.1 Code Selection for DAG Patterns

In the previous section we have introduced an approach whose patterns resemble simple tree fragments, *e.g.,*, there is not support for machine instructions with mul-

tiple results. This restricts the modeling of advanced features commonly found in embedded architectures and SIMD extensions of nowadays CPUs.

Consider the introductory example shown in Figure 24.4. Most architectures have some form of auto-increment addressing modes. On such a machine, the load and the increment of both `p` and `q` can be done in a single instruction benefiting both code size and performance. However, post-increment loads cannot be modeled using a single tree-shaped pattern. Instead, it produces multiple results and spans across two non-adjacent nodes in the SSA graph, with the only restriction that their arguments have to be the same.

Similar examples can be found in most architectures, *e.g.*,, the `DIVU` instruction in the Motorola 68K architecture performs the division and the modulo operation for the same pair of inputs. Other examples are the `RMS` (read-modify-store) instructions on the IA32/AMD64 architecture, autoincrement- and decrement addressing modes of several embedded systems architectures, the `IRC` instruction of the HPPA architecture, or `fsincos` instructions of various math libraries. Compiler writers are forced to pre- or post-process these patterns heuristically often missing much of the optimization potential [**?**]. These architecture-specific tweaks also complicate re-targeting, especially in situations where patterns are automatically derived from generic architecture descriptions.

Supporting these patterns, however, is complicated by dependency constraints among complex patterns. For any graph cover, the existence of a topological order is necessary in order to generate correct code. Cycles would imply that operations are executed on the target hardware before the values of the operands are available. The partial order among nodes is defined by the edges in the SSA graph and eventual memory dependencies.

In principle, such issues can even arise for tree-shaped patterns. An example is given in Figure 24.7, which shows a typical read-modify-store (RMS) pattern ("`add r/m32, imm32`") for the IA32/AMD64 architecture. A corresponding production rule might be formulated as

```
stmt <- ST(x:reg, ADD(LD(x), imm)).
```

If we have to assume that `p` and `q` might address the same memory location, we have to account for the antidependency among statements `(1)` and `(2)` and the output dependency among statements `(2)` and `(4)`; depicted in Figure 24.7*(b)* by dotted lines. There is obviously no topological order among the highlighted part forming the RMS pattern and the store corresponding to instruction `(2)`. Thus, we cannot apply the pattern without introducing cyclic dependencies.

However, for tree-shaped patterns, we can always perform a *local* check that determines if a particular pattern is or is not applicable. For DAG patterns, this is not necessarily the case. Again, an example is given in Figure 24.8. The code fragment contains three feasible examples of a post-increment store pattern. Assuming that we know that p, q, and r point to mutually distinct memory locations, there are no further dependencies apart from the edges shown in the SSA graph. However, the situation is now different. The example gives rise to a topological order as long as we do not select *all* three instances of the post-increment store pattern concurrently.

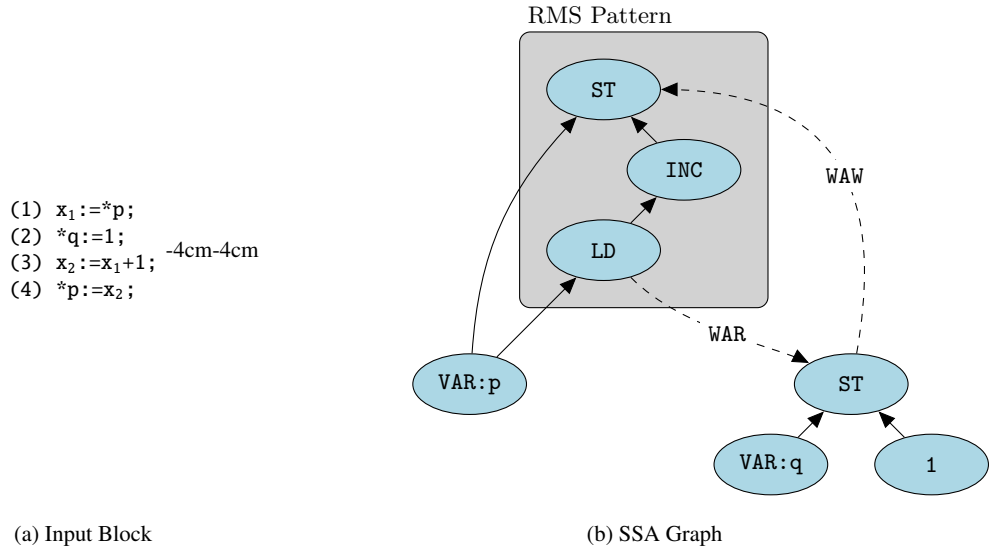


Fig. 24.7 Potential memory dependencies for a tree-shaped read-modify-store (RMS) pattern.

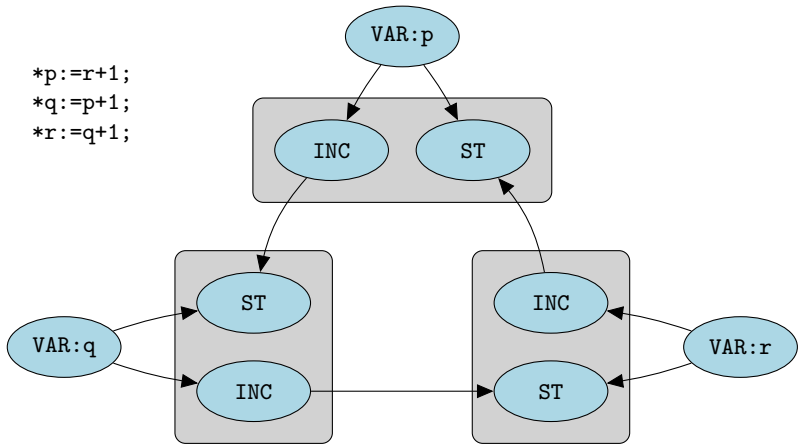


Fig. 24.8 DAG patterns may introduce cyclic data dependencies.

Thus, the existence of a topological order is now a global property, which has to be considered in the modeling of the problem.

A proper generalization of the PBQP based instruction selection [**?**] is outlined in Algorithm 9. Only steps (1), (2), and (5) differ from the approach discussed so far. First, we identify concrete tuples of nodes in the SSA graph that can be used to form complex patterns. Next, we transform the problem to an instance of PBQP that is processed using a generic solver library. The problem formulation ensures the existence of a topological order among the chosen productions and allows for

Warning: package "algorithmic" is no longer used. Please use algorithm2e instead.

```
[1]
  \STATE identify instances of complex patterns within basic blocks
  \STATE transform the problem to an instance of PBQP
  \STATE obtain a solution for the PBQP instance using a generic
  solver
  \FORALL{basic blocks $b$}
    \STATE compute a topological order for the subgraph
    induced by basic block $b$
    \STATE apply the semantic rules associated with the chosen
    productions in the order computed in step \texttt{(5)}.
  \ENDFOR
```

Algorithm 9: Generalized PBQP instruction selection

a straight-forward back-transformation that maps a solution vector of PBQP to a complete graph cover. The partial order among the particular nodes is defined by the edges in the SSA graph and additional data dependencies among load and store instructions. We can thus use a reversed post-order traversal to apply the semantic actions associated with the chosen productions in a proper order on the subgraphs induced by individual basic blocks.

24.5.1.1 Modeling

As for tree patterns, DAG patterns are decomposed into simple base rules for the purpose of modeling, *e.g.*,, the postincrement store pattern

```
P1: tmt <- ST(x:reg, reg), reg <- INC(x) : 3
```

is decomposed into the individual pattern fragments

```
P1.1: stmt <- ST(reg, reg)
P1.2: reg  <- INC(reg)
```

The matcher explicitly enumerates *instances* of complex patterns in step (1) of Algorithm 9, *i.e.*,, concrete tuples of nodes that match the terminal symbols specified in a particular production. In the example shown in Figure 24.8, there are three instances of the postincrement store pattern. Dependencies among these instances (denoted in the following by $\prec$) arise from real dependencies among individual nodes (edges in the SSA graph) and eventual memory dependencies as outlined in Figure 24.7.

For instances of DAG patterns, additional decision variables are required that encode if the particular pattern is to be selected. Thus, we have two sets of decision variables $X = X_1 \cup X_2$ for regular SSA nodes and instances of complex patterns respectively.

The domain for variables in X_1 is defined by the set of applicable base rules arising from two different sources:

1. Simple productions consisting of a single base rule; these are handled just like before for tree grammars.
2. Base rules arising from the decomposition of DAG patterns. All identical base rules obtained from the decomposition of complex productions contribute only to a single element in the decision vector.

While the former group represents the set of simple patterns that can be used to obtain a cover for a particular node, the second class of base rules can be seen as a proxy for the whole set of instances of (possibly different) complex productions including the node. The costs for these proxy states are 0, otherwise they reflect the real costs of the corresponding tree rule.

Variables in set X_2 are created for each enumerated instance of a complex production. They encode whether a particular instance is chosen or not, *i.e.*,, the domain basically consists of the elements *on* and *off*. The local costs reflect the costs for the particular pattern except for the *off* state, which is set to 0.

The PBQP for the SSA graph introduced in Figure 24.8 is shown in Figure 24.9. In addition to the postincrement store pattern with costs 3, we assume regular tree patterns for the store and the increment nodes with costs two denoted by `P2` and `P3` respectively. Rules for the `VAR` nodes are omitted for simplicity.

Nodes one to six correspond to the nodes in the SSA graph. Their domain is defined by the simple base rule with costs two and the proxy state obtained from the decomposition of the postincrement store pattern. Nodes 7, 8, and 9 correspond to the three instances identified for the postincrement store pattern. As noted before, we have to to guarantee the existence of a topological order among the chosen nodes. One approach is to exploit the property that every acyclic directed subgraph gives rise to a not necessarily unique topological order. We can refine the state *on* such that it reflects a particular index in a concrete topological order. Matrices among these nodes account for data dependencies, *e.g.*,, consider the matrix established among nodes 7 and 8. Assuming instance 7 is *on* at index 2, The only remaining choices for instance 8 are not to use the pattern or to enable it at index three, as node 7 has to precede node 8.

A different class of constraint matrices is required to ensure that the corresponding proxy state is selected on all the variables forming a particular pattern instance. Therefore, we create matrix costs $C_{ul}^{X_1 \to X_2}$ such that the costs are zero if x_l is set to *off* or x_u is set to a base rule that is not associated to the instance. Otherwise, costs are set to ∞. Thus, when one of the instances correlated to a particular node u in the SSA graph is selected, the only remaining element in the domain of u with costs less than ∞ is the associated proxy state corresponding to the particular base rule fragment.

So far, the formulation allows the trivial solution where all of the related variables encoding the selection of a complex pattern are set to *off* (accounting for 0 costs) even though the artificial proxy state has been selected. We can overcome this problem by adding a large integer value M to the costs for all proxy states. In exchange, we subtract these costs from the cost vector of instances. Thus, the penalties for the proxy states are effectively eliminated unless an invalid solution is selected.

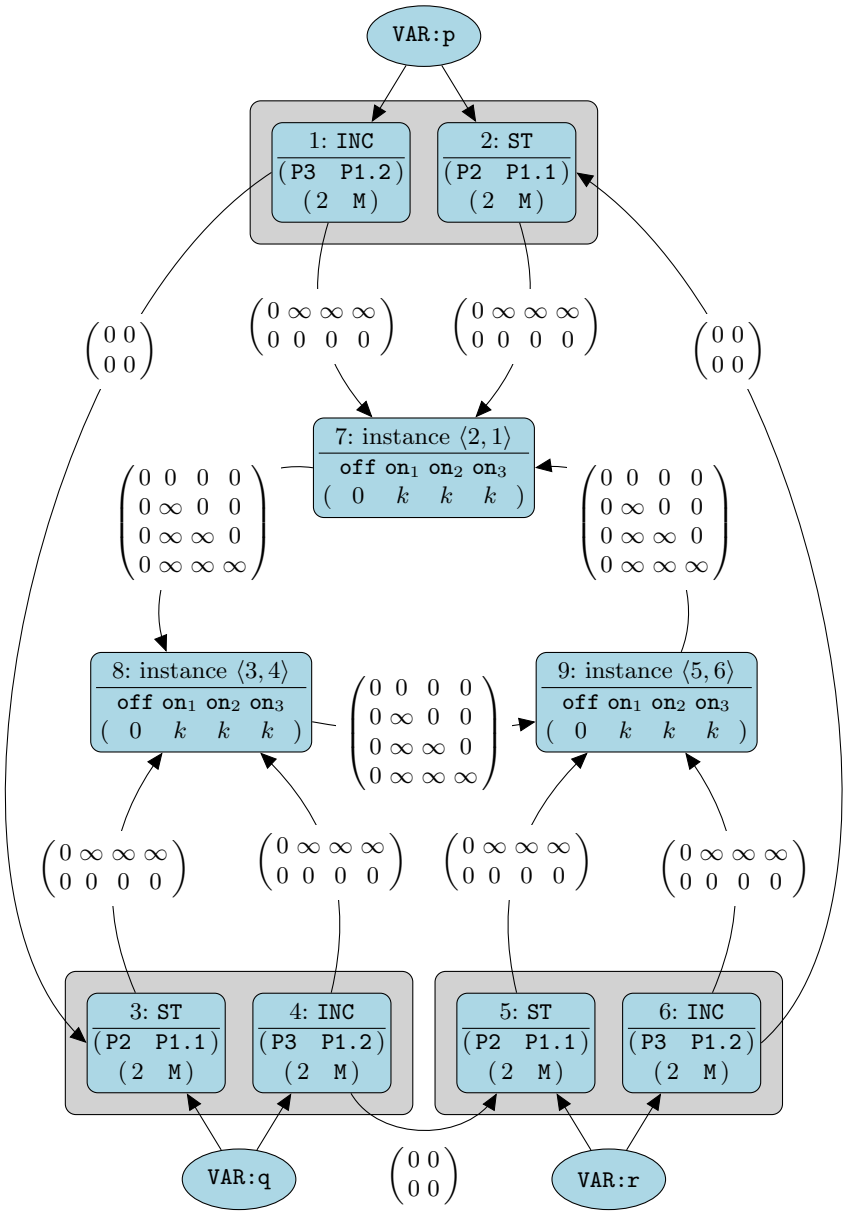


Fig. 24.9 PBQP Graph for the Example shown in Figure 24.8. We use k as a shorthand for the term $3 - 2M$.

Cost matrices among nodes one to six do not differ from the basic approach discussed before and reflect the costs of converting the nonterminal symbols involved.

It should be noted that for general grammars and irreducible graphs, the heuristic algorithm proposed in Section 24.3.2 cannot be guaranteed to deliver a solution that satisfies all constraints modeled in terms of ∞ costs. This would be a NP-complete problem. One way to work around this limitation is to include a small set of rules that cover each node individually and that can be used as a fallback rule in situations where no feasible solution has been obtained. This corresponds to macro substitution techniques and ensures a correct but possibly suboptimal matching. In practice, this is no severe limitation as grammars are usually written by defining a simple but complete set of rules covering each node individually and adding more complex rules later on. These limitations do not apply to exact techniques such as the branch-&-bound algorithm introduced in Section 24.3.3. It is also straight-forward to extend the heuristic algorithm with a backtracking scheme on RN reductions, which would clearly also be exponential in the worst case.

24.5.2 Chain Rule Placement

Chain rules can either be emitted at the basic block corresponding to the source node or right before each use, *i.e.,*, at the destination node. In [**?**], a more sophisticated technique is introduced that allows a more efficient placement of chain rules across basic block boundaries. This technique is orthogonal to the generalization to complex patterns. An optimal placement is computed by the construction of a min-cut problem for the given control flow graph. A solution for this problem can be found in polynomial time. There is a trade-off among performance and code size that is captured accurately in the proposed network flow model.

Consider the example in Figure 24.10(a) which shows a fraction of the control flow graph. The nodes of the graph represent basic blocks in the control flow graph and edges represent flow of control between basic blocks. Execution frequencies are given for each basic block which are obtained by profiling. Assume that the example is in pruned SSA form, and in node d is a definition of the form $\mathtt{x}_0$ `= ...` and in node u is a use of the definition `...`$\mathtt{x}_0$`...`. For a given graph grammar, let's assume we need to place a chain rule that has to be executed at least once on all paths from d to u. To find a cost optimal placement, we reduce the problem to an $s-t$ min cut as shown in the Figure 24.10(b). Each node v in the control flow graph corresponds to two vertices v_e and v_x in the network of the $s-t$ min cut. An edge between v_e and v_x represents the cost of placing the chain rule in node v. For each edge (v, w) in the control flow graph, there exists an edge (v_x, w_e) in the network that has infinite cost weights. Nodes s and t have connections to the definition node d_e and the use node u_x respectively. A cut of such a network can only consists of edges (v_e, v_x) because all other edges have infinite costs. All cut edges represent nodes where we can place chain rules rules optimally and ensure that on all paths from d to u the chain rule is executed. An $s-t$ min cut of the newly constructed network, gives us an optimal

placement of chain rules. For the example the optimal cut is node b since (b_e, b_x) separates node s from node t in the network with only a single cost unit.

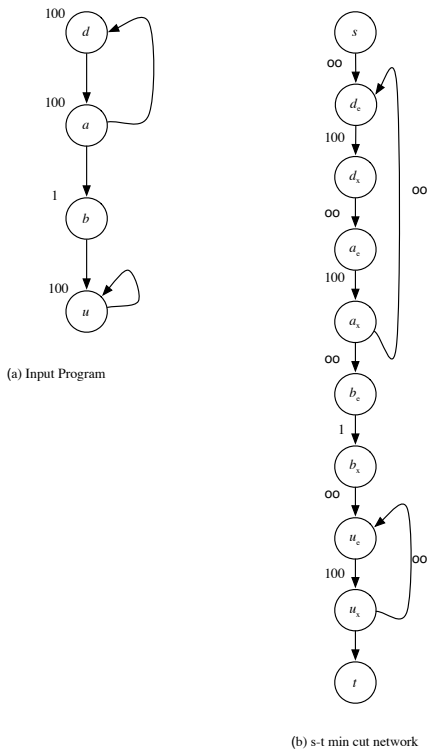


Fig. 24.10 Example for Chain Rule Placement

CHAPTER 25

Automatic Parallelization and Sequencialization *S. Pop*

Progress: 0%

Material gathering in progress

Author: S.Pop

One of the applications of data flow analysis is the translation of sequential code into parallel code. This translation is usually performed by compilers to adapt the program to the target machine hardware: vector instructions, multi or single processors. This chapter presents some translation algorithms that start from the SSA representation and target parallel, vector and sequential code.

CHAPTER 26

If-Conversion

C. Bruel

Progress: 45%

Material gathering in progress

26.1 Overview/Motivations

We describe in this chapter an if-conversion algorithm in SSA. If-conversion is the process to optimize a control flow region into an equivalent region where branches have been removed. We start with a description of a full SSA model, taking as input a SSA region producing a SSA region using speculation. We then describe how this framework is extended to use predicated execution, using the ψ-SSA form presented in chapter ???.

26.1.1 Introduction

A way to increase the instruction throughput is to increase the IPC (number of instructions per cycle), either by increasing pipeline depth, or by increasing the number of instructions that are executed simultaneously. This can be achieved by multiple issue processors. Either out-of-order superscalar execution models, using dynamic scheduling models and expensive hardware complexity or else in-order execution model in embedded processors because of lower hardware complexity, implying lower production and usage costs.

Instruction Level Parallelism (ILP) was demonstrated to be very effective on embedded processors, for example in the mathematical for evaluation of polynomial

expressions [**?**], or to sustain performance in loop intensive computation required for multimedia applications [**?**].

In order to exploit this ILP, those applications delegate to the compiler the responsibility to organize the parallel execution of the execution flow [**?**].

To permit the compiler to organize the parallelism explicitly, VLIW (Very Large Instruction World) or EPIC (Explicitly Parallel Architectures) make the parallelism visible into the ISA representation, relaying on the static scheduler to organize the compiler output so that multiple instructions can be issued for each cycle.

Unfortunately, programs contain many sequences of instructions that cannot be parallelized, because of control or data dependencies. Different Instruction Level Parallelism techniques are implemented in the hardware, such as dynamic scheduling, branch prediction, or in the compiler, such as software pipeline, vectorization or loop optimizations. But still, despite those efforts, the execution flow is still disrupted by conditional branches, and many ALU cycles are wasted.

Conditional branches introduce control dependencies between instructions. An instruction is control dependent on a preceding instruction if the first one determines the execution of the second one [**?**]. They limit the scope of ILP optimizations, because they reduce the number of instructions that are not data dependent and thus can be executed in parallel. By reducing the average size of a basic block, they act as a bottleneck for exposing parallelism.

If-conversion is the process of transforming a control flow region containing several basic blocks into a single basic block of conditional instructions. Thus removing control branches from this region [**?**]. The conditional instructions are implemented on the target machine using the predication or speculation techniques described bellow.

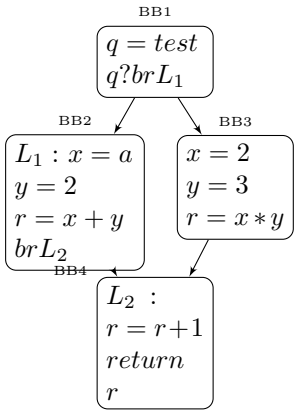
Removing branches improves performance in several ways: by removing the misprediction penalty, the instruction fetch throughput is increased and the instruction cache miss penalty reduced. Enlarging the size of the basic blocks allows the static scheduler to schedule earlier long latency operations and to improve the ILP by merging multiple control paths into a single flow of execution. Many compiler optimizations are impacted by branches. Software Pipelining for example, cannot efficiently schedule loops with conditional branches [**?**].

Consider the simple example 26.1, that represents the execution of an if-then-else statement on a 4-issue processor. Branches are highly biased (a branch is highly biased when it is known to be frequently taken or not), so the basic blocks can be reordered to favor the critical path. Even with this optimistic case, the schedule height is still 5 cycles (assuming that all instructions have a one cycle latency), with a maximal schedule height of 6 instructions.

After if-conversion the execution path is reduced to 4 cycles, regardless of the tests outcomes.

From this introductory example, we can observe that the transformation implies:

- The merge of execution paths into a single execution path, implying a better exploitation of available resources.
- The reduction of the schedule length, because instructions can be speculated before the branch.



(a) Control Flow

	ALU1	ALU2	ALU3	ALU 4
	$q = test$	-	-	-
	q ? br L_1	-	-	-
	$x = 2$	$y = 3$	-	-
	$r = x * y$	-	-	-
L_2	$r = r + 1$	$return$ r	-	-
L_1	$x = a$	$y = 2$	-	-
	$r = x + b$	-	-	-
	br L_2	-	-	-

(b) with basic block ordering

ALU1	ALU2	ALU3	ALU 4
$q = test$	-	-	-
$x_1 = 2$	$y_1 = 3$	$x_2 = a$	$y_2 = 2$
$r_1 = x_1 * y_1$	$r_2 = x_2 + y_2$	-	-
$r = q$? $r_1 : r_2$	$return$ r	-	-

(c) After if-conversion

Fig. 26.1 Static schedule of a simple region on a 4 issue processor

- That the ariables need to be renamed,
- The introduction of a *merge* pseudo operation.
- The production of new data dependency. If-conversion transforms control dependencies into data dependencies [**?**].

Considering that the merging point can be materialized into the original control flow by the ϕ pseudo operation and that the renaming is a basic property of SSA, the transformation from SSA to If-converted code seems straightforward.

26.1.2 Architectural requirements

The example illustrates also that the *merge* pseudo operations needs to be mapped to a conditional form of execution on the target's architecture. Depending on the available support, the algorithm must be able to match the ISA, to provide support for conditional execution of instructions, or to conditionally commit the result of an

instruction. Basically, several kinds of support for predication can (co-)exist: predicated execution or speculative execution by mean of conditional moves. A partial predication model is used to predicate only the subset of the ISA that are not easily speculable, usually memory operations.

Predication is an architectural feature that allows an instruction to be executed conditionally by means of a predicate.

fully predicated	select	cmov
$p\ ?\ x = a + b$	$t = a + b$	$t = a + b$
$\overline{p}\ ?\ x = 0$	$x = p\ ?\ t : 0$	$x = 0$
		$x = cmoveq\ t$

Fig. 26.2 Conditional execution using different models

Speculation is a compiler technique that allows instructions that are control dependent to be executed unconditionally. The result can be committed only when the condition is known to be true. Examples of conditional moves are the *cmov* (Pentium Pro) or the *select* (Multiflow, ST231 VLIW) instructions.

Example 26.2 shows a conditional execution using the different modes.

To be speculated, an instruction should not potentially create any other side effects, or hazards. For instance a memory load should not trap (cache misses can also be considered for performance). This of course prevents unconditional execution by means of speculation of memory access for which the address is valid only on the selected path. Because of those reasons memory operations are a major impediment to ILP.

Since load are long latency operations, speculative load support is effective to fetch datas earlier in the instruction stream, reducing stalls. Modern architectures provide architectural supports to dismiss invalid address exceptions. Examples are the *ldw.d* dismissible load operation of the $ST231$ processor or the $IA64speculativeloads$. Note that $IA64$ provides both speculative and predicate loads, so the former will be preferred by if-conversion.

IA64 speculative load	Multiflow dismissible load
$t = ldw.s(addr)$	$t = ldw.d(addr)$
$jump\ l_1 :$	$x = select\ ?\ t : x$
$check.s\ t$	
$x = t$	
base store hoisting	index store hoisting
$x = select\ p\ ?\ addr : dummy$	$index = select\ p\ ?\ i : j$
$stw(x, value)$	$stw(x[index], value)$

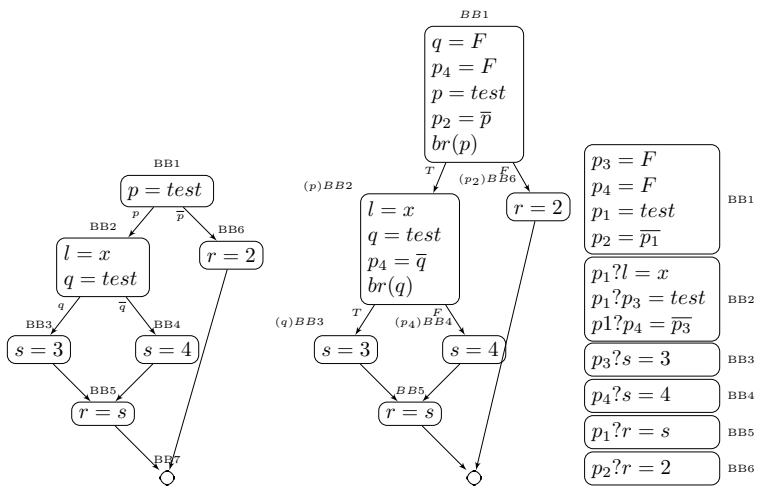
Fig. 26.3 Example of speculated memory operations

Stores can be performed by speculating the address value, and forcing it to be a dummy valid address in the case of not taken path. Naturally store hoisting can only

be done if there is no aliasing conflict between different memory operations merged into a single execution path.

26.2 Classical If-Conversion process

Established if-conversion processes addresses the problem in the following order. First allocates boolean variables to the basic blocks, on an identified sub-region, then perform predicate initialization placement and then restructure the CFG sub-region into the final if-converted region. As an starting example we describe briefly the RK algorithm proposed by Park and Schlansker [**?**] on the nested tests from figure In 26.4a. The RK algorithm starts to compute the Control Dependence Graph [**?**]. It is necessary to map the basic blocks to guard conditions, and the guards and the set of basic blocks that need to set them. The algorithm creates a new guard, eventually initialized to false, for each condition. **??** shows the sub-region with new conditions initialized, and 26.5 the RK mappings.



(a) Nested test **(b)** Predicate Assignments **(c)** After Instruction layout

Fig. 26.4 Classical support for partial predication

The process finishes by emitting the conditioned instructions, and removes the branches. **??**.

There are several problems with this approach. First, it strongly assumes that all instructions can be predicated, in particular predicate setting operations (test) need to be conditioned. Without predicated support for comparison instruction, a merge

Basic Block	BB1	BB2	BB3	BB4	BB5	BB6
Guard	T	p	q	p4	p	p2

(a) R mapping

Guard	p	q	p4	p2
Basic Block	1	2,-0	-2,-0	-1

(b) K mapping

Fig. 26.5 Output from RK algorithm

must be done between the condition setting. For instance in the given example the initialization sequence $p_1?p_3 = test$ needs to be rewritten as $t = test; p_1 = p_3\&t$. Speculative based approach, necessary to handle efficiently partially predicated architectures are then penalized by this approach, since a rewritten pass is needed to convert a conditional instruction into a temporary register and a conditional move. Another pitfall, the algorithm works a a pre-selected control flow subregion. One of the challenge of all if-conversion frameworks is to be able to predict that the conversion will be beneficial before effectively doing it. If the the number of computed guards exceed the number of predicate registers, then a new region has to be re-considered and the process restarted. Even with full predication support, speculative execution can be beneficial. Traditional algorithm tend to predicate all the instructions defined by a basic block, without considering their use. For instance in $BB2$ the instruction $l = x$ doesn't have any use other than the done defined by its predicate. Finally, those framework required intensive computing resources to compute the Control Dependence Graph, Register Renaming, a Def-Use chains or post-phase Predicate Reduction processing.

We propose in the remaining of the chapter an application of SSA to efficiently deal with those problems

26.3 SSA If-Conversion

SSA provides an efficient intermediate representation in which false data dependencies have been removed, removing the need for register renaming when merging two definitions into one. Each assignment target (def) is assigned a unique register name. Registers are re-materialized at join nodes using ϕ operations. A procedure is in minimal SSA form if the number of ϕ instructions at join nodes is as small as possible. In relation to if-conversion, The SSA representation ensures that a variable is assigned only once, and their merging point is exposed as ϕ operations. SSA alleviates the need for renaming and only the definitions used in a ϕ need to be predicated.

Consider the equivalent SSA representation of the example 26.4a.

Note that basically, since a ϕ expresses the merging point of n definitions from n predecessors. Thanks to SSA the definition point is immediately available, since it corresponds to the path by which the ϕ operand is reached. By construct, the definition of a ϕ's use is control dependent from the first condition that dominates it.

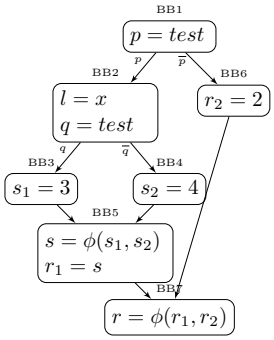


Fig. 26.6 Nested test in SSA

For example, $s\phi$'s use in $BB3$ is control dependent of $BB2$. Locally, SSA information replace the need for a control dependence graph, which is an engineering advantage. Since ϕ functions are optimally placed thanks to the iterated dominance frontier criterion, we know that the predicate definition will be optimal, so a minimal set of condition is needed.

A very big difference in nature is in the way the merging of conditional is realized using data dependencies instead of predicate selection. In SSA the condition merge assignment is not realized thru merging of predicates, but using two successive conditional moves operations.

This algorithm takes as input a structured region in pure SSA form and produces a pure SSA using conditional move instructions to realize join points. One benefit of this approach is that all SSA properties are maintained, and scalar optimizations, such as constant propagation, can be executed without predicate awareness. Without SSA speculative if-conversion, implementation of a predicated intermediate representation is very complex, because optimization and def-use chains must deal with partial definitions.

26.3.1 SSA operations on basic blocks

The basic control flow structure for the transformation is based on hammocks graphs. A hammock is a single-entry single-exit control flow region H with a distinguished entry node in H and a distinguished exit node not in H. [**?**] and was demonstrated to be efficient for parallelization and global code optimization [**?**]. Is is also very powerful for predication because the describe a well defined scope and the number of conditional variables are minimized.

We restrict for now the transformation to the speculative and conditional move model. We use the *select* like notation $r = c? : r_1 : r_2$, to represent the speculative execution of instruction defining r_1 and r_2. r takes the value of r_1 if c is True, else r_2. We will study how to generate code for predicated instruction in the next section.

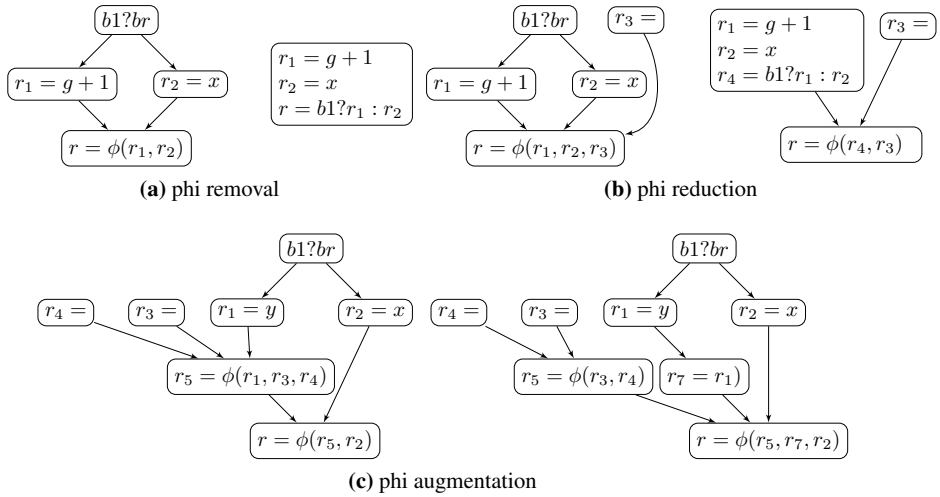


Fig. 26.7 SSA transformations

Consider a conditional branch depending on a predicate p and a region starting at *BBhead*. Let BBp be the set of single exit basic blocks $(BBi, \ldots, BB_n)$ that are on the taken path if p is true and BBq be the set of single exit basic blocks $(BBj, \ldots, BB_m)$ that are executed if p is false. The merge point of the if-converted region is at $BBjoin$. We distinguish 4 types of basic SSA transformations:

26.3.1.1 ϕ *removal*

(figure 26.7a) The join node of the considered region has two predecessors and ϕ instructions are of the form $r = \phi(r_1, r_2)$. After speculation of the definitions of r_1 and r_2 the instruction can be rewritten as $r = p?r_1 : r_2$. Once BBp and BBq have been promoted into *BBhead*, $BBjoin$ has for only predecessor *BBhead* so it is not in its dominance frontier. The ϕ can then be discarded.

26.3.1.2 ϕ *reduction*

(figure 26.7b) The join node of the considered region has n predecessors that have ϕ instructions of the form $r = \phi(r_1, r_i, r_j, \ldots, r_n)$. After merging, the ϕs are rewritten $t = p?r_i, ?r_j$ and a new ϕ $r = \phi(r_1, t, \ldots, r_{n-1})$ is created with one operand less. The merging instruction is inserted after the speculated r_i and r_j definitions into *BBhead*. $BBjoin$ is still in the dominance frontier of *BBhead*, A ϕ must be redefined with the new definitions. The two corresponding ϕ edges from BBq and BBp can be replaced

by the *BBhead* edge. The join block E has n predecessors with $n > 2$, and n blocks in its dominance frontier if it contains ϕs.

26.3.1.3 ϕ augmentation

(figure 26.7c) The objective is to remove incoming edges into the region, so the code is restructured during the if-conversion process to form a hammock. Consider a join node with n predecessors among which p is duplicated into q. The join node of the considered region have ϕ instructions of the form $r = \phi(r_1, p, r_i, r_j, \ldots, r_{n-1})$. The instruction is rewritten $t = select(p, r_i, r_j)$ and $r = \phi(r_1, t, q, \ldots, r_n)$. The duplicated blocks are new dominators of $BBjoin$ that define new $defs$. These blocks are now in the dominance frontier of $BBjoin$. SSA is maintained with new ϕ upgraded with the new reaching points.

Since data flow dependencies are broken, new opportunities for propagation or scalar optimization arise. Here the local t appears temporary, since $r2$ can be propagated into the ϕ.

A difficulty is that in a Hammock region, eg no side entries, the control dependency ϕ operands's definition BBp is easily deduced from its first dominator. However the BBp is not necessary a direct successor of the *BBhead*, therefore, knowing that we are control dependent is not enough. We also need to know if the dependencies is on the false condition or the true condition, This is easily solved by walking back the chain of basic blocks until the edge coming from *BBhead* is found.

26.3.2 SSA representation of conditional instructions

One problem of the described SSA-speculative framework is that it doesn't fit very well with a predicated model of conditional execution. Since ϕ are transformed to realize joint points into conditional moves, we should backtrack to relink the conditional moves into predicated instructions. This process of transforming speculation into predication is not convenient for many reasons: - Predication introduces a renaming problem: Since a same register name can have 2 different assignments (under different predicates), this breaks SSA rules. The solution is to re-use the framework to realize join points into an extended SSA representation:

In [**?**] ψ-SSA was described as a way to express a predicated form of SSA extension. Such representation is usually built from already predicated code, non SSA, originating from inlined assembly, peepholes, intrinsics functions or local transformations of control flow idioms.

ψ-SSA expose the edge dependency from the basic block into which the definition of the ϕ argument is defined in the original CFG by a new data dependency. This dependency needs to be materialized into a *select* or ψ operation. Note that the *select* operation is a real instruction that don't need to be replaced by the out-of-ssa process. If the target architecture doesn't provide such instruction to switch

between speculated instruction, it can be emulated using two conditional moves. One advantage to generate *select* instruction at this stage is that the program stays in full SSA form and make all the data dependencies explicit, and can be feed to all SSA optimizers.

SSA:	SSA-speculative form:	ψ-SSA form:
$if(p)$	$x_1 = a + b$	$x_1 = a + b$
$x_1 = a + b$	$x_2 = 0$	$x_2 = 0$
$else$	$x = p\ ?\ x_1 : x_2$	$x = \psi(p\ ?\ x_1, \overline{p}x_2)$
$x_2 = 0$		
$x = \phi(x_1, x_2)$		

Note that unlike ϕ arguments that are executed simultaneously (they don't depend each other), ψ arguments are executed sequentially and ordered from their definition predicate set. This property is necessary because of the new data dependencies introduced into the the straight line of predicated code stream.

The basic idea behind the SSA transformations is to replace ϕ operations by predicated instructions merging into a ψ or speculated instructions merging into a *select* equivalent instruction, while maintaining the SSA properties.

26.3.3 SSA promotion

To know the values that need to be conditionally defined, we only need to look at the defining instructions of the ϕ instructions. By elimination, all temporaries that do not have a join point into the considered region and that don't have a side effect are unconditionally speculated during the SSA transformations processes. Instructions with a side effect will need to be guarded. This is a considerable advantage over traditional if-conversion algorithms that marks all instructions in a conditional basic block as dependent on the predicate. By removing this predicate dependency, which is a data dependency, the instruction can be moved before the predicate assignment.

Our algorithm is applied iteratively on a control flow in SSA form until no more reductions are possible. The quality of the SSA taken as input does not affect the correctness of the algorithm: if the control flow is in pruned SSA, i.e two paths $x- > +z$ and $y- > +z$ converge at node z, then a ϕ node is inserted at z only if z is alive in or after z. if which case x and y are promoted and no *select* operation is generated. If the SSA is minimal, a *select* instruction would be generated and removed by dead code. Inserting dead code from minimal SSA only introduces noise in the local scheduling heuristics because of the false data dependencies.

26.3.3.1 Partial redefinition

A ψ operation exposes new data dependencies, by expressing the merge of two definitions. Note that the order of the partial definition is important, so a definition partially redefines the preceding ones. We use this property to speculate the first definitions, so it becomes speculated instead of disjoint. This local optimization allows the removal of apredicate dependency but also creates a new partial dependency (predicates are not disjoint).

disjoint predicates	optimized order predicates
$p = test$	$x_1 = a + b$
$p \ ? \ x_1 = a + b$	$p = test$
$\overline{p} \ ? \ x_2 = c$	$\overline{p} \ ? \ x_2 = c$
(1) $x = \psi(p \ ? \ x_1, \overline{p} \ ? \ x_2)$	$x = \psi(T \ ? \ x_1, \overline{p} \ ? \ x_2)$

T represents the $True$ predicate. This optimization is useful to save one predicate register and to remove a data dependency between a predicate definition in its use. The ψ definition is defined on the T predicate set, therefore it is speculable, as shown here:

26.3.3.2 ψ speculation properties

Since ψ operations are part of the intermediate representation, they can be considered for inclusion in a candidate region for if-conversion. Conditional operations defined are thus in turn speculated or conditioned. We define here promotion rules for ψ operands, whereas the instructions defining the ψ operands will be speculated, and the condition realized by the use of the ψ definition.

Consider the instructions **??** containing a subregion already processed. The ψ operation can be safely speculated if all the instructions defining its operands can be speculated. So 1) they don't produce hazardous execution and they don't produce any side effects, 3) there exist a conditional move instruction to merge the results. Then the block can be executed regardless of the value of c. The use of the ψ result is also unconditionally executed, and the condition realized in the ϕ transformation.

26.3.3.3 ψ predication properties

If the instructions are not speculable, they must be predicated: In **??**, the c condition merges with all conditions under which the ψ operands are defined. Here a new predicate p_1 is created to hold the predicate definition for the instructions defined under c.

Note that conceptually, the speculative ψ execution allows a predicate definition domain larger that the original one, which the predicative transformation exactly

matches the initial definition domain, at the expense of more data dependencies and predicate computation.

We can see with this example that the decision to speculate or predicate can be done at the level of each joining definition, allowing a mix of them in the final program. The advantage to speculation over predication is a reduced dependency length. The disadvantage of speculation is that it increases register pressure until the merge point, and puts long latency operations on the critical path.

nested if	speculated nested if	predicated nested if
$if(c)$	$x_1 = a + b$	$p_1 = \overline{p} \& c$
{	$\overline{p}\ ?\ x_2 = c$	$c\ ?\ x_1 = a + b$
$x_1 = a + b$	$x = \psi(T\ ?\ x_1, \overline{p}\ ?\ x_2)$	$p_1\ ?\ x_2 = c$
$\overline{p}\ ?\ x_2 = c$	$d_1 = use(x)$	$x = \psi(c\ ?\ x_1, p_1\ ?\ x_2)$
$x = \psi(T\ ?\ x_1, \overline{p}\ ?\ x_2)$	$\overline{c}\ ?\ d_2 = 3$	$d_1 = use(x)$
$d_1 = use(x)$	$d = \psi(T\ ?\ d_1, \overline{c}\ ?\ d_2)$	$\overline{c}\ ?\ d_2 = 3$
}		$d = \psi(T\ ?\ d_1, \overline{c}\ ?\ d_2)$
$else$		
$d_2 = 3$		
$d = \phi(d_1, d_2)$		

Fig. 26.8 Inner region ψ

26.3.3.4 Predicate merging

During the region formation, subregions containing a block that is reached from two conditions can be optimized by merging predicates. A new predicate is computed using a logical operation on both basic blocks' predicates after a normalization pass. Simple logical operations are usually caught as a peephole or during the instruction selection mechanism, but making it part of the if-conversion process allows it to handle more complex regions because our predicate merging algorithm is not limited to basic blocks that only define predicates, making instructions depending on predicates merge part of the generic SSA speculation framework. However, this transformation is very sensitive to biased branches since each conditional operation now depends on two predicates instead of one that cannot be scheduled together because of the computation of the logical operation making it more difficult to compensate for the branch removal. In order to avoid long sequences data dependencies and break schedule, we exclude from this promotion predicates that depend on long latency operations. Because of new data dependencies introduced by the new computed predicate the performance contribution of this transformation mainly comes from the branch removal (two conditional branches and two direct branches are removed from a if-converted region whose predicates have been merged) rather than local ILP. Predicate promotion and merge is more effective in loop nest regions where more optimizations may extract ILP from it, for example modulo scheduling can extract ILP from such if-converted bodies by overlapping different iterations.

We allow two successive conditional blocks sharing one immediate post-dominator to be merged with logical operations after a normalization transformation. The normalization transformation ensures that conditional blocks sharing a common target can be merged by defining a wired *or*, or a wired *and* share the same branch characteristics using branch reordering, test inversion or *de − morgan* transformations.

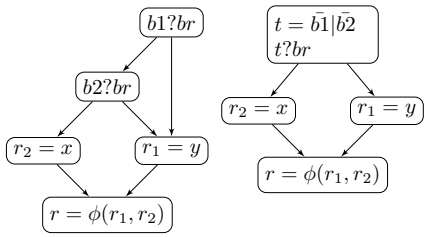


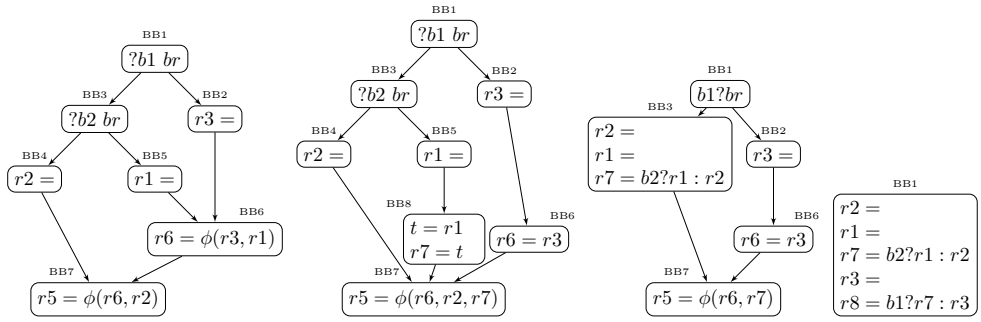
Fig. 26.9 Predicate merging

26.3.3.5 Block duplication

Block duplication is used to remove side edges and to remove the constraints on control dependencies in order to enable the algorithm to find a set of basic blocks to if-convert. Unless applied carefully, block duplication could be the cause of code bloating without performance improvement. However experience has shown that when applied carefully it can be the source of very efficient if-conversion. Consider for example in the figure 26.10. Since we are if-converting from the inner most regions, the algorithm first considers the region BB3,BB4,BB5,BB6,BB7, and discards the edge coming from BB2 by duplicating BB6 into BB8. The ϕ becomes a move in the duplicated block with a renamed definition. The new ϕ operands are updated from the new edge. Note that in the implementation the block does not need to be created since it will be next promoted into BB3. We have two nested hammocks and the process can iterate. The dependency that we removed in the control flow is now expressed as a data dependency between the two *select* instructions.

The algorithm to perform SSA block duplication is decomposed into three steps:

- Extract the ϕs 's def to be conditioned from the duplicated block creating a *move* instruction and a new reduced ϕ (or two *move* instructions if the duplicated block had only two incoming edges).
- Then the ϕs in the tail basic block are augmented with the new def created by the new repair instruction. If the ϕ was live-out after the tail block a move must be inserted to avoid propagating renaming outside of the region considered.
- The last step consists of renaming the new definitions to keep the region in SSA form.



(a) Original **(b)** After duplication **(c)** sub-region 1 **(d)** sub-region 2

Fig. 26.10 Side entry removal using block duplication

26.4 Hyperblock formation in SSA

Unfortunately, critical regions are rarely composed of simple hammocks or one level nested ifs. At the same time, processors have limited resources. the number of registers will determine the acceptable level of data dependencies to minimize register pressure, the number of predicate registers will determine the depth of the if-conversion so that the number of conditions doesn't exceed the number of available predicates.

The region must be just large enough to swell the scope of scheduling, without over-commiting the machine resources. On the contrary, although if-conversion removes branches, It also adds some overhead (new instructions to merge predicates, register pressure for speculative conditional, heigher dependence height because of the presence of new data dependencies on the predicates). The newly formed basic block should not have a higher schedule estimation than the one that would have been taken if the code was not if-converted.

For this reason, existing approaches to if-conversion are either limited to a single conditional branch using peepholes-style pattern matching, intrinsic functions (conditional code is inlined by the compiler in the internal representation), or scoping larger but restricted regions such as loops hyperblocks. Such approaches require that the region is isolated from the rest of the control flow, and it requires a complicated estimation of the future if-converted region.

A hyperblock [**?**] is a region of code with a single entry and possible, multiple exits where inner branches have been removed by if-conversion. Hyperblock are the basis to enlarge the scope for scheduling in two ways: First, since basick blocks swell as the scope of predicated region grows, static scheduling can now be unconstrained, second, instructions can freely be speculated without the need for compensation code. Conditional code to exit the hyperblock does not impact the execution flow, using a proper code basic block reordering to favor fall thru execution. Tail duplication is used to exclude from the Hyperblock basic blocks that can't be if-

converted, either because the contain hazardous instructions, or because of heuristic decisions.

Generally, hyperblocks are created in the following steps:

- Create a trace and block selection. A trace is a sequence of basic blocks that can be scheduled together.
- Remove side entries with tail duplication, This step removes scheduling constraints imposed by side entries, or allow regions to be if-converted despite hazards.
- Perform if-conversion.

We describe here how Hyperblocks are build using this SSA framework, and how they are inherently part of the if-conversion process.

26.4.1 SSA Iterative if-conversion

The region to if-convert must be carefully selected. Because of resource consumption issues, merging different paths together can over commit the architectural ability to execute in parallel the multiple instructions: The data dependences and registers renaming, introduce new register constraints. Moving operations earlier in the instruction stream increases live-ranges. Another pitfall is to increase the critical path by merging long latency operation from a less frequently executed path, thus moving to the critical paths more instructions than the parallelism can absorb. Finally, for each basic block considered into the region, a predicate must be computed and assigned to the corresponding instructions. Those predicate computations introduce new instructions and new data dependencies.

More classical if-conversion algorithms cover the region-selecton and if-conversion separately, so the region is isolated first and then the work is left to the if-conversion algorithm. All those steps make the if-conversion process a complex optimization, forcing designers to be extremely conservative, as the objective functions must contain variables not known at the start of the transformation. (such as the complexity of the predicate equations) since moving multiple control paths together can easily exceed processors resources (leading to excessive register pressure) or move infrequently used expensive instruction (memory loads) into the critical path.

In contrast, the SSA iterative based if-conversion brings together region selection and region transformation, The algorithm is to walk the control flow iteratively

The algorithm is straightforward. It iterates a structured directed control flow in postorder traversal the list of candidate conditional blocks of the control flow. Postorder traversal garantees that each inner region will be processed before the regions of larger scopes. No sub-region needs to be selected, because the decision to if-convert will be retaken incrementally which each nested gets if-converted and the Hyperblock grows. When the region cannot grow anymore because of resources, or because a basic block cannot be if-converted, then another region is considered.

The bigger adantage on an incremental optimisation of the control flow is that since nested regions are already predicated when evaluating the if-conversion of a branch, all side effects introduced by the if-conversion, such as new predicate merging instructions, new conditional moves merging flow or new data dependencies will be taken into account. Furtheremore, the predicate assignment is not needed, since new predicates are mechanically inserted when merging inner regions containing conditional code.

Since the algorithm processes the control flow in postorder traveral, the dominator tree does not change, and it's possible to maintain the SSA locally to the inner region. By recurrence the if-converted region can be in turn optimized out if it's head belongs to the dominance frontier of an outer region.

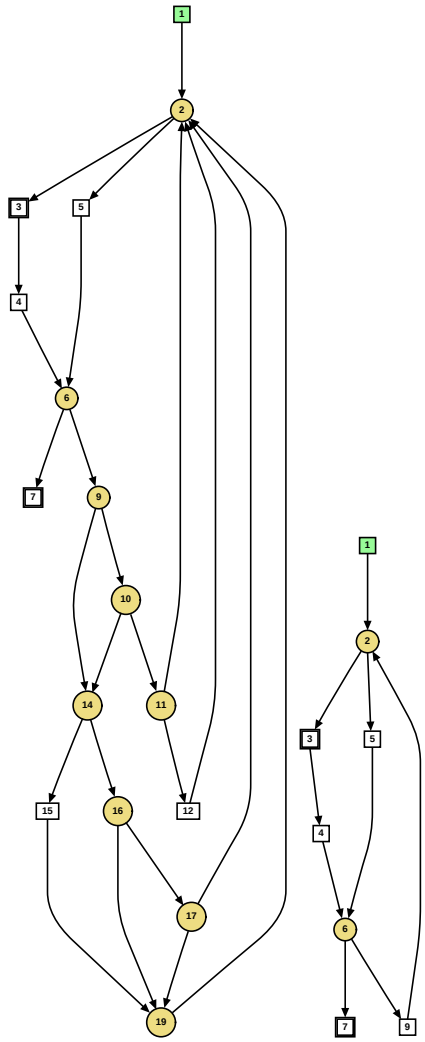
We have proposed a few transformations to convert control flow complex regions into nested Hammocks, such as Basic Block duplication and Predicate merging. The prevalent idea is that the inner region once predicated will be viewed as a single basic region by the outer scopes.

Consider for example the control flow transformations for the *wc* program (figure 26.11a). Exit nodes is BB7. Nodes BB7 and BB3 contains

The postorder list of the basic nested regions is BB11, BB17, BB16, BB14, BB10, BB9, BB6, BB2. The head of nested hammocks is represented by circle nodes. The nodes BB3 and BB7 contains calls, so they need to be excluded from the if-conversion. The first hammock region starting at BB11 to BB2 contains BB12, that is promoted and BB2 becomes the single successor of BB11. For the region starting at BB17, BB19 cannot yet be promoted because of the side entry, so it is duplicated into BB15, that has now BB2 as successor. BB16 is the start of a region containing BB17 and BB19. The former can be merged from the 2 conditions coming from BB16 and BB17. BB17 can be promoted into BB16 under BB16's conditions and BB19 under BB16 and BB17 conditions. Note that at this stage BB16's successors are BB19 and BB2. BB14 is the head of the newly created region where BB15, BB16 and BB17 can be promoted. From BB9 a merging predicate is computed with the one in BB10. Instructions in BB14 are conditionalized on this new predicate and BB10 promoted, forming a new hammock at BB9 with BB14 and BB11 that can be promoted. The last candidate, BB2 is the start of a region that cannot be promoted, because of the call in BB3. The region is now if-converted, leaving a single back-edge, removing 7 branches inside the body loop. At this stage, a local decision can be made to decide if tail-call must be performed to form an hyperblock containing only BB2,BB5,BB6 and BB9, excluding BB3.

26.4.2 Tail Duplication

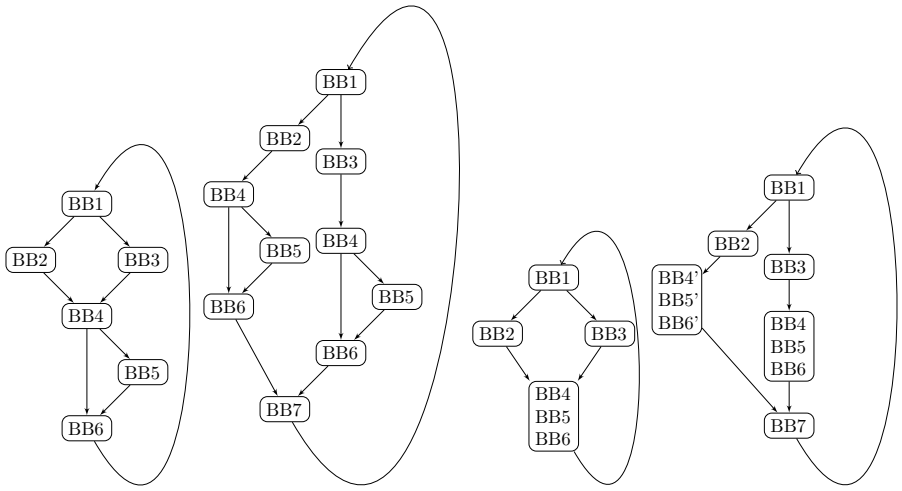
Consider the example of Hyerblock formation 26.11a. This loop contains two branches, and so if-converting it would be profitable. However, block selection has excluded BB2. and has integrated BB5, because heuristics have determined that the schedule of BB5 inside BB4 would be beneficial. The hyperblock contains BB1,



(a) Before if-conversion **(b)** After if-conversion

BB3, BB4, BB5, BB6. Since BB4 has a side entry, it must be removed by tail duplication. 26.11b shows the control flow after block duplication. Notice that a new node, BB7 has been added after the tail duplication by a process called branch coalescing. Finally 26.11d shows the code once if-converted.

The fact that the if-conversion happens after tail-duplication. In the resulting code happens to be difficult to schedule efficiently, over committing resources. Hyperblock formation, if-conversion and speculation introduce a major phase ordering problem. A technique called "Reverse If-Conversion" [**?**] overcomes this problem,



(a) loop **(b)** standard tail-duplication **(c)** after SSA if-conversion **(d)** final flow

allowing use of an aggressive if-conversion framework and reconstructing the control flow at schedule time.

Consider in contrast how tail-duplication is performed in a lazy way, after the code have been if-converted. 26.11c shows the same loop body where the second *if* region has been SSA if-converted. The decision to if-convert the region formed by BB1, BB2, BB3 is now local and can be taken conservatively. Only at this stage, if necessary, tail-duplication can be performed to remove the side entry coming from BB2. Duplicating a single predicated block is now a very simple operation.

26.4.3 Profitability

The decision to include a basic block into an hyperblock is one of the main difficult tradoff that a good if-conversion algorithm as to deal with. It should be aggressive enough to maximize the use of resources, but without overcomitting them, because of the risk of spilling registers or bad schedule if multiple instruction dependent of a scare resource. e.g a multiplier, a data bus a floating point unit. Conventional frameworks that separate the basic block selection from the if-conversion process have a hard time to find the good fit, and need sometime to do a reverse if-conversion [**?**], when the choice appear at-posteriory to be too aggressive.

In contrast, the SSA iterative restructuring framework described earlier reduce the scope for the decision function to a very well localized. Furthermore, the predicate instructions or temporary register to hold the speculated instructions are already

in the code that we are considering. Consequently, SSA iterative if-conversion construction is much more precise. The following factors are considered locally:

- Semantic. The merge of different execution path should guarantee that the semantics of the program are preserved. - If-conversion increases register pressure, for two reasons: live ranges are merged and register renaming imposes new constraints - Schedule length: New data dependencies between a predicate definition and it's use, or between a speculated instruction and it's merge, can impede the schedule.

Since the execution cannot use dynamic branch prediction schemes to statically schedule the code, it is essential for the compiler to rely on accurate static branch prediction information [**?**]. Using such information, the block selection process can decide whether the inclusion of one path into the other will be beneficial.

The first consequence is that all registers are now live simultaneously, (live range), so the benefits of branch removal and better scheduling balanced with increased live range and register pressure : If-conversion must rely on an efficient decision model.

The basic idea for decision taking is that a region can be if-converted if the cost of the resulting if-converted basic block is smaller than the cost of each region taken separately weighted by the branch frequencies. The cost is a factor of 2 parameters: resource usage and schedule length.

We compare the pondered cost of the region before if-conversion. So the cost of a path is the schedule estimation of all the blocks in the path pondered by the probability of execution:

$$Cost(path) = Freq(path) * \sum_{k=1}^{n}(Cost(bb_k)) \tag{26.1}$$

The cost of the region starting at basic block *head* before if-conversion is therefore the cost of all the basic blocks in the considered region, on each path.

$$Cost(head) = Cost(bb_{head}) + branch\ lat + Cost(taken\ path) + Cost(fall - though\ path) \tag{26.2}$$

One of the paths can be empty, in this case only the non empty path will be speculated and it has a zero cost. The Cost after if-conversion is estimated with:

$$Cost(head) = Cost(bb_{head} o bb_{takenpath} o bb_{fall-thoughpath}) \tag{26.3}$$

Where o is the composition function that merges basic blocks together, removes associated branches and creates the predicate operations. The resulting $Cost$ applied to the new basic block represents the estimated schedule after if-conversion, that will be effective only if $Cost(head_{beforeifc}) > Cost(head_{afterifc})$.

The cost function uses the target's machine interface information for instruction latencies, resources usage and scheduling constraints to estimate the local scheduling. The local dependencies computed between instructions are used to compute the dependence height. The branch frequency is obtained either from static branch prediction heuristics, profile information or user inserted directives.

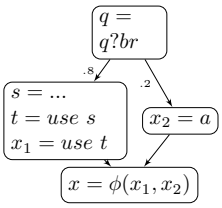


Fig. 26.11 example of profitable if-conversion

In example 26.11 the dependence height of the fall-through path is 4 cycles. The dependence height of the taken path is 2 cycles. So the penetrated estimated cost of the region is 1+2.4+0.2+1=4.6 cycles. The estimated cost of the if-converted region is the schedule height estimation of the instructions without the branches. In this case, 4 cycles. Since the schedule height of the if-converted region is smaller than the profiled estimation, then it is profitable. It should be noted that this heuristic does not take into account increased register pressure in the case of speculation, because the variables now interfere. Computing the inteference graph and simulating a register coalescing process would be extremely expensive at this stage. In the exampe above, a conservative approach would evaluate a register pressure of 4, without counting than *s* and *t* coalesce and that in reality the register budget would be 3. Note that this problem is general to every if-conversion algorithm, but might be more easy to deal with in the SSA framework because of the locality of the region.

26.5 Conclusion

We presented in this chapter a novel if-conversion algorithm that takes advantage of the SSA properties to efficiently assign predicates and lay out the new control flow in an iterative, bottom up process. As opposed to the other top-down approach, this algorithm is very conservative, since the benefit of if-conversion is reevaluated at each nested transformation. We show that we reunified region-formation and if-conversion into a single process, hyperblocks being created lazily, using well known techniques such as tail-duplication or branch coalescing only when the benefit is established. Predication and speculation are often presented as two different alternatives for if-conversion. While it is true than they both require different hardware support, they should coexist in an efficient if-conversion process so every model of conditional execution is accepted. Thanks to conditional moves and ψ transformations, they are now generated together in the same framework. The algorithm has been proved to be efficient in real world algorithms. For example, polynomial evaluation, for which specific ILP algorithm are designed can be fully if-converted in a straight-line of code [**?**], or we measured speedups up to 30% on linux-scalar (spec benchs) or embedded (eembc) application, without code size penalty.

CHAPTER 27

Hardware Compilation using SSA

Pedro C. Diniz
Philip Brisk

Progress: 70%

Structural reordering in progress

Abstract This chapter describes the use of SSA-based high-level program representation for the realization of the corresponding computation using hardware circuits. We begin by highlighting the benefits of using a compiler SSA-based intermediate representation in this hardware mapping process using an illustrative example. The following sections describe hardware translation schemes for the core hardware logic or data-paths of hardware circuits. In this context we also outline several compiler transformations that benefit from the SSA-based representation of a computation. We conclude with a brief survey various hardware compilation efforts both from academia as well as industry that have adopted SSA-based internal representations.

27.1 Brief History and Overview

Hardware compilation is the process by which a high-level language, or behavioral, description of a computation is translated into a hardware-based implementation i.e., a circuit expressed in a hardware design language such as VHDL [**?**] or Verilog [**?**] which can be directly realized as an electrical (often digital) circuit.

While, initially, developers were forced to design hardware circuits using schematic-capture tools, high-level behavioral synthesis allowed them over the years to leverage the wealth of hardware-mapping and design exploration techniques to realize substantial productivity gains. As an example Figure 27.1 illustrates these concepts of hardware mapping for the computation expressed as `x = (a * b) (c * d) +`

`f`. In figure 27.1(b) a graphical representation of a circuit that directly implements this computation is presented. Here there is a direct mapping of hardware operator such as adders and multipliers to the operations in the computation. Values are stored in register and the entire computation lasts a single (albeit long) clock cycle. The input data values are stored in the input registers on the left and a clock cycle later the corresponding results of the computation are ready and saved in the output registers on the right-hand-side of the figure. Overall this direct implementation uses two multipliers, two adders/subtractors and six registers. The execution using this design implementation can use a very simple control scheme, as it simply uses the data in the input registers, waits for a single clock cycles and stores the outputs of the operations in the output registers. In figure 27.1(c) we depict a different implementation variant of the same computation, this time using nine registers and the same amount of adders and subtractors. The increased number of registers allows for the circuit to be clocked at higher frequency as well as to be executed in a pipelined fashion. Finally, in figure 27.1(d) we depict yet another possible implementation of the same computation but using a single multiplier operator. This last version allows for the reuse in time of the multiplier operator and required thirteen register as well as multiplexers to route the inputs to the multiplier in two distinct control steps. As it is apparent, the reduction of number of operators, in this particular case the multipliers carries a penalty on increased number of registers, multiplexers [1] and increased complexity of the control scheme.

This example illustrates the many degrees of freedom in high-level behavioral hardware synthesis. Synthesis techniques perform the classical tasks of allocation, binding and scheduling of the various operations in a computation given specific target hardware resources. For instance, a designer can use behavioral synthesis tools (e.g., Mentor Graphics's Monet®) to automatically derive an implementation for a computation as expressed in the example in Figure 27.1(a) by declaring that it pretends to use a single adder and a single multiplier automatically deriving an implementation that resembles the one depicts in figure 27.1(d). The tool then derives the control scheme required to route the data from registers to the selected units as to meet the designers' goals.

Despite the introduction of high-level behavioral synthesis techniques in commercially available tools, hardware synthesis and thus hardware compilation has never enjoyed the same level of success as traditional, software compilation. Sequential programming paradigms popularized by programming languages such as C/C++ and more recently by Java, allow programmers to easily reason about program behavior as a sequence of program memory state transitions. The underlying processors and the corresponding system-level implementations present a number of simple unified abstractions – such as a unified memory model, a stack, and a heap

[1] A 2×1 multiplexor is a combinatorial circuit with two data inputs, a single output and a control input, where the control input selects which of the two data inputs is transmitted to the output. It can be viewed as a hardware implementation of the C programming language selection operator : `out = (sel ? in1 : in2)`.

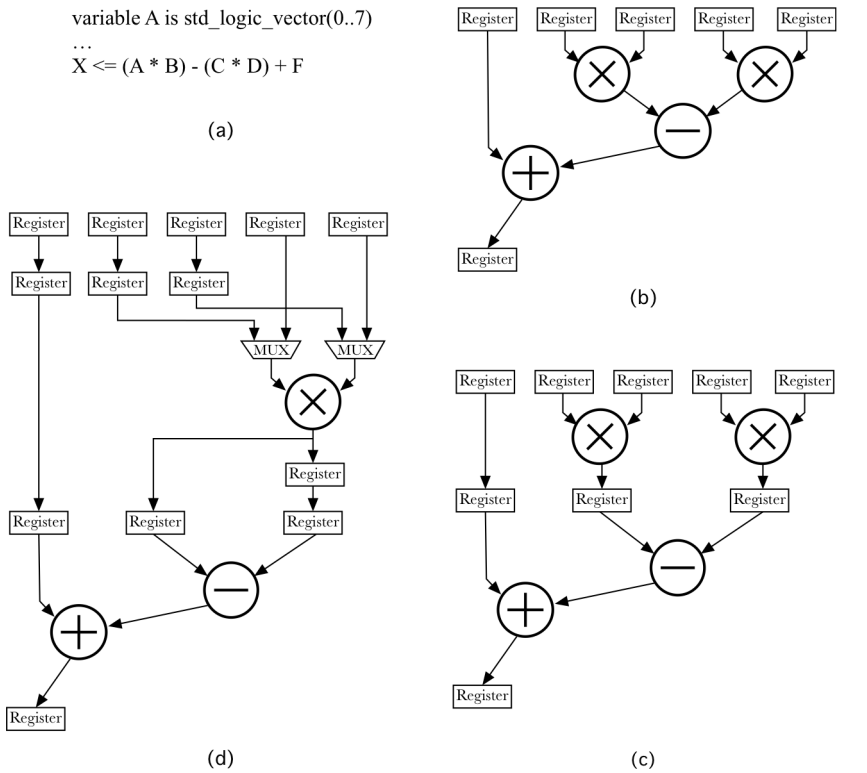


Fig. 27.1 Variants of mapping a computation to hardware: High-level source code (a); a simple hardware design (b); pipelined hardware design (c) design sharing resources (d).

that do not exist (and often do not make sense) in customized hardware designs.

Hardware compilation, in contrast, has faced numerous obstacles that have impeded its progress and generality. When developing hardware solutions, designers must understand the concept of spatial concurrency that circuits do offer. Precise timing and synchronization between distinct hardware components are key abstractions in hardware. Solid and robust hardware design implies a detailed understanding of the precise timing of specific operations, including I/O, that simply cannot be expressed in language such as C, C++, or Java; for this reason, alternatives, such as SystemC [**?**] have emerged in recent years, which give the programmer considerably more control over these issues. The inherent complexity of hardware designs has hampered the development of robust synthesis tools that can offer high-level programming abstractions enjoyed by tools that target traditional architecture and software systems, thus substantially raising the barrier of entry for hardware designers in terms of productivity and robustness of the generated hardware solutions. At best, today hardware compilers can only handle certain subsets of mainstream high-level languages, and at worst, are limited to purely arithmetic sequences of op-

erations with strong restrictions on control flow.

Nevertheless, the emergence of multi-core processing has led to the introduction of new parallel programming languages and parallel programming constructs that may be more amenable to hardware compilation than traditional languages and, similarly, abstractions such as a heap (e.g., pointer-based data structures) and a unified memory address space are proving to be bottlenecks with respect to effective parallel programming. For example, MapReduce, originally introduced by Google to spread parallel jobs across clusters of servers [**?**], has been an effective programming model for FPGAs [**?**]; similarly, high-level languages based on parallel models of computation such as synchronous data flow [**?**], or functional single-assignment languages have also been shown to be good choices for hardware compilation [**?**, **?**, **?**].

Although the remainder of this chapter will be limited primarily to hardware compilation for imperative high-level programming languages with an obvious focus on SSA Form many of the emerging parallel languages will retain sequential constructs, such as control flow graphs, within a larger parallel framework. The extension of SSA Form, and SSA-like constructs, to these emerging languages, is an open area for future research; however, the fundamental uses of SSA Form for hardware compilation, as discussed in this chapter, are likely to remain generally useful.

27.2 Why use SSA for Hardware Compilation?

Hardware compilation, unlike its software counterpart, offers a spatially oriented computational infrastructure that presents opportunities that can leverage information exposed by the SSA representation. We illustrate the direct connection between SSA representation form and hardware compilation using the mapping of a computation example in Figure 27.2(a). Here the value of a variable `v` depends on the control-flow of the computation as the temporary variable `t` can be assigned different values depending on the value of the `p` predicate. The representation of this computation is depicted in Figure 27.2(b) where a ϕ-function is introduced to capture the two possible assignments to the temporary variable `t` in both control branches of the `if-then-else` construct. Lastly, we illustrate in Figure 27.2(c) the corresponding mapping to hardware.

The basic observation is that the confluence of values for a given program variable leads to the use of a ϕ-function. This ϕ-function abstraction thus corresponds in terms of hardware implementation of the insertion of a multiplexer logic circuit. This logic circuit uses the Boolean value of a control input to select which of its input's value is to be propagated to its output. The selection or control input of a multiplexer thus acts as a gated transfer of value that parallels the actions of an

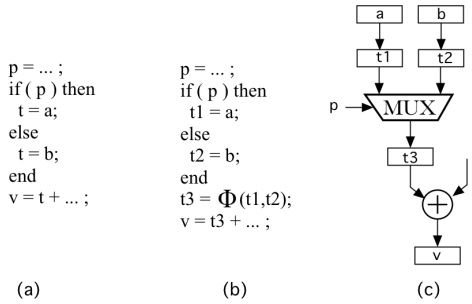


Fig. 27.2 Basic hardware mapping using SSA representation.

`if-then-else` construct in software.

Equally important in this mapping to hardware is the notion that the computation in hardware can now take a spatial dimension. In the suggested hardware circuit in Figure 27.2(c) the computation derived from the statement in both branches of the `if-then-else` construct can be evaluated concurrently by distinct logic circuits. After the evaluation of both circuits the multiplexer will define which set of values are used based on the value of its control input, in this case of the value of the computation associated with `p`.

In a sequential software execution environment, the predicate `p` would be evaluated first, and then either branches of the `if-then-else` construct would be evaluated, based on the value of p; as long as the register allocator is able to assign `t1`, `t2`, and `t3` to the same register, then the ϕ-function is executed implicitly; if not, it is executed as a register-to-register copy.

There have been some efforts that could automatically convert the sequential program above into a semi-spatial representation that could obtain some speedup if executed on a VLIW type of processor. For example, `if-conversion` would convert the control dependency into a data dependency [**?**]: statements from the `if`- and `else` blocks could be interleaved, as long as they do not overwrite one another's values, and the proper result (the ϕ-function) could be selected using a conditional-move instruction; however, in the worst case, this would effectively require the computation of both sides of the branch, rather than one, so it could actually lengthen the amount of time required to resolve the computation; in the spatial representation, in contrast, the correct result can be output as soon as two of the three inputs to the multiplexer are known (`s`, and one of `a` or `b`, depending on the value of `s`).

When targeting a hardware platform, that advantage of the SSA representation is that assigning a value to each scalar variable exactly once makes the sequences of definitions and uses of each variable both formal and explicit. A spatially-oriented infrastructure can leverage this information to perform the computation in ways that would make no sense in a traditional processor. For example, in an FPGA, one can

use multiple registers in space and in time to hold the same variable, and even simultaneously assign distinct values to it; then, based on the outcome of specific predicates in the program, the hardware implementation can select the appropriate register to hold the outcome of the computation. In fact, this is precisely what was done using a multiplexer in the example above.

This example highlights the potential benefits of a SSA-based, when targeting hardware architectures that can easily exploit spatial computation, namely:

- Exposes the data dependences in each variable computation by explicitly incorporating into the representation each variable *def-use* chains. This allows a compiler to isolate the specific values and thus possible registers that contribute to each of the assumed values of the variable. Using separate registers in hardware registers for disjoint live ranges, allows hardware generation to reduce the amount of resources in multiplexer circuits, for example.
- Exposes the potential for the sharing of hardware register not only in time and but also in space providing insights for the high-level synthesis steps of allocation, binding and scheduling.
- Provides insight into control dependent regions where control predicates and thus the corresponding circuitry is identical and can thus be shared. This aspect has been so far neglected but might play an important role in the context of energy minimization.

While the Electronic Design Automation (EDA) community had for several decades now exploited similar information regarding data and control dependences for the generation of hardware circuits from increasingly higher-level representations (e.g., Behavioral HDL), SSA makes these dependences explicit in the intermediate representation itself. Similarly, the more classical compiler representations, using three-address instructions augmented with the *def-use* chains includes, in reality, the same information as the SSA-based representation. In the later however, and as we will explore in the next section, facilitates the mapping and selection of hardware resources.

27.3 Mapping a Control Flow Graph to Hardware

In this section we are interested in hardware implementations or circuit that are spatial in nature and thus we do not address the mapping to architectures such as VLIW [**?**] or Systolic Arrays [**?**]. While these architectures pose very interesting and challenging issues, namely scheduling and resource use, we are more interested in exploring and highlighting the benefits of SSA representation which, we believe, are more naturally (although not exclusively) exposed in the context of spatial hard-

ware computations.

27.3.1 Basic Block Mapping

As a basic block is a straight-line sequence of three-address instructions, a simple hardware mapping approach consists in composing or evaluating the operations in each instruction as a data-flow graph. The inputs and outputs of the instructions are transformed into registers[2] connected by nodes in the graph that represent the operators.

As a result of the ”evaluation” of the instructions in the basic block this algorithm constructs a hardware circuit that has as input registers that will hold the values of the input variables to the various instructions and will have as outputs registers that hold only variables that are live outside the basic block.

27.3.2 Basic Control-Flow-Graph Mapping

One can combined the various hardware circuits corresponding to a control-flow graph in two basic approaches, respectively, *spatial* and *temporal*. The spatial form of combining the hardware circuits consists in laying out the various circuits spatially by connecting variables that are live at the output of a basic block (and therefore corresponding to output registers of the corresponding hardware circuit) to the registers that will hold the values of those same variables in a subsequent hardware circuit corresponds to basic blocks that execute in sequence [**?**, **?**].

In the temporal approach the hardware circuits corresponding to the various CFG basic blocks are not directly interconnected. Instead their input and output registers are connected via dedicated buses to a local storage module. An execution controller ”activates” a basic block or a set of basic blocks by transferring data between the storage and the input registers of the hardware circuits to be activated. Upon execution completion the controller transfer the data from the output registers of each hardware circuit to storage. These data transfers need not necessarily be carried out sequentially but instead can leverage the aggregation of the outputs of each hardware circuits to reduce transfer time to and from storage via dedicated wide buses.

The temporal approach described above is well suited for the scenario where the target architecture does not have sufficient resources to simultaneously implement the hardware circuits corresponding to all basic blocks of interest as it trades off

[2] As a first approach these registers are virtual and then after synthesis some of them are materialized to physical registers in a process similar to register allocation in software-oriented compilation.

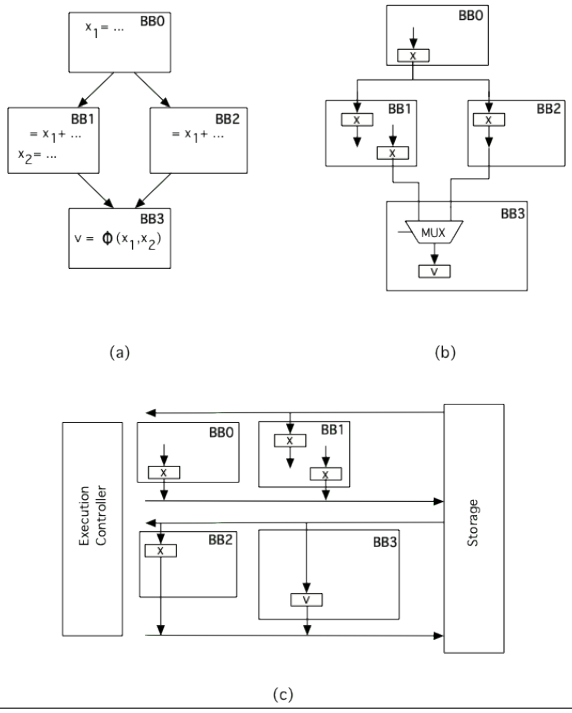


Fig. 27.3 Combination of hardware circuits for multiple basic blocks. CFG representation (a); Spatial mapping (b); Temporal mapping (c).

execution time for hardware resources.

In such a scenario, where hardware resources are very limited or the hardware circuit corresponding to a set of basic blocks is exceedingly large, one could opt for partitioning a basic block or set of basic block into smaller blocks until the space constraints for the realization of each hardware circuit are met. In reality this is the common approach in every processor today. It limits the hardware resources to the resources required for each of the ISA instructions and schedules them in time at each step saving the state (registers) that were the output of the previous instruction. The computation thus proceed as described above by saving the values of the output registers of the hardware circuit corresponding to each smaller block.

These two approaches, illustrated in Figure 27.3, can obviously be merged in a hybrid implementation, lead to distinct control schemes for the orchestration of the execution of computation in hardware and their choice depends heavily on the nature and granularity of the target hardware architecture. For fine-grain hardware architectures such as FPGAs a spatial mapping can be favored, for coarse-grain architectures a temporal mapping is common [**?**].

While the overall execution control for the temporal mapping approach is simpler, as it transfers to and from storage of the variables upon the transfer of control between hardware circuits, a spatial mapping approach makes it more amenable to take advantage of pipelining execution techniques and speculation [3]. The temporal mapping approach can be, however, area-inefficient, as often only one basic block will execute at any point in time. This issue can, nevertheless, be mitigated by exposing additional amounts of instruction-level parallelism by merging multiple basic blocks into a single *hyper-block* and combining this aggregation with loop unrolling. Still, as these transformations and their combination, can lead to a substantial increase of the required hardware resources, a compiler can exploit resource sharing between the hardware units corresponding to distinct basic blocks to reduce the pressure on resource requirements and thus lead to feasible hardware implementation designs (see e.g. [**?**]). As these optimizations are not specific to the SSA representation, we will not discuss them further here.

27.3.3 Control-Flow-Graph Mapping using SSA

In the case of the spatial mapping approach (described in section 27.3.2 the SSA form plays an important role in the minimization of multiplexers and thus in the simplification of the corresponding data-path logic and execution control.

Consider the illustrative example in Figure 27.4(a). Here basic block `BB0` defines a value for the variables `x` and `y`. One of the two subsequent basic blocks `BB1` redefines the value of `x` whereas the other basic block `B2` only reads them.

A naive implementation based exclusively on *live* variable analysis would use for both variables `x` and `y` multiplexers to merge their values as inputs to the hardware circuit implementing basic block `BB3` as depicted in Figure 27.4(b). As can be observed, however, the SSA-form representation captures the fact that such a multiplexer is only required for variable `x`. The value for the variable `y` can be propagated either from the output value in the hardware circuit for basic block `BB0` (as shown in Figure 27.4(c)) or from any other register that has a valid copy of the `y` variable. The direct flow of the single definition point to all its uses, across the hardware circuits corresponding to the various basic blocks in the SSA form thus allows a compiler to use the minimal number of multiplexer strictly required[4].

---

[3] Speculation is also possible in the temporal mode by activating the inputs and execution of multiple hardware blocks and is only limited by the available storage bandwidth to restore the input context in each block which in the spatial approach is trivial.

[4] Under the scenarios of a spatial mapping and with the common disclaimers about static control-flow analysis.

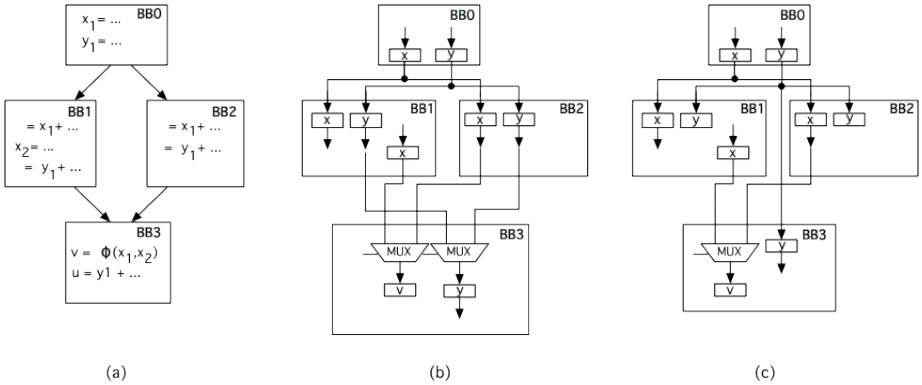


Fig. 27.4 Mapping of variable values across hardware circuit using spatial mapping; CFG in SSA form (a); naive hardware circuit using live variable information for multiplexer placement (b); hardware circuit using ϕ-function for multiplexer placement (c).

An important aspect regarding the implementation of a multiplexer associated with a ϕ-function is the definition and evaluation of the predicate associated with each multiplexer's control (or selection) input signal. In the basic SSA representation the selection predicates are not explicitly defined, as the execution of each ϕ-function is implicit when the control-flow reaches it. When mapping a computation the ϕ-function to hardware, however, one has to clearly define which predicate to use to be included part of the hardware logic circuit that defines the value of the multiplexer circuit's selection input signal. The Gated-SSA form (see e.g. [**?**]) of the SSA form explicitly captures the necessary information in the representation. As the names associated with each value correspond to the precise value (with the correct number index) used in the predicate evaluation this variant of the SSA form is particularly useful for hardware synthesis. The generation of the hardware circuit simply uses the register's value defined in the representation an intermediate value corresponding to the name of each of the variable referenced in the predicate expression of this Gated-SSA form[5]. Figure 27.5 illustrates an example of a mapping using the information gated-SSA form.

When combining multiple predicates in the Gated-SSA form it is often desirable to leverage the control-flow representation in the form of the Program-Dependence-Graph (PDG) [**?**]. In this representation regions of the code, and hence basic blocks, that shared common execution predicates (i.e., both execute under the same predicate conditions) are linked to the same *region* nodes in this representation. Nested execution conditions are easily recognized as the corresponding nodes are hierarchically organized in the PDG representation. As such, when generating code for a given basic block an algorithm will examine the various region nodes associated with a given basic block and compose (using AND operators) the outputs of the logic

[5] As opposed to the base name of the variable as this might have been assigned a value that does not correspond to the actual value used in the predicate.

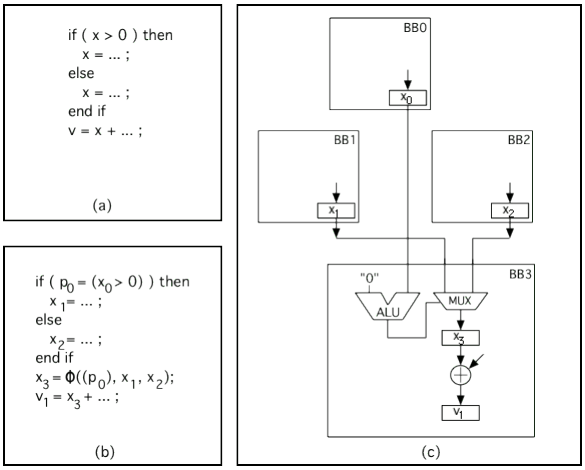


Fig. 27.5 Hardware generation example using Gated-SSA form; original code (a); Gated-SSA representation (b); hardware circuit implementation using spatial mapping (c).

circuits that implementat the predicates associated with these nodes. If an hardware circuit already exists for the same predicate, the implementation simply reuses its output signal. Otherwise, creates a new hardware circuit. This lazy code generation and predicate composition achieves the goal of hardware circuit sharing as illustrated by the example in Figure 27.6 where some of the details were omitted for simplicity.

27.3.4 ϕ-Function and Multiplexer Optimizations

We now describe a set of hardware-oriented transformations that can be applied to improve the amount of hardware resources devoted to multiplexer implementation. Although these transformations are not specific to the mapping of computations to hardware, the explicit representation of the selection constructs in SSA makes it very natural to map and therefore manipulate/transform the resulting hardware circuit using multiplexer. Other operations in the intermediate representation (e.g., predicated instructions) can also yield multiplexers in hardware without the explicit use of SSA Form.

A first transformation is motivated by a well-known result in computer arithmetic: integer addition scales with the number of operands. Building a large unified k-input integer addition circuit is more efficient than adding k integers two at a time [**?**, **?**]. Moreover, hardware multipliers naturally contain multi-operand adders as building blocks: a partial product generator (a layer of AND gates) is followed

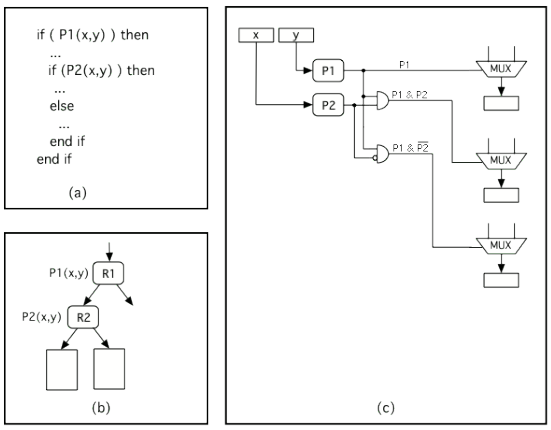


Fig. 27.6 Use of the predicates in region nodes of the PDG for mapping into the multiplexers associated with each ϕ-function; a) source code; b) the PDG representation skeleton; c) outline of hardware design implementation.

by a multi-operand adder called a *partial product reduction tree*. For these reasons, there have been several efforts in recent years to apply high-level algebraic transformations to source code with the goal of merging multiple addition operations with partial product reduction trees of multiplication operations. The basic flavor of these transformations is to push the addition operators toward the outputs of a data flow graph, so that they can be merged at the bottom. Example of these transformations that use multiplexers are depicted in Figure 27.7(a,b). In the case of Figure 27.7(a) the transformation leads to the fact that an addition is always executed unlike in the original hardware design. Figure 27.7(c) depicts a similar transformation that merges two multiplexers sharing a common input, while exploiting the commutative property of the addition operator [**?**]. The SSA-based representation facilitates these transformations as it explicitly indicates which values, and thus by tracing backwards in the representation, are involved in the computation of the corresponding values. For the examples in Figures 27.7(a,b) the compiler would quickly understand the common variables `a` in the two expressions associated with the ϕ-function for the `y` variable.

A second transformation that can be applied to multiplexers is specific FPGA whose basic building block consists of a k-input lookup table (LUT) logic element.

which can be programmed to implement any k-input logic function. For example, a 3-input LUT (3-LUT) can be programmed to implement a multiplexer with two data inputs and one selection bit; similarly, a 6-LUT can be programmed to implement a multiplexer with four data inputs and two selection bits. In particular, many FPGAs devices are organized using 4-LUTs, which are too small to implement a multiplexer with four data inputs, but leave one input unused when implementing multiplexers with two data inputs. This features can be explored to reduce the number of 4-LUTs required to implement a tree of multiplexers [**?**].

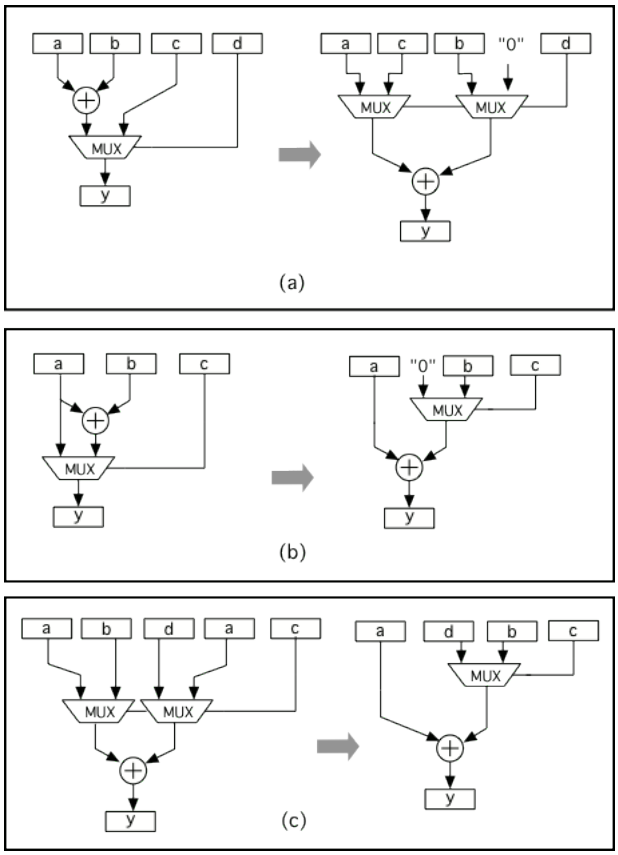


Fig. 27.7 Multiplexer-Operator transformations: juxtapose the positions of a multiplexer and an adder (a,b); reducing the number of multiplexers placed on the input of an adder (c).

27.3.5 Implications of using SSA-form in Floor-planing

For spatial oriented hardware circuits moving a ϕ-function from one basic block to another can alter the length of the wires that are required to transmit data from the hardware circuits corresponding to the various basic blocks. As the boundaries of basic blocks are natural synchronization points, where values are captured in hardware registers, the length of wires dictate the maximum allowed hardware clock rate for synchronous designs. We illustrate this effect via an example as depicted in Figure 27.8 [**?**]. In this figure each basic block is mapped to a distinct hardware unit, whose spatial implementation is approximated by a rectangle. A floor-planning algorithm must place each of the units in a two-dimensional plane while ensuring that no two units overlap. As can be seen in Figure 27.8(a) placing the block 5 on the right-hand-side of the plane will results in several mid-range and one long-range wire connections. However, placing block 5 at the center of the design will virtually eliminate all mid-range connections as all connections corresponding to the transmission of the values for variable **x** are now next-neighboring connections.

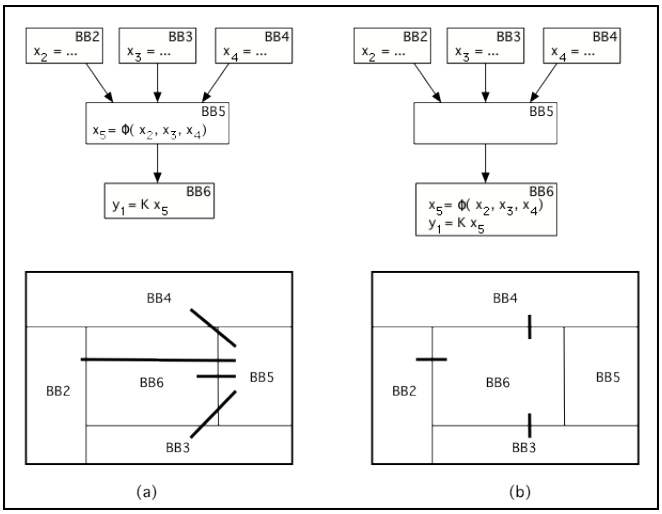


Fig. 27.8 Example of the impact of ϕ-function movement in reducing hardware wire length.

As illustrated by this example, moving a multiplexer from one hardware unit to another can significantly change the dimensions of the resulting unit, which is not under the control of the compiler. Changing the dimensions of the hardware units fundamentally changes the placement of modules, so it is very difficult to predict whether moving a ϕ-function will actually be beneficial. For this reason, compiler optimizations that attempt to improve the physical layout must be performed using a feedback loop so that the results of the lower-level CAD tools that produce the

layout can be reported back to the compiler.

27.4 Existing SSA-based Hardware Compilation Efforts

Several research projects have relied on SSA-based intermediate representation that leverage control- and data-flow information to exploit fine grain parallelism. Often, but not always, these efforts have been geared towards mapping computations to fine-grain hardware structure such as the ones offered by FPGAs.

The standard approach followed in these compilers has been to translate a high-level programming language such as Java in the case of the Sea Cucumber [**?**] compiler or C in the case of the ROCCC [**?**] to sequences of intermediate instructions. These sequences are then organized in basic blocks that compose the control-flow graph (CFG). For each basic block a data-flow-graph (DFG) is typically extracted followed by conversion in to SSA representation, possibly using predicates associated with the control-flow in the CFG thus explicitly using Predicated SSA representation (PSSA [**?**, **?**, **?**]).

As an approach to increase the potential amount of exploitable instruction-level parallelism (ILP) at the instruction level, many of these efforts (as well as others such as the earlier Garp compiler [**?**]) restructure the CFG into hyper-blocks [**?**]. An hyper-block consists on a single-entry multi-exit regions derived from the aggregation of multiple basic blocks thus serializing longer sequences of instruction. As not all instructions are executed in a hyper-block (due to early exit of a block), hardware circuit implementation must rely on predication to exploit the potential for additional ILP.

The CASH compiler [**?**] uses an augmented predicated SSA representation with tokens to explicitly express synchronization and handle *may-dependences* thus supporting speculative execution. This fine-grain synchronization mechanism is also used to serialize the execution of consecutive hyper-blocks, thus greatly simplifying the code generation.

CHAPTER 28

php experience report

P. Biggar

Progress: 55%

Review in progress

28.1 Abstract

Constructing SSA form for static languages is a well-studied problem. Techniques exist to handle the most common features of static languages, and these solutions have been tried and tested in production level compilers over many years.

In our study of optimizing dynamic scripting languages, specifically PHP, we find this is not the case. The information required to build SSA form—that is, some conservatively complete set of unaliased scalars, and the locations of their uses and definitions—is not available directly from the program source, and can not be derived from a simple analysis. Instead, we find a litany of features whose presence must be ruled out, or heavily analysed, in order to obtain a non-pessimistic estimate.

Scripting languages commonly feature run-time code generation, built-in functions, and variable-variables, all of which may alter arbitrary unnamed variables. Less common—but still possible—features include the existence of object handlers which have the ability to alter any part of the program state, most dangerously a function's local symbol-table.

Ruling out the presence of these features requires precise, inter-procedural, whole-program analysis. We discuss the futility of the pessimistic solution, the analyses required to provide a precise SSA form, and how the presence of variables of unknown types affect the precision of SSA.

28.2 Motivation

Dynamic scripting languages such as PHP, Python and Javascript are among the most widely-used and fastest growing programming languages.

Scripting languages provide flexible high-level features, a fast modify-compile-test environment for rapid prototyping, strong integration with traditional programming languages, and an extensive standard library. Scripting languages are widely used in the implementation of many of the best known web applications of the last decade such as Facebook, Wikipedia, Gmail and Twitter. Many prominent web sites make use significant of scripting, not least because of strong support from browsers and simple integration with back-end databases.

One of the most widely used scripting languages is PHP, a general-purpose language that was originally designed for creating dynamic web pages. PHP has many of the features that are typical of dynamic scripting languages. These include simple syntax, dynamic typing, interpreted execution and run-time code generation. These simple, flexible features facilitate rapid prototyping, exploratory programming, and in the case of many non-professional websites, a copy-paste-and-modify approach to scripting by non-expert programmers.

Building SSA form for programs in languages like C/C++ and Java is reasonably straightforward. In these traditional languages it is not difficult to identify a set of scalar variables that can be safely renamed. Better analysis may lead to more such variables being identified, but significant numbers of such variables can be found with very simple analysis.

In PHP, however, simply finding the variables that which may be renamed into SSA is difficult, and in fact, requires the kind of analysis that we would like to do in SSA form. The rest of this chapter describes our experiences with building SSA in PHC, our open source compiler for PHP. We identify the features of PHP that make building SSA difficult, outline the solutions we found to these some of these challenges, and draw some lessons about the use of SSA in analysis frameworks for PHP.

28.3 SSA form in “traditional” languages

In traditional, non-scripting languages, such as C, C++ and Java, it is straightforward to identify which variables may be converted into SSA form. In Java, all scalars may be renamed. In C and C++, any scalar variables which do not have their address taken may be renamed. Figure 1 shows a simple C function, whose local variables may all be converted into SSA form. In Figure 1, it is trivial to convert each variable into SSA form. For each statement, the list of variables which are used and defined are immediately obvious from a simple syntactic check.

By contrast, Figure 2 contains variables which cannot be trivially converted into SSA form. On line 2, x has its address taken. As a result, to convert x into SSA form, we must know that line 3 modifies x, and change x's subscript accordingly. Chapter hssa describes HSSA form, a powerful way to represent *indirect* modifications to scalars in SSA form.

To discover the modification of x on line 3 requires an *alias analysis*. Alias analyses detect when multiple program names (*aliases*) represent the same memory location. In Figure 2, x and $*y$ alias each other.

There are many forms of alias analysis, of varying complexity. The most complex analyse the entire program text, taking into account control-flow, function calls, and multiple levels of pointers. However, it is not difficult to perform a very simple alias analysis. *Address-taken alias analysis* identifies all variables whose addresses are taken by a referencing assignment. All these variables in the program are considered to alias each other (that is, all address-taken variables are in the same *alias set*).

When building SSA form with address-taken alias analysis, variables in the alias set are not renamed into SSA form. All other variables are converted. Variables not in SSA form do not possess any SSA properties, and pessimistic assumptions must be made.

As a result of address-taken alias analysis, it is straight-forward to convert any C program into SSA form, without sophisticated analysis. In fact, this allows more complex alias analyses to be performed on the SSA form.

28.3.1 PHP

28.3.1.1 Aliases

In PHP, it is not possible to perform an address-taken alias analysis. The syntactic clues that C provides—notably as a result of static typing—are not available in PHP. Indeed, statements which may appear innocuous may perform complex operations behind the scenes, affecting any variable in the function.

In addition, it is not possible to syntactically identify scalars in PHP. PHP variables may be references at one point in the program, and stop being references a moment later. PHP's dynamic typing makes it difficult to simply identify when this occurs, and a sophisticated whole program analysis is essential to building a conservative SSA form.

Consider the PHP program in Figure 3. Superficially, it resembles the C function in Figure 4. In the C version, we know that y is a scalar which does not alias x, so the addition operation has no effect on the result.

In the PHP version, $\$x$ may alias $\$y$ upon function entry. The addition operation may therefore change the value of $\$y$. It therefore more closely resembles the C function in Figure 5.

In the PHP version, there are no syntactic clues that the variables may alias. As a result, a conservative aliasing estimate—like C's address-taken alias analysis—

would need to place all variables in the alias set. This would leave no variables available for conversion to SSA form.

28.3.2 Whole-program analysis

PHP's dynamic typing means that program analysis cannot be performed a fucntion at a time. As function signatures do not indicate whether parameters are references, this information must be determined by program analysis. Furthermore, each function must be analysed with full knowledge of its calling context. This requires a whole-program analysis.

We present an overview of the analysis below. A full description is beyond the scope of this chapter, and can be found elsewhere[**?**].

The analysis is structured as a symbolic execution. This means the program is analysed by processing each statement in turn, and modelling the affect of the statement on the program state. This is similar in principle to the SCCP algorithm presented in Chapter sccp.[1]

The SCCP algorithm models a function at a time. Instead, our algorithm models the entire program. The execution state of the program begins empty, and the analysis begins at the first statement in the program, which is placed in a worklist. The worklist is then processed a statement at a time.

For each analysed statement, the results of the analysis are stored. If the analysis results change, the statements successors (in the control-flow graph) are added to the worklist. The first time a statement is processed, its results are considered to always change. This is similar to CFG-edges in the SCCP algorithm. There is no parallel to the SSA edges, since the analysis is not performed on the SSA form. Instead, loops must be fully analysed if their header's change.

This analysis is therefore less efficient than the SCCP algorithm, in terms of time. It is also less efficient in terms of space. As described in Figure gcc, SSA form allows results to be compactly stored in a single array, using the SSA index as an array index. This is very space efficient. In our analysis, we must instead store a table of variable results at all points in the program.

Upon reaching a function or method call, the analysis begins analysing the callee function, pausing the caller's analysis. A new worklist is created, and initialized with the first statement in the callee function. The worklist is then run until it is exhausted. If another function call is analysed, the process recurses.

Upon reaching a callee function, the analysis results are copied into the scope of the callee. Once a worklist has ended, the analysis results for the exit node of the function are copied back to the calling statement's results.

[1] The analysis is actually based on a variation, CCP [**?**].

28.3.3 Analysis results

Our analysis stores a number of kinds of results. Each kind of result is stored at each point in the program.

The first models the alias relationships in the program in a *points-to graph* [**?**]. The graph contains variable names as nodes, and the edges between them indicate aliasing relationships. An aliasing relationship indicates that two variables either *must*-alias, or *may*-alias. Two unconnected nodes cannot alias. A points-to graph is stored for each point in the program. Graphs can be merged at CFG join points.

It is also important to store the types of variables in the program. Since PHP is an object-oriented language, polymorphic method calls are possible, and they must be analysed. As such, the set of types of each variable is stored at each point in the program. This portion of the analysis closely resembles using SCCP for type propagation [**?**], as described in Chapter sccp.

Finally, like the SCCP algorithm, constant are recorded. The phc compiler creates many statically-resolvable branches during early stages of compilation. Resolving these branches statically eliminates unreachable paths, leading to significantly more precise results.

28.3.4 Object handlers

PHP's reference implementation allows classes to be partially written in C. Objects which are instantiations of these classes have special behaviour in certain cases. For example, the objects may be dereferenced using array syntax, or a special handler may be called when the variable holding the object is read or written to.

The handlers for these special cases are generally unrestricted. Being written in C, they are given access to the entire program state, including all local and global variables.

There two characteristics of handlers make them very powerful, but break any attempts at function-at-a-time analysis. If one of these objects is passed in to a function, and is read, it may overwrite all variables in the local symbol table. The overwritten variables might then have the same handlers. These can then be returned from the function, or passed to any other called functions (indeed it can also call any function). This means that a single unconstrained variable in a function can propagate to any other point in the program.

28.3.5 Building def-use chains

Def-use chains are required to build SSA form. Since we cannot create these syntactically, we build them as we analyse the program. Using the alias analysis, any variables which may be written to or read from during a statement's executation,

are added to a set of defs and uses for that statement. These are then used during construction of the SSA form.

For an assignment by copy, $x = $y:

1. $x's value is defined.
2. $x's reference is used (by the assignment to $x).
3. for each alias $x' of $x, $x''s value is defined. If the alias is possible, $x''s value is may-defined instead of defined. In addition, $x''s reference is used.
4. $y's value is used.
5. $y's reference is used.

For an assignment by reference, $x =& $y:

1. $x's value is defined.
2. $x's reference is defined (it is not used—$x does not maintain its previous reference relationships).
3. $y's value is used.
4. $y's reference is used.

28.3.6 HSSA

After building def-use chains, SSA form can be build in the normal way. However, may-definitions must be represented. Many of PHP's features appear as may-definitions: indirect assignments through references, variable-variables, special function calls. As such, we use a variation of HSSA, described in Chapter hssa, to build our SSA form.

We add CHI nodes, as in HSSA, to model may-definitions. In the phc compiler, we did not implement any further features of HSSA form, such as zero versioning or global value numbering. However, there is no reason these could not be used to reduce the memory consumption of our SSA form.

The final distinction from traditional SSA form is that we do not only model variables. All names in the program, such as fields of objects or the contents of arrays, can be represented in SSA form.

28.3.6.1 Run-time symbol tables

Figure 6 shows a program which accesses a variable indirectly. On line 2, a string value is read from the user, and stored in the variable $\$var_name$. On line 3, some variable—whose name is the value in $\$var_name$—is set to 5. That is, any variable can be updated, and the updated variable is chosen by the user at run-time. It is not possible to know whether the user has provided the value "x", and so know whether $\$x$ has the value 5 or 6.

This feature is known as *variable-variables*. They are possible because a function's symbol-table in PHP is available at run-time. Variables in PHP are not the same as variables in C. A C local variable is a name which represents a memory location on the stack. A PHP local variable is the domain of a map from strings to values. The same run-time value may be the domain of multiple string keys (references, discussed above). Similarly, variable-variables allow the symbol-table to be accessed dynamically at run-time, allowing arbitrary values to be read and written.

Upon seeing a variable-variable, all variables may be written to. In HSSA form, this creates a CHI node for each variable in the program. In order to avoid this, the variable-variable may be modelled using string analysis **??**. String analysis models the structure of strings. For example, we may know that the variable begins with the letter "x". In this case, all variables which do not begin with "x" do not require a CHI node, leading to a more precise SSA form.

Variable-variables are not the only PHP feature which can affect the local symbol-table. Figure 7 shows a function call which can affect the variables in the scope of the caller. Figure 8 shows the use of an *eval* statement, which executes an arbitrary string. Both of these may be modelled using the same string analysis techniques as above [**?**]. The *eval* statement may also be handled using profiling [**?**], which restricts the set of possible *eval* statements to those which actually are used in practice.

28.3.7 Implications

- SSA cannot be used as end-to-end IR.
  - propagation algorithms do not benefit from the sparse form

The use of HSSA form means that optimizations which lead to overlapping live-ranges, are not possible. Since the phc compiler does not perform register allocation, this is not a problem.

The implications of needing whole-program analysis are more severe, however. As this analysis is not performed on the SSA form, it is not possible to have an end-to-end SSA-based compiler for PHP.

The whole program analysis described above is based on the CCP form. It would be more efficient, in terms of both space and time, to use SCCP to perform the analysis.