Compiler Optimisation
of
Typeless Languages

Panayis Fourniotis Pavlatos

Trinity College
University of Cambridge

A dissertation submitted for the degree of
Doctor of Philosophy

May 1998
Summary

We have written an optimising compiler for a typeless, imperative, modular programming language. The optimiser, which works on a 3-address intermediate representation generated from the source program, uses some novel techniques described in this thesis. The techniques are universally applicable, although some are particularly useful in typeless compilation.

We present a new register allocation and assignment scheme. Unlike traditional "colouring" allocators, our method separates the problem into distinct allocation and assignment phases. The former is achieved by using an iterative process to extend a local (within basic blocks) allocation method to the global (across basic blocks) domain. This obviates the need for a sophisticated assignment algorithm; we show how to use simple heuristics to assign registers after allocation.

We also present a simple method for identifying loops in a program's intermediate representation and assigning loop nesting levels. Unlike traditional methods, this does not rely on the concept of flowgraph dominators, and is able to deal sensibly with irreducible flowgraphs and "unstructured" loops that interlock or partially overlap.

The major part of the thesis concerns value range analysis. Based on the theoretical framework of abstract interpretation, we describe an analysis of the intermediate code that predicts safe approximations to the run-time value ranges of variables and memory used by the program being compiled. To be useful in compiling a typeless language, this analysis must be able to handle values of different kinds (integers, pointers, function addresses, etc.)

We show how we can subsume some traditional optimisation techniques, such as constant propagation, into more powerful methods that take advantage of value range information to optimise a wider variety of cases. We also show how this information can be used to recover most of the benefits of types, without sacrificing the flexibility of typelessness.

Besides the above, value range analysis allows a number of optimisations that were heretofore impossible. Many of these are improvements to register allocation; we investigate better treatments for variables that can be accessed by address. We also describe a method of removing memory accesses by allowing variables that are simultaneously live to share registers, and suggest a similar scheme for values stored in memory.

Finally, we show how the results of value range analysis can be shared across different program modules and different compiler runs. The method used is powerful enough to be useful, but simple enough to integrate with old code that cannot be recompiled. Inter-modular optimisation can be transparent to the user, improving the results of value range analysis within a module without altering its functionality; or it can be visible, optimising modules with respect to each other.
Declaration

Except as otherwise stated in the text, this dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration.

This dissertation is not substantially the same as any that I have submitted for a degree or diploma or other qualification at any other University.

No part of my dissertation has already been or is being concurrently submitted for any such degree, diploma or other qualification.

Acknowledgements

The first three years of my graduate studies were funded by a Trinity College Internal Graduate Studentship.

I am grateful to my supervisor, Dr. Martin Richards, for his guidance, comments, and patience; my parents, for their moral and financial support; and my friends, on and off the Net, for keeping me sane throughout my time as a graduate student.
Contents

1 Introduction .................................................. 1
  1.1 Languages and Compilers .................................. 1
  1.2 Compilation and Optimisation ............................ 2
  1.3 Effect of Language Design on Optimisation ............... 2
    1.3.1 Low-level Languages .................................. 2
    1.3.2 High-level Languages ................................. 3
  1.4 Scope and Outline of This Thesis .......................... 3

2 P: The Language .............................................. 5
  2.1 Introduction .............................................. 5
  2.2 Why Another Language? ..................................... 5
  2.3 P: A Typeless Language ................................... 6
    2.3.1 What Is a Type? ....................................... 6
    2.3.2 Typeless Languages ................................... 7
  2.4 P: An Imperative Language ............................... 7
    2.4.1 Imperative vs. Functional Programming ............. 7
    2.4.2 Where Does P Stand? .................................. 8
  2.5 P: A Modular Language ................................... 9
    2.5.1 Separate Compilation .................................. 9
    2.5.2 Problems with Header Files .......................... 10
    2.5.3 Modules ............................................... 10
    2.5.4 The P Approach to Modules ........................... 11
  2.6 Other Features ........................................... 12
    2.6.1 Reserved Words ....................................... 12
    2.6.2 Flow of Control Constructs ......................... 12
    2.6.3 Labels and Procedures; Jumps and Calls ............. 12
    2.6.4 Operators ............................................. 13
    2.6.5 Memory Access ........................................ 13
    2.6.6 Standard Library ...................................... 14

3 The P Compiler: Conventional Features ..................... 17
  3.1 Introduction .............................................. 17
  3.2 Why Another Compiler? ................................... 17
  3.3 The Front End ............................................. 18
    3.3.1 Overview ............................................. 18
    3.3.2 The Lexical Analyser .................................. 19
    3.3.3 The Program Parser ................................... 19
    3.3.4 The Translator ....................................... 21
  3.4 The Back End .............................................. 24
    3.4.1 Overview ............................................. 24
    3.4.2 Peephole Optimisation ................................ 24
    3.4.3 Code Generation ...................................... 24
CONTENTS

3.5 The Optimiser ................................................. 25
   3.5.1 Overview ............................................. 25
   3.5.2 Dead Code Elimination ................................. 25
   3.5.3 Liveness Analysis ..................................... 29
   3.5.4 Simple Applications of Liveness Information .............. 31

4 Loop Detection ................................................. 33
   4.1 Introduction ............................................. 33
   4.2 Overview of Related Work ................................ 33
   4.3 Loop Detection: Objectives ............................... 35
   4.4 Detecting Loops and Assigning Nesting Levels ............... 36
      4.4.1 Concepts ........................................... 36
      4.4.2 The Algorithm ...................................... 36
      4.4.3 Properties ......................................... 37
      4.4.4 Examples ........................................... 38

5 Register Allocation ............................................ 39
   5.1 Introduction ............................................. 39
   5.2 Overview of Related Work ................................ 39
      5.2.1 Introduction ......................................... 39
      5.2.2 Local Binding ....................................... 40
      5.2.3 Global Binding ....................................... 40
   5.3 Objectives ................................................ 43
   5.4 Introduction to Register Allocation ......................... 44
   5.5 Distances to Next Use ................................... 45
      5.5.1 Non-Branching Code .................................. 45
      5.5.2 Code with Branches ................................... 46
      5.5.3 Convergence ......................................... 47
      5.5.4 Implementation Details .............................. 47
      5.5.5 Execution Cost ....................................... 48
      5.5.6 Remarks ............................................. 49
   5.6 Introducing Spills ........................................ 51
      5.6.1 Non-Branching Code: Reloading Vregs ................. 51
      5.6.2 Non-Branching Code: Spilling Vregs ................. 51
      5.6.3 Non-Branching Code: Special Cases .................... 52
      5.6.4 Non-Branching Code: Wrapping it Up ................... 53
      5.6.5 Code with Branches ................................... 55
      5.6.6 Convergence ......................................... 57
      5.6.7 Implementation Details .............................. 57
      5.6.8 Execution Cost ....................................... 59
      5.6.9 Remarks ............................................. 59
   5.7 An Example ............................................... 61

6 Register Assignment ............................................. 65
   6.1 Introduction ............................................. 65
   6.2 Span Limits ............................................... 65
   6.3 Span Boundary Conditions ................................. 66
   6.4 Definition of Spans ...................................... 67
   6.5 Identifying Spans: Implementation ......................... 68
   6.6 Assignment: Choice of Algorithms ......................... 71
   6.7 Candidate Propagation ................................... 72
      6.7.1 The Algorithm ....................................... 72
      6.7.2 Execution Cost ....................................... 74
   6.8 Register Assignment ...................................... 74
6.8.1 The Algorithm .................................. 74
6.8.2 Execution Cost .................................. 75
6.9 Assignment Synchronisation .......................... 75
6.10 An Example ....................................... 77

7 Range Analysis: Introduction and Theory ................. 81
  7.1 Introduction ..................................... 81
  7.2 Overview of Related Work .......................... 81
  7.3 Objectives ....................................... 83
  7.4 Abstract Interpretation ............................ 84
     7.4.1 Introduction and Apologetics ................. 84
     7.4.2 Operational Semantics ....................... 84
     7.4.3 Static Semantics ............................. 88
     7.4.4 Abstract Semantics ......................... 89
     7.4.5 Widening and Narrowing ...................... 90

8 The Value Range Analysis Lattice ......................... 93
  8.1 Introduction ..................................... 93
  8.2 Abstract Values .................................. 93
     8.2.1 How Much to Abstract? ...................... 93
     8.2.2 Introduction to Value Kinds ................. 94
     8.2.3 Program Labels ................................ 96
     8.2.4 Procedure Addresses ......................... 96
     8.2.5 Variable Addresses .......................... 96
     8.2.6 Integers .................................... 96
     8.2.7 Static Data Addresses ....................... 97
     8.2.8 Dynamic Stack Data Addresses .............. 98
     8.2.9 Dynamic Heap Data Addresses ............... 98
  8.3 Value Ranges, Environments and Associations ......... 99
     8.3.1 Scalar Value Ranges ......................... 99
     8.3.2 Ordering the Abstract Values ............... 101
     8.3.3 Block Value Ranges ......................... 101
     8.3.4 Abstract Contexts ........................... 102
     8.3.5 A Problem: Antisocial Value Ranges .......... 102
     8.3.6 A Solution: Associations .................... 103
  8.4 Fundamental Operations on Abstract Contexts ........... 105
     8.4.1 Updating Component Identifiers .............. 105
     8.4.2 Rebuilding Associations ..................... 105
     8.4.3 The Join Operator ........................... 107
     8.4.4 The Restrict Operator ....................... 107
     8.4.5 The Split Operator .......................... 108
     8.4.6 Enhanced Split and Restrict Operators ........ 110
     8.4.7 Ordering the Abstract Contexts Lattice .... 110
  8.5 Abstraction and Concretisation Operators ............ 111
     8.5.1 Introduction ................................ 111
     8.5.2 Concretisation .............................. 112
     8.5.3 Abstraction ................................ 113
  8.6 Widening and Narrowing Operators .................... 115
     8.6.1 Introduction ................................. 115
     8.6.2 Widening and Narrowing of Integer Intervals .. 115
     8.6.3 Widening and Narrowing of Scalar Ranges ...... 116
     8.6.4 Widening and Narrowing of Memory Contents .... 117
     8.6.5 Widening and Narrowing of Abstract Contexts .... 117
     8.6.6 Remarks .................................... 118
CONTENTS

8.7 Implementation Details ........................................ 119
  8.7.1 Introduction .................................................. 119
  8.7.2 Scalar Value Ranges .......................................... 119
  8.7.3 Memory Block Value Ranges .................................. 119
  8.7.4 Range Tables .................................................. 120
  8.7.5 Associations .................................................. 120
  8.7.6 Operations .................................................... 120

9 Abstract Interpretation of WIM3 Code ....................... 123
  9.1 Introduction and Overview .................................... 123
  9.2 Insn Interpretation: Preliminaries ........................... 123
    9.2.1 Obtaining Ranges of WIM3 Arguments ..................... 123
    9.2.2 Normalisation of Insn Results ............................ 124
    9.2.3 Storing Results of WIM3 Insns ............................ 125
    9.2.4 Block Entry ................................................ 125
  9.3 Arithmetic and Logic Insns ................................... 126
    9.3.1 Addition and Subtraction ................................ 126
    9.3.2 Multiplication and Division ................................ 128
    9.3.3 Shifts ..................................................... 129
    9.3.4 Bitwise Logical Operations ............................... 130
  9.4 Conditional and Jump Insns ................................... 130
    9.4.1 Set On Less Than ........................................... 130
    9.4.2 Jump if Less Than ......................................... 132
  9.5 Memory Access Insns ........................................... 133
    9.5.1 Load from Memory .......................................... 133
    9.5.2 Store to Memory ............................................ 134
  9.6 Miscellaneous Insns ............................................ 137
    9.6.1 Procedure Entry ............................................ 137
    9.6.2 Procedure Return .......................................... 137
    9.6.3 Procedure Call .............................................. 137
    9.6.4 Spills and Reloads ........................................... 137
  9.7 Convergence and Correctness ................................ 137
    9.7.1 Introduction ................................................. 137
    9.7.2 Monotonicity of $\text{NEXT}_\text{abs}$ ..................... 138
    9.7.3 Consistency of $\text{NEXT}_\text{cost}$ and $\text{NEXT}_\text{abs}$ 138
  9.8 Implementation Details ....................................... 139
    9.8.1 Introduction ................................................ 139
    9.8.2 Applying the Interpretation ............................... 139
    9.8.3 Execution Cost .............................................. 139
    9.8.4 Addition and Subtraction .................................. 140
    9.8.5 Multiplication, Division, and Shifts ..................... 140
    9.8.6 Other Insns ............................................... 140

10 Range Analysis: Applications ............................... 141
  10.1 Abstract Interpretation Phases ............................ 141
  10.2 Simple Applications of Value Range Analysis .............. 142
    10.2.1 Introduction ............................................... 142
    10.2.2 Jump Destination Determination ........................... 142
    10.2.3 Constant Propagation ...................................... 143
    10.2.4 Conditional Jump Reduction ............................... 145
    10.2.5 Alias Analysis .............................................. 146
    10.2.6 Accesses to Addresses of Variables ..................... 146
    10.2.7 Advisory Type Checking .................................... 147
  10.3 Interaction with Register Binding .......................... 151
10.3.1 Introduction ........................................ 151
10.3.2 Elimination of Precautionary Stores and Spills ......... 151
10.3.3 Value Sizes ........................................ 151
10.3.4 Register Stuffing: Theory ................................ 153
10.3.5 Register Stuffing: Practice ................................ 157
10.3.6 Register Stuffing: Implementation Details .......... 163
10.3.7 Register Stuffing: Conclusions ......................... 163
10.3.8 Variations on the Theme ................................ 164
10.4 String Packing ....................................... 165
10.4.1 Can We Do It? ....................................... 165
10.4.2 How to Do It ....................................... 165
10.5 Language Extensions .................................. 167
10.5.1 Introduction ....................................... 167
10.5.2 Extensions to Range Analysis ......................... 167
10.5.3 Other Language Extensions ............................ 168
10.6 Examples ............................................ 170
10.6.1 Introduction ....................................... 170
10.6.2 Optimisation of Conditionals ......................... 171
10.6.3 Optimisation of MCPL Matches ........................ 171

11 Optimisations Across Module Boundaries .......................... 175
11.1 Introduction ........................................ 175
11.2 Overview of Related Work ................................ 176
11.3 Interfaces and Annotations ................................ 176
11.3.1 What Is an Annotation File? ......................... 176
11.3.2 Format of Annotation Files ............................ 177
11.3.3 Contents of Annotation Files: Exported Variables .... 177
11.3.4 Contents of Annotation Files: Procedures ............ 177
11.3.5 Conditional Annotations .............................. 178
11.3.6 Interpreting the enter insn .......................... 179
11.4 Building Annotation Files ................................ 179
11.4.1 Representing Annotations ............................. 179
11.4.2 Preliminaries ....................................... 180
11.4.3 Argument Count ..................................... 181
11.4.4 Return Values ....................................... 181
11.4.5 Procedure Calls ..................................... 181
11.4.6 Memory References .................................. 182
11.4.7 Writing to Memory ................................... 182
11.4.8 Generating Annotation Files ........................... 183
11.5 Refining Annotations .................................... 183
11.5.1 Introduction ....................................... 183
11.5.2 Responding to Reference Annotations .................. 183
11.5.3 Kind Inference ...................................... 184
11.6 Using Annotations .................................... 186
11.6.1 Introduction ....................................... 186
11.6.2 Detecting Recursive Procedures ....................... 186
11.6.3 Elimination of Precautionary Stores and Spills ....... 186
11.6.4 Interpreting the call insn ............................ 187
11.6.5 Specialising Modules ................................. 188
11.7 Implementation Details .................................. 189
11.7.1 The Annotation Repository ............................ 189
11.7.2 Reading Annotation Files ............................. 189
11.7.3 Writing Annotation Files .............................. 190
11.8 Extending Annotation Files ................................ 190
CONTENTS

11.9 Examples ............................................. 191
  11.9.1 Register Allocation ................................. 191
  11.9.2 Conditional Annotations ............................. 191
  11.9.3 Character Input .................................. 193
  11.9.4 Typeless Objects .................................. 194

12 Conclusions and Further Work ........................... 199
  12.1 Conclusions .......................................... 199
  12.2 Further Work ......................................... 200

A Mathematical Notation ................................... 203

B P Reference Manual ....................................... 211
  B.1 Syntax .............................................. 211
  B.2 Informal Semantics ................................... 214
    B.2.1 Values and Evaluation ............................. 214
    B.2.2 Top Level ....................................... 214
    B.2.3 Procedures ...................................... 215
    B.2.4 Variable Declarations ............................ 215
    B.2.5 Sequence Expressions and Results .................. 216
    B.2.6 Conditionals ..................................... 216
    B.2.7 Loops ............................................ 216
    B.2.8 Labels and Jumps ................................ 217
    B.2.9 Arrays ........................................... 217
    B.2.10 Dyadic Operators ................................ 218
    B.2.11 Prefix Operators ................................ 219
    B.2.12 Postfix Operators ................................ 220
    B.2.13 Numeric Constants ............................... 220
    B.2.14 Symbolic Constants .............................. 221

C The WIM3 Intermediate Code ............................... 223
  C.1 Introduction ........................................ 223
  C.2 Basic Blocks and Insns ............................... 223
  C.3 Insn Operand Types .................................. 224
  C.4 Insn OpCodes ....................................... 225

D P Tools Quick Reference ................................ 227
  D.1 Introduction ........................................ 227
  D.2 The Compiler ........................................ 227
    D.2.1 Introduction ..................................... 227
    D.2.2 Command Line Options ............................. 227
    D.2.3 Configuration File ................................ 229
  D.3 The Library Stub Compiler ............................. 230
  D.4 The C to P Translator ................................ 231

Bibliography ................................................ 233
Chapter 1

Introduction

1.1 Languages and Compilers

This is a thesis on programming language compiler optimisation; before embarking on it, it would be wise to define exactly what these terms mean to the author, and which aspects of them will be explored herein.

A programming language is a notation for specifying algorithms in a form acceptable to a computer. As such, a programming language has two main purposes: formalisation of the problem to be solved, and abstraction from machine-specific implementation details.

Languages can be classified, according to the formal model of computation they mimic most closely, in two main groups: imperative (based on the Turing or Minsky machine formal models [Tur36, Min67]) and functional (based on the various λ calculi [Chu41]). The two models are equivalent in “computing power”, i.e. anything computable by one is computable by the other; thus there is no clear distinction between the two language classes. Functional programs can be written in imperative languages, and most functional languages have imperative features.

According to the level of abstraction they implement, languages may be characterised as low-level (LLL) or high-level (HLL); the former reveal more of the machine’s hardware structure to the programmer, allowing more efficient (and also more error-prone, and less portable) code to be written, while the latter provide more hardware-independent facilities to make the programmer’s work easier. Low-level features have traditionally been associated with imperative languages, but this is not now, and has never been, necessarily the case.

The archetypal subclass of LLLs, collectively known as machine code, contains the imperative native tongues of computer hardware. Programs in other languages must be interpreted, instruction by instruction, by a machine code program; alternatively, they may be translated automatically into machine code and executed directly. A compiler is the program charged with the job of performing this translation.

\footnote{Compare, for example, two early languages: the low-level functional LISP 1.5 [MAE+62] and the imperative, yet comparatively high-level APL [Ive62].}
1.2 Compilation and Optimisation

As with almost all processes involving computers, compilation is easy if done sloppily. Indeed, for a long time it was thought that that was the only way to do it, and programs were still written in machine code, if efficiency was of the essence, well into the 1980s. The modern view is that the production of efficient code is mostly the compiler's responsibility, and that language designers should focus on portability, ease of use, and ease of compilation. This has accordingly shifted the attention of the academic community from LLLs, like LISP 1.5, C [KR88] and BCPL [RWS79], to HLLs, like Common Lisp [Ste90], ML [HMT89] and Modula-3 [CDG90].

Given a program that satisfies the criteria of syntactic validity, there are two kinds of approaches modern compilers use to avoid sloppiness in converting it to executable form.

Optimisation encompasses the methods by which the compiled program is turned into something that behaves identically (as far as its users are concerned), but runs faster and/or consumes fewer memory resources\(^2\). The other approach which, for want of a better name, we will call verification, involves checking the internal consistency of the program and ensuring that it contains no obvious blunders that will cause it to fail at run time.

Optimisation includes processes like constant propagation, loop invariant elimination, register allocation/assignment, and unused variable detection; verification includes type-checking and undefined variable detection. The two approaches are closely related, as they both rely on analyses that extract properties of the program: verification checks that these properties lie within specified acceptable bounds, while optimisation exploits them (to decide how to transform the program) and is constrained by them (since the user-visible behaviour of the program must be preserved).

1.3 Effect of Language Design on Optimisation

1.3.1 Low-level Languages

It was mentioned earlier that an objective in the design of modern HLLs is ease of compilation. It should therefore not be surprising that LLLs, although designed for writing efficient programs, are actually harder for compilers to optimise. Examples of this abound:

1. The ways in which LLL programs are more efficient are often highly machine dependent; ported to new machines, they will at best confuse the compiler, at worst failing to work\(^3\).

2. LLLs often allow the programmer to abuse looping and conditional constructs and wreak havoc with the flow of control. Many optimisation algorithms depend on loops having one entry and one exit; anything else will confuse and disable them\(^4\).

\(^2\)Please note that by "compiler optimisation" we mean the optimisations performed by compilers; although the optimisation of a compiler is important in determining what optimisations it can apply cost-effectively.

\(^3\)See, for example, entry 169 in HAKMEM [BGS72] for a cunning method of counting bits in a PDP-6 word that is slower than the straightforward approach on all modern processors.

\(^4\)For an infamous example of this, see the entry on Duff's Device in [Ray93].
3. LLLs do not enforce strict typing conventions; it is easy to convert values from one type to another, and the compiler (not to mention the programmer) can easily lose track of values this way, failing to notice when e.g. a number is called as if it was a procedure. This is carried to extremes by typeless languages, such as BCPL and LISP 1.5, which do not make any distinctions between values of different types.

4. LLLs typically place no restrictions on the ways data can be accessed and modified. An assignment in one part of the program can, through pointers and aliasing, invalidate the compiler’s notions of the state of seemingly unrelated program segments. This can have far-reaching consequences.

1.3.2 High-level Languages

While it is true that most modern languages try to constrain the excesses of programmers and to provide hints to their compilers regarding the program’s behaviour, there are inherent conflicts between some of the things people want to do and some of the things compilers are happiest with:

1. People want to be able to split programs into independent, reusable modules, with as little information about them as possible being visible to the outside world through their interfaces; compilers would like information about what goes on inside a module, when compiling programs that use it, so they can take the module’s effect on the state of the program into account.

2. People want to be able to encapsulate data structures and the procedures that operate on them in objects, so that they can treat them as extendible functional units and derive specialised versions of them by adding more data and procedures; compilers would prefer to keep code and data completely separated, so that procedure calls jump to constant addresses and can be resolved at compile-time.

3. The efficient implementation of some common data structures (such as packed bitvectors) requires knowledge of aspects of the computer’s hardware architecture. Porting such programs to machines with different characteristics can be tricky, but portable alternatives tend to be unwieldy and inefficient.

1.4 Scope and Outline of This Thesis

To summarise:

1. Optimisation (and verification) depend on information about the program being available, or easily obtainable.

2. This information is much harder to obtain for LLL programs.

3. HLLs try hard to make the information readily available, but some of their desirable features are, by their very nature, information-obscuring.

The main aim of our research has been to find and demonstrate better methods of extracting program properties to be used by optimisers. To this end, we have designed a typeless, low-level, modular language, incorporating all the compiler-hostile features mentioned above, and constructed a highly optimising compiler for it. The main body of this thesis is laid out as follows:
Chapter 2 describes the P language, traces its ancestry, and points out just how hard it is to optimise.

Chapter 3 gives an overview of the P compiler, its internal organisation and its more conventional features (including simple, well-known optimisations).

Chapter 4 describes a method of loop detection that can deal with irreducible flowgraphs; its results will be used in...

Chapter 5, which describes a new register allocation algorithm implemented in the P compiler. This algorithm does a better job than currently widely used methods and is designed to interact with the rest of the optimiser.

Chapter 6 presents a simple register assignment algorithm to go with the allocator of chapter 5.

Chapter 7 reviews the concepts of abstract interpretation that will be used extensively in the following chapters.

Chapter 8 introduces the compiler's value range analysis framework.

Chapter 9 describes the operation of the value range analysis engine in detail.

Chapter 10 describes the optimisations the compiler performs based on information gathered by the range analysis subsystem.

Chapter 11 introduces a method of extending the above optimisations across module boundaries; and finally,

Chapter 12 reviews our results and points out further work that can be done based on them.
Chapter 2

P: The Language

2.1 Introduction

P is a typeless imperative language with module facilities. Its philosophy owes much to BCPL, but its morphology is closest to that of C. Some features from ML and Modula-2 have also crept in. See figure 2.1 on the following page for a pictorial representation of the ancestry of P. This chapter describes some of the distinctive features of the language; see appendix B for a more complete definition.

2.2 Why Another Language?

Before describing the language itself, it is worth examining why it needs to exist at all. There are several reasons:

Why not? Optimisations are source language-independent. This is an important point: all the optimisations described in this thesis can be applied to code compiled from source written in any language. Applying them to a new language first does not restrict their scope or generality.

Ease of compilation. For reasons discussed in the next chapter, the compiler for this language was to be written from scratch. The design of a custom-made language can be such that the uninteresting phases of compilation (notably lexical analysis, parsing and translation) are easy to implement.

Lack of historical baggage. Compromises are made in the design of all widely used programming languages. Introduced to maintain backwards compatibility\(^1\), or forced by political considerations in the standardisation process, these compromises tend to complicate the compiler writer's job without increasing the research potential.

Resurrection of typeless programming. The optimisation algorithms that will concern us in this thesis are universally applicable, but one of the points we are trying to make is that they are especially relevant in typeless programming, allowing it to compete with higher-level approaches. There was a need for a typeless language with a clean, modern approach to modular programming; such a language did not exist.

---

\(^1\)The retention of old-style declarations in ISO C is an example of this.
Semantic subtleties. Programmers in traditional low-level languages like C or BCPL tend to rely on values *wrapping round* when they get too large to be represented in their datatype. For example, the ISO C standard [ISO90] requires that adding one to the largest representable unsigned integer should yield the value 0. Besides making programs dependent on the word size, this “feature” interferes with some of the processes presented in chapter 7. The semantics of a new implementation could have been changed (at the cost of non-conformance to the standard), but creating a new language is a safer and more honest approach.

### 2.3 P: A Typeless Language

#### 2.3.1 What Is a Type?

The concept of *datatype* in programming languages has evolved from the earliest days of FORTRAN I [B54, B56], whose types were simply attributes of variables specifying whether fixed- or floating-point operations should be applied to them, to the modern type theories underlying functional languages like ML [DM82, Dam85]. There are now many ways of viewing datatypes, but since the language we are concerned with is not supposed to have any, the following simple definition will do:

1. The datatype of a variable is a set containing all the admissible values for the variable.

2. The datatype of an expression is determined by the operators and the datatypes of the values involved, according to some well-defined rules.

For example, in the C language, if `a` and `b` are variables of type `double` (double precision floating point number), the expression `a + b` is also of type `double`, since the `+` operator yields `double` results given `double` operands.

Given this (admittedly very general) definition, “typeless” as applied to languages is something of a misnomer, since every value can be viewed as belonging to some set. Such languages normally dispense with all the complexity
typing entails by admitting only one datatype, which they can subsequently ignore, since every value belongs to it.

### 2.3.2 Typeless Languages

If we disregard machine codes and the groundbreaking but unimplemented Plankalkül [Zus76], the first typeless language was LISP; its single datatype was the 36-bit word, usually interpreted as a cons cell (two 15-bit pointers with two groups of 3 bits left over). The idea did not catch on, and the first imperative typeless language, BCPL, came 10 years later, largely as a reaction to the complexity of its ancestor CPL [BBH+63].

In P, as in BCPL, all values are n-bit words, where n is the size of a memory word on the host machine. This makes the early phases of compilation (parsing and translation; see chapter 3) much simpler, but is widely considered a Bad Thing. There are many reasons for this; at this point, two should be obvious:

- **Lack of type checking**: or, the “adding apples to oranges” effect. Typed language compilers can warn the programmer when the types of values are not appropriate for the operations they are involved in. For example, they will moan if the programmer attempts to invoke an integer as if it were a procedure, or add two addresses together, or multiply a table by a string.

  Typeless language compilers allow all such constructs, assuming that “the programmer knows best”. The problem is that programmers are far more likely to stumble upon such features unwittingly than to exploit them elegantly and without errors. We will see what we can do about this in section 10.2.7.

- **Portability**: The word size is a fundamental attribute of any particular implementation of BCPL-like languages. Since there are no mechanisms to provide higher-level type abstractions, most programs will rely on the word size in many ways, both obvious and subtle: the upper and lower bounds of representable values, the packing of bit fields and character strings, and in some cases the values of array offsets will vary across platforms with different word sizes, making the porting of programs that use such features a daunting task.

  P is better than most LLLs in this respect, because it only guarantees a word size of 32 bits, and disallows overflowing calculations. This is a simpler and more elegant approach than the abstraction used by some HLLs (e.g., the Word.i3 bitwise operations interface of Modula-3). It also helps with the analyses we wish to perform; see section 8.6.6 for a more extensive discussion of this issue.

### 2.4 P: An Imperative Language

#### 2.4.1 Imperative vs. Functional Programming

An important feature of all but the most trivial programs is that they contain symbolic identifiers. When the code is executed, values must be associated with these identifiers; the notional set of bindings providing these values is called the environment.

The fundamental difference between imperative and functional programs lies in the treatment of the environment: the former are allowed to modify it
(using some sort of assignment statement), whereas the latter are not. Thus there are no "variables" in functional programs, since the values of symbolic identifiers are not allowed to change. This is not as restrictive as it sounds, because not all of the program is executed in the same environment; although the value of any particular identifier is constant, the occurrences of similarly named identifiers in different scopes (and even instances of the same identifier in different invocations of the same procedure) may have different values.

The main advantage of the functional approach is referential transparency: at any point in a functional program, any subexpression may be replaced by the result of computing it, without affecting the outcome. This makes the analysis of programs much easier, and also simplifies the compiler's job since many constraints on the ordering of code are lifted.

The imperative approach treats symbolic identifiers as references to updatable cells. Programs consist of a series of operations that have side effects on the values held in these cells; the order of evaluation becomes significant. The semantic analysis of the program also becomes slightly harder, but not sufficiently so to explain why so little effort has been expended on it.

As has been mentioned in the Introduction (see section 1.1 on page 1), practically all\(^2\) languages display a built-in preference for one of these two programming styles, but it is almost always possible, within certain limits, to write functional programs in imperative languages\(^3\) and vice versa\(^4\).

### 2.4.2 Where Does P Stand?

Almost all imperative languages distinguish between two kinds of program building blocks: expressions, which may have side effects, but also have a value, and commands, which do not reduce to values and are executed purely for their side effects.

P makes no distinction between expressions and commands: everything notionally returns a value, even if the value is meaningless (such as the value of a \(^5\)for\(^6\) loop), never delivered to its destination (such as the value of a \(^5\)return\(^6\) expression\(^5\)), or subsequently ignored. This keeps the syntax simple and eliminates some of BCPL's syntactic uglinesses, such as the distinction between procedures and functions or between block commands and VALOF blocks (the difference between the former and latter members of each pair being whether they return a value or not)\(^6\). It also makes it easy to write simple programs in the functional style, which often makes them more readable: compare, for example, the four implementations of a factorial procedure in figure 2.2 on the next page.

However, P does not venture further on the functional path: it treats procedures as constants, its higher-order procedure facilities are limited to those available in traditional imperative languages, and its flow-of-control structures include sequences, loops and the infamous \[^{Dij68}\] "go to".

\(^2\)With the notable exception of logical languages of the Prolog family [CM81].

\(^3\)Except in languages like COBOL [ANS85] which do not allow recursion.

\(^4\)Except in thoroughbred functional languages like Miranda [Tur85] which have no imperative features at all.

\(^5\)This may need some explanation: the \(^5\)return\(^6\) expression takes one argument and returns it to the current procedure's caller. It thus terminates execution of the current procedure, and any value that would have been returned by the \(^5\)return\(^6\) to its enclosing expression is necessarily lost.

\(^6\)To avoid confusion with mathematical functions, we will use "procedure" as the generic term for callable entities, even though P procedures always return a value.
2.5. **P: A Modular Language**

### 2.5.1 Separate Compilation

Producing an executable version of a program is rarely as simple as invoking the compiler on a single file of source code. Besides the pre-compiled standard libraries (which are linked to the program by a linker, loader or in some cases [Ric96b, Fou93] by concatenation\(^7\)), the program itself is usually split into smaller *translation units* which are fed to the compiler independently. This approach has decisive advantages:

- Editing normally affects only a small number of translation units; the rest of the program need not be recompiled every time.
- Each invocation of the compiler needs fewer computing resources and runs faster, minimising the delay in the normal compile-link-run-debug cycle of program development.
- Different groups of people can work on different parts of the program at the same time, with only minimal effort spent in synchronisation and avoiding resource conflicts.

However, the translation units cannot be *entirely* independent: they have to share information in some way, if they are to form an integrated whole. The sharing mechanisms are a design feature of the language and typically involve common *header files*, which declare (but do not define) shared types.

---

\(^7\)The elegance of this approach is only possible due to, and outweighed by the necessity of, the explicit definition of the **GLOBAL** vector's layout in BCPL programs.
symbolic constants, variables and procedures (hereafter referred to collectively as entities). A header file is included by every translation unit that needs access to any of its contents.

### 2.5.2 Problems with Header Files

Header files allow the splitting of programs into separately compilable units, but they do little else. The following are the most common problems associated with them:

- Entities only used within a translation unit can be "hidden" from the rest of the program by not being mentioned in any header files, but if they need to be shared with even one more unit, they become potentially visible to any other unit that includes the appropriate header; more importantly, they become visible to the whole program at the linking stage. This means that identifiers used in interfaces must be unique; no two groups of translation units can use the same identifier to denote different resources shared within the groups, but invisible outside them.

- There is no distinction between units that define entities declared in a header, and units that use them: headers are always included textually, as if they were part of the unit itself. This means that
  1. Different units may define the same subsets of a header's contents. This can be identified as a compile-time error.
  2. Units may define some entities from a header, and leave the rest undefined. This will usually go undetected until link time, and will lead to run-time errors in languages that define variables implicitly.

- Translation units are allowed to make entities not mentioned in headers visible to the rest of the program. This is a common occurrence and largely defeats the whole purpose of header files.

### 2.5.3 Modules

All these difficulties can be circumnavigated using the concept of modules, first explored in Modula-2 [Wir85]. A module is a part of a program which is only allowed to interact with the rest through interfaces. Each module imports the interfaces declaring the entities it wishes to use, and exports interfaces declaring entities it defines for use by the rest of the program. The desirable consequences of this scheme are:

- There is a clear distinction between definition and use of a module's facilities.

- Anything not exported from a module is local to it by default, so accidental leakage of definitions is impossible.

- The whole of an interface must be defined by a single module, and this can be checked at compile-time.

- The names of entities declared by an interface must be prefixed by the interface's name in all references to them outside the defining module. This gets around the flat name space problem.

---

8Notably in the ISO C language, where procedures and file-scope variables are assumed to be globally visible unless explicitly declared static; before the ISO standard, C did not have file-scope variables and procedures at all!
Interfaces are written in a slightly restricted form of the programming language: they may include variable and procedure declarations (but not their definitions), and declarations or definitions of types and symbolic constants. They also need to provide for nested interface imports, since types and constants from other interfaces may be necessary to define the types of return values and arguments (and, in Modula-3, the default values of the latter) in procedure declarations.

### 2.5.4 The P Approach to Modules

P is, as far as we have been able to ascertain, the first typeless language with module facilities. It turns out that typelessness has some interesting side effects on modularity:

1. There is no need for nested interface imports. There are no types to be imported, and the declarations of procedures do not include information about the arguments they take (since that would be a form of typing), so there are no default values either.

2. Since P has no symbolic constants other than procedures and labels, and since the latter are rarely used outside their defining modules, it makes sense to only allow the exportation of variables. Procedures and labels can be exported by assignment to such variables; this is in accordance with the spirit of BCPL, whose globally visible procedures are assigned to variables in the GLOBAL vector.

3. Having got rid of everything but the variables, there is no point in keeping the syntax of the language either: interfaces devolve into lists of variable names, which can be placed (hopefully liberally interspersed with comments) in any arrangement the programmer finds aesthetically pleasing.

Thus, for example, the `main` interface, exported by all P programs to define their entry point, consists of a file containing the single word `main`, to declare the global variable `main`. `main` whose value is the procedure to call on starting the program.

It might seem that the module system we are left with is not much more advanced than C's header files. This is not the case: all the advantages of modular programming mentioned in section 2.5.3 still hold, and the export restrictions on non-variable entities do not lead to inefficiency, as most constant-valued variables can be optimised away (see chapter 10 for details).

It is also important to realise that module systems are not meant to isolate bits of the program from each other at run time: in any modular language, the state of a module can be modified by message passing from parts of the program that are not even aware of its existence. Other modules' exported variables (whose values may well be procedures) can be modified; the fact that in P global procedure addresses always reside in variables does not make this any more likely, or dangerous. In any case, the mechanisms described in chapter 10 can warn the programmer if this may happen inadvertently.
2.6 Other Features

2.6.1 Reserved Words

To identify control constructs, most languages use some of the names that would otherwise be available to the user as identifiers; for instance, C uses the identifier `int' to introduce declarations of integer objects, and `if' to introduce a conditional. In some languages (such as certain dialects of BA-
USIC [KK64]), user identifiers can be distinguished from the names significant
to the language purely by their context. In most cases, however, the use of
these reserved identifiers as names of user-defined objects is not allowed.

A problem arises when languages are extended so that words that used
to be available to the user become reserved. For example, C++ [Str91], an
extension of C, reserves the words `new' and `class', among others. This can
break many preexisting programs which would otherwise be valid in the ex-
tended language.

One way to get around the problem is to eliminate reserved words al-
together. APL [Ive62] uses a host of non-alphanumeric operators instead,
but this tends to make programs unreadable (besides requiring a specially
equipped terminal to write them). P does not have reserved words either, but
this is achieved by segregating user name space from that of the language's
keywords. The latter all start with a special character (`.'), not allowed in
user identifiers (except as a separator between interface names and the vari-
ables they export). Incidentally, this also makes the compiler's lexical analyser
tsimpler and faster.

2.6.2 Flow of Control Constructs

P provides all the usual looping and conditional constructs provided by modern
languages; in particular, every C statement has a P analogue. See appendix B
for a full description; at this point, it will be sufficient to note that

- The "dangling else" ambiguity [ASU86, p. 174] in a language's syntax can
  be avoided by using a more complicated grammar; but P follows BCPL by
  using different keywords to introduce if-then and if-then-else constructs.
  This is safer (as mismatched "else" clauses will cause parse errors) and
  helps to keep the parser simple too.

- There is no need for a ternary conditional operator, since the conditional
  statement returns a value; similarly, block statements may return val-
  ues using the .result keyword. See section 2.4.2 on page 8 for more
  information.

2.6.3 Labels and Procedures; Jumps and Calls

Labels are defined in P programs using the .place label-id expression. A label
may be referenced in the procedure that defines it using the .label label-id
expression. Because of this, label name space is distinct from that of variables;
labels and variables with the same name can coexist happily.

By analogy to labels, procedures are introduced by the .defun proc-id ex-
pression and referenced using .fun proc-id. They thus have their own name-
space too. Because procedures are local to their translation units and must
be assigned to variables in order to be exported, a common idiom found in P
programs is
which defines procedure func-name and assigns its address to a variable of the same name.\footnote{The distinction between the procedure itself (e.g. \texttt{.defun sitting\_on\_gate}), its name (e.g. \texttt{.fun the\_aged\_aged\_man}) and the name the procedure is called, or the name of the variable holding its address (e.g. \texttt{haddock\_eyes}) is unlikely to confuse Lewis Carroll fans [Car71]; but in P, procedures and their corresponding variables are usually given the same name, for the sake of the programmer's sanity.  

Control is transferred to labels using the \texttt{.goto} command (whose argument should normally evaluate to a label reference). Similarly, procedures are called using an explicit \texttt{call operator}, '@'. Besides maintaining the analogy, this also keeps the parser simple.

P follows C in not allowing procedure definitions to be nested. Everything that could be achieved by such nesting can (and should) be emulated by liberal use of module subdivisions. This also helps to simplify the intermediate code generator by obviating the need for a static chain or display in the stack frame.

### 2.6.4 Operators

P uses the same operators as the C language, with the following exceptions:

- Indexing is denoted by a `!' between the base and offset, instead of by enclosing the offset in square brackets (`[' ']'). See section 2.6.5 for more details on indexing in typeless languages.

- Because of the above use of `!', the logical negation operator is `~~'. This is more consistent with the convention of using doubled versions of bitwise operators as their logical counterparts.

- Assignment is denoted by the asymmetric operators `<-` and `->`, with the value to be assigned on the left- and right-hand side respectively. This bypasses a common criticism of C, namely that `=' can be mistaken for the equality operator.

- There are no compound assignment operators (`*=' etc).

In addition, it should be noted that, since there are distinct bitwise and logical operators, there is no need for a special Truthvalue evaluation mode: P, like C, evaluates expressions in Lmode (yielding locations in which values are to be stored), Rmode (yielding values) or Constant mode (for expressions whose value must be known at compile time).

### 2.6.5 Memory Access

Besides variables, P programs can store data in static tables and strings (whose contents are not updatable), as well as dynamic vectors (allocated, like variables, on entry to a block) and heap blocks (requested from the operating system using the facilities of the stdlib interface). These objects (which we will collectively call arrays) consist of a linear collection of cells, each containing one P word: individual cells can be accessed relative to the whole object using indexing operators (denoted by `!' in languages of the BCPL family).

Ideally, there should be a one-to-one mapping between array cells and the units of memory addressable by the hardware. This is unfortunately not the case, and typeless languages have been forced to compromise so as to conform to the hardware designer's expectations.
The problem is that computer memory is almost always byte addressed: the smallest addressable unit is the 8-bit byte. Machine words (usually the largest addressable units, and the ones corresponding to typeless language words) may consist of 2, 4, or 8 bytes, so the addresses of consecutive words differ by 2, 4 or 8.

Typeless language designers may choose (as in the case of BCPL) to insulate users from byte addressing, by presenting them with a word-addressed virtual machine. This is an elegant solution at the source code level: the binary indexing operator becomes commutative, the increment-by-one operator may be used to iterate along an array, and, as long as no assumptions are made about the capacity of a word, programs are easy to port to implementations with larger word sizes. However, the overhead of multiplications by the word/byte size ratio in the executable program can be substantial.

Alternatively, we may choose not to hide byte addressing from the programmers. This is the path grudgingly taken by MCPL and P, and it is much more efficient, but makes programs less portable unless the discrepancy is abstracted away using additional increment operators or symbolic word size constants. MCPL does the former, P the latter.

One may argue that this mess is not really the languages' fault: byte access is almost always a legacy of the past, implemented either to maintain compatibility with earlier processors, or as a means of speeding up accesses to character-sized objects. The latter reason is not valid anymore, since the extra logic needed on the processor-to-memory datapath slows down all memory accesses, and most operations on character data can be done in parallel (on word-sized chunks of it) anyway. Indeed, some modern processors (such as the DEC Alpha 21064 [Sit92]) have no byte access instructions (although they retain byte addressing). Other machines, such as the family of parallel vector supercomputers originating from the Cray-1 [Rus78], operating in markets where backwards compatibility is not much of an issue, would make ideal BCPL platforms as they have given up byte addressing and gone back to the word addressed stores of the earliest computers [WKT51].

Another problem is partly a result of feedback: as long as byte access instructions are available in the machine code, and since they make the manipulation of character data much simpler, typeless language designers have felt obliged to include source-level versions of them. BCPL and MCPL have the \texttt{\%m\ n} infix operator, which accesses the \texttt{m}th byte relative to the \texttt{n}th word\textsuperscript{10}. We believe that the byte access operator violates the spirit, if not the letter, of typelessness; P does not provide it.

Is the programmer forced to use obscure (and non-portable) bitwise operations to extract character data from machine words? No: just like everything else in the typeless world, characters should take up one whole word. Language designers are beginning to realise that the restriction of character data to 8-bit units is extremely short-sighted (see, for example, Amendment 1 to the ISO C standard, [ISO95]). Later in this thesis we will examine transparent and portable ways to eliminate the waste of 3 bytes per 8-bit ASCII character stored, without compromising the ability to store larger characters.

\section{Standard Library}

P does not yet have a standard library of its own, so it uses the facilities of the ISO C library. ISO C defines a number of header files declaring procedures for input/output, interaction with the operating system, common mathematical

\footnote{The reader may be interested to note that \% is commutative in MCPL, just like \texttt{!} is in BCPL.}
functions and string processing, as well as some system-dependent symbolic constants. P maps each of these header files to an interface of the same name: for example, procedure \texttt{printf}, declared in the C header file \texttt{<stdio.h>}, becomes \texttt{stdio.printf} of the \texttt{stdio} interface.

The C library exports the procedures themselves, whereas P expects their addresses to be stored in global variables; this is dealt with by a library stub compiler. See appendix D for details.

Use of the C library has the side-effect of introducing hard-coded byte strings into P, at least as long as the standard string processing routines are used, but the effort involved (and lack of research potential) in writing a standard library from scratch makes this an acceptable compromise.
Chapter 3

The P Compiler: Conventional Features

3.1 Introduction

The P compiler is a multi-pass optimising compiler written from scratch in the C language. It consists of about 20000 lines of code, split into 52 source files interfacing via 45 header files. Roughly one half of this code implements a traditional optimising compiler, and is described in this chapter. The rest of the code represents the optimisations described in the rest of the thesis.

When talking about the structure of a compiler, it is convenient to split it into three relatively independent parts. The front end is responsible for the source language-specific parts of compilation: lexical and syntax analysis and intermediate code generation. The back end is responsible for the target architecture-specific parts of compilation: turning the intermediate code into machine code, or assembly language (which will have to be fed to the system's assembler). The optimiser, or "middle" end, should ideally depend only on the intermediate code, and be portable to many source languages and target machines. Its input is qualitatively similar to its output, both consisting of intermediate code, but the output is hopefully optimised in some sense.

This chapter is divided into three parts, each describing the conventional features of the compiler's three "ends". But just as with the description of the language, before describing the compiler we should ask...

3.2 Why Another Compiler?

After all, there are freely available compilers, for widely used languages, to which the new optimisation methods could have been added, thus saving a lot of work. Let us look at the choices:

The GNU project compiler [Sta93] represents an immense effort targeted at making a free optimising compiler available for all commonly used processor architectures. It is written with portability in mind, and so uses a horrendously complicated, LISP-inspired intermediate representation; it is meant to be adaptable to different source languages, but in practice this involves major changes to little-documented parts of the system; and the whole thing (excluding the target machine descriptions and alternative front ends) consists of over 260,000 lines of code. It was felt
that the effort required to become sufficiently familiar with it to be able to extend and heavily modify it would not be worthwhile.

The lc c retargetable C compiler from Princeton [FH91] is a tenth the size of gcc but is emphatically a C compiler with no provision for alternative front ends. Its intermediate representation is more lightweight than gcc’s but still more complicated than necessary for a research project, the source code is sparsely commented, and the automatically generated back end includes a register allocator, which would be hard to replace.

The cintcode BCPL compiler [Ric96b] is small (4300 lines of BCPL), but uses a stack-based intermediate representation and generates code for a bytecode interpreter. It would require a major rewrite to be suitable for the work envisaged.

None of the above choices were satisfactory. Writing a new compiler, on the other hand, allowed us to tailor it to the job at hand, while becoming familiar with all parts of the system, commenting it heavily to maintain this familiarity, introducing diagnostic and statistic output routines, and designing a new language (described in the previous chapter) to go with it.

3.3 The Front End

3.3.1 Overview

Figure 3.1 is a pictorial representation of the flow of information (→) and control (⇒) within the compiler’s front end. Briefly, the front end consists of:

- A lexical analyser (or lexer), which scans input files turning collections of characters into tokens. The lexer can be used to process many sorts of input files; see section 3.3.2.
A function to solve the 8 Queens problem in P.

```lisp
.var count < 0, try < - .defun try [ld, row, rd]
   .test [row == 0xff]
       count++
   .else
       { .var poss < 0xff & ~(ld | row | rd);
         .while [poss]
           { .var p < poss & -poss;
             poss < poss - p;
             @try [(ld+p) << 1, (row+p), (rd+p) >> 1, 0];
           };
       };
```  

- A *program parser*, which builds a syntax tree out of tokens supplied by the lexer; see section 3.3.3 for details.

- An *intermediate code generator*, or *translator*, which turns the syntax tree into WIM3 intermediate code. The translator is described in section 3.3.4, and a full specification of the intermediate code can be found in appendix C.

- An *annotation parser*, which collects information about the behaviour of the program from *annotation files* processed by the lexer. Annotation files are discussed in chapter 11.

### 3.3.2 The Lexical Analyser

Lexical analysis is the process of turning sequences of characters from an input file into *tokens*, the parser's fundamental units of input. Lexers typically follow the design first described in [JPAR68], and employ automatic lexer generators\(^1\) which turn regular expression descriptions of the tokens into table-based scanners.

However, there are only four types of tokens in P programs, distinguishable from the first token character\(^2\), so it was simpler to write the lexer by hand, thus avoiding the table overhead\(^3\) and improving debugging and error detection facilities.

### 3.3.3 The Program Parser

The *grammar* of a programming language specifies how tokens may be combined to form programs. The meanings of programs are not defined by the grammar, but every meaningful program must obey the grammatical rules (although the converse is not usually the case).

The parser is a major component of the front end which:

\(^1\)Such as the standard Unix tool *lex*.

\(^2\)Names start with alphabetic characters; strings with a double quote ("), numbers (which include character constants) with a numeral or single quote ('), and reserved identifiers with a dot (.) or other symbol.

\(^3\)The hard-coded P lexer takes 12.9 CPU seconds to scan 4.6Mb of source code on an Alpha workstation; a P lexer written in *lex* takes 17.3 seconds, while the best *flex* (a faster implementation of *lex*) can do is 15.4 seconds.
1. Ensures that the input program is grammatically valid, and reports errors to the user if it is not.

2. Builds an abstract syntax tree in memory, which makes the hierarchical organisation of the program explicit.

Grammars are usually described in some variant of the BNF notation introduced in [Nau63]; a BNF specification of P's grammar can be found in appendix B. Such specifications can be turned into code automatically using parser generators\(^4\), but this is not necessary for P: its grammar is so simple that it can be handled by a top-down predictive parser, with no look-ahead or backtracking. Such parsers can be easily and efficiently written by hand using the Recursive Descent algorithm [Luc61], and this what our compiler uses\(^5\).

Once the syntax tree is generated, it is subjected to constant folding, which reduces expressions involving constants (e.g. turning $1+1$ into $2$). This is not strictly necessary, as the optimisations described in Chapter 10 subsume constant folding, but doing it at this point saves some unneeded processing.

The compiler can generate listings of the syntax tree in various formats, both human- and machine-readable. For example, figure 3.3 is an automatically generated image of the syntax tree for the procedure shown in figure 3.2 on the preceding page.

\(^4\) Most notably, the standard Unix tool yacc and its derivatives.

\(^5\) The P parser consists of just 500 lines of C; by contrast, the C parser used in the c2p C-to-P translation tool contains 2400 lines of C code automatically generated from a 700-line yacc input file.
3.3. THE FRONT END

3.3.4 The Translator

3.3.4.1 Introduction

The translator is responsible for producing intermediate code from the syntax tree and catching all the compile-time errors that could not be detected by the parser, such as scope and module use errors. It should be noted that this error checking cannot be deferred to the optimiser, which is a conceivably optional part of the compiler whose existence should not affect the success of a compilation run; the optimiser is restricted to issuing warnings, although in some cases these will point out things that will definitely go wrong at run time.

3.3.4.2 Intermediate Code

WIM3, the intermediate code generated by the translator, is a simple three-address code\footnote{Note that the traditional term "three-address code" should be interpreted as "three-operand code" in this case, since WIM3 primarily deals with virtual register operands, following the load-store paradigm of memory access.} fully described in appendix C. Its instructions correspond to the machine opcodes of modern RISC processors, and each of their operands has a type\footnote{Not to be confused with programming language types.}, so that procedures, registers, etc. can be described. Other notable features are:

**Virtual registers:** WIM3 provides an unlimited number of virtual registers (abbreviated to vregs hereafter; Vregs will be used to refer to the set of all vregs) in which variables and temporary results are stored.

**Virtual and machine registers have the same type:** In fact, vregs are a superset of the machine registers (sometimes abbreviated to hregs, and collectively referred to as Hregs). This allows the procedure call standard to be parametrised in terms of machine registers used or corrupted at procedure calls and returns, without interfering with optimisation (which can mostly deal with both as vregs, without making distinctions). Machine registers, global and local variables can be distinguished if necessary, since they use different ranges of the available register identifiers.

**Load-store architecture:** WIM3 implements memory access through load and store instructions only, with no complex addressing modes. This fits well with the three-operand scheme.

**Basic block organisation:** WIM3 code is emitted in sequences of instructions that contain no labels or jumps; this ensures that each such sequence (called a basic block) can only be entered at the first instruction and left at the last one. Basic blocks are fundamental units for many optimisation algorithms; identifying them while translation is taking place allows a flexible internal representation, so that blocks may be extended, copied and merged with minimal overhead.

3.3.4.3 Translation

Our view is that as little effort as possible should be expended on compiler components specific to the source language, and that it is the optimiser's job to optimise the program. Thus the P compiler's translator makes no effort to
produce efficient code: the syntax tree is converted into intermediate code in a straightforward, top-down, left-to-right recursive traversal.

First, the interfaces imported by the program are dealt with. Each interface file is read by the lexer and the appropriate variables are declared. In addition, the compiler looks for an annotation file for each interface examined. This will not concern us for now; section 11.7.2 has the details.

Subsequently, the translator examines the top-level declarations in the syntax tree. These need to be treated specially, since they declare global variables. All variables are assigned vregs, but globals are different from locals in many subtle ways that will become apparent as optimisation is discussed. In addition, this is the only level where procedure definitions are allowed in P; the translator will reject them if they occur anywhere else.

Below the top level, components of the tree may be translated in one of three ways, corresponding to the three expression evaluation modes of P: Rmode for expressions, Lmode for assignment destinations and operands of the address-of operator ("\`\`), and Constant mode for the components of static objects (strings and read-only tables).

As each subtree is translated the scopes of identifiers are checked, and
Figure 3.5: Flowgraph of the function in Figure 3.2. Nodes in the graph correspond to the basic blocks of Figure 3.4.

scope errors are signalled where necessary. If all is well, intermediate code is generated and placed in the current basic block. New basic blocks are started as necessary to deal with looping constructs, labels, jumps and procedure definitions.

Once the whole program has been processed, the translator reads the interface files of any modules exported by it and checks that all the exported variables have been declared.

As has been mentioned above, the translator was not written with code quality as the primary objective; its output is full of unnecessary regs, basic blocks and instructions. As it turns out, this does not matter much, because all these inefficiencies disappear during optimisation—most of them so early that they do not affect the optimiser’s efficiency.

The compiler can produce listings of the intermediate code for the program it is working on, in textual or graphical formats. The intermediate representation for the procedure of figure 3.2 is shown in figures 3.4 and 3.5; the latter is a representation of the program’s flowgraph, a directed graph containing one vertex for each basic block and one edge for each possible path between blocks. The flowgraph is important because many optimisation algorithms work by recursively traversing it; since its structure is implicit in the interconnections between basic blocks, it does not need to be stored separately, and
the term "flowgraph" will often be used informally to describe the collection of blocks and the links between them.

3.4 The Back End

3.4.1 Overview

The back end is responsible for turning the intermediate representation of the program into an object module that can be linked with other object modules and libraries to produce an executable program. This involves

1. Mapping vregs into hregs (and main memory locations, when the registers run out). Techniques for doing this are discussed in Chapters 5, 6 and 10.

2. Performing "peephole" optimisations, most of which are highly dependent on the target architecture. Thanks to its powerful main optimiser and the simplicity of the intended targets, the P compiler's peephole optimiser need not be sophisticated; the few facilities it provides are described in section 3.4.2.

3. Turning the intermediate code into assembly language for the target machine. This is done by the code generator, described in section 3.4.3.

4. Invoking the system's assembler to turn the assembly code into an object module.

3.4.2 Peephole Optimisation

This optimisation examines short, contiguous sequences of instructions (called peepholes) with the purpose of rewriting them to use fewer or faster instructions. On RISC machines, most instructions take the same amount of time\(^8\), and our peephole optimiser only deals with the following:

- **No-ops.** Instructions that have no effect (such as copying a register to itself, or subtracting zero from it) are discarded. The assembler will reorder instructions and reintroduce no-ops wherever necessary to ensure that instructions can be scheduled successfully by the CPU.

- **Macros.** Some WIM3 instructions would be rejected or turned into multi-instruction sequences by the assembler, if translated into assembly language in the straightforward way. This is because machine language places more restrictions on instruction operands than WIM3 does. These problems can often be bypassed by trivial instruction rewriting, such as exchanging the operands of commutative operations.

3.4.3 Code Generation

The translation of peephole-optimised WIM3 code into assembly language is a straightforward process; most WIM3 operations translate into equivalent assembly instructions. The parametrisation of registers used or corrupted by procedure calls is done at a higher level (in the compiler's central configuration file), so the code generator does not even have to deal with that. What it does have to do is:

\(^8\)Ignoring instruction interdependencies on superscalar architectures, which will not concern us for the moment.
• Calculate how much stack space is used by each procedure, set up and restore stack frames on procedure entry or exit, and allocate stack slots to spilt vregs.

• Declare procedure names and exported variables using assembler syntax so that linking with other object modules is possible.

• Deal with architectural quirks, such as the multiplication and division result registers of MIPS processors.

• Provide symbol table, file/line number and other miscellaneous debugging information in the assembly file; this information will be carried over to the object module and assist in debugging the program at source level, if necessary.

At the moment, the P compiler has a code generator for the MIPS R2000 and R3000 processors [KH92]; the resulting code will also run on later MIPS CPUs, without using their extra features. Porting to other RISC architectures should be easy, and an Alpha [Sit92] port is in preparation.

3.5 The Optimiser

3.5.1 Overview

The optimiser is the most interesting subsystem of the P compiler. It integrates traditional optimisation methods with a powerful range analysis engine that can assist the other methods and perform optimisations on its own; see figure 3.6 on the following page for a pictorial representation of the optimiser's components and their interaction.

The features of the optimiser that will be described here are:

• The dead code eliminator, which analyses the flowgraph, classifying the connections between basic blocks and identifying loops while removing blocks that cannot be reached; see section 3.5.2 for details.

• The liveness analyser, which produces sets of live variables for each basic block. This is described in section 3.5.3.

• The unused computation eliminator, which uses liveness information to eliminate useless code from within basic blocks and warns about the existence of uninitialised and unused vregs. See section 3.5.4.

3.5.2 Dead Code Elimination

3.5.2.1 Basic Blocks Revisited

Before discussing the elimination of dead code, it would be useful to look at the notion of a basic block (introduced in section 3.3.4) in greater detail.

We use the Dragon book's definition [ASU86, p. 528], which states that

A basic block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.
The "statements" are, in this case, WIM3 instructions (insns). Flow of control may reach a basic block from its predecessors (also basic blocks) or, in the case of a procedure entry block, from unspecified calling procedures (which do not count as predecessors). Thus, a block may have zero or more predecessors.

The block may be terminated by a return-from-procedure insn (leading back to the caller) or by a (possibly conditional) jump. Conditional jumps are implemented as a pair of insns, specifying the jump targets for true and false condition results respectively. This reflects the target machine code more accurately, although it necessitates some care to avoid violating the basic block definition. The two insns must be treated as one by the optimiser; in particular, no other insns should be inserted between them.

Jumps lead to the successors of the basic block that contains them. Identifying successors is easy if the target of the jump is a basic block; but the target may be a vreg, which will hopefully evaluate to a basic block address at run time, and whose value is not immediately available to the compiler. Rather than give up in such cases, the compiler makes a conservative estimate and regards all blocks which are referenced in non-jump insns (i.e. all blocks whose address could have ended up in a vreg at some point) as possible successors. Therefore, blocks may have zero (in the case of procedure return), one (for unconditional jumps), two (for conditional jumps) or more (for jumps to register) successors. It should be noted that procedure calls are not treated as jumps and do not terminate basic blocks.

3.5.2.2 Notation

From here on, it will often be convenient to use mathematical notation to refer to blocks, insns, their contents, predecessors and successors. Instead of introducing such notation wherever it is needed, thus running the risk of unnecessary duplication or large gaps between definitions and uses\(^9\), we will refer the reader to appendix C on page 223 for definitions of notation pertaining to the intermediate code. All notation used in this thesis is summarised in appendix A.

3.5.2.3 Dead Blocks

Each procedure has only one entrypoint, and jumps between procedures (other than calls) are not allowed. Hence it should be obvious that the flowgraph

\(^9\) Both of which will be recognised as bad practice by compiler engineers.
Figure 3.7: A function containing dead code.

```plaintext
.var test <- .defun test [ ]
{
.var a <- 1; // Block 2: Live
.goto .label one;
    a++; // Block 8: Dead
.place one;
    a++; // Block 4: Live
.goto .label three;
.place two;
    a++; // Block 5: Dead
.goto .label two;
.place three; // Block 6: Live
}
```

consists of a number of disconnected subgraphs, each containing one procedure entrypoint. Nodes that cannot be reached by following a succession of edges from procedure entries can be guaranteed to correspond to unreachable, or dead, basic blocks. In other words, the set \( \text{LiveB} \) of non-dead blocks can be defined recursively by

\[
\text{LiveB} = \{ b \mid b \in \text{EntryB} \lor (\exists c \in \text{LiveB} . b \in \text{Succ}_B[c]) \}
\]

and the set of dead blocks is simply

\[
\text{DeadB} = \text{Blocks} \setminus \text{LiveB}
\]

Dead blocks may be removed from the program without affecting its run-time behaviour.

As an example, the program of figure 3.7 compiles to intermediate code whose flowgraph is shown in figure 3.8 on the next page. Basic blocks 2, 4 and 6 correspond to reachable code; blocks 8 and 5 correspond to expressions that are unreachable because they are preceded by unconditional jumps; and blocks 9 and 10, also unreachable, are artifacts of translation, due to the intermediate code generator's naiveté. All but the first three will be pronounced dead by the optimiser.

3.5.2.4 The Algorithm

The well-known dead block elimination algorithm performs a recursive traversal of the flowgraph. Visited blocks are marked, and when the traversal is complete, all unmarked blocks are discarded. One may note the similarity of this method to mark-and-sweep garbage collection algorithms, and, just as with garbage collection, care needs to be taken to avoid getting caught in cycles in the graph (resulting from the compilation of loops); this is done by keeping track of which blocks have already been visited, and setting edge labels appropriately.

\[
\text{EDGE}_B(b, s) \in \{ \text{forward, cross, back} \}
\]

being the label of the edge leading from block \( b \) to block \( s \).

We have augmented the algorithm with some block merging and duplication features. We have also incorporated the generation of sets \( \text{REACH}_B[b] \) indicating the blocks that can be reached from any block \( b \) in the flowgraph's
spanning DAG (Directed Acyclic Graph); these will be useful later on (see section 4.4). The algorithm, in detail, is:

Algorithm 3.1 (Dead Block Elimination)

1. Place current block (b say) on the current path, and mark it as live.

2. For each successor s of b:
   
   (a) If $\text{Succ}_B(b) = \{s\}$, i.e. s is the sole successor of b, then
       
       (i) if $\text{Pred}_B(s) = \{b\}$, i.e. b is the sole predecessor of s, then replace
           the final jump in b with the whole code of s and set $\text{Succ}_B(b)$ to
           $\text{Succ}_B(s)$;
       
       (ii) otherwise, if s is a small block, replace the final jump in b with
            a copy of the code of s and set $\text{Succ}_B(b)$ to $\text{Succ}_B(s)$.
   
   (b) If s is on the current path, set $\text{Edge}_B(b, s) = \text{back}$.
   
   (c) If s is not on the path but has been visited earlier (i.e. is not cur-
       rently marked dead), the edge leading to it is a “cross” edge; add
       the current contents of $\text{Reach}_B(s)$ to the $\text{Reach}_B$ set of every node in
       the current path.
   
   (d) If s has not been visited before, the edge leading to it is a “forward”
       edge; add s to the $\text{Reach}_B$ set of every node on the current path, and
       apply the algorithm to s recursively.

3. Remove current node from path.

The blocks are all deemed dead initially, and the process is repeated until all
the initial values $b \in \text{Entry}_B$ are exhausted. The maximum size of a “small”
block is configurable, and we normally set it to 4 insns.

The following points should be noted:

1. There exist at least three different definitions of edge labels. Ours is
   in agreement with [FO76], rather than the four-way classification used
   in recent textbooks [Muc97, §7.2] or the traditional dominator-based
definition of the Dragon book [ASU86, p. 604]. Avoidance of the latter, in particular, ensures that all edges can be labelled, at the expense of failing to identify irreducible flowgraphs. We will not be concerned with the notion of reducibility, since all the algorithms presented in this thesis work on any flowgraph.

2. By considering labelled edges we can cover the flowgraph in different ways: the subset of the graph consisting of all nodes and just the forward edges is a spanning tree for the flowgraph, while adding the cross edges yields a spanning DAG. These will be useful in the recursive application of other optimisations.

3. The $\text{REACH}_B$ sets may be implemented as compressed bitvectors, using the same infrastructure used for liveness information, to be described in section 3.5.3.

### 3.5.3 Liveness Analysis

The fact that a vreg has a well-defined value at some point in the program does not necessarily mean that the value will be used subsequently. Conversely, the fact that a vreg is used does not mean that it has a well-defined value. Liveness analysis is a well-known way of determining whether the values contained in vregs are useful or usable in this sense. It is an aspect of dataflow analysis, which, in its most general form, consists of setting up equations describing properties of the program at any point in terms of the properties at related points and solving them, usually in an iterative process. Dataflow analysis was first studied systematically in [All70] and [Coc70], although the iterative style employed here had been in use since the early 1960s [VW63].

A vreg is said to be live at some point in a program when its value may be used in a computation later on in the program. A vreg is dead if it is not live. Liveness information is easy to compute, although computing it efficiently can be tricky. The set of vregs live just before any insn $i$ is executed can be defined recursively by

$$\text{LIVE}_V(i) = \left( \bigcup_{j \in \text{Succ}(i)} \text{LIVE}_V(j) \right) \setminus \text{DEF}_V(i) \cup \text{REF}_V(i)$$

(3.1)

In other words, the vregs live before $i$ are the vregs live after $i$, minus the ones defined in $i$ (whose old values, if any, are lost), plus the ones referenced by $i$ (whose old values are used). This assumes that insns reference their arguments before defining their results, which is invariably true.

The algorithm for applying this equation is straightforward; see [HU75] for a proof of its correctness.

**Algorithm 3.2 (Live Variable Detection)**

1. Initialise the live sets for all insns to the empty set.

2. While there are changes in any live set:

   (a) Traverse the flowgraph's spanning tree in postorder (i.e. each node's successors are processed before the node itself, and only forward edges are followed). For each node (block) do:

   (i) Apply equation 3.1 to all insns in the block, starting with the last one and proceeding to the start of the block.
Figur e 3.9: An uncompressed set (1); The same set, compressed from $2 + n$ words to just 4 (2). This figure uses 5-bit words for clarity; the compiler uses $b$ bits, where $b$ is the native word size.

Contrasting this analysis with dead code elimination, we may note that:

- Information propagates in different directions. Whereas the liveness of a basic block depends on that of its predecessors, the liveness of a vreg at an insn depends on its liveness in the insn’s successors. The former is a backward analysis, the latter a forward one. In the following chapters we will encounter many other analyses of both styles, and even one that can be applied in either direction.

- Whereas a single pass is sufficient to determine the liveness of basic blocks, the vreg liveness equation needs to be applied to each insn repeatedly for the results to converge to safe values. This is because unlike vreg liveness information, block liveness does not propagate around loops: a loop is reachable only if one of its blocks has a predecessor outside the loop.

The implementation is slightly more complicated than would be apparent from algorithm 3.2. The problem is that storing a full set of live variables for each insn is unrealistic: programs with tens of thousands of insns and vregs are not uncommon, and a bitvector representation of vreg sets (which allocates a single bit for every potential member of the set, and allows efficient parallel processing of set operations) will result in many megabytes of liveness information, while a representation that only stores the identifiers of live vregs will be too inefficient.

The P compiler’s liveness analyser deals with these problems by reducing the granularity of its operations: it computes reference and definition sets for whole basic blocks and derives the contents of live sets on entry to each block from them. It then provides facilities for the reconstruction of live sets within blocks given the sets at block boundaries.

Live sets are stored as compressed bitvectors. The compression method exploits the locality and interdependence of vreg use: most vregs are active for short, contiguous spans of the program, and vregs with similar identifiers are likely to be active in the same region\(^1\). The compiler uses a variant of run-length encoding which compresses long runs of ones or zeros (contiguously numbered vregs that are all live or all dead) without increasing the amount of storage needed for less uniform data\(^1\); the method is illustrated in figure 3.9.

A complete collection of hand-optimised set operations is provided, since the bitvector infrastructure is used to encode many other sorts of optimisation.

---

\(^1\) Since the translator allocates them as needed while scanning the code.

\(^1\) One-word sequences of ones or zeros are expanded to two words, but this is counterbalanced by the existence of longer sequences.
data; these operations map to combinations of logical OR (for union), AND (for intersection), and NOT (for set complement), so they can be applied to 32 or 64 set elements at a time on modern processors.

Finally, it should be noted that vregs corresponding to global variables (collectively referred to as Globals) are special: they are always live, regardless of their usage patterns, since liveness analysis is done at the intraprocedural level, and the fact that a global is not used later in the current procedure does not mean that it will not be used anywhere else. Vregs corresponding to variables whose address has been taken (collectively called Addrvars hereafter) are also special, since they may be referenced and defined indirectly; extra care must be taken when dealing with them, and this includes assuming that they are live all through the procedure that declares them.

3.5.4 Simple Applications of Liveness Information

Some of the simplest and most well known applications of liveness information are the elimination of unnecessary computations and the detection of uninitialised and unused variables.

Unnecessary computations (confusingly called "dead code" in some of the bibliography) are insns yielding dead vregs as results; they often arise as by-products of constant propagation and other transformations, and their detection was probably the motivation behind early dataflow analysis efforts. The detection of uninitialised variables is usually described in terms of "anomalous path expressions", introduced by [FO76], but it is equally valid to state that a non-global variable whose address is never taken is uninitialised if it is live on entry to the procedure that declares it.

Unused variables are usually the result of programming oversights, detectable at the parse tree level; it is more convenient, however, to check for them here. The following algorithm takes care of all of these:

Algorithm 3.3 (Unused/Uninitialised Variable Detection)

1. Initialise a set Used of vregs to the empty set.
2. Traverse flowgraph in postorder, and for each basic block do:
   (a) Recompute the set of live vregs at the end of the block (the "current" live set) from the stored live sets at the beginnings of its successors, by taking their union.
   (b) For each insn in the block, starting with the last one and moving backwards, do:
      (i) If the insn defines a vreg, and is not a call (i.e. has no invisible side effects), and the vreg is dead after the insn, then delete the insn and repeat this step with the previous insn.
      (ii) At this point, the current insn cannot be deleted: update the current live set to reflect the situation before the insn, add all the vregs accessed by the insn to the Used set, and go back to step 2(b)(i).
   (c) Replace the set of vregs live on entry to the block with the value of the current set.
   (d) If the block is a procedure entrypoint, issue warnings about all live vregs corresponding to non-global variables whose addresses are never taken: their being live at this point means that they are used before being defined in at least one of the possible paths through the flowgraph.
3. Issue warnings about vregs *not* present in the Used set that correspond to non-global variables; these variables are declared, but not used in reachable parts of the program.
Chapter 4

Loop Detection

4.1 Introduction

Traditionally, locating loops and determining the nesting level of code written in a structured language has been trivial: one simply counts the number of syntactic looping constructs currently active. However, in languages that allow unstructured code via “go to” statements the situation can become more involved: loops can interact in a variety of unusual and fascinating ways, including interlocking and sharing code. Nesting relationships can be ambiguous and nesting levels are far from obvious; in some cases, one might be forgiven for wondering whether the concept has any validity.

However, some of the algorithms we will be examining later in this thesis require nesting level values for the code they work on. In this chapter, we will try to devise a method for detecting loops and assigning nesting levels that produces reasonable results irrespective of the flowgraph’s complexity.

4.2 Overview of Related Work

One simple loop detection method defines a loop as the intermediate code resulting from the compilation of a source-level looping construct, such as a for or while statement. This widely used\(^1\) approach leaves a lot to be desired, since one can use goto statements to build loops that behave like for loops but are not detected as such. Better methods have to rely on the flowgraph, rather than the source code.

The classical definition of a loop [LM69] uses the notion of flowgraph node dominance: a node \(b_1\) dominates node \(b_2\) if every path from the entry node to \(b_2\) passes through \(b_1\). A back edge is then defined as an edge whose head dominates its tail, and a loop is the collection of nodes from which a back edge’s tail can be reached without passing through its head (the header of the loop). This approach has a number of disadvantages as far as we are concerned:

- It only works on reducible flowgraphs. Essentially, this means that only single-entry loops are handled; loops like the one pictured in figure 4.1(c) are not.

- It requires the dominance relation to be computed (typically by constructing a dominator tree); there exist efficient algorithms for this purpose [Tar74], but we would like to avoid it if possible.

\(^1\)By, for example, gcc.
Figure 4.1: Loop structures. Nodes are labelled with $b: l/n$, where $b$ the node number, $l$ the loop identifier and $n$ the nesting level. Edges are labelled with $f$ for forward, $c$ for cross and $b$ for back.

- It does not use the definition of a back edge used by the rest of our compiler (see section 3.5.2.4). The dominator-based definition will fail to label some edges in irreducible graphs; this is very inconvenient since other flowgraph-spanning algorithms rely on edge labels to proceed.

Attempts have been made recently to extend the classical definition by merging the flowgraph and dominator tree and using the augmented DJ graph to assist loop detection [SGL96]. This approach retains the last two disadvantages listed above, but it does detect irreducible loops, although it has to treat them as a special case. However, it is rather complex and tends to impose an arbitrary nesting hierarchy on symmetric constructs (e.g. the interlocking and Siamese loops of figure 4.1).

The third traditional loop detection method is based on *interval analy-
sis [Coc70]: the interval with header b is a flowgraph subset containing node b and all other nodes whose predecessors all belong to the interval; intervals containing nodes leading back to the header are deemed to be loops. This only detects one level of loop nesting, but since intervals are disjoint, we may create a graph with nodes representing the intervals and edges connecting intervals that are connected in the original graph, and repeat the process to detect outer loops. This method eliminates some of the ambiguities of the dominator methods; unfortunately, since the interval header dominates every node in the interval, it fails to detect multiple-entry loops. It also fails to detect loops that are not properly nested.

T1-T2 analysis [Ull73] attempts to detect loops by applying transformations that remove nodes from the flowgraph (while preserving edges) and looking for nodes with edges leading back to themselves. Unfortunately, the order in which the transformations are applied can affect both the number and the contents of loops identified: the flowgraph of figure 4.1(g) can be interpreted in three different ways under T1-T2 analysis.

Finally, the control dependence method [Wol92] relies on the notion of post-dominance, which is simply the dominance relation on an inverted flowgraph. A node b₁ is control-dependent on node b₂ if there is a path from b₂ to b₁ such that b₂ postdominates every node on the path except b₁, and either b₂ is b₁ or b₂ does not postdominate b₁. Cycles in the graph representing the control dependence relation correspond to loops in the flowgraph. The results are similar to those of the DJ graph method and the same criticisms apply, although this method carries less historical baggage and is simpler.

4.3 Loop Detection: Objectives

There is an obvious link between loops and flowgraph cycles, although the two are not identical. Since all cycles can be eliminated by removing the back edges from a flowgraph, a link between loop detection and back edges seems natural, and we would not like to ignore it; but we would like to avoid the much stronger assumption, made by traditional loop detectors, of exactly one loop per back edge.

Another point to keep in mind is that since we are not using the dominator-based definition, edge labels partly depend on the order in which the flowgraph is traversed. We would like our loop detector's results to be the same for any valid choice of back edges.

All the methods we have seen so far attempt to determine a nesting order for the loops they find: either two loops are entirely disjoint, or one is entirely nested within the other. This can lead to ambiguous or arbitrary results in some cases. For example, in the flowgraph depicted in figure 4.1(d), most methods detect loops \{3, 4, 5\} and \{2, 3, 4, 5\}, while the control dependence method replaces the first loop with \{2, 3, 4\}; both nest the former within the latter. We would like our algorithm to treat such cases consistently, without ignoring the symmetric nature of the graph but also without sidestepping the nesting question altogether.

Another question related to the above concerns alternate paths within the body of a loop. Is node 5 in figure 4.1(e) part of the same loop as nodes 3 and 4? Traditional loop detectors would answer affirmatively; we will see later that this is not necessarily a good idea.

Finally, we would like to do something useful about all irreducible loops (such as the one in figure 4.1(c)), without compromising our handling of simple looping constructs.
We believe that the algorithm presented in the next section addresses all of these points.

### 4.4 Detecting Loops and Assigning Nesting Levels

#### 4.4.1 Concepts

Our algorithm is similar to the dominator-based one, but has the following key differences:

1. It uses the flowgraph-based definition of back edges, so that irreducible graphs can be fully labelled.

2. It separates the notion of a loop from that of a “strongly connected” flowgraph component, or SCC [Tar72]. This is achieved by inserting a level of indirection between the identification of regions spanned by back edges and the distribution of nodes to loops.

An SCC is a flowgraph subset all of whose nodes can be reached from all the others. In our algorithm, the closest equivalent to this notion is the set \( \text{FROMTo}(b_1, b_2) \) containing all nodes on a path from \( b_1 \) to \( b_2 \) \textbf{which does not include any back edges}. We will restrict our attention to cases where \( b_1 \) and \( b_2 \) are respectively the head and tail of a back edge, so that \( \text{FROMTo}(b_1, b_2) \) corresponds closely to our intuitive notion of a loop: it is identical to the corresponding SCC if the flowgraph is reducible, but is easier to compute and deals more gracefully with irreducible graphs. The disadvantage is that no well-defined nesting hierarchy can be imposed on \( \text{FROMTo}(b_1, b_2) \) sets.

To get around this, our loop detection algorithm assigns a \textit{nesting level}, \( \text{NEST}_B(b) \), and a \textit{loop identifier}, \( \text{LOOPID}_B(b) \), to each flowgraph node (basic block) \( b \). It also assigns nesting levels, \( \text{NEST}_E(p, s) \), to every edge leading from a block \( p \) to a block \( s \). These values are non-negative integers; the nesting level indicates the number of loops the node (or edge) belongs to, while the loop identifier identifies the “innermost” of these loops.

The algorithm relies on the existence of a \( \text{REACH}_B(b) \) set for every block \( b \); this set contains \( b \) plus all the blocks that can be reached from \( b \) by following forward and cross edges only. \( \text{REACH}_B(b) \) sets can be computed during dead code elimination and edge labelling as described in section 3.5.2.4, or can be derived by exponentiation of an adjacency matrix for the flowgraph.

#### 4.4.2 The Algorithm

**Algorithm 4.1 (Loop Detection)**

1. Set \( \text{NEST}_B(b) \) and \( \text{LOOPID}_B(b) \) to 0 for every block \( b \).

2. Set \( \text{NEST}_E(b_1, b_2) \) to 0 for every edge from \( b_1 \) to \( b_2 \).

3. For every back edge leading from block \( b_1 \) to block \( b_2 \), do:

   (a) Initialise to 0 values an array \( \text{MAP}_{\text{loop}} \) indexed by loop identifier.

   (b) Compute \( \text{FROMTo}(b_1, b_2) = \{ b : b \in \text{REACH}_B(b_2) \land b_1 \in \text{REACH}_B(b) \} \).

      For every \( b \in \text{FROMTo}(b_1, b_2) \) do:

      (i) Increment \( \text{NEST}_B(b) \) by one.
(ii) For every \( s \in \text{Succ}_{B}(b) \cap \text{FromTo}(b_1, b_2) \), increment \( \text{Nest}_{E}(b, s) \) by one.

(iii) If \( \text{Map}_{\text{loop}}[\text{Loop}_{B}(b)] = 0 \), set it to a fresh (unused) loop identifier value.

(iv) Set \( \text{Loop}_{B}(b) \) to \( \text{Map}_{\text{loop}}[\text{Loop}_{B}(b)] \).

With the \( \text{REACH}_{B} \) sets implemented as bitvectors, so that determining reachability is a more-or-less constant time operation, the algorithm's running cost is proportional to the number of back edges times the average number of nodes between each back edge's head and tail.

4.4.3 Properties

The significance of the algorithm's results is not intuitively obvious, so we will examine some of their properties before looking at practical examples. As above, only \( \text{FromTo} \) sets corresponding to back edges are considered.

Nodes with loop identifiers of 0 do not belong to loops, because all nodes between the head and tail of any back edge have their initial 0 identifiers replaced at some point by step 3(b)(iv) of the algorithm.

Nodes belonging to disjoint \( \text{FromTo} \) sets must have different identifiers; also,

Nodes belonging to the same \( \text{FromTo} \) set may have different identifiers. Both these properties are enforced by steps 3(b)(iii)-3(b)(iv) which essentially partition \( \text{FromTo} \) sets into non-overlapping components whose members share a loop identifier.

Nodes with the same identifier are part of the same loop. To be precise, each will be executed at most once on each complete loop iteration, if we define "iteration" as the longest sequence of same-identifier nodes visited without going through a back edge. This is because the previous observation guarantees that nodes sharing a loop identifier belong to the same \( \text{FromTo} \) set(s), implying that any back edge reachable from one of them is also reachable from the others, and leads to a point "above" all of them in the flowgraph.

Nesting levels are only comparable at neighbouring nodes. If \( b_2 \in \text{Succ}_{B}(b_1) \) and the following inequalities hold:

\[
\text{Nest}_{B}(b_1) > \text{Nest}_{B}(b_2) \quad (4.1)
\]

\[
\text{Nest}_{E}(b_1, b_2) \geq \text{Nest}_{B}(b_2) \quad (4.2)
\]

then \( b_1 \) may be executed more than once on every iteration of the loop containing \( b_2 \); the loop containing \( b_1 \) is deemed to be nested within \( b_2 \)'s loop. This is because equation 4.2 ensures that \( b_1 \) belongs to all the \( \text{FromTo} \) sets that \( b_2 \) does, while equation 4.1 makes it necessary for \( b_1 \) to belong to more \( \text{FromTo} \) sets than \( b_2 \). Similar conditions hold on entering the loop; thus, with \( b_1, b_2 \) as before,

\[
\text{Nest}_{B}(b_1) < \text{Nest}_{B}(b_2)
\]

\[
\text{Nest}_{E}(b_1, b_2) \leq \text{Nest}_{B}(b_2)
\]

together imply that \( b_2 \)'s loop is nested inside the one containing \( b_1 \).
4.4.4 Examples

The results of the algorithm can best be understood by looking at some examples (figure 4.1 on page 34):

**Nested loops**, fig. 4.1(a): Two loops are identified, containing nodes \{2, 5\} and \{3, 4\}. According to the interpretation of loop nesting levels given above, the \{3, 4\} loop is nested inside the one containing \{2, 5\}. We do not consider the nodes of an inner loop to be part of the enclosing loop, but this is purely a matter of convention.

**Multi-exit loops**, fig. 4.1(b): Only one loop is found, containing the nodes \{2, 3, 4\}. Dominator algorithms will also detect this loop, even though it has two exits (from nodes 3 and 4).

**Multi-entry loops**, fig. 4.1(c): Again, only one loop is identified, containing nodes \{3, 4, 5\}. The loop can be entered at nodes 3 or 4, and would not have been detected by the classical approach, despite its similarity to the previous example. Note that the results are the same when the algorithm is applied to the same flowgraph with alternative edge labels (in parentheses).

**Interlocking loops**, fig. 4.1(d): Here we have three loops according to our algorithm, even though there are only two back edges. The loops are \{2\}, \{5\}, and \{3, 4\}. The \{3, 4\} loop is nested within both of the others; no statement can be made about the relative nesting of the \{2\} and \{5\} loops.

**Semi-nested loops**, fig. 4.1(e): Again, our algorithm assigns three loop identifiers; the loops contain nodes \{2, 6\}, \{3, 4\}, and \{5\}. Note that \{5\} has a nesting level of 1, lower than that of loop \{3, 4\}. This is reasonable since \{3, 4\} may be executed any number of times for each time 5 is, but not vice versa. \{3, 4\} should be regarded as being nested within both the other loops, although, as in the previous case, the relative nesting of those is not defined. The similarity to the Siamese loops example below should be noted.

**If-Then loop**, fig. 4.1(f): This is the flowgraph of a simple loop containing a conditional statement. Note the similarity to the previous loop graph, and the absence of a second back edge which explains why only one loop \{2, 3, 4, 5, 6\} is detected. Also note that not all nodes in the body of a loop have to be executed on every iteration: 3, 4, and 5 are each executed at most once for every time the back edge is traversed. Contrast with the previous example, where \{3, 4\} form an inner loop since they are executed at least once (and possibly many times) for every time the outer loop's back edge is traversed.

**Siamese loops**, fig. 4.1(g): Similar to the interlocking loops above, except that the two outer loops \{4\}, \{5\} are both on the same side of ("below") the inner loop \{2, 3\}; also similar to the semi-nested loops example, except that both back edges lead to the same node and both loops have exit arcs. The symmetry of the graph is reflected in the nesting levels and loop identifiers; our algorithm does not attempt to nest the outer loops within each other, nor does it regard them as a single loop.
Chapter 5

Register Allocation

5.1 Introduction

As we have already seen (see section 3.3.4.2), the intermediate code generated by the translator contains references to virtual registers (or vregs) in which values of variables and compiler-generated temporaries are stored. The translator may issue any number of vregs, but the target processor has only a few actual registers (or hregs). The compiler must therefore arrange for vregs to be mapped to hregs or memory locations at some point between translation and final code generation.

Register allocation is the process of deciding which vregs get to live in hregs; register assignment, that of deciding which particular hregs they get to live in. As we will see below, most compilers combine allocation and assignment into one phase. This has the undesirable side effect of blurring the distinction between the two terms, which are often used interchangeably. To avoid confusion, we will coin a new term, "register binding", to describe the whole process. The P compiler works on the problems one at a time, and the bulk of this chapter will be concerned with a description of the allocation method it adopts. Assignment will be dealt with in chapter 6.

The next section surveys the development of register binding methods, giving a brief description of each; since most traditional methods treat the binding problem as a whole, we include an overview of assignment methods here.

5.2 Overview of Related Work

5.2.1 Introduction

The binding of values to hregs has in the past been generally divided into a local phase, operating on non-branching code (equivalent to our basic blocks), and a global phase, taking care of variables and temporaries whose values are used in more than one block. The local phase, which invariably performs assignment as well as allocation, is usually considered the easiest of the two.

\footnote{Note that "global" in this case refers to the entire flowgraph of a given function (as opposed to the individual basic blocks it contains); global binding is not interprocedural binding.}
5.2.2 Local Binding

The local binding problem lies not so much in assigning a value to an hreg, as in deciding which value to displace when there are no more hregs available. An early attempt was based on Belady's MIN algorithm [Bel66], originally proposed as a way to manage virtual memory\(^2\). When it runs out of hregs, it reuses the one containing the value whose next use is farthest away, in terms of number of instructions. The rationale is that this hreg will thus be available for the longest possible time, and that the greater the distance to next use, the more likely that the hreg would have to be reused in the meantime anyway.

A popular alternative, whose results are almost as good, is to keep a usage count for each value, initialised to the total number of uses in the block and decremented whenever a use is encountered [Fre74]. When more hregs are needed, the one containing the value with the lowest usage count is reused.

Both the above methods assumed that all results are written to memory as they are generated, so there is no need to save the contents of hregs before they are reused. This "write-through" approach generates more stores than necessary; a better method was proposed in [Kim78]. According to this, results are kept in hregs (without being copied to memory) for as long as there are hregs available; when the hregs run out, the one whose next use is farthest away is reused, but its old contents are first saved to memory (an operation known as spilling)\(^3\). This represents a significant improvement, and forms the basis of the method used by the P compiler.

Finding a truly optimal solution to even the local binding problem is not easy. Taking the above method as a starting point, when an hreg is considered for reuse we do not always have to spill the value it contains: if it had been loaded from memory originally, and has only been referenced (without being updated) in the hreg, the copy in memory will still be up to date. Thus some choices of values to spill will cause the introduction of a memory store instruction, while others will not, and this should affect our choice.

An algorithm for solving this problem in the assignment of index registers was presented in [HKMW66], and extended to general-purpose registers for load-store architectures in [HFG89]. It involves building a DAG containing a vertex for each possible assignment at each instruction in the block. Edges link vertices that differ in the assignment of a single register, and are weighted by the cost of performing the assignment change; an optimal assignment is the shortest path connecting a vertex for the block's first instruction to a vertex for the last one. The cost of this process grows exponentially with respect to basic block size, so heuristics must be used to restrict the search, slightly defeating the purpose of the whole exercise.

5.2.3 Global Binding

5.2.3.1 Introduction

The fundamental difference between local and global binding is the existence of branches. In a global context we do not in general have information about the path taken to reach any given node in the flowgraph, so we must ensure that all binding decisions made in any node's immediate ancestors are consistent. This complicates the problem significantly.

\(^2\)There exists a hierarchy of caching problems: registers vs. memory, cache vs. main memory, main vs. virtual memory—all of which are only superficially similar. One should always be suspicious of applying algorithms developed for one to the others.

\(^3\)One may note the similarity of this and the previous approach, both in operation and in performance, to write-through and write-back strategies of cache memory management.
The simplest way to deal with global register binding is to not do it: starting with a local assignment method, we can ensure that all hreg-hosted values are saved before the end of each basic block, and assume that, on entry to a block, no values are in hregs. Needless to say, this does not result in very good code.

The next level of complexity consists of mapping the problem onto the local case. This can be done by somehow choosing a most commonly executed path through the flowgraph, treating it as one large block, assigning registers for it and inserting appropriate stores and loads at the boundaries of other blocks interfacing with it. However, the information needed to choose this path is usually unobtainable and often non-existent, since there may be more than one commonly executed path.

Other global binding methods can be subdivided into two classes: those that do something to improve the results of local binding, and those that deal with the whole problem without employing a local method.

### 5.2.3.2 Global Binding with Local Assignment

Like many other optimisations, register binding is of most benefit in the bodies of loops. A simple global method, proposed in [Mar62] and first used in the implementation of FORTRAN H [LM69], involves setting aside a few hregs to hold certain values for the entire duration of a loop; the remaining hregs (and any reserved hregs not used due to lack of values to put in them) can then be offered to the local assigner. This method has several problems: it limits the number of hregs that can be used globally, or may end up reserving too many hregs for values used globally but sparsely, causing the local assigner to "choke" and generate large numbers of memory references to short-lived, but very busy, temporaries.

In general, it is a better idea to do local assignment first, then try to improve the results, typically by removing loads and stores at the boundaries of basic blocks. Many ways of doing this have been suggested in the past, starting with one that performs allocation and assignment separately, concentrating on the movement of the local allocator's loads and stores out of the inner regions of loops [Bea74].

Possibly the best method of this sort is probabilistic, or demand-driven, register allocation [PF92, PF96]. This uses graph colouring techniques (see below) to perform assignment; allocation is done by calculating the probability of a value having an hreg after some instruction, given that it had one before. The non-conditional probability of the value already having an hreg when it is loaded is then the product of all the conditional probabilities for all paths going back from the load to the value's nearest definitions. An hreg is allocated to the value with highest probability; this makes the load redundant and causes other probabilities to change, so the whole process needs to be iterated again. The algorithm is wasteful of memory and computing resources (the authors report that their prototype implementation is about two orders of magnitude slower than the colouring allocator it replaces), but it does produce good results.

---

4This effect will be familiar to users of older C compilers, which honoured register storage class specifiers unquestioningly.
5.2.3.3 Global Binding Only: Register Colouring

Since the early 1980s, register binding research has focussed on the “register colouring” approach introduced by Chaitin [Cha81, Cha82]. This method is based on the observation that two values can share an hreg if they are not simultaneously live. Given liveness information (obtained as described in section 3.5.3), we may construct an interference graph whose vertices are values (or vreg live ranges) connected by edges to other values whose live ranges partially overlap with them. Finding a register assignment then boils down to using $N_h$ different colours (where $N_h$ is the number of available hregs) to colour the vertices so that no two neighbouring vertices have the same colour.

This is the well-known map colouring problem; unfortunately, while for two-dimensional maps 4 colours are sufficient [AH76], the topology of register interference graphs is more complicated. Given any value of $N_h$, we can always produce a graph that cannot be coloured. In any case, colouring is known to be NP-complete, so we cannot usually afford to find one even if it exists.

The problem is solved in the usual way—by applying heuristics. Values with a small number of conflicts (smaller than $N_h$) can always be coloured, so they (and all their edges) are removed from the graph and placed on a stack; when the graph is empty, the stack entries are popped one by one, put back in the graph and coloured. If, at some point during the decomposition of the graph, no vertex has fewer than $N_h$ conflicts, some vertex must be removed by other means. If this is not done, the process may fail to find a colouring even in trivial cases, such as the square graph of figure 5.1.

As the authors of [PF96] point out, register colouring is an assignment method. The “other means” mentioned above usually involve spilling (which performs the allocation function) and are not part of the colouring algorithm. Chaitin simply spilled the value with the highest number of conflicts for the whole of its lifetime, but a lot of effort has been expended in improving on this crude approach. It is not our intention to present all the increasingly arcane heuristics and variations on the algorithm that have been proposed; section 7 of [BCT94] is a comprehensive overview of the field. We will restrict ourselves to mentioning a few general trends:

- Better heuristics may be used to decide which values to spill [BCKT89, BCT94]; since no heuristic is perfect, multiple heuristics may be employed to build different spill choice sequences, the best of which is then adopted [BGG+89].
- Live range splitting splits long live ranges and colours the components individually. This can be achieved by using heuristics at the spilling stage [CH84, LH86], or by decomposing the interference graph [CD88, GSS89, GSO94] or the flowgraph [CK91] into a number of separately coloured components, which are then merged.

---

5 The relevance of the graph colouring paradigm to memory management problems had been recognised twenty years earlier, and its applicability to register binding was well-known in the 1970s, but Chaitin produced the first viable implementation; see [BCT94] for details.
Register coalescing is the opposite of live range splitting; it attempts to merge consecutive live ranges for different vregs holding the same value, eliminating some copy instructions. This is especially important for static single assignment (SSA) [CFR+89, CFR+91] intermediate codes, where no vreg is defined more than once. Improvements to Chaitin's coalescing heuristics can be found in [GA96, BCT94].

Finally, we should note that the "bin-packing" algorithm developed by Digital for their VAX compilers [ACJM82] is essentially a sophisticated (for its time) graph colouring assigner. It partitions the flowgraph at points where no vregs are live and processes the subcomponents individually, uses a complicated heuristic to order the assignments, and tries to retain the values of spilt vregs in hregs as long as possible.

5.3 Objectives

Having surveyed the existing register binding methods, we are now in a position to discuss their deficiencies.

- No form of local binding, however sophisticated, is going to be enough, because no interesting programs consist of pure non-branching code. We must either use an additional global binding method, or dispense with local binding altogether.

- Since we have to perform global binding too, we would prefer to let it deal with the whole problem. This is because local methods only use local information, but, unless their domain is clearly delineated from that of the global methods (a not altogether successful approach, as we have seen), the two will have to interact. Thus global binding will be affected by analyses that do not consider the program's global structure.

- However, the only cost-effective, purely global method available (register colouring) is too global in its original form: the locations of register conflicts do not figure in the algorithm. This leads to problems such as the square graph of figure 5.1 on the facing page. The improved colouring heuristics mentioned above address this problem with varying degrees of success, but they are complicated and costly; we would prefer to avoid them.

- While the square graph of figure 5.1 can be 2-coloured given a sufficiently powerful heuristic, the triangle graph cannot; yet there are program fragments represented by it where two registers are enough to hold all currently live values. See, for example, the program fragment of figure 5.2 on the next page. The problem can be addressed by range splitting, but it represents a fundamental failure of the colouring approach to allocation, and is due to the fact that colouring, as an assignment method, is out of its depth here. It strikes us as inelegant to attempt to patch it up using more ad hoc complications.

- Algorithmic complexity is also an issue. We have mentioned the exponential costs of optimal local allocation and graph colouring; we should also point out that the interference graph used in the latter is intrinsically an $O(n^2)$ datastructure, where $n$ is the number of virtual registers in the entire program. This is reasonable most of the time, but, as the authors of [MN92] report, in certain rare but not pathological cases $n$
Figure 5.2: A 2-colourable program fragment corresponding to the triangle graph of figure 5.1. \( n \) \( op \) \( m \) denotes any ALU operation on \( n \) and \( m \).

```c
.do
{
a <- \( op \) \( b \);
b <- \( c \) \( op \) \( a \);
c <- \( b \) \( op \) \( a \);
}
/* while */ [b];
```

can reach the thousands. This is too much, and requires complicated compression schemes, unless a local method is introduced to take care of most of the registers [Sta93]—leading to problems discussed above. It would be nice to have a method whose cost depends on the number of live vregs, rather than the total number.

The method employed by the P compiler addresses all of the above concerns. Its main features are:

- Its allocation and assignment phases are completely independent; other optimisation passes may be inserted between them.
- It is conceptually simple and uses no complicated heuristics.
- Its allocator is based on a local method, extended to take global information into account.
- It interfaces cleanly with the rest of the optimiser, making use of information provided by the abstract interpretation engine if it is available, and falling back on safe approximations if not.
- It considers all vregs as candidates for allocation—including those representing global variables, or variables whose address is taken.
- It moves values from hregs to memory according to the requirements of the surrounding code, without the need for range splitting heuristics.
- The size of its datastructures is proportional to the product of the number of blocks in the program and the number of vregs live at any point in it, rather than the square of the total vreg count.

## 5.4 Introduction to Register Allocation

The underlying principle of our register allocator is that at any insn \( i \) in the program where the number of live vregs, \( | \text{LIVE} [i] | \), is not greater than \( N_h \) (the number of hregs available to the allocator), all vregs can be accommodated in hregs. There may exist constraints forcing the assignment of hregs to vregs to change between instructions, but this should be dealt with by the assignment phase. Data memory accesses are more expensive than instruction memory accesses (which are more predictable and can benefit from hardware prefetching), and vastly more expensive than operations that can be performed inside the CPU; it is therefore worth introducing instructions to shuffle the registers around rather than spilling and reloading them, even if this causes a small increase in code size.
Thus, the allocation algorithm is only concerned with constraining the number of hreg-allocated vregs to a value smaller than, or equal to, $N_h$. This is done by:

- Checking that all arguments of an insn are in hregs just before the insn is executed, and reloading them if they are not.

- Spilling as many vregs as necessary before an insn (and any reloads it requires) to make room for the insn’s arguments and results. Vregs whose next use is farthest away are spilt first.

- Making sure that the program’s notions of which vregs are spilt at the boundaries of neighbouring basic blocks agree.

- Taking care of complications, such as variables referred to by their addresses, globals, function calls, entries and returns, etc.

Allocation is performed in three stages:

**Next use calculation.** An iterative algorithm calculates the average distances to next use of live variables at basic block boundaries.

**Spill convergence.** The compiler pretends that it is inserting spills and reloads as necessary, iterating until the sets of spilt vregs converge to stable values.

**Spill insertion.** A final pass inserts the spills and reloads simulated in the previous stage.

### 5.5 Distances to Next Use

#### 5.5.1 Non-Branching Code

In non-branching code, the *distance to the next use* of a vreg $v$ at an insn $i$, $\text{Dist}(v, i)$, is the number of insns between $i$ and the next insn that references $v$, as long as $v$’s value is not changed in the meantime. Note that

- The next reference may be in $i$ itself, in which case the distance is zero.

- If no such insn exists, either because $v$ is redefined before the next reference or because we run out of code, we assume that the distance to its next use is infinite. Thus $\text{Dist}(v, i)$ takes values from the set $\mathbb{N} \cup \{\infty\}$; operations on natural numbers may be extended in the usual way so that $n + \infty = \infty$, $n/0 = \infty$, etc. This will doubtlessly make pure mathematicians cringe, but it works well enough for our purposes.

Equivalently, we may define $\text{Dist}(v, i)$ in an inductive manner, remembering that in non-branching code insns have at most one successor:

1. If $i$ references $v$, the distance is 0.

2. Otherwise, if $i$ defines $v$ or $i$ has no successor, the distance is $\infty$.

3. Otherwise, it is one greater than the distance at $i$’s successor.
CHAPTER 5. REGISTER ALLOCATION

Now, by definition, a vreg is dead at an insn whenever there is no path leading from the insn to a reference of the vreg without passing through a definition. It should therefore be obvious that \( \text{Dist}(v, i) \) is \( \infty \) whenever \( v \) is dead at \( i \), and vice versa. So we can rewrite our inductive definition as

\[
\text{Dist}(v, i) = \begin{cases} 
\infty & \text{if } v \not\in \text{Live}_V(i) \\
0 & \text{if } v \in \text{Ref}_V(i) \\
1 + \text{Dist}(v, j) & \text{if } \text{Succ}_i(t) = |j|
\end{cases}
\]

(noting that \( \text{Ref}_V(i) \subseteq \text{Live}_V(i) \), for any \( i \).)

This describes a simple backwards analysis that can assign distance values to all vregs at every point in the code sequence, in a single pass from the end of the code to its entrypoint.

5.5.2 Code with Branches

Our objective is to generalise the above definition so that it places no restrictions on the number of insn successors. This may at first seem impossible: branching paths in the program may have different distances to the next use of any register, and there is no way to tell, prior to the branch, which of the paths will be taken. Our claim is that the average value of the distances on all possible paths is a reasonable guide for register allocation. We will now show how to obtain these average values and prove that the process converges.

The simplest approach would be to modify only the third line in the definition of \( \text{Dist} \) above:

\[
\text{Dist}(v, i) = \begin{cases} 
\infty & \text{if } v \not\in \text{Live}_V(i) \\
0 & \text{if } v \in \text{Ref}_V(i) \\
1 + \frac{\sum_{j \in \text{Succ}_i(t)} \text{Dist}(v, j)}{|\text{Succ}_i(t)|} & \text{otherwise}
\end{cases}
\]

This unfortunately has a problem with vregs that are not live in all the successors of an insn. In straight line code, if a live vreg is not referenced at an insn, we can be sure that it will be live at the insn’s successor. The introduction of multiple successors implies that the vreg may be dead (and therefore have an infinite distance value) at some of them; the infinite value will dominate the average calculation, so all vregs dying at some of the insn’s successors will end up with the same (infinite) distance value.

To get around this, we must replace infinity with a finite value, \( D_\infty \). This should be larger than all “legitimate” distance values (i.e. no distance value should be allowed to become greater than it) but small enough so that values averaged with it do not lose their ordering. \( D_\infty \) is a configurable constant in the P compiler, but there is no reason why it should not be set to something appropriate for each function (e.g. a value proportional to the longest path from the function’s entry to its exit).

The other problem is that we have been assuming that all branch destinations are equally likely. This is the best we can do for most branches, but there is a notable exception: loop exits are typically executed much less frequently than branches that return to the beginning of a loop. Since the P compiler has a means of assigning loop nesting values to basic blocks, we may define \( \text{Nest}_i(t) \) to be the nesting level of the block containing insn \( i \). Then \( \text{NSucc}_i(t) \), the set of successors of \( i \) that are at the same or a higher nesting level, can be defined as

\[
\text{NSucc}_i(t) = \{ j \mid j \in \text{Succ}_i(t) \land \text{Nest}_i(j) \geq \text{Nest}_i(i) \}
\]
and we may modify our definition of \( \text{DIST} \) as follows:

\[
\text{DIST}(v, i) = \begin{cases} 
D_\infty & \text{if } v \not\in \text{LIVE}_V(i) \\
0 & \text{if } v \in \text{REF}_V(i) \\
\min \left\{ D_\infty, 1 + \frac{\sum_{j \in \text{NSucc}(i)} \text{DIST}(v, j)}{|\text{NSucc}(i)|} \right\} & \text{otherwise}
\end{cases}
\] (5.1)

Equation 5.1 describes a backward analysis similar in structure to the liveness analysis presented in section 3.5.3 on page 29. An algorithm similar to 3.2 suggests itself:

**Algorithm 5.1 (Distance Value Computation)**

1. Initialise all distances to next use of live vregs to \( D_\infty \).
2. While there are changes to any distance value:
   
   (a) Traverse the flowgraph’s spanning tree recursively in postorder. For each node (block) do:

   (i) Apply equation 5.1 to all insns in the block, starting with the last one and proceeding backwards.

5.5.3 Convergence

Proving that the distance computation converges is easy: it is sufficient to note that

1. All initial values are finite and equal to \( D_\infty \).
2. During the iterative phase, no distance value can ever become negative, since the least value ever assigned is zero, all constants used are positive, and the set of non-negative integers is closed under all the operations used (assuming that division yields results rounded to the nearest integer).
3. Furthermore, the computation is monotone: no distance value can ever increase from one iteration to the next. This can be shown by an induction argument: distance values at insns with no successors are constant (and equal to \( D_\infty \)). Given that the new distance values for any other insn’s successors are no greater than they were in the previous iteration, the third clause of equation 5.1 ensures that the same holds for the insn in question.

The first two observations ensure that distance values are bounded; the third shows that they cannot oscillate. Therefore, they will have to settle down to stable values within a finite number of iterations.

5.5.4 Implementation Details

The distance calculation algorithm can be implemented exactly as described in the preceding sections, but a practical implementation can be made much more efficient. This section is a description of the implementation used in the P compiler.

Rather than holding distance value lists for each and every insn in the program, the compiler allocates two small hash tables for each basic block. These tables contain an entry for each vreg live at the beginning and end of the basic
block respectively. Dead vregs do not have entries, since their distance values may be inferred from their absence from the corresponding live sets. The tables have a fixed (configurable at run time) number of buckets, each holding an unsorted linked list of entries for vregs. This configuration is a good compromise between the compactness of linked lists and the constant-time accessibility of arrays, and is used extensively in the compiler.

As each basic block is examined (in a postorder traversal of the flowgraph), the following operations are performed:

1. The (uncompressed) set of live vregs at the end of the block is reconstructed from the corresponding compressed sets at the beginning of its successors. This is a standard operation involving the simultaneous decompression and union of compressed bitvectors. It is used frequently but is highly optimised and can be performed in a single pass through the vectors.

2. The distance values of live vregs from the block entry tables of successors that satisfy the nesting criterion are averaged and placed in the current block-exit table. If any of the new values are different from the old ones, a further iteration over the entire flowgraph will be required.

3. The new distance values are also put in the current block-entry table, with the size of the block added. This takes care of calculating the distance values at the start of the block for all vregs live on entry to the block but not referenced in it (and therefore also live on exit from it).

4. The block’s insns are processed backwards (starting with the last one). Vregs referenced by an insn have their block-entry distance table values set to the insn’s offset from the start of the block. This ensures that when the start of the block is reached, the block-entry table will contain distance values valid at that point.

Note that

- No effort is made to keep all distance values up to date while a block is being processed, as we are only interested in the values at block boundaries.

- As far as the P compiler is concerned, hregs are just a subclass of vregs. However, hregs by definition cannot be spilt, so there is no need to calculate distance values for them. The compiler ignores explicit references to hregs during the distance calculation phase.

Once the computation converges, the block-entry tables are discarded and the unordered hash buckets of each block-exit table are merged into lists and ordered by distance value (largest first) using insertion sort. These lists contain sufficient information for the spill introduction phase that follows; that phase will recreate similar lists for each insn i in the block being processed. We will refer to these ordered distance value lists as DIST_L[i], and use DIST_L[i][n] to refer to the nth vreg in the list for insn i.

5.5.5 Execution Cost

The memory requirements of the algorithm are essentially those of the per-block distance lists, and are proportional to the number of blocks times the average number of live vregs.
The algorithm’s first two steps each involve as many hash table updates as there are vregs live on exit from the block; each table has a number of entries equal to the number of registers that are live somewhere in the block, and the total space requirement for the tables is of the order of the product of this and the number of blocks. The third step involves as many hash table updates as there are vreg references in the block. We can assume the following:

1. Table lookups have unit cost.
2. The number of successors of a block is independent of code size.
3. All insns reference roughly the same number of vregs.

Under these assumptions, the cost of a single distance calculation pass is proportional to the number of basic blocks times the average number of live vregs, plus the number of insns in the program. The latter factor, which is of the same order as the cost of liveness analysis, dominates in practice, although it is negligible compared to the cost of other processes we will encounter.

### 5.5.6 Remarks

It should be obvious that the distance to next use is a good criterion for deciding which vregs to spill first in non-branching code: as was mentioned earlier, spilling the vreg whose next use is farthest away will free an hreg for the longest possible time. Better heuristics do exist, but, in terms of loads and stores avoided, they do not perform any better in short basic blocks and are only marginally better in long blocks using many vregs (coming within 1% of true optimal allocation, as opposed to 3% for the farthest-first approach; results from [HFG89]). It should also be noted that long basic blocks are rare in practice⁶, and, while some short blocks may be extended arbitrarily by loop unrolling, this does not increase the number of vregs they access and hardly affects spilling performance.

It may seem less obvious that averaging distance values at branches is the right thing to do. One way to think about this is to consider vregs accessed in all the branches of a conditional; assuming that all branches are equally likely (an assumption which, in the absence of profiling information and given that jumps out of loops may be treated specially, as in equation 5.1, is reasonable), the spilling priority of such vregs before the conditional should be the same as it would be if there had been no jump at all (since they will be accessed no matter which branch is taken), and their next reference had been somewhere close to where their actual alternative references are. As a way of determining the value of “somewhere close to”, the average of these alternative distances is good enough.

Vregs referenced at approximately the same distances from the conditional, but in fewer of its branches, should have a smaller chance of being reloaded since the reload is more likely to be wasted; their distance value should be higher, and our method takes care of this by using $D_{\infty}$ as the distance for branches with no references when computing the average.

When comparing vregs used in all paths, but very far away, with ones used in only a few paths but close to the jump, we should allow some of the latter to take precedence over some of the former (this would prevent, for example, a single reference to a vreg at the end of a single-exit function, where all paths

---

⁶Except in numerical analysis code, where floating-point throughput is usually the bottleneck and improved spilling performance is not that important.
converge, from monopolising an hreg). The extent to which this is done depends on the value of $D_\infty$, which can be adjusted as appropriate for the height and breadth (branching factor) of each function's flowgraph, although the use of a constant value in the compiler seems to produce quite acceptable results.

One complication that has not been mentioned so far concerns vregs corresponding to global variables, or ones whose address is taken at some point. These are deemed to be live everywhere, even if they are not referenced before the next definition or the end of the procedure. Our distance calculation equations fail to capture this case, but the algorithm and implementation presented above deal with it correctly by assuming that the distance value of a vreg not referenced later is $D_\infty$. Thus unreferenced globals have the same spilling priority as dead vregs, but will be saved to memory rather than discarded, because they are still live.

We should note that most modern architectures (such as the Alpha [Sit92], ARM [Jag97], MIPS [KH92], PA-RISC [Kan96], and Power [IBM90]) are based on the load/store paradigm: all instructions, except for loads and stores, expect their operands to be in registers. Thus, vregs must be loaded into a register before they are used, and the option of keeping rarely-used vregs in memory where they can be accessed directly at a slightly higher cost does not exist. This is why we only need to take the first next use of a vreg into account when deciding whether to spill it. At that point, there is no choice but to load the vreg anyway; any subsequent spilling decisions are completely independent.

However, one may envisage a generalised distance computation algorithm that also takes later references into account. This would give a measure of reference density, as well as distance, and might be useful in the allocation of registers for older architectures (such as the VAX [Leo87] and ia32\(^7\) [Int97a]), where some or all operands of instructions may be memory-resident. This method has not been covered by our research, but it seems certain that it will need a lot of fine-tuning if it is to be useful (or indeed usable) since it may violate our assumption about distance values not growing from one iteration to the next.

\(^7\)The architecture formerly known as ix86.
5.6 Introducing Spills

5.6.1 Non-Branching Code: Reloading Vregs

Just before the execution of any insn \( i \) in a sequence of non-branching code (such as a basic block), some of the live vregs are resident in hregs while the rest are kept in memory. Some of the latter may need to be reloaded, i.e., moved to hregs, if the insn references them; some of the former may need to be split, i.e., saved to memory, if there would be no hregs left for them after the reloads.

Let us denote the set of live vregs that are resident in memory just before insn \( i \) (the spill set of \( i \)) by \( \text{SPILL}_{\text{bef}}(i) \), and the corresponding set just after the insn by \( \text{SPILL}_{\text{aft}}(i) \). The spill set may be defined as:

\[
\text{SPILL}_{\text{bef}}(i) = (\text{SPILL}_{\text{aft}}(j) \setminus \text{LOAD}_V(i)) \cup \text{DISPLACE}_V(i)
\]

where \( j \) is the sole predecessor of \( i \) (i.e., \( \text{PRE}_{\text{D}}(i) = \{ j \} \)) and \( \text{LOAD}_V(i) \) and \( \text{DISPLACE}_V(i) \) are the sets of vregs that should be reloaded and split respectively between insns \( j \) and \( i \). (Needless to say, the spills should happen before the reloads.) Note that, while reloading always involves an actual load from memory, splitting does not necessarily require a store; nor are all stores introduced by the allocator necessarily spills. We will use \( \text{SAVE}_V(i) \) to refer to the set of vregs for which saves to memory are issued just before insn \( i \), regardless of whether they are then considered split or not.

Determining the contents of \( \text{LOAD}_V(i) \) is the easy part of the register allocator's job. It contains all the vregs referenced by \( i \) that are not in vregs after \( i \)'s predecessor:

\[
\text{LOAD}_V(i) = \text{REF}_V(i) \cap \text{SPILL}_{\text{aft}}(j)
\]

Note that in equation 5.2 we may use \( \text{REF}_V(i) \) instead of \( \text{LOAD}_V(i) \).

5.6.2 Non-Branching Code: Spilling Vregs

The number of live vregs assigned to hregs just before \( i \) is executed is simply \( |\text{LIVE}_V(i) \setminus \text{SPILL}_{\text{bef}}(i)| \). In determining the contents of \( \text{DISPLACE}_V(i) \) we must ensure that this number is smaller than, or equal to, the number \( N_h \) of hregs available for allocation. However, this is not enough: executing the insn may cause the number of live vregs to increase, and, since vregs have to start their lifetimes in an hreg, we will be left with more vregs than we can accommodate. We need to ensure that \( N_h \) is also not less than the number of CPU-resident vregs after the insn. In other words, if we define

\[
N_s(i, S) = N_h - |\text{LIVE}_V(i) \setminus S|
\]

we want

\[
N_s(i, S) \geq 0 \land N_s(k, S) \geq 0
\]

for \( S \) to be a suitable spill set for insn \( i \), whose successor is \( k \).

We must now decide which vregs to include in \( \text{DISPLACE}_V(i) \) to ensure that \( N_s(i, \text{SPILL}_{\text{bef}}(i)) \) remains non-negative; we can use the distance value lists computed previously to make this decision. To a first approximation, all we need to do is spill as many of the first few elements of \( \text{DIST}_L(i) \) that are not already split (i.e., not members of \( \text{SPILL}_{\text{aft}}(j) \)) as necessary to satisfy the condition of equation 5.3. There are two complications:
1. In some cases, insns with anomalous behaviour require that certain vregs be spilt before they are executed. We will refer to the set of such forced spills as $\text{Spill}_{\text{ext}}(i)$; its definition will be given in the next section.

2. Vregs corresponding to global variables, or variables whose address is taken at some point, are assumed to be live everywhere. As a result, any one of them may be live just before an insn that defines it without referencing it. This will give it a default distance value of $D_{\infty}$, making it a prime candidate for spilling, but spilling must not be allowed to happen, since it would leave the insn with no hreg in which to store its result.

Let us therefore define $\text{DIST}_S(i, n)$ as the set containing the first $n$ elements of $\text{DIST}_L(i)$ (some of which may already be spilt in $\text{Spill}_{\text{ext}}(j)$), except for those that are also members of $\text{DEF}_V(i)$:

$$\text{DIST}_S(i, n) = \{ v | k \leq n \wedge \text{DIST}_L(i)[k] = v \wedge \text{DEF}_V(i) \}$$

The set of vregs to displace is then the union of the set of forced spills and the smallest $\text{DIST}_S(i, n)$ that causes our condition to be satisfied:

$$\text{DISPLACE}_V(i) = \text{Spill}_{\text{ext}}(i) \cup \text{DIST}_S(i, \min \{ n | N_v(i, S(n)) \geq 0 \wedge N_v(k, S(n)) \geq 0 \})$$

where

$$S[n] = |\text{Spill}_{\text{ext}}(j) \setminus \text{LOAD}_V(i) \cup \text{Spill}_{\text{ext}}(i) \cup \text{DIST}_S(i, n)|$$

and $k$ is, as before, $i$'s successor.

### 5.6.3 Non-Branching Code: Special Cases

Lack of hregs in which to accommodate vregs is only one of the two factors determining how many vregs need to be spilt. The other is the nature of the insn to follow: some insns have far-reaching implications with regard to the flow of control in the program, and we are often forced to make conservative assumptions about which vregs they may corrupt. These insns are:

**Procedure Entry:** On entry to a procedure, any vregs that are not dead are regarded as spilt. This includes globals and locals whose addresses have been taken. The only exceptions are, of course, the callee-saved hregs and the hregs holding the procedure's arguments and return address, as defined by the procedure calling standard (PCS); being physical registers, they cannot be spilt. The translator will have introduced code to copy them into vregs immediately after the procedure entry insn; the vregs may then be spilt, moved around, or left undisturbed as necessary.

**Procedure Return:** All global variables are spilt before procedure return insns. This is necessary since register allocation information is not propagated into and out of procedure calls, at this point; any changes made to globals would be lost otherwise. There is no need to worry about any local variables, since they all fall out of scope at this point, and any references to their memory locations cease to be valid. The translator will have introduced code to restore the values of callee-saved hregs from the appropriate vregs, so they need not concern us either.

**Load from Memory:** Load insns originating in the source program may refer to any variable whose address could have been taken. This includes all
5.6. INTRODUCING SPILLS

global variables and the local variables whose address is taken somewhere in the procedure. The allocator must ensure that the copies of such variables held in memory are up to date, possibly by saving them:

\[ \text{SAVE}_{\text{load}}(i) = \begin{cases} \text{Globals} \cup \text{Addrvars} & \text{if } \text{Op}(i) = \text{load} \\ \{ \} & \text{otherwise} \end{cases} \quad (5.6) \]

These saves are not spills, as the copies of the vregs held in hregs will still be valid afterwards; \( \text{SAVE}_{\text{load}}(i) \) will form part of \( \text{SAVE}_V(i) \), but not \( \text{SPILL}_{\text{bef}}(i) \).

Procedure Call: Calls are assumed to corrupt hregs specified as caller-saved by the PCS, but this is handled in the liveness analysis code by making these hregs live for the duration of the call (between the \text{call} and back insns; see Appendix C) and the allocator need not be aware of it. Hence, and as long as we have no interprocedural information, calls behave similarly to stores: a procedure may reference or define, in any order, any global variable (or local whose address has been taken in some procedure further up the call stack). Calls are therefore treated like stores (whose treatment is a superset of that for loads); see below.

Store to Memory: Stores originating in the source program may affect the same vregs as loads and are treated in a similar way. The difference is that the possibly affected vregs need to be truly spilled, not just saved to memory, so that their next use will cause the possibly new value to be reloaded. We define

\[ \text{SAVE}_{\text{st-cl}}(i) = \begin{cases} \text{Globals} \cup \text{Addrvars} & \text{if } \text{Op}(i) \in \{\text{stor, call}\} \\ \{ \} & \text{otherwise} \end{cases} \quad (5.7) \]

The only complication has to do with vregs that are referenced by the insn explicitly through its arguments, but may also be defined by it implicitly by writing to the corresponding memory location. Such vregs must be saved before the insn (so that the copy in memory is valid even if not actually affected by the store), but not considered spilled until after the insn (so that the copy in the hreg is available for the insn to reference directly). Thus, while \( \text{SAVE}_{\text{st-cl}}(i) \) must form part of \( \text{SAVE}_V(i) \), only \( \text{SAVE}_{\text{st-cl}}(i) \setminus \text{REF}_V(i) \) should be added to \( \text{SPILL}_{\text{bef}}(i) \); the remaining elements should go into \( \text{SPILL}_{\text{aft}}(i) \).

We are now in a position to define the set \( \text{SPILL}_{\text{ext}}(i) \) mentioned above:

\[ \text{SPILL}_{\text{ext}}(i) = \begin{cases} \text{Globals} & \text{if } \text{Op}(i) = \text{ret}\text{n} \\ \text{SAVE}_{\text{st-cl}}(i) \setminus \text{REF}_V(i) & \text{otherwise} \end{cases} \]

5.6.4 Non-Branching Code: Wrapping it Up

All that remains now is to define the set \( \text{SAVE}_V(i) \) and the relationship between \( \text{SPILL}_{\text{bef}}(i) \) and \( \text{SPILL}_{\text{aft}}(i) \), for every insn \( i \).

As far as the second objective is concerned, we may note that in non-branching code vregs start and end their lifetimes inside the CPU: memory-resident vregs cannot be referenced or defined explicitly, so their liveness state
cannot change\(^8\). Thus any vreg contained in an insn’s spill set will also be contained in the set of split vregs after the insn, with the sole exception mentioned above for vregs referenced explicitly by insns that define them implicitly:

\[
\text{SPILL}_{\text{def}}(i) = \text{SPILL}_{\text{bef}}(i) \cup (\text{SAVE}_{\text{st-cl}}(i) \cap \text{REF}_V(i))
\]

We may define \(\text{SAVE}_V(i)\) in a silly but safe way by saving every vreg that has been added to the spill set, together with the ones specified in the discussion of special-case insns. However, the values of some vregs do not need to be saved, since we can prove that the values already in memory are up to date. To do this, we need to introduce another dataflow concept.

A live vreg is out of sync with its memory-resident copy if it is not split and the value held in its hreg is not the same as that held in memory. The set of such registers just after insn \(i\) may be determined by a forward analysis based on the following dataflow equation (also valid in code with branches):

\[
\text{ASYNC}_V(i) = \left( \bigcup_{j \in \text{Pred}(i)} \text{ASYNC}_V(j) \right) \setminus (\text{SPILL}_{\text{bef}}(i) \cup \text{SAVE}_V(i)) \cup \text{DEF}_V(i)
\]

(5.8)

i.e. memory copies of vregs are synchronised by spills and other saves to memory, and rendered out of date by definitions. The idea is that only unsynchronised vregs need saving, so

\[
\text{SAVE}_V(i) = (\text{DISPLACE}_V(i) \cup \text{SAVE}_{\text{load}}(i)) \setminus \text{ASYNC}_V(j)
\]

where, as before, \(j\) is \(i\)’s only predecessor.

This concludes our analysis of register allocation for non-branching code. It may sound complicated, but most of the complications arise in the handling of special cases, and all can be implemented quite efficiently, as will be seen later.

---

\(^8\)Vregs corresponding to global variables, or to variables whose address has been taken, may be referenced or defined indirectly, but we assume that such vregs are live everywhere anyway.
on. All the information that is not already available (i.e. everything except for live sets and distance lists) flows forwards through the code, as demonstrated in the dependency graph of figure 5.3 on the preceding page, so a single pass is sufficient to complete the allocation and no discussion of convergence is necessary.

5.6.5 Code with Branches

Two or more flowgraph paths merging at the beginning of a basic block will not necessarily carry the same spill sets. We must find a way to derive a consistent spill set for the beginning of the block, and adjust the spill sets of its predecessors to this new value by introducing more spills and reloads on the edges leading from the predecessors to the block in question.

To do this, we need to adjust our definitions of $\text{SPILL}_{\text{def}}(i)$ and $\text{LOAD}_{V}(i)$ to:

$$\text{SPILL}_{\text{def}}(i) = (\text{SPILL}_{\text{block}}(i) \setminus \text{LOAD}_{V}(i)) \cup \text{DISPLACE}_{V}(i)$$

$$\text{LOAD}_{V}(i) = \text{REF}_{V}(i) \cap \text{SPILL}_{\text{block}}(i)$$

where $\text{SPILL}_{\text{block}}(i)$ is the set of vregs spilt after the paths joining at $i$ (presumably the first insn in a block) have merged, but before any reloads or spills due to $i$ itself are introduced. How can we define $\text{SPILL}_{\text{block}}(i)$?

We can start by placing constraints on it. Let us look at the intersection and union of spill sets at the point where the paths merge, and the difference between them:

$$\text{SPILL}_{\text{and}}(i) = \bigcap_{j \in \text{PRED}(i)} \text{SPILL}_{\text{aft}}(j)$$

$$\text{SPILL}_{\text{or}}(i) = \bigcup_{j \in \text{PRED}(i)} \text{SPILL}_{\text{aft}}(j)$$

$$\text{SPILL}_{\text{pos}}(i) = \text{SPILL}_{\text{or}}(i) \setminus \text{SPILL}_{\text{and}}(i)$$

Clearly, $\text{SPILL}_{\text{block}}(i)$ should be a superset of the first and a subset of the second: it makes no sense to reload vregs spilt in every predecessor before we find out whether they will be needed, nor to spill vregs resident in hregs in all predecessors before we run out of hregs in the current block. We only need to decide which elements of $\text{SPILL}_{\text{pos}}(i)$ we should include.

There are many ways of doing this. We may attempt to form the spill set so that:

1. The total number of spill and reload insns introduced is minimal, or
2. The average number of insns introduced on any one path into the block is minimal, or
3. The least possible number of useless spills and reloads (ones that will be revoked soon due to references or lack of space respectively) is introduced.

The first approach would be tricky, not only because of the fact that not all vregs spilt need to be saved to memory, but also because some of the spills and reloads would be wasted, since their distance to next use would not be taken into account. The second approach is even worse: it attempts to ensure

\footnote{Note that for insns with only one predecessor, we have $\text{SPILL}_{\text{and}}(i) = \text{SPILL}_{\text{or}}(i)$ and $\text{SPILL}_{\text{pos}}(i)$ is empty, so that equation 5.9 reduces to 5.2, as expected.}
minimal average execution time for the entry into the block, but has all the
problems of the first method which largely defeat this attempt, while making
the code larger.

We have chosen to use the third approach, because it can be implemented
as a natural extension of our intra-block allocation method: we simply spill,
in the order they appear in DISTL(i), as many of the elements of SPILLpos(i) as
necessary to reduce the number of non-spilt vregs to Nh.

If we spill every element of SPILLpos(i) but still fail to achieve this objec-
tive, we should also spill some of the vregs not spilt on exit from any of the
block's predecessors; but this is handled by the existing mechanism described
in equation 5.5 and need not worry us here.

If, on the other hand, there are any hregs left over, vregs left in SPILLpos(i)
are reloaded in the order they appear in DISTL(i), in the hope that they won't
have to be spilt again before their use:

$$\text{SPILL}_{\text{block}}(i) = \text{SPILL}_{\text{and}}(i) \cup (\text{DIST}_5(i, \min(n | N_s(i), 
\text{SPILL}_{\text{and}}(i) \cup (\text{DIST}_5(i, n) \cap \text{SPILL}_{\text{pos}}(i))) \geq 0) \cap \text{SPILL}_{\text{pos}}(i))$$  (5.11)

The only problem with this strategy is that in some cases it may reload more
vregs than it should. If a region generating many new live ranges intervenes
between i and the insns that eventually use the reloaded vregs, some of the
vregs may need to be spilt again. In practice this effect has limited impact:

- Vregs with small distance values (i.e. the ones reloaded first) will either
die on being referenced (thus freeing an hreg) or stay alive, but
with increased distance values, reflecting the location of their next ref-
ence. This may make them better candidates for spilling than other
vregs reloaded at the beginning of the block but not yet used.

- Even if such a spill occurs, it only affects the program's performance on
paths needing reloads to adjust their spill sets to the value of SPILL_{block}(i).
For predecessors that do not spill the vreg concerned (and therefore do
not need to reload it before i), the only effect is to postpone the spill
from before i (where it would have been placed had we decided to spill,
rather than reload, the vreg on reaching i) to its actual location.

Having obtained a set of equations for computing spill sets throughout the
flowgraph, we may observe that it describes a computation similar to liveness
analysis turned upside down: spill sets depend on what comes before (rather
than after) them in the code, but clearly we still need some form of iterative
process to compute spills in loops. Such a process is described by the following
algorithm:

**Algorithm 5.2 (Register Allocation)**

1. Initialise the spill sets of all insns to Globals \(\cup\) Addrvars.
2. Initialise the out-of-sync sets of all insns to the empty set.
3. While there are changes to any spill or out-of-sync set:
   
   (a) Traverse the flowgraph's spanning tree recursively in preorder. For
each node (block) do:
      
      (i) Recompute the spill set on entry to the block according to equa-
tions 5.9 and 5.11.
      
      (ii) Recompute the spill and out-of-sync sets of all insns in the block
in turn according to equations 5.2 and 5.8, starting with the first
insn and proceeding forwards through the code.
5.6.6 Convergence

Our register allocation method has introduced two new iterative computations, performed simultaneously by algorithm 5.2. Their results are the sets of spilt vregs and the sets of vregs out of sync with their memory-resident copies; ensuring that these computations eventually terminate is essential.

Unfortunately, proving the convergence of spill set calculations is significantly harder than other dataflow analysis convergence proofs, such as that for liveness computation: the usual monotonicity argument cannot be employed, since spill sets approach their final values in decreasing oscillations. Thus, rather than attempting to present a proof, we will note an experimental result—that an exhaustively tested implementation of the algorithm as described above has yet to fail to converge—and present a method by which the computation may be terminated at any point, should the need arise, with sub-optimal but safe results.

An important observation is that a single pass of the algorithm always produces consistent allocations within basic blocks: there are always enough hregs to accommodate non-spilt vregs. Iteration is only required to optimise the additional spills and reloads at interfaces between basic blocks; a final, consistent solution may be reached after any iteration by failing to apply equation 5.11 to the first insn i of each block, instead setting \texttt{SPILL}\texttt{block}(i) to the old value of \texttt{SPILL}_{bef}(i). Synchronising the assignments is then simply a matter of spilling the contents of \texttt{SPILL}_{bef}(i) \backslash \texttt{SPILL}_{aft}(j), and reloading the contents of \texttt{SPILL}_{aft}(j) \backslash \texttt{SPILL}_{bef}(i), on the paths from every j ∈ \texttt{PRED}_{i} to i. This can be done by adding \textit{spill blocks} (discussed in the following section), just as if the computation had converged.

5.6.7 Implementation Details

The implementation of the P compiler’s register allocator is much simpler than the above discussion would lead one to expect. This is due partly to code reuse (the compiler’s standard bitvector manipulation routines are used to handle live, spill and synchronisation sets) and partly to storage reuse: most allocation information flows downwards through the flowgraph, and most quantities can be updated in place, so only a few of the entities shown in figure 5.3 on page 54 need to be explicitly implemented.

To allocate registers for a basic block, the compiler needs the following information:

1. The set of vregs which are spilt at the time the block is entered.
2. The set of hreg-resident vregs whose value has been modified since they were last saved to memory, at the time the block is entered.
3. The set of vregs live before every insn in the block.
4. The lists of distances to next use of vregs before every insn in the block.

The first two items are the ones being computed; the last two need to be recovered from the results of previous analyses (see sections 5.5.4 and 3.5.3) by scanning the block backwards according to the following algorithm:

\textbf{Algorithm 5.3}

1. Allocate arrays to hold the live sets and distance lists for every insn in the block (with an extra entry for the end of the block).
2. Initialise the last entry of the distances array to the value saved by the
distance computation phase (see the end of section 5.5.4).

3. Initialise the last entry of the liveness array from the live sets on entry
to the block's successors, using the same standard operation as in the
distance value algorithm (again, see section 5.5.4).

4. For every insn \( i \), starting with the last one and proceeding to the begin-
ing of the block, do the following:

   (a) Add any vrregs the insn references to the initially empty \( \text{DIST}_{L(i)} \) list.
   (b) Compute \( \text{LIVE}_{V(i)} \) from the live set of \( i \)'s successor (or the live set at
       the end of the block, if \( i \) is the last insn). This is another common
       operation implemented efficiently in the compiler's liveness analysis
       component.
   (c) Prepend a copy of the list of distances to next use after the insn
       (i.e. the list for \( i \)'s successor, or the end-of-block list) to \( \text{DIST}_{L(i)} \),
       discarding entries for vrregs that are dead or referenced by \( i \).

Note that:

- No new code is needed to handle live sets; All necessary facilities are pro-
  vided by the standard routines of the compiler's liveness analysis com-
  ponent.

- The actual distance values do not matter, and are not computed or stored
  in the lists. All that matters is the ordering of vrregs within the lists.

- Unlike the initial distance computation, which could leave entries for
dead vrregs in the lists and sort them out at the top of the block, this
phase produces accurate results: each distance list contains exactly one
entry for each vreg live at the corresponding point in the code.

The second stage, the one that actually does the spilling, proceeds forwards
through the block according to the following algorithm. It should be noted
that, since the process needs to be iterated until the spill sets converge, the
actual insns and spill adjustment blocks mentioned below are emitted in a
final pass, after convergence has been achieved; previous passes only pretend
that the insns have been issued.

**Algorithm 5.4**

1. Determine the spill and out-of-sync sets (bitvectors) on entry to the
   block. The latter is the result of ORing together the corresponding bitvec-
tors at the end of the block's predecessors; the creation of the former is
described later.

2. For each insn \( i \) in the block, starting with the first one and proceeding to
   the last one, perform the following:

   (a) Obtain the live set and distance list on entry to the insn \( \text{LIVE}_{V(i)} \)
       and \( \text{DIST}_{L(i)} \), respectively) from the arrays computed in the previous
       stage.
   (b) Decide which vrregs must be reloaded before the current insn: all
       split vrregs referenced by \( i \) are removed from the current spill set.
   (c) Make sure that any vrregs defined by the current insn are not in-
       cluded in \( \text{SPILL}_{\text{bef}}(i) \) (as per equation 5.4), and add them to the set
       of unsynchronised vrregs.
(d) Initialise the “current register count” to the number of elements in the live set (before or after the insn, depending on which is bigger; see equation 5.3). These count values may be computed during the generation of live sets at negligible extra cost.

(e) Handle special insns as described in section 5.6.3. This involves testing every vreg that could be affected (the elements of Globals and possibly Addrvars), inserting the appropriate saves before i and updating the spill and sync sets and the current register count.

(f) While the register count is greater than \(N_h\) (the number of hregs available to the allocator) repeat the following:

(i) Remove the first entry of the distance list: if the vreg in it is not already in the spill set, add it to the set (unless it is referenced directly and defined indirectly by the insn, as described in section 5.6.3) and decrement the register count. If it is also in the unsynchronised vregs set, remove it and insert a spill for it before \(i\).

(g) Issue reload insns according to the decisions made in step 2b above before \(i\) (but after the spills).

(h) Add the vregs spilt, but not marked as such in step 2(f)(i) above to the current spill set.

\[\text{3. Compress and save the current spill and sync sets (indicating the situation at the end of the block).}\]

To compute the spill set on entry to the block, we first compute the intersection and union of the spill sets on exit from the predecessors, initialise the spill set to the former, count the number of live and still unspilt vregs, then follow a procedure similar to that in 2(f)(i) above to reduce the count to \(N_h\).

Once this spill set is obtained, new basic blocks (called spill blocks) are inserted between the current block and its predecessors; these blocks contain spill and reload insns that adjust the predecessors’ spill sets to this new value.

### 5.6.8 Execution Cost

The memory requirements of the algorithm are similar to those of liveness analysis; each block needs two additional bitvectors to represent the spill and sync sets.

In a program requiring no spills, the cost of executing a single pass of the algorithm is dominated by bitvector operations and is again similar to that of liveness analysis. If spills are required, the depopulation of the distance list dominates, and the cost becomes proportional to the size of the program times the average number of spilt vregs. In practice, no more than three or four passes through the code are required.

### 5.6.9 Remarks

Now that the allocation method has been described, a number of important points need to be made:

- A common assumption among compiler writers is that small basic blocks and large numbers of jumps are a Bad Thing, so the insertion of extra blocks to adjust spill sets between the original ones may sound outrageous. It is not. In most cases, spill blocks are either their predecessor’s
sole successor, or their successor's sole predecessor, and can be merged with them. Even when this does not happen, the penalty is minimal since the extra jumps are unconditional (spill blocks have only one successor each). Unconditional jumps need not cause delays in execution, because their targets are known in advance and may be prefetched, pipelined and scheduled in the same way as non-branching code. Most modern architectures are capable of this (the processors formerly known as RS/6000, now part of the Power family [IBM90], have had a Branch Prediction Unit since the 1980s; see [Die94]).

- Spilling and reloading are performed lazily: no register is spilt before it has to be, and once it has been spilt, it is not reloaded until just before it is needed. This ensures that memory traffic is kept to a minimum, but it also means that reloads will invariably cause delays since the insn following a reload (which uses the reloaded vreg) has to wait for the value to arrive from memory.

It is our view that this is not a problem worth investigating. Instruction scheduling techniques, which reorder instructions to resolve such data dependencies and avoid resource conflicts, are well understood (the bibliography is extensive; a survey of the field can be found in [Kri90]).
5.7. AN EXAMPLE

Finding a "real" program that is short enough to be used as an example, but long enough to illustrate all the features of a register allocator, is impossible. Rather than give a series of examples, we have devised a program (figure 5.4 on the preceding page) that serves no useful purpose other than to show our allocator in action. We will use the same example in section 6.10 of the next chapter to demonstrate our assignment method.

Figure 5.5 on the following page contains a commented listing of the program's intermediate code after allocation. The compiler has been configured to use only 9 of the processor's 31 general purpose hregs for assignment, to make spills and reloads more common. A subset of the PCS is used: hregs 4–6 hold procedure arguments, 2–3 hold return values, 31 the return link, and 16 the callee-saved value which should be preserved across procedure calls. Let us examine the allocation in detail.

**Procedure entry:** Insns 1–5 of block 2 copy the PCS hregs into vregs, so that they can be spilt, reloaded, and reassigned. Assignment will hopefully reduce these copy insns to no-ops.

**Handling loads from memory:** Variable 'a', resident in vreg 106, has its address taken at insn 7 of block 2. Before the load of insn 10 is reached, the allocator inserts a spill of 'a' (insn 9) to ensure that the appropriate value is found in memory, if the load happens to reference 'a'.

However, the vreg-resident copy of a is not displaced, since its memory-resident value has not changed. Vreg 106 is used again by insn 12, which also updates it, leaving its memory copy out of sync.

For example, gcc includes a sophisticated scheduling phase [Tie89] that can be tuned according to the characteristics of the particular processor it is compiling for. It was not deemed worthwhile to implement such a system for the P compiler, since the assembler (to which the compiler's output is fed) makes an acceptable scheduling effort.

Besides, static software scheduling is on the way to obsolescence: modern OOO (Out Of Order execution) processors can do a better job at run time. For example, the latest members of the PA-RISC architecture family include a 56-entry Instruction Reorder Buffer whose contents are executed according to their dependencies, rather than their layout in memory [HP95]. Similar features are present in the latest Alpha [Kel96], MIPS [Mip96] and PowerPC processors.

- Finally, it should be noted that although the algorithm presented here is not optimal for local allocation, the extra memory accesses elided by true optimal allocators are usually stores (since they may prefer to spill a vreg whose next use is not farthest away, but which can be spilled at no cost because its current value is already in memory). On modern processors with write-back cache mechanisms, stores are less costly than loads, since they can be buffered while the CPU gets on with something else. Stores are also friendlier to scheduling mechanisms (both software and hardware), since they typically have no delay slots (their argument hregs may be reused immediately afterwards) and few forward dependencies. Thus the small number of additional memory accesses reflects an even smaller difference in performance between the two methods.
Figure 5.5: W1M3 code for the procedure of Figure 5.4, after register allocation.

Basic block 2
Predecessors: 2
Successors : 5 (f), 6 (f)
Code:
0: enter $0, $3, $3 // main.main entry
1: add R104, $16, $0
2: add R105, $31, $0
3: add R106, $4, $0
4: add R107, $6, $0
5: add R108, $6, $0
6: add R116, $0, $0 // i <= 0
7: add R118, R108, $29
8: add R106, R118, $0 // a <= .a
9: spill R106, $0, $0 // move before load
10: load R119, $0, R106
11: add R110, R119, $0 // x <= ta
12: add R120, R106, R110
13: add R106, R120, $0 // a <= a + x
14: spill R106, $0, // spill before store
15: stor R106, $0, R110 // !a <= a
16: unsp R106, $0, // unload a
17: xor R121, R106, R108
18: add R111, R121, $0 // y <= a - b
19: spill R105, $0, // spill link
20: orrr R122, R111, R107
21: add R112, R122, $0 // z <= y + b
22: jeq L5, R112, $0 // block 5 if z=0
23: jeq L5, R112, $0 // block 5 otherwise

Basic block 5
Predecessors: 2
Successors : 15 (s)
Code:
0: nop $0, $0, $0
1: add R116, R111, $0 // v <= y
2: add R136, $0, $0
3: add $6, R135, $0 // call arg: v
4: add $4, R136, $0 // call arg: "%\n"
5: spill R104, $0, // spill preserved
6: spill R115, $0, // spill v
7: spill R108, $0, // spill c
8: spill R107, $0, // spill b
9: spill R106, $0, // spill a
10: spill R105, $0, // spill x
11: unsp g75, $0, // reload printf
12: call g75, $2, $2
13: hack $0, $0, $0 // call printf
14: jeq L15, $0, $0 // go to block 15

Basic block 6
Predecessors: 2
Successors : 18 (s)
Code:
0: nop $0, $0, $0
1: add R116, R112, $0 // v <= z
3: jeq L18, $0, $0 // go to block 18

Basic block 7
Predecessors: 8
Successors : 16 (s)
Code:
0: nop $0, $0, $0
1: jeq L16, $0, $0 // go to block 16

Basic block 8
Predecessors: 17 18
Successors : 17 (s), 7 (f)
Code:
0: spill R106, $0, // spill a
1: unsp R105, $0, // reload a
2: spill R117, $0, $0 // go to block 8
3: sub R120, R115, R116

Basic block 9
Predecessors: 6
Successors : 8 (f)
Code:
0: spill R106, $0, // spill a
1: unsp R105, $0, // reload a
2: spill R117, $0, $0 // go to block 8

Static data:
1: mli R127, R114, R107
2: add R113, R127, $0 // u <= v * b
3: sub R120, R115, R116
Handling stores to memory: Insn 15 is a store that could (as far as the allocator knows) modify the value of `a`. Accordingly, the vreg- and memory-resident copies are synchronised again, with a spill at insn 14. This time, vreg 106 must be marked as no longer resident in the processor, since the store may have altered its memory value; when it is needed again, it will be reloaded. As it happens, it is needed immediately, and insn 16 reloads it.

Note that, although the spill happens at insn 14, the allocator only considers it to be in effect after insn 15, because the latter also references the vreg directly.

Real spills: Computing variables `y` and `z` pushes the number of live vregs over the available hregs limit, and a spill has to be introduced (insn 19). This is the first “real” spill we have encountered (as opposed to spills forced by memory accesses).

Calling sequence: Block 5 starts the .else branch of the .test expression, and contains a procedure call. Insns 3 and 4 set up the arguments for it; note that they are placed in hregs as specified by the PCS.

Calls entail a dramatic reduction in the number of available hregs, as argument, caller-saved, return link and return value hregs are corrupted; the long sequence of spills (insns 5–10) reflects this. The value we found in the preserved hreg on entry to this procedure is spilt, as are a number of our variables. Just before the call, the global variable (vreg 75) holding the procedure to call is reloaded.

Loop: Block 6 contains the initialising clause of the .for loop. Rather than jumping into the loop immediately, the code proceeds through block 18, the first example of a spill block. Blocks 6 and 18 will be merged later, saving a jump insn.

The body of the loop is block 8. We have arranged variables `u`, `v` and `w` so that they all conflict with each other, although no more than two of them are live at any time; we will see later that only two hregs will be needed to store them. The allocator only runs out of hregs on generating the loop exit condition, and has to spill `c` (vreg 108).

Unfortunately, this means that vreg 108 has to be reloaded (in block 17) before the jump to the top of the loop, to synchronise its spill status. Since `c` is not used in the loop, it would have been better to “lift” the spill out of the loop, thus saving two memory accesses per iteration.

This is the “laziness” discussed in section 5.6.9, and as we mentioned there, we feel it is more of a scheduling problem than an allocation one.

Spill synchronisation: The allocations at the ends of the two branches of the .test must be synchronised before entering block 11, the final part of the procedure; two new blocks take care of this. The loop has no need for the stdio.printf variable, so it is loaded in block 16, which then jumps to 11. The other branch has already loaded the procedure variable, but has spilt a number of other vregs due to the call it contains; these are reloaded in block 15.

Procedure return: Block 11 starts by computing an expression, two of whose operands need to be reloaded (insns 2 and 5). We then have another procedure call, preceded by the familiar spills to make room for corrupted
hregs (insns 10–11). Only two spills are needed, since all local variables\(^{10}\)
have died by now.

Insns 14–15 set up the return values of the procedure, as required by the
PCS (the second hreg is needed for compatibility with a future exception
mechanism). The callee-saved hreg (16) and return link (hreg 31) are
restored to the values they had on entry, and the procedure terminates.

\(^{10}\)Except ‘a’, which is live everywhere (since its address is taken) but has already been spilt.
Chapter 6

Register Assignment

6.1 Introduction

Once the intermediate code has undergone register allocation, we can guarantee that every live, non-spilt vreg can be assigned a real machine register at any point in the program. However, this assignment cannot be guaranteed to remain the same throughout the live range: even if complications such as corruption of, and explicit assignments to, hregs due to the procedure call standard are ignored, we still need to deal with the synchronisation of register assignments at the points where flowgraph paths merge or diverge. It is therefore obvious that the live range is not the appropriate unit for register assignment; we will introduce the notion of live range subcomponents called register spans as fundamental assignment units.

Having defined register spans, we will show that, although assignment could be performed by span colouring, this would not necessarily be a good idea. We will introduce a simple heuristic algorithm that produces effective assignments at a relatively low cost.

6.2 Span Limits

A register span is a section of a non-spilt component of a vreg's live range during which changing its hreg assignment is not only unnecessary, but detrimental to the program (in terms of extra insns needed to perform the assignment change).

The limits of spans can be found at

- Vreg births and deaths, since dead vregs do not require hregs. In particular, a vreg definition indicates the beginning of a span; if the defining insn also references the vreg, its previous span ends where the new one begins, and the new value may be assigned a different hreg.

- Spills and reloads, since spilt vregs do not require hregs either.

- Basic block boundaries, since a vreg may be assigned different hregs on the paths converging or diverging at the boundary, and these assignments will need to be interfaced by appropriate copy insns.

- Explicit assignments to hregs, which will displace any vregs resident in the hregs affected, and may cause a "ripple" effect causing any other non-spilt vreg to move.
In other words, if \( i \) is the first insn in its block or \( \text{DEF}_V(i) \cap \text{Hregs} \neq \{ \} \) (i.e., \( i \) defines or corrupts some hregs), the set of vregs for which a span starts at \( i \) is given by:

\[
\text{SPAN}_{\text{beg}}(i) = \text{LIVE}_V(i) \setminus \text{SPILL}_\text{beg}(i)
\]

Otherwise, it is given by

\[
\text{SPAN}_{\text{beg}}(i) = \left( \text{DEF}_V(i) \cap \bigcup_{j \in \text{SUCG}(i)} \text{LIVE}_V(j) \right) \cup \left( \text{SPILL}_\text{alt}(i) \setminus \text{SPILL}_\text{beg}(i) \right)
\]

The set of vregs ending a span at \( i \) is

\[
\text{SPAN}_{\text{end}}(i) = \text{LIVE}_V(i) \setminus \text{SPILL}_\text{alt}(i)
\]

if \( i \) is the last insn in its block or \( \text{DEF}_V(i) \cap \text{Hregs} \neq \{ \} \), and

\[
\text{SPAN}_{\text{end}}(i) = \left( \text{DEF}_V(i) \cap \text{LIVE}_V(i) \right) \cup \left( \text{SPILL}_\text{beg}(i) \setminus \text{SPILL}_\text{alt}(i) \right)
\]

otherwise.

Note that our spans are not the same as the vregs of SSA intermediate representations [CFR'89, CFR'91]: SSA values are born at definitions and block entries, but they do not die at spills, nor at explicit assignments to hregs.

### 6.3 Span Boundary Conditions

To make intelligent decisions about the assignment of hregs to spans, we need to know how the vreg's value is obtained at the beginning of the span and what happens to it at the span's end. We distinguish between the following possibilities, or **span boundary conditions**:

- **copy(\( v \))**: At the start of a span, this indicates that the new value is a copy of vreg \( v \). If this results in the death of \( v \), we should try to use its old hreg for the new vreg, to eliminate the copy insn. At the end of a span, this condition indicates that the vreg dies by being copied to another vreg.

- **cont**: Indicates the beginning or end of a span at the beginning or end of a basic block.

- **move(\( h \))**: At the end of a span, this condition indicates that an explicit assignment to hreg \( h \) is taking place in the insn concerned, so the vreg we are looking at may have to move elsewhere. A new span is generated for it, starting at the same point and with the same condition.

- **spill**: The vreg is being reloaded from, or spilt to, the stack at the start or end of a span respectively.

- **uspil**: As above, but the spill is necessary to ensure that memory and hreg-resident values of the vreg are synchronised (see section 5.6.3). This precludes some spill optimisations described in later chapters.

- **other**: The vreg is born (start of span) or dies (end of span) as a result of an arithmetic, logic or memory access insn.

We denote the boundary condition of vreg \( v \) beginning or ending a span at insn \( i \) by \( \text{BOUND}_{\text{beg}}(v,i) \) and \( \text{BOUND}_{\text{end}}(v,i) \) respectively. These functions may be defined in the obvious way based on the discussion above; they are undefined for vregs not starting or ending a span at the specified insn.
6.4 Definition of Spans

More formally, then, a span may be defined as a tuple,

\[ s = (v, i_b, i_e, f_b, f_e, \text{ref}, n) \]

where

- \( v \) is the vreg concerned,
- \( i_b \) is the insn at which the span starts,
- \( i_e \) is the insn ending the span (in the same block as \( i_b \)),
- \( f_b \) describes the origins of the vreg's value,
- \( f_e \) describes the vreg's condition at the end of the span,
- \( \text{ref} \) is the number of references to \( v \) during the span, and
- \( n \) is an integer whose purpose will be explained later (see section 10.3.4).

The following projection functions will be useful later. With \( s \) as above, we define:

\[
\text{PROJ}_{\text{vreg}}(s) = v \\
\text{PROJ}_{\text{ibeg}}(s) = i_b \\
\text{PROJ}_{\text{iend}}(s) = i_e \\
\text{PROJ}_{\text{fbegin}}(s) = f_b \\
\text{PROJ}_{\text{fend}}(s) = f_e \\
\text{PROJ}_{\text{refs}}(s) = \text{ref} \\
\text{PROJ}_{\text{size}}(s) = n
\]

We may now define the set \( \text{Spans} \) of all register spans:

\[
\text{Spans} = \{(v, i_b, i_e, B\text{OUND}_{\text{beg}}(v, i_b), B\text{OUND}_{\text{end}}(v, i_e), \text{ref}, n) : \\
\exists b \in \text{Blocks} . (i_b, i_e) \subseteq \text{INNS}(b) \} \land \\
v \in \text{SPAN}_{\text{beg}}(i_b) \land v \in \text{SPAN}_{\text{end}}(i_e) \land \\
(\forall i \in \text{INTERVAL}(i_b, i_e) . v \not\in \text{SPAN}_{\text{end}}(i)) \land \\
\text{ref} = |\{i | i \in \text{INTERVAL}[i_b, i_e] \land v \in \text{REF}_v(i)\}|
\]

where \( \text{INTERVAL}(i, j) \) is the set of all insns between (but not including) \( i \) and \( j \), assuming that \( i \) and \( j \) are in the same basic block:

\[
\begin{align*}
&j \not\in \text{Succ}(i) \Rightarrow \text{Succ}(i) \subseteq \text{INTERVAL}(i, j) \\
&\forall k \in \text{INTERVAL}(i, j) . (j \not\in \text{Succ}(k) \Rightarrow \text{Succ}(k) \subseteq \text{INTERVAL}(i, j))
\end{align*}
\]

The number of references during the span, \( \text{ref} \), and \( n \), which is intended to hold a measure of the “size” of \( v \)'s value, are not needed during register assignment; section 10.3.4.4 reintroduces them, but we can ignore them for the rest of this chapter.
6.5 Identifying Spans: Implementation

Spans are identified during the last pass of the allocation algorithm (i.e. the pass that inserts spill and reload insns). The tuples are stored in small hash tables, one for each basic block. The tables are indexed by vreg identifier; in the entry for a vreg, spans are linked in a list ordered by initial insn (the "vreg-specific" list). Each table's entries also form a linked list of spans (the "block-specific" list) ordered by initial insn; see figure 6.1 for an illustration of this.

Identifying spans and building the table are low-cost operations, since most of the checks they need to perform are also required by the allocator. The operations to be added are as follows:

Algorithm 6.1 (Span Identification)

1. **Initialisation:**
   Allocate an array of pointers to spans (one for each vreg in the program), an "active" bitvector, and a "last span" pointer.

2. **On entry to a basic block:**

**Figure 6.2:** Basic block whose span table is shown in figure 6.1.

0: nop $0, $0, $0 ; no-op
1: add RC, RA, $0 ; A is copied to C and dies here
2: add RB, RC, RB ; B dies and is reborn
3: spill RC, $0, $0 ; C is spilled
4: add RA, RB, RB ; B dies, A is born
Figure 6.3: The span table of figure 6.1 under construction, on entry to the block: Hash table entries and incomplete spans have been generated for all vregs live at this point (A and B). The scratch array and last span pointer (right) will be discarded when the end of the block is reached.

(a) Clear the active bitvector.
(b) Allocate a hash table for the block.
(c) Generate span tuples for vregs that are live and unspilt on entry to the block, using the liveness and spill bitvectors to find them. The initial boundary condition is \texttt{cont}.
(d) Add the tuples to the hash table as the first entries of the corresponding vreg-specific lists.
(e) Update the appropriate array entries to point to the tuples, and set the corresponding bits in the active bitvector.
(f) Link all new entries together in the block-specific list; update the last span pointer to point to the last one.

3. \textbf{On adding a live vreg to the spill set}:
   Locate its current tuple through the array and complete it (adding the terminating insn and \texttt{spill} or \texttt{uspil} condition).

4. \textbf{On reloading a vreg}:
   (a) Generate a new span tuple for the vreg, with initial condition \texttt{spill}.
   (b) If the corresponding array entry is empty, create an entry for the vreg in the hash table and put the tuple there.
   (c) Otherwise, link the new tuple to the one pointed to by the corresponding array entry (thus adding it to the appropriate vreg-specific list) and update the array entry to point to the new tuple.
(d) Set the corresponding bit in the active bitvector.
(e) Link the new tuple to the one pointed to by the last span pointer (thus adding it to the block-specific list), and update the pointer.

5. To handle a vreg reference:
   (a) If the vreg is still live after the referring insn, do nothing.
   (b) Otherwise, terminate its current span in the same way as for split vregs, except that the final condition will be copy or other (determined by examining the insn).

6. To handle a vreg definition:
   Definitions are handled after references, so most vregs concerned will be dead at this point.
   (a) If it is a global vreg definition, the vreg will still be live: terminate its current span as above.
   (b) If it is an explicit hreg definition: terminate the spans of, and generate new spans for, all currently live, non-split, non-hard vregs, using the move boundary condition.
   (c) Start a new span for the defined vreg in the same way as for reloads, except for the initial condition which will be other or copy.

7. At the end of a basic block:
(a) Terminate the block-specific span list.
(b) Terminate the spans of all live, unspilt vregs.
(c) Clear the current span array. Clearing every entry could be a costly operation if there are many vregs, but we need only clear entries corresponding to vregs present in the active bitvector, since other vregs have been dead throughout the block and their array entries have been left undisturbed.

8. Discard the current span array and active bitvector.

Note that, at the cost of one array of size equal to the total number of vregs, we have managed to eliminate all searches through the vreg-specific lists without sacrificing the ordering of these lists, which will prove useful during the register assignment process. The cost of maintaining the auxiliary array is small, thanks to the use of a bitvector to track vregs having spans in the current block. Figures 6.3 and 6.4 illustrate two steps in the construction of the span table of figure 6.1 on page 68.

6.6 Assignment: Choice of Algorithms

Having formed the span tables, we wish to assign an hreg to each span so that no span is left without an hreg and no two spans that are simultaneously active share an hreg. Of course, there are many assignments that are valid in that they satisfy these conditions; however, some of them have a higher run-time cost than others, in terms of additional copy insns needed to shuffle vregs around if their assignment changes during their lifetime. We would like to reduce this run-time cost while keeping the compile-time cost (i.e. the work done by the compiler) within reasonable bounds. There are three main directions from which this problem may be approached:

Register Colouring. In the previous chapter's introductory sections, we identified one of the failings of colouring as an allocation method to be the fact that it is really an assignment method, with no built-in way of handling spilling. Having taken care of spilling, we may now use colouring for its intended purpose.

Unfortunately, this is not as good an idea as it may seem. The elimination of copy insns by clever reassignment of hregs is another aspect of the binding problem not tackled by colouring in its pure form, and in our case this is particularly significant, since the assignment unit (span, rather than live range) is so much smaller than in ordinary allocators, and the number of span pairs that need to be interfaced is correspondingly larger.

Copy propagation is the removal of copy insns that transfer a value from one vreg to another, killing the original in the process. Colouring allocators implement it by actually merging the live ranges concerned, and considering the aggregate range as a single vreg, to be spilt or not as a whole. This "all or nothing" approach is suboptimal, but sufficient for colouring allocators; in our case, it is not even guaranteed to produce correct code, since we rely on the ability of spans to be assigned independently so that assignments can be synchronised at block boundaries, and so that vregs can move out of the way if the hreg they occupy is needed by the PCS.
Even if this problem is ignored, we need to decide whether to apply colouring locally or globally. In a global (inter-block) context, there are many more spans than vregs, and management of the at best unwieldy conflict matrix becomes unrealistic. In a local context, we would still need to meddle with constraints at block limits to avoid arbitrary assignment changes from one block to the next.

**Optimisation.** In this case, we are referring to the mathematical meaning of the term: it does not take much effort to recognise the assignment problem as that of minimising a cost function (related to the number of extra copy insns) subject to constraints (no two spans should have the same hreg at the same point). Adapting one of the standard optimisation methods, such as simulated annealing \([KGV83]\) or a genetic algorithm (GA) \([Hol75]\), would not be very hard either. GAs are particularly well suited to this problem:

- They do not rely on the good mathematical behaviour of the underlying functions.
- Their basic assumption (i.e. that globally "good" chromosomes are mostly built out of locally "good" genes) is reflected in the assignment problem (a good assignment for one small region of a program is likely to be part of a good global assignment).
- However, they are flexible enough to be able to recover from pathological situations where the above does not hold.
- The longer they are allowed to run, the better the results will be. Thus they provide a user-adjustable compile/run-time cost ratio.

In terms of execution cost, however, they are little better than a brute force approach and we would prefer to avoid them if any other half decent method suggests itself.

**The Empirical Approach.** Starting with the hints already obvious in the program (e.g. if a vreg is copied to an hreg explicitly, that hreg might be a good assignment choice for the vreg) we may try to propagate information through the flowgraph and see how many of the really bad solutions we can eliminate. If we then assign the remaining vregs more or less at random, the resulting solution may be acceptable. At worst, we can use the result of this process as input to a GA to accelerate convergence.

We decided to try the latter approach first. As it turns out, the results are good enough to render further processing unnecessary.

Assignment is performed in two stages: the first propagates information throughout the flowgraph, suggesting assignment candidates for each span, and the second selects a candidate for each span, taking care to honour the constraints placed by the code and other assignments.

### 6.7 Candidate Propagation

#### 6.7.1 The Algorithm

This phase is primarily concerned with the propagation of copies of values originating at, or ending up in, explicitly referenced hregs. Each span is given a set of "candidate" hregs; these will be the assigner's first choice when the
allocation is done, but do not preclude the use of some other hreg if they are unavailable.

The current implementation allows candidate sets of up to two elements (discarding further additions), and climbs through the flowgraph repeatedly in both directions (i.e. in both preorder and postorder) until the candidate sets stabilise. The small maximum set size may sound restrictive, but it does have a number of advantages:

**It’s good enough.** Explicit references to, and definitions of, hregs are not very common: they only occur where the PCS is invoked, i.e. on entry to, exit from, and calls to procedures. It takes such an occurrence combined with a chain of copy conditions to propagate a candidate, and long chains of pure copies with no arithmetic, logic or memory accesses are rare.

**Faster propagation.** It is much easier to maintain and update a small, constant number of candidates than a set with variable numbers of elements.

**Enforced convergence.** Once a candidate has been added, it may not be removed, so the sets can only grow or remain the same; the upper limit on the number of elements ensures that they will eventually stop growing.

**No need for priorities.** Candidates originating close to the span in question, and thus likelier to be advantageous choices for assignment than ones that have been propagated through a number of copies, are added to the set first. Since the set has a constant size, less likely candidates are only included if there are not enough likely ones. This gives a crude but sufficient ordering of candidates without the need for hreg preference values.

**Algorithm 6.2 (Candidate Propagation)**

1. Until there are no more changes to the assignment candidate sets, traverse the flowgraph, first in preorder, then in postorder, and for each block do:

   (a) While there are changes in the candidate sets of this block’s spans, repeat the following for each span:

      (i) If it is a span for an explicitly referenced hreg, add itself as its only assignment candidate and continue with the next span.

      (ii) Choose candidates according to the vreg’s condition at the beginning of the span:

          **copy(v):** If the vreg v being copied into this span is an hreg, the hreg is added as a candidate for this span. If it is an ordinary vreg, the span from which the value is being copied is located and one of its candidates (if it has any) is added to this span.

          **move:** Not all vregs have to change their assignments when an hreg is defined, so any candidates found in the current vreg’s previous span (the one interrupted by the hreg definition) are copied to this one.

          **cont:** The current vreg’s last spans from each of the predecessors of the current block are located, and as many of their candidates as will fit are copied to this span.

      (iii) Now choose candidates according to the vreg’s condition at the end of the span:
**copy(v):** If v is an hreg, that hreg is added as a candidate for the span. If it is an ordinary vreg, the span to which the value is being copied is examined and its candidates (if any) are added to this span.

**move:** Any candidates found in the current vreg's next span (the one continuing the live range interrupted here by an hreg definition) are copied to this span.

**cont:** The current vreg's first spans for each of the successors of the current block are located, and as many as possible of their candidates are copied to this span.

**All other conditions:** No candidates are added.

Note that the candidate selection algorithm only deals with the span datastructures; it does not need to look at the intermediate code.

### 6.7.2 Execution Cost

The memory requirements of span datastructures are proportional to the number of vregs times the number of spans per vreg. The latter varies, but rarely exceeds two or three per basic block, so the memory needed is roughly proportional to the number of blocks times the average number of vregs live in each block.

Each pass of candidate propagation examines each span (and its predecessors/successors) once. The hash tables allow span accesses at unit cost, so the temporal cost of the algorithm is similar to its memory usage. In practice, no more than three or four passes are necessary.

### 6.8 Register Assignment

#### 6.8.1 The Algorithm

Like candidate propagation, assignment only depends on the span datastructures. However, we do need to replace vregs with hregs in the code, and this may be done concurrently, saving a second scan through the span tables.

Assignment is performed a block at a time. The assigner will try to propagate assignments across block boundaries (just as it did for assignment candidates), but this will not always be possible. The order in which blocks are treated is important in minimising the number of additional copy insns caused by this, and, although the current implementation does a good job by using a straightforward preorder scan of the flowgraph, better results may be obtained by first assigning blocks nested deeply in loops.

To perform assignment efficiently we need, in addition to the span tables, an array of current vreg-to-hreg mappings and a small bitvector identifying the set of hregs currently in use. The assigner depends on the fact that the first insn of each block references no vregs and introduces no-ops where necessary to enforce this; the no-ops will be removed at the code generation stage. The following algorithm is applied to each basic block:

**Algorithm 6.3 (Register Assignment)**

1. Initialise "current" and "future" span pointers to the first element of the block-specific span list. Initialise empty lists to hold spans containing **move** and **cont** conditions.
2. For each insn in the block, starting with the first one, do:

   (a) Replace all vregs referenced by the insn with the corresponding hregs from the current assignments array.

   (b) Scan the block-specific span list from the current to the future pointers, and for each span ending at the current insn do:

      (i) Remove it from the list, and remove its hreg from the hregs-in-use set.

      (ii) If it contains a move or cont condition, save it in the appropriate list(s).

      (iii) If it ends with a move condition, add the hreg it was assigned to as a candidate to the vreg's next span.

   (c) Advance the future span pointer over any spans starting at the current insn, and for each of those do:

      (i) If it is not a span for an explicitly referenced hreg, and it starts (or ends) with a cont condition, add the hregs assigned to the corresponding spans in any predecessor (or successor) blocks that have already undergone assignment as candidates for the current span.

      (ii) If any of the span's candidates are available (i.e. not in the hregs-in-use set), assign one of them to the span, add it to the hregs-in-use set, and update the vreg's entry in the current assignments array.

   (d) If any spans between the current and future pointers are still unassigned (i.e. none of their candidates were available), choose some other hreg for each of them, starting with the callee-saved hregs and proceeding to the caller-saved, argument, result and return address hregs as necessary. Add any hregs used to the in-use set, and update the current assignments array appropriately.

   (e) Replace any vregs defined by the current insn by the corresponding hregs, as indicated by the current assignments array.

Note that the candidate mechanism is being used not only as an aid to hreg selection, but also to propagate information not available during the candidate propagation phase (i.e. the actual assignments that have taken place). This helps minimise the work done, and the insns added, during the final step of the register assignment process.

6.8.2 Execution Cost

Assignment also examines each span once, and, while it may have to look through all the hregs to find one that is available, this can be regarded as a unit-cost operation since the number of hregs is limited. Thus, its cost is similar to that of candidate propagation. No additional memory is needed.

6.9 Assignment Synchronisation

The movement of values from one register to another takes care of itself, mostly. However, attentive readers will have noticed that, although the assignment algorithm above makes an effort to maintain assignments of vregs across basic block boundaries and explicit hreg definitions, it cannot actually
achieve this in every case. This, then, is the point where we have to introduce some copy insns to ensure that values follow the vregs that hold them when assignments change.

Reassignments within and between blocks are slightly different, but they have a basic operation in common: given two sets of assignments of $n$ vregs to hregs, we need to turn one into the other. One way of doing this is by using an auxiliary hreg for temporary storage; this will take between $n$ and $3n/2$ copies, depending on how many assignment changes have to go through the auxiliary register (consider, for example, an even number of vregs exchanging their hregs in pairs; every exchange will have to use the auxiliary hreg).

Setting aside an hreg for temporary storage is unlikely to be worth it, and normally we would only want to use this method if there are any hregs already available, but on some processors it may be possible to use special purpose registers, such as those of the floating point unit, or (in the case of the MIPS architecture) the multiplication result registers. Some processors (such as versions 3 and higher of the ARM, and most CISC architectures including the Intel ia32 series) even provide "exchange" instructions that remove the need for temporary storage altogether, and reduce the number of instructions needed to at most $n$. Obviously, such tricks are highly architecture-dependent; there is still a need for a generic mechanism in case all else fails.

Such a mechanism exists, although it is slightly costlier since it may take up to $3n$ operations to perform the reassignment. It is based on an old programmer's trick: to exchange the contents of two registers, all we need to do is to take the exclusive disjunction of their contents three times. For example, if hregs R1 and R2 contain the values A and B respectively:

- $R1 \leftarrow R1 \text{ XOR } R2$ now contains $A \text{ XOR } B$.
- $R2 \leftarrow R1 \text{ XOR } R2$ now contains $A$.
- $R1 \leftarrow R1 \text{ XOR } R2$ now contains $B$.

This may be a "dirty trick" as far as high-level programming is concerned, but there is nothing to stop us using it at the machine code level. It is also universally applicable\(^1\).

The algorithm we use to perform the reassignment is reminiscent of selection sort: the difference is that we already know where the values should end up, so all we have to do is perform the exchanges.

**Algorithm 6.4**

1. Mark all hregs used in the old assignment as "occupied".

2. While the new assignment has not been completed, and there are vregs whose new hreg is not occupied:

   (a) For each such vreg:

      (i) Issue a copy insn to move it to its new hreg.

      (ii) Mark the old hreg as "unoccupied".

3. While the new assignment has not been completed:

   (a) For each vreg that has not reached its new hreg:

\(^1\) Of the over 20 architectures we have examined, only the HP Saturn [Don95] lacks a machine code XOR instruction. The Saturn, whose use is fortunately confined to Hewlett-Packard calculators, also has other oddities (nibble-addressed memory, registers of various sizes from 4 to 64 bits) that would render most of our work irrelevant.
(i) Move the vreg from its current location to its new hreg by exchanging it with the vreg that is already there. Update the latter's recorded location to reflect the exchange.

The rest of the synchronisation operation is easy. We have already set aside the spans involved (i.e. the ones containing cont and move conditions). The following algorithm handles the insertion of copies to synchronise assignments within a basic block:

Algorithm 6.5 (Intra-block Assignment Synchronisation)

1. For each span ending in a move:
   (a) Add the span's assigned hreg, along with the hreg of the vreg's next span (one hop away in the vreg-specific span list) to a list of reassignments for the insn where the span terminates.

2. For each insn with a list of reassignments:
   (a) Use algorithm 6.4 to insert copies and exchanges after the insn.

The synchronisation of assignments across basic block boundaries is similar to the synchronisation of spill sets, in that it becomes necessary to generate new basic blocks to accommodate the new insns. The arguments about why this is not as costly as it may sound are the same as those presented in section 5.6.9. Here is how to do it:

Algorithm 6.6 (Inter-block Assignment Synchronisation)

1. For each basic block, do:
   (a) For each of the block's predecessors, do:
      (i) Add the hregs of spans ending in cont in the predecessor, along with the hregs of corresponding spans starting with cont in the current block, to a list of reassignments.
      (ii) Generate a sequence of copies and exchanges to interface the two blocks, using algorithm 6.4.
      (iii) If any insns have been generated, create a new basic block containing the insns and insert it between the current block and the predecessor in the flowgraph.

6.10 An Example

To demonstrate the register assignment phase of the P compiler, we will revisit the example of section 5.7. Figure 6.5 on the next page contains the code of figure 5.5 (page 62) after assignment and elimination of no-op adds. The assigner is using only 9 hregs, since that is how we configured the allocator.

Let us examine the program once more:

Procedure entry: Block 2 has now shrunk by 9 insns, because the compiler was able to assign every PCS hreg to itself. Thus, the first add insn represents the initialisation of 'i' (vreg 116), which is assigned hreg 2.

Unfortunately, the compiler has not done as well for the redefinition of 'a', failing to eliminate the temporary (vreg 118) that held its address.
Figure 6.5: WIM3 code for the procedure of Figure 5.4, after register assignment.

Basic block 2
Predecessors: 1

Successors: 5 (f), 6 (f)

Code:
0: opr *0, #0, #3 // main-main entry
1: add $2, $0, $0 // i := 0
2: add $3, $29, #100
3: add $4, $3, $0 // a := w
4: spill R106, $0, #4 // save before load
5: load $3, $0, #4 // x := !a
6: add $9, $4, $3 // a := a + x
7: spill R106, $0, #9 // spill before store
8: for $9, $0, $3 // tx := x
9: unsp R106, $0, #9 // reload a
10: xor $4, $9, $5 // y := a ^ b
11: spill R106, $0, #11 // spill link
12: or $3, $4, $5 // z := y | b
13: jeq L5, $3, $0 // block 5 if z = 0
14: jeq L6, $0, $0 // block 6 otherwise

Basic block 5
Predecessors: 2

Successors: 11 (f)

Code:
0: opr $0, $0, $0
1: add $2, $0, $0
2: add $31, $5, $0
3: add $5, $4, $0 // v := y
4: add $9, $4, $3 // a := w
5: add $4, $2, #0 // call arg: "\%d\n"
6: add $2, $8, $0
7: spill R106, $0, #7 // spill preserved
8: spill R106, $0, #8 // spill w
9: spill R106, $0, #9 // spill c
10: spill R106, $0, #10 // spill b
11: spill R106, $0, #11 // spill a
12: spill R106, $0, #12 // spill z
13: unsp g75, $0, #13 // reload printf
14: call g16, $2, $2 // call printf
15: back $0, $0, $0 // call printf
16: unsp R106, $0, #16 // reload preserved
17: unsp R106, $0, #17 // reload link
18: unsp R106, $0, #18 // reload b
19: unsp R106, $0, #19 // reload x
20: unsp R106, $0, #20 // reload w
21: jeq L11, $0, $0 // go to block 11

Basic block 6
Predecessors: 2

Successors: 8 (f)

Code:
0: opr $0, $0, $0
1: add $4, $31, $0 // v := z
2: add $31, $4, $0 // v := w
3: spill R106, $0, #3 // spill m
4: unsp R105, $0, #4 // reload link
5: jeq L8, $0, $0 // go to block 8

Basic block 7
Predecessors: 8

Successors: 19 (c)

Code:
0: opr $0, $0, $0
1: unsp g75, $0, #1 // unsp saved
2: jeq L19, $0, $0 // go to block 19

Basic block 8
Predecessors: 17

Successors: 6 (f)

Code:
0: opr $0, $0, $0
1: mul $31, $31, #5 // z := v * b
2: sub $4, $4, #2 // v := v - 1
3: div $31, $31, #2 // v := v / 2
4: opr $0, $0, $0
5: add $2, $2, #1 // i := i + 1
6: spill R106, $0, #6 // spill c
7: spill R106, $0, #7 // spill link

Basic block 11

Code:
0: opr $0, $0, $0
1: add $3, $0, $0 // call arg: "\%d\n"
2: unsp R106, $0, #2 // unload c
3: add $4, $4, $31 // set up call arg
4: add $9, $4, #8 // spill preserved
5: spill R106, $0, #5 // spill link
6: spill R106, $0, #6 // spill preserved
7: spill R106, $0, #7 // spill b
8: spill R106, $0, #8 // spill a
9: spill R106, $0, #9 // spill z
10: spill R106, $0, #10 // spill preserved
11: call g16, $2, $2 // call printf
12: back $0, $0, $0 // call printf
13: back $0, $0, $0 // call printf
14: add $3, $0, $0 // main return value
15: add $4, $4, #0 // main return value
16: unsp R106, $0, #16 // reload preserved
17: unsp R106, $0, #17 // reload link
18: unsp R106, $0, #18 // reload w
19: unsp R106, $0, #19 // reload x
20: unsp R106, $0, #20 // reload b
21: jeq L11, $0, $0 // go to block 11

Basic block 19

Code:
0: opr $0, $0, $0
1: unsp g75, $0, #1 // unsp saved
2: jeq L19, $0, $0 // go to block 19

Static data:
"\%d\n"
It is not clear whether this is a problem with the algorithm or the implementation, but it is the only case in this block where the assigner has done a sub-optimal job.

Note that `a` (vreg 106) gets two different hregs in its two definitions (insn 3 and 6). This entails no extra cost, but the reason the original assignment (hreg 4) is not suitable later is subtle: the value of the second definition stays live for most of the code, including the procedure call of block 5. Hreg 4 is updated with the call's first argument a few insns before the call, whereas the new assignment (hreg 9) is not corrupted until the call itself, by which time `a` will have to be split anyway.

It is a pity, therefore, that the assigner (through no fault of its own) soon runs out of registers and has to use hreg 4 to store the value of `y` (vreg 121).

**Calling sequence**: Block 15, added by the allocator, has now been merged with block 5. Insns 3 and 5 of the larger block set up the arguments for the call to `stdio.printf`. The extra moves in insns 1–6 shuffle the registers to allow this to take place: remember that the compiler has no instruction reordering or scheduling capabilities, and is forced to move `y` (vreg 121) out of hreg 4 (insns 4 and 6) before spilling it. However, we have managed to eliminate the copy into argument hreg 5 by assigning `w` to the same hreg.

**Loop entry**: Block 18 has been merged with block 6; insn 1 of the latter assigns hreg 4 to `w`, while `z` dies in the same insn, allowing its hreg (31) to be reused immediately (insn 2). The peephole optimiser can detect 2 as a superfluous insn, and eliminate it.

**Loop body**: The assigner has been particularly successful with block 8, eliminating 4 insns. Remember that we should need only two hregs for vregs 113–115 (`u`, `v` and `w`), even though they all conflict with each other, and this is what happens: `u` and `w` both get hreg 31. This requires an assignment exchange at some point in the loop, handled along with the spill synchronisation in insns 1–3 of block 17, and sacrifices 3 of the 4 insns we saved earlier.

**Assignment synchronisation**: Even more shuffling of hregs has to happen before the branches of the `.test rejoin`. Block 19 handles this, and will later be merged with block 7, which has already incorporated block 16.
Chapter 7

Range Analysis: Introduction and Theory

7.1 Introduction

We have already seen (chapter 3) that the P compiler's optimiser is centred around a range analysis engine. The engine's operation is based on abstract interpretation, a well-known technique that will be introduced later in this chapter.

Range analysis, as its name might suggest, is the process of finding the possible values of variables at any point in a program. We use the term "variables" here in the widest possible sense; we are interested in any quantity that is readable or modifiable by the program at run time. This includes the contents of registers, stack locations, and dynamically allocated memory.

In a way, this sort of analysis is straightforward: we can obtain extremely accurate results by simulating the compiled program's execution. Such simulators are easy to build and very powerful, having the same theoretical capabilities as the machine the code would normally run on. It is their power that makes them unsuitable for our purposes, since the Halting Problem is an unavoidable consequence of it.

Formal statements of the Halting Problem may be found in the literature [Tur36, HU79], but its essence is that, given a machine of sufficient power, there does not exist an algorithm that will predict, in finite time, whether a given program for it eventually terminates. Non-terminating algorithms are not a desirable feature in a compiler, so we must use a method that derives enough information about the program to be useful, but not enough to fall into the realm of the Halting Problem. Abstract interpretation is such a method, but before introducing it, let us presents a survey of program analysis and optimisation methods related to what we are attempting.

7.2 Overview of Related Work

Range analysis is not a new idea. Constant propagation, which may be regarded as a basic form of range analysis, has been a part of the compiler writer's repertoire since the dawn of time, and the idea of predicting the value ranges of non-constant variables was first put forward twenty years ago.

\[\text{In fact, the P compiler contains a complete WIM3 virtual machine, a relic of its early testing phase, which consists of less than 300 lines of code written and debugged in less than 3 days.}\]
This first treatment was more or less empirical, and in fact unnecessarily inaccurate and costly, as pointed out in [Bou90]. Interestingly, the fundamental paper on abstract interpretation, which laid the groundwork for later, more formal approaches to the problem, was published the same year [CC77b]; but this was largely ignored by compiler writers, who considered the problem intractable given the hardware of the day.

However, practical approaches to similar problems were developed early on; type checking in particular had already become quite fashionable with the Algol language family [Nau63, vWMPK69]. A type checking compiler ensures that the programmer is not mixing values of different allowed ranges (i.e. types) thus computing nonsensical results. This may be regarded as a rudimentary form of range analysis where all the range information is provided by the type declarations in programs. As long as the typing rules of the language being compiled are strict enough, it is possible to derive type information in the absence of declarations [Hin69, Mil78]; this is a major feature of the ML language [Dam85, DM82].

Another technique that attempts to extract value information from programs is alias analysis [JM81, CC77a], which considers whether two or more "names" (variables, or expressions evaluating to pointers) may refer to the same storage location. This is useful in enabling loop parallelisation and other code moving transformations.

Abstract interpretation (AIN) [CC77b] is a formal way of abstracting the semantics of a program to a domain where interesting analyses are tractable. It was first applied to an operational semantic framework (of the style introduced in [SS71]) and illustrated with range analysis as its object, but it is in fact a general method that can model other dataflow techniques, such as liveness analysis, and can be layered on top of denotational [MS76] as well as operational semantics [Nie84]. It has been used in a wide variety of applications, including some of the analyses mentioned above:

- **Type inference** can be seen as an AIN process [HK88]; this, and other AIN-based optimisations, have been used in logic language compilers [JS87, Sø86, Mel87]. Also, extensions to type inference, such as the identification of compound types admitting the equality relation, can benefit from AIN [Gun91].

- **Strictness analysis** for lazy functional languages [Myc81] uses AIN to locate values that may be computed strictly without changing the program's behaviour, thus avoiding the production of expensive closures (also known as thunks).

- **Reference counting** is used in heap-based languages to determine when unused storage may be recycled. AIN can be used to model the process at compile time, thus minimising the amount of work that needs to be done at run time [Hud87].

- **Alias analysis** [Deu94] benefits from the greater expressive power of AIN in describing access paths to memory locations.

- **Branch prediction** [Pat95] produces estimates of the frequency with which conditional branches are taken and is useful in optimising code for processors with speculative execution capabilities.

- **Abstract debugging** [Bou90, Bou93] is an interactive debugging process which consists of verifying assertions about the values of variables inserted in source code by the programmer.
**7.3 Objectives**

As can be seen above, AIN has mostly been used to analyse declarative (logic and functional) languages. The consensus of opinion has been that imperative languages are not worth the effort of in-depth static analysis during compilation, due to their complicated control-flow and memory management mechanisms; AIN would yield small profits for quite a high investment in processor time.

We contend that this is not true any more (if indeed it ever was), and claim that any discouraging results have been due to the limited scope of the analyses concerned. Thus the SYNTOX abstract debugger [Bou93], which restricts its analysis of Pascal to integer variables and displays impressive insights into simple programs manipulating such entities, is likely to fall flat on its face as soon as a value is stored in memory rather than a variable, a procedure- or label-valued variable is used, or an external procedure called. The same holds for the branch-predicting optimisation of [Pat95].

We call this phenomenon a "leak catastrophe", since it often takes just a single leak of data into or out of unmodelled domains of execution to render most of our abstract model meaningless. We would like to find a model that reduces the frequency of such catastrophes and limits their scope.

Part of the problem has also been the fact that static analysis techniques have usually been approached with a specific objective in mind, be that better alias analysis, more effective branch prediction, or whatever. Other benefits obtained essentially for free with these processes have largely been disregarded. In particular, many compiler optimisations of this sort have been restricted to array bounds checking, which is not even relevant for some modern languages; the fact that the same information could be used for dead code elimination or even register allocation has usually gone unnoticed.

Another problem is that virtually all such analyses have been done at the source code level, or close to it. This has a few advantages, notably more information about what the programmer intended and lower complexity; but these are dubious at best (the former, in particular, can be misleading as often as not, and is in any case less relevant in low-level languages), while there are also a number of disadvantages, such as:

- No information about the target machine code, which could otherwise benefit from architecture-dependent optimisations.

- No chance to optimise away naïve code inserted by the translator, thus requiring a more sophisticated and therefore complicated front end.

- No register-level model of the processor, which would enable optimisation of register binding and procedure calling.

- No source language independence.

We would like to have a model of the program's behaviour that

- Reflects what happens at the intermediate code level, and can be used to assist low-level optimisations while being independent of source and target languages.
Can model, to some extent, everything the program is allowed to do, thus reducing the frequency of leak catastrophes.

and, keeping in mind that we are working with a typeless, modular language,

- Is not fazed by the opaqueness of source code modules.

- Will allow us to provide to the user some of the advantages of typed programming, in the form of warnings about suspicious operations, without tampering with the fundamental assumption of typeless programming (i.e. that all values are bit patterns and may be manipulated as such).

## 7.4 Abstract Interpretation

### 7.4.1 Introduction and Apologetics

The rest of this chapter is a brief introduction to the theory of abstract interpretation. We aim to present concepts that will be useful later on without getting bogged down in mathematical or implementation details. A rigorous treatment, based on the framework for reasoning about program analyses presented in [CC79], can be found in [Cou81]; what is presented here closely follows the treatment in [CC77b], but is extended and adapted to the context of WIM3 intermediate code. Traditional operational semantics are used (rather than the *structural* operational semantics currently in favour [Plo81]), but this suits us as they are a better model of our intermediate code.

The reader should be warned that we are not going to present a complete formal semantic definition of the WIM3 intermediate code. This is intentional: the code is simple enough for the intended semantics of individual instructions to be obvious, and it was felt that taking the scenic route, from code to concrete semantic description to abstract interpretation to usable results, would be an inefficient process, especially since the semantics are irrelevant to the compiler, which only simulates an abstraction thereof.

We therefore sacrifice mathematical rigour for efficiency, claiming that this does not affect the correctness of the results, or even the provability of correctness. After all, a proof of correctness is only as valid as the semantics it is based on, and debugging semantics is not easier than debugging code. On the contrary, the existence of a semantics may lull their user into a false sense of security and prevent the discovery of errors; see, for example, the errata listings in the definition of ML [HMT89, pp. iii–vii].

Thus, we point the reader to appendix C for an informal definition of the intermediate code, and, aside from a couple of examples for the sake of illustration, leave the semantics implicit in that.

### 7.4.2 Operational Semantics

#### 7.4.2.1 The Lattice of Values

Let $V$ be the set of all possible scalar values dealt with by the WIM3 intermediate representation of a P program. As we will see later, these values are just bitpatterns of fixed length; we can think of $V$ as a subset of the set of integers.

---

2 Less relevant, but still poignant examples of the uneasy relationship between semantics and implementations are call-by-name in Algol-60 (beautiful semantics leading to horrid implementations) and dynamic binding in most LISP dialects (horrid semantics arising from an early implementation blunder).
Z. We define a partially ordered set \( \text{Values}(\sqsubseteq_{\text{val}}) \) by adding elements \( \top \) (Top) and \( \bot \) (Bottom) to \( V \):

\[
\text{Values} = V \cup \{ \top, \bot \}
\]

and defining an order relation \( \subseteq_{\text{val}} \) such that:

\[
\forall v \in \text{Values} . \, v \subseteq_{\text{val}} \top \land v \subseteq_{\text{val}} \bot
\]

while pairs of elements of \( V \) are unrelated by \( \subseteq_{\text{val}} \). A complete lattice is a partially ordered set with least upper and greatest lower bounds for all its subsets. We denote the least upper bound (or join) and greatest lower bound (or meet) operators by \( \sqcup \) and \( \sqcap \) respectively. Clearly, \( \text{Values}(\sqsubseteq_{\text{val}} \bot, \top, \sqcup, \sqcap) \) is a complete lattice with \textit{infimum} \( \sqcap \{ \text{Values} \} = \bot \) and \textit{supremum} \( \sqcup \{ \text{Values} \} = \top \).

### 7.4.2.2 Semantic Environments and States

WIM3 can store values in vregs and memory. We will represent these using \textit{environment} functions; we define

\[
\text{MEnv} = \text{Values} \rightarrow \text{Values}
\]

as the set of functions mapping values, interpreted as bitpatterns giving memory addresses, to the values in the corresponding memory location, and

\[
\text{VEnv} = (V \text{regs} \cup \{ \bot \}) \rightarrow \text{Values}
\]

as the set of functions mapping vregs to the values they hold and \( \bot \) to itself. Note that some vregs are coupled to memory locations with whose contents their values need to agree; we will worry about this later.

An \textit{arc} represents the flow of control from one insn to the next. The set of arcs is a subset of \( \text{Insns} \times \text{Insns} \) and contains all pairs of insns linked by a predecessor/successor relationship:

\[
\text{Arcs} = \{ (i_p, i_c) \mid i_p \in \text{Pred}_1(i_c) \land i_c \in \text{Succ}_1(i_p) \}
\]

Arcs are therefore the equivalent of flowgraph edges at the insn level.

The \textit{state} of a running WIM3 program may be described as a member of the set

\[
\text{States} = \text{Arcs} \times \text{VEnv} \times \text{MEnv}
\]

so that \( \langle (i_p, i_c), v, m \rangle \in \text{States} \) describes a program that has just executed insn \( i_p \) and is about to execute insn \( i_c \) with vreg and memory contents \( v \) and \( m \) respectively.

We use a \textit{substitution} notation for environment updates. Given any environment function \( E \in \text{VEnv} \cup \text{MEnv} \), we may define:

\[
E[v/S][x] = \begin{cases} 
\bot & \text{if } x = \bot \\
v & \text{if } x \in S \\
E(x) & \text{if } x \not\in S
\end{cases}
\]

i.e. \( E[v/S] \) is a function identical to \( E \), except that it returns \( v \) for every element of \( S \).
7.4.2.3 Constant Values

To execute a program we need, in addition to the environment, some information which is immutable throughout the program's execution. This includes:

- The mappings from labels (i.e. basic blocks), procedures and static object (i.e. string and table) identifiers to memory addresses:

  \[
  \begin{align*}
  \text{ADDR}_{\text{lab}} & \in \text{Blocks} \rightarrow \text{Values} \\
  \text{ADDR}_{\text{fun}} & \in \text{Procs} \rightarrow \text{Values} \\
  \text{ADDR}_{\text{str}} & \in \text{Statics} \rightarrow \text{Values}
  \end{align*}
  \]

  respectively.

- The mapping from insns to addresses and vice versa:

  \[
  \begin{align*}
  \text{ADDR}_{\text{insn}} & \in \text{Insns} \rightarrow \text{Values} \\
  \text{INSN}_{\text{addr}} & \in \text{Values} \rightarrow \text{Insns}
  \end{align*}
  \]

- The mapping from vregs to addresses and vice versa:

  \[
  \begin{align*}
  \text{ADDR}_{\text{vreg}} & \in (\text{Vregs} \cup \bot) \rightarrow \text{Values} \\
  \text{VREG}_{\text{addr}} & \in \text{Values} \rightarrow (\text{Vregs} \cup \bot)
  \end{align*}
  \]

- The offsets of vector bases from the bottom of the stack frame:

  \[
  \text{Off}_{\text{vec}} \in \text{Vectors} \rightarrow \text{Values}
  \]

These functions cannot be defined without reference to the assembly, link and load process for the finished program, but for the purposes of semantic analysis it will be sufficient to note some of their properties.

1. The insn corresponding to the address of a block or procedure must be the first one in the block or procedure:

   \[
   \begin{align*}
   \text{INSN}_{\text{addr}}(\text{ADDR}_{\text{lab}}(b)) &= \text{FIRST}_1(b) \\
   \text{INSN}_{\text{addr}}(\text{ADDR}_{\text{fun}}(f)) &= \text{FIRST}_1(\text{FIRST}_B(f)) \\
   \forall v \in \text{Vregs} . \text{VREG}_{\text{addr}}(\text{ADDR}_{\text{vreg}}(v)) &= v
   \end{align*}
   \]

2. Only vregs whose address is taken at some point in the source program have addresses, and not all addresses correspond to vregs:

   \[
   \forall v \notin \text{Addrvars} . \text{ADDR}_{\text{vreg}}(v) = \bot \\
   \text{VREG}_{\text{addr}}(a) = \bot \iff \exists v \in \text{Vregs} . \text{ADDR}_{\text{vreg}}(v) = a
   \]

3. The synchronisation condition:

   \[
   \forall v \in \text{Vregs} . \text{ADDR}_{\text{vreg}}(v) = \bot \lor \text{E}_m(\text{ADDR}_{\text{vreg}}(v)) = \text{E}_v(v)
   \]

   ensures that for any state \((i, \text{E}_v, \text{E}_m)\), the contents of vregs agree with their copies in memory.

Finally, note that \text{ADDR}_{\text{vreg}} and \text{VREG}_{\text{addr}} are only constant within a WIM3 procedure; the activation of other stack frames by calls and returns causes them to change. We are not modelling this, since we will not deal with procedure calls until chapter 11.
7.4.2.4 Interpretation ofInsn Operands

We can now define the interpretation of arguments referenced by the insns containing them.

\[
\text{ARGV} \in \text{States} \times \{1, 2, 3, 4\} \rightarrow \text{Values}
\]

is such that \(\text{ARGV}(S, n)\) gives the value of the \(n\)th argument of the insn at the head of the arc in state \(S\), assuming that this argument is not a destination for an ALU operation or load from memory.

\[
\text{ARGV}([\langle i_p, i_c \rangle, E_v, E_m], n) = \begin{cases} 
\text{ARG}[i_c, n] & \text{if } \text{ARG}[i_c, n] = \text{con} \\
\text{ADDR}_{\text{reg}}[\text{ARG}[i_c, n]] & \text{if } \text{ARG}[i_c, n] = \text{adr} \\
E_v[\text{ARG}[i_c, n]] & \text{if } \text{ARG}[i_c, n] = \text{reg} \\
\text{ADDR}_{\text{add}}[\text{ARG}[i_c, n]] & \text{if } \text{ARG}[i_c, n] = \text{bb1} \\
\text{ADDR}_{\text{fun}}[\text{ARG}[i_c, n]] & \text{if } \text{ARG}[i_c, n] = \text{fun} \\
\text{ADDR}_{\text{str}}[\text{ARG}[i_c, n]] & \text{if } \text{ARG}[i_c, n] = \text{str} \\
\text{OFF}_{\text{vec}}[\text{ARG}[i_c, n]] & \text{if } \text{ARG}[i_c, n] = \text{vec} 
\end{cases}
\]

7.4.2.5 State Transitions andInsn Semantics

The heart of WIM3 semantics is the state transition function,

\[
\text{NEXT}_{\text{state}} \in \text{States} \rightarrow \text{States}
\]

which describes what each insn does to the state of the program. This function will not be presented in full, but it will be instructive to look at some common cases. In what follows, \(S = [\langle i_p, i_c \rangle, E_v, E_m]\); \(i_s \in \text{Succ}[i_c]\);\(^3\) and \(+\_\text{val}\) is an addition operator for Values, which behaves normally for numbers but returns \(T\) if the result is out of range or an operand is \(T\), and \(\bot\) if an operand is \(\bot\).

The \text{add} insn assigns the sum of its last two arguments to the \text{vreg} specified by its first argument, and leaves memory untouched:

\[
[O_{i_c} = \text{add}] \Rightarrow \text{NEXT}_{\text{state}}(S) =
[\langle i_c, i_s, E_v[\text{ARGV}(S, 2) + +\_\text{val}\_\text{ARGV}(S, 3)], E_m[\text{ARGV}(S, 1)]]
\]

The \text{stor} insn updates a memory location, whose address is obtained by summing its last two arguments, with the value of its first argument. If the address is that of a vreg, the vreg also has to be updated:

\[
[O_{i_c} = \text{stor}] \Rightarrow \text{NEXT}_{\text{state}}(S) =
[\langle i_c, i_s, E_v[\text{ARGV}(S, 1) + +\_\text{val}\_\text{ARGV}(S, 3)]],
E_m[\text{ARGV}(S, 1) + +\_\text{val}\_\text{ARGV}(S, 3)]]
\]

The “jump on equal” insn selects one of two successor arguments depending on whether its last two arguments are equal:

\[
[O_{i_c} = \text{jeq}] \Rightarrow \text{NEXT}_{\text{state}}(S) =
\begin{cases} 
[\langle i_c, \text{INSN}_{\text{add}}[\text{ARGV}(S, 1)], E_v, E_m \rangle] & \text{if } \text{ARGV}(S, 2) = \text{ARGV}(S, 3) \\
[\langle i_c, \text{INSN}_{\text{add}}[\text{ARGV}(S, 1)], E_v, E_m \rangle] & \text{if } \text{ARGV}(S, 2) \neq \text{ARGV}(S, 3) 
\end{cases}
\]

To avoid modelling procedure calls, we will treat \text{call} insns by assuming the existence of

\[
\text{FUN\_Call} \in \text{INSN} \times \text{VENV} \times \text{MENV} \rightarrow \text{VENV} \times \text{MENV}
\]

\(^3\)Since all insns except for jumps and calls have a sole successor, \(i_s\) is unique wherever it is used.
which returns the contents of vregs and memory after a call to the procedure with the given first insn. Thus,

$$\text{OP}_1(i_c) = \text{call} \land \text{FUNCALL} \text{INSN}_{\text{addr}}(\text{ARGV}(S, 1), E_v, E_m) = (E'_v, E'_m) \Rightarrow$$

$$\text{NEXT}_{\text{state}}(S) = (\langle i_c, i_s \rangle, E'_v, E'_m)$$

Again ignoring the interprocedural realm, we will consider the computation finished when a return insn is reached:

$$\text{OP}_1(i_c) - \text{return} \Rightarrow \text{NEXT}_{\text{state}}(S) = S$$

This effectively stops the interpretation at a return insn by forcing the program to stay in the same state.

### 7.4.2.6 Semantics of a Running Program

A *computation sequence* is a succession of states experienced by the program during its execution. If $\text{States}_{\text{init}}$ is the set of possible initial states, and $S_{\text{init}} \in \text{States}_{\text{init}}$ is the particular initial state we are interested in, then the $n$th element of the sequence is $\text{NEXT}_{\text{state}}^n(S_{\text{init}})$. ($f^n$ is defined as $f$ composed with itself $n$ times, so that $f^0$ is an identity function and $f^{n+1} = f \circ f^n$.)

Finally, a computation completes when further application of the transition function does not yield a change in state (we have ensured this can only happen when a *return* insn is reached), so the transition function from the initial to the final state is given by the least fixed point of

$$\lambda f. \text{NEXT}_{\text{state}} \circ f$$

### 7.4.3 Static Semantics

The operational semantics outlined above is just a formalised equivalent of program execution. While this gives us the maximum amount of information, it is definitely not what we want our compiler to do, since it does not circumvent the Halting Problem.

However, we can use the operational semantics as a starting point in deriving an analysis that does a less thorough, but more efficient, job. We can start by observing that our approach assigns more than one state to each insn, depending on the number of ways the insn can be reached during execution; an insn in a loop that is executed 10 times will have 10 different states associated with it in the computation sequence. We can try to group all the states for each insn together, thus obtaining a *static* (or *collecting*, or *concrete*) semantics for the program.

In the static semantics, each arc $a$ is associated with a *context*. $C(a) \in \text{Contexts}$, defined by

$$C(a) = \{ \langle E_v, E_m \rangle \mid \exists n \in \mathbb{N} : \exists S_{\text{init}} \in \text{States}_{\text{init}} : \text{NEXT}_{\text{state}}^n(S_{\text{init}}) = \langle a, E_v, E_m \rangle \}$$

i.e. the set of all environment pairs from states for arc $a$. Given the context vector $C$, we can define a context vector transition function, yielding the context at $a$ by examining the contexts at $a$'s predecessors:

$$\text{NEXT}_{\text{cont}}(C) = \lambda a. \{ \langle E_v, E_m \rangle \mid a = \langle i_p, i_c \rangle \land b = \langle i_{pp}, i_p \rangle \land$$

$$b \in \text{ArCs} \land S \in C(b) \land$$

$$\text{NEXT}_{\text{state}}(S) = \langle a, E_v, E_m \rangle \}$$
7.4. ABSTRACT INTERPRETATION

But \( \text{NEXT}_{\text{cont}}(C) \) is just \( C \), and if it can be shown that \( \text{NEXT}_{\text{cont}} \) has fixed points and that \( C \) is contained in all of them, it follows [Tar55] that \( C \) is the least fixed point of \( \text{NEXT}_{\text{cont}} \).

Even though they do not give us any information about which of the states in an arc's context will occur at any given activation of the arc, the static semantics are still too accurate: the contexts contain all the states that will be reached by the program during its execution, and only those states. We cannot guarantee termination yet, since we are effectively executing every possible path through the program in computing accurate context vectors.

7.4.4 Abstract Semantics

An abstract interpretation (AIN) is a tuple\(^4\)

\[
\langle \text{Contexts}_{\text{abs}}, \alpha, \subseteq, \perp, \text{NEXT}_{\text{abs}} \rangle
\]

\( \text{Contexts}_{\text{abs}} \), the set of abstract contexts, is the underlying set of a complete lattice with bounds \( \top \) and \( \perp \), ordering \( \subseteq \) and join or meet operator (we only need one) \( \circ \). The interpretation function

\[
\text{NEXT}_{\text{abs}} \in \langle \text{ArCs} \rightarrow \text{Contexts}_{\text{abs}} \rangle \rightarrow \langle \text{ArCs} \rightarrow \text{Contexts}_{\text{abs}} \rangle
\]

defines the actual interpretation, just as \( \text{NEXT}_{\text{cont}} \) does for concrete semantics; if \( C_{\text{abs}} \) is an abstract context vector, \( \text{NEXT}_{\text{abs}}(C_{\text{abs}})(a) \) gives the abstract context for arc \( a \). We require \( \text{NEXT}_{\text{abs}} \) to be order-preserving with respect to \( \subseteq_{\text{abs}} \), i.e. given any two abstract context vectors \( C_{\text{abs}:1} \) and \( C_{\text{abs}:2} \), we expect

\[
(\forall a \in \text{ArCs} . \ C_{\text{abs}:1}(a) \subseteq_{\text{abs}} C_{\text{abs}:2}(a)) \Rightarrow
(\forall a \in \text{ArCs} . \ \text{NEXT}_{\text{abs}}(C_{\text{abs}:1})(a) \subseteq_{\text{abs}} \text{NEXT}_{\text{abs}}(C_{\text{abs}:2})(a))
\]

This ensures that it has fixed points, which are the context vectors we are looking for.

To exhibit the link between AINs and program semantics, it is sufficient to note that the static semantics we have presented form an AIN:

\[
\langle \text{Contexts}_{\text{abs}}, \cup, \subseteq, \text{VEnv} \times \text{MEnv}, \{ \}, \text{NEXT}_{\text{cont}} \rangle
\]

where \( \cup \) and \( \subseteq \) denote the set union operator and subset relation respectively. We can consider this the "least abstract" interpretation (hence the name concrete semantics) and devise less accurate interpretations that are consistent with it, in the familiar sense of safely approximating it. To this end, given any "more abstract" interpretation we define mappings from concrete to abstract contexts and vice versa, \( \alpha \) and \( \gamma \) respectively. These mappings must be order-preserving, and must normally obey

\[
\forall x \in \text{Contexts}_{\text{abs}} . \ x = \alpha(\gamma(x)) \quad (7.2)
\]

\[
\forall x \in \text{Contexts} . \ x \subseteq \gamma(\alpha(x)) \quad (7.3)
\]

We can extend these mappings to context vectors in the natural way:

\[
\alpha(C) = \lambda a . \alpha(C(a))
\]

\[
\gamma(C_{\text{abs}}) = \lambda a . \gamma(C_{\text{abs}}(a))
\]

\(^4\)Note that this definition differs, for the sake of simplicity, from that of [CC77b], where \( \text{NEXT}_{\text{abs}} \) is replaced by a context (rather than context vector) transition function, as one can be derived from the other, the two are interchangeable.
For our AIN to be consistent, we now require
\[ \forall a \in \text{Aracs} . \ C(a) \subseteq \gamma(C_{\text{abs}})(a) \land \alpha(C)(a) \subseteq \gamma(C_{\text{abs}})(a) \]  
where \( C \) and \( C_{\text{abs}} \) are context vector solutions to the concrete and abstract interpretations respectively. It can be shown that this “global” condition is equivalent to the pair of “local” conditions:
\[ \forall a \in \text{Aracs} . \]
\[ (\forall C_{\text{abs}} . \ \text{NEXT}_{\text{cont}}[\gamma(C_{\text{abs}})(a)] \subseteq \gamma(\text{NEXT}_{\text{abs}}(C_{\text{abs}})(a))) \land \]
\[ (\forall C . \ \alpha(\text{NEXT}_{\text{cont}}(C)(a)) \subseteq \text{abs} \ \text{NEXT}_{\text{abs}}(\alpha(C)(a))) \]  

7.4.5 Widening and Narrowing

Fixed point solutions of the AIN equation may be found by iterative means. Starting with an empty context vector returning \( \bot \) for every vreg and memory location, we can apply the \text{NEXT}_{\text{abs}} function repeatedly until no more changes are observed, i.e. until a fixed point is reached:
\[ C_0 = \bot \]
\[ C_{n+1} = \text{NEXT}_{\text{abs}}(C_n) \]

Unfortunately, this is not always feasible; the process can only be guaranteed to terminate in finite (or reasonable) time if the lattice we are working with contains no infinite (or very long) ascending chains.

This is a problem because many of the interesting lattices we would like to use, such as the lattice of integer intervals, do contain infinite (or, in the case of machine integers, very long) ascending chains. The solution is to use some sort of shortcut, accelerating the convergence process while trading off as little of the already limited (due to abstraction) accuracy we have left as possible.

It can be shown [CC77b] that the iterative process can be made to converge by

1. Defining a widening operator, \( \triangledown \), on the lattice. This operator should be such that for any elements \( c_1, c_2, \ldots, c_n \) of the lattice,
\[ (c_1 \circ c_2) \subseteq \text{abs} \ (c_1 \triangledown c_2) \]

and the sequence
\[ S_1 = c_1 \]
\[ S_{n+1} = S_n \triangledown c_n \]
is not strictly increasing.

2. Defining a set of widening arcs, \( \text{Widen} \), such that every cycle contains at least one widening arc. For the applications we are interested in, “cycle” refers to flowgraph cycles, such as loops (for a more general definition, see [CC77b]). We can thus define \( \text{Widen} \) as the set of all arcs corresponding to the flowgraph’s back edges:
\[ \text{Widen} = \{(i_1, i_2) | i_1 = \text{LAST}_1(b_1) \land i_2 = \text{FIRST}_1(b_2) \land \text{EDGE}_B(b_1, b_2) = \text{back} \} \]

3. Modifying the iterative process as follows:
\[ C_0 = \bot \]
\[ C_{n+1} = \lambda a . \begin{cases} \text{NEXT}_{\text{abs}}(C_n)(a), & \text{if } a \notin \text{Widen} \\ C_n(a) \triangledown \text{NEXT}_{\text{abs}}(C_n)(a), & \text{if } a \in \text{Widen} \end{cases} \]
The result of this modified analysis is not guaranteed to be a fixed point of the original \texttt{NEXT}_{\texttt{abs}}, but it \textit{is} guaranteed to not be less than the least fixed point, so it is an admissible (although slightly less accurate) solution. By replacing the widening operator with a \textit{narrowing} one, $\triangle$, and repeating the process, we can get a better approximation to the least fixed point. Given elements $c_1, c_2, \ldots c_n$ of the lattice, $\triangle$ should be such that

$$ (c_1 \sqsubseteq_{\texttt{abs}} c_2) \Rightarrow (c_1 \sqsubseteq_{\texttt{abs}} (c_2 \triangle c_1) \sqsubseteq_{\texttt{abs}} c_2) $$

and the sequence

$$
\begin{align*}
S_1 &= c_1 \\
S_{n+1} &= S_n \triangle c_n
\end{align*}
$$

is not strictly decreasing.
Chapter 8

The Value Range Analysis Lattice

8.1 Introduction

Having presented the theory behind AIN, we can now introduce the particular AIN framework we will use to perform value range analysis on our intermediate code. This chapter describes the construction of an abstract lattice describing program states out of subsidiary abstractions for values.

Our framework preserves much of the structure of the static semantics: The context assigned to an arc will consist of a pair of environment functions giving the abstract values of vregs and memory, and the ordering of the abstract lattice will be based on the subset relation. However, we will adopt two qualitatively different sorts of abstraction:

1. Inaccuracies due to widening, uncertainty about initial values and runtime input, operations undefined in the source language, and lack of information about results of jumps and procedure calls.

2. Failure to distinguish between the contents of certain collections of memory locations.

We will first define a lattice of abstract values,

\[ \text{Values}_{\text{abs}}[E_{\text{abv}}, \bot_{\text{abv}}, \top_{\text{abv}}, \cup, \cap] \]

representing the possible contents of vregs and memory locations and allowing for the first type of abstraction described above. This will later be extended to describe contexts and context vectors, and the second type of abstraction will be dealt with.

8.2 Abstract Values

8.2.1 How Much to Abstract?

Given a single P variable, what could its contents be? Quoting the BCPL Standard [MFRW79]:

The only unit of data is the BCPL word which for any given implementation is a bit string of fixed length not less than 16 bits. A
BCPL word may be used to represent values of many different types [...]. The compiler [...] always assumes that operands have the required type.

The basic unit of storage is a cell which is large enough to hold a BCPL word. Each available cell has an integer address which can be operated upon and stored in the same way as any other BCPL word.

The above is a fairly good description of the interpretation of values in any typeless language, but is not actually very helpful: modelling P values as bit strings is not a very good idea. There are several reasons for this:

- Procedure addresses and the values of labels are not known at compile time. Depending on the implementation, they may not even be known at link time, and may vary from one run to the next\(^1\).

- Different computers have different word sizes. Machines currently in use span the whole range from 16 to 64 bits, and we would like our programs to be portable. Locating word size dependencies by a word size-dependent analysis is hard.

- The bit patterns would still have to be interpreted in symbolic ways when used as addresses of code or labels; otherwise the modelled domain would become unmanageably huge (at least as large as the target machine’s virtual address space).

It is obvious that we have to model at least some values in another way, closer to their normal intended use. But since all values are bit patterns, and since bit patterns as such have to be part of the model, the possibility of having two different abstractions of the same bit pattern arises. There is therefore a guarantee and a tradeoff to be made.

The guarantee is that of safety: users who know what they are doing should still be able to use typeless programming tricks ranging from the clever, such as jump/call vector tables stored in arrays, through the unusual, such as calls to machine code stored in arrays, to the perverse, such as self-decompiling and self-modifying code\(^2\), without causing the compiler to generate incorrect code. Not all of the above need be modelled, but safe approximations have to be made when modelling is impossible.

The tradeoff is in how much of the above is modelled without hampering the analysis of normal programs\(^3\). The model presented below strikes, we hope, a reasonable balance between the two.

### 8.2.2 Introduction to Value Kinds

The key observation is that most typeless programs behave essentially like typed ones. For example, bitpatterns intended as procedure addresses will probably not be incremented by integer amounts, even though we could conceive of circumstances where this would be useful. We therefore divide the values manipulated by P programs into a number of “kinds”, roughly corresponding to types.

---

\(^1\) This is the case for the Cintcode [Ric96b] and SIC [Fou93] implementations of BCPL which link by concatenation and relocate on loading.

\(^2\) Modulo the restrictions of instruction cache coherence on modern processors.

\(^3\) There is also the minor tradeoff of flagging suspicious code with compile-time warnings versus annoying the Programmer Who Knows Better; this is discussed in chapter 10.
Rather like the "animal kinds" of Biblical Creation "Science" [Bir89], we will admit evolution of entities within some kinds (e.g. integer arithmetic on numbers, offset calculation and pointer arithmetic on data addresses) but refuse to contemplate migration of entities from one kind to another (e.g. the interpretation of numbers as addresses and vice versa). The difference is that instead of ignoring such occurrences, we will attempt to make allowances by providing "don't know" values to abstract them.

Keeping in mind all of the above, we may now give a more detailed answer to the question of the possible contents of P variables. They have to represent a value of at least one of the following kinds:

**Program Labels.** These are the values defined by .place expressions and returned by .label expressions; they are typically used as arguments to .goto. They correspond to addresses in the final executable code.

**Procedure Addresses.** Conceptually similar to the above, these also correspond to code addresses but their use must conform to the architecture's procedure calling standard. In P they are defined by .defun, returned by .fun and used as the arguments of procedure call expressions.

**Variable Addresses.** These are the values returned by P's address-of (':') operator. They represent addresses in the program's data space where the contents of scalar variables are supposed to be held; due to register allocation, this may not always be true, but the compiler has to maintain the illusion.

**Integers.** Integer values, including numeric and character constants as well as the results of applying arithmetic operations to them, populate this kind.

**Static Data.** These are addresses referring to the program's read-only data area, populated by tables and strings (produced by .table and "...") expressions.

**Dynamic Stack Data.** This kind contains addresses belonging to stack memory allocated at procedure calls to hold vectors (called arrays in most modern languages; "vectors" emphasises their one-dimensional nature).

**Dynamic Heap Data.** Addresses in heap memory blocks allocated by the runtime system in response to calls of library procedures belong here.

Each of the above has its own peculiarities, and will be discussed individually below. but before examining them in more detail it is worth noting that the first three are different from the last four in an important way: because of the tradeoff we have made, it is not possible to obtain a different value of the same kind by repeatedly applying the same operation (e.g. adding one to the current value). A procedure address to which a number has been added is not a procedure address any more, and will not be modelled as such. A disadvantage of this is that subtracting the same number will not yield the original address as far as our analysis is concerned, and a safe assumption will have to be made; the advantage is that these "non-iterative" value kinds can be modelled precisely using a lattice containing one element for each procedure, label and variable address. Such a lattice has no infinite chains, so we do not need to worry about widening and narrowing operators for it. We will discuss widening and narrowing for the "iterative" kinds later on.
8.2.3 Program Labels

There is only a finite number of labels in a program, and no way to generate new labels by iterative operations. Thus it is possible to represent labels with full accuracy: the set Blocks of labels forms the first of our value kinds.

8.2.4 Procedure Addresses

There is also a finite number of procedures in a program. Representing these accurately is even more important than for labels, if we are to deal with interprocedural analysis at all. Thus, the set Proc of procedure addresses should be included in our values lattice.

8.2.5 Variable Addresses

Variables that have their address taken at some point in a program are special in many ways; dealing with them is tricky, as we have already seen in the chapter on register allocation (see section 5.6.4). Dealing with their addresses also requires care.

The addresses of variables may be identified with the variables themselves as long as we are clear about which of the two we are referring to, and we may incorporate the set Addrvars in our lattice. It is important, however, to note that this identification introduces abstraction: we are essentially identifying "syntactic" variables which, if declared in recursive procedures, will have varying numbers of simultaneously active incarnations at run time, and an equivalent number of different addresses, one for each active stack frame.

In our static analysis, each element of Addrvars will correspond to a number of actual addresses. This blurring of the distinction between variable instances is a deliberate feature: contexts containing such addresses abstract all the states in which one of the instances is being referred to. This has implications for the interpretation of loads and stores which we will discuss later (see section 9.5).

8.2.6 Integers

The semantics of integers are easy to understand. Integers are just numbers; we don't even need to worry about identifying them uniquely. The problems we have with them are more subtle than that, and arise from the fact that not only are there billions and billions of them, but it is easy to obtain new numbers from old ones by performing arithmetic. We need a succinct notation for representing more than one integer at a time, without losing too much accuracy in common cases.
Such a notation has been used in AIN from the very beginning. We define the set of integer intervals, Intervals $\subseteq P(\mathbb{Z} \cup \{0\ldots\infty\})$, such that
\[
\forall S \in \text{Intervals}, \forall n \in \mathbb{Z}, \ \min(S) \leq n \leq \max(S) \Rightarrow n \in S
\]
and represent the interval containing all integers between $m$ and $n$ inclusive by $[m \ldots n]$. Single numbers are represented by intervals having the same upper and lower limit, wherever such accuracy is possible. Intervals is ordered by inclusion; see figure 8.2 for an illustration of the resulting lattice.

One subtle point that needs to be raised here involves the minimum and maximum values of the set of machine integers. Our analyses will often produce results containing "infinite" values (e.g. the range of a non-negative integer about which nothing else is known will be $[0 \ldots +\infty]$) but the implementation can only represent numbers that can fit in a machine word. We will consider all results outside the bounds of representable integers to be either positive or negative infinities; this is consistent with the semantics of $P$, which leaves the behaviour of overflows and underflows undefined, and also helps us to increase the accuracy of our analysis in the widening phase, as we will see later.

8.2.7 Static Data Addresses

Each $P$ module may have an amount of read-only static storage, consisting of character strings and tables. The contents of these objects are available to the optimiser, so we need a way of representing pointers to them.

Unlike the static objects themselves, which are finite in number, there is a large number of possible pointers to them, since a pointer may address any element of the object. We deal with this by associating an object identifier with an offset range, consisting of integer intervals that represent the possible offsets of the element being pointed to from the object's base:

\[
\text{StatVals} = \{(s, i) \mid s \in \text{Statics} \land i \in \text{Offsets}\}
\]

The formal definition of Offsets will be presented later.
8.2.8 Dynamic Stack Data Addresses

Vectors declared in a procedure are similar to static data, in that we need offset ranges to represent pointers into them. The difference is that, since vectors are allocated in the procedure's stack frame at run time, there may be more than one instance of a vector in existence at any one time. Thus even vector ranges with single integer offsets may refer to more than one memory address, in the same way as variable addresses, if the containing procedure is recursive. We will have to take this into account when modelling loads and stores, but in all other respects this kind of value is similar to the previous one:

\[ \text{VectVals} = \{(v, i) \mid v \in \text{Vectors} \land i \in \text{Offsets}\} \]

8.2.9 Dynamic Heap Data Addresses

This is the hardest kind of value to represent because at compile time there is no indication of the number of heap blocks the program will use, and no way to associate an identifier with each of them. To model them efficiently we will again have to deliberately fail to distinguish between some blocks.

We could, of course, regard all references to the heap as accessing the same area, as long as our handling of stores is non-destructive (to allow for the alternate areas we are not modelling), but since programs use the heap to store many different kinds of data, this approach will soon degenerate into reading and writing random values, as far as our analysis is concerned. Fortunately, we can do better than that.

The method we adopt treats all blocks allocated by the same insn as one. Rather than introduce a new insn type, we will arrange for our interprocedural analysis system (to be described later) to return a special value indicating a "fresh" block whenever a memory allocation library routine is called. When the calling procedure is analysed, the fresh identifier is replaced with one unique to the call insn. All references to blocks allocated by the same insn update (non-destructively) the same abstract construct in our analysis, but blocks allocated by different calls are distinguished. Of course, each reference to a collection of blocks will also require an offset range, just as for vectors and statics:

\[ \text{HeapVals} = \{(b, i) \mid b \in (\text{Insns} \cup \{\text{Fresh}\}) \land i \in \text{Offsets}\} \]

This approach reflects common programming practice, since blocks likely to hold the same sort of data are usually allocated at a small number of repeatedly executed call sites; for example, a linked list may be formed by repeated calls of the allocation procedure in a loop. It is also a flexible approach: by adjusting the "freshness" of pointers returned by procedures we can bypass the library's allocation mechanism, bring the abstract model closer to the intended meaning of the program, or compromise between accurate and succinct representations.

However, we should note that in distinguishing between blocks allocated by different insns, we are making an implicit assumption—namely, that two insns cannot return the same block. This is not strictly true, since a freed block may be reused; but, according to the ISO specifications of the C library [ISO90], which P uses for heap management, accesses to freed blocks are undefined operations. Thus, when a block is reused in a standard-conforming program, all prior references to it must have vanished, and we may safely assume that it is a fresh block with undefined contents.
8.3 Value Ranges, Environments and Associations

8.3.1 Scalar Value Ranges

It should be obvious that representing the value of a variable or memory location by a single item belonging to one of the above value kinds will not be sufficient. This is not only because our analysis will often not be accurate enough to decide which of two or more possible value kinds is the appropriate one, but also because we are dealing with a typeless language, where the same variable may take on values of different kinds depending on the flow of control through the program.

The solution is to regard the elements of the value kinds as *range components* and the value ranges themselves as collections of such components. The set of all possible component values is

$$\text{CompVals} = \text{Blocks} \cup \text{ProcS} \cup \text{Addrvars} \cup \text{Intervals} \cup \text{StatVals} \cup \text{VectVals} \cup \text{HeapVals}$$

and the underlying set of our abstract values lattice is

$$\text{Values}_{\text{abs}} \subset \mathcal{P}(\text{Comps})$$

i.e. part of the set of all subsets of Comps. The latter is the set of value range components, defined as

$$\text{Comps} = \text{CompIds} \times \text{CompVals} \times \text{Assocs}$$

i.e. the set of all value kinds taken together, and bundled with elements from CompIds and Assocs which will be explained later. If

$$c = (\text{id}, \text{val}, \text{asc}) \in \text{rng} \in \text{Values}_{\text{abs}}$$

i.e. $c \in \text{Comps}$ is a value range component, consisting of the value val bundled with id $\in \text{CompIds}$ and asc $\in \text{Assocs}$, and belonging to value range rng, we...
define projection functions:

\[
\begin{align*}
\text{PROJ}_\text{id}[c] & = \text{id} \\
\text{PROJ}_\text{val}[c] & = \text{val} \\
\text{PROJ}_\text{asc}[c] & = \text{asc} \\
\text{PROJ}_\text{id}(\text{rng}) & = \{\text{id} | \langle \text{id}, \text{val}, \text{asc} \rangle \in \text{rng}\} \\
\text{PROJ}_\text{val}(\text{rng}) & = \{\text{val} | \langle \text{id}, \text{val}, \text{asc} \rangle \in \text{rng}\} \\
\text{PROJ}_\text{asc}(\text{rng}) & = \{\text{asc} | \langle \text{id}, \text{val}, \text{asc} \rangle \in \text{rng}\}
\end{align*}
\]

Thus the range of a variable whose value could be one of two procedure addresses, like that of variable \(j\) at point (1) in the program of figure 8.3 on the preceding page, will have a \(\text{PROJ}_\text{vals}\) consisting of a set containing two items of the procedure kind, one for each address:

\[
\{\text{fun mod.} \text{foo}, \text{fun mod.} \text{bar}\}
\]

while a variable like \(b\) in the example program, whose value is an integer either less than 0 or greater than 10, will have a value range whose \(\text{PROJ}_\text{vals}\) consists of two integer intervals:

\[
\{[-\infty \ldots -1], [+11 \ldots +\infty]\}
\]

and a variable whose value is that of one of the above (such as \(k\) in the example program) might have a range consisting of the union of the previous two ranges, with a \(\text{PROJ}_\text{vals}\) of:

\[
\{[-\infty \ldots -1], [+11 \ldots +\infty], \text{fun mod.} \text{foo}, \text{fun mod.} \text{bar}\}
\]

It is important to note that \(\text{Values}_{\text{abs}}\) is only a subset of the powerset of \(\text{Comps}\); this is because we wish to exclude some component combinations, namely overlapping integer (or offset) intervals and duplicated address bases. We will not specify this requirement formally, but we will later ensure that the results of all our operations on value ranges conform to it; see the definition of the normalisation operator in section 9.2.2.

Some of the value kinds have features in common, and we will often find it useful to work on partitions of a value range containing only its integer intervals, or components with offsets, or other component kinds. We define functions to extract these:

\[
\begin{align*}
\text{PART}_{\text{ints}}(\tau) & = \{c \mid c \in \tau \land \text{PROJ}_\text{val}[c] \in \text{Intervals}\} \\
\text{PART}_\text{addr}(\tau) & = \{c \mid c \in \tau \land \text{PROJ}_\text{val}[c] \in \text{StatVals} \cup \text{VectVals} \cup \text{HeapVals}\} \\
\text{PART}_\text{misc}(\tau) & = \{c \mid c \in \tau \land \text{PROJ}_\text{val}[c] \in \text{Blocks} \cup \text{Proc} \cup \text{Addrvars}\}
\end{align*}
\]

Finally, we can now define the set of offset ranges, which consists of value ranges containing only integer interval components:

\[
\text{Offsets} \subseteq \mathcal{P}(\text{CompIds} \times \text{Intervals} \times \text{Assocs})
\]

and a couple of projection functions for the value kinds with offsets, so that if \(c = \langle \text{id}, \langle \text{bas}, \text{off} \rangle, \text{asc} \rangle\),

\[
\begin{align*}
\text{PROJ}_\text{bas}[c] & = \text{bas} \\
\text{PROJ}_\text{off}[c] & = \text{off}
\end{align*}
\]
8.3.2 Ordering the Abstract Values

As has been mentioned before, the ordering of the abstract values is based on the subset relation: a value range is “smaller” than any range containing all the intervals, addresses and other values represented by it. More formally, if $\text{rng}_1, \text{rng}_2 \in \text{Values}_{\text{abs}}$:

$$\text{rng}_1 \sqsubseteq_{\text{abv}} \text{rng}_2 \iff \forall c_1 \in \text{rng}_1. \exists c_2 \in \text{rng}_2 . \begin{cases} \text{PROJ}_{\text{val}}(c_1) \subseteq \text{PROJ}_{\text{val}}(c_2) & \text{if } \text{PROJ}_{\text{val}}(c_1) \in \text{Intervals} \\ \text{PROJ}_{\text{bas}}(c_1) = \text{PROJ}_{\text{bas}}(c_2) \land \text{PROJ}_{\text{off}}(c_1) \sqsubseteq_{\text{abv}} \text{PROJ}_{\text{off}}(c_2) & \text{if } \text{PROJ}_{\text{val}}(c_1) \in \text{StatVals} \cup \text{VectVals} \cup \text{HeapVals} \\ \text{PROJ}_{\text{val}}(c_1) = \text{PROJ}_{\text{val}}(c_2) & \text{otherwise} \end{cases}$$

so that for example

$$\{(\text{id}_1, [-10 \ldots -5], \text{asc}_1)\} \sqsubseteq_{\text{abv}} \{(\text{id}_2, [-\infty \ldots 0], \text{asc}_2)\}$$

$$\{(\text{id}_4, [-\infty \ldots 0], \text{asc}_4), (\text{id}_5, \text{fun mod foo}, \text{asc}_5)\} \sqsubseteq_{\text{abv}} \{(\text{id}_1, [-10 \ldots -5], \text{asc}_1)\}$$

Assuming that CompIds and Assocs are singleton sets, we can present the top and bottom elements of $\text{Values}_{\text{abs}}$:

- $T_{\text{abv}} = \{\}$
- $\bot_{\text{abv}} = \{\langle \text{id}_0, \text{val}, \emptyset \rangle \mid \text{val} \in \{[-\infty \ldots +\infty]\} \cup \text{Procs} \cup \text{Blocks} \cup \text{Addrvars} \cup \text{Memobjs} \land \text{off} = \{(\text{id}_0, [\text{id}_0, \text{id}_0, \ldots, \text{id}_0, \text{id}_0], \text{id}_0)\})\}$

where $\text{id}_0$ and the empty set are the only elements of CompIds and Assocs respectively, and Memobjs is the set of memory object identifiers:

$$\text{Memobjs} = \text{Vectors} \cup \text{Statics} \cup \text{Insns}$$

In practice, CompIds and Assocs are not singleton. Thus $\sqsubseteq_{\text{abv}}$ is not anti-symmetric and the abstract value ranges do not form a lattice. This does not matter, as long as the abstract contexts do; we will take care in defining the ordering of contexts, and the value ranges admissible in them, to ensure this is the case. We will continue using $T_{\text{abv}}$ and $\bot_{\text{abv}}$, relying on later adjustments to make their CompIds elements valid (see section 8.4.1).

8.3.3 Block Value Ranges

Having decided how to represent ranges of memory addresses, it is relatively easy to represent what is stored in them: we simply map each of the offset subcomponents of the address to a scalar range. In other words, a memory block value range consists of pairs of integer intervals and scalar ranges:

$$\text{Memvals}_{\text{abs}} = \mathcal{P}(\text{Intervals} \times \text{Values}_{\text{abs}})$$

We will need a function to return the contents of $m \in \text{Memvals}_{\text{abs}}$ for a given interval of indices $\text{ind} \in \text{Intervals}$, and a projection to extract all indices from $m$:

$$\text{CONTENTS}(m, \text{ind}) = \{c \mid \langle j, c \rangle \in m \land j \cap \text{ind} \neq \emptyset \}$$

$$\text{PROJ}_{\text{ind}}(m) = \{\text{ind} \mid \langle \text{ind}, \text{val} \rangle \in m\}$$
This representation is very flexible because of its adjustable accuracy: the offset range can be made detailed enough to represent each element separately, or generalised to the point where a single entry suffices. For example, a vector with five elements containing the values 1, 42, 10, 8, 9 can be represented in any of the following ways:

\[
\begin{align*}
&\langle[0 \ldots 0], [1 \ldots 1]\rangle, \\
&\langle[1 \ldots 1], [42 \ldots 42]\rangle, \\
&\langle[2 \ldots 2], [10 \ldots 10]\rangle, \\
&\langle[3 \ldots 3], [8 \ldots 8]\rangle, \\
&\langle[4 \ldots 4], [9 \ldots 9]\rangle, \\
&\{[0 \ldots 4], [1 \ldots 42]\} \\
&\{[0 \ldots 0], [1 \ldots 1]\rangle, \\
&\langle[1 \ldots 1], [42 \ldots 42]\rangle, \\
&\langle[2 \ldots 4], [8 \ldots 10]\rangle \\
&\langle[0 \ldots 4], [1 \ldots 1], [8 \ldots 10], [42 \ldots 42]\rangle \end{align*}
\]

8.3.4 Abstract Contexts

The Values_{abs} lattice is our abstract version of the operational semantics' Values lattice. The next step is to construct an abstract equivalent of the semantic environments, which hold the values of vregs and memory. The sets of vreg and memory abstract environments are, respectively,

\[
\begin{align*}
VEnv_{abs} &= Vregs \cup \{\bot\} \rightarrow Values_{abs} \\
MEnv_{abs} &= Memobjs \rightarrow Memvals_{abs}
\end{align*}
\]

and the set of abstract contexts underlying our interpretation is:

\[
\text{Contexts}_{abs} \subset VEnv_{abs} \times MEnv_{abs} \quad (8.1)
\]

Given a context \(C_{abs} = \langle A_v, A_m \rangle\), we will extract its environments by

\[
\begin{align*}
\text{PROJ}_V(C_{abs}) &= A_v \\
\text{PROJ}_M(C_{abs}) &= A_m
\end{align*}
\]

We are now almost able to rigorously define the abstract contexts lattice itself: we still need to define the order relation, the join and meet operations, and the significance of the Assocs and CompIds elements of value range components. We will deal with the latter first.

8.3.5 A Problem: Antisocial Value Ranges

Our approach so far has closely followed the static semantics, and, barring the addition of memory environments, is little more than a generalisation of the integer interval example of [CC77b] to non-integer value kinds. Useful results have been obtained even within such a restrictive framework, and one might expect that our more general formalism would be enough for most purposes. Unfortunately, this is not the case.

The problem is that, although values are shrink-wrapped in abstract contexts, there are no connections between them. Consider, for example, applying an analysis similar to ours to the code in figure 8.4 at the source code level: in both cases, we can express the fact that ‘a’ has a value in the range \([0 \ldots 10]\) when the conditional is reached, and in the case of fig. 8.4(a), this is enough to deduce that the argument of the call to ‘baz’ is zero. In the case of fig. 8.4(b), however, there is no way to convey the fact that ‘b’ is zero if and only if ‘a’ is, so the argument to ‘baz’ could have any value from 0 to 10 as far as the abstract interpreter is concerned.
Figure 8.4: Two more or less equivalent code fragments, demonstrating the need for value range associations.

```
(a) Obvious

{ 
  a <- @foo[]; // returns a value between 0 and 10
  .test [a > 0]
    @bar[a] // call 'bar' if a non-0
  .else
    @baz[a]; // call 'baz' if a is 0
}

(b) Circuitous

{ 
  a <- @foo[]; // returns a value between 0 and 10
  b <- -- -- a; // sets b to 0 if a is 0, 1 otherwise
  .test [b]
    @bar[a] // call 'bar' if a and b non-0
  .else
    @baz[a]; // call 'baz' if a and b are 0
}
```

The problem becomes even more pronounced when the analysis is applied at the intermediate code level, because translation introduces many temporary vregs and intermediate operations that obscure the link between operands and results; for instance, the P compiler compiles the two examples above almost identically, since it uses the `slt` (Set if Less Than) WIM3 operation to implement the conditional. We need a way of relating values to each other and of recovering information from these relations; in other words, we want values to be aware of the existence of other values and to “talk” to each other.

### 8.3.6 A Solution: Associations

The solution we have chosen makes use of the previously unspecified elements of Comps tuples to construct *associations* (or *assoc* for short) between individual range components of the value ranges in an abstract environment.

ComplIds is a set of value and offset range component identifiers; we can uniquely determine each component’s identifier from its value and the vreg or memory cell it is assigned to in its abstract context, and we assume the existence of a function to perform this mapping:

\[
\text{ComplId} \in \left( Vregs \cup \left( \text{Memobis} \times \text{Intervals} \right) \right) \times \text{Comps} \times \left( \text{Comps} \cup \{ \text{nilcomp} \} \right) \rightarrow \text{ComplIds}
\]

\(^4\text{This can be achieved by, for example, Gödel-numbering.}\)
takes a vreg or a memory block identifier with offset interval, a range component and an (optional) offset component, and returns the appropriate component identifier. For example, COMPID(v, c₁, c₂) is the identifier of component c₂ of the offset range of component c₁ in a value range for vreg v, while COMPID(⟨b, [2...4]), c, nil comp) is the identifier of component c of the value range for offsets 2 to 4 into memory block b.

This definition implies that we cannot allocate identifiers for components whose place in a context is not yet known, but we will gleefully allocate bogus identifiers and rely on fixup functions to legitimise them on assignment to a context; see section 8.4.1.

Assocs is the set

\[ \mathcal{P}(\text{CompIds} \times \{\text{exact, vague}\}) \]

so that an element of ASSOCs contains a collection of identifiers of components we wish to associate with, and an indication of whether the assoc is “exact” or “vague”.

The interpretation of vague assoc is simple. Given two value ranges r₁ and r₂ corresponding to vregs v₁ and v₂ respectively, if we can somehow ascertain that the value of v₁ actually lies in a particular component (c₁ say) of r₁, and that component contains a vague assoc to some component (c₂ say) of r₂, then the value of v₂ can be guaranteed to lie in c₂.

Exact assoc s are used to make even stronger assertions about integer intervals. Using the same notation as above, we require PROJₗ([c₁, c₂] = [a...b]) and PROJᵢ([c₉, c₁] = [c...d] with a, b, c, d ∈ \( \mathbb{Z} \cup \{\pm \infty\} \). If we can determine that the value of v₁ is actually [n...m], such that a ≤ n ≤ m ≤ b, then the existence of an exact assoc from c₁ to c₂ allows us to deduce that the value of v₂ is

\[
\text{AExact}([a...b], [c...d], [n...m]) = \begin{cases} 
[c - a + n...c - a + m] & \text{if } a \neq -\infty \\
[d - b + n...d - b + m] & \text{if } b \neq +\infty \\
[n...m] & \text{otherwise} 
\end{cases}
\]

We will present more formal definitions of the handling of assoc s when discussing the “restrict” and “split” operations (see sections 8.4.4-8.4.6) while the formation of assoc s will be described when we examine the “join” operation (section 8.4.3) and the abstract interpretation of WIM3 insns (chapter 9). For the moment, we should note that certain restrictions must be placed on the formation of assoc s to ensure that the results are meaningful. In the case of exact assoc s like the above, we require

\[
a = c \quad \text{if } -\infty \in [a, c] \\
b = d \quad \text{if } +\infty \in [b, d] \\
b - a = d - c \quad \text{otherwise}
\]

i.e. associated intervals must have the same size, or extend to infinity in the same direction(s); and for t ∈ {exact, vague}, we require

\[ \langle \text{id}_2, t \rangle \in \text{asc}_1 \iff \langle \text{id}_1, t \rangle \in \text{asc}_2 \]

i.e. all assoc s must be reciprocal.
8.4 Fundamental Operations on Abstract Contexts

8.4.1 Updating Component Identifiers

Some of the operations we have seen so far, and many of the ones we will encounter later, leave the identifiers of value range components in an inconsistent state. We will now present a way to restore consistency.

Given an abstract context $C_{abs} = (A_v, A_m)$, we define a mapping from old identifiers to new ones:

$$ \text{MAP}_{id} \in \text{Contexts}_{abs} \rightarrow \{\text{CompIds} \rightarrow \text{CompIds}\} $$

The mapping is defined as follows:

$$ \forall v \in Vregs . \quad (\forall c \in A_v[v]. \quad \text{MAP}_{id}(C_{abs})(\text{Proj}_{id}(c)) = \text{CompId}(v, c, \text{nilcomp})) \land $$

$$ (\forall c \in \text{PART}_{addr}(A_v[v]). \quad \forall c' \in \text{Proj}_{off}(c). \quad \text{MAP}_{id}(C_{abs})(\text{Proj}_{id}(c')) = \text{CompId}(v, c, c')) $$

$$ \forall b \in \text{Memobjs} . \quad (\forall i \in \text{Proj}_{inds}(A_m[b]). \quad (\forall c \in \text{CONTENTS}(A_m[b], i). \quad \text{MAP}_{id}(C_{abs})(\text{Proj}_{id}(c)) = \text{CompId}((b, i), c, \text{nilcomp})) \land $$

$$ (\forall c \in \text{PART}_{addr}((\text{CONTENTS}(A_m[b], i))). \quad \forall c' \in \text{Proj}_{off}(c). \quad \text{MAP}_{id}(C_{abs})(\text{Proj}_{id}(c')) = \text{CompId}((b, i), c, c')) $$

We can now translate the identifiers in an assoc set by

$$ \text{MAP}_{asc}(\text{map}, \text{asc}) = \{\text{id}(\text{map}, \text{t}) | (\text{id}, \text{t}) \in \text{asc}\} $$

and those in a value range by

$$ \text{MAP}_{abv}(\text{map}, \text{r}) = $$

$$ \{\langle \text{id}, \text{val}, \text{asc} \rangle \in \text{PART}_{ints}(\text{r}) \cup \text{PART}_{misc}(\text{r}) \land $$

$$ ((\text{id}, \langle \text{bas}, \text{off} \rangle, \text{asc}) \in \text{PART}_{addr}(\text{r}) \land \text{val} = \langle \text{bas}, \text{MAP}_{abv}(\text{map}, \text{off}) \rangle)\} $$

so the whole context can be updated using

$$ \text{UPDATEIDS}(C_{abs}, \text{MAP}_{id}(C_{abs})) $$

where

$$ \text{UPDATEIDS}(C_{abs}, \text{map}) = \langle \lambda v . \quad \text{MAP}_{abv}(\text{map}, A_v[v]), \lambda b . \quad \{\langle i, \text{MAP}_{abv}(\text{map}, \text{val}) | (i, \text{val}) \in A_m[b]\} $$

8.4.2 Rebuilding Associations

When processing a WIM3 insn, it is often convenient to consider the ranges of its arguments and results only. However, if any associs need to be transferred from arguments to results, the reciprocity condition of section 8.3.6 may require associs to be added to arbitrary value ranges in the current context. As we will see in the next section, the same may happen when a "restrict" operation is executed.
We could do this while processing insns and restricting contexts, but it is easier to restore assoc's after the event. We will define the ContAssocs operation, which adds reverse assoc's wherever necessary and deletes ones that reference defunct components, thus ensuring the consistency of the context.

We first define the set of all range components present in an abstract context $C_{abs} = \langle A_v, A_m \rangle$ as follows:

\[
\text{COMPONENTS}(A_v, A_m) = \text{COMPONENTS}_{\text{vreg}}(A_v) \cup \text{COMPONENTS}_{\text{mem}}(A_m)
\]

where the components of vreg and memory ranges are provided by

\[
\text{COMPONENTS}_{\text{vreg}}(A_v) = \bigcup_{v \in \text{Vregs}} \{ r \mid r = A_v(v) \lor (\text{id}, (\text{bas}, r), \text{asc}) \in A_v(v) \}
\]

\[
\text{COMPONENTS}_{\text{mem}}(A_m) = \bigcup_{b \in \text{Memobjs}} \{ r \mid (\text{int}, r) \in A_m(b) \lor (\text{int}, (\text{id}, \text{bas}, r), \text{asc}) \in A_m(b) \}
\]

We then define the assoc set for a component $c$ in the context by keeping those of its old assoc's referring to components still in the context, and adding reciprocal assoc's for any assoc's pointing to it from other components:

\[
\text{NEWASSOCs}(c, C_{abs}) = \{ \langle \text{PROJ}_{\text{id}}(d), t \rangle \mid d \in \text{COMPONENTS}(C_{abs}) \land \langle \text{PROJ}_{\text{id}}(d), t \rangle \in \text{PROJ}_{\text{asc}}(c) \lor \langle \text{PROJ}_{\text{id}}(c), t \rangle \in \text{PROJ}_{\text{asc}}(d) \}
\]

As well as adding assoc's, we may need to remove some: assoc's linking every one of a range's components to every component of some other range are redundant. (This is discussed further in the section on the restrict operator, 8.4.4.) Given ranges $r_1$ and $r_2$, the redundant assoc's to remove from the elements of $r_1$ are

\[
\text{REDASSOCs}(r_1, r_2) = \{ (\text{id}, t) \mid \text{id} \in \text{PROJ}_{\text{id}}(r_2) \land t \in \{\text{exact}, \text{vague}\} \land \forall c_1 \in r_1 . \forall c_2 \in r_2 . \langle \text{PROJ}_{\text{id}}(c_1), t \rangle \in \text{PROJ}_{\text{asc}}(c_2) \lor \langle \text{PROJ}_{\text{id}}(c_2), t \rangle \in \text{PROJ}_{\text{asc}}(c_1) \}
\]

and the range for $r_1$ in context $C_{abs}$ with redundant assoc's removed is

\[
\text{REDRANGE}(r_1, C_{abs}) = \{ (\text{id}, \text{val}, \text{asc} \setminus \text{red}) \mid (\text{id}, \text{val}, \text{asc}) \in r_1 \land \text{ranges} = \{ r_2 \mid (\exists v \in \text{Vregs} . A_v(v) = r_2) \lor (\exists b \in \text{Memobjs} . \exists i \in \text{PROJ}_{\text{inds}}(A_m(b))) . \text{CONTENTS}(A_m(b), i = r_2) \land r_2 \neq r_1 \land \text{red} = \bigcup_{r_2 \in \text{ranges}} \text{REDASSOCs}(r_1, r_2) \}
\]

Given a value range, we can now update the old assoc's:

\[
\text{RANGEASSOCs}(r, C_{abs}) = \text{REDRANGE}\{ (\langle \text{id}, \text{val}, \text{NEWASSOCs}(\text{asc}, C_{abs}) \rangle) \mid (\text{id}, \text{val}, \text{asc}) \in \text{PART}_{\text{int}}(r) \cup \text{PART}_{\text{misc}}(r) \cup (\langle \text{id}, \text{bas}, \text{RANGEASSOCs}(\text{off}, C_{abs}) \rangle, \text{NEWASSOCs}(\text{asc}, C_{abs})) \mid (\text{id}, \text{bas}, \text{off}, \text{asc}) \in \text{PART}_{\text{addr}}(r), C_{abs} \}
\]
(Note that the definition will not recur indefinitely, since offset ranges only contain integer interval components.) The whole context can then be updated by

\[
\text{CONTASSOS}(C_{\text{abs}}) = \\
(\lambda v . \text{RANGEASSOS}(A_v[v], C_{\text{abs}}), \\
\lambda b . (\langle \text{int}, \text{RANGEASSOS}(\text{val}, C_{\text{abs}}) \rangle | (\text{int}, \text{val}) \in A_m[b]))
\]

### 8.4.3 The Join Operator

The join operation computes the "union" of two abstract contexts; it produces an abstract context representing every concrete context representable by the join's operands. Unfortunately, producing a componentwise union of the operands' constituents is not enough. We need to remove overlapping components from the composite value ranges and add assocs between components originating in the same context. If \(C_{\text{abs}:1} = \langle A_v, A_m \rangle\) and \(C_{\text{abs}:2} = \langle B_v, B_m \rangle\), then

\[
C_{\text{abs}:1} \cup_{\text{abs}} C_{\text{abs}:2} = \text{CONTASSOS}(\langle C_v, C_m \rangle)
\]

where

\[
C_v[v] = \text{NORMALISE}[|\langle \text{id}, \text{val}, \text{asc} \cup \text{new} \rangle | \text{new} = |\langle \text{id}', \text{vague} \rangle | \\
|\langle \text{id}, \text{val}, \text{asc} \rangle \in A_v[v] \land \\
(\exists w \in \text{Vregs} . w \neq v \land \text{id}' \in \text{PROJids}(A_v[w])) \lor \\
(\exists b \in \text{Memobjs} . \exists i \in \text{PROJids}(A_m(b)) . \\
\text{id}' \in \text{PROJids}(\text{CONTENTS}(A_m(b), i)))]]] \\
\lor (\langle \text{id}, \text{val}, \text{asc} \rangle \in B_v[v] \land \\
(\exists w \in \text{Vregs} . w \neq v \land \text{id}' \in \text{PROJids}(A_v[w])) \lor \\
(\exists b \in \text{Memobjs} . \exists i \in \text{PROJids}(A_m(b)) . \\
\text{id}' \in \text{PROJids}(\text{CONTENTS}(B_m(b), i))))), v, \text{nilcomp})
\]

In other words, the new range for a vreg is computed by taking all components from its range in each of the original contexts, linking them to every component of other ranges in that context, then normalising the whole thing to reduce overlapping components. (The \text{NORMALISE} operation will be described in section 9.2.2.) The memory context, \(C_m\), is computed in a similar way.

Of course, this process implies a worst-case exponential blowup in the number of assocs at every join. In practice, things are not that bad, because many assocs are eliminated in \text{CONTASSOS} due to there being little difference in the values of the original contexts; much setting up and tearing down of redundant assocs can be circumvented in a practical implementation, and we have run into no problems even on fairly substantial test programs. However, a production compiler would need to check for situations where problems do occur, and use heuristics to obtain cheaper, less accurate results. Only introducing assocs for the ranges of vregs representing procedure arguments, or variables used in conditionals, would be a good start.

### 8.4.4 The Restrict Operator

The main reason given for the introduction of assocs between range components was that they allow the recovery of more accurate ranges for vregs and blocks in the current context. This is done by the restrict operator.
Consider a scalar value range \( r_1 \) and a subset of another range, \( r_2 \). The set of components of \( r_1 \) associated to components of \( r_2 \) is:

\[
\text{ASSOC}_{\text{plain}}(r_1, r_2) = \\
\{ c_1 | c_1 \in r_1 \land \exists c_2 \in r_2 \cdot (\text{PROJ}_{\text{id}}(c_2), t) \in \text{PROJ}_{\text{asc}}(c_1) \} \\
\cup \{ c_1 | c_1 \in r_1 \land \exists c_2 \in \text{PART}_{\text{addr}}(r_2) \cdot \\
\exists c'_2 \in \text{PROJ}_{\text{off}}(c_2) \cdot (\text{PROJ}_{\text{id}}(c'_2), t) \in \text{PROJ}_{\text{asc}}(c_1) \}
\]

The above takes care of “top-level” components of \( r_1 \) associated to components of \( r_2 \) or their offsets. The set of subcomponents of \( r_1 \) whose offsets are associated to components of \( r_2 \) is given by

\[
\text{ASSOC}_{\text{off}}(r_1, r_2) = \\
\{ (\text{PROJ}_{\text{id}}(c), (\text{PROJ}_{\text{bas}}(c), \text{ASSOC}_{\text{plain}}(\text{PROJ}_{\text{off}}(c), r_2)), \text{PROJ}_{\text{asc}}(c)) | c \in \text{PART}_{\text{addr}}(r_1) \}
\]

The “restrict” operation on value ranges is then defined by

\[
\text{RESTRICT}_{\text{abs}}(r_1, r_2) = \\
\begin{cases} 
    r_2 & \text{if } r_2 \subseteq r_1 \\
    r_1 & \text{if } \text{ASSOC}_{\text{plain}}(r_1, r_2) = \\
    \text{ASSOC}_{\text{off}}(r_1, r_2) & \text{otherwise}
\end{cases}
\]

so that if \( r_1 \) contains any components associated with components of \( r_2 \) (or their offsets), the result contains only those components (and only those offsets); if not, it consists of the whole range.

We can now extend this operation to contexts: if \( C_{\text{abs}} = \langle A_v, A_m \rangle \), we define

\[
\text{RESTRICT}_{\text{abs}}(C_{\text{abs}}, r_2) = \langle \lambda v . \text{RESTRICT}_{\text{abs}}(A_v, r_2), \\
\lambda b . (\langle i, \text{RESTRICT}_{\text{abs}}(r_1, r_2) \rangle | \langle i, r_1 \rangle \in A_m(b)) \rangle
\]

The following points should be noted:

- No attempt is made to cull assocs whose referents are not in the resulting ranges; this is handled by the \text{CONTASSOC} operation defined in section 8.4.2.

- The types of the assocs do not matter at this point, because the restrict operation never splits components. For that, we need the “split” operation, defined in the next section.

- Consider a pair of value ranges connected by assocs: the more assocs there are, the broader the range left over after a \text{RESTRICT}. Too many assocs between two ranges tell us as little as no assocs at all, and in some cases it may not be worth forming them in the first place. This should be kept in mind when examining the interpretation of WIM3 insns.

### 8.4.5 The Split Operator

The \text{RESTRICT}_{\text{abs}} operation selects existing components to be included in a new context, but does not alter the components themselves. Thus, it cannot exploit the additional information provided by exact assocs. The “split” operation described here takes a range and an integer, and transforms the former so that the latter does not fall “in the middle” of any of its components. If it has to
split one of the existing intervals to achieve this, it also follows its exact assocs
and splits all associated intervals in the context at the appropriate point, so
that exact assocs can be maintained.

Given the vreg range \( r \) and integer \( i \), let us find the component to be split, if any:

\[
\text{SPLITComp}(r, i) = \{ c | c \in \text{PARTint}(r) \land \text{PROJval}(c) = [a \ldots b] \land a < i < b \}
\]

Now we postulate the existence of identifier generator functions \( \text{LEFTId}(id) \)
and \( \text{RIGHTId}(id) \). Given a component identifier \( id \), these return two identi-
fiers guaranteed to be unique and not in the result range of \( \text{COMPId} \). We will
use them to generate temporary identifiers for components we split as we follow assocs; these will later be replaced with the appropriate identifiers via \( \text{UPDATEIds} \).

To split an arbitrary range \( r_2 \) in a context \( C_{abs} = \langle A_v, A_m \rangle \), given that another range \( r_1 \) is being split at the integer \( i \):

\[
\text{SPLITabv}(C_{abs}, r_1, r_2, i) = \text{PARTmsg}(r_2) \cup
\{ c | c \in \text{PARTint}(r_2) \land
\forall s \in \text{SPLITComp}(r_1, i) : \langle \text{PROJid}(s), \text{exact} \rangle \notin \text{PROJasc}(c) \} \cup
\langle \langle \text{LEFTId}(id), \text{AEXACT}([a \ldots b], \text{PROJval}(c), [a \ldots i - 1]), \text{asc} \rangle | c \in \text{PARTint}(r_2) \land s \in \text{SPLITComp}(r_1, i) \land
\langle s = c \lor (s, \text{exact}) \in \text{PROJasc}(c) \land \text{PROJval}(s) = [a \ldots b] \land
\text{asc} = \langle \langle \text{id}, t \rangle | \langle \text{id}', t \rangle \in \text{PROJasc}(c) \land ((t = \text{vague} \land \text{id} = \text{id}') \lor
(t = \text{exact} \land \text{id} = \text{LEFTId}('id'))) \rangle \cup
\langle \langle \text{RIGHTId}(id), \text{AEXACT}([a \ldots b], \text{PROJval}(c), [i \ldots b]), \text{asc} \rangle | c \in \text{PARTint}(r_2) \land s \in \text{SPLITComp}(r_1, i) \land
\langle s = c \lor (s, \text{exact}) \in \text{PROJasc}(c) \land \text{PROJval}(s) = [a \ldots b] \land
\text{asc} = \langle \langle \text{id}, t \rangle | \langle \text{id}', t \rangle \in \text{PROJasc}(c) \land ((t = \text{vague} \land \text{id} = \text{id}') \lor
(t = \text{exact} \land \text{id} = \text{RIGHTId}('id'))) \rangle \cup
\langle \langle \text{id}, \langle \text{bas}, \text{SPLITabv}(C_{abs}, r_1, \text{off}, i) \rangle, \text{asc} \rangle | \langle \text{id}, \langle \text{bas}, \text{off} \rangle, \text{asc} \rangle \in \text{PARTadd}(r_2) \}
\]

The first three lines of this definition handle non-interval components, as well
as intervals that do not need to be split. The next two groups of five lines
each handle the "left" and "right" parts of components with exact assocs to
the component in question; these collapse to empty sets if \( i \) is the lower limit of a component of \( r_1 \), or falls outside \( r_1 \), so that no split is necessary. The last
two lines split offset ranges recursively.

The whole context can then be transformed by

\[
\text{SPLITabv}(C_{abs}, r, i) = \text{UPDATEIds}(C_{abs}', \text{MAPid}(C_{abs}'))
\]

where

\[
C_{abs}' = \langle \lambda v . \text{SPLITabv}(C_{abs}, r, \lambda v (v), i), \lambda b . \langle \langle \text{int}, \text{SPLITabv}(C_{abs}, r, \text{val}, i) \rangle | \langle \text{int}, \text{val} \rangle \in A_m(b) \rangle \rangle
\]

and \( r \) is the range to be split at \( i \in Z \).

After a \( \text{SPLITabv} \) operation, we can use \( \text{CONTASSOCs} \) to rebuild the assocs that
are left dangling, and then apply \( \text{RESTRICT} \) to use these assocs in restricting the
context. We will see an application of this when we discuss the handling of conditional jumps in section 9.4.2.
8.4.6 Enhanced Split and Restrict Operators

In most cases, the $\text{Split}_{\text{abs}}$ operator is enough, but we sometimes need to split a context with respect to many ranges at many points simultaneously. For this purpose, we introduce a multiple split operator, which takes a context and a set as arguments. The set consists of ranges from the context and corresponding integers at which to split:

$$\text{Split}_{\text{abs}}^n(C_{\text{abs}}, \{\langle r_1, k_1 \rangle, \langle r_2, k_2 \rangle, \ldots, \langle r_n, k_n \rangle\}) = \text{Split}_{\text{abs}}(\text{Split}_{\text{abs}}^n(C_{\text{abs}}, \langle r_1, k_1 \rangle, \langle r_2, k_2 \rangle \ldots), r_n, k_n)$$

Next we need a way to extract the limits of a set of intervals:

$$\text{Low}(\text{val}) = \{m \mid [m \ldots n] \in \text{val}\}$$
$$\text{High}(\text{val}) = \{n \mid [m \ldots n] \in \text{val}\}$$

We will use the above to split a range $r_1$ in a context at every point where one of its components crosses the limits of some component of another range $r_2 \subseteq_{\text{abs}} r_1$. $r_2$ is not necessarily part of the context.

$$\text{SplitAll}(C_{\text{abs}}, r_1, r_2) = \text{Split}_{\text{abs}}^n(\text{Split}_{\text{abs}}^m(C_{\text{abs}}, \{\langle r_1, k \rangle \mid k \in \text{Low}(\text{PROJ}_{\text{vals}}(\text{PART}_{\text{ints}}(r_2))) \lor \text{Low}(\text{PROJ}_{\text{vals}}(\text{PART}_{\text{ints}}(r_2)))) \lor k - 1 \in \text{High}(\text{PROJ}_{\text{vals}}(\text{PART}_{\text{ints}}(r_2))))}, \{\langle r_1', k \rangle \mid c_1 \in \text{PART}_{\text{addr}}(r_1) \land c_2 \in \text{PART}_{\text{addr}}(r_2) \land \text{PROJ}_{\text{bas}}(c_1) = \text{PROJ}_{\text{bas}}(c_2) \land r_1' = \text{PROJ}_{\text{off}}(c_1) \land k \in \text{Low}(\text{PROJ}_{\text{vals}}(\text{PROJ}_{\text{off}}(c_2))) \lor k - 1 \in \text{High}(\text{PROJ}_{\text{vals}}(\text{PROJ}_{\text{off}}(c_2))))\}$$

In a context split in this way, $r_1$ and $r_2$ have no partially overlapping components, and it is easy to restrict $r_1$ and the context with respect to the values of $r_2$, even though $r_2$ carries no associ to the context:

$$\text{RestrictAll}(C_{\text{abs}}, r_1, r_2) = \text{Restrict}_{\text{abs}}(C_{\text{abs}}, r_2')$$

where

$$C_{\text{abs}}' = \text{SplitAll}(C_{\text{abs}}, r_1, r_2)$$
$$r_1' = \begin{cases} \text{PROJ}(C_{\text{abs}})(v) & \text{if } r_1 = \text{PROJ}(C_{\text{abs}})(v) \\ \text{CONTENT}(\text{PROJ}(C_{\text{abs}})(b), \text{ind}) & \text{if } \langle \text{ind}, r_1 \rangle \in \text{PROJ}(C_{\text{abs}})(b) \end{cases}$$
$$r_2' = \{c_1 \mid (c_1 \in r_1' \land (\exists c_2 \in \text{PART}_{\text{misc}}(r_2) \cdot \text{PROJ}_{\text{vals}}(c_1) = \text{PROJ}_{\text{vals}}(c_2)) \lor (\exists c_2 \in \text{PART}_{\text{ints}}(r_2) \cdot \text{PROJ}_{\text{vals}}(c_1) \subseteq \text{PROJ}_{\text{vals}}(c_2))) \lor(c \in \text{PART}_{\text{addr}}(r_2') \land c_1 = \text{PROJ}_{\text{ids}}(c), (\text{PROJ}_{\text{bas}}(c), \text{off}), \text{PROJ}_{\text{asc}}(c)) \land \text{off} = \{c' \mid c' \in \text{PROJ}_{\text{off}}(c) \land \text{PROJ}_{\text{vals}}(c') \cap \text{PROJ}_{\text{vals}}(\text{PROJ}_{\text{off}}(c)) \neq \emptyset\}
$$

8.4.7 Ordering the Abstract Contexts Lattice

Ordering the abstract contexts by componentwise application of the $\subseteq_{\text{abs}}$ relation is not sufficient, which is why we have avoided looking at the order relation until now. We need a way to capture the fact that an abstract context containing associ is “smaller” (in that it represents fewer concrete contexts) than one with no associ. We have such a facility in the $\text{Restrict}_{\text{abs}}$ operation.

Given two contexts, $C_{\text{abs}}=\langle A_v, A_m \rangle$ and $C_{\text{abs}}'=\langle B_v, B_m \rangle$, we first define their ordering when they are restricted according to a single range component
To show that our abstract interpretation is consistent with the concrete semantics, we need to establish a mapping between them and prove some properties of this mapping, as outlined in section 7.4.4. The mapping is specified by the abstraction, $\alpha : \text{Contexts} \rightarrow \text{Contexts}_{\text{abs}}$, and concretisation, $\gamma : \text{Contexts}_{\text{abs}} \rightarrow \text{Contexts}$.
γ ∈ Contexts_{abs} → Contexts functions, and the time has come to define such functions for our lattice. This will turn out not to be as painless as we might imagine.

8.5.2 Concretisation

The definition of the concretisation function is straightforward, but will ultimately cause the most trouble. We first define a value range concretisation, γ_{abs} ∈ Values_{abs} → 2[Values], which maps every range component to the values it could possibly represent:

\[
γ_{abs}(r) = \begin{cases}
\int & \exists c ∈ \text{PART}_{\text{int}}(r) . \text{int} ∈ \text{PROJ}_{\text{val}}(c) ∪ \\
\text{ADDR}_{\text{bb}}(b) & \exists c ∈ r . b = \text{PROJ}_{\text{val}}(c) ∧ b ∈ \text{Blocks} ∪ \\
\text{ADDR}_{\text{fun}}(f) & \exists c ∈ r . f = \text{PROJ}_{\text{val}}(c) ∧ f ∈ \text{Procs} ∪ \\
\text{ADDR}_{\text{reg}}(v) & \exists c ∈ r . v = \text{PROJ}_{\text{val}}(c) ∧ v ∈ \text{Addrvars} ∪ \\
\text{ADDR}_{\text{str}}(s) +_{\text{val}} \text{int} & \exists c ∈ r . \langle s, \text{off} \rangle = \text{PROJ}_{\text{val}}(c) ∧ s ∈ \text{Statics} ∧ \text{int} ∈ d ∈ \text{PROJ}_{\text{val}}(\text{off}) ∪ \\
\text{addr} +_{\text{val}} \text{int} & \exists c ∈ r . \langle i, \text{off} \rangle = \text{PROJ}_{\text{val}}(c) ∧ i ∈ \text{Insns} ∧ \text{int} ∈ d ∈ \text{PROJ}_{\text{val}}(\text{off}) ∧ \text{addr} ∈ \text{ADDR}_{\text{heap}}(i) ∪ \\
\text{addr} +_{\text{val}} \text{offset}_{\text{vec}}(v) +_{\text{val}} \text{int} & \exists (E_v, E_m) ∈ \text{States}_{\text{init}} . \text{addr} = E_v(\text{PcsSp}) ∧ \exists c ∈ r . \langle v, \text{off} \rangle = \text{PROJ}_{\text{val}}(c) ∧ v ∈ \text{Vectors} ∧ \text{int} ∈ d ∈ \text{PROJ}_{\text{val}}(\text{off})
\end{cases}
\]

where States_{\text{init}} is the set of possible initial states for the procedure being interpreted (see section 7.4.2.6), PcsSp is the vreg holding the stack pointer, and

\[\text{ADDR}_{\text{heap}} ∈ \text{Insns} → 2[\text{Values}]\]

is a function returning the base addresses of all heap blocks that could conceivably be allocated at a given insn.

A simple version of the complete γ consists of a straightforward application of γ_{abs} to everything in the context, but we can do better than that by constraining the results based on the assocs present in the abstract context. We first need a way to break down interval and offset components into individual integers:

\[\text{INDIVIDUAL}(c) = \{\langle \text{id}_0, \text{val}, \{\} \rangle | \]

\[\langle \text{PROJ}_{\text{val}}(c) ∈ \text{Intervals} ∧ n ∈ \text{PROJ}_{\text{val}}(c) ∧ \text{val} = [n \ldots n] \rangle ∨ \]

\[\langle \text{PROJ}_{\text{val}}(c) ∈ \text{StatVals} ∪ \text{VecVals} ∪ \text{HeapVals} ∧ n ∈ d ∈ \text{PROJ}_{\text{val}}(\text{PROJ}_{\text{off}}(c)) ∧ \text{val} = (\text{PROJ}_{\text{bas}}(c), \langle \text{id}_0, [n \ldots n], \{\} \rangle ) \rangle ∨ \]

\[\langle \text{PROJ}_{\text{val}}(c) ∈ \text{Procs} ∪ \text{Blocks} ∪ \text{Addrvars} ∧ \text{val} = \text{PROJ}_{\text{val}}(c) \rangle \]

where \text{id}_0 represents a component identifier whose value does not matter, because it will be replaced in the context concretisation, given by:

\[\gamma(C_{abs}) = \{\langle E_v, E_m \rangle | \]

\[\forall c ∈ \text{COMPONENTS}(C_{abs}) . \forall c' ∈ \text{INDIVIDUAL}(c) . \]

\[c' = \text{REST}_{\text{ALL}}(C_{abs}, \{c, \{c'\}\}) ∧ \]

\[\langle \forall v ∈ \text{Vregs} . E_v(v) ∈ γ_{abs}(\text{PROJ}_{\text{v}}(C_{abs})(v)) \rangle ∧ \]

\[\forall m ∈ \text{Values} . \]

\[\text{REPLACE}(\langle \text{id}_0, \text{val}, \{\} \rangle) \]
This fairly scary definition is actually quite simple in concept:

- Line 8.2: We take each range component present in the abstract environment in turn, and subdivide it into components with single integers or offsets.
- Line 8.3: We restrict the context assuming that the subcomponent selected previously holds the actual value of the object its range represents.
- Line 8.4: We deal with vreg environments by assigning every bitpattern represented by a vreg’s range to some concretised environment.
- Lines 8.5–8.7: We enumerate all the possible memory address ranges with single integer offsets and concretise them to get the corresponding addresses.
- Line 8.8: We assign contents to the addresses in concrete environments by concretising the contents of corresponding address ranges in the restricted abstract context.
- Lines 8.9–8.10: For vregs that have addresses, we allow memory environments with the appropriate values in the corresponding memory locations.

### 8.5.3 Abstraction

The abstraction function is trickier to define: Concrete values contain no hints of their intended meanings, and given such a value, we have no indication of how the abstract interpreter will choose to view its abstract equivalent (i.e., which value kind it will choose to assign it to). The best we can do is to form a range containing components of all the kinds the concrete value could possibly match:

\[
\alpha_{abv}(r) = \begin{cases} 
(\{[b], [a \ldots a], \},) & | a \in x \} \cup \\
(\{[b], [a \ldots a], \},) & | b \in Blocks \land ADDR_{lab}(b) \in x \} \cup \\
(\{[f], [a \ldots a], \},) & | f \in Proc \land ADDR_{fun}(f) \in x \} \cup \\
(\{[v], [a \ldots a], \},) & | v \in Addrvars \land ADDR_{vreg}(v) \in x \} \cup \\
(\{[s], [a \ldots a], \},) & | s \in Statics \land ADDR_{str}(s) + val a \in x \} \cup \\
(\{[i], [a \ldots a], \},) & | i \in Insns \land ADDR_{heap}(i) + val a \in x \} \cup \\
(\{[v], [a \ldots a], \},) & | v \in Vectors \land \exists(E_v, E_m) \in States_{init} . \\
& E_v(Proc) + val OffsetVec(v) + val a \in x \\
\end{cases}
\]

We then collect all these components in defining the abstraction of a concrete context C:

\[
\alpha(C) = \bigcup_{(E_v, E_m) \in C} (\lambda v . \text{NORMALISE}[\alpha_{abv}[E_v(v)], v, \text{nilcomp}], \\
\lambda b . \{[a \ldots a], v | r = \text{NORMALISE}[\alpha_{abv}[E_m(m)], \\
& m = \gamma_{abv}([[[i], [a \ldots a], \}, \}, \},)])], \\
& [b, [a \ldots a], \text{nilcomp}])}
\]
There is no problem with this to a certain extent—at worst, it will give useless but safe results—but if we try to abstract the concretisation of an abstract value, we will find that the result will be greater than what we started with, thus violating condition 7.2 on page 89.

The root of the problem is that, due to the overlap of value kinds, our AIN does not conform to the “Very Reasonable Assumption” presented in section 5.1 of [CC97]; namely, that there is a unique minimal element of Contexts$_{abs}$ approximating any given element of Contexts. There are a number of things we can try to get around this:

1. We could redefine the ordering $\subseteq_{abv}$ of value ranges, so that $r_1 \subseteq_{abv} r_2$ if any member of $\gamma_{abv}[r_1]$ could possibly be less than any member of $\gamma_{abv}[r_2]$. This would introduce a relational link between virtually any pair of ranges, make the ordering impossible to determine at compile time, and probably violate the partially ordered nature of Contexts$_{abs}$ by allowing

$$r_1 \subseteq_{abv} r_2 \land r_2 \subseteq_{abv} r_1 \land r_1 \neq r_2$$

to hold in some cases.

2. We could augment the concrete values lattice with tags for each value, indicating its intended interpretation. The resulting system would be rather similar to a typed semantics, but would not cause execution to fail on a tag mismatch; it would set the tag to “unknown” and carry on operating on the bitpatterns as usual. It would then be easy to map the combination of tag and value to an appropriate value kind when abstracting; “unknown” tags would result in $T_{abv}$ value ranges.

This is an elegant solution, and would be our first choice if we were theoreticians, but rewriting the operational semantics (which the implementation does not use), just so that the mapping between them and our abstract lattice becomes neater, seems pointless to us as compiler writers. Besides, we find the current, bare-bones, machine-level concrete semantics more pleasing aesthetically.

3. The classical solution put forth in [CC97] introduces new elements into Contexts$_{abs}$ so that every concrete context does have a minimum abstract equivalent. In practice, this would mean introducing new value kinds to represent values that are valid e.g. both as integers and as pointers to vectors, and so on.

Besides the proliferation of value kinds, which can only complicate our implementation, this approach is not consistent with some of our goals: It is only rarely that programmers use values interpretable as members of more than one value kind at the same time, and we would rather issue a warning and fall back on safe approximations when this happens, than cheerfully and silently handle operations probably resulting from programming errors. To clean up the semantics, this approach would bloat our compiler and reduce its usefulness.

Rather than doing any of the above, we will keep the mappings we have already. Noting that

$$\alpha(\gamma(x)) = \alpha(\gamma(\alpha(x)))$$

even though

$$x \neq \alpha(\gamma(x))$$
we claim that, as long as \( \alpha \) and \( \gamma \) are order-preserving and obey equations 7.3 and 7.4, they will be sufficient to guarantee the consistency of our interpretation.

### 8.6 Widening and Narrowing Operators

#### 8.6.1 Introduction

As explained in section 7.4.5, widening is necessary when the abstract lattice contains infinite ascending chains. In our case, long ascending chains occur due to the presence of integer intervals: there are a manageable number of procedures and vreg addresses in any program, but the possible integer intervals (and memory locations, which arise from a finite number of base addresses but have integer offsets) are many. Accordingly, we will define widening and narrowing operators for ranges consisting of integer intervals only and then extend them to the full lattice.

#### 8.6.2 Widening and Narrowing of Integer Intervals

Let \( r \in \text{Offsets} \) be a value range such that \( r \neq \emptyset \), and every component \( c \in r \) has \( \text{PRO} \text{val}(c) \in \text{Intervals} \). We can obtain the minimum and maximum (by the usual \((\leq)\) ordering of the integers) of the whole range:

\[
\begin{align*}
\min(r) &= \min\{a \mid c \in r \land \text{PRO} \text{val}(c) = [a \ldots b]\} \\
\max(r) &= \max\{b \mid c \in r \land \text{PRO} \text{val}(c) = [a \ldots b]\}
\end{align*}
\]

The widening of two integer ranges is then defined as follows:

\[
\begin{align*}
\tau_1 \oplus \tau_2 &= \{[\text{id}_{\text{new}}, [w_a \ldots w_b], \text{INEXACT}[\text{PRO}\text{ASC}(\tau_2)]]\}
\end{align*}
\]

where

\[
\begin{align*}
w_a &= \begin{cases} 
-\infty & \text{if } \min(\tau_2) < \min(\tau_1) \\
\min(\tau_1) & \text{otherwise}
\end{cases} \\
w_b &= \begin{cases} 
+\infty & \text{if } \max(\tau_2) > \max(\tau_1) \\
\max(\tau_1) & \text{otherwise}
\end{cases}
\end{align*}
\]

and

\[
\text{INEXACT}(\text{ASC}) = \{[\text{id}, \text{vague}] \mid ([\text{id}, t]) \in \text{ASC}\}
\]

In other words, all gaps in the ranges are discarded, and if the right-hand operand then extends past the limits of the left-hand one, the corresponding limits of the result range are set to infinity. All associés carried by the original components are inherited, but rendered inexact. \( \text{id}_{\text{new}} \) is an identifier guaranteed to be unique and never before used; it will later be replaced with the appropriate component identifier when the range is placed in a context. Narrowing is similar:

\[
\begin{align*}
\tau_1 \cap \tau_2 &= \{[\text{id}_{\text{new}}, [n_a \ldots n_b], \text{INEXACT}[\text{PRO}\text{ASC}(\tau_2)]]\}
\end{align*}
\]

where
Of course, discarding the gaps between intervals is not always necessary in a narrowing operation; however, since the ranges involved will already have been subjected to a widening interpretation, gaps are unlikely to exist, and the added complexity required to preserve them would be of little benefit.

It is obvious from the definition that \( r_1 \uparrow \downarrow_{\text{int}} r_2 \) consists of a single interval that contains all the integers contained in the intervals of \( r_1 \) and \( r_2 \); thus,

\[
(r_1 \cup r_2) \subseteq_{\text{abv}} (r_1 \downarrow \downarrow_{\text{int}} r_2)
\]

satisfying one of the constraints on the behaviour of widening operators presented in section 7.4.5. The other constraint is also satisfied, since there are only two directions in which a range may be extended once all gaps have been removed, and all such extensions lead to \( \pm \infty \) allowing no further widening, so all increasing chains of successively widened integer ranges are variations of the following general pattern (for the sake of clarity, identifiers and assocs are not shown):

\[
\begin{array}{c}
[a + n_1 \ldots b - n_2] \quad [a - n_3 \ldots b - n_4] \quad [l_3 \ldots b + n_6]
\end{array}
\]

\[
\begin{array}{c}
\downarrow \\
[a \ldots i_1] \quad [i_2 \ldots b] \\
\quad \quad [a \ldots b] \quad [\infty \ldots b] \quad [\infty \ldots \infty]
\end{array}
\]

where all \( i \in \mathbb{Z} \), all \( n \in \mathbb{N} \), and successive entries in the strictly increasing sequence of the lower line are formed by widening together the two ranges in the preceding column. The required properties of the narrowing operator can be proved similarly.

### 8.6.3 Widening and Narrowing of Scalar Ranges

Given value ranges \( r_1 \) and \( r_2 \), we first consider their memory address components, and find the memory bases that are only present in one of the ranges:

\[
\mathit{PART}_{\text{bases}}(r_1, r_2) = \{ c \mid \\
\quad (c \in \mathit{PART}_{\text{addr}}(r_1) \land \\
\quad \neg (d \in \mathit{PART}_{\text{addr}}(r_2) \land \mathit{PROJ}_{\text{bas}}(c) = \mathit{PROJ}_{\text{bas}}(d))) \lor \\
\quad (c \in \mathit{PART}_{\text{addr}}(r_2) \land \\
\quad \neg (d \in \mathit{PART}_{\text{addr}}(r_1) \land \mathit{PROJ}_{\text{bas}}(c) = \mathit{PROJ}_{\text{bas}}(d))) \}
\]

We then join this value with the remaining values, widened as appropriate:

\[
r_1 \uparrow \downarrow_{\text{abv}} r_2 = \\
\mathit{PART}_{\text{misc}}(r_1) \cup \mathit{PART}_{\text{misc}}(r_2) \cup (\mathit{PART}_{\text{int}}(r_1) \uparrow \downarrow_{\text{int}} \mathit{PART}_{\text{int}}(r_2)) \cup \\
\{ (\text{id}_{\text{new}}, (d, \mathit{PROJOFF}(c_1) \downarrow \downarrow_{\text{int}} \mathit{PROJOFF}(c_2), \text{asc}) ) \mid \\
\quad c_1 \in \mathit{PART}_{\text{addr}}(r_1) \land c_2 \in \mathit{PART}_{\text{addr}}(r_2) \land \\
\quad d = \mathit{PROJ}_{\text{bas}}(c_1) = \mathit{PROJ}_{\text{bas}}(c_2) \land \\
\quad \text{asc} = \text{INEXACT}(\mathit{PROJ}_{\text{asc}}(c_2)) \}
\]

\[
\cup \mathit{PART}_{\text{bases}}(r_1, r_2)
\]
8.6.  WIDENING AND NARROWING OPERATORS

i.e. integer intervals are widened as before, memory addresses with the same base have their offsets widened, and all other components are left untouched.

The required properties follow from those for integer ranges: since integers and offsets are handled as before, the only way new increasing chains can form is by

1. adding procedure, label or vreg address components, or static components with new base addresses; but there are only a finite number of these in any given program, and they will eventually run out.

2. adding vector components with new base addresses; but since we do not distinguish between different incarnations of a vector in recursive calls, and there are only a finite number of vector declarations in a program, these will run out too.

3. adding dynamic (heap) storage components with new base addresses; but we do not distinguish between heap blocks allocated by different executions of the same insn, and there are a finite number of insns in a program, so this is not a problem either.

The narrowing operator $\triangle_{abv}$ is defined in the same way, using $\triangle_{int}$ instead of $\nabla_{int}$ wherever necessary.

8.6.4  Widening and Narrowing of Memory Contents

Since all memory blocks are finite in size, it suffices to apply scalar widening and narrowing to each of their components respectively. First, given a set of integer intervals $\mathbf{int} \subseteq \text{Intervals}$, we need a way to remove overlaps from it.

We define a flattening operation to extract the set of integers contained in the intervals of $\mathbf{int}$:

$$\text{Flatten}[\mathbf{int}] = \{i \mid i \in c \in \mathbf{int}\}$$

The minimal set of intervals covering a set of integers is given by

$$\text{Continuous}[\mathbf{S}] = \{[a \ldots b] \mid (\forall c . a \leq c \leq b \Rightarrow c \in \mathbf{S}) \land
\left[(a = -\infty \lor a - 1 \notin \mathbf{S}) \land (b = +\infty \lor b + 1 \notin \mathbf{S})\right]\}
$$

Thus overlaps can be removed from $\mathbf{int}$ by $\text{Continuous}[\text{Flatten}[\mathbf{int}]]$.

Given memory ranges $m_1, m_2 \in \text{Memvals}_{abv}$, we can now use the above to construct a set of appropriate offsets for their widening:

$$m_1 \nabla_{mem} m_2 =
\{(\text{ind}, \text{rng}) \mid \text{ind} \in \text{Continuous}(\text{Flatten}(\text{Proj}_{\text{inds}}(m_1) \cup \text{Proj}_{\text{inds}}(m_2))) \land
\text{rng} = \text{Contents}(m_1, \text{ind}) \nabla_{abv} \text{Contents}(m_2, \text{ind})\}
$$

As usual, we define the narrowing operator $\Delta_{mem}$ by replacing $\nabla_{abv}$ with $\triangle_{abv}$ in the definition of $\nabla_{mem}$.

8.6.5  Widening and Narrowing of Abstract Contexts

Applying the widening operators to contexts in a componentwise fashion is not enough to widen the contexts themselves: all the assoc pointing to components that have been modified would be left dangling, and since the components containing them may also have been modified, $\text{ContAssoc}$ would not be capable of restoring them.
What we need is a mapping from old to new component identifiers that can be fed to $\text{UPDATE\text{IDS}}$. For contexts $C_{\text{abs}:1} = \langle A_v, A_m \rangle$, $C_{\text{abs}:2} = \langle B_v, B_m \rangle$, we define

$$\text{MAP}_\text{wid} \in \text{Contexts}_{\text{abs}} \times \text{Contexts}_{\text{abs}} \rightarrow \{\text{Comp\text{Ids}} \rightarrow \text{Comp\text{Ids}}\}$$

as follows:

$$\forall v \in \text{V\text{regs}}.$$  

$$\text{PART}_{\text{ints}}(A_v(v)) \sqcap \text{PART}_{\text{ints}}(B_v(v)) \ = \ \{d\} \land \forall c \in \text{PART}_{\text{ints}}(A_v(v)) \land \text{MAP}_\text{wid}(C_{\text{abs}:1}, C_{\text{abs}:2})(\text{PROJ}_\text{id}(c)) = \text{COMP\text{Id}}(v, d, \text{nil\text{comp}}) \land \forall c_1 \in \text{PART}_{\text{adds}}(A_v(v)) \land \forall c_2 \in \text{PART}_{\text{adds}}(B_v(v)) \land \text{PROJ}_\text{abs}(c_1) \neq \text{PROJ}_\text{abs}(c_2) \lor |\text{PROJ}_{\text{off}}(c_1) \sqcap \text{PROJ}_{\text{off}}(c_2) = \{d\} \land \forall c' \in \text{PROJ}_{\text{off}}(c_1) \land \text{MAP}_\text{wid}(C_{\text{abs}:1}, C_{\text{abs}:2})(\text{PROJ}_\text{id}(c')) = \text{COMP\text{Id}}(v, c_1, d)$$

and similarly for the memory block component of the widened context; any identifiers not handled above are mapped to themselves.

Then the widening operator for the abstract contexts lattice is:

$$C_{\text{abs}:1} \sqcap \text{abs}C_{\text{abs}:2} = \text{CONT\text{ASSOC}}(\text{UPDATE\text{IDS}})\text{\{(\langle A_v, A_m(v) \sqcap \text{abs}B_v(v) \rangle, \langle A_m(b) \sqcap \text{abs}B_m(b) \rangle), \text{MAP}_\text{wid}(C_{\text{abs}:1}, C_{\text{abs}:2})\})$$

The mapping operator for context narrowing is defined in a similar way.

### 8.6.6 Remarks

Widening approximates a concrete semantics in which values can be incremented indefinitely, without wrapping round. It is fortunate that we have defined $P$ to be compatible with this (see section 2.3.2), but we should not think of it as a compromise. It is a portability feature: standard-conforming $P$ programs cannot rely on any particular word size\(^5\). If wrapping is necessary, it can be implemented explicitly. The compiler is perfectly capable of optimising the extra code to oblivion if the word size is appropriate; if not, it will still produce correct (albeit slightly slower) code.

It may be noted that non-wrapping semantics are not without precedent: the Transputer architecture [1NM88] always traps overflowing arithmetic, allowing the word size to be increased without changing the instruction set.

Another interesting point concerns the maintenance of integer intervals in ranges. In section 8.6.2 we noted that all gaps between the intervals of integer ranges are removed by the widening operator, leaving one large interval. One might ask whether there is anything to gain by allowing separate intervals in any context, since interesting programs will probably contain a loop where widening is needed; the answer is that the increased accuracy of interpretation between, and within, loops is often worth the extra effort.

Consider, for example, the program of figure 8.5 on the next page: using our representation, the range of 'a' before the test at point (1) contains the intervals $[-10 \ldots -10]$ and $[+10 \ldots +10]$, so after the test we know its exact value on entry to each of the loops, and can deduce that it takes values from $[-20 \ldots -10]$ and $[20 \ldots 10]$ in the two loop bodies respectively. Had we used only a single interval, we would have a value of $[-10 \ldots +10]$ for 'a' before the test, and could only deduce values in the ranges $[-20 \ldots -1] \text{ and } [0 \ldots 20]$ for the two loop bodies.

\(^{5}\)Anyone who has ported legacy code to any of the emerging 64-bit architectures will surely agree that this is a Good Thing.
8.7 Implementation Details

8.7.1 Introduction

The mathematical framework described above translates fairly easily into a practical implementation. This section discusses the data structures used to represent the framework in the P compiler, and outlines some issues arising in the implementation of the basic operations on them.

8.7.2 Scalar Value Ranges

Scalar value ranges are linked lists of data structures that represent range components. Integer intervals are nearest the head of the list, ordered by lower bound. Heap, dynamic and static memory addresses follow, ordered by identifier within each kind. When components are added to a range, care is taken to ensure that only one memory address component of any given kind and identifier exists by joining offset ranges where appropriate. Procedure, label and vreg addresses follow. This ordering helps optimise linear searches arising from inclusion operators in the formal definitions.

An empty list indicates the \( \bot_{abv} \) range, and there is a special value kind to represent the \( T_{abv} \) range which would take too many components to represent conventionally.

We should mention the realisation, only made with the benefit of hindsight, that fixed size arrays would have been a better choice for the value range data structure. The abstract interpreter spends much of its time chasing pointers around: arrays would eliminate this, while reducing the memory allocation overhead and improving reference locality (and thus cache performance). Since value ranges rarely have more than three or four components, the memory waste would be small.

8.7.3 Memory Block Value Ranges

Memory value ranges consist of lists of integer intervals to which scalar value ranges are attached. The intervals represent offsets into the block, and the values are those of the memory contents. A linked list of intervals is necessary because the representation of the block’s contents may vary in accuracy during interpretation (see section 8.3.3) and offset intervals may have to be inserted or deleted.
8.7.4 Range Tables

A range table is the compiler’s equivalent of an abstract context. Like the span tables used in register assignment (see section 6.5), these are small hash tables with a fixed number of buckets, each a linked list of entries for vregs and memory blocks. Each entry contains the vreg or block identifier, an indication of which of the two it is, and a pointer to the corresponding value range.

8.7.5 Associations

Assoc s are implemented as linked lists hanging off range components. Each entry in an assoc list contains an indication of whether it is exact or vague, a pointer to the associated component, and the vreg to whose context value that component belongs.

Component identifiers as such do not exist, since pointers can identify components uniquely; the vreg identifier in an assoc is simply an aid to the abstract interpreter, which would otherwise have to search through the entire context in some cases (see, for example, the definition of NewAssocs in section 8.4.2). 

Assocs are symmetric and transitive, so that if component a is associated to b which is associated to c, each of the three will have associations to the other two. To avoid the expensive allocation and deallocation of list entries, each entry also contains an “in use” bit; unused entries may be reused, or garbage-collected when the value range containing them is discarded.

As with value ranges, an array implementation would probably have been more efficient, although components sometimes acquire large numbers of assocs and some mechanism for extending assoc arrays would need to be provided.

8.7.6 Operations

We will not discuss the implementation of the basic abstract context operations, because most of them consist of complicated but uninteresting pointer-chasing. It should be noted, though, that the clean, structured approach taken in the formal definitions is not always beneficial in implementations; indeed, some of the formal concepts become irrelevant. The following are some of the issues with which we had to contend:

- Most of the convoluted operations needed to guarantee component identifier consistency vanish, because every data object has an implicit unique identifier—its address in memory. UpdateIds, CompId, LeftId, RightId etc. do not have equivalents in the compiler.

- On the other hand, assocs can be a pain to implement: set inclusion operators (⊂) typically translate to searches through linear lists, and in order to minimise these, we have to do assoc generation “on the fly”. Rather than post-processing each context with an implementation of ContAssocs, we ensure that every new assoc pointer is symmetric by following it and checking that an equivalent assoc exists in its target.

- The compiler works entirely in the abstract domain and does not need implementations of the α and γ operators, which is why we have breezed over the difficulties of defining them.

- Integer interval arithmetic can subtly go wrong when cross-compiling: if the host machine has a larger word size than the target, we can handle
results that would actually overflow and cause undefined behaviour at run time. To avoid this, the host has to simulate the target’s word size in arithmetic and check that the results are not too large. Fortunately, these checks can be stripped away as dead code when the host and target word sizes are the same.
Chapter 9

Abstract Interpretation of WIM3 Code

9.1 Introduction and Overview

Given an inaccurate approximation to the values of vregs and memory at every insn, the abstract context vector transition function delivers a better guess. In this section, we discuss processes which, taken together, constitute the transition function for our interpretation of intermediate code. After a brief overview of the whole process, we will describe the AIN engine's actions on entry to each basic block, and the interpretation of individual WIM3 insns.

In a practical implementation, neither the complete context vector nor the transition function are likely to be available at any point: the tradeoffs between storage space and recomputation cost which we have already encountered in dataflow analyses also apply here. We can only keep a few (two or three, as we will see later) range tables for each basic block, and will be forced to recompute the rest as needed.

Similarly, the transition function has to be broken down into a large number of steps, one for each insn. Each step receives an abstract context from its predecessor(s), modifies it as appropriate, and passes it on to the next step. This is done by traversing the flowgraph in preorder; the traversal is repeated until the results converge, then narrowing is substituted for widening and the flowgraph is again traversed repeatedly until convergence is achieved.

9.2 Insn Interpretation: Preliminaries

9.2.1 Obtaining Ranges of WIM3 Arguments

In section 7.4.2.4 we saw how to extract the value of a referenced operand from the insn containing it and the dynamic state in which it is executed; we will now show how to do the same for the abstract semantics. The function

$$\text{Arv}_{\text{abs}}(C_{\text{abs}}, i, n) \in \text{Contexts}_{\text{abs}} \times \text{Insns} \times \{1, 2, 3, J\} \rightarrow \text{Values}_{\text{abs}}$$

gives the abstract value of the $n$th argument of insn $i$ executed in a context $C_{\text{abs}} = (\lambda_v, \lambda_m)$, assuming that the argument is meant to be referenced and does not specify a vreg to be defined. The value is obtained from the context, if the argument is a vreg, or constructed from scratch if it is a constant. Note
the provision of a definite zero offset for block address constants:

\[
\text{ArgV}_{\text{abs}}(\langle A_v, A_m \rangle, i, n) = \\
\begin{cases}
\{\langle d_0, |\text{Arg}_v(i, n) \ldots |\text{Arg}_v(i, n) \rangle, [] \} \quad & \text{if ArgT(i, n) = con} \\
\{A_v | \text{Arg}_v(i, n) \} \quad & \text{if ArgT(i, n) = reg} \\
\{\langle d_0, \{0 \ldots 0 \}, [] \rangle, [] \} \quad & \text{if ArgT(i, n) \in \{str, vec\}} \\
\{\langle d_0, |\text{Arg}_v(i, n) \rangle \} \quad & \text{if ArgT(i, n) \in \{adr, bbl, fun\}}
\end{cases}
\]

9.2.2 Normalisation of Insn Results

The results of most complicated insns are obtained as a collection of value ranges, which are then joined together. This can result in ranges containing integer intervals or offsets that overlap, or multiple address components with the same base. For the sake of clarity and implementation efficiency, we would like to minimise this proliferation of range components; hence the need for a normalisation operator to reduce ranges to some canonical form.

The method we use works as follows:

- Non-integer, non-address components differing only in their associs are merged.
- Integer interval components that do not overlap with other components of the range are retained as they are.
- Intervals that do overlap are reduced to a single interval corresponding to their union; all associs of the originals are maintained, but any that are exact are converted to vague.
- Address components with the same base are merged; their offset ranges are joined and normalised as above.

More precisely, we define the set of non-overlapping integer components of a range \(r\):

\[
\text{Distinct}(r) = \{c \mid c \in \text{PartInts}(r) \land \\
\forall d \in \text{PartInts}(r) \cdot (d = c \lor \text{ProjVal}(c) \cap \text{ProjVal}(d) = \{\})
\]

We then recall the definition of \text{Continuous} from section 8.6.4 and normalise a range \(r\) destined for vreg (or memory entry) \(v\) in some context by applying:

\[
\text{Normalise}(r, v, c_0) = \text{Distinct}(r) \cup \\
\{(\text{CompId}(v, \{d_0, \text{val, asc}\}, c_0), \text{val, asc}) \mid \\
\text{asc} = [a \mid (\text{id}, \text{val}, a) \in \text{Part Misc}(r)]\} \\
\cup \{(\text{CompId}(v, \{d_0, \text{int, asc}\}, c_0), \text{int, asc}) \mid \\
\text{int} \not\in \text{ProjVal}(\text{Distinct}(r) \land \text{int} \in \text{Continuous}
\left(\bigcup_{i, v} v\right) \land \\
\text{asc} = [\langle \text{id}, \text{vague} \rangle \mid (\text{id}, t) \in \text{ProjAsc}(c) \land \\
\text{c} \in \text{PartInts}(r) \land \text{ProjVal}(c) \cap \text{int} \neq \{\}]\} \\
\cup \{c \mid c = (\text{CompId}(v, \{d_0, \text{bas, off}, \text{asc}\}, c_0), \{\text{bas, off}, \text{asc}\}) \land \\
\text{off} \rightarrow \text{Normalise}(\left(\bigcup_{i, v} c', v, c\right) \land \text{asc} - \left(\bigcup_{i, v} a\right))
\}
\]

The following points should be obvious from the above:
Like the `RangeAssocs` operation of section 8.4.2, normalisation cannot recur indefinitely, since offset ranges consist of integer intervals only.

For any range \( r, r \subseteq abv \) `Normalise(r, v, nilcomp)` (for appropriate values of \( v \)).

The normal form is unique; its integer components correspond to the continuous subsets of the componentwise intersection of all intervals in the original range.

Normalisation reduces the amount of information recoverable through assocs. It also leaves the context containing the normalised range in an inconsistent state, with dangling and unreciprocated assocs; this can be fixed using the `ContAssocs` operation of section 8.4.2.

### 9.2.3 Storing Results of WIM3 Insns

All arithmetic and logic insns, as well as the `load` and `larg` (load from memory) insns, update the context only by storing a result in the vreg given as their first argument. Rather than replicating the environment update for each of these insns, we describe it here in terms of a function `Resulti(Cabs, (ip, ic))` which returns the result to be stored in `Argi(ic, 1)` and will be defined, little by little, as we describe the interpretation of insns in detail.

Thus, consider \( i \in \text{Insns} \) such that

\[
\text{Op}_i(i) \in \{\text{add, sub, mul, div, mod, lsh, rsh, ash, and, orr, nor, xor, load, larg}\}
\]

Let \( a_p, a_s \in \text{Arcs} \) be the arcs to and from \( i \) respectively: the latter must be unique, since \( i \) is not a jump or procedure return, and the former can be regarded as unique without loss of generality, since we can add no-ops at the start of any basic block. If \( C_{abs} \) is an abstract context vector such that \( C_{abs}(a_p) = (A_v, A_m) \), we can define the entry for \( a_s \) in the context vector after one step of abstract interpretation by

\[
\text{Nextabs}(C_{abs})(a_s) = \text{ContAssocs}([\text{UpdateIDS}(C_{abs}, \text{Map}_{id}(C_{abs})])]
\]

where \( C_{abs} \) is defined using the substitution notation to update the vreg environment in \( C_{abs}(a_p) \), leaving the memory environment intact:

\[
C_{abs} = (A_v, \text{Normalise} \text{Resulti}(C_{abs}, a_p), \text{Argi}(i, 1), \text{nilcomp})/[\text{Argi}(i, 1)], A_m)
\]

### 9.2.4 Block Entry

The contexts on all arcs leading to a block need to be joined together to produce the context in which the block's first insn executes. This will also be the point where widening is performed, if necessary. Assuming \( a = (i_j, i_s) \) is the arc leading into the block from a notional "joining" insn \( i_j \), we have

\[
\text{Nextabs}(C_{abs})(a) = \bigcup_{i_p \in \text{Pred}_i(i_j)} C_{abs}(i_p, i_j)
\]

if no back edges lead to the block, and

\[
\text{Nextabs}(C_{abs})(a) = C_{abs}(a) \nabla C_{abs}(\bigcup_{i_p \in \text{Pred}_i(i_j)} C_{abs}(i_p, i_j))
\]
if the block is at the head of a back edge.

Having dealt with block entries, we can assume that each insn has a single predecessor from now on.

9.3 Arithmetic and Logic Insns

9.3.1 Addition and Subtraction

Addition and subtraction are the simplest arithmetic instructions to implement, and also the most fundamental, since they are meaningful for most kinds of value. The non-commutativity of subtraction makes definitions simpler, so we will start by defining an abstract value subtraction operator, \(-_{abv}\).

First of all, we note that the results of subtractions involving labels, procedures and addresses of vregs are highly implementation-dependent and cannot be expressed in our scheme. We will err on the side of caution as far as they are concerned:

\[
\begin{align*}
\tau_1 -_{abv} \tau_2 &= \\
\quad \top_{abv} \quad \text{if } \text{PART}_{misc}(\tau_1) \cup \text{PART}_{misc}(\tau_2) \neq \{\}
\end{align*}
\]

There are also some severe restrictions on the subtraction of memory addresses: if the subtrahend contains a memory address component, the minuend should consist of just one such component, with the same base. If it does not, or if either operand contains more than one memory address component, the result could be any value:

\[
\begin{align*}
\tau_1 -_{abv} \tau_2 &= \\
\quad \top_{abv} \quad \text{if } (|\text{PART}_{addr}(\tau_2)| > 1) \lor \\
& \quad (|\text{PART}_{addr}(\tau_1)| > 1 \land \text{PART}_{addr}(\tau_2) \neq \{\}) \lor \\
& \quad |\text{PART}_{addr}(\tau_1)| = |c_1| \land \text{PART}_{addr}(\tau_2) = |c_2| \land \\
& \quad \text{PROJ}_{bas}(c_1) \neq \text{PROJ}_{bas}(c_2)
\end{align*}
\]  

(9.1)

We now define a trivial interval subtraction operator, \(-_{int}\), which will be useful in what follows:

\[
[a \ldots b] -_{int} [c \ldots d] = [a - b - c]
\]

We distinguish between two kinds of subtractions: those whose subtrahend is a known constant, and all the rest. The former are simpler, but offer greater opportunities for assoc generation; we will handle them first.

\[
\begin{align*}
\tau_1 -_{abv} \tau_2 &= \\
\quad (\langle \text{id}_0, \text{val}_1 \rangle -_{int} [a \ldots a], \\
& \quad \text{asc}_1 \cup \{\langle \text{id}_1, \text{exact} \rangle \} \\
& \quad \langle \text{id}_1, \text{val}_1, \text{asc}_1 \rangle \in \text{PART}_{int}[\tau_1] \rangle \cup \\
& \quad (\langle \text{id}_0, \langle \text{bas}_1, \text{off}_1 -_{abv} \tau_2, \text{asc}_1 \rangle \rangle \\
& \quad \langle \text{id}_1, \langle \text{bas}_1, \text{off}_1 \rangle, \text{asc}_1 \rangle \in \text{PART}_{addr}[\tau_1] \rangle \quad \text{if } \tau_2 = (\langle \text{id}_2, \langle a \ldots a, \text{asc}_2 \rangle \rangle)
\end{align*}
\]

This takes care of numeric constant subtrahends; note that each component of the result obtains a copy of the assoc of the minuend’s corresponding component, and an exact assoc to that component itself. For address constants, we require the base addresses of minuend and subtrahend to be the same, and apply the previous clause of the definition to the offsets:

\[
\begin{align*}
\tau_1 -_{abv} \tau_2 &= \\
\quad (\langle \text{id}_0, \text{off}_1 -_{abv} \text{off}_2, \text{asc}_1 \rangle \rangle \quad \text{if } \tau_1 = (\langle \text{id}_1, \langle \text{bas}_1, \text{off}_1 \rangle, \text{asc}_1 \rangle) \land \\
& \quad \tau_2 = (\langle \text{id}_2, \langle \text{bas}_1, \text{off}_2 \rangle, \text{asc}_2 \rangle) \land \\
& \quad \text{PROJ}_{val_{bas}}(\text{off}_2) = [a \ldots a]
\end{align*}
\]
Note that in all cases so far, the resulting range is normalised if the operands are. This is not true for subtractions with two non-constant operands: the result of such operations “spread out”, reducing the gaps between components or even introducing overlaps. Consider, for example,

$$([-5 \ldots -2], [0 \ldots 7], [10 \ldots 13]) -_{abv} ([3 \ldots 6]) = \{[-11 \ldots -5], [-6 \ldots 4], [4 \ldots 10]\}$$

(Component identifiers and assos have been omitted for clarity.) This result must be normalised to thwart the proliferation of overlapping components, giving

$$\text{NORMALISE}([-11 \ldots -5], [-6 \ldots 4], [4 \ldots 10]) = \{[-11 \ldots 10]\}$$

Thus the necessity of normalisation (incorporated in the definition of $\text{NEXT}_{abv}$) implies a reduction in the number of components. This in turn indicates that forming assos for the result is probably not worth the effort. (See also the notes in section 8.4.4.)

Ignoring assos, we form the result by a componentwise subtraction of

- intervals from intervals, giving intervals;
- intervals from addresses, giving addresses; and
- addresses from addresses, giving intervals.

$$r_1 -_{abv} r_2 =$$

$$\{\langle id_0, \text{val}_1 -_{\text{int}} \text{val}_2, []; [] \rangle |$$

$$\langle id_1, \text{val}_1, \text{asc}_1 \rangle \in \text{PART}_{\text{int}}(r_1) \land \langle id_2, \text{val}_2, \text{asc}_2 \rangle \in \text{PART}_{\text{int}}(r_2) \} \cup$$

$$\{\langle id_0, \langle \{id_0, \text{val}_1 -_{\text{int}} \text{val}_2, []; [] \} \rangle, []; [] \rangle |$$

$$\langle id_1, \langle \{\text{bas}_1, \text{off}_1\}, \text{asc}_1 \rangle \in \text{PART}_{\text{addr}}(r_1) \land \langle id'_1, \text{val}_1, \text{asc}'_1 \rangle \in \text{off}_1 \land$$

$$\langle id_2, \text{val}_2, \text{asc}_2 \rangle \in \text{PART}_{\text{int}}(r_2) \} \cup$$

$$\{\langle id_0, \text{val}_1 -_{\text{int}} \text{val}_2, []; [] \rangle |$$

$$\langle id_1, \langle \{\text{bas}_1, \text{off}_1\}, \text{asc}_1 \rangle \in \text{PART}_{\text{addr}}(r_1) \land \langle id'_1, \text{val}_1, \text{asc}'_1 \rangle \in \text{off}_1 \land$$

$$\langle id_2, \langle \text{bas}_2, \text{off}_2\rangle, \text{asc}_2 \rangle \in \text{PART}_{\text{addr}}(r_2) \land \langle id'_2, \text{val}_2, \text{asc}'_2 \rangle \in \text{off}_2 \}$$

Note that, when subtracting addresses from addresses, we know that each operand has only one memory address component, and they both have the same base; other cases will have been caught by equation 9.1.

The abstract addition operator, $+_abv$, is defined similarly; the only differences are:

- Because of commutativity, the non-symmetric clauses of the definition have to be duplicated for both operands.
- Addition of addresses to addresses returns a $T_{abv}$ result.

Using the $+_abv$ and $-_{abv}$ operators, we can define the abstract result of an add or subtract insn. There is one special case to be taken care of: vector addresses can only enter the program’s data space via instructions of the form

$$\text{add dest}, \$SP, \text{vec}$$

where $\$SP$ is the stack pointer, $\text{vec}$ is the vector identifier, and dest the destination register. Our abstract interpretation does not keep track of the stack pointer’s value; it is always assumed to be zero. This is safe as long as instructions like the above are handled specially, because the stack pointer is
invisible in all other cases (details of the call sequence are encapsulated in the call, enter and return insns and are not visible at the WIM3 level). Thus, given an insn $i_c$ with $\text{Op}_1(i_c) = \text{add}$:

$$\text{RESULT}_1(\text{C}_{\text{abs}}(i_p, i_c)) =$$

$$\begin{cases} 
\{(\text{id}_0, \text{id}_1, [0 \ldots 0], \text{id}_1)\} & \text{if } \text{Arg}_1(i_c, 2) = \text{PcsSp} \land \text{Arg}_1(i_c, 3) = \text{v} \\
\text{Arg}_\text{abs}(\text{C}_{\text{abs}}, i_c, 2) +_{\text{abv}} \text{Arg}_\text{abs}(\text{C}_{\text{abs}}, i_c, 3) & \text{otherwise}
\end{cases}$$

where $\text{C}_{\text{abs}} = \text{C}_{\text{abs}}(i_p, i_c)$.

Obtaining the result of a subtraction is more straightforward:

$$\text{RESULT}_1(\text{C}_{\text{abs}}, (i_p, i_c)) = \text{Arg}_\text{abs}(\text{C}_{\text{abs}}, i_c, 2) -_{\text{abv}} \text{Arg}_\text{abs}(\text{C}_{\text{abs}}, i_c, 3) \quad \text{if } \text{Op}_1(i_c) = \text{sub}$$

### 9.3.2 Multiplication and Division

Multiplication and division are simpler than subtraction and addition since they can only be applied meaningfully to integer intervals, but they do have some complicating features:

- If the operands have different signs, the result is always negative; if they do not, it is always positive. This means that in some cases, interval boundaries must be exchanged to maintain order, and intervals that straddle the zero point have to be split.

- Integer division rounds to zero; this means that even if the dividend is normalised and the divisor is a constant, the result may contain overlapping components.

- Divisions by zero must be confronted.

First let us dispose of the cases we cannot handle:

$$r_1 \div_{\text{abv}} r_2 = T_{\text{abv}} \quad \text{if } r_1 \neq T_{\text{ints}} \lor r_2 \neq T_{\text{ints}}$$

where $\div_{\text{abv}}$ is our abstract value division operator.

Next, let us find a way to extract the positive or negative components of an integer range. First we devise a way to split such a range into components less than, and greater than or equal to, any given integer $n$:

$$\text{PART}_{\geq n}(r) = \{(\text{id}, [c \ldots b], \text{asc}') | (\text{id}, [a \ldots b], \text{asc}) \in r \land ((a \geq 0 \land b \geq 0 \land c = n \land \text{asc}' = \text{INEXACT}(\text{asc})) \lor (a < 0 \land b > 0 \land c = 0 \land \text{asc}' = \text{INEXACT}(\text{asc})))$$

$$\text{PART}_{< n}(r) = \{(\text{id}, [a \ldots c], \text{asc}') | (\text{id}, [a \ldots b], \text{asc}) \in r \land ((b < 0 \land b = c \land \text{asc}' = \text{asc}) \lor (b < 0 \land a < 0 \land c = -1 \land \text{asc}' = \text{INEXACT}(\text{asc})))$$

Note that the above also converts assoc in the partitioned range components (but not the ones that are not split) to the vague form. Also note that identifiers are reused for split components; this does not matter, since wherever these functions are used, we either make sure that no components are split, or repair the identifiers later. (The latter is also true of the following operations.)

Now, the positive and negative partitions of a range $r$ are given by $\text{PART}_{\geq n}(r)$ and $\text{PART}_{< n}(r)$ respectively, and the result of a division is the union of four
subranges, one for each possible combination of operand signs, not forgetting to convert all assocs to vague ones. (The result will probably contain many overlapping components, but these will be taken care of by the normalisation inherent in \(N_{\text{abs}}\).)

\[
\begin{align*}
\tau_1 \div_{abv} \tau_2 &= \{ \langle d_2, [a \div d \ldots b \div c], \text{INEXACT}(a \cup c) \rangle \mid \\
&\quad \langle \langle d_1, [a \ldots b], c \rangle \in \text{PART}_2(\tau_1) \land \langle \langle d_2, [c \ldots d], c \rangle \in \text{PART}_2(\tau_2) \rangle \lor \\
&\quad \langle \langle d_1, [b \ldots a], c \rangle \in \text{PART}_0(\tau_1) \land \langle \langle d_2, [c \ldots d], c \rangle \in \text{PART}_0(\tau_2) \rangle \lor \\
&\quad \langle \langle d_1, [a \ldots b], c \rangle \in \text{PART}_0(\tau_1) \land \langle \langle d_2, [d \ldots c], c \rangle \in \text{PART}_0(\tau_2) \rangle \lor \\
&\quad \langle \langle d_1, [b \ldots a], c \rangle \in \text{PART}_0(\tau_1) \land \langle \langle d_2, [d \ldots c], c \rangle \in \text{PART}_0(\tau_2) \rangle \}
\end{align*}
\]

where \(\div\) is the integer division operator, modified so that divisions by zero act like divisions by one (i.e., \(a \div 0 = a\)).

The above statement may seem preposterous at first glance: mathematicians might, with a little grumbling and as long as \(a \neq 0\), allow us to set the result of such an operation to infinity, since infinite values are part of our lattice; but \(a \div 0 = a\) is incorrect, no matter how one looks at it.

However, we should keep in mind that dividing by zero will result in a runtime error and the termination of the program; our analysis assumes that this does not happen, and can ignore zero divisors (after issuing an appropriate warning, as we will see in the next chapter). Thanks to the partitioning operators, zero can only occur at the beginning of an interval of the divisor range; if we ignore it, the interval will vanish (if it consists of zero alone) or shrink so that its first element is 1. In the latter case, our result will be correct; in the former, it will simply add a spurious value to the result range, which does no harm since our analysis is inexact in any case.

Given the abstract division operator, the result of a division insn can now be defined:

\[
\begin{align*}
\text{RESULT}(C_{\text{abs}}, \langle t_p, t_c \rangle) &= \\
\text{ARC}V_{\text{abs}}(C_{\text{abs}}, t_c, 2) \div_{abv} \text{ARC}V_{\text{abs}}(C_{\text{abs}}, t_c, 3) &\text{if } O_{\text{int}}[i] = \text{div}
\end{align*}
\]

where \(C_{\text{abs}} = C_{\text{abs}}(\langle t_p, t_c \rangle)\). Multiplication is handled in a similar way.

It should be noted that the assocs surviving a multiplication or division operation are not likely to be very useful, and are not actually maintained in the implementation; but when multiplying by a constant, it may be worth forming assocs between result and multiplicand, just as we did in the case of subtraction, except that these assocs will be vague rather than exact since the two ranges have differently sized components.

### 9.3.3 Shifts

Arithmetic shifts are mostly equivalent to multiplications and divisions, and can be handled as such. There is the same restriction to numeric operands: if \(\triangleright\triangleright_{abv}\) is the abstract arithmetic right shift operator.

\[
\begin{align*}
\tau_1 \triangleright\triangleright_{abv} \tau_2 &= T_{abv} \quad \text{if } \tau_1 \neq \text{PART}_{\text{int}}(\tau_1) \lor \tau_2 \neq \text{PART}_{\text{int}}(\tau_2)
\end{align*}
\]

Taking advantage of the facts that shift amounts must be positive and that shifts by more than the word size (in bits) give constant results, we can convert a range of shift offsets into a multiplier or divisor:

\[
\begin{align*}
\tau_1 \triangleright\triangleright_{abv} \tau_2 &= \\
\tau_1 \div_{abv} \{ \langle d_0, [2^a \ldots 2^a], \emptyset \rangle \mid a \in C_{\text{provals}}(\tau_2) \} \cup \\
\{ \langle d_0, [-1 \ldots -1], \emptyset \rangle \mid (\exists n \in C_1 \in C_{\text{provals}}(\text{PART}_{\text{int}}(\tau_1))) \land \\
\exists a \in C_2 \in C_{\text{provals}}(\tau_2). -n \leq 2^a - 1 \}
\end{align*}
\]
Note that overflowing shifts of negative numbers are the only case differing from division: the result on overflow is an all-0 or all-1 word, and the latter is equivalent to −1, assuming a 2's-complement system.

The logical shifts are similar, with the following differences:

- In the left shift, multiplication is used instead of division, but shifts that can move non-zero bits into the sign bit must be treated specially.
- If the value to be shifted is positive, logical and arithmetic shifts to the right have the same effects; if not, the result of the logical shift will be either equal to that of the arithmetic shift or positive, giving us a reasonable approximation.
- 0 is the only possible result for logical shifts that move all bits of the left-hand operand out of the result.

The results of shift insns are formed in the same way as those of divisions, with the appropriate shift operator substituted for $\div_{abv}$ in the definition of $\text{Result}_1$.

### 9.3.4 Bitwise Logical Operations

These correspond to and, or, xor, nor and xor insns. (WIM3 does not have a negation insn, and uses nor with 0 where this is needed.) With one exception mentioned later, we only handle logical operations on integers, returning $T_{abv}$ for other value kinds, just as for the shift operations; so for example, with $\vee_{abv}$ as the inclusive disjunction operator, we have

$$r_1 \vee_{abv} r_2 = T_{abv} \quad \text{if } r_1 \neq \text{PART}_{\text{int}}(r_1) \lor r_2 \neq \text{PART}_{\text{int}}(r_2)$$

A reasonable approximation to disjunction for ranges of positive integers ($r_1 = \text{PART}_{\leq 0}(r_1)$) is:

$$r_1 \vee_{abv} r_2 = \{|id_0, [0 \ldots 2^a - 1], \emptyset \} \mid \forall n \in c \in (r_1 \cup r_2) . 2^a > n |$$

Other cases are similar.

The only non-integer case worth handling is that of conjunction between addresses and certain integers; this is a programmer's trick sometimes used to align byte addresses to the nearest word boundary. Since the run-time system ensures that all memory blocks start on a word boundary, the alignment of arbitrary addresses reduces to alignment of their offsets:

$$r \land_{abv} R_N = \{|id, \langle \text{bas}, \text{off} \land_{abv} R_N \rangle, \text{asc} \} \mid (id, \langle \text{bas}, \text{off} \rangle, \text{asc}) \in r |$$

where $R_N$ is a range containing a single integer which happens to be the negation of $2^W - 1$, $W$ being the target architecture's word size.

### 9.4 Conditional and Jump Insns

#### 9.4.1 Set On Less Than

The $\text{slt}$ insn sets the vreg indicated by its first argument to 1 (or 0) if the value of the second argument is (or, respectively, is not) less than that of the third. The easy way to handle this is to set the result's value range to $[0\ldots 1]$, but since $\text{slt}$ is used to implement the relational operators, this will ruin any
chance of restricting the values of the original arguments on a subsequent conditional jump, as described in section 8.3.5.

The solution is to form associs between the result and operand ranges, and in what follows, SltOp denotes a function partly responsible for the creation of these associs; it takes a context and the ranges of its insn’s arguments and returns an updated context where ranges have been subdivided in preparation for the insertion of associs.

Let us handle some simple special cases first:

- Comparisons involving labels, or procedure/vreg addresses, are unlikely to be useful since these values may change from one compilation to the next:

\[
\text{SltOp}(C_{abs}, r_1, r_2) = C_{abs} \quad \text{if } \text{Part}_{misc}(r_1) \cup \text{Part}_{misc}(r_2) \neq \emptyset
\]

- Comparisons of ranges containing both integers and addresses are not handled either:

\[
\text{SltOp}(C_{abs}, r_1, r_2) = C_{abs} \quad \text{if } \emptyset \notin \{\text{Part}_{ints}(r_1), \text{Part}_{addr}(r_1)\} \vee
\emptyset \notin \{\text{Part}_{ints}(r_2), \text{Part}_{addr}(r_2)\}
\]

- Comparisons between address ranges can only be handled accurately if the addresses have a single, common base:

\[
\text{SltOp}(C_{abs}, r_1, r_2) = C_{abs} \quad \text{if } \exists b. \quad \text{Part}_{addr}(r_1) = [c_1] \wedge
\text{Part}_{addr}(r_2) = [c_2] \wedge
\text{Proj}_{base}(c_1) = \text{Proj}_{base}(c_2)
\]

- In other cases, a comparison between addresses is the same as a comparison between their offset ranges, so if

\[
\text{Part}_{addr}(r_1) = [c_1] \wedge \text{Part}_{addr}(r_2) = [c_2]
\]

then

\[
\text{SltOp}(C_{abs}, r_1, r_2) = \text{SltOp}(C_{abs}, \text{Proj}_{off}(c_1), \text{Proj}_{off}(c_2)) \tag{9.2}
\]

- In some cases, the results of a comparison are constant at compile time and no associs need to be formed:

\[
\text{SltOp}(C_{abs}, r_1, r_2) = C_{abs} \quad \text{if } \max(r_1) < \min(r_2) \vee \min(r_1) \geq \max(r_2)
\]

with min, max extended to integer ranges as in section 8.6.2.

Using Result_{slt} to denote the result of a _slt insn, in all the above cases except the last one we have

\[
\text{Result}_{slt}(r_1, r_2) = \{[id_0, [0...0], []], [id_1, [1...1], []]\}
\]

(the reason we do not merge the two components will become apparent later) while the results corresponding to the last case are

\[
\text{Result}_{slt}(r_1, r_2) =
\begin{cases}
\{[id_0, [1...1], []]\} & \text{if } \max(r_1) < \min(r_2) \\
\{[id_1, [0...0], []]\} & \text{if } \min(r_1) \geq \max(r_2)
\end{cases}
\]
In the remaining general case, the two operand ranges will consist of integer intervals and will overlap partially. We need to split each operand into three parts (up to two of which may be empty), at the minimum and maximum of the other operand, using the \texttt{Splitabs} operator defined in section 8.4.5:

\[
\text{SLTOP}(C_{\text{abs}}, r_1, r_2) = \text{CONTASSOC}(\text{SLTOP}(C_{\text{abs}}, r_1, r_2), \text{CONTASSOC}(\text{SLTOP}(C_{\text{abs}}, r_1, \min(r_2))), \text{CONTASSOC}(\text{SLTOP}(C_{\text{abs}}, r_1, \max(r_2))), \text{CONTASSOC}(\text{SLTOP}(C_{\text{abs}}, r_2, \min(r_1))), \text{CONTASSOC}(\text{SLTOP}(C_{\text{abs}}, r_2, \max(r_1))))
\]

We can now associate parts of the operands with appropriate result values: the “middle” portions of each operand (lying between the limits of the other operand) will bear assocs to both possible result values, while the low/high parts will be associated with the 1/0 (left-hand operand) or 0/1 (right-hand operand) result values respectively. In doing this, we can use the \texttt{P_{ART_{<n}}} and \texttt{P_{ART_{\geq n}}} functions safely, knowing that they will not cause any range to be split further:

\[
\text{RESULT}_{\text{slt}}(v, r_1, r_2) =
\begin{cases}
\langle \langle \text{COMPID}(v, \{id_0, [1 \ldots 1], \text{asc}, \text{nilcomp}), [1 \ldots 1], \text{asc} \rangle, \text{id}, \text{vague} \rangle \mid
\langle \exists c \in \text{P_{ART_{<n}}}(r_2) \{r_1\} . \text{id} = \text{PROJ}_{id}(c) \lor \langle \text{id}, t \rangle \in \text{PROJ}_{\text{asc}}(c) \rangle \lor
\langle \exists c \in \text{P_{ART_{\geq n}}}(r_1) \{r_2\} . \text{id} = \text{PROJ}_{id}(c) \lor \langle \text{id}, t \rangle \in \text{PROJ}_{\text{asc}}(c) \rangle \rangle \\
\langle \langle \text{COMPID}(v, \{id_0, [0 \ldots 0], \text{asc}, \text{nilcomp}), [0 \ldots 0], \text{asc} \rangle, \text{id}, \text{vague} \rangle \mid
\langle \exists c \in \text{P_{ART_{<n}}}(r_1) \{r_2\} . \text{id} = \text{PROJ}_{id}(c) \lor \langle \text{id}, t \rangle \in \text{PROJ}_{\text{asc}}(c) \rangle \lor
\langle \exists c \in \text{P_{ART_{\geq n}}}(r_1) \{r_2\} . \text{id} = \text{PROJ}_{id}(c) \lor \langle \text{id}, t \rangle \in \text{PROJ}_{\text{asc}}(c) \rangle \rangle
\end{cases}
\]

The context \(C_{\text{abs}}\) is then updated by

\[
\text{NEXT}_{\text{abs}}(C_{\text{abs}})(a_s) =
\text{CONTASSOC}(\langle a'_v, A'_m \rangle | \text{RESULT}_{\text{slt}}(\text{ARG}[i, 1], \text{ARGV}_{\text{abs}}(C'_{\text{abs}}, i, 2), \text{ARGV}_{\text{abs}}(C'_{\text{abs}}, i, 3) | (\text{ARG}[i, 1], A'_m))
\]

where

\[
C'_{\text{abs}} = \langle A'_v, A'_m \rangle = \text{SLTOP}(C_{\text{abs}}(a_p), \text{ARGV}_{\text{abs}}(C_{\text{abs}}(a_p), i, 2),
\text{ARGV}_{\text{abs}}(C_{\text{abs}}(a_p), i, 3))
\]

and \(a_p, a_s \in \text{Arcs}\) are the arcs leading to/from the insn \(i\), as usual.

### 9.4.2 Jump if Less Than

Jump insns do not modify the contents of vregs and memory locations, but our inferences as to what these contents are may change as a result of a conditional jump. In interpreting such an insn, our purpose is to generate two new abstract contexts—one for each arc leaving the insn—from the single context on entry to it.
A conditional jump can be regarded as a combination of two operations, namely the conditional test and the jump. In the case of the jlt (Jump if Less Than) insn, the test is similar to the alt insn we have just described; the difference is that, instead of assigning the result to a vreg, we use it to process the jump and then discard it.

In a static analysis, processing a jump is equivalent to deciding which parts of the context correspond to each of the possible test results, and assigning them to the two new contexts for the jump destinations. We have already seen how to do this; it is the Restrict abs operation.

Consider $a_p, a_t, a_f \in \text{ARCS}; a_p$ is the arc leading to a jlt insn $i$, $a_t$ represents a branch to a successor of $i$ on which the conditional is true, and $a_f$ a "false" branch\(^1\). Let $v_0$ be a fresh vreg identifier; we will use it to simulate a alt insn, inserting it into the context temporarily and removing it once we have obtained the results. First, let us split the original context:

$$C'_{abs} = \langle A'_{t}, A'_{m} \rangle = \text{SLTOp}(C_{abs}(a_{p}), \text{ARGV}_{abs}(C_{abs}(a_{p}), i, 2), \text{ARGV}_{abs}(C_{abs}(a_{p}), i, 3))$$

Now we can insert $v_0$ as the result of a alt:

$$C''_{abs} = \langle A''_{t}, A''_{m} \rangle = \langle A'_{t}[\text{RESULT}_{slt}(v_0), \text{ARGV}_{abs}(C'_{abs}, i, 2), \text{ARGV}_{abs}(C'_{abs}, i, 3)]/\{v_0\}, A'_{m} \rangle$$

We then restrict to obtain contexts for the branches:

$$C^t_{abs} = \langle A^t_{t}, A^t_{m} \rangle = \text{Restrict}_{abs}(\text{ContAssoc}(C''_{abs}, \text{Part}_{21}(A''_{v}(v_0))))$$
$$C^f_{abs} = \langle A^f_{t}, A^f_{m} \rangle = \text{Restrict}_{abs}(\text{ContAssoc}(C''_{abs}, \text{Part}_{21}(A''_{v}(v_0))))$$

Finally, the interpretation of $i$ is given by

$$\text{NEXT}_{abs}(C_{abs}|a_{t}) = \text{ContAssoc}(\langle A^t_{t}, \text{abs}v/\{v_0\}, A^t_{m} \rangle)$$
$$\text{NEXT}_{abs}(C_{abs}|a_{f}) = \text{ContAssoc}(\langle A^f_{t}, \text{abs}v/\{v_0\}, A^f_{m} \rangle)$$

Note that the $\text{Part}_{21}$ and $\text{Part}_{21}$ operations do not need to split the $[0 \ldots 1]$ result range, since we took care to generate it as two components in equation 9.4.1.

The other conditional jumps (jeq and jne) are handled in a similar way, although the generation of the conditional result is slightly more complicated since we can determine equality for labels, procedures and vreg addresses as well as integers.

### 9.5 Memory Access Insns

#### 9.5.1 Load from Memory

The load insn sets its first argument vreg to the value stored in the memory address given by the sum of its last two arguments. In our abstract interpretation, the destination vreg's value range is obtained by joining the value ranges of all possible load sources in the value range of the address. Note that:

- If the address range contains non-address components, we have to assume that the programmer knows best and return an "unknown" result.

\(^1\)Note that there may be more than one of each, since the jump destination could be a vreg; the results only depend on the outcome of the test, not the actual destination.
- Both memory block and vreg addresses are valid as sources of the vreg's new value, but we cannot distinguish between the addresses of different instances of the same vreg in different stack frames of a recursive procedure.

- The offsets of memory addresses held in value ranges refer to the number of words from the block's base, so the byte-addressed offsets calculated by WIM3 code need to be adjusted before being looked up in the abstract memory environment.

Our load operator takes a context vector \( C_{\text{abs}} \), an arc \( a \) leading to a load insn such that \( C_{\text{abs}}(a) = \langle A_v, A_m \rangle \), and a source address value range \( r \) as operands. It checks for non-address, and multiple-instance variable address, components in the range, and returns the union of the corresponding contents as obtained from the context. We assume that \( \text{DECL}(p) \) is the set of all vregs and vectors declared in procedure \( p \), and \( \text{RECURSIVE}(C_{\text{abs}}, p) \) is true if \( p \) is recursive. We cannot define \( \text{RECURSIVE} \) at this point, because we need interprocedural information; see section 11.6.2.

\[
\text{LOADOP}(C_{\text{abs}}, r) =
\begin{cases}
  T_{\text{abs}} & \text{if } \text{PROJvals}(r) \cap (\text{Labels} \cup \text{Procs} \cup \text{Intervals}) \neq \emptyset \lor \\
  \exists p \in \text{Procs} \cdot \text{RECURSIVE}(C_{\text{abs}}, p) \land \\
  (\text{PROJvals}[\text{PARTmisc}(r)] \setminus \text{Globals}) \cap \text{DECL}(p) \neq \emptyset \\
  \{ \text{CONTENT}(A_m(b), \\
    \text{PROJval}(c) \mid c \in o \neqv_{\text{abs}} R_w \land \\
    \langle \text{id}, (b, o), a \rangle \in \text{PARTaddr}(r) \} \cup \\
  \langle A_v(v) \mid \langle \text{id}, v, o \rangle \in \text{PARTmisc}(r) \} \quad \text{otherwise}
\end{cases}
\]

where \( R_w \) is a range containing a single interval component specifying the target architecture's word size, \( W \):

\[
R_w = \{ \langle \text{id}_0, [W \ldots W], [\rangle \}\}
\]

This is an ugly (albeit effective) solution; we refer the reader to section 2.6.5 for reasons why byte addressing is a misfeature that can only be handled by kludges such as the above in typeless languages.

If \( \text{OP}_i = \text{load} \) for some insn \( i_c \), the result is given by

\[
\text{RESULT}(C_{\text{abs}}, \langle i_p, i_c \rangle) = \text{NORMALISE}(\text{LOADOP}(C_{\text{abs}}, \\
\text{ARGVabs}(C_{\text{abs}}, i_c, 2) +_{\text{abs}} \text{ARGVabs}(C_{\text{abs}}, i_c, 3)), \text{ARG}_{\text{id}}(i_c, 1), \text{nilcomp})
\]

and is inserted into the context in exactly the same way as the results of arithmetic and logic operations.

### 9.5.2 Store to Memory

The \text{sto }r insn stores the value of its first argument in the memory location specified by the sum of its other two arguments. Despite its similarity to the load insn, its interpretation is much less accurate.

The problem is that if the abstract destination address address corresponds to more than one address in the concrete context, we cannot update any part of the environment destructively: the new values should join the old ones without replacing them. This happens much more often than not, especially for accesses to heap memory whose uniqueness is impossible to determine. Addresses of vregs and vectors in recursive procedures present the same problem.
This is not as bad as it sounds. We may not be able to determine the exact contents of a memory location after a store, but we will probably be able to determine the kinds of values stored there, and this is usually enough for our purposes. Furthermore, we can handle stores more effectively if we know that there is only one possible target address.

Given a store insn \(i\), we obtain the value it stores from its first argument, and the destination address range from the sum of its last two arguments; hereafter,

\[
\begin{align*}
\text{cont} &= \text{ArgV}_{\text{abs}}(\text{C}_{\text{abs}}, i, 1) \\
\text{addr} &= \text{ArgV}_{\text{abs}}(\text{C}_{\text{abs}}, i, 2) +_{\text{abv}} \text{ArgV}_{\text{abs}}(\text{C}_{\text{abs}}, i, 3)
\end{align*}
\]

An address is \textit{deterministic} if there is only one location it could possibly refer to. Not only should it refer to a single abstract memory cell, but the cell must not be part of a heap block, or vreg/vector declared in a recursive procedure. Non-deterministic addresses can be detected by

\[
\text{NonDet}(\text{C}_{\text{abs}}, r) \iff r \not\equiv (c) \lor
\begin{array}{l}
\exists p \in \text{Procs} : (\text{Proj}_{\text{val}}(c) \in \text{DECL}(p) \land \text{RECURSIVE}(\text{C}_{\text{abs}}, p)) \lor \\
(\text{Part}_{\text{addr}}(r) \not\equiv \{\}) \land \\
(\text{Proj}_{\text{bas}}(c) \not\equiv \text{Vectors}) \lor \\
\exists n \in \mathbb{Z} : \text{Proj}_{\text{abs}}(\text{Proj}_{\text{off}}(c)) = [n \ldots n]\lor \\
\exists p \in \text{Procs} : (\text{Proj}_{\text{bas}}(c) \in \text{DECL}(p) \land \text{RECURSIVE}(\text{C}_{\text{abs}}, p))
\end{array}
\]

We can update the contents of deterministic addresses destructively, but for the rest we need to keep the old values as well, and destroy any assoc to the result. We need a way to remove assoc to components with given identifiers \(\text{ids}\) from a range \(r\):

\[
\text{Strip}_{\text{abv}}(r, \text{ids}) =
\begin{array}{ll}
\langle \langle \text{id}, \text{val}, \text{Strip}_{\text{asc}}(\text{asc}, \text{ids}) \rangle | (\text{id}, \text{val}, \text{asc}) \in \text{Part}_{\text{inst}}(r) \cup \text{Part}_{\text{misc}}(r) \rangle \\
\langle \langle \text{id}, \langle \text{bas}, \text{Strip}_{\text{abv}}(\text{off}, \text{ids}) \rangle, \text{Strip}_{\text{asc}}(\text{asc}, \text{ids}) \rangle | \\
(\text{id}, \langle \text{bas}, \text{off} \rangle, \text{asc}) \in \text{Part}_{\text{addr}}(r) \rangle
\end{array}
\]

where

\[
\text{Strip}_{\text{asc}}(a, \text{ids}) = \{\langle \text{id}, t \rangle | (\text{id}, t) \in a \land \text{id} \not\equiv \text{ids}\}
\]

We can now remove assoc to components with identifiers in \(\text{ids}\) from an entire context:

\[
\text{Strip}_{\text{abs}}(\text{C}_{\text{abs}}, \text{ids}) = \langle \lambda v . \text{Strip}_{\text{abv}}(\text{A}_{v}(v), \text{ids}) , \\
\lambda b . \langle \langle i, \text{Strip}_{\text{abv}}(r) \rangle | (i, r) \in \text{A}_{m}(b) \rangle \rangle
\]

Now, let \(a_{n}, a_{s} \in \text{Atcs}\) be the arcs leading to/from a store insn, and

\[
\text{C}_{\text{abs}}(a_{n}) = \text{C}_{\text{abs}} = \langle \text{A}_{v}, \text{A}_{m}\rangle
\]

be the context on entering it. We define its interpretation as

\[
\text{Next}_{\text{abs}}(\text{C}_{\text{abs}} | a_{s}) = \text{CONTASSOC} | \text{Strip}_{\text{abs}}(\langle \text{A}_{v}^{'}, \text{A}_{m}^{'} \rangle, \text{delids})
\]

where the vreg environment after the insn is

\[
\text{A}_{v}^{'} = A_{v} | \text{StoreVREG}(\text{C}_{\text{abs}}, \text{cont}, v, \text{NonDet}(\text{C}_{\text{abs}}, \text{addr}))/ \{w | w \in \text{Vregs} \cap \text{Proj}_{\text{val}}(\text{addr})\}
\]
STOREVREG gives the new value of a vreg \( v \) updated in \( C_{abs} \) with the value range \( r \), by a store whose determinacy is given by \( \text{nondet} \):

\[
\text{STOREVREG}(C_{abs}, r, v, \text{nondet}) = \\
\begin{cases} 
\text{STRIP}_{abv}(\text{NORMALISE}(r \cup A_v(v), v, \text{nilcomp}), \text{CompIds}) & \text{if nondet} \\
\text{NORMALISE}(r, v, \text{nilcomp}) & \text{otherwise}
\end{cases}
\]

The new memory environment is trickier to define. Recalling the definitions of \( \text{LOW} \) and \( \text{HIGH} \) from section 8.4.6, we can compute the offsets for which the block range resulting from the load instr is defined:

\[
\text{OFFSETS}(\text{val}_1, \text{val}_2) = \\
\{[m \ldots n] | (m \in \text{LOW}(\text{val}_1 \cup \text{val}_2) \lor (m - 1) \in \text{HIGH}(\text{val}_2)) \land \\
(n \in \text{HIGH}(\text{val}_1 \cup \text{val}_2) \lor (n + 1) \in \text{LOW}(\text{val}_2)) \land \\
(\forall \text{int} \in \text{val}_1 \cup \text{val}_2 . [m \ldots n] \subseteq \text{int} \lor [m \ldots n] \cap \text{int} = \{\})\}
\]

The contents of each memory block addressed in \( \text{addr} \) are computed by assigning:

- the union of \( \text{cont} \) and the old values, to offsets covered in \( \text{addr} \); if it is non-deterministic;
- \( \text{cont} \) alone, to offsets covered in \( \text{addr} \) if it is deterministic;
- the old values alone, to offsets untouched by the store.

Again, assscs are deleted if the address is non-deterministic.

\[
A'_m = \text{STOREMEM}(C_{abs}, \text{ab}, \text{cont}, \text{addr}) = \lambda b . \\
\begin{cases} 
\{[m \ldots n], \text{STRIP}_{abv}(\text{NORMALISE}(r, \langle b, [m \ldots n]\rangle, \text{nilcomp}), \text{CompIds}) | \\
\text{NONDET}(C_{abs}, \text{addr}) \land r = \text{cont} \cup \text{CONTENTS}(A_m(b), [m \ldots n]) \land \\
[m \ldots n] \in \text{OFFSETS}(\text{Pro} \text{inds}(A_m(b)), \text{Proj} \text{vals}(\text{off})) \land \\
[m \ldots n] \in \text{FLATTEN}(\text{Proj} \text{vals}(\text{off}))\} \\
\} \\
\begin{cases} 
\{[m \ldots n], \text{NORMALISE}(r, \langle b, [m \ldots n]\rangle, \text{nilcomp}) | \\
\text{NONDET}(C_{abs}, \text{addr}) \land r = \text{cont} \land \\
[m \ldots n] \in \text{OFFSETS}(\text{Pro} \text{inds}(A_m(b)), \text{Proj} \text{vals}(\text{off})) \land \\
[m \ldots n] \in \text{FLATTEN}(\text{Proj} \text{vals}(\text{off}))\} \\
\} \\
\{[m \ldots n], \text{NORMALISE}(r, \langle b, [m \ldots n]\rangle, \text{nilcomp}) | \\
r = \text{CONTENTS}(A_m(b), [m \ldots n]) \land \\
[m \ldots n] \in \text{OFFSETS}(\text{Pro} \text{inds}(A_m(b)), \text{Proj} \text{vals}(\text{off})) \land \\
[m \ldots n] \not\in \text{FLATTEN}(\text{Proj} \text{vals}(\text{off}))\} \\
A_m(b) \quad \text{if } \exists c \in \text{addr} . \text{PROJbas}(c) = b \\
\text{otherwise}
\end{cases}
\]

The set delids of identifiers assscs to which should be deleted is:

\[
\text{delids} = \{\text{Pro} \text{ids}(\text{STOREVREG}(C_{abs}, \text{cont}, v, 1 = 0)) | \\
v \in \text{Vregs} \cap \text{Proj} \text{vals}(\text{addr})\} \cup \\
\{\text{Pro} \text{ids}(\text{NORMALISE}(r, \langle b, [m \ldots n]\rangle, \text{nilcomp})) | \\
\text{NONDET}(C_{abs}, \text{addr}) \land r = \text{cont} \cup \text{CONTENTS}(A_m(b), [m \ldots n]) \land \\
[m \ldots n] \in \text{OFFSETS}(\text{Pro} \text{inds}(A_m(b)), \text{Proj} \text{vals}(\text{off})) \land \\
[m \ldots n] \in \text{FLATTEN}(\text{Proj} \text{vals}(\text{off})) \land \\
c \in \text{addr} \land b = \text{PROJbas}(c)\}
\]
9.6 Miscellaneous Insns

9.6.1 Procedure Entry

Since our analysis is still intraprocedural, we have no information about the program’s state on entering a procedure, unless it happens to be main. The program entrypoint. Thus the `ent` insn installs an initial abstract context mapping every vreg and memory location to `T_{abv}`, except for the memory corresponding to static data (tables and strings) which is immutable; if the procedure we are analysing happens to be the program entrypoint, we can also assume that all global variables have the initial values specified in the program, and all heap blocks allocated by `stdlib.calloc` (which initialises the memory it allocates) contain zeroes.

9.6.2 Procedure Return

At this point, the `retn` insn is effectively a no-op; when such an insn is reached, the abstract context is of no further use to anyone but the interprocedural optimizer described in chapter 11.

9.6.3 Procedure Call

Read-only static data, and non-global vregs whose addresses are not taken anywhere in the program, are the only components of the abstract context that will always survive a procedure call. Everything else is potentially modifiable, and should be set to `T_{abv}` to ensure the safety of our analysis. This includes caller-saved hregs, and the hregs holding result values and return addresses.

Of course, this utterly defeats the rest of our analyses; chapter 11 (in particular, sections 11.3.6 and 11.4.5) describes what can be done to improve on this pessimistic analysis.

9.6.4 Spills and Reloads

Spill and reload insns do not exist at this stage of compilation, since register allocation runs as the last pass of value range analysis and the new insns are not inserted until after the old ones are processed. However, even if they did exist, spills and reloads would have made no difference to the abstract interpreter; they are no-ops, as far as it is concerned, since the possible discrepancies between memory- and hreg-resident copies of vregs are invisible to it. See section 5.6.3 for details of how this illusion is maintained by the register allocator.

9.7 Convergence and Correctness

9.7.1 Introduction

We are not going to present a proof of our AIN’s consistency with the concrete semantics. Part of the reason is that we have not fully specified the latter, but more importantly, the reasons for this lack of specification given in section 7.4.1 also apply here: given the complexity of the analysis and the intended application, we believe that concentrating on the implementation is a better way to convince ourselves that the scheme is consistent. Semantic specifications and their proofs are no easier to debug than compilers, while
having the significant disadvantage of being much harder to verify by application to real code.

One might wonder why we have given such a formal specification of the abstract semantics, when we are not intending to prove their correctness, and are vague about the concrete semantics they abstract. The answer is that a specification is not just a means to an end, but an end in itself. It is more concise, accessible, and comprehensible than a compiler; it is easier to produce; it is not constrained by considerations of efficiency. It exposes theoretical issues that could otherwise be overlooked in implementation, but hides some of the goriest programming details. At the same time, it is more accurate and less prone to misunderstandings than an informal natural language description. Just as a language grammar can be useful without a specification of its semantics, so can the semantics be useful without a proof of their correctness.

Having said all this, it is worth giving a superficial outline of such a proof, if only to show how the various elements of the AIN fit together. The outline comes in two parts, discussed in the following two sections.

9.7.2 Monotonicity of \( \text{NEXT}_{\text{abs}} \)

To prove that the AIN converges, we have to show that its transition function is monotone (see section 7.4.4). We assume the existence of context vectors \( C_{\text{abs}}:1 \) and \( C_{\text{abs}}:2 \) such that

\[
\forall \alpha \in \text{Arcs} \cdot C_{\text{abs}}:1(\alpha) \subseteq C_{\text{abs}}:2(\alpha)
\]

and try to prove that the same is true after the vectors are passed through \( \text{NEXT}_{\text{abs}} \).

Now, we know that the context at an arc only depends on that arc's predecessors, and, according to section 9.6.1, the context at entry arcs is constant throughout, so it is sufficient to prove:

1. The monotonicity of \( u_{\text{abs}} \) (for arcs entering a block).
2. The monotonicity of interpreting a single insn, i.e.

\[
\forall \alpha \in \text{Arcs} \cdot \text{NEXT}_{\text{abs}}(C_{\text{abs}}:1)(\alpha') \subseteq \text{NEXT}_{\text{abs}}(C_{\text{abs}}:2)(\alpha')
\]

where \( \alpha' \) is the only successor of \( \alpha \) (for arcs within a block).

Given the above, the rest of the proof follows by induction on the structure of the code. To prove that the interpretation of a single insn is monotone, we have to examine every insn opcode in turn; we will not attempt to do this here.

9.7.3 Consistency of \( \text{NEXT}_{\text{cont}} \) and \( \text{NEXT}_{\text{abs}} \)

Recall the local consistency condition (equation 7.5 in section 7.4.4); we need to prove both halves of this to ensure that our AIN is consistent. We can use a structural induction argument similar to that of the previous section, so that for the first half, we only need to show

\[
\forall \alpha \in \text{Arcs} \cdot \text{NEXT}_{\text{cont}}(\gamma(C_{\text{abs}}))(\alpha') \subseteq \gamma(\text{NEXT}_{\text{abs}}(C_{\text{abs}})(\alpha'))
\]

where \( \alpha' \) is the only successor of \( \alpha \). Again, this involves a tedious case-by-case analysis of insn handling semantics; it also requires a fuller definition of \( \text{NEXT}_{\text{cont}} \) than the one we have given.
9.8 Implementation Details

9.8.1 Introduction

As with the basic operations on abstract contexts (see section 8.7.6), the abstract interpreter's implementation of insn handling consists mostly of following pointers around and is not very interesting. The following sections give an idea of what is involved, but first let us look at how the entire analysis is put together.

9.8.2 Applying the Interpretation

In the next chapter, we will see how the abstract interpretation integrates with the optimiser that invokes it (see algorithm 10.1 in section 10.1). Here we present the algorithm that applies the analysis to the code. Recall that a range table is the compiler's equivalent of an abstract context (see section 8.7.4); the algorithm generates range tables for block boundaries, from which tables describing the state within a block can be reconstructed.\(^2\)

Algorithm 9.1 (Value Range Analysis)

1. For each basic block do:

   (a) If the block has a conditional exit, allocate two range tables for values on block exit along each branch; otherwise, only allocate one.

   (b) If the block has a back edge pointing to it, allocate another range table for values on entry to the block.

   (c) Initialise all range tables to \(\perp_{abv}\) values.

2. Traverse the flowgraph's spanning DAG in preorder and for each block \(b\) do:

   (a) Create a new range table \(t\) and populate it by joining all range tables on exit from the predecessors of \(b\), along paths leading to \(b\).

   (b) If there is a back edge leading to \(b\), replace \(t\) and \(b\)'s entry table with the result of widening the two.

   (c) Update \(t\) according to the interpretation of each insn in \(b\), in turn.

   (d) If \(b\) has an unconditional exit, replace its exit table with \(t\).

   (e) If the exit is conditional, produce two copies of \(t\) restricted with respect to the different outcomes of the conditional; replace \(b\)'s exit tables with these, and discard \(t\).

This process is repeated until the results converge (i.e. until the old and new exit tables of every block are the same), then performed again, with narrowing in place of widening.

9.8.3 Execution Cost

The interpretation consists of many different operations whose frequency depends on the nature, as well as the size, of the code. Even if we assume that all operations on value ranges have unit cost, there is no way to estimate how

\(^2\)The reader may recall that we have used the same technique for the liveness information required by the register allocator; see algorithm 5.3 on page 57.
tightly knit the values are by their net of assocs. Any claims as to the cost of
the analysis would be misleading, so we will not make any.

We will, however, mention that the costliest example we will consider in
this thesis (see section 10.6.3), which consists of 81 basic blocks and
requires about 50 passes through the range engine to be fully optimised, takes
about 9.5 real seconds to compile on a multiuser Alpha workstation. Of those,
3.6 seconds represent processor time. Given that the rest of the time is spent
waiting for the assembler and linker, we deem the cost acceptable.

9.8.4 Addition and Subtraction

Addition and subtraction insns are handled by the same code. A common rou-
tine traverses a list of components, adding an interval to each of them; to add
two ranges, each interval of the second is added to the first and each resulting
range is unioned into the result. Subtraction is carried out by negating and
reversing the limits of each interval added to the first range. Offsets of address
components are handled recursively. If any components that cannot be added
are found, the special \( T_{abv} \) component is returned as the result.

As with context-level operations, we take care to keep assocs consistent,
so that they do not need to be fixed later.

9.8.5 Multiplication, Division, and Shifts

Shifts are implemented as multiplications or divisions by the appropriate val-
ues; the compiler does not attempt to handle shifts by non-constant amounts.

Multiplication and division share code in the same way that addition and
subtraction do. At the moment, only constant multipliers and divisors are
supported; other values cause an interval spanning all the integers to be re-
turned. Non-interval components in the operands return \( T_{abv} \) results.

In acceptable cases, the result is formed by multiplying (or dividing) the
limits of intervals in the multiplicand (or dividend) by the multiplier (or di-
visors); if the latter is negative, the limits are reversed, and the result range
is also built in reverse order so that the numerical ordering of components is
maintained. A final scan of the resulting range merges overlapping compo-
nents.

9.8.6 Other Insns

Most other insns are straightforward implementations of the formal defini-
tions given earlier in this chapter, except for \( \text{entr} \), \( \text{retn} \), and \( \text{call} \) insns which
take advantage of the interprocedural methods presented in chapter 11, and
bitwise operations, which are handled conservatively. Unconditional jumps
are ignored, since they do not affect the context.

Conditional jumps have already been mentioned in the block-level algo-


\footnote{To put this into perspective, consider that typical compilers turn off all optimisations for
procedures whose size exceeds 300 blocks.}
Chapter 10

Range Analysis: Applications

10.1 Abstract Interpretation Phases

The previous two chapters largely consisted of a "zoom in" to the internals of the abstract interpretation engine. In this chapter, we will describe how its results can be used to produce better code and diagnostics, but in order to do this we will first have to "zoom out" and consider the embedding of our abstract interpretation in the rest of the compiler.

At least one widening interpretation has to converge before we can start relying on the results; a subsequent narrowing interpretation is recommended, but using the results to modify the code will render them invalid and the process will have to be restarted. Even when no more changes to the code are to be made, we will have to run a few more narrowing passes to recover value range information for the benefit of other components of the compiler, such as the inter-modular optimiser and the register allocator. The information is not otherwise readily accessible, since it is stored between interpretations only at basic block boundaries.

The whole process, as implemented in the P compiler, is as follows:

Algorithm 10.1 (Value Range Analysis Driver)

1. Widening and narrowing interpretations are performed, and the lists of possible destinations for jumps to vregs are updated (see section 10.2.2).

2. Another pair of widening and narrowing interpretations is performed, during which constants are propagated (section 10.2.3) and conditional jumps turned into unconditional ones wherever possible (section 10.2.4).

3. The dead code and unnecessary computation eliminators are invoked (sections 3.5.2 and 3.5.4), to remove any newly found dead or redundant code.

4. If any changes were made to the code in the previous steps, all value range information is thrown away and we go back to step 1.

5. A single iteration of the narrowing computation is performed; helpful warnings are issued (section 10.2.7), constants are propagated again and other optimisations are carried out.

6. Another narrowing iteration collects information for inter-modular optimisation.
7. Distances to next use of vregs are computed (section 5.5), then the register allocator is invoked from within a final narrowing iteration so that it has access to value range information.

The rest of this chapter will discuss the actions taken at each of the above stages.

10.2 Simple Applications of Value Range Analysis

10.2.1 Introduction

In the introduction to this thesis we identified two approaches which modern compilers can employ to produce "better" code: optimisation, which alters the program to make it run faster, and verification, which attempts to check that the program behaves as intended and enables optimisations that are only valid under certain non-obvious conditions. The applications of range analysis mentioned in algorithm 10.1 cover both areas, and the next few sections will present some of them in more detail.

10.2.2 Jump Destination Determination

Occasionally, jumps whose destination is a vreg are encountered in the intermediate code. These are normally the result of .goto statements in the source code, and so far we have been assuming that the successors of blocks containing such insns include every block whose address could have ended up in a
vreg—that is, every block whose identifier is an operand of a non-jump insn (see section 3.5.2.1).

However, after the first widening interpretation, we will have a reasonably accurate idea of the relevant vreg's contents, and will be able to redefine the insn's successor set. Let \( b \in \text{Blocks} \) be the block containing a jump insn \( i \), and \( C_{\text{abs}} \) the abstract context just before \( i \). We define

\[
\text{Succ}_{\text{abs}}(b) = \text{PROJ}_{\text{vals}}(\text{ArgV}_{\text{abs}}(C_{\text{abs}}, i, 1) \cup \text{ArgV}_{\text{abs}}(C_{\text{abs}}, i, J))
\]

If \( \text{Succ}_{\text{abs}}(b) \subseteq \text{Blocks} \), i.e. the destination ranges do not contain any non-label values, we redefine \( \text{Succ}_{\text{abs}}(b) \) to be equal to \( \text{Succ}_{\text{abs}}(b) \) for all subsequent operations; if not, we have either lost track of the destination in abstracting, or the programmer is attempting something silly, and we should keep our old approximation.

Note that the above optimisation does not alter the code itself in any way; it just simplifies the compiler's notion of the flowgraph by removing some redundant edges. Some of what it achieves could have been done by constant propagation, but our method goes much further than that. Consider, for example, the program of figure 10.1.

The program uses a label variable (\`dest\') to defer the results of a conditional so that it can modify variables (\`arg1'...\`arg4') used in it before taking any action on their old values. There are three labels in the code, and the original approximation would have considered any of them as targets for the .goto; this would have introduced a cycle back to start, fooling the compiler into thinking that there is an extra loop in the procedure. The range analysis engine is capable of determining that the value of \`dest\' at the .goto is either middle or end, and will simplify the flowgraph accordingly. A constant propagation approach would not have worked here, since the relevant value of \`dest\' depends on the (unknown) values of the \`arg's and is not a constant.

We have thus solved the problem that forces languages like C to disallow label-valued variables; P programmers (as well as anyone else using a compiler that implements this technique, and a language that allows it) are free to use labels, unfettered by fears of compiler retribution in the form of appallingly sub-optimal code.

It should also be mentioned that, to save some processing, this optimisation can usefully be made while the narrowing phase of value range computation is still converging. The results of any iteration after the widening phase are safe, and label-valued variables are unlikely to be constrained further since labels are a part of the lattice with finite depth and do not require widening or narrowing, while their interactions with value kinds that do require them are very simple in most, if not all, sanely written programs.

Of course, to take full advantage of this optimisation, we will have to recompute all information that depends on the flowgraph, including live vreg sets and value ranges, and ask the dead code and redundant computation eliminators to rescan the code. This is why the first three steps of algorithm 10.1 have to be repeated.

### 10.2.3 Constant Propagation

Constant propagation (CP) is, of course, a very old and well-known optimisation technique: It consists of the replacement of vregs which can be shown to hold constant values by the values themselves. The analysis which underlies CP is subsumed by our range analysis engine, so we can do CP at negligible added cost.
Moreover, our engine can detect more constants than traditional CP. The contrived example of figure 10.2 illustrates this: we need the full power of range analysis to determine that \( b \) has the value 27 at the call to \( \text{bar} \), since simple CP cannot tell that its value is less than or equal to 27 before the second conditional.

Implementing CP is easy: We need to modify insn arguments that

1. are vregs,
2. have a constant value range, and
3. are referenced (rather than defined) by the insn.

Given an abstract context \( C_{abs} \), an insn \( i \) and an identifier \( n \in \{1,2,3,4\} \) for a non-destination vreg argument (i.e. any argument with \( \text{ArgT}(i,n) = \text{reg} \), except the first argument of insns listed in section 9.2.3), we can redefine

\[
\text{ArgT}(i,n) \leftarrow \left\{ \begin{array}{ll}
\text{con} & \text{if } \text{Projvals}[\text{ArgV}_{abs}(C_{abs},i,n)] = [m \ldots m] \land m \in \mathbb{Z} \cup \{\pm \infty\} \\
\text{str} & \text{if } \text{Projvals}[\text{ArgV}_{abs}(C_{abs},i,n)] = [s] \land s \in \text{Statics} \\
\text{bbl} & \text{if } \text{Projvals}[\text{ArgV}_{abs}(C_{abs},i,n)] = [b] \land b \in \text{Blocks} \\
\text{fun} & \text{if } \text{Projvals}[\text{ArgV}_{abs}(C_{abs},i,n)] = [p] \land p \in \text{Procs} \\
\text{reg} & \text{otherwise}
\end{array} \right.
\]

and

\[
\text{ArgI}(i,n) \leftarrow \left\{ \begin{array}{ll}
m & \text{if } \text{Projvals}[\text{ArgV}_{abs}(C_{abs},i,n)] = [m \ldots m] \land m \in \mathbb{Z} \cup \{\pm \infty\} \\
s & \text{if } \text{Projvals}[\text{ArgV}_{abs}(C_{abs},i,n)] = [s] \land s \in \text{Statics} \\
b & \text{if } \text{Projvals}[\text{ArgV}_{abs}(C_{abs},i,n)] = [b] \land b \in \text{Blocks} \\
p & \text{if } \text{Projvals}[\text{ArgV}_{abs}(C_{abs},i,n)] = [p] \land p \in \text{Procs} \\
\text{ArgI}(i,n) & \text{otherwise}
\end{array} \right.
\]

where we use ‘\( \leftarrow \)’ to stress the fact that the values on the left hand side are changed, nothing being claimed about their previous values.

Note that we cannot propagate heap addresses, because of their inherent indeterminacy, or vector and vreg addresses, because they are relative to the stack pointer. We do, however, propagate procedure and label addresses;
this solves an efficiency problem inherent in, for example, BCPL, where all
intermodular procedure calls must be preceded by a reference to the global
variable holding the address.

Also note that machine-specific constraints may apply; on some processors,
loading an immediate constant may be slower than accessing some registers, so
it may be worth leaving the vreg reference in place, even if it causes more work
for the register allocator. RISC machines do not generally have this problem,
but they do have constraints on the placement of immediate constants in some
insns: they may be required to fit in a specified number of bits (e.g. MIPS) or
consist of a small number of bits shifted by an arbitrary amount (e.g. ARM), or
even occur on the right-hand side of the operator only (MIPS again). Usually,
the constraints are less tight on jump and call insns, so we will still be able to
optimise away most of P’s indirect calls and .gotos.

CP does not remove the assignments to vregs whose uses it eliminates, so
it will be necessary to recompute liveness information and run the redundant
insn eliminator again, as shown in step 3 of algorithm 10.1.

10.2.4 Conditional Jump Reduction

As we noted in section 9.4.1, we can sometimes determine the result of a
comparison at compile time. In the case of the \texttt{slt} insn, the constant result
will be propagated as described above, but conditional jumps do not store the
result in a vreg and cannot be handled in this way. We need to notice when
jump conditions are constant and turn the jumps into unconditional ones, so
that a subsequent pass of the dead code eliminator will be able to remove any
unused code on the branch that is never taken.

Consider, for example, an insn \(i\) with \(OP_i = \texttt{jeq}\), executed in a context
\(C_{\text{abs}}\). If one of

\[
\begin{align*}
\text{ArgV}_{\text{abs}}(C_{\text{abs}}, i, 2) &= \text{PartInt}(\text{ArgV}_{\text{abs}}(C_{\text{abs}}, i, 2)) \land \\
\text{ArgV}_{\text{abs}}(C_{\text{abs}}, i, 3) &= \text{PartInt}(\text{ArgV}_{\text{abs}}(C_{\text{abs}}, i, 3)) \land \\
\max(\text{Flatten}(\text{ProjVal}(\text{ArgV}_{\text{abs}}(C_{\text{abs}}, i, 2)))) &< \\
\min(\text{Flatten}(\text{ProjVal}(\text{ArgV}_{\text{abs}}(C_{\text{abs}}, i, 3))))
\end{align*}
\]

is true, i.e. the condition is always satisfied, we can rewrite the insn so that

\[
\begin{align*}
OP_i(i) &\leftarrow \texttt{jeq} \\
\text{ArgT}(i, 2) &\leftarrow \text{ArgT}(i, 3) \leftarrow \text{reg} \\
\text{ArgI}(i, 2) &\leftarrow \text{ArgI}(i, 3) \leftarrow 0
\end{align*}
\]

We also need to remove the unconditional jump that follows the insn and
adjust successors and predecessors accordingly. The conversion for a condition
which is never satisfied is similar, except that the jump destination is taken
from the following unconditional jump (accessed as the J argument of \(i\)).

Since we are already checking for constant results in generating the result
range of a conditional, this optimisation is trivial to perform. One might also
expect it to be of little consequence; this is untrue for two reasons:
1. In combination with the jump threading and block duplication performed by the dead code eliminator, it forms a generalisation of an optimisation that most compilers handle as a special case: the simplification of logical expressions used in source-level conditionals. We will discuss this further in section 10.6.2.

2. When a module is specialised (see section 11.6.5), many of its tests may become unconditional, either because they check for error conditions that the caller never provokes, or because they are part of procedures whose abilities are not exercised fully by their callers.

10.2.5 Alias Analysis

There exists a host of specialised optimisations that can be employed by compilers for machines with vector processors or massively parallel architectures. Investigating these is beyond the scope of our thesis, but it is worth mentioning that many of these optimisations rely on effective alias analysis, which verifies whether two pointers can point to the same location.

There exist sophisticated approaches to alias analysis, some of them involving abstract interpretations of pointer values [Deu94]. Our value range lattice gives us a relatively unsophisticated alias analyser for free:

\[
\text{ALIAS}(r_1, r_2) \iff \exists c_1 \in r_1 \cdot \exists c_2 \in \text{RESTRICT}_{abv}(r_2, \{c_1\}) .
\]

\[
\begin{align*}
& (\text{PROJ}_{bas}(c_1) = \text{PROJ}_{bas}(c_2) \land \\
& \text{FLATTEN}(\text{PROJ}_{vals}(\text{PROJ}_{off}(c_1))) \cap \\
& \text{FLATTEN}(\text{PROJ}_{vals}(\text{PROJ}_{off}(c_2))) \neq \{\} )
\end{align*}
\]

is true if value ranges \(r_1\) and \(r_2\) can simultaneously point to the same memory block.

10.2.6 Accesses to Addresses of Variables

Traditionally, compilers encountering a variable whose address is taken do not attempt to treat it as a register, virtual or otherwise; they keep it on the stack, and use load/store insns to access it. In chapter 5, we showed how to avoid this, keeping the variable in a vreg which is allocated to an hreg as usual and is only split to the stack every now and then as a precaution to ensure the stack- and vreg-bound values are in step.

One way in which value range information can reduce such memory traffic is by determining whether the vreg's memory copy can be affected by a load or store, allowing the elimination of some precautionary spills; we will describe this in more detail in section 10.3.2. Another way it can help is by converting accesses to the vreg's memory copy via load/store insns into simple vreg moves.

A load (or store) insn is eligible for conversion to a vreg move if:

1. The source (or destination) address range consists of a single vreg address.

2. The vreg is declared in the procedure containing the insn.

3. The procedure is not recursive.
We recall the functions \texttt{DECL} and \texttt{RECURSIVE} used in section 9.5.1, and state the condition as follows:
\[
\text{PRo} \forall \text{abs} \left( \text{ARG} \left( \text{abs} \left( i \right) \right), \text{abs} \left( i \right) \right) + \text{ abs} \left( \text{abs} \left( i \right) \right) = \left\{ v \right\} \land \\
\forall \text{regs} \land v \in \text{DECL} \left( p \right) \land i \in \text{INSNS} \left( p \right) \land \neg \text{RECURSIVE} \left( \text{abs} \left( p \right) \right)
\]

If the condition holds, we define
\[
\begin{align*}
\text{source} & = \begin{cases} 
v & \text{if } \text{OP}_i \left( i \right) = \text{load} \\
\text{ARG}_i \left( i, 1 \right) & \text{if } \text{OP}_i \left( i \right) = \text{stor}
\end{cases} \\
\text{dest} & \in \left\{ v, \text{ARG}_i \left( i, 1 \right) \right\} \setminus \{ \text{source} \}
\end{align*}
\]

and modify the insn so that
\[
\begin{align*}
\text{OP}_i \left( i \right) & \leftarrow \text{add} \\
\text{ARG}_i \left( i, 1 \right) & \leftarrow \text{ARG}_i \left( i, 2 \right) \leftarrow \text{ARG}_i \left( i, 3 \right) \leftarrow \text{reg} \\
\text{ARG}_i \left( i, 1 \right) & \leftarrow \text{dest} \\
\text{ARG}_i \left( i, 2 \right) & \leftarrow \text{source} \\
\text{ARG}_i \left( i, 3 \right) & \leftarrow 0
\end{align*}
\]

This makes the code faster by removing a memory reference and allowing its replacement to be eliminated altogether by clever register assignment.

\section*{10.2.7 Advisory Type Checking}

The most common complaint about typeless languages is that they lack type checking. Stated in this way, the complaint may seem to beg the question, but it \textit{does} have a core of validity in that the flexibility of languages like BCPL and P does not come for free. Common mistakes of the “adding apples to oranges” genre can pass through a typeless compiler undetected, as can off-by-one errors and misaligned offsets into memory structures that cause fields to be misinterpreted. Type checking cannot solve all such problems, but even the weak types of a language like C protect against the most common blunders.

Our objective here is to verify the program so that as much of the benefit of type checking as possible is maintained, without losing any of the flexibility of the language. We do this by checking for common problems as we interpret the code and issuing warnings.

As far as possible, we try to do this without asking for user intervention; thus the approach is different to \textit{abstract debugging} \cite{Bou93}, which requires the user to provide assertions to be checked. It is also different to the \textit{type inference} systems \cite{Hin69} of languages like ML, since the warnings are \textit{advisory}—they do not affect the outcome of the compilation, and programmers are free to ignore them (although they probably should not).

We distinguish between two kinds of warnings, informing the user of conditions that will \textit{definitely} occur at run time, almost certainly indicating a programming error, and conditions that might \textit{possibly} occur. The uncertainty of the latter may be due to the inherent inaccuracy of a static analysis (i.e. to the fact that multiple executions of the same insn are not distinguished), the generality of some value kinds (i.e. the blurring of the distinction between different incarnations of vectors and heap blocks), or the inaccuracy of the interpretation itself (due to widening, or the simplified handling of range arithmetic). “Definite” warnings are more useful, and the user may wish to disable “possible” warnings, especially during the early phases of compilation when no inter-module information is available and the analysis is still quite vague.
Roughly in order of increasing obnoxiousness, the warnings produced by
the P compiler concern:

**Writes to static data.** Strings and tables are placed in read-only storage, and
attempts to write to them will result in run-time errors. For an insn \( i \) such that \( \text{OP}(i) = \text{stor} \), executed in a context \( C_{\text{abs}} \), let \( r \) be the destination
address:

\[
    r = \text{ARGV}_{\text{abs}}(C_{\text{abs}}, i, 2) + \text{argv}_{\text{abs}}(C_{\text{abs}}, i, 3)
\]

We should issue a "possible" warning if

\[
    \exists c \in \text{PART}_{\text{addr}}(r) : \text{PROJ}_{\text{bas}}(c) \in \text{Statics}
\]

and we should upgrade this to a "definite" warning if we also have

\[
    r = \text{PART}_{\text{addr}}(r) \land \forall c \in r. \text{PROJ}_{\text{bas}}(c) \in \text{Statics}
\]

**Out-of-bounds accesses to vectors, strings and tables.** The bounds of vectors
and static objects, unlike those of heap blocks, are known at compile
time; we assume the existence of a function

\[
    \text{BOUNDS} : \text{Vectors} \cup \text{Statics} \rightarrow \text{Intervals}
\]

returning the bounds of a vector or static as a single integer interval.
Given \( r \), the source/destination address range of a \text{load}/\text{stor} insn (defined as above), we will issue a "possible" warning if

\[
    \exists c \in \text{PART}_{\text{addr}}(r) \cap (\text{Vectors} \cup \text{Statics}) . \text{FLATTEN}([\text{PROJ}_{\text{vals}}(\text{PROJ}_{\text{off}}(c))] \setminus \text{BOUNDS}([\text{PROJ}_{\text{bas}}(c)]) \neq \{}\]

and will escalate this to a "definite" warning if

\[
    r = \text{PART}_{\text{addr}}(r) \land \forall c \in \text{PART}_{\text{addr}}(r) . (\text{PROJ}_{\text{bas}}(c) \in (\text{Vectors} \cup \text{Statics}) \land \text{FLATTEN}([\text{PROJ}_{\text{vals}}(\text{PROJ}_{\text{off}}(c))] \cap \text{BOUNDS}([\text{PROJ}_{\text{bas}}(c)]) = \{}\]

i.e. if every possible address corresponds to a vector or static and its
offset range does not contain any valid offsets for that particular memory
object.

There is nothing to stop us extending this to heap blocks as well, since
at the point where the block is allocated, an estimate of its size will be
available (if only as an integer range). However, since heap blocks often
have a size that is not a compile-time constant (or a constant of any kind),
this could cause a flood of warnings, most of which would be irrelevant,
while failing to catch some overruns and annoying the user into ignoring
valid warnings along with them. The compiler's credibility being at stake,
we decided to eliminate such warnings altogether.

**Invalid calls, jumps and memory accesses.** More closely related to traditional
type checking, these detect common errors which are usually due to the
inability to distinguish between procedures and variables exported by
interfaces, errors in indexing typeless structures and objects (see also
section 11.9.4), or missing procedure arguments. In detail, given \( i \in \text{Insns} \) executed in a context \( C_{\text{abs}} \):
We should issue a "possible" warning if \( O_P(i) = \text{call} \) and
\[
\text{PROJvals} \{ \text{ARGVabs} \{ \text{CAbs}, i, 1 \} \} \not\subseteq \text{Procs}
\]
(i.e. the called address range contains non-procedures), upgrading it to a "definite" warning if
\[
\text{PROJvals} \{ \text{ARGVabs} \{ \text{CAbs}, i, 1 \} \} \cap \text{Procs} = \{\}
\]
(i.e. the called address range does not contain any procedures).

We should "possibly" warn the user if \( O_P(i) \in \{ \text{jeq, jlt, jne} \} \) and
\[
\text{PROJvals} \{ \text{ARGVabs} \{ \text{CAbs}, i, 1 \} \} \not\subseteq \text{Blocks} \lor \\
\text{PROJvals} \{ \text{ARGVabs} \{ \text{CAbs}, i, J \} \} \not\subseteq \text{Blocks}
\]
but "definitely" warn them if we also have
\[
\text{PROJvals} \{ \text{ARGVabs} \{ \text{CAbs}, i, 1 \} \} \cap \text{Blocks} = \{\} \land \\
\text{PROJvals} \{ \text{ARGVabs} \{ \text{CAbs}, i, J \} \} \cap \text{Blocks} = \{\}
\]
This case is similar to the previous one, substituting jumps for calls and labels (basic block identifiers) for procedure identifiers.

If \( O_P(i) \in \{ \text{load, stor} \} \), \( r \) is the destination address, i.e.
\[
r = \text{ARGVabs} \{ \text{CAbs}, i, 2 \} +_{\text{abv}} \text{ARGVabs} \{ \text{CAbs}, i, 3 \}
\]
and we have
\[
\text{PARTINT}[r] \neq [\} \lor \text{PROJvals} \{ \text{PARTmisc}[r] \} \not\subseteq \text{Vregs}
\]
we should issue a "possible" warning, indicating that the memory address range contains non-address components; if we also have
\[
\text{PARTadd}[r] = \{ \text{PROJvals} \{ \text{PARTmisc}[r] \} \cap \text{Vregs} \} = \{\}
\]
the warning should be a "definite" one, since there are no addresses in the range.

**Uninitialised variables.** In section 3.5.4, we presented a traditional method for detecting uninitialised variables by looking for vregs live at the start of a procedure. The problem with this is that the anomaly is reported where it is detected, not at the reference which causes it to occur; adding datastructures to hold this information, or tracing through the flowgraph to recover it every time a problem is detected, is inelegant and wasteful. Range analysis lets us detect uninitialised variables at their use, or the nearest point to it where their value is certain to be undefined; the method works on non-global vregs, as well as the contents of vectors in non-recursive procedures.

The method relies on the fact that \( \perp_{\text{abv}} \) values are only generated on entry to a procedure, or on dead branches of conditional jumps. The latter are eliminated as described in section 10.2.4, leaving only the former which may be checked for at \text{load} insns and vreg references. These checks result in "definite" warnings; vregs (or vector cells) left uninitialised only on some branches leading to a use can be detected at the point where valid and \( \perp_{\text{abv}} \) values merge during a \( \perp_{\text{abs}} \) operation, at which point a "possible" warning is issued.
Miscellaneous grocery mixups. While discussing the interpretation of insns in the previous chapter, we encountered a number of cases which we handle by producing a $T_{ab}$ result, simply because the operation is unlikely to make sense. Additions of addresses to other addresses and the use of labels in arithmetic are among these cases, which are too numerous to present individually. We might want to produce warnings about each of these, but unlike similar "type mismatches" mentioned above, these are not only rarer, but also usually intended when they do occur (e.g. adding an offset to a label might make sense, if we have an idea of the machine code residing in the label's vicinity).

If one decides to implement such warnings, it should be kept in mind that they are likely to proliferate during early compilation stages due to the lack of intermodular information, and should probably be disabled until more sense can be made of the results.

Divisions by zero. A check that is trivially simple to do given the range analysis framework, but helps prevent a common run-time error: Given a div insn $i$ in context $C_{abs}$, we will issue a "possible" warning if

$$0 \in \text{Flatten} [\text{Proj}_{\text{vals}} [\text{Part}_{\text{ints}} [\text{Arg}_{\text{abs}} (C_{abs}, i, 3)]]]$$

or a "definite" warning if

$$\text{Proj}_{\text{vals}} [\text{Arg}_{\text{abs}} (C_{abs}, i, 3)] = \{0 \ldots 0\}$$

Unfortunately, the "possible" version of the warning is likely to show up more often than necessary due to the inaccuracy of range arithmetic; consider, for example,

$$[-2 \ldots -1] \times [+1 \ldots +2] = [-4 \ldots +4]$$

where the resulting interval contains $0$, even though it is not a possible result of the operation. Using this range as the divisor in a subsequent operation will cause the warning to appear, and it will be up to the programmer to ascertain that it is only a false alarm.

Jumps with more than one possible destination. If $i$ is a jump insn, we might like to issue a warning if

$$|\text{Arg}_{\text{abs}} (C_{abs}, i, 1)| \neq 1 \lor |\text{Arg}_{\text{abs}} (C_{abs}, i, J)| \neq 1$$

This will cause the compiler to moan whenever jumps with more than one (or two, in the case of conditionals) destination(s) are encountered. Useful for people who hate .gotos, are used to the C convention, or are porting legacy code, this feature is bound to annoy seasoned typeless programmers.
10.3 Interaction with Register Binding

10.3.1 Introduction

The range analyser interacts with the process of register binding in two ways: It provides information to allow the removal of some spills, and augments register spans (introduced in section 6.2) with information about the sizes of their values. The latter is used to perform a new optimisation which, for want of a better term, we have named register stuffing.

10.3.2 Elimination of Precautionary Stores and Spills

Section 5.6.3 mentions that members of the set $\text{Globals} \cup \text{Addrvars}$ have to be saved or spilt before load and store insns. These precautions are unnecessary in most cases, but a register allocator that does not interact with the range analysis engine has no way of knowing which regs are safe from such manipulation.

Algorithm 10.1 performs register allocation while value range information is still accessible (in step 7), so that given an insn $i$ with $\text{OP}_{i}(i) \in \{\text{load, stor}\}$ executed in a context $C_{\text{abs}}$, we can replace $\text{Globals} \cup \text{Addrvars}$ in the definitions of $\text{SAVE}_{\text{load}}(i)$ and $\text{SAVE}_{\text{ret-c1}}(i)$ (equations 5.6 and 5.7) by

\[
\text{SAVE}_{\text{abs}}(C_{\text{abs}}, i) = \begin{cases} 
\text{Globals} \cup \text{Addrvars} & \text{if } \text{PART}_{\text{intc}}(\tau) \neq \emptyset \\
\{v \mid v \in V\text{regs} \cap \text{PROJ}_{\text{vals}}(\text{PART}_{\text{misc}}(\tau))\} & \text{otherwise}
\end{cases}
\]

where $\tau = \text{ARGV}_{\text{abs}}(C_{\text{abs}}, i, 2) +_\text{abv} \text{ARGV}_{\text{abs}}(C_{\text{abs}}, i, 3)$ is the source/destination address of the load/store.

As most loads and stores can be determined to refer to vectors, statics or heap blocks, this optimisation will usually eliminate all but the truly necessary precautionary stores, thus removing the last performance obstacle in the way of allocating hregs for global and addressed vregs.

Of course, the above only applies to memory access insns; in the case of procedure calls, $\text{SAVE}_{\text{ret-c1}}$ cannot be modified with the information we have. In section 11.6.3, we will show how to use information from other translation units to perform a similar optimisation for call insns.

10.3.3 Value Sizes

10.3.3.1 Optimising with Respect to Value Sizes

In a typed language, each datatype has a "size", normally measured in bytes, representing the amount of storage space a value of that type requires: Character values, for example, are typically one byte long, while integers and pointers take up to 4 or 8 bytes. Knowledge of a value's size sometimes allows better use to be made of registers or, more commonly, main memory: Some processors have small registers which can be put to good use holding boolean or character values, and most systems store 4 or 8 character values in each memory word of the corresponding size.

In typeless languages, all values have the same, maximum size, although inelegant workarounds are sometimes used to pack bytes in memory (but we have chosen not to indulge in such practices; see section 2.6.5). The performance difference is not immense, but we claim that this is because typed languages do not make use of all the information they have; even on a typical
RISC processor, it would be easy to stuff 4 character variables, or 32 booleans, into a single hreg transparently, leaving the programmer none the wiser. Instead of this, such languages make half-hearted attempts to save a little main memory using the vulgar vernacular of C-style bitfields\(^1\) and Pascal family packed arrays.

Such "features" are ruled out by the typeless philosophy, leaving us to place the burden of optimisation on the compiler. Just as we managed to perform advisory type checking without types, so we will attempt storage packing by recovering type sizes from value ranges; and, since we are not constrained by the quantisation of types to byte units, we will probably do a better job.

Section 10.4 explains what we can do about memory references, and 10.3.4 discusses the packing of values in registers, but before doing any of this, let us show how to compute the size of a value (in bits) from its range.

### 10.3.3.2 Determining the Size of a Value

The size of a value is the number of bits needed to describe it. In general, we do not know the value itself, but we do know the range of values it can take, and can produce a safe estimate of its size. The discussion below assumes that values are stored as their binary equivalents, with no compression of any sort other than the removal of high-order zero bits.

All value kinds describing pointer values have the same size, since there are no restrictions on where the heap, stack, or static data, and the various procedures and labels, will end up in memory. This size is architecture-dependent; it is usually the same as the word size \(W\), but there are exceptions. For example, some ARM processors have 32-bit words with 26-bit addresses, while some operating systems use 32-bit addresses on 64-bit processors (e.g. Solaris on UltraSPARC, Windows NT on Alpha). We use \(A\) to denote the address size.

Integer values can usually be encoded in fewer bits. Assuming a two’s-complement notation, values belonging to integer ranges that are known to be non-negative require a number of bits one less than the smallest integer greater than the base-2 logarithm of the maximum of their range:

\[
\log_2(\max(v)) + 1 - 1
\]

where \(v = \text{FLATTEN}([ \text{PROJ}_{\text{VALS}}(\text{PART}_{\text{INTS}}(r))] \) and \(r\) is an integer value range.

Ranges containing negative values have high-order bits set, so their size must be assumed to be equal to the word size unless arithmetic can be done to a precision other than \(W\), or extra code is introduced to reconstitute the high-order bits when the value is to be used.

If one of these options is available, we can reduce the storage required for integers that can be negative to roughly one bit more than for integers that can only be zero or positive:

\[
\max([\log_2(\max(v)) + 1], [\log_2(\min(v))] + 1)
\]

Note that negative values take marginally less space that positive values of the same magnitude; \(-4\) can be represented in three two’s-complement bits, but \(+4\) cannot.

If the extra code to reconstruct the original value is allowed to grow, we might conceive of increasingly complex procedures to reduce the number of bits needed: We could negate all-negative ranges, and add or subtract constants

\(^1\)The ISO C definition of a structure bitfield, allowing different meanings for signed, unsigned and plain int while disallowing enumeration types, is a masterful display of semantic scarology that could only have originated with a standards committee.
to minimise the average magnitude of the range. Even if we cannot reduce the size of negative integers to less than $W$, however, we should not despair; many important values, such as booleans and array indices, are usually non-negative and will benefit from the optimisations we can perform on them.

Given a value range $r$, then, we will assume that any value in it can be represented using

$$
\text{Size}_{abv}(r) = \max \{ |A| \mid \text{Part}_{\text{addr}}(r) \cup \text{Part}_{\text{misc}}(r) \neq \emptyset \} \cup \\
\{ |W| \mid \text{Part}_{\leq 0}(\text{Part}_{\text{ints}}(r)) \neq \emptyset \} \cup \\
\{ |\log_2(\max(\text{Flatten}(\text{PROJ}_{\text{vreg}}(\text{Part}_{\text{ints}}(r)))) + 1)| - 1 \mid \\
\text{Part}_{> 0}(\text{Part}_{\text{ints}}(r)) \neq \emptyset \}
$$

bits.

### 10.3.4 Register Stuffing: Theory

#### 10.3.4.1 The Idea

All register allocators, including the one we presented in chapter 5, regard the typical RISC register file as a collection of $n$ registers, each of which can hold a value. The general idea we would like to pursue is to view the registers as an $n \times W$ array of bits and the values of vregs as $1 \times k$ bitvectors, where $k$ is the size of the value as determined by the methods of the previous section. We then need to tile the bit array with these vectors so as to minimise wastage.

#### 10.3.4.2 The Obstacles

This is a highly non-trivial undertaking: It is essentially a bin packing problem with $n$ bins of size $W$—and such problems are well known for being NP-complete and thus computationally intractable at the scale required here. (See [MT90] for a thorough analysis of bin packing and the related knapsack problems.) Furthermore, the discussion so far only considers one insn; the packing will have to be recomputed, or at least modified, for every insn, while any or all of the following are happening:

- Old values die; new ones, probably of different sizes, are born and must be accommodated with a minimal amount of shuffling.
- Flowgraph paths merge and divide, enforcing constraints on the location of values at basic block limits.
- Values are unpacked so that packing-unaware machine instructions can operate on them; the number of hregs reserved for this reduces the number of bits available for vregs but increases the number of values eligible for packing.
- The exact placement of values in the hregs affects the cost of unpacking in a highly machine-dependent way, so the cost of each placement does not depend only on the size of the value; features of generalised knapsack problems creep in.
- When enough hregs are available, we can save some processing by packing them more sparsely; but then calls cost more, since there will be more live hregs to save.
- Occasionally, we run out of space and must spill some values; relying on the
distance to next use is not enough since we can now spill more than
one value at once.

The above observations also indicate that we can no longer treat allocation and
assignment as separate processes, since the number of vregs we can house in
the register file is not constant and depends on the location each of them is
assigned.

If one is unfazed by the above, consideration of the extent to which architec-
tural features affect packing should be enough:

- Access to the floating-point register file, which can be used to store integers as well; some processors (e.g. the Intel ia32 series) can only access it via main memory.

- Extent of bit-twiddling available, such as the mask generation instructions of the Power architecture, the low-cost (and on some versions, free) shifts of the ARM, or the bit test and modify instructions of most CISC architectures.

- Types of memory access available; e.g. the half-word load/stores of the
MIPS, the multiple load/stores of the ARM, and the rotating unaligned
accesses of some RISC machines.

- Availability of pseudo-vectorised operations, such as Intel MMX, Sun VIS, or Motorola AltiVec [Mot98] extensions.

10.3.4.3 A Compromise

The register tiling problem is intractable, but this does not mean we have
to give up on the whole idea. Rather than attempting to solve the complete
binding problem in one fell swoop, we can start with the two-phase process
we already have and ask whether we can use the tiling idea to improve the
code it generates.

It turns out that we can: Register stuffing is an optimisation algorithm that
sits between the allocator and assigner, operating on the span datastructures
generated by the former and consumed by the latter. It attempts to reduce
memory accesses by stuffing values that would normally be spilt into the un-
used high-order bits of other vregs, and extracting them again when the cor-
responding reloads are encountered.

Since it only depends on the spans, register stuffing can cooperate with any
allocation or assignment algorithm, as long as the two are distinct. It does
require that spans carry value size information, but this is no problem, since
we are already running the register allocator from within the range analysis
engine to allow the optimisations of section 10.3.2. We simply define the last
entry of a span tuple, which we have, with amazing foresight, left unused until
now:

\[
\langle v, i_b, i_e, \text{BOUND}_{\text{beg}}(v, i_p), \text{BOUND}_{\text{end}}(v, i_e), n \rangle \in \text{Spans} \Rightarrow n = \text{SIZE}_{\text{abv}}(A_v[v])
\]

where

\[
\langle A_v, A_m \rangle = C_{\text{abs}}(i_p, i_e)
\]

is the abstract context just before the end of the span at insn \(i_e\).
10.3.4.4 Finding Host Candidates

In order for a vreg $v_h$ to be a suitable host for a spilt vreg $v_s$, all of the following conditions must be satisfied:

1. The insns during which $v_s$ remains spilt must all belong to the same live range of $v_h$.
2. $v_h$ must be in an hreg at all the relevant reloads and spills of $v_s$, although it can be spilt and reloaded in between.
3. The spill of $v_s$ must not be compulsory. Compulsory spills occur when a vreg's memory copy must be synchronised with its hreg-resident one, and include precautionary spills and spills of globals before procedure calls (see section 5.6.3); they are indicated by uspil span boundary conditions.
4. The size of the value of $v_s$ during the spill, plus the size of the value of $v_h$ during the live range that covers it, must be less than or equal to the word size $W$.

To express these conditions we must first formalise two notions we have been using casually. A *live range of a vreg* $v$ is a set of successive insns during which $v$ remains live and is not redefined; if we subdivide the program into maximal such sets, each insn $i$ will belong to at most one live range:

$$j \in \text{LIVE}(v, i) \iff v \in \text{LIVE}(j) \land (j = i \lor k \in \text{LIVE}(v, i) \land (j \in \text{SUCC}(k) \land v \not\in \text{DEF}(k)) \lor (j \in \text{PRED}(k) \land v \not\in \text{DEF}(k))))$$

A *spill range of a vreg* $v$ is a set of successive insns during which $v$ is live and spilt. Again, each insn will belong to at most one spill range if we consider the largest possible ones:

$$j \in \text{SPILL}(v, i) \iff v \in \text{LIVE}(j) \land v \in \text{SPILL}(j) \land (j = i \lor k \in \text{SPILL}(v, i)) \land (j \in \text{SUCC}(k) \lor j \in \text{PRED}(k)))$$

Each live range contains a number of spans, and each spill range has a fringe of spans around it. These are given by

$$\text{LIVE}(v, i) = \{ s \mid s \in \text{Spans} \land \text{PRO}(v, s) = v \land \text{PRO}(s) \in \text{LIVE}(v, i) \}$$

and

$$\text{SPILL}(v, i) = \{ s \mid s \in \text{Spans} \land \text{PRO}(v, s) = v \land (\text{PRO}(s) \in \text{SPILL}(v, i)) \cap \text{SPILL}(v, i) \neq \{\} \}$$

The first condition above requires that the relevant spill range be a subset of one of the host's live ranges; the next two depend on what happens at the boundaries of the spill range. The second condition can be expressed as:

$$\text{NOTSPILL}(v_s, v_h, i) = \forall s \in \text{SPILL}(v_h, i) .$$

$$\text{PRO}(v_h, s) \not\in \text{SPILL}(v_s, i) \lor v_h \not\in \text{SPILL}(\text{PRO}(v_h, s))) \land$$

$$\text{PRO}(v_s, s) \not\in \text{SPILL}(v_h, i) \lor v_h \not\in \text{SPILL}(\text{PRO}(v_h, s)))$$
which is true if the host \( v_h \) is in an hreg at the insns forming the boundary of the spill range of \( v_s \). This is equivalent to ensuring that these insns lie within spans of \( v_h \), which is more convenient for the algorithm to check, as we will see later.

The third condition only depends on the kinds of spills at the boundaries of the range:

\[
\text{AVOIDABLE}(v, i) = \forall s \in \text{SPILLSPANS}(v, i).
\]

\[
(\text{PROJ}_{\text{end}}[s] \not\in \text{SPILLRANGE}(v, i) \lor \text{PROJ}_{\text{end}}[s] \neq \text{uspil})
\]

is true if none of the spill range's "upper" fringes (i.e., the places where \( v \) is spilt, rather than reloaded) contain unavoidable spills.

For the fourth condition, we need to retrieve the value sizes from the corresponding spans:

\[
\text{LIVESIZE}(v, i) = \max\{\text{PROJ}_{\text{size}}[s] \mid s \in \text{LIVESPANS}(v, i)\}
\]

\[
\text{SPILLSIZE}(v, i) = \max\{\text{PROJ}_{\text{size}}[s] \mid s \in \text{SPILLSPANS}(v, i)\}
\]

The value of a vreg is the same all through a live or spill range, but the value range (and hence, the value sizes in the spans) may vary according to the deductions our abstract interpretation engine can make. This is why every span has to be checked and the maximum value size used.

Hosting a spilt vreg imposes a penalty on references, since the host vreg must be extracted and its top bits wiped whenever it is referenced. Thus, the host candidates that meet the above conditions are not all equally desirable; their relative desirability will depend on the number of times they are accessed, a number available in the corresponding spans. We will denote the actual rating by \( \text{HostPRI}(v_s, v_h, i) \) for host vreg \( v_h \) and spilt vreg \( v_s \) in the spill range containing insn \( i \), and will discuss its definition later.

The suitable hosts, and their ratings, for a vreg \( v_s \) at insn \( i \) are then

\[
\text{HOSTS}(v_s, i) = \{ (v_h, \text{pri}) \mid \text{SPILLRANGE}(v_s, i) \subseteq \text{LIVERANGE}(v_h, i) \land
\]

\[
\text{SPILLSIZE}(v_s, i) + \text{LIVESIZE}(v_h, i) \leq W \land
\]

\[
\text{AVOIDABLE}(v_s, i) \land \text{NOTSPILT}(v_s, v_h, i) \land
\]

\[
\text{pri} = \text{HostPRI}(v_s, v_h, i)
\]

### 10.3.4.5 Rating Host Candidates

In selecting a host in which to stuff a spilt vreg, we may apply a number of different criteria. If we want to minimise code growth, we must sum together all the references to each host, and select the one with the lowest total:

\[
\text{HostPRI}(v_s, v_h, i) = \left\{ j \mid j \in \text{SPILLRANGE}(v_s, i) \land v_h \in \text{REF}_{v}(j) \right\}
\]

If we care more about execution speed than code size, the number of references to the host on each path through the spill range should be averaged, in a calculation similar to that of distances to next use (see section 5.5). Of course, both ratings can be considered in some sort of hybrid heuristic.

However, the discussion of code growth due to stuffing does point out an important issue: If the memory accesses eliminated by stuffing are replaced by a similar number of instructions to extract the hosts, instructions which themselves have to be fetched from memory, do we actually gain anything?

The answer is that instruction accesses are different in a number of ways from data memory accesses. First of all, they are more predictable; branch
predictors and pre-fetch units can do a very good job of loading them into the CPU in advance while the memory access hardware is not otherwise occupied, making pipeline stalls less likely.

Secondly, stuffing reduces scheduling pressure: Even when all the relevant memory is in the cache, there are usually architectural constraints on when the results of loads and stores become valid. With few exceptions (e.g. early MIP processors and the Intel i860 family [Mar90]) these constraints are invisible since the pipeline can be stalled transparently, but a good schedule will always make the program go faster, and stuffing increases the probability of finding one.

Finally, the overhead is negligible on some processors, such as early ARMs which could do shifts for free. This will be discussed further later on.

### 10.3.5 Register Stuffing: Practice

#### 10.3.5.1 Introduction

Rather than looking at the whole spill range at once, the register stuffing algorithm scans through the code sequentially. When it encounters a spill range, it compiles a list of vregs that look like suitable host candidates at that point, and whittles it down as more constraints become apparent in its movement through the code. Candidate sets are carried through the entire spill range, intersected where flow graph paths meet, and propagated to any branches.

One interesting feature of the algorithm is that it can be applied in either direction (i.e. in both preorder and postorder scans of the flowgraph). The following formulation is for a postorder (backwards) scan, which is easier to integrate with the rest of our implementation since it allows live sets to be regenerated on the fly.

#### 10.3.5.2 Datastructures

The algorithm maintains lists of the vregs it is trying to accommodate and their potential hosts. There is one such "stuffing list" for every basic block, updated during each backward scan. The information it holds for each vreg includes:

- The size of the value it holds.
- A marker indicating the status of its spill range:
  - `exit`, if it extends past the end of the block;
  - `entry`, if it extends past the beginning and end of the block;
  - `dead`, if it is contained in the block and not currently in effect; or
  - `other`, if it starts in the block and is currently being processed.
- A set of host candidates, and a priority for each.
- An indication of whether a candidate has been selected.

If a block contains (parts of) more than one spill range for a vreg, that vreg will have a similar number of entries in the stuffing list.
10.3.5.3 Host Candidate Culling

Given a vreg $v_h$, the following algorithm decides whether it could be a suitable host for a spill range of $v_s$ which goes through (part of) block $b$. If $v_s$ is reloaded in $b$, its first span after the reload, $s$, is supplied; otherwise, $s$ is undefined. The value of $v_s$ in question is of size $z$.

**Algorithm 10.2 (Host Candidate Culling)**

1. If $s$ does not exist or is the first span of $v_s$ in $b$, find the first span $h_1$ of $v_h$ in $b$; if it does not start with a cont condition, give up.

2. If $s$ exists and is not the first span of $v_s$ in $b$, search for a span $h_1$ of $v_h$, during which the span of $v_s$ preceding $s$ ends; if no such span exists, give up.

3. If $\text{PROJ}_{\text{size}}(h_1) + z > W$ (i.e. $v_h$ does not have enough free bits), give up.

4. Examine every span $h'$ of $v_h$ from (but not including) $h_1$ up to (and including) the last span $h_n$ starting before $s$ (or the end of $b$, if $s$ is undefined). If any of them are such that

   \[ \text{PROJ}_{\text{beg}}[h'] \not\in \{\text{spil, move}(v)\} \]

   (i.e. $v_h$ is redefined while $v_s$ is spilt), give up.

5. If $s$ exists and $h_n$ ends before $s$ starts (i.e. $v_h$ does not have a span at the point where $v_s$ is reloaded), or $s$ is undefined and $h_n$ does not end at the end of $b$ with a cont condition (i.e. $v_s$ is not reloaded in $b$ and $v_h$ is not left alone until the end of the block), give up.

6. Return the number of references to $v_h$ in $b$ overlapping with the spill range of $v_s$; to avoid examining the code, this can be approximated by

   \[ \sum_{h=h_1}^{h_n} \text{PROJ}_{\text{size}}[h] \]

   with $h_1 \ldots h_n$ as above.

10.3.5.4 Populating the Stuffing Lists

The stuffing lists of basic blocks are all initially empty. The following algorithm is applied to each basic block in repeated postorder traversals of the flowgraph, until the contents of stuffing lists on entry to every block (i.e. the vregs present in the list, and the candidates in each vreg's entry, but not the costs thereof) stabilise:

**Algorithm 10.3 (Host Candidate Detection)**

1. For each spill vreg $v_s$ appearing in any of the stuffing lists of the current block's successors with an entry marker, do:

   (a) Add an entry for $v_s$ in the current block's list, with an exit marker and an indication that no candidate has been chosen. Only add candidates $v_h$ present in every successor block in whose list $v_s$ is present and marked entry. Set the cost of each $v_h$ to the sum of the costs in the successors, or their average, depending on the rating method chosen; see section 10.3.4.5.
(b) Replace the candidate sets of $v_s$ (and their attendant costs), and the flag indicating whether a candidate has been chosen, in the successors' stuffing lists with references to the new candidate set, so they inherit any further changes automatically.

2. Scan all vreg spans for this block in reverse (i.e. the span that starts last is scanned first). For each span $s$ belonging to some vreg $v_s$ that starts with a spill condition (i.e. begins with a reload) do:

   (a) Allocate a new entry for $v_s$ at the head of the stuffing list; give it an other marker, an empty candidate set, and indicate that no candidate has been chosen yet.

   (b) Apply algorithm 10.2 to each vreg $v_h$ that is live and not spilled at the point where $s$ starts; if a cost value is returned, add $v_h$ and its cost to the stuffing list entry for $v_s$ in the current block's list.

3. For each span $s$ belonging to some vreg $v_s$ that ends with a spill condition (i.e. terminates at a spill) do:

   (a) Find the first entry for $v_s$ in the stuffing list.

   (b) If it has an exit marker, we need to restrict its candidates to those allowed in the current block; apply algorithm 10.2 to each of them, removing the ones that fail and updating the costs of the others. Then remove it from the stuffing list. Any changes to its candidates will have been inherited by the lists of downstream blocks due to the redirection in step 1b.

   (c) If the entry has an other marker, change it to dead and leave it in the list.

4. After all the block's spans have been processed, apply algorithm 10.2 to each of the candidates of every entry with an exit marker, removing those that fail and updating the costs of the others as in step 3b.

5. Vregs with entries bearing exit or other markers at this point may also have entry entries left over from the previous pass of the algorithm. Merge the former into the latter by intersecting the candidate sets as in step 1a, and giving the result the entry marker.

10.3.5.5 Convergence

Ignoring candidate sets for the moment, it should be obvious that the vregs present in the stuffing list at any given point in the program converge: there is an upper limit (the entire set of vregs) and the list never shrinks from one iteration to the next (trivial induction argument).

For any given vreg entry in some stuffing list, we now need to prove that the contents of its candidate set converge (ignoring the costs, which increase on every iteration but remain proportionately valid). The candidates originate either in the block containing the list, in which case they are constant, or in its successors.

In the latter case, taking advantage of the fact that the merging operation at the exit from the block produces a candidate set no bigger than the smallest candidate set in the successor blocks' lists, we can again use induction to show that the candidate list does not grow between iterations after the vregs present in the lists have stabilised.
10.3.5.6 Stuffing Vregs: Introduction

When the process converges, the stuffing list of each block will contain one entry for every spill range passing through the block; if the ordering of the list is maintained, the dead entries will be ordered by the locations of corresponding reloads in the block. We now have to

- choose a host candidate for each spill range,
- ensure that it is not used to stuff any spill range that overlaps with it even partially,
- rewrite the spills and reloads, as well as all references to the host in the spill range, doing the necessary bit-twiddling to stuff and extract the values, and
- modify the span datastructures to indicate these changes.

The last step is necessary because it is conceivable that the host will have a few bits left over even after the guest is accommodated. By introducing new spans to indicate the locations and aggregate value sizes of stuffed vregs, we can apply the whole process again and try to stuff more data into them. We need to introduce two new span boundary conditions:

- `host(v)` indicates that the vreg concerned starts (or stops) being a host for a stuffed vreg `v` at the point where the span begins (or ends).
- `guest(v)` is intended to replace the spill conditions of vregs that have been stuffed into the upper bits of some host `v`.

10.3.5.7 Stuffing Vregs: Splitting Spans

The span of a host vreg `v_h` active at a reload insn `i` for guest vreg `v_s` is split into two and rewritten as follows:

\[
\langle v_h, i_b, i_e, f_b, f_e, \text{ref}, n \rangle \rightarrow \begin{cases} 
\langle v_h, i_b, i, f_b, \text{host}(v_s), \text{ref}/2, n + m \rangle \\
\langle v_h, i, i_e, \text{other}, f_e, \text{ref}/2, n \rangle
\end{cases}
\]

where `m` is the number of additional bits carried by `v_h` while it is stuffed.

Note that the actual reference count in each of the two parts is estimated as half the reference count of the original span; this is not too bad an approximation, since spans are confined within basic blocks and will not usually encompass many insns.

Also note that we use `other` as the boundary condition for the unencumbered span; this is reasonable for everyday architectures where the stuffing and unstuffing operations do not introduce hreg binding preferences.

The span of `v_h` at a spill of `v_s` is split similarly:

\[
\langle v_h, i_b, i_e, f_b, f_e, \text{ref}, n \rangle \rightarrow \begin{cases} 
\langle v_h, i_b, i, f_b, \text{other}, \text{ref}/2, n \rangle \\
\langle v_h, i, i_e, \text{host}(v_s), f_e, \text{ref}/2, n + m \rangle
\end{cases}
\]

The guest vreg does not gain any new spans, but the boundary conditions at the fringes of its spill range change. If `i` is a spill, we have

\[
\langle v_s, i_b, i_e, f_b, f_e, \text{spil}, \text{ref}, n \rangle \rightarrow \langle v_s, i_b, i, f_b, \text{guest}(v_h), \text{ref}, n \rangle
\]

and if it is a reload,

\[
\langle v_s, i_b, i_e, f_b, f_e, \text{ref}, n \rangle \rightarrow \langle v_s, i_b, i, f_b, \text{guest}(v_h), \text{ref}, n \rangle
\]
10.3.5.8 Stuffing Vregs: Committing to Candidates

Given a stuffing list entry and a list of entries that have already been assigned hosts, the following process is used to find a host for the spill range it represents:

Algorithm 10.4

1. If a host has not already been selected for the spill range,

   (a) Remove the hosts of all entries in the current hosts list from the candidate set of the spill range.

   (b) If there are any candidates left, select the one with the lowest cost in the set, and remove all the others.

   (c) Change the entry's flag to indicate that it has gone through the candidate selection process.

2. Move the spill range entry to the current hosts list.

This procedure is used during the final stage of the stuffing process, which involves a single pass through the intermediate code. The order in which blocks are scanned does not matter, as long as the following algorithm is applied to each block:

Algorithm 10.5 (Host Candidate Selection)

1. Initialise an empty current hosts list.

2. Regenerate entries with exit markers, as in step 1a of algorithm 10.3.

3. Perform algorithm 10.4 for every entry and exit record in the block's stuffing list. This will attempt to assign hosts for spill ranges active at the end of the block, and move all such ranges to the current hosts list.

4. Scan the insns in the block backwards:

   (a) For every unsp (reload) insn:

      (i) Find the last dead entry for the vreg v_s being reloaded in the block's stuffing list, and perform algorithm 10.4 on it. This will attempt to assign a host for the spill range ending here.

      (ii) If a host v_h was found, rewrite the insn appropriately, find the host and guest spans containing the reload insn and split them as described in section 10.3.5.7.

   (b) For every spill insn:

      (i) Remove the corresponding entry from the current hosts list.

      (ii) If the entry has a host, rewrite the spill appropriately, find the host and guest spans containing the spill insn and split them as described in section 10.3.5.7.

      (c) Rewrite every insn referencing any host in the current hosts list as appropriate.

5. Discard the current hosts list.
10.3.5.9 Rewriting Insns: Machine-Specific Considerations

As we have mentioned earlier, different architectures have different varieties of instructions that can be useful in stuffing and extracting regs. For our proof-of-concept implementation, we will only be using bitwise conjunction, disjunction and shifts, which are available on all the processors we have encountered\(^2\), but even with these restrictions, we have a number of choices to make.

The handling of the spill and reload insns themselves is straightforward. If reg R001 can be stuffed into the upper bits of R002, we replace a spill

\[
\text{spil } R001, \ 0, \ 0
\]

with the two insns

\[
\text{lsh } R001, R001, \#cn1 \\
\text{orr } R002, R002, \text{R001}
\]

and a reload

\[
\text{unsp } R001, \ 0, \ 0
\]

with

\[
\text{rsh } R002, R001, \#cn1 \\
\text{and } R001, R001, \#cn2
\]

where \#cn1 is the value size of R002, and \#cn2 = \(2^{\#cn1} - 1\). However, note the use of a logical, rather than arithmetic, right shift; an alternative would have been to align the guest reg with its most significant bit in the host's leftmost free bit, and allow an arithmetic shift to extract the value with proper sign extension. This precludes the stuffing of a second guest into this host, but allows us to handle negative values, which we have been assuming are inappropriate guests (see section 10.3.3.2).

Rewriting the references to host regs is slightly trickier. To utilise the hosts fully, we will need to reserve at least one reg into which to extract their values in order to operate on them without interference from guests: this implies that stuffing is not worth the effort unless we can stuff two regs at the same time. However, with two stuffed hosts, the possibility of an operation involving them both arises, and in some of these cases one auxiliary reg is not enough.

Consider, for example, the insn

\[
\text{add } R001, R002, R003
\]

and suppose that R002 holds another value in its high-order bits. To render the result of the addition valid, we can mask out R002's guest before the operation:

\[
\text{and } \$aux, R002, \#con \\
\text{add } R001, \$aux, R003
\]

where \$aux is the auxiliary reg and \#con is a constant mask covering the size of the value of R002 with ones. However, if R003 also has a guest, this is not enough; using the same approach, we would need another auxiliary register and one more operation:

\[
\text{and } \$ax1, R002, \#cn1 \\
\text{and } \$ax2, R003, \#cn2 \\
\text{add } R001, \$ax1, \$ax2
\]

\(^2\)Including the idiosyncratic Saturn, mentioned in a footnote to section 6.9.
We can sometimes avoid some of the overhead of cases such as this. The sequence

\begin{verbatim}
add R001, R002, R003
and R001, R001, #con
\end{verbatim}

is valid if \#con is a mask corresponding to the intended value of the result (R001), and the two operand vregs have been stuffed as if they had the same size as the result.

This operation uses no auxiliary hreg, but it reduces the number of bits available to the guests. Moreover, it only works with operations in which the flow of information is not from most to least significant bits: addition, multiplication, shifts to the left and bitwise logical operators qualify, but subtraction, division and right shifts do not.

Thus, the amount of stuffing we can do with conventional operations depends on the number of hregs we set aside; and we must achieve at least one stuffing per auxiliary hreg to expect any results. Computing the effectiveness of the tradeoff in advance is not easy, and given the small number of stuffings achievable, probably not worth the effort. Our implementation reserves one hreg, and refuses to use a vreg as a host if it is involved in an operation with another host, unless the insn is an addition or logical operation, in which case it tries to use the last technique described above. We believe that this represents a reasonable compromise in most common circumstances.

### 10.3.6 Register Stuffing: Implementation Details

The P compiler's implementation of register stuffing follows the algorithms described above, with a few minor differences. We have just discussed the additional constraints on stuffing to minimise auxiliary hreg use; another difference is that the implementation scans the code on every iteration and not just when the insns have to be modified, because it needs to regenerate liveness information. Since it is scanning the insns anyway, it is more efficient for the implementation to detect spills and reloads by looking at the opcodes rather than the span limits; spans are looked up and examined only when such insns are found. Finally, the current version of the code does not attempt to nest stuffed vregs, and thus does not need to rewrite the spans of affected registers as described in section 10.3.5.7.

The sharing of host candidate sets is assisted by a form of garbage collection; each candidate set has a reference count, which is initially set to 1, incremented every time some block's stuffing list acquires a reference to it, and decremented every time a reference is removed (steps 1b and 3b of algorithm 10.3 respectively). When the reference count reaches zero, the set itself is discarded.

### 10.3.7 Register Stuffing: Conclusions

We have mentioned that register stuffing is independent of allocation and assignment strategies, as long as span datastructures are available for it to work on. Unfortunately, the style of allocation can affect stuffing performance, and our allocator is particularly bad in this respect: stuffing is happy with many nested live/spill ranges, but the allocator does its best to reduce non-nested spill ranges by deferring reloads for as long as possible.

Although we have found cases where register stuffing is advantageous in practice, a combination of compiler limitations and the sizes of the programs
involved conspire to prevent us from including any examples here. The architecture we are compiling for is also partly to blame: the rather elderly MIPS R3000 is slow enough to afford a one-cycle cache latency, so stuffing is not worthwhile for very busy hosts. However:

- Other operating systems should do better. On our reference platform, registers holding addresses cannot be stuffed at all, but we have already mentioned (section 10.3.3.2) other systems where two addresses can be stored in the same hreg.

- Other architectures should do better. Contemporary processors hide enormous latencies behind the interlocks of their superscalar pipelines, and stuffing operations can keep idle functional units busy.

- Other allocators should do better. Register colouring produces many nested spill ranges, as some vregs get hregs for their entire lifetimes while others are shuttled to and from the stack whenever they are needed.

- A major tenet of the RISC philosophy is that processor design should be guided by compiler technology advances. Enhanced bit manipulation primitives and wider registers can help: in future architectures, we would like to see operations such as sign extension from arbitrary bits, atomic mask-and-shift, and MMX-like pseudo-vector operations on integer registers.

10.3.8 Variations on the Theme

Aside from spill stuffing, there are other ways we can share registers between values. One obvious example concerns boolean variables.

Traditionally, programs using lots of boolean values implement them as bitfields of a C struct or mask them into and out of integer variables. The former approach is infeasible in a typeless language, and the latter is error-prone and harder to read, but if the target processor has bit manipulation operations, one may use a separate variable for each flag and rely on the compiler to place them all in one hreg.

Even if one does not want to set aside an hreg for boolean values, some architectures have status registers containing little-used flags that are not necessarily updated by every instruction. The ARM is such a design, and its Overflow or Negative flags can be exploited thus between conditionals.

Sadly, the MIPS has no bit operations and no condition flags, so we have not pursued this approach any further.
10.4 String Packing

10.4.1 Can We Do It?

In the previous section, we showed how to pack multiple values in registers, as long as the values are short and the registers long enough. One might be tempted to ask whether it is possible to do the same for main memory.

Computer architectures provide a limited selection of memory access instructions: Typically, values that can be loaded have an aligned address, and their size is a power of two not greater than the word size. With a typed language, we can select the appropriate access size based on the type of the accessed value alone; even in cases where the selection of loads and stores is more limited, we can consider optimisations that trade CPU time, which is usually abundant, for relatively scarce memory bandwidth by packing and unpacking memory values in registers. This is necessary on early Alpha processors, for example, where the smallest accessible memory unit is 32 bits wide.

Typeless languages usually provide a couple of operators to let the programmer select the access size, but aside from introducing asymmetries and potential for error, these cannot take full advantage of architectures which provide three or more sizes. With the proliferation of 64-bit RISC machines that can do 8-, 32- and 64-bit loads and stores, and the continuing popularity of CISC processors most of which allow 16-bit accesses, our inability to utilise these features is becoming embarrassing. For P the embarrassment is even more acute, since we refuse to use more than one memory access operator on aesthetic grounds.

Finally, storage packing enables other optimisations, such as the "poor man's vectorisation" techniques used by Intel's reference compiler and partly responsible for the impressive benchmark scores of the Pentium Pro [Int97b].

10.4.2 How to Do It

Since there is so much scope for improvement, at no extra cost, in the selection of appropriate memory access sizes, we will not examine the stuffing of multiple values in a single memory location. It should be noted that any such optimisation is bound to be similar to register stuffing as described in section 10.3.

Perhaps not surprisingly, memory packing by selection of access size is much simpler than vreg stuffing. This is due to two factors:

**Uniformity of memory objects.** An access to a vector entry is likely to be similar to accesses to any of its other entries.

**Long lifetimes.** Memory objects tend to have long, untraceable lifetimes; assuming that they are alive forever is a safe and almost always accurate approximation, so we do not have to distinguish between live ranges.

An address in a memory block is formed by adding an offset to the base address that identifies the block. To pack the block, we need to modify the offset, but not the base address; thus, the code must be modified both at the actual memory accesses (so that a smaller amount of memory is accessed) and at the insns computing addresses for the accesses (so that the offset is scaled). This implies some conditions to be met by a memory block before we attempt packing:
The insns computing the address for every load and store must be readily identifiable.

All other blocks accessible by the same insns and sharing the offset computations must also be packed.

There should be no undetermined or invalid values in the address range of any load or store possibly referring to the block, since they might access both packed and unpacked blocks.

Detecting memory areas whose values are small enough to be packed is easy: all we need to do is check the initial values of static objects, and the values stored by store insns. If $C_{abs}$ is a context vector solution of the abstract interpretation,

$$\text{PACKABLE}(C_{abs}, n) =$$

$$\{ b | b \in \text{V} \land \forall \langle i_p, i_c \rangle \in \text{A} \land \langle \text{OP}(i_c), b \rangle = \text{stor} \land$$

$$\text{addr} = \text{PART}(\text{addr}, \langle \text{addr} \rangle) = \text{ARGV}_{abs}(C_{abs}(\langle i_p, i_c \rangle), i_c, 2) + \text{abv}$$

$$\text{ARGV}_{abs}(C_{abs}(\langle i_p, i_c \rangle), i_c, 3) \land$$

$$\exists c \in \text{addr} \land \text{PROJ}_{base}(c) = b \Rightarrow$$

$$\text{SIZE}_{abv}(\text{ARGV}_{abs}(C_{abs}(\langle i_p, i_c \rangle), i_c, 1) \leq n) \land$$

$$\{ b | b \in \text{Statics} \land \forall \langle i_p, i_c \rangle \in \text{A} \land \langle \text{OP}(i_c), b \rangle = \text{entr} \Rightarrow$$

$$\text{SIZE}_{abv}(\text{CONTENT}_{0}(\text{PROJ}_{base}(C_{abs}(\langle i_p, i_c \rangle))(b), [0...\infty]) \leq n) \}$$

is the set of identifiers of memory objects whose cells can be represented with $n$ bits each.

Identifying the insns that compute the address is slightly harder, and not always possible. Generally, addresses for memory references are either produced by the translator from source-level indirection operators, or computed additively in a loop (see the examples of figure 10.3): both methods result in recognisable intermediate code idioms, and where these are located, the offsets from the base of the memory block can be adjusted.

If the insns cannot be located, we might be able to regenerate the appropriate address by adding an appropriate offset to the block base, although this
is only possible for memory blocks identified uniquely by the base of their address range—that is, static objects and vectors declared in non-recursive procedures. However, if even one of the loads and stores affecting the block cannot be modified in this way, we will have to leave the block in an unpacked state.

10.5 Language Extensions

10.5.1 Introduction

So far, we have been examining optimisations that can be applied to programs written in a more-or-less traditional typeless language: a language that was not designed with the availability of a range analysis engine in mind. There are many reasons why this is still the best thing to do:

- Not all compilers have range analysers. We do not want to restrict the language to a single compiler architecture. On the contrary, P was designed so that a naive, non-optimising implementation is very easy to produce from scratch.

- The techniques we describe are independent of the source language, and can be used in compilers for other, well-established languages.

- The programmer does not have to be aware of the inner workings of the compiler, or become familiar with yet another programming paradigm.

However, it is worth investigating what we could do in a world where range analysers are ubiquitous and the above points are moot. A number of suggestions are considered in section 10.5.2.

In section 10.5.3, we consider some less exotic features already present in other languages, whose implementation can be optimised by the use of range analysis.

10.5.2 Extensions to Range Analysis

10.5.2.1 Assertions

ISO C provides the assert() macro, which terminates the program at run time if its argument, evaluated as a logical expression, is false. A similar facility can be provided in P, with the added bonus that some assertions can be checked and warned about at compile time; abstract debuggers [Bou93] are already capable of this.

However, we can go one step further by introducing another type of assertion, which the compiler is meant to accept without checking. This would be useful in informing the range analyser of facts it cannot possibly derive (e.g. the expected values of data read from files), and in refining its assumptions and conclusions around particularly tricky pieces of code. It can also be used as a generalisation of traditional type checking, which essentially enforces assertions before every use of a typed value.

Such "forced" assertions could be implemented by compiling the logical expression and then restricting the abstract context with respect to a truthful outcome. Some inconsistent assertions could be detected (by looking for empty, \( \bot_{\text{aby}} \) value ranges in the resulting context), and the assertion code could be either discarded before final code generation, or left in the program to catch run-time assertion failures.
10.5.2.2 Fuzzy Values

In some cases, the programmer does not really care about the value of an expression, as long as it satisfies certain conditions. A typical example is the return value of the string `strcmp` procedure, which has to be zero if the two arguments point to identical strings, and less (or greater) than zero if the strings are (or are not) in lexicographic order. The actual non-zero return values do not matter, but when implementing such a routine, the programmer is often forced to use some arbitrary value (typically ±1) simply because there are no appropriate values in any live variables.

There is another problem, besides the potential inefficiency: users of the value will undoubtedly come to depend on its undocumented constancy, and a later change may cause problems even if it preserves the advertised behaviour.

One way around this is to allow the programmer to specify ranges of values as well as constants; for example, the expression `1..10` would denote an unspecified value from 1 to 10. When such a range is used, the compiler could scan its abstract context for an appropriate value in the existing vregs (most of which are temporaries not visible to the programmer), and generate a new value only if nothing suitable is found. With a little more effort, the compiler could keep track of, and scavenge, the values left over in hregs when vregs die.

10.5.2.3 Source-Level Range Arithmetic

Having introduced fuzzy values, and given the existing framework for dealing with value ranges in the compiler, we might as well make range arithmetic available to the programmer: operations on fuzzy values can yield values in the expected result ranges. We do not have any applications in mind for this idea, but as programmers become more familiar with range-analysing compilers, it is doubtless that they will find some.

10.5.3 Other Language Extensions

10.5.3.1 Exceptions

Exceptions are a way of informing the program that some extraordinary condition has arisen. An exception causes control to be transferred to the `exception handler` surrounding the point where it was `raised`: if the handler does not `catch` it, it is forwarded to an exception handler in the next procedure up the
calligraph. Uncaught exceptions ultimately cause the run-time system to abort the program.

One way of implementing exceptions is by using a second return value for procedure calls; a non-zero value indicates an exception, and an appropriate conditional after each call can transfer control to the handler if necessary. This constitutes a significant overhead, which is partly avoidable in a typed language where the set of exceptions a procedure can raise is known at compile time.

In typeless languages, exceptions are just additional return values, and their ranges can be treated as such; determining the exceptions raised by a procedure is no harder than determining its possible return values (we will see how to do this in the next chapter), and the handler’s conditionals can be optimised away wherever possible by the established interaction between our range analyser and the dead code eliminator.

10.5.3.2 Higher-Order Procedures

At the moment, a procedure call with some arguments missing will produce a warning but will proceed as normal, with the unspecified arguments having undefined values. To turn P into a true functional language, we would like such partially specified calls to result in carrying: the production of a procedure taking the remaining arguments and returning the original procedure’s result.

Entry stubs for curried procedures would be allocated on the heap (and deallocated using standard library calls), but the range analyser should be able to optimise some of these indirect calls away, and possibly optimise the procedures with respect to curried arguments, as described in the next chapter.
### Figure 10.6: WIM3 code for the procedure of Figure 10.4, after optimisation.

<table>
<thead>
<tr>
<th>Basic block</th>
<th>Predecessors</th>
<th>Successors</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td>5 (f), 6 (f)</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>10 (c)</td>
<td></td>
</tr>
</tbody>
</table>

**Basic block 2**

**Predecessors:** 6, 6

**Successors:** 5 (f), 6 (f)

**Code:**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>entr</td>
<td>0, 1, 1</td>
<td></td>
</tr>
<tr>
<td>1:</td>
<td>add R104, $16, 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2:</td>
<td>add R105, $31, 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:</td>
<td>add R106, $4, 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4:</td>
<td>add R109, $0, 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5:</td>
<td>jeq L5, R106, R109 // block 5 if argc=1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6:</td>
<td>jeq L6, 0, 0 // block 6 otherwise</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Basic block 5**

**Predecessors:** 2

**Successors:** 10 (f)

**Code:**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>add R111, n0, 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1:</td>
<td>add $5, #1, 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2:</td>
<td>add $4, R111, 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:</td>
<td>call g75, #2, #2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4:</td>
<td>hack 0, 0, 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5:</td>
<td>jeq L10, 0, 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Basic block 6**

**Predecessors:** 2

**Successors:** 10 (c)

**Code:**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>add R113, $2, 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1:</td>
<td>add $5, R106, 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2:</td>
<td>add $4, R113, 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:</td>
<td>call g75, #2, #2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4:</td>
<td>hack 0, 0, 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5:</td>
<td>jeq L10, 0, 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Basic block 10**

**Predecessors:** 6, 5

**Successors:** 10 (f)

**Code:**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0:</td>
<td>nob</td>
<td>0, 0, 0</td>
<td></td>
</tr>
<tr>
<td>1:</td>
<td>add R103, R107, 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2:</td>
<td>add $3, R103, 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3:</td>
<td>add $3, 0, 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4:</td>
<td>add $16, R104, 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5:</td>
<td>add $31, R105, 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6:</td>
<td>rets 0, 0, 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

### 10.5.3.3 Argument Vector Abstraction

P does not have variadic procedures (i.e., procedures taking a variable number of arguments). Languages that do, such as C, suffer from ugly syntax (typically macro calls) that is restrictive (e.g., unnamed arguments must be accessed sequentially) and often has to be detected and treated specially by the compiler to produce efficient code.

One way around this problem is to create the illusion that all procedures take the address of an argument vector as their single argument (in P, this is even consistent with our syntax for vectors and calls). Some of the arguments are named, and the names are treated as aliases of the first few vector cells, like the contents of MATCH vectors in MCPL; the rest are accessed using the indirection operator.

An implementation adhering strictly to this paradigm would be inefficient, since it would not take advantage of argument passing in registers\(^3\), but we can optimise it without altering our range analyser in the least: named arguments are no different from ordinary variables whose addresses are taken, and we have already seen how the memory references involved can, in most cases, be elided (see sections 10.2.6 and 10.3.2).

### 10.6 Examples

#### 10.6.1 Introduction

In the following sections we will give some examples of the value range engine's applications. These will be simple by necessity, since we are constrained by the lack of interprocedural information; besides, it is not our intention to swamp the reader with listings of source and intermediate code. We hope that these contrived examples will give an idea of what the optimiser can do for more substantial programs.

\(^3\)This does not matter so much for MCPL, which is designed to be interpreted by a virtual machine with a small number of registers.
10.6.2 Optimisation of Conditionals

Consider the trivial program of figure 10.4 on page 168; the initial intermediate code for it can be found in figure 10.5 on page 169. The ‘==’ operator is defined to return 1 if its operands are equal and 0 otherwise, so the translator emits code which computes this result (blocks 5, 6) and then checks (block 7) it to decide which branch of the .test to take (blocks 8, 9).

The intermediate result is unnecessary and should be optimised away. Unfortunately, although the compiler can derive the correct values of vreg 110 (the conditional result) on both branches of the .test thanks to associativity in the context, it cannot modify the jumps in blocks 5 and 6, since they are unconditional.

However, we have already seen (section 3.5.2.1) that the dead code eliminator can duplicate small blocks to remove jumps. This is what happens to block 7 on a subsequent pass. Its jumps are copied to the ends of blocks 5 and 6, and are rendered unconditional as described in section 10.2.4. The result can be seen in figure 10.6 on the facing page.

10.6.3 Optimisation of MCPL Matches

MCPL [Ric92] is a typeless language similar to BCPL, with pattern matching abilities borrowed from ML. Pattern matching (done implicitly on procedure arguments, and explicitly using the MATCH command) allows succinct, intuitive implementations of many compilation algorithms, but it also encour-
ages sloppy programming with redundant conditionals. Value range analysis can often remedy this.

P does not yet have a MATCH expression, but we can simulate one using successive conditionals. Figure 10.7 on the preceding page contains an example MCPL procedure and its translation into P. The redundancy in the conditionals should be obvious; it certainly is to our range-based optimiser. By merging blocks, detecting needless comparisons and eliminating them as dead code, it manages to delete 52 of the 81 basic blocks produced by the translator. This simplifies the flowgraph considerably: compare figures 10.8 and 10.9.

Our compiler does not do jump threading (i.e. the conversion of jumps leading to jumps into jumps to the final destination). If it did, it would have been able to optimise further: block 9, for example, contains no useful code, and removing it would allow us to merge blocks 5 and 8, eliminating a conditional.

We cannot exhibit the values predicted for the procedure's vregs, since even the simplified flowgraph is quite complex. We will revisit this example in section 11.9.2, where we will be able to make a concise statement of the analyser's results.
Figure 10.8: Flowgraph for the procedure of figure 10.7 before optimisation.
**Figure 10.9:** Flowgraph for the procedure of figure 10.7 after optimisation.
Chapter 11

Optimisations Across Module Boundaries

11.1 Introduction

It is obvious to even the most inexperienced programmer that no program of substantial size can be compiled as a whole. This is because

- An error early on in the program can affect the course of the compilation, making any further diagnostic output useless.
- The time taken to recompile after correcting trivial errors is prohibitive.
- The computing resources available may not be enough to handle the whole program in one go.

Aside from the above, there are other good reasons for splitting a program into separately compiled fragments, or translation units (TUs):

- Code may be reused in other projects.
- If changes can be confined to the internals of a fragment so that they are not visible at its interface, the rest of the program need not be modified.
- Specification of the fragments is easier, and makes the distribution of work to members of a team simpler.

These points have been obvious for some time, and all languages currently in use provide some sort of separate compilation system, ranging from the primitive, preprocessor-based #include mechanism of C to the sophisticated structure-signature modules of ML [Mac86]. All such systems have one basic feature in common: they distinguish between TU interfaces, which are visible to other TUs, and implementations, which are not\(^1\).

However, code modularisation is largely inconsistent with optimisation: the optimiser needs information on the context in which the code it is working on will be used, but details of the internals of modules are not available to other TUs. Thus, for example, the values of external datastructures cannot be assumed to lie in any range, and calls to external procedures must be assumed to have potentially any effect on the program's state.

\(^1\)Although in C the distinction is blurred, since executable code is often included in header files as inline procedures or macro definitions.
Traditional languages and compilers do not provide much in the way of inter-TU information, although type checking, where available, does help. For our value-tracking optimiser, we need to do better: we have to find a way to propagate information about value ranges and procedure side effects across TU boundaries.

11.2 Overview of Related Work

Despite the wealth of research into interprocedural optimisation, little work has been done on extending it beyond TU boundaries. The simplest, and commonest, approach involves compiling to intermediate, rather than object, code and optimising at link time. Register allocation has been handled this way [Wal86, Wal92], with encouraging results. Of course, such methods are incompatible with existing linkers and legacy code.

A more interesting approach [SO90] uses a program database in which register usage information is placed by the compiler. Subsequent runs can query the database and optimise accordingly.

Unfortunately, more general-purpose analyses like the extraction of reference and definition information for global variables [Cho83] have been shown to make almost no difference on scalar processors [RG89], although they are important in parallelising compilers [Cal88]. Integrated parallel program development systems, like PIPS [IJT91] and FIAT [HMCR95], have evolved from earlier systems like [CT86]; they all work by placing summary information about the program's procedures in a centralised database.

Our framework uses a distributed database, and places in it much more information that is available to conventional optimisers. In addition to enabling optimisation of calling procedures, this allows us to perform a form of program specialisation (see section 11.6.5). Specialisation by partial evaluation is common for functional languages; see [JGS93] for a comprehensive overview of the field. More recently, the technique has been applied to imperative languages like C [CN96, And94].

11.3 Interfaces and Annotations

11.3.1 What Is an Annotation File?

An interface is the user-visible "surface" of a module. In typed languages like Modula-3, it contains a wealth of information; nested interface inclusion is allowed, and the syntax is almost as complex as that of the module implementation.

As we saw in section 2.5, P interfaces are just lists of names, with no other information whatsoever. This is not of much use in optimisation, but we would rather not change it: nothing more is needed for unoptimised compilation, and we believe that the optimisation process should be entirely invisible to the user. Besides, any extra information would have to be supplied by the user, thus violating the language's typeless nature.

Our solution is to have the compiler generate an annotation file for each module implementation, and place all the information it considers relevant in it. This file is then read by compilation runs working on TUs that import the interface, and the information used to enhance the optimiser's performance. Annotations breach the insulation of modules from each other, but they do so "under the surface"; the average programmer will never need to read an
annotation, and can pretend that the interfaces are all the compiler knows about other TUs.

However, it should be stressed that annotations are associated with the implementation of a module; if the implementation changes, the annotations might change too, even if the interface remains the same. With inter-TU optimisations enabled, modules depending on an interface whose implementation has changed will need to be recompiled. This can be handled automatically by program building systems (such as Unix `make`, or more sophisticated optimisation-aware schemes [CKT86, Hal91]), but it does imply a certain amount of extra work for the computer; optimisation always has its price.

### 11.3.2 Format of Annotation Files

Unlike the centralised databases of other inter-TU optimisers, annotation files consist of text. They are meant to be generated and read automatically, so their syntax is not user-friendly, but it is still possible to write annotation files by hand. This solves the problem of the provision of optimisation information for modules whose source code is not available. It also allows annotation files to be hand-tuned in cases where the compiler has not been very successful at deducing the module’s properties.

To keep the compiler’s size down, annotations employ the same lexical tokens as P programs, and can be read using the same lexical analyser; as the lexical analyser is already restartable to cope with reading imported interfaces, no extra work needs to be done on it. We do need a new parser, but since user-friendliness is not an issue, we can use a format that can be read and written with the minimum amount of effort. The syntax is thus dictated by the format of the datastructures to be represented. We will see examples later on.

### 11.3.3 Contents of Annotation Files: Exported Variables

The contents of annotation files fall in two broad categories: initial values of exported variables, and information on the behaviour of procedures whether they are exported or not.

The initial values of variables are constants determined by the translator. Every such constant can be represented as an element of the abstract values lattice, and it is textual representations of these values that are placed in the annotation files. This information is very important to the optimiser, since procedures are exported by having their addresses placed in variables; determining the values of these variables (and the fact that they are constant) is the only way to deduce which extra-modular procedures are referenced at call sites.

### 11.3.4 Contents of Annotation Files: Procedures

To determine what happens once such procedures are called, we also place reports of their behaviour among the annotations. It is not sufficient to report on procedures assigned to exported variables, as these may in turn call procedures not visible outside the module, or even place addresses of such in globally accessible storage. In its greedy search for information, the annotation-aware optimiser is thus not constrained by module boundaries at all, although it maintains this illusion for the sake of the programmer’s sanity.

A procedure annotation contains the following:

- The number of arguments it is declared to take.
• The range of its return value.
• The memory locations it may reference.
• The memory locations it may define, with their final values.
• A list of other procedures it may call, with elements of the context at the call site that may be useful to the callee.

Of these, the first item is trivially obtainable from the translator; the others are obtained from the abstract context at the procedure's return (.return/retn), dereferencing (!/load, stor) and call (@/call) expressions/insns. Thus, our annotations are richer than the procedure summary information used by some other systems.

One issue that should be discussed here concerns referential transparency, or rather the lack of it. The results of a referentially transparent piece of code are invariant with respect to the evaluation order of its subcomponents; imperative languages, including P and W1M3, do not have this property. Our annotations summarise the effects of procedures, and can inherit their referential opacity; we must take care to apply the components of annotations in the right order, or make sure that they are generated in such a way that the order does not matter. It turns out that the latter way is easier, although, following a pattern that should be familiar by now, it does incur a cost in terms of accuracy of the analysis.

11.3.5 Conditional Annotations

As discussed so far, annotations are unconditional: they describe effects on the abstract context without reference to its original contents. This is unfortunate because many of the optimisations we would like to perform based on inter-TU information do depend on the context before a call; a simple example would be the elision of argument checking code on entry to a procedure.

The existence of associations between the value ranges in an abstract context facilitates the introduction of such conditional annotations, but does not render it entirely trivial. The problem is that, as noted above, the information we need comes from contexts deep in the called procedure, at which point the original contents of vregs and memory may be altered beyond recognition. As we do not allow assocs between different contexts, we cannot link, for example, argument and return values.

The solution is to keep, in every context within the procedure, “shadow” copies of the elements of the original context that we are interested in. Assocs to these copies are maintained as usual, and used to produce conditional annotations wherever necessary. Of course, we must exercise discretion in selecting the parts of the original context to keep, or we run the risk of doubling the size of our contexts; in the current implementation, we only keep the values of procedure arguments. This has proved to be sufficient for most purposes, but more sophisticated schemes that use heuristics to select global variables and memory blocks to be kept can be devised if necessary.

For the moment, the argument copies can be generated in the abstract interpretation of the entr (procedure entry) insn. To hold them, we introduce a third component in every abstract context, so that

$$\text{Contexts}_{abs} \subseteq \text{VEnv}_{abs} \times \text{MEnv}_{abs} \times \text{SEnv}_{abs}$$

where

$$\text{SEnv}_{abs} = \mathbb{N} \rightarrow \text{Values}_{abs}$$
and a shadow environment $A_s \in \text{SEnv}_{\text{abs}}$ is such that $A_s(n)$ is the range of possible initial abstract values of the $n$th argument to the procedure concerned, assuming that the flow of control has reached the point where the corresponding context represents the code.

The above may be compared with the old definition of $\text{Contexts}_{\text{abs}}$ in equation 8.1. We will not present updated definitions of every context-handling operation. The only changes necessary are to assoc updates, context splits and restrictions; these can be done in ways similar to the handling of the $A_v$ component. Given a context $C_{\text{abs}} = (A_v, A_m, A_s)$, we will extract the shadow component by

$$\text{PROJS}(C_{\text{abs}}) = A_s$$

We will also extend $\text{COMPLD}$ to take numbers as its first argument, so we can generate identifiers for the components of the shadow environment.

### 11.3.6 Interpreting the $\text{ent} \text{r} \text{insn}$

More precisely, assume that $\text{PcsArg}(n)$ is the hreg holding the $n$th argument on entry to a procedure, according to the machine’s calling standard; also assume the existence of

$$\begin{align*}
\text{INIT}_{\text{reg}} \in \text{Procs} & \rightarrow \text{VEnv}_{\text{abs}} \\
\text{INIT}_{\text{mem}} \in \text{Procs} & \rightarrow \text{MEnv}_{\text{abs}}
\end{align*}$$

such that $\text{INIT}_{\text{reg}}(p)(v)$ is the initial abstract value of reg $v$, and $\text{INIT}_{\text{mem}}(p)(b)$ the initial abstract contents of memory block $b$, on entry to procedure $p$.

If $i$ is the first insn in $p$, i.e. $\text{OP}[i] = \text{ent} \text{r}$, and $a \in \text{Arcs}$ the arc leading away from it\footnote{Which can trivially be made unique by adding a $\text{nop} \text{ insn}$ as the successor of $i$.}, then we define the insn’s interpretation as

$$\text{NEXT}_{\text{abs}}(C_{\text{abs}})(a) = (A_v, A_m, A_s)$$

where the components of the new context are given by

$$\begin{align*}
A_v &= \lambda v . \left\{ \langle \text{id}, \text{val}, \text{asc} \rangle \mid \text{id} = \text{COMPLD}(v, \langle \text{id}_0, \text{val}, [] \rangle, \text{nilcomp}) \land \\
& \quad \text{val} \in \text{PROJvals}(\text{INIT}_{\text{reg}}(p)(v)) \land \\
& \quad \text{asc} = \langle \langle \text{sid}, \text{exact} \rangle \rangle \land \\
& \quad \langle \text{sid}, \text{val}, \langle \langle \text{id}, \text{exact} \rangle \rangle \rangle \in A_s(n) \} \text{ if } \exists n \in \mathbb{N} : v = \text{PcsArg}(n) \\
& \quad \text{otherwise}
\end{align*}$$

$$\begin{align*}
A_s &= \lambda n . \left\{ \langle \text{sid}, \text{val}, \langle \langle \text{id}, \text{exact} \rangle \rangle \rangle \mid \text{sid} = \text{COMPLD}(n, \langle \text{id}_0, \text{val}, [] \rangle, \text{nilcomp}) \land \\
& \quad \text{val} = \text{PROJvals}(\text{INIT}_{\text{reg}}(p)(v)) \} \in A_v(\text{PcsArg}(n))
\end{align*}$$

$$A_m = \lambda b . \text{INIT}_{\text{mem}}(p)(b)$$

Definitions for $\text{INIT}_{\text{reg}}$ and $\text{INIT}_{\text{mem}}$ will be given later; see section 11.6.5.

### 11.4 Building Annotation Files

#### 11.4.1 Representing Annotations

We have mentioned that a procedure’s annotations are functions of the arguments to the particular call that interests us, and that is exactly how we will represent them. In most cases, their domain will be

$$\text{AnnoDom} = \langle \text{Arcs} \rightarrow \text{Contexts}_{\text{abs}} \rangle \times \text{Procs} \times (\mathbb{N} \rightarrow \text{Values}_{\text{abs}})$$
representing a context vector solution to the abstract interpretation, a procedure to be annotated, and a mapping from argument numbers to corresponding abstract values. The functions themselves are

\[
\begin{align*}
\text{ANNOargs} & 
\in \text{Procs} \rightarrow \mathbb{N} \\
\text{ANNOretn} & 
\in \text{AnnoDom} \rightarrow \text{Values}_{abs} \\
\text{ANNOrefs} & 
\in \text{AnnoDom} \rightarrow \text{Values}_{abs} \\
\text{ANNOdefs} & 
\in \text{AnnoDom} \rightarrow \mathbb{P}[\text{Values}_{abs} \times \text{Values}_{abs}] \\
\text{ANNOcall} & 
\in \text{AnnoDom} \rightarrow \mathbb{P}[\text{Procs} \times \text{Contexts}_{abs}]
\end{align*}
\]

\(\text{ANNOargs}(p)\) returns the number of arguments expected by a procedure \(p \in \text{Procs}\). The remaining functions return

- The possible return values, as a scalar value range (\(\text{ANNOretn}\)).
- The memory locations possibly referenced, as a scalar range (\(\text{ANNOrefs}\)).
- The memory locations possibly defined, and their contents on exit from the procedure, as a collection of pairs of value ranges representing addresses and corresponding values (\(\text{ANNOdefs}\)).
- The procedures possibly called, and the environments supplied to them, as a collection of tuples (\(\text{ANNOcall}\)).

### 11.4.2 Preliminaries

#### 11.4.2.1 Restricting Contexts for Given Arguments

In generating annotations, we will often need to restrict a context with respect to given values for the procedure’s initial arguments. If \(p \in \text{Procs}\) is a procedure, \(C_{abs}\) is a context, and \(\text{args}\) a function returning the “interesting” initial argument values for \(p\) given the argument number, we recall the definitions of \(\text{RESTART}\) and \(\text{SPLITALL}\) from section 8.4.6 and define

\[
\text{RESTART}(C_{abs}, p, \text{args}) = \text{RESTART}(\text{RESTART}(\ldots (C'_{abs},
\text{PROJS}(C'_{abs}))(1), \text{PROJS}(C'_{abs}))(2), \text{args}(2))\ldots),
\text{PROJS}(C'_{abs})[\text{ANNOargs}(p)], \text{args}[\text{ANNOargs}(p)])
\]

where

\[
\begin{align*}
C'_{abs} & = \text{SPLITALL}(\text{SPLITALL}(\ldots (C_{abs},
\text{PROJS}(C_{abs}))(1), \text{PROJS}(C_{abs}))(2), \text{args}(2))\ldots),
\text{PROJS}(C_{abs})[\text{ANNOargs}(p)], \text{args}[\text{ANNOargs}(p)])
\end{align*}
\]

#### 11.4.2.2 Collecting Insns by Opcode

To pick out arcs leading to insns with a given opcode, we define

\[
\text{INSNARCHS}(p, \text{insn}) = \{(i_p, i_c) \mid \text{OP}_{i_c}(i_c) = \text{insn} \land \{i_p, i_c\} \in \text{INSNS}(p)\}
\]

#### 11.4.2.3 Collecting Procedure Arguments

Suppose \(C_{abs}\) is the context just before a call insn. The arguments to the called procedure can be collected by

\[
\text{CALLARGS}(C_{abs}) = \lambda n \rightarrow \text{PROJS}(C_{abs})(\text{PCSARG}(n))
\]
11.4.3 Argument Count

The number of arguments expected by a procedure is encoded in its \texttt{entr} insn as the last WIM3 argument. Thus
\[
\text{ANNOn}_{\text{arg}}(p) = \text{ARG}_i(\text{entr}, 3)
\]

where
\[
(\text{entr}) = \text{INSNARCS}(p, \text{entr})
\]
and that’s all there is to it.

11.4.4 Return Values

The return value of a procedure \( p \in \text{Procs} \) is the value assigned to the return value vr egr, \( \text{PcsRv} \in \text{Vregs} \), when a \texttt{retn} insn is reached. Thus we must read the value ranges of \( \text{PcsRv} \) from the abstract contexts at all such insns, restrict each of them with respect to the corresponding original argument values, and return the union of the results:
\[
\text{ANNOn}_{\text{ret}}(\text{abs}, p, \text{args}) = \text{NORMALISE}\left( \bigcup_{a \in \text{INSNARCS}(p, \text{ret})} \text{PROJ}(\text{RESTARGS}(\text{abs}(a), p, \text{args}))(\text{PcsRv}) \right).
\]

11.4.5 Procedure Calls

Procedures may only be called using \texttt{call} insns, so to collect the possible destinations, we need to examine the value of the first argument of each such insn. If its value range contains non-procedure components, we have to assume that any procedure may be called. We also need to save the procedure’s arguments in the shadow component of a context.
\[
\text{PROCalls}(\text{abs}, p, \text{args}) = \bigcup_{c \in \text{AnnV}_{\text{abs}}(\text{abs}, \text{call})} \bigcup_{(i_p, i_c) \in \text{INSNARCS}(p, \text{call})} \left\{ \begin{array}{ll}
\{ \langle \text{PROJ}(c), \text{abs} \rangle \} & \text{if } \text{PROJ}(c) \in \text{Procs} \\
\{ \langle p', \text{abs} \rangle \} & \text{otherwise}
\end{array} \right.
\]

where
\[
\text{abs} = \text{RESTARGS}(\text{abs}(\langle i_p, i_c \rangle), p, \text{args})
\]
and \( \text{abs}' = \langle \text{abs}', \text{abs}', \text{abs}' \rangle \) is given by
\[
\text{abs}' = \lambda v. \{ \begin{array}{ll}
\text{abs}(v) & \text{if } \exists n. v = \text{PcsArg}(n) \\
\text{T}_{\text{abv}} & \text{otherwise}
\end{array}
\]
\[
\text{abs}' = \lambda b. \{(\rightarrow \ldots + \infty), \text{T}_{\text{abv}}\}
\]
\[
\text{abs}' = \text{CALLARGS}(\text{abs})
\]
so that the only practical information in the context is the argument values. We will later have an opportunity to define more accurate versions of \( \text{abs}' \) and \( \text{abs}' \); see section 11.5.2.

Like other annotations, the results of \text{ANNOn}_{\text{call}} contain the effects of nested calls, and are defined as follows:
\[
\text{ANNOn}_{\text{call}}(\text{abs}, p, \text{args}) = \text{PROCalls}(\text{abs}, p, \text{args}) \cup \\
\bigcup_{(p', (\text{abs}', \text{abs}', \text{args}')) \in \text{PROCalls}(\text{abs}, p, \text{args})} \text{PROCalls}(\text{abs}, p, \text{args})
\]
11.4.6 Memory References

There are three kinds of references we would like to report:

- References to elements of memory blocks. This includes vectors, statics, and heap blocks.
- References to imported and locally declared global variables.
- References to other variables whose addresses have been passed to the current TU.

Note that all of the above are resident in memory at least part of the time, so they all have addresses by which they can be identified. Thus, the entire set of references made in the procedure can be represented by a single value range consisting of block and variable address components. To this we should add any references made by procedures called by the one we are examining.

“Visible” (i.e., imported or locally declared) global variables are referenced at any insn referencing their vregs. The remaining references are collected from the target addresses of load insns, and the reference annotations of any procedures called:

\[
\text{ANNOTREFS}(\text{Cabs}, p, \text{args}) = \\
\left\{ (\text{id}_0, \text{v}, []) \mid \exists i \in \text{INSNS}(p), v \in \text{DEFV}(i) \cap \text{Globals} \right\} \cup \\
\bigcup_{(i_p, i_v) \in \text{INSNARCS}(p, \text{LOAD})} [\text{ARGVabs}(\text{Cabs}, i_v, 2) +_{\text{ABV}} \text{ARGVabs}(\text{Cabs}, i_v, 3)] \cup \\
\bigcup_{(p', (A'_i, A''_i, \text{args}')) \in \text{ANNOTDEF}(\text{Cabs}, p, \text{args})} \text{ANNOTDEF}(\text{Cabs}, p', \text{args}')
\]

where, as before,

\[
\text{Cabs} = \text{RESTARGS}(\text{Cabs}(\langle i_p, i_v \rangle), p, \text{args})
\]

11.4.7 Writing to Memory

The order in which memory references are reported does not matter, since the values referenced are not collected. When dealing with assignments to memory, it would be nice if we could report the new value, but this value may be affected by assignments to overlapping memory address ranges later on in the procedure.

The exact flow of control, and therefore the order of assignments, cannot be determined at compile time. Fortunately, this is not a problem for us: we are only interested in the final range of values in memory locations that may have been affected by assignments. All we need to do is collect the relevant addresses just as we did for memory references, then read their values in all contexts corresponding to return instructions, like we did for the procedure’s return values. The former is achieved by

\[
\text{MEMUPDATES}(\text{Cabs}, p, \text{args}) = \\
\left\{ (\text{id}_0, \text{v}, []) \mid \exists i \in \text{INSNS}(p), v \in \text{DEFV}(i) \cap \text{Globals} \right\} \cup \\
\bigcup_{(i_p, i_v) \in \text{INSNARCS}(p, \text{STORE})} [\text{ARGVabs}(\text{Cabs}, i_v, 2) +_{\text{ABV}} \text{ARGVabs}(\text{Cabs}, i_v, 3)] \cup \\
\{ \text{addr} \mid \langle \text{addr}, \text{cont} \rangle \in \\
\bigcup_{(p', (A'_i, A''_i, \text{args}')) \in \text{ANNOTDEF}(\text{Cabs}, p, \text{args})} \text{ANNOTDEF}(\text{Cabs}, p', \text{args}')
\]
where, again,

\[ C_{\text{abs}} = \text{RESTARGS}(C_{\text{abs}}(i_p, i_c), p, \text{args}) \]

and the latter is given by

\[
\begin{align*}
\text{ANNOdefs}(C_{\text{abs}}, p, \text{args}) &= \{ (\text{addr}, \text{cont}) | \\
& \quad \{ (i_d, \text{addr}, \text{asc}) \in \text{MEMUPDATES}(C_{\text{abs}}, p, \text{args}) \land \text{addr} \in \text{Vregs} \land \\
& \quad \text{cont} = \bigcup_{a \in \text{INSNargs}(p, \text{retn})} \text{PROJ}(\text{RESTARGS}(C_{\text{abs}}(a), p, \text{args})(\text{addr})) \lor \\
& \quad \{ (i_d, (b, \text{off}), \text{asc}) \in \text{MEMUPDATES}(C_{\text{abs}}, p, \text{args}) \land c \in \text{off} \land \\
& \quad \text{addr} = (i_d0, (b, [c]), []) \land \\
& \quad \text{cont} = \bigcup_{a \in \text{INSNargs}(p, \text{retn})} \text{CONTENTS}(\text{PROJ}(\text{RESTARGS}(C_{\text{abs}}(a), p, \text{args})([b]), \text{PROJval}(c)))) \}
\end{align*}
\]

We thus ensure referential transparency, as discussed in section 11.3.4.

### 11.4.8 Generating Annotation Files

It is, of course, impossible to place a full description of an annotation function in the corresponding file, if only because the procedure arguments for which results will be required cannot be predicted at this stage. Instead, we have to select some appropriate argument ranges and give the annotation results for them, simultaneously ensuring that any other argument ranges are also covered, perhaps by less accurate versions of the annotations (e.g. assuming all arguments have the value \( T_{\text{abs}} \)).

The compiler achieves this by examining the contexts at insns where annotations are gathered—\textit{call} for \textit{ANNOcall}, \textit{load} for \textit{ANNOrefs}, etc—and generating the results for argument values present in the shadow components of such contexts. We will see some examples of this later on.

### 11.5 Refining Annotations

#### 11.5.1 Introduction

As presented above, annotations will not be of much use in optimisation: analysing a procedure with no information about the context in which it is called is likely to result in spectacularly unhelpful annotations.

What we need is more information about the abstract contexts on entry to procedures, and we will use two methods to obtain it: pre-existing annotations, and a very weak form of type inference.

#### 11.5.2 Responding to Reference Annotations

Referencing a memory block does not affect the abstract context, so the reader would be justified in wondering why we collect such information.

The answer is that a subsequent compilation pass through a TU that calls the procedure in question can examine these \textit{ANNOrefs} annotations and provide the contents of the referenced cells along with its \textit{ANNOcall} annotations. If we then recompile the callee and indicate which modules are allowed to call it, we are provided with initial values for global vregs and memory blocks in the procedure. The same should be done for the \textit{ANNOdefs} annotations, since non-deterministic stores also reference their destinations in producing their new values (see section 9.5.2).
We will discuss the creation of initial contexts later (section 11.6.5), but for the moment, let us give a more accurate definition of $\text{ANNOCall}$.

Recall the safe definitions of $\Lambda'_v$ and $\Lambda'_m$ from section 11.4.5. If annotations exist for all procedures $\text{PROjVal}(c)$ called by our procedure $p$, we collect the addresses referenced or defined in them as

$$\text{addrs} = \text{ANNOdefs}(C_{abs}, \text{PROjVal}(c), \text{CALLARGS}(C_{abs})) \cup \{c \mid c \in T \land (t, \text{cont}) \in \text{ANNODEfs}(C_{abs}, \text{PROjVal}(c), \text{CALLARGS}(C_{abs}))\}$$

and redefine

$$\Lambda'_v = \lambda v \cdot \begin{cases} \Lambda_v(v) & \text{if } v \in \text{PROjVal}(\text{addrs}) \\ \Lambda_v(v) & \text{if } \exists n : n = \text{PCSArg}(n) \\ T_{\text{abv}} & \text{otherwise} \end{cases}$$

$$\Lambda'_m = \lambda b \cdot \begin{cases} \{\langle o, \text{val} \rangle \mid o \in \text{PROjVal}(\text{PROjOff}(d)) \land \text{val} = \text{NORMALISE}([\text{CONTENTS}(A_m(b), o), \{b, o\}, \text{nilcomp}]) \}
& \text{if } d \in \text{addrs} \land \text{PROjBas}(d) = b \\ \{[-\infty \ldots +\infty], T_{\text{abv}}\} & \text{otherwise} \end{cases}$$

### 11.5.3 Kind Inference

#### 11.5.3.1 Introduction

Kind inference is a form of type inference weakened almost to the point of atrophy. It makes no deductions about the underlying concrete values of the abstract values it manipulates, and is prepared to reach different decisions about the same value on different paths of execution, or to reach no conclusion at all. Its relationship to ML-style "strong" type inference is similar to that of our advisory type checking (see section 10.2.7) to ML-style type checking.

Kind inference depends on two facts:

- Abstract value kinds overlap, in that a concrete value may be abstracted into more than one value kind. For example,
  $$\gamma_{\text{abv}}(T_{\text{abv}}) = \bigcup_{r \in \text{Intervals}} \gamma_{\text{abv}}(r) = \mathbb{Z}$$
  i.e. restricting an unknown value to the integer interval kind does not actually restrict the bitpatterns it can abstract.

- Some operations, such as calling an integer or adding two procedures together, are rare and are treated conservatively in the abstract domain.

The idea is to restrict (some of) the unknown values in a procedure's initial context, in such a way that no underlying concrete values cease to be represented, except possibly ones leading to undefined results.

Thus, for example, we will take the addition of a number and an unknown value as an indication that the unknown is itself a number or a memory address, since adding a number to procedure, variable or label addresses leads to unknown results. Our assumption will allow us to restrict the value if the opportunity arises later on in the program; but if, due to programmer perversity, it turns out that we have made the wrong guess, the conservative nature of our interpretation combined with the fact that no bitpatterns have been excluded ensures that the results will be safe.

If we are feeling adventurous, or vindictive towards programmers that stretch the limits of the semantics, we may make even stronger assumptions:
we may assume that calls/jumps to unknown values mean the values are procedure/label addresses, for example. Such assumptions remove some of the possible concretisations from the abstract values (since not all bitpatterns are valid code addresses), and may produce incorrect code for ill-behaved programs; they should not be enabled by default.

Unlike strong type inference, kind inference does not use unification; it does not propagate upwards in the flowgraph, so the same unknown value may be treated differently on different branches. The abstract context join operation (see section 8.4.3) already handles the eventuality of the branches rejoining.

11.5.3.2 Details

We define two functions, \( \text{INFER}_2 \) and \( \text{INFER}_3 \), that take the opcode of an insn and the value ranges of the two operands it references; they return the adjusted value of one of the operands. The former handles the second WIM3 operand, the latter the third.

For symmetric operations, like add, both functions are defined in the same way. If \( r \) is the operand to adjust and \( \text{arg} \) the other operand, we have

\[
\text{INFER}_2(\text{add}, r, \text{arg}) = \text{INFER}_3(\text{add}, r, \text{arg}) = \text{INFER}_3(\text{sub}, r, \text{arg}) =
\]

\[
\begin{cases}
\text{r} & \text{if } r \neq \text{T} \text{abv} \\
\text{int} & \text{if } \text{arg} = \text{PART}\text{add} \text{d}(\text{arg}) \\
\text{int} \cup \text{addr} & \text{if } \text{PART}\text{misc}(\text{arg}) = \emptyset \\
\text{T} \text{abv} & \text{otherwise}
\end{cases}
\]

where

\[
\text{int} = \{ \langle \text{id}_0, \langle -\infty \ldots +\infty \rangle, \emptyset \rangle \}
\]

\[
\text{addr} = \{ \langle \text{id}_0, \langle \text{bas}, \langle -\infty \ldots +\infty \rangle, \emptyset \rangle \rangle \text{bas} \in \text{Memobjs} \}
\]

are upper bounds of the integer intervals and block addresses in the abstract values lattice. Thus values added to addresses are assumed to be integers, while if added to integers, they are assumed to be integers or addresses.

Inference of the value kind of a minuend can be more accurate, although addresses cannot be validly subtracted from anything but other addresses:

\[
\text{INFER}_2(\text{sub}, r, \text{arg}) =
\]

\[
\begin{cases}
\text{r} & \text{if } r \neq \text{T} \text{abv} \\
\text{num} & \text{if } \text{arg} = \text{PART}\text{int} \text{d}(\text{arg}) \\
\text{addr} & \text{if } \text{arg} = \text{PART}\text{add} \text{d}(\text{arg}) \\
\text{num} \cup \text{addr} & \text{if } \text{PART}\text{misc}(\text{arg}) = \emptyset \\
\text{T} \text{abv} & \text{otherwise}
\end{cases}
\]

Similar definitions apply for other insn types.

To implement kind inference, then, we define an operator \( \text{INFER}_{\text{abs}} \) that uses the above to modify the context before insns that take vregs as their operands.

This operator will be applied to the abstract context before the interpretation of every insn in the program. For a context \( C_{\text{abs}} = \langle A_v, A_m, A_c \rangle \) just before insn \( i \), we update the vreg environment as follows:

\[
\text{PROJ}_v(\text{INFER}_{\text{abs}}(C_{\text{abs}}, i)) = \lambda v . \text{NORMALISE}(r, v, \text{nilcomp})
\]

where

\[
\begin{align*}
\text{INFER}_2(\text{OP}_i[i], \text{ARG}_v \text{abs}(C_{\text{abs}}, i, 2), \text{ARG}_v \text{abs}(C_{\text{abs}}, i, 3)) & =
\end{align*}
\]

\[
\begin{cases}
\text{r} & \text{if } \text{ARG}_{T}[i, 2] = \text{reg} \wedge \text{ARG}_{T}[i, 2] = v \\
\text{INFER}_3(\text{OP}_i[i], \text{ARG}_v \text{abs}(C_{\text{abs}}, i, 3), \text{ARG}_v \text{abs}(C_{\text{abs}}, i, 2)) & =
\end{cases}
\]

\[
\begin{cases}
\text{r} & \text{if } \text{ARG}_{T}[i, 3] = \text{reg} \wedge \text{ARG}_{T}[i, 3] = v \\
A_v(v) & \text{otherwise}
\end{cases}
\]
The memory environment $A_m$ is left unchanged, and the entries in $A_s$ with 
asssocs to $A_v$ are modified accordingly.

11.6 Using Annotations

11.6.1 Introduction

The uses of annotations fall into two broad categories:

**Caller-side** uses, such as the provision of memory contents described in section 11.5.2, and updates to the context after a procedure call according to the callee’s side effects.

**Callee-side** uses, i.e. enrichment of the callee’s initial context with information from its potential callers.

We have already seen hints of the latter; the following sections will present the complete picture, after completing the discussion of the former (and our abstract WIM3 semantics) with the detection of recursive procedures and the annotation-aware interpretation of the call insn.

11.6.2 Detecting Recursive Procedures

A procedure is recursive if there can be more than one instance of its stack frame on the stack during execution of the program. This means that any procedure that calls itself, or that calls other procedures which might call it back, is recursive. Because of this, traditional compilers have to assume that calls to procedures outside the current TU make the caller recursive.

With our annotated TUs, we can do better. A procedure is recursive if it, or $T_{abv}$, is contained in its own call annotation; the function $\text{RECURSIVE}$ used in sections 9.5 and 10.2.6 can be defined thus:

$$\text{RECURSIVE}(C_{abs}, p) \iff \exists C_{abs} \in \text{Contexts}_{abs} . \langle p, C_{abs} \rangle \in \text{ANNO}_\text{call}(C_{abs}, p, \lambda n. T_{abv})$$

Of course, we can determine the recursivity of a procedure given certain arguments by supplying those arguments to the $\text{ANNO}_\text{call}$ above.

11.6.3 Elimination of Precautionary Stores and Spills

In section 10.3.2, we discussed how to minimise the number of vregs spilt before a memory access based on value range information. We can now do the same for calls. If $i_c$ is a call insn, we replace its set $\text{SAVE}_{\text{st-cl}}$ of vregs spilt before the call (see equation 5.7 in section 5.6.3) with the set of vregs mentioned in definition annotations for procedures possibly called by $i_c$:

$$\text{SPI}LL_{\text{anno}}(C_{abs}, (i_p, i_c), \text{args}) =$$

$$\begin{cases} \text{Globals} \cup \text{Addrvars} \quad &\text{if } \text{PROJ}_{\text{vals}}[\text{ARGV}_{\text{abs}}|C_{\text{abs}}, i_c, 1]| \not\subseteq \text{Procs} \not= \emptyset \\
\{ v \mid v \in \text{Vregs} \cap \{ v | (v, \text{cont}) \in \left( \bigcup \text{ANNO}_{\text{defs}}(C_{abs}, p, \text{CALLARGS}(C_{abs})) \right) \text{ otherwise} \}
\end{cases}$$

We also modify the definition of $\text{SAVE}_{\text{load}}$, the set of vregs saved before a load insn but not marked as spilt (equation 5.6), to accept call insns as well, and
return the vregs mentioned in reference annotations for the callees:

\[
\text{SAVEanno}(C_{\text{abs}}, \langle i_p, i_c, \text{args} \rangle) = \\
\begin{cases} 
\text{Globals} \cup \text{Addrvars} & \text{if } \text{PROJvals}(\text{ARGV}_{\text{abs}}(C_{\text{abs}}, i_c, 1)) \setminus \text{Procs} \neq \emptyset \\
\bigcup_{p \in \text{PROCabs}(\text{ARGV}_{\text{abs}}(C_{\text{abs}}, i_c, 1))} \text{ANNOrefs}(C_{\text{abs}}, p, \text{CALLARGS}(C_{\text{abs}})) & \text{otherwise}
\end{cases}
\]

In both the above,

\[
C_{\text{abs}} = \text{RESTARGS}(C_{\text{abs}}, \langle i_p, i_c \rangle, p, \text{args})
\]

where \( p \) is the procedure containing \( i_c \) and \( \text{args} \) the arguments it takes.

### 11.6.4 Interpreting the call insn

Up to this point, we have treated procedure calls as dives into “black boxes” that can affect every aspect of the program’s state they can reach. Procedures for which no annotations are available will still be treated this way, but for the rest we can do better:

- Rather than setting the contents of all heap blocks to unknown values, we can modify just those indicated by the \( \text{ANNOdefs} \) function for the procedure in question.
- The same holds for global variables, and non-globals whose addresses have been taken.
- We have mentioned (section 8.2.9) that procedures may return “fresh” heap blocks, which are adopted by the calling insn. This must be done now.
- Finally, the procedure’s return value as indicated by the \( \text{ANNOrets} \) function can be assigned to the appropriate vreg.

First let us devise a way to turn “fresh” blocks into blocks seemingly allocated by an insn \( i \):

\[
\text{DEFRESH}(i, \tau) = \text{PARTint}(\tau) \cup \text{PARTmisc}(\tau) \cup \\
\{ (\langle \text{id}, \langle b', \text{off} \rangle, \text{asc} \rangle) | (\langle \text{id}, \langle b, \text{off} \rangle, \text{asc} \rangle) \in \tau \land \\
([b = \text{Fresh} \land b' = i] \lor [b \neq \text{Fresh} \land b' = b])\}
\]

Now, assume \( a_p = \langle i_p, i_c \rangle, a_n \) are the arcs leading to and from a call insn \( i_c \) respectively. If

\[
C_{\text{abs}}(a_p) = C_{\text{abs}} = \langle A_v, A_m, A_s \rangle
\]

is the context before the call, the return value will be

\[
\text{retval} = \text{DEFRESH}(i, \bigcup_{c \in \text{ARGV}_{\text{abs}}(C_{\text{abs}}, i_c, 1)} \text{ANNOrets}(C_{\text{abs}}, \text{PROJval}(c), \text{CALLARGS}(C_{\text{abs}})))
\]

and the set of addresses to update will be

\[
\text{update} = \bigcup_{c \in \text{ARGV}_{\text{abs}}(C_{\text{abs}}, i_c, 1)} \text{ANNOdefs}(C_{\text{abs}}, \text{PROJval}(c), \text{CALLARGS}(C_{\text{abs}}))
\]
We can now define the interpretation of the insn as

$$\text{Next}_\text{abs} (C_{\text{abs}})(a) = (A'_{v}, A'_{m}, A_v)$$

where the modified environments are given by the following, in which we recall the definition of StoreMem from section 9.5.2:

$$A'_{v} = \lambda v . \begin{cases} \text{Normalise}(v, \text{v}, \text{nilcomp}) & \text{if } v = \text{PcsRv} \\ \text{Normalise}(\text{cont}, v, \text{nilcomp}) & \text{if } (\text{addr}, \text{cont}) \in \text{update} \land v \in \text{Proj}_\text{val}(\text{addr}) \\ A_v(v) & \text{otherwise} \end{cases}$$

$$A'_{m} = \text{StoreMem}(\text{StoreMem}([\cdots | C_{\text{abs}}, a_1, a_1, a_2, c_1, a_1, a_2, a_2 | \cdots], a_1, c_1, a_1)$$

where

$$\text{Part}_{\text{addr}}(\text{update}) = \langle c_1, a_1 \rangle, \langle c_2, a_2 \rangle, \ldots, \langle c_n, a_n \rangle$$

Note that we replace the old values with the new ones rather than merging them, since the definition of anno.dets ensures that the latter contain the former, wherever necessary. Also note that fresh block identifiers cannot arise through any mechanism in our analysis; library writers will have to manually modify the annotations for memory allocation routines.

### 11.6.5 Specialising Modules

Given annotations about how a procedure is called, we can construct an accurate initial context for it. Besides improving the results of value range analysis, this will hopefully also allow us to specialise the procedure by throwing away dead code.

Essentially, we are getting a form of partial evaluation for free, since our optimiser already has all the facilities required (see section 10.2.4). Traditionally, partial evaluation has been a source-level, interactive process based on constant values in a procedure's initial context; our version is non-interactive, acts on the intermediate code, and tries to do the best it can based on partial information about values in the context.

The initial context for a procedure $p$ is specified by $\text{Init}_\text{reg}(p)$ and $\text{Init}_\text{mem}(p)$ (see section 11.3.6). For an unspecialised procedure,

$$\text{Init}_\text{reg}(p)(v) = \text{Contents}(\text{Init}_\text{mem}(p)(b), \text{int}) = T_{abv}$$

for any vreg $v$, non-static memory block base $b$ and interval $\text{int}$; we have no information on the initial context and rely on kind inference to deduce the value kinds of objects referenced in the procedure. The only exceptions to this concern:

1. Memory blocks $b \in \text{Statics}$, which are immutable; these are set to their constant values.
2. Vregs $v \in \text{Globals}$ on entry to the procedure $\text{main.main}$, the program entrypoint; these are set to the initial values specified in the corresponding annotation files.
3. Heap memory blocks allocated by library procedures that initialise their contents; these are set to zero on entry to $\text{main.main}$, even though they are not yet allocated at that point. They can then be inherited by called procedures via anno.call annotations.

---

3This is not a problem; the blocks cannot be accessed until they are actually allocated.
11.7. IMPLEMENTATION DETAILS

Given a set \( \text{calls} \) of \( \text{ANOCall} \) annotations for procedures allowed to call \( p \), we can define

\[
\begin{align*}
\text{INIT}_{\text{reg}}(p) &= \text{PROJ} \left( \bigcup_{(p, C_{\text{abs}}) \in \text{calls}} C_{\text{abs}} \right) \\
\text{INIT}_{\text{mem}}(p) &= \text{PROJ} \left( \bigcup_{(p, C_{\text{abs}}) \in \text{calls}} C_{\text{abs}} \right)
\end{align*}
\]

so that the initial context is obtained by joining all the possible initial contexts provided by the calls to \( p \). The set \( \text{calls} \) is specified on the compiler's command line as a list of modules; the \( \text{ANOCall} \) annotations of all procedures in these modules are placed in \( \text{calls} \). The names of the calling modules are also placed in the specialised module's annotation file, so that the compiler can issue warnings about specialisation mismatches. Abstract interpretation then proceeds as before, but the smaller initial context allows more precise optimisations.

11.7 Implementation Details

11.7.1 The Annotation Repository

To implement the inter-TU optimisation framework described in this chapter, the P compiler needs facilities to input, store, query, generate, and output annotations. These facilities are provided by the annotation repository.

The annotation repository is implemented as a pair of hash tables, each containing one entry per procedure; each entry holds the values of the various annotation functions for the procedure concerned. The same compiler run may both use and generate annotations for a procedure, so one hash table is used to hold the incoming annotations, while the other holds the outgoing ones. Information requests are directed to the former, and newly generated information is stored in the latter.

Table entries are collections of scalar and block value ranges, while the bucket-level structure of the table is the same as that of the hash tables we have used in register allocation and context representation. Code may be reused quite effectively to keep the repository overhead down.

11.7.2 Reading Annotation Files

Annotation files are read at translation time, along with the corresponding interface files. The process has been introduced in section 3.3.4.3, but we can now describe it in more detail.

The main complication is that we would like to read annotations for modules used indirectly, but not imported, by the current TU; variables and procedures from such modules will probably appear in the annotation files for directly imported modules, and we would like to know what their effects are since their addresses may be passed back via memory or return values. To read an annotation file, we have to read the corresponding interface too (so that the appropriate names and vreg identifiers are assigned in the compiler's internal datastructures), but we cannot read the interface before translating the current TU because that would make the names in it visible and confound the compiler's scope checking.
To get around this, the translator keeps lists of already imported interfaces and interfaces whose importation has been deferred; both these lists are initially empty. Whenever an import expression is reached in the first scan of the parse tree, the following algorithm is applied:

Algorithm 11.1 (Interface and Annotation Imports)

1. If the interface is on the "already imported" list, return.
2. If the interface is on the list of deferred imports, remove it from the list.
3. Locate and parse the interface file; if any errors occur, report them and return.
4. Locate the corresponding annotation file; if it does not exist, return.
5. Parse the annotation file, placing its contents in the "incoming" table of the repository.
6. The annotation file also contains a list of other interfaces imported by this interface's implementation; place these in the list of deferred imports.
7. Place the current interface in the "already imported" list.

Once the entire TU has been translated, the same algorithm is applied to any interfaces remaining in the deferred imports list.

The actual parsing is done by a predictive parser cooperating with the compiler's standard lexical analyser; the algorithms used are too commonplace to mention here.

11.7.3 Writing Annotation Files

Annotation files can be generated as soon as the information they contain is available, given that any possibility of errors in the code has been eliminated. This means we can produce them before the final, register-allocating pass of abstract interpretation, but in practice this step is also performed before annotation generation to allow for the future addition of register-related annotations.

The actual generation of the output is simple and will not be described here. Sample annotation files can be found in the examples of section 11.9.

11.8 Extending Annotation Files

The annotation mechanism in the P compiler is useful enough, but there are a number of ways in which it can be extended.

Register usage information is one obvious candidate, since, as we saw in section 11.2, it has been proposed before. We can use annotation files to optimise away the constraints imposed by the architecture's PCS. The simplest way to do this is to inform called procedures of the global variables to which their callers have assigned registers, thus saving needless reloads.

Given the flexible nature of annotation files, such schemes can be extended. Consider a sequence of nested procedures: rather than saving and restoring registers around every call, we can arrange things so that most of the procedures use non-overlapping register sets, with only a few of them saving many registers at the same time. On some architectures with good data prefetching
or specialised bulk memory access instructions (such as the ARM's `ldm/stm`),
this can save many clock cycles.

By placing value size information among the annotations, we could even attempt
inter-procedural register stuffing.

Annotations can also help with inlining. In conventional compilers, inter-
translation unit inlining is only possible if the inlined procedure is placed in
a header or interface file; an annotation-aware compiler could decide what
procedures are small enough to be inlined and place them in the annotation
file. Thus we can obtain all the benefits of inlining without sacrificing the
distinction between interfaces and implementations.

### 11.9 Examples

#### 11.9.1 Register Allocation

Rather than produce another code listing, we refer the reader back to the example of figure 5.5 on page 62; note that, after the call to `stdio.printf`
in insn 12 of block 5, the global variable holding that routine's address is
not considered to be split. This is because the compiler has read the `stdio`
annotations and found out that `printf` does not modify any variables. This
enables the optimisation discussed in section 11.6.3.

#### 11.9.2 Conditional Annotations

Recall the MATCH example of section 10.6.3 (code in figure 10.7 on page 171).
The annotation file produced after running it through the compiler can be
found in figure 11.1.

The file contains the initial value of the `test` variable, which is the address
of the corresponding procedure. It also states that this procedure takes 4
arguments (`arg 4`) and lists its possible return values. The notable points
are:
Figure 11.2: A program that reads characters from the terminal.

```plaintext
.import stdio;

.var main <- .defun main[]
{
    .var EOF <- -1; // End Of File marker
    .var str <- .vec 128 [0]; // a 128-word zero-initialised vector
    .var c, i; // two variables
    .for [i <- 0; i < 128; i++] // loop while there is room in the vector
    {
        c <- @stdio.getchar[]; // get a character from standard input
        .if [c == EOF] .endloop; // exit loop if end of file
        str!i <- c; // save character in vector
    }
}

.module main

- The annotation file is not smaller than the implementation. This is characteristic of annotations for contrived examples, which contain many return expressions and calls to different procedures little other processing. "Real" programs do not suffer from this, as they often contain multiple calls to the same procedure (all of which are collapsed into one annotation) and do non-trivial processing of their environment (of which only the final results may be visible, as reference/definition annotations).

- Each .result clause is essentially the value of the ANNOretn function for some combination of initial context values: it gives a return value range followed by a list of argument ranges. The argument values are derived from the contexts at retn insns and broken down according to their assocs, so we are guaranteed full coverage of the possible return values.

- The compiler has no information about the arguments, but as it climbs through the flowgraph it finds comparisons with numeric constants, and can apply kind inference to deduce that the first two arguments must be numbers too.

  The only exception is the first .result clause, resulting from the first conditional in the program: this only tests the second argument, and no assumptions can be made about the first one. The result is also unknown ('?'), being the sum of the last two (unknown) arguments.

- The compiler recognises that the second conditional, returning 3 if successful, has been subsumed into the first. It is eliminated altogether, and no annotation clause results.

- The third .result clause gives a return value between \(-\infty\) (.min) and 17, for first and second arguments less than 35 and greater than 19 respectively. As the result in this case is supposed to be half the first argument, and half of 34 is 17, this is correct. The lower limits of the argument and result values are the same, since half of \(-\infty\) is still \(-\infty\).

- The fourth clause, resulting from the fifth conditional, gives the result \([20\ldots36]\) even though the conditional only checks that 'b' is less than 37.
11.9. EXAMPLeS

Figure 11.3: Annotations for the stdio interface (top) and the program of figure 11.2 (bottom).

/* stdio.n: Pebble annotation file for standard input and output */

.fun stdio.getchar: .arg 0;
.result [-1..255] []
.var stdio.getchar: .fun stdio.getchar

/* END of stdio.n */

/* main.n: annotation file, automatically generated by
   pebble compiler (alpha->mips), version 0.99
   Time-stamp: <1998/05/10 20:39:11 qfp10@ouse.cl.cam.ac.uk>
*/

.var main.main: .fun main.main
.import stdio
.fun main.main: .arg 0;
/* main calls getchar */ 0 .fun stdio.getchar []
/* with no arguments */ []
/* and no environment */ { } [];
/* main defines str */ -> .vec main.main.0 [0..127]: [0..255] [];
/* main refs getchar */ <- stdio.getchar []

/* END of annotation file */

This is because ‘b’ has already been restricted to values not less than 20
by the first conditional.

- As well as capturing the conditions on argument 3, clauses 5 and 6
  indicate that ‘b’ should not be less than ‘a’ by giving them both the range
  [36...+∞].

11.9.3 Character Input

One annoyance in the C language has to do with character input. C lacks
exceptions, so when reading characters from a file, it has to signal the file’s
end with a special value, distinct from any valid character. Thus, character-
reading routines cannot return a character type, and their results cannot be
stored in character variables immediately.

Our annotation mechanism gets around this. The annotation file for the
stdio interface (relevant parts of which can be seen at the top of figure 11.3)
specifies that the getchar routine returns either a single-byte character (with
value 0...255), or -1 (the C EOF value).

Hence, the program of figure 11.2 on the facing page, which fills a vector
with character input, can ascertain that the values stored in the vector are
byte-sized, since the store happens after EOF has been checked for. The
program’s annotation file (bottom of figure 11.3) reflects this: not only does it tell
us that the stdio.getchar variable is referenced and the procedure it contains
called, it also mentions that the 128 words of the vector\(^4\) are set to values not

\(^4\)Identified as .vec main.main.0, since the variable str it is assigned to is local to main.
Figure 11.4: Interface (top) and implementation (bottom) of a string object.

```plaintext
settext gettext length print // object methods
create destroy // object operations
```

```plaintext
.import stdio;
.import string;
.import stdlib;

.var settext <- 0, // method fields
   gettext <- 1,
   length <- 2,
   print <- 3;
// .var text <- 4; data field -- not exported

.defun generic_gettext [str] str!4;
.defun generic_settext [str, t] str!4 <- t;
.defun generic_length [str] @string.strlen[str!gettext];
.defun generic_print [str] @stdio.printf ["%s", str!gettext];

.var create <- .defun create [] // object constructor
{
 .var object <- @stdlib.calloc[5, .big]; // allocate array
 object!0 <- .fun generic_settext; // set method fields
 object!1 <- .fun generic_gettext;
 object!2 <- .fun generic_length;
 object!3 <- .fun generic_print;
 .return object;

 .var destroy <- .defun destroy [object] // object destructor
    @stdlib.free[object]; // free array

.module strobj
```

greater than 255. This information can then be used to enable string packing optimisations (see section 10.4).

11.9.4 Typeless Objects

The typeless approach to object-oriented programming (OOP), as recently implemented in BCPL [Ric96a], is to regard objects as pointers to arrays of method addresses and data. The arrays are initialised explicitly by calling an appropriate procedure when an object is created; subclassing is done by copying the object arrays into larger ones and chaining the initialisation procedures of the two classes, methods are overridden by assignments to fields of the parent object, and casting to a superclass is simply a matter of throwing away the tail ends of object arrays and reversing the overrides. Thus, the only support needed in the compiler is some syntactic sugar (the ‘#’ operator) to turn method calls into calls of dereferenced object fields.

P does not have this condiment, but we can emulate the OOP style using
**Figure 11.5:** A program that uses string objects.

```
.import strobj;
.var main <- .defun main []
{
    .var message <- @strobj.create[]; // create a string object
    #strobj.settext[message, "Hello, world!\n"]; // set its text
    // actually implemented as:
    // @(message!strobj.settext)[message, ...]
    #strobj.print[message]; // print the string
    // actually implemented as:
    // @(message!strobj.print)[message];
}

.module main
```

normal memory accesses. The `strobj` module of figure 11.4 on the preceding page implements a "string" class; each string has methods to change or print its contents, and to measure their length.

Each object is an array\(^5\) whose method field offsets are exported as variables; the data field is "hidden" in that its offset has no such variable. The methods themselves are also hidden, and can only be accessed via the object fields to which the constructor procedure assigns them. These fields can, of course, be overridden by a superclass.

The program of figure 11.5 creates a string object, initialises it, then prints it out, using only the facilities of the `strobj` interface. Comments indicate how the object operations are actually implemented.

If we run the compiler on the two modules a few times, indicating that `strobj` is only used by the main program, we obtain the annotations shown in figures 11.7 and 11.6. Points to note are:

- Thanks to the reference annotations in `strobj`, the compiler is able to provide initial contexts for all calls made by `main`.

- The above does not mean that `main` has been specialised; any other implementation of `strobj` can be used with the current object code, although, if it references different parts of the context, it may not find the information it needs in `main`'s annotations and may not optimise as well.

- The values of the heap block containing the object (identified as `.new strobj.create.18`) supplied to the call of `strobj.create` are valid, even though the object does not yet exist at that point; they come into effect as soon as it is allocated.

- The annotations for all called procedures are included in those for `main`, since annotations need to be idempotent and referentially transparent.

- The compiler is currently incapable of passing constant strings in annotations, which is why they sometimes show up as `.table` arrays with unspecified contents, and sometimes as unknown (`?`) values. This is an implementation issue.

\(^5\)We use a single array, rather than the pair of arrays of methods and data used in BCPL, for the sake of simplicity.
Being parts of a heap block, the object's method fields keep their old values (0) as well as the new ones. This is a problem because the first method call will consider 0 as a procedure address and fall back on conservative assumptions about what it does. There is no easy way around this, and we have allowed the compiler to make the more liberal assumption that calls to address 0 can be safely ignored.

As this can fail in compiling low-level programs do jump to address 0, a command-line option should exist to disable it. Alternatively, we could
Figure 11.7: Annotation file for the string object implementation of figure 11.4. The annotations of some functions have been removed to save space, and some comments have been added.

```c
/* strobj.n: annotation file, automatically generated by
    p compiler (alpha>mips), version 0.99
    Time-stamp: <1998/05/10 15:22:01 qfp1@mouse.cl.can.ac.uk>
*/

/* initial values of method offsets */
.var strobj.settext: [0..0]
.var strobj.gettext: [1..1]
.var strobj.length: [2..2]
.var strobj.print: [3..3]

/* initial values of exported variables */
.var strobj.create: .fun strobj.create
.var strobj.destroy: .fun strobj.destroy

/* interfaces imported */
.import stdlib
.import string
.import stdio

/* module wrt which this one is specialised */
.module main

/* ...annotation for strobj.generic_gettext goes here... */

.fun strobj.generic_settext: .arg 2;
/* settext returns... */ .result .table 0 ([0..0])
/* ...its 2nd arg */ [.new strobj.create.18 ([0..0]), .table 0 ([0..0])];
/* and modifies... */ -> .new strobj.create.18 [4..4]: ?
/* message data field */ .new strobj.create.18 [([0..0]), .table 0 ([0..0])]

/* ...annotations for strobj.generic_print and strobj.generic_length go here... */

.fun strobj.create: .arg 0;
/* return heap block */ .result .new strobj.create.18 ([0..0]) [];
/* calls calloc... */ .fun stdlib.calloc []
/* 1st arg is 5 */ [[5..5], []]
/* 2nd arg is .big */ [4..4] []
/* no environment */ [] [];
/* new block contents */ .new strobj.create.18 [0..0]: [0..0] |
    .fun strobj.generic_settext [];
    .fun strobj.generic_gettext [];
    .fun strobj.generic_length [];
    .fun strobj.generic_print [];
/* references calloc */ <= stdlib.calloc []

/* destroy is not used: no useful information about it */
.fun strobj.destroy: .arg 1;
    .result ? [7];
    [] [7];
    {} [7];
    .new = [.min...max]: ? [7];
    <= stdlib.free [7];
    <= .new = [.min...max] [7]

/* END of annotation file */
```

use a non-initialising allocation routine (like stdlib.malloc) and set the blocks’ initial contents to \( \bot_{abv} \), but this would fail to detect references to uninitialised storage (see section 10.2.7).

- The compiler manages to deduce that `main` calls the routines to set and print the object’s text, as well as the object argument they are given. This eliminates the dereferencing of method fields which are a large part
of the OOP overhead.

This would also extend to methods added by subclasses, but not to overridden methods. In the latter case, we would not have been able to decide whether the old or new method would be invoked, because of the non-destructive updates of heap blocks that caused the previous problem. However, this is still better than not knowing which procedure is called at all.
Chapter 12

Conclusions and Further Work

12.1 Conclusions

Rather than concentrating on a single aspect of optimising compilation, our work has ranged far and wide over the field. Our conclusions are accordingly varied and numerous; they can be summarised as follows:

1. **Typeless languages are far from obsolete.** On the contrary, their simplicity is an advantage when it comes to integrating them with modern paradigms such as modular programming (see section 2.5.4), and their main disadvantages—difficulty of optimisation and the lack of safety inherent in untyped operations—can be overcome by clever compilation, as shown in chapter 10.

2. **Imperative and functional languages are closer than they appear.** Adopting the “everything returns a value” philosophy of functional languages can simplify the syntax of an imperative language and allow the use of concise, functional idioms; see section 2.4.2.

3. **Finding loops is not always a simple problem,** but by slightly altering the question from “where are the loops?” to “what loops does this code belong to?” we can extend the concept to irreducible flowgraphs. See section 4.4 and the rest of chapter 4.

4. **There exist alternatives to register colouring.** By rejecting the restrictive bond between allocation and assignment imposed by colouring allocators and dividing the problem into more or less independent tasks, we can produce more flexible and conceptually cleaner algorithms that can do a better job. See section 5.3, and the rest of chapter 5.

5. **Register assignment can be simple,** given a good register allocator. We can get away with simple assignment heuristics, such as those presented in chapter 6, which eliminate problems such as the $O(n^2)$ size of colouring matrices.

6. **In the real world, semantics are little more enlightening than the implementations they describe.** Even abstract semantics for a real programming language can be daunting, and just as prone to bugs as an implementation of the concepts they describe. This should be obvious from the complexity of the abstract interpretation presented in chapters 8 and 9. The situation is not improved by the fact that:
7. For a sufficiently complicated lattice, mathematical rigour can be a hindrance as well as a benefit. The problems arising in specification and implementation are largely orthogonal. Many of the former (evident, for example, in the operations of section 8.4) disappear in practice, without forewarning the implementor of issues arising in the actual code. Despite all this:

8. Elaborate abstract interpretations are computationally feasible, given the computing resources available today. In particular, the recovery of value ranges for variables and memory is a cost-effective process; see section 9.8.3.

9. Such analyses can recreate most of the typing information lacking from typeless languages, and enable type-like warnings and optimisations (see section 10.2.7). However:

10. The benefits are not limited to typeless compilation, since the information recovered is often more accurate than that provided by types. Better code verification becomes possible; out-of-bounds accesses, aliasing, divisions by zero and other run-time errors can often be detected ahead of time. Section 10.2 discusses this at length.

11. An interpretation of the intermediate code is simpler, since there are fewer operations to deal with than in the source code; See section 3.3.4 for an introduction to intermediate code, appendix C for a full specification, and chapter 9 for its abstract interpretation. This approach is also more portable, because many source languages can be translated to the same intermediate one.

12. Intermediate code interpretation enables low-level optimisations that are out of the realm of source-level analyses. Sections 10.3 and 10.4 contain the details. Finally,

13. Extending the framework across module boundaries is easy, and can be entirely transparent to the user (see chapter 11). This provides a solution to the conflict between sharing information for optimisation and hiding information for implementation independence.

### 12.2 Further Work

There are a number of directions in which the work described in this thesis can be extended. In particular, we would like to look at the following:

1. A number of the algorithms described in this thesis are implemented in a sub-optimal way by the P compiler. This is common in research code, but now that the algorithms are more or less well understood, some of them could be reimplemented for a substantial performance gain.

2. Various extensions to the P language, and their effect on our optimiser, could be investigated; see section 10.5 for details. In particular, assertion checking extensions would also allow us to assess the application of such optimisations to typed languages, since a type declaration is essentially a globally valid assertion of the variable's value range.

3. Better register assignment strategies should be investigated. The register span abstraction allows a number of approaches to be tried; whether
they will perform better than the current, naïve but adequate, algorithm remains to be seen.

4. The value range analysis engine (in particular, its memory block value aspect) could benefit from knowledge of the increments of loop induction variables. Given the hazy concept of a loop advocated here, this is a non-trivial problem, but work is already well underway on an algorithm to deal with this.

5. There exist methods for combining different abstract interpretations of the same code. We would like to implement some alternative interpretations and investigate the improvements such combinations could produce; value numbering is one obvious candidate.

6. Our minimal investigation of processor-specific optimisations based on value range analysis needs to be explored further. The effects of different memory latencies, cache sizes, prefetch strategies, instruction scheduling (at both compile and run time), as well as the different bit manipulation and memory access operations available should be examined. Also, now that CISC processors are back in fashion, the translation of our RISC intermediate code to such target architectures should be investigated, as should its interaction with register allocation.

7. Finally, writing optimisers can be a tedious and error-prone process, especially when the implementation has to be derived from a theoretical representation of the semantics. Since optimisation is the only compiler phase that has mostly escaped the grasp of automatic compiler construction systems, it would be interesting to investigate the feasibility of writing an "optimiser generator" to turn the formal descriptions presented in this thesis into working optimisers.
Appendix A

Mathematical Notation

Table A.1 lists the notation used for relations and operations on sets and logic quantities. Table A.2 lists the notation used to refer to entities (such as blocks, vregs, etc.) and functions involved in optimisation, along with the pages where they are introduced.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ x</td>
<td>\phi }</td>
</tr>
<tr>
<td>x \in \mathcal{S}</td>
<td>Set membership: x is a member of the set \mathcal{S}.</td>
</tr>
<tr>
<td>\mathcal{P} \subseteq \mathcal{Q}</td>
<td>Subset relation: \mathcal{P} is a subset of \mathcal{Q}.</td>
</tr>
<tr>
<td>\mathcal{P} \cup \mathcal{Q}</td>
<td>Set union: a set containing all the elements of \mathcal{P} and \mathcal{Q}.</td>
</tr>
<tr>
<td>\mathcal{P} \cap \mathcal{Q}</td>
<td>Set intersection: a set containing the elements common to \mathcal{P} and \mathcal{Q}.</td>
</tr>
<tr>
<td>\mathcal{P} \setminus \mathcal{Q}</td>
<td>Set difference: a set containing the elements of \mathcal{P} that are not in \mathcal{Q}.</td>
</tr>
<tr>
<td>\mathcal{P} \rightarrow \mathcal{Q}</td>
<td>Set of functions from \mathcal{P} to \mathcal{Q}.</td>
</tr>
<tr>
<td>\min(\mathcal{P})</td>
<td>Minimum element of \mathcal{P}, where \mathcal{P} \subseteq \mathbb{N}.</td>
</tr>
<tr>
<td>\max(\mathcal{P})</td>
<td>Maximum element of \mathcal{P}, where \mathcal{P} \subseteq \mathbb{N}.</td>
</tr>
<tr>
<td></td>
<td>Number of elements in \mathcal{P}.</td>
</tr>
<tr>
<td>\neg \phi</td>
<td>Logical negation: true if \phi is false.</td>
</tr>
<tr>
<td>\phi \land \psi</td>
<td>Logical conjunction: true if both \phi and \psi are true.</td>
</tr>
<tr>
<td>\phi \lor \psi</td>
<td>Logical disjunction: true if either \phi or \psi is true.</td>
</tr>
<tr>
<td>\forall x \in \mathcal{S} . \phi</td>
<td>Universal quantification: \phi is true for all values of x in the set \mathcal{S}.</td>
</tr>
<tr>
<td>\exists x \in \mathcal{S} . \phi</td>
<td>Existential quantification: \phi is true for some values of x in the set \mathcal{S}.</td>
</tr>
<tr>
<td>\lambda x . \text{expr}</td>
<td>Single-argument function returning expr for argument x.</td>
</tr>
</tbody>
</table>
Table A.2: Entities and functions involved in optimisation.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>α(⌜C⌝)</td>
<td>Abstraction of concrete context ⌝C⌟.</td>
<td>89</td>
</tr>
<tr>
<td>α(⌜C⌟)</td>
<td>Abstraction of concrete context vector ⌝C⌟.</td>
<td>89</td>
</tr>
<tr>
<td>γ(⌜abs⌟)</td>
<td>Concretisation of abstract context ⌝abs⌟.</td>
<td>89</td>
</tr>
<tr>
<td>γ(⌜abs⌟)</td>
<td>Concretisation of abstract context vector ⌝abs⌟.</td>
<td>89</td>
</tr>
<tr>
<td>A</td>
<td>Address size.</td>
<td>152</td>
</tr>
<tr>
<td>Addr(p)</td>
<td>Address of procedure p in compiled program.</td>
<td>86</td>
</tr>
<tr>
<td>Addrheap(i)</td>
<td>Addresses of heap blocks allocated at insn i.</td>
<td>112</td>
</tr>
<tr>
<td>Addrinsn(i)</td>
<td>Address of insn i in compiled program.</td>
<td>86</td>
</tr>
<tr>
<td>Addrbb[b]</td>
<td>Address of block b in compiled program.</td>
<td>86</td>
</tr>
<tr>
<td>Addrst(s)</td>
<td>Address of static object s in compiled program.</td>
<td>86</td>
</tr>
<tr>
<td>Addrreg(v)</td>
<td>Address of memory copy of vreg v (if any) in</td>
<td>86</td>
</tr>
<tr>
<td></td>
<td>compiled program.</td>
<td></td>
</tr>
<tr>
<td>Addrvars</td>
<td>Virtual registers corresponding to variables</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>whose address has been taken at some point.</td>
<td></td>
</tr>
<tr>
<td>Aexact[i1, i2, i3]</td>
<td>Restriction of interval i2 assuming that i1 is</td>
<td>104</td>
</tr>
<tr>
<td></td>
<td>restricted to i3.</td>
<td></td>
</tr>
<tr>
<td>Alias(r1, r2)</td>
<td>True if pointer represented by value range r1</td>
<td>146</td>
</tr>
<tr>
<td></td>
<td>may alias pointer represented by r2.</td>
<td></td>
</tr>
<tr>
<td>Annoarg(p)</td>
<td>Argument count annotation for procedure p.</td>
<td>180</td>
</tr>
<tr>
<td>Annocalc(⌜abs⌟, p, args)</td>
<td>Procedure call annotation for procedure p given</td>
<td>180</td>
</tr>
<tr>
<td></td>
<td>argument ranges args.</td>
<td></td>
</tr>
<tr>
<td>Annodef(⌜abs⌟, p, args)</td>
<td>Data definitions annotation for procedure p given</td>
<td>180</td>
</tr>
<tr>
<td></td>
<td>argument ranges args.</td>
<td></td>
</tr>
<tr>
<td>Annorefs(⌜abs⌟, p, args)</td>
<td>Data references annotation for procedure p given</td>
<td>180</td>
</tr>
<tr>
<td></td>
<td>argument ranges args.</td>
<td></td>
</tr>
<tr>
<td>Annorets(⌜abs⌟, p, args)</td>
<td>Return value annotation for procedure p given</td>
<td>180</td>
</tr>
<tr>
<td></td>
<td>argument ranges args.</td>
<td></td>
</tr>
<tr>
<td>Arcs</td>
<td>Arcs between insns.</td>
<td>85</td>
</tr>
<tr>
<td>Arg(i, n)</td>
<td>nth argument of insn i.</td>
<td>224</td>
</tr>
<tr>
<td>ArgT(i, n)</td>
<td>WIM3 type of nth argument of insn i.</td>
<td>224</td>
</tr>
<tr>
<td>ArgV(S, n)</td>
<td>Value of argument n of insn in state S.</td>
<td>87</td>
</tr>
<tr>
<td>ArgVabs(⌜abs⌟, i, n)</td>
<td>Abstract value of argument n of insn i in</td>
<td>123</td>
</tr>
<tr>
<td></td>
<td>context ⌝abs⌟.</td>
<td></td>
</tr>
<tr>
<td>Assocoff(r1, r2)</td>
<td>Subcomponents of value range r1 with offsets</td>
<td>108</td>
</tr>
<tr>
<td></td>
<td>associated to r2.</td>
<td></td>
</tr>
<tr>
<td>Assocub(s)</td>
<td>Top-level components of value range r1 associated</td>
<td>108</td>
</tr>
<tr>
<td></td>
<td>with r2.</td>
<td></td>
</tr>
<tr>
<td>Assocs</td>
<td>Range component associations.</td>
<td>104</td>
</tr>
<tr>
<td>Asyncv[i]</td>
<td>Vregs not saved to memory since their last</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>modification.</td>
<td></td>
</tr>
<tr>
<td>Avoidable(v, i)</td>
<td>True if the spill range of v containing i is not</td>
<td>156</td>
</tr>
<tr>
<td></td>
<td>compulsory.</td>
<td></td>
</tr>
<tr>
<td>Block(i)</td>
<td>Basic block containing insn i.</td>
<td>223</td>
</tr>
<tr>
<td>Blocks</td>
<td>Basic blocks.</td>
<td>223</td>
</tr>
<tr>
<td>Boundbeg(v, i)</td>
<td>Boundary condition of vreg v starting a span at</td>
<td>66</td>
</tr>
<tr>
<td></td>
<td>insn i.</td>
<td></td>
</tr>
<tr>
<td>Boundend(v, i)</td>
<td>Boundary condition of vreg v ending a span at</td>
<td>66</td>
</tr>
<tr>
<td></td>
<td>insn i.</td>
<td></td>
</tr>
</tbody>
</table>

continued on next page
### Symbol Table

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{BOUNDS}(m)</td>
<td>Bounds of vector or static m, as an integer interval.</td>
<td>148</td>
</tr>
<tr>
<td>\textbf{CALLARGS}(C_{abs})</td>
<td>Argument ranges for an imminent procedure call in context ( C_{abs} ).</td>
<td>180</td>
</tr>
<tr>
<td>\textbf{COMPID}(v, c_1, c_2)</td>
<td>Range component identifier generator function.</td>
<td>103</td>
</tr>
<tr>
<td>\textbf{CompIds}</td>
<td>Value range component identifiers.</td>
<td>103</td>
</tr>
<tr>
<td>\textbf{COMPONENTS}(C_{abs})</td>
<td>Value range components in abstract context ( C_{abs} ).</td>
<td>106</td>
</tr>
<tr>
<td>\textbf{Comps}</td>
<td>Value range components.</td>
<td>99</td>
</tr>
<tr>
<td>\textbf{CompVals}</td>
<td>Scalar value range component values.</td>
<td>99</td>
</tr>
<tr>
<td>\textbf{CONTASSOCs}(C_{abs})</td>
<td>Abstract context ( C_{abs} ) with rebuilt assoc.</td>
<td>107</td>
</tr>
<tr>
<td>\textbf{CONTENTS}(m, i)</td>
<td>Contents of memory range m at offset interval i.</td>
<td>101</td>
</tr>
<tr>
<td>\textbf{Contexts}</td>
<td>Concrete contexts.</td>
<td>88</td>
</tr>
<tr>
<td>\textbf{Contexts}_{abs}</td>
<td>Abstract contexts.</td>
<td>89</td>
</tr>
<tr>
<td>\textbf{CONTINUOUS}(S)</td>
<td>Minimal set of intervals containing all integers in S.</td>
<td>117</td>
</tr>
<tr>
<td>\textbf{DeadB}</td>
<td>Dead (unreachable) blocks.</td>
<td>27</td>
</tr>
<tr>
<td>\textbf{DECL}(p)</td>
<td>Vregs and vectors declared in procedure ( p ).</td>
<td>134</td>
</tr>
<tr>
<td>\textbf{DEFRESH}(i, r)</td>
<td>Value range ( r ) with fresh heap ids replaced by insn ( i ).</td>
<td>187</td>
</tr>
<tr>
<td>\textbf{DEFV}(i)</td>
<td>Vregs defined by insn ( i ).</td>
<td>224</td>
</tr>
<tr>
<td>\textbf{DISPLACEV}(i)</td>
<td>Vregs to be spilt just before insn ( i ).</td>
<td>51</td>
</tr>
<tr>
<td>\textbf{DIST}(v, i)</td>
<td>Average distance to next use of vreg ( v ) from insn ( i ).</td>
<td>47</td>
</tr>
<tr>
<td>\textbf{DISTINCT}(r)</td>
<td>Non-overlapping interval components of value range ( r ).</td>
<td>124</td>
</tr>
<tr>
<td>\textbf{DISTL}(i)</td>
<td>List of vregs in decreasing order of distance to next use at insn ( i ).</td>
<td>48</td>
</tr>
<tr>
<td>\textbf{DIST5}(i, n)</td>
<td>First ( n ) elements of ( \text{DISTL}(i) ), except for ones present in ( \text{DEFV}(i) ).</td>
<td>52</td>
</tr>
<tr>
<td>\textbf{EDGE}(b_1, b_2)</td>
<td>Label of edge leading from block ( b_1 ) to ( b_2 ).</td>
<td>27</td>
</tr>
<tr>
<td>\textbf{EntryB}</td>
<td>Blocks containing procedure entrypoints.</td>
<td>223</td>
</tr>
<tr>
<td>\textbf{FIRST}(p)</td>
<td>First block of procedure ( p ).</td>
<td>223</td>
</tr>
<tr>
<td>\textbf{FIRST}(b)</td>
<td>First insn of block ( b ).</td>
<td>223</td>
</tr>
<tr>
<td>\textbf{FLATTEN}(i)</td>
<td>All integers contained in set of intervals ( i ).</td>
<td>117</td>
</tr>
<tr>
<td>\textbf{FROMT}(b_1, b_2)</td>
<td>Blocks that can reach ( b_2 ) and are reachable from ( b_1 ) via forward and cross edges only.</td>
<td>36</td>
</tr>
<tr>
<td>\textbf{FUNCALL}(i, V, M)</td>
<td>Environment after return from call insn ( i ) in environment ( (V, M) ).</td>
<td>87</td>
</tr>
<tr>
<td>\textbf{Globals}</td>
<td>Virtual registers corresponding to global variables.</td>
<td>31</td>
</tr>
<tr>
<td>\textbf{HeapVals}</td>
<td>Heap memory range components.</td>
<td>98</td>
</tr>
<tr>
<td>\textbf{HIGH}(S)</td>
<td>Upper limits of intervals in set ( S ).</td>
<td>110</td>
</tr>
<tr>
<td>\textbf{Hregs}</td>
<td>Real registers (Hregs ( \subseteq ) Vregs).</td>
<td>21</td>
</tr>
<tr>
<td>\textbf{HostPri}(v_n, v, i)</td>
<td>Rating of vreg ( v_n ) as a host for ( v ) during spill range containing insn ( i ).</td>
<td>156</td>
</tr>
<tr>
<td>\textbf{HOSTS}(v, i)</td>
<td>Tuples containing host candidates for vreg ( v ) at insn ( i ), and their ratings.</td>
<td>156</td>
</tr>
<tr>
<td>\textbf{INDIVIDUAL}(c)</td>
<td>Singleton intervals containing all integers in value range component ( c ) and/or its offset.</td>
<td>112</td>
</tr>
</tbody>
</table>

\textit{continued on next page}
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{INEXACT}{a}</td>
<td>Assoc set (a) with all assoc s converted to inexact.</td>
<td>115</td>
</tr>
<tr>
<td>\text{INFER}_2{op, r, arg}</td>
<td>Restriction of value range (r) of second operand to an insn with opcode (op) and third operand (arg), according to kind inference.</td>
<td>185</td>
</tr>
<tr>
<td>\text{INFER}_3{op, r, arg}</td>
<td>Restriction of value range (r) of third operand to an insn with opcode (op) and second operand (arg), according to kind inference.</td>
<td>185</td>
</tr>
<tr>
<td>\text{INFER}<em>{\text{abs}}{C</em>{\text{abs}}, i}</td>
<td>Kind inference operator for abstract context (C_{\text{abs}}) before insn (i).</td>
<td>185</td>
</tr>
<tr>
<td>\text{INIT}_{\text{vreg}}{p}</td>
<td>Initial vreg environment for procedure (p).</td>
<td>179</td>
</tr>
<tr>
<td>\text{INIT}_{\text{mem}}{p}</td>
<td>Initial memory environment for procedure (p).</td>
<td>179</td>
</tr>
<tr>
<td>\text{INSN}_{\text{addr}}{v}</td>
<td>Insn with address (v) in compiled program.</td>
<td>86</td>
</tr>
<tr>
<td>\text{INSN}_{\text{ARCS}}{p, opcode}</td>
<td>Arcs in procedure (p) leading to an insn with the given opcode.</td>
<td>180</td>
</tr>
<tr>
<td>\text{INSTS}</td>
<td>All the insns in the program.</td>
<td>223</td>
</tr>
<tr>
<td>\text{INSTS}{p}</td>
<td>Insns in basic block or procedure (p).</td>
<td>223</td>
</tr>
<tr>
<td>\text{INTERVAL}{i_1, i_2}</td>
<td>Insns between (i_1) and (i_2) inclusive.</td>
<td>67</td>
</tr>
<tr>
<td>\text{INTERVALs}</td>
<td>Integer intervals.</td>
<td>97</td>
</tr>
<tr>
<td>\text{LAST}_{\text{b}}{b}</td>
<td>Last insn of block (b).</td>
<td>223</td>
</tr>
<tr>
<td>\text{LEFT}_{\text{ID}}{id}</td>
<td>Unique identifier generator function.</td>
<td>109</td>
</tr>
<tr>
<td>\text{LiveB}</td>
<td>Live blocks.</td>
<td>27</td>
</tr>
<tr>
<td>\text{LIVE}_{\text{v}}{i}</td>
<td>Vregs live just before insn (i).</td>
<td>29</td>
</tr>
<tr>
<td>\text{LIVE}_{\text{RANGE}}{v, i}</td>
<td>Live range of vreg (v) containing insn (i).</td>
<td>155</td>
</tr>
<tr>
<td>\text{LIVE}_{\text{SIZE}}{v, i}</td>
<td>Maximum size of vreg (v) 's value in range containing insn (i).</td>
<td>156</td>
</tr>
<tr>
<td>\text{LIVE}_{\text{SPANS}}{v, i}</td>
<td>Spans contained in live range of vreg (v) containing insn (i).</td>
<td>155</td>
</tr>
<tr>
<td>\text{LOAD}_{\text{v}}{i}</td>
<td>Vregs reloaded just before insn (i).</td>
<td>51</td>
</tr>
<tr>
<td>\text{LOAD}<em>{\text{OP}}{C</em>{\text{abs}}, a, r}</td>
<td>Contents of (C_{\text{abs}}{a}) corresponding to addresses in value range (r).</td>
<td>134</td>
</tr>
<tr>
<td>\text{LOOP}_{\text{ID}}{b}</td>
<td>Loop identifier of block (b).</td>
<td>36</td>
</tr>
<tr>
<td>\text{MAP}_{\text{loop}}{l}</td>
<td>New mapping of loop identifier (l) during loop detection.</td>
<td>36</td>
</tr>
<tr>
<td>\text{LOW}{S}</td>
<td>Lower limits of intervals in set (S).</td>
<td>110</td>
</tr>
<tr>
<td>\text{MAP}_{\text{abs}}{\text{map}, r}</td>
<td>Value range (r) updated according to identifier mapping (\text{map}).</td>
<td>105</td>
</tr>
<tr>
<td>\text{MAP}_{\text{ASG}}{\text{map}, asc}</td>
<td>Assoc set (asc) updated according to identifier mapping (\text{map}).</td>
<td>105</td>
</tr>
<tr>
<td>\text{MAP}<em>{\text{ab}}{C</em>{\text{abs}}}</td>
<td>Mapping for updated component identifiers in context (C_{\text{abs}}).</td>
<td>105</td>
</tr>
<tr>
<td>\text{MAP}<em>{\text{AF}}{C</em>{\text{abs}1}, C_{\text{abs}2}}</td>
<td>Mapping for updated component identifiers after widening of context (C_{\text{abs}1}) by (C_{\text{abs}2}).</td>
<td>118</td>
</tr>
<tr>
<td>\text{Memobjs}</td>
<td>Memory block identifiers.</td>
<td>101</td>
</tr>
<tr>
<td>\text{MEM}<em>{\text{UPDATES}}{C</em>{\text{abs}}, p, args}</td>
<td>Value range of data items updated by procedure (p) given argument ranges (args).</td>
<td>182</td>
</tr>
<tr>
<td>\text{Memvals}_{\text{abs}}</td>
<td>Memory block range values.</td>
<td>101</td>
</tr>
<tr>
<td>\text{MEnv}</td>
<td>Environments of memory values.</td>
<td>85</td>
</tr>
<tr>
<td>\text{MEnv}_{\text{abs}}</td>
<td>Environments of abstract memory values.</td>
<td>102</td>
</tr>
<tr>
<td>(N_s{i, S})</td>
<td>Number of live, unspilt vregs at insn (i) with spill set (S).</td>
<td>51</td>
</tr>
</tbody>
</table>

\textit{continued on next page}
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{NEST}_b(b)</td>
<td>Nesting level of block b.</td>
<td>36</td>
</tr>
<tr>
<td>\texttt{NEST}_b(b_1, b_2)</td>
<td>Nesting level of edge leading from b_1 to b_2.</td>
<td>36</td>
</tr>
<tr>
<td>\texttt{NEST}_i(i)</td>
<td>Nesting level of insn i.</td>
<td>46</td>
</tr>
<tr>
<td>\texttt{NEWASSOC}_c(C, C_abs)</td>
<td>Rebuilt assoc for value range component c in abstract context C_abs.</td>
<td>106</td>
</tr>
<tr>
<td>\texttt{NEXT}_abs(C_abs)</td>
<td>Context vector transition function for WIM3 abstract semantics.</td>
<td>89</td>
</tr>
<tr>
<td>\texttt{NEXT}_con(C)</td>
<td>Context vector transition function for WIM3 concrete semantics.</td>
<td>88</td>
</tr>
<tr>
<td>\texttt{NEWSTATE}(S)</td>
<td>State transition function for WIM3 operational semantics.</td>
<td>87</td>
</tr>
<tr>
<td>\texttt{NORMALISE}(\tau)</td>
<td>Normalised equivalent of value range \tau.</td>
<td>124</td>
</tr>
<tr>
<td>\texttt{NONDET}(C_abs, \tau)</td>
<td>True if address range \tau is not deterministic.</td>
<td>135</td>
</tr>
<tr>
<td>\texttt{NOTSPLIT}(v_h, v_b, i)</td>
<td>True if vreg _h is not split at the fringes of the spill range of v_b containing insn i.</td>
<td>155</td>
</tr>
<tr>
<td>\texttt{NSUCCESSOR}(i)</td>
<td>Successors of insn i at the same or higher nesting level.</td>
<td>46</td>
</tr>
<tr>
<td>Offsets</td>
<td>Memory block offset ranges.</td>
<td>100</td>
</tr>
<tr>
<td>Offsets[val_1, val_2]</td>
<td>Offset intervals for which a memory range with offsets val_1 should be defined after a store to offsets val_2.</td>
<td>136</td>
</tr>
<tr>
<td>\texttt{OFFSET}(v)</td>
<td>Offset of base of vector v from the bottom of its stack frame.</td>
<td>86</td>
</tr>
<tr>
<td>\texttt{OP}(i)</td>
<td>Opcode of insn i.</td>
<td>225</td>
</tr>
<tr>
<td>\texttt{PACKABLE}(C_abs, n)</td>
<td>Identifiers of memory objects each of whose cells can be packed in n bits.</td>
<td>166</td>
</tr>
<tr>
<td>\texttt{PART}__n(\tau)</td>
<td>Range \tau restricted to values greater than or equal to integer n.</td>
<td>128</td>
</tr>
<tr>
<td>\texttt{PART}__n(\tau)</td>
<td>Range \tau restricted to values less than integer n.</td>
<td>128</td>
</tr>
<tr>
<td>\texttt{PART}__n(\tau)</td>
<td>Heap, static and vector components of value range \tau.</td>
<td>100</td>
</tr>
<tr>
<td>\texttt{PART}_base(\tau_1, \tau_2)</td>
<td>Components of value range \tau_1 with address bases not present in \tau_2 and vice versa.</td>
<td>116</td>
</tr>
<tr>
<td>\texttt{PART}_int(\tau)</td>
<td>Integer interval components of value range \tau.</td>
<td>100</td>
</tr>
<tr>
<td>\texttt{PART}_int(\tau)</td>
<td>Label, procedure, and vreg address components of value range \tau.</td>
<td>100</td>
</tr>
<tr>
<td>\texttt{PCSARG}(n)</td>
<td>Vreg corresponding to argument n in the procedure calling standard.</td>
<td>179</td>
</tr>
<tr>
<td>\texttt{PCSRETURN}</td>
<td>Return value hreg according to the procedure calling standard.</td>
<td>181</td>
</tr>
<tr>
<td>\texttt{PCSSP}</td>
<td>Stack pointer hreg according to the procedure calling standard.</td>
<td>112</td>
</tr>
<tr>
<td>\texttt{PREDECESSORS}(b)</td>
<td>Predecessors of block b.</td>
<td>223</td>
</tr>
<tr>
<td>\texttt{PREDECESSORS}(i)</td>
<td>Predecessors of insn i.</td>
<td>223</td>
</tr>
<tr>
<td>\texttt{PROC}__calls(C, p, args)</td>
<td>Procedures and contexts called directly from p, given arguments args.</td>
<td>181</td>
</tr>
<tr>
<td>\texttt{PROC}__calls(p)</td>
<td>Procedures.</td>
<td>223</td>
</tr>
<tr>
<td>\texttt{PROC}__calls(C, c)</td>
<td>Assoc set of value range component c.</td>
<td>100</td>
</tr>
<tr>
<td>\texttt{PROC}__calls(r)</td>
<td>Assoc sets of components of value range r.</td>
<td>100</td>
</tr>
<tr>
<td>\texttt{PROC}__calls(c)</td>
<td>Base of abstract address component c.</td>
<td>100</td>
</tr>
<tr>
<td>Symbol</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>PROIbeg(s)</td>
<td>Boundary condition at start of span s.</td>
<td></td>
</tr>
<tr>
<td>PROIend(s)</td>
<td>Boundary condition at end of span s.</td>
<td></td>
</tr>
<tr>
<td>PROIid(c)</td>
<td>Identifier of value range component c.</td>
<td></td>
</tr>
<tr>
<td>PROIidg(τ)</td>
<td>Identifiers of components of value range τ.</td>
<td></td>
</tr>
<tr>
<td>PROIibeg(s)</td>
<td>Start insn of span s.</td>
<td></td>
</tr>
<tr>
<td>PROIiend(s)</td>
<td>End insn of span s.</td>
<td></td>
</tr>
<tr>
<td>PROIinds(τ, m)</td>
<td>Indices in memory value m.</td>
<td></td>
</tr>
<tr>
<td>PROIM(Cabs)</td>
<td>Memory environment from abstract context Cabs.</td>
<td></td>
</tr>
<tr>
<td>PROIoff(c)</td>
<td>Offset of abstract address component c.</td>
<td></td>
</tr>
<tr>
<td>PROIref(s)</td>
<td>Reference count for span s.</td>
<td></td>
</tr>
<tr>
<td>PROIsize(s)</td>
<td>Size of value held in span s.</td>
<td></td>
</tr>
<tr>
<td>PROI(Cabs)</td>
<td>Vreg environment from abstract context Cabs.</td>
<td></td>
</tr>
<tr>
<td>PROIv(c)</td>
<td>Value of range component c.</td>
<td></td>
</tr>
<tr>
<td>PROIvbe(τ)</td>
<td>Values of components of range τ.</td>
<td></td>
</tr>
<tr>
<td>PROIvreg(s)</td>
<td>Vreg to which span s belongs.</td>
<td></td>
</tr>
<tr>
<td>R_W</td>
<td>Word size, as a value range.</td>
<td></td>
</tr>
<tr>
<td>RANGEASSCS(τ, Cabs)</td>
<td>Value range τ with its assocs rebuilt from context Cabs.</td>
<td></td>
</tr>
<tr>
<td>REACHb(b)</td>
<td>Blocks reachable from block b via forward and cross edges only.</td>
<td></td>
</tr>
<tr>
<td>RECURSIVE(Cabs, p)</td>
<td>True if p is a recursive procedure.</td>
<td></td>
</tr>
<tr>
<td>REDASSCS(τ1, τ2)</td>
<td>Redundant assocs from value range τ1 to τ2.</td>
<td></td>
</tr>
<tr>
<td>REDRANGE(τ1, Cabs)</td>
<td>Value range τ1 with redundant assocs to members of context Cabs removed.</td>
<td></td>
</tr>
<tr>
<td>REFv(i)</td>
<td>Vregs referenced by insn i.</td>
<td></td>
</tr>
<tr>
<td>RESLESS(Cabs:1, c1, Cabs:2, c2)</td>
<td>True if abstract context Cabs:1 restricted by component c1 is less than Cabs:2 restricted by c2.</td>
<td></td>
</tr>
<tr>
<td>RESTALL(Cabs, τ1, τ2)</td>
<td>Abstract context Cabs restricted assuming value range τ1 is constrained to values from τ2.</td>
<td></td>
</tr>
<tr>
<td>RESTARGS(Cabs, p, args)</td>
<td>Abstract context Cabs restricted assuming procedure p takes argument ranges args.</td>
<td></td>
</tr>
<tr>
<td>RESTRICTabs(Cabs, τ)</td>
<td>Abstract context Cabs restricted to values consistent with value range τ.</td>
<td></td>
</tr>
<tr>
<td>RESTRICTabs(τ1, τ2)</td>
<td>Range τ1 restricted to values consistent with τ2.</td>
<td></td>
</tr>
<tr>
<td>RESULT(s, a)</td>
<td>Result of insn at the head of arc a.</td>
<td></td>
</tr>
<tr>
<td>RESULTd(τ1, τ2)</td>
<td>Result operator for all insns.</td>
<td></td>
</tr>
<tr>
<td>RIGHTId(id)</td>
<td>Unique identifier generator function.</td>
<td></td>
</tr>
<tr>
<td>SAVEab(i)</td>
<td>Vregs saved to memory due to side effects of load/store insn i, as updated by value range analysis.</td>
<td></td>
</tr>
<tr>
<td>SAVEanno(s, a, args)</td>
<td>Vregs to save before a call at arc a with arguments args.</td>
<td></td>
</tr>
<tr>
<td>SAVEext(i)</td>
<td>Vregs saved to memory due to side effects of load insn i.</td>
<td></td>
</tr>
<tr>
<td>SAVEext-c3(i)</td>
<td>Vregs saved to memory due to side effects of store or call insn i.</td>
<td></td>
</tr>
</tbody>
</table>
Union of spill set on exit from memory environment after a store of value to spill just before insn. Vregs range of vreg to spill just before a call at arcc.

Span fringe of spill range of vreg.

Vregs ending a span at insn.

States/

WIM3 operational semantics states.

StatesInit

WIM3 operational semantics initial states.

Statics

Static object identifiers.

StatVals

Static memory range components.

StoreVReg(Cabs, cont, v, addr) Value range of vreg v updated in context Cabs by a store of range cont to range addr.

StoreMem(Cabs, a, cont, addr) Memory environment after a store of value range cont to address range addr in context Cabs(a).

continued from previous page

Symbol | Description |
--- | --- |
SAVEV[i] | Vregs saved to memory before insn i. |
SEnv | "Shadow" environments of initial argument values. |
SIZEabs(r) | Number of bits needed to represent a value from range r. |
SltOp(Cabs, r1, r2) | Abstract context Cabs with ranges split in preparation for a comparison between r1 and r2. |
SPANbeg(i) | Vregs starting a span at insn i. |
SPANend(i) | Vregs ending a span at insn i. |
Spans | Register spans. |
SPILLbeg(i) | Vregs spilt just before insn i. |
SPILLend(i) | Intersection of spill sets on exit from i's predecessors. |
SPILLrange(Cabs, a, args) | Vregs to spill before a call at arc a with arguments args. |
SPILLbef(i) | Vregs spilt just before insn i. |
SPILLSblock(i) | Spills before the reloads and saves preceding insn i. |
SPILLdec(i) | Vregs spilt before insn i due to i's side effects. |
SPILLer(i) | Union of spill sets on exit from i's predecessors. |
SPILLpos(i) | Difference of SPILLer(i) and SPILLend(i). |
SPILLRange(v, i) | Spill range of vreg v containing insn i. |
SPILLSize(v, i) | Maximum size of vreg v's value in spill range containing insn i. |
SPILLSpan(v, i) | Span fringe of spill range of vreg v containing insn i. |
SPIVAll(Cabs, r1, r2) | Abstract context Cabs split so that value ranges r1, r2 have no partially overlapping components. |
SPLITabs(Cabs, r, n) | Abstract context Cabs split in accordance with split of value range r at integer n. |
SPLITabs(Cabs, S) | Abstract context Cabs after a multiple split as specified by range/integer pairs from set S. |
SPLIT(Cabs, r1, r2, n) | Split of value range r2 corresponding to split of r1 at integer n in context Cabs. |
SPLITComp(r, n) | Any components of value range r containing integer n (but not as their lower limit). |
States |

WIM3 operational semantics states.

StatesInit

WIM3 operational semantics initial states.

Statics

Static object identifiers.

StatVals

Static memory range components.

StoreVReg(Cabs, cont, v, addr) Value range of vreg v updated in context Cabs by a store of range cont to range addr.

StoreMem(Cabs, a, cont, addr) Memory environment after a store of value range cont to address range addr in context Cabs(a). 

continued on next page
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{STRIP}_{\text{as}}(a, \text{ids})$</td>
<td>Assoc set $a$ without associ to components with identifiers in $\text{ids}$.</td>
<td>135</td>
</tr>
<tr>
<td>$\text{STRIP}_{\text{ar}}(r, \text{ids})$</td>
<td>Value range $r$ without associ to components with identifiers in $\text{ids}$.</td>
<td>135</td>
</tr>
<tr>
<td>$\text{STRIP}<em>{\text{abs}}(C</em>{\text{abs}}, \text{ids})$</td>
<td>Abstract context $C_{\text{abs}}$ without associ to components with identifiers in $\text{ids}$.</td>
<td>135</td>
</tr>
<tr>
<td>$\text{Succ}_{\text{abs}}(b)$</td>
<td>Successors of block $b$, updated by value range analysis.</td>
<td>143</td>
</tr>
<tr>
<td>$\text{Succ}_{\text{i}}(i)$</td>
<td>Successors of insn $i$.</td>
<td>223</td>
</tr>
<tr>
<td>$\text{Succ}_{\text{b}}(b)$</td>
<td>Successors of block $b$.</td>
<td>223</td>
</tr>
<tr>
<td>$\text{UPDATE}<em>{\text{ids}}(C</em>{\text{abs}}, \text{map})$</td>
<td>Abstract context $C_{\text{abs}}$ with components identifiers updated by $\text{map}$.</td>
<td>105</td>
</tr>
<tr>
<td>$\text{Used}$</td>
<td>Used vregs.</td>
<td>31</td>
</tr>
<tr>
<td>$\text{Values}$</td>
<td>Concrete values.</td>
<td>85</td>
</tr>
<tr>
<td>$\text{Values}_{\text{abs}}$</td>
<td>Abstract values.</td>
<td>93</td>
</tr>
<tr>
<td>$\text{Vectors}$</td>
<td>Vector identifiers.</td>
<td>224</td>
</tr>
<tr>
<td>$\text{VectVals}$</td>
<td>Vector range components.</td>
<td>98</td>
</tr>
<tr>
<td>$\text{VEnv}_{\text{v}}$</td>
<td>Environments of vreg values.</td>
<td>85</td>
</tr>
<tr>
<td>$\text{VEnv}_{\text{abs}}$</td>
<td>Environments of abstract vreg values.</td>
<td>102</td>
</tr>
<tr>
<td>$\text{VReg}_{\text{abs}}(v)$</td>
<td>Vreg whose memory copy (if any) has address $v$ in compiled program.</td>
<td>86</td>
</tr>
<tr>
<td>$\text{Vregs}$</td>
<td>Virtual registers.</td>
<td>21</td>
</tr>
<tr>
<td>$\mathcal{W}$</td>
<td>Word size.</td>
<td>130</td>
</tr>
<tr>
<td>$\text{Widen}$</td>
<td>Widening/narrowing arcs.</td>
<td>90</td>
</tr>
</tbody>
</table>
Appendix B

P Reference Manual

B.1 Syntax

The following is a summary of the P language's grammar. Terminal symbols of the "reserved" (system identifier) type are shown in typewriter style; all other symbols are non-terminal, except for name, number and string. The grammar's target symbol is translation-unit. An opt subscript indicates that the symbol is optional. Numbers in brackets next to productions indicate the sections in which the corresponding constructs are discussed. Note that a program's conformance to this grammar guarantees that it will be accepted by the P compiler's parser; it does not guarantee that the program is free of compile-time errors, many of which are detected in the translation phase. This, along with P's typelessness and its lack of distinctions between expressions and statements, accounts for the fact that the grammar is only two and a half pages long. (By way of comparison, the canonical C grammar, as presented in Appendix A of [KR88], is 6 pages long.)

\[
\begin{align*}
\text{translation-unit} &::= \text{expr-colon-list} & \quad \text{(B.2.2)} \\
\text{arithmetic} &::= (\text{expr}) & \quad \text{(B.2.1)} \\
& | \text{expr postfix-op} & \quad \text{(B.2.12)} \\
& | \text{prefix-op expr} & \quad \text{(B.2.11)} \\
& | \text{expr dyadic-op expr} & \quad \text{(B.2.10)} \\
\text{array} &::= \text{.vec array-spec} & \quad \text{(B.2.9)} \\
& | \text{.table array-spec} & \quad \text{(B.2.9)} \\
& | \text{.new expr} & \quad \text{(B.2.9)} \\
& | \text{.free expr} & \quad \text{(B.2.9)} \\
\text{array-spec} &::= \text{expropt} [\text{expr-comma-list}] & \quad \text{(B.2.9)} \\
& | \text{expr} & \quad \text{(B.2.9)} \\
\text{call} &::= @\text{expr} [\text{expr-comma-listopt}] & \quad \text{(B.2.3)} \\
\text{conditional} &::= \text{.if} [\text{expr}] \text{expr} & \quad \text{(B.2.6)} \\
& | \text{.test} [\text{expr}] \text{expr} \text{else} \text{expr} & \quad \text{(B.2.6)} \\
& | \text{.switch} [\text{expr}] \text{expr} & \quad \text{(B.2.6)} \\
\text{decl-list} &::= \text{name initialiseropt} & \quad \text{(B.2.2)} \\
& | \text{name initialiseropt} , \text{decl-list} \\
\text{declaration} &::= \text{.var} \text{decl-list} & \quad \text{(B.2.4)} \\
\text{dyadic-op} &::= ! | << | >> | >> > | & | ^ | | * | / | % | + | - | == | <> | > | < | > = | <= | && | --- | || | - > | <= & \quad \text{(B.2.10)}
\end{align*}
\]
Grammar ambiguities are resolved by assigning precedence values to the operators, as shown in table B.1 on the next page. All dyadic operators, except for `<errorеть`<errorить>, associate to the left.

The syntax of interface files is extremely simple and can be described thus:

\[
\text{interface-file} \::= \text{name}_{\text{opt}}
\]
The name, number and string tokens are terminals as far as the parser is concerned; they are recognised by the lexical analyser according to the following rules:

\[
\begin{align*}
\text{name} &::= \text{simple-name} \\
& \quad | \text{simple-name} . \text{name} \\
\text{number} &::= \text{character}' \\
& \quad | \text{decimal} \\
& \quad | \text{octal} \\
& \quad | \text{hexadecimal} \\
\text{string} &::= "\text{string-chars}\text{opt}" \\
\text{alpha} &::= \text{alpha} | \text{dec-digit} | \text{punctuation} | \_ \\
& \quad | \text{escape} \\
\text{dec-digits} &::= \text{dec-digit} \text{dec-digits}\text{opt} \\
\text{dec-digit} &::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 \\
\text{decimal} &::= 0 \\
& \quad | \text{non-zero-digit} \text{dec-digits}\text{opt} \\
\text{escape} &::= \text{a} | \text{b} | \text{f} | \text{n} | \text{r} | \text{t} | \text{v} | \_ | ? | ' | \\
\end{align*}
\]

Table B.1: P operator precedence.
| x hex-digit hex-digit |
| oct-digit oct-digit oct-digit |

**hex-digit** ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f |
A | B | C | D | E | F

**hex-digits** ::= hex-digit hex-digits opt

**hexadecimal** ::= 0x hex-digits

**name-char** ::= alpha | dec-digit | _

**name-chars** ::= name-char name-chars opt

**non-zero-digit** ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

**octal** ::= 0 oct-digits

**oct-digit** ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

**oct-digits** ::= oct-digit oct-digits opt

**punctuation** ::= ! | # | $ | % | & | ( | ) | * | + | - | . | / | : | ; | < |
= | > | ? | [ | ] | ^ | _ | '{' | '|' | '}'

**simple-name** ::= alpha name-chars opt
| _ name-chars opt

**string-chars** ::= character string-chars opt
| ' string-chars opt

### B.2 Informal Semantics

#### B.2.1 Values and Evaluation

All P values are machine words, i.e. \( n \)-bit binary integer quantities where \( n \) is implementation dependent. Every P expression returns such a value, but in some cases the value returned may be undefined.

Expressions may be evaluated in two or three different ways. **Rmode** evaluation returns the value of the expression itself, whereas **Lmode** evaluation returns the address (or register) in which that value is stored. Not all expressions can be evaluated in both modes; valid Rmode expressions are called **Rvalues**, and **Lvalues** are defined similarly. The subset of Rvalues which can be evaluated at compile-time is used in arguments to case labels (see section B.2.6) and table initialisers (see section B.2.9), and is sometimes said to be evaluated in C (for Constant) mode.

As has been mentioned already, the order of evaluation within an expression is governed by the precedences of the operators. These may be overridden by using parentheses in the usual way.

Unless otherwise specified, all the language constructs listed below are only valid in Rmode.

#### B.2.2 Top Level

The kinds of expressions that may appear at the top level of a P translation unit (hereafter TU) are restricted to procedure definitions, variable declarations, assignments, and import/export expressions.

Every variable declared at this level is **global**, i.e. its scope is the whole TU, unless its name is listed in one of the interfaces the TU exports, in which case its scope extends to any TU that imports the interface.
The initial values of global variables are undefined unless explicitly given in the declarations or specified by assignment statements at the top level. The initialising expressions are evaluated in Constant mode, i.e. should not be defined in terms of other variables.

Import expressions cause the corresponding interface files to be opened and their contents to be made visible to the rest of the TU as imported variables, each prefixed by the interface name and a dot (\texttt{.}). (Import expressions are usually found at the beginning of a TU, so that the imported variables will be visible to the whole of the unit.) Importing the same interface more than once is an error. Interface names must conform to the rules for variable names described in section B.2.4.

Export (\texttt{.module}) expressions cause the corresponding interface files to be opened and their contents to be looked up in the current scope. The corresponding global variables are made available to every other translation unit that imports the interface. It is an error to export an interface when not all of its members are defined as globals in the current TU (thus export expressions are usually found at the end of the TU, when all globals have been defined).

The values returned by import and export expressions are undefined.

### B.2.3 Procedures

Procedures may only be defined at the top level, either stand-alone or as values of global variables. The expression

\texttt{.defun name [ arglist ] expr}

defines a procedure, with formal arguments listed in \texttt{arglist} (which is comma-delimited, and possibly empty) and \texttt{expr} as the procedure body; it returns the address of the procedure in memory.

Once a procedure has been defined, its address may be referenced using \texttt{.fun name}, where \texttt{name} is the name given in the procedure definition. Defining a second procedure with the same name is an error, but variables and labels with the same name may be defined, since their name spaces are separate.

Procedures are called using the call expression:

\texttt{.@ address [ exprlist ]}

This causes the body of the procedure whose address is \texttt{address} to be evaluated in an environment where the formal arguments are bound to the values of the actual parameters (from \texttt{exprlist}, comma-separated), and return the result.

### B.2.4 Variable Declarations

Variables are declared using the \texttt{.var} expression. The variable's scope extends from the declaration to the end of the smallest enclosing sequence expression, procedure body or TU. If another variable with the same name exists in an enclosing scope, it is hidden for the duration of the inner scope; it is an error to declare two variables with the same name in the same scope.

Variable names in declarations should be unqualified, i.e. should not contain an interface prefix. The components of variable names (i.e. interface names and unqualified parts) may consist of any sequence of alphanumeric characters and underscores, starting with a non-numeric character. Case is significant.

The initial values of variables are undefined, unless explicitly given in a declaration of the form

\texttt{.var varname <- expr} .
An expression consisting of the name of a variable returns its current value (in Rmode) or the location in which it is stored (in Lmode); the latter may be a memory address or register, but the difference should be invisible to the user. Accessing the value of an undeclared variable is an error.

### B.2.5 Sequence Expressions and Results

A sequence expression (SE) has the form

\[
\{ \text{expr; expr; ... } \}
\]

When evaluated, an SE causes each of its component expressions to be evaluated in sequence. The return value of an SE is the argument of the first `result` expression executed in it.

A result expression `result expr` causes the execution of the smallest enclosing SE to be terminated, returning the value of `expr`. A return expression `return expr` is similar, but causes the execution of the current procedure to be terminated, returning `expr` as the result of the procedure. It is an error to use `.result` and `.return` outside SEs and procedures respectively\(^1\).

### B.2.6 Conditionals

The expression

\[
\text{if [ condition ] expr}
\]

causes `condition` to be evaluated; if its value is non-zero, `expr` is also evaluated. It is thus equivalent to the `&&` operator (see section B.2.10), except that its return value is undefined. The expression

\[
\text{test [ condition ] expr .else expr'}
\]

behaves similarly, but additionally evaluates `expr'` if the value of `condition` is zero, and returns the value of whichever expression it evaluated.

The switch statement allows multi-way branches:

\[
\text{switch [ condition ] sequence}
\]

`sequence` may be any expression, but should be an SE containing `.case` expressions for the switch to have any effect. `condition` is evaluated and control is transferred to the expression following the `.case` whose constant argument matches the value returned by `condition`. If none of the cases match, control is transferred to the expression following the `.default` in the SE; if there is none, the SE is skipped. Having more than one default, or more than one case with the same argument, in the same switch block is an error.

The `.switch`, `.case` and `.default` expressions return undefined values.

### B.2.7 Loops

The expression

\[
\text{while [ condition ] body}
\]

is a while-loop: `condition` is evaluated and, if it is non-zero, `body` is also evaluated and the whole process repeated. The do-loop

\[
\text{do body [ condition ]}
\]

is equivalent to

\(^1\)Thus `.defun foo [] 1 .defun foo [] .return 1 .defun foo [] { .result 1 ; }` and `.defun foo [] { .return 1 ; }` are equivalent, but `.defun foo [] .result 1` is erroneous, since the procedure body is not an SE.
and the for-loop,

\[
\text{.for [init; cond; inc] body}
\]

is equivalent to

\[
\{ \text{init; .while [cond] } \{ \text{body; inc} \} \}
\]

with the exception that the implied SEs have no effect as far as scoping and .result expressions are concerned.

The values returned by loop expressions are undefined.

### B.2.8 Labels and Jumps

A label definition expression (.place name) is only meaningful as a component of an SE: it defines a label called name whose (constant) value is the address in memory of the expression following the definition in the SE, and returns this value. The value is undefined if the definition appears anywhere else. It is an error to define two labels with the same name in the same translation unit, but labels have their own name space, so procedures and variables with the same name may be defined.

A label reference expression (.label name) may be placed at any point in the SE containing the corresponding label definition; it returns the address of the label.

A goto expression (.goto addr) transfers control to the address addr, causing the expression starting there to be executed next. addr should be the value returned by a .place or .label expression; transferring control to a label defined in a procedure other than the one containing the goto will have undefined results.

The .loop expression causes the rest of the body of the smallest enclosing loop to be skipped, so that the loop condition is checked again. The .endloop expression causes the smallest enclosing loop to be terminated immediately.

### B.2.9 Arrays

Arrays are blocks of memory words. There are four kinds of arrays in P:

**Vectors**: Vectors are defined using the .vec expression. Vectors are unique within procedure invocations, i.e., all evaluations of the same .vec expression in the same procedure invocation will return the same vector, but different .vec expressions, or the same expression in different (possibly recursive) invocations of a procedure, will return different vectors. The lifetime of a vector is that of the current invocation of the procedure in which it is defined.

**Tables**: Tables are defined with the .table expression. They are static, i.e., allocated at compile time and immutable. Their lifetime is that of the whole run of the program containing them; any number of evaluations of the same table expression, in any context, will return the same table. Tables whose contents are identical are allowed to share storage (the P compiler does this by default, unless instructed otherwise).

**Strings**: These are similar to tables, but are defined in terms of ASCII characters, rather than numeric values. Strings are enclosed in double quotes (e.g., "foo"). The characters allowed in strings are the same as those allowed in character constants (see section B.2.13), with the exception of \" and the addition of \'.\"
Heap blocks: These are defined using the \texttt{new} expression, or by calling the \texttt{malloc} or \texttt{calloc} procedures of the \texttt{stdlib} interface. A new heap block is returned every time. The lifetime of heap blocks is defined by the user, who must call the \texttt{stdlib.free} procedure or invoke the appropriate \texttt{free} expression to deallocate them.

All the expressions described above return the address of the corresponding arrays. The \texttt{vec} and \texttt{table} expressions must be followed by one or both of:

- A number, indicating how many values the array should be able to hold.
- A comma-separated list of constant expressions in square brackets, giving initial values for array entries.

An error results if there are more entries in the initialisation list than are allowed by the array size specifier; if the initialisation list has fewer elements than specified, the rest are initialised to zero. If the size specifier is absent, the size is that of the initialisation list; if the list is absent, the initial values are undefined (for vectors) or zero (for tables).

Strings consist of a sequence of characters in double quotation marks (" "). See section B.2.13 for the character values allowed.

### B.2.10 Dyadic Operators

The infix dyadic operators provided by P, and their meanings, are listed in Table B.2. A few explanatory notes follow:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>Array indexing</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>Shift to the right</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Logical shift to the left</td>
</tr>
<tr>
<td>&gt;&gt;&gt;</td>
<td>Arithmetic shift right</td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td>^</td>
<td>Bitwise XOR</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Integer division</td>
</tr>
<tr>
<td>%</td>
<td>Remainder</td>
</tr>
<tr>
<td>+</td>
<td>Addition</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
</tr>
<tr>
<td>==</td>
<td>Equality</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Inequality</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or Equal</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or Equal</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical AND</td>
</tr>
<tr>
<td>^^</td>
<td>Logical XOR</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>-&gt;</td>
<td>Assign to right operand</td>
</tr>
<tr>
<td>&lt;-</td>
<td>Assign to left operand</td>
</tr>
</tbody>
</table>
The array indexing operator \( a \{ b \} \), may be evaluated in Lmode or Rmode. In Rmode, it returns the value stored in memory \( b \) words on from location \( a \) (see section 2.6.5 for why this is aesthetically displeasing). In Lmode, it returns the address of that memory location. In both cases, its arguments are evaluated in Rmode.

The arithmetic right shift operator performs sign extension while shifting, to preserve the sign of negative quantities. It is provided because P, being typeless, cannot distinguish between signed and unsigned values.

Logical operators treat all non-zero values as true, but they (and the comparison operators) return 1 for true and 0 for false. Logical AND and OR only evaluate as many arguments as necessary (from left to right) to establish the value of the result.

Assignment operators evaluate the value to assign in Rmode and the destination of the assignment in Lmode. Which is which depends on the direction of the arrow, but only left-pointing arrows are allowed in initialisers of variable declarations.

**B.2.11 Prefix Operators**

P's prefix operators are listed in table B.3. Please note the following:

**Pre-increment and decrement:** The value of the argument is incremented or decremented by 1, and the new value is returned. This means that the argument is evaluated in both Lmode and Rmode; it must occur in an Rmode context, but should be a valid Lvalue.

**Unary plus:** This has no effect, but is present for the sake of symmetry.

**Dereference:** \( !a \) is equivalent to \( a!0 \), in both its Lmode and Rmode incarnations.

**Table B.3:** P's prefix operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>++</td>
<td>Pre-increment</td>
</tr>
<tr>
<td>--</td>
<td>Pre-decrement</td>
</tr>
<tr>
<td>-</td>
<td>Arithmetic negation</td>
</tr>
<tr>
<td>+</td>
<td>No effect</td>
</tr>
<tr>
<td>-</td>
<td>Bitwise negation</td>
</tr>
<tr>
<td>!</td>
<td>Derefence</td>
</tr>
<tr>
<td>:</td>
<td>Address of</td>
</tr>
<tr>
<td>~~</td>
<td>Logical negation</td>
</tr>
</tbody>
</table>

**Table B.4:** P's postfix operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>++</td>
<td>Post-increment</td>
</tr>
<tr>
<td>--</td>
<td>Post-decrement</td>
</tr>
</tbody>
</table>
Address-of: Only allowed in Rmode, this operator causes its argument to be evaluated in Lmode so that its address in memory is returned (the compiler must arrange for this address to exist, even if the object addressed would normally reside in a register).

B.2.12 Postfix Operators

P only provides two postfix operators (see table B.4 on the preceding page). These are similar to the pre-increment and decrement operators (see section B.2.11), but they return the old value of their argument instead of the new one.

B.2.13 Numeric Constants

P's set of notations for numeric constants is borrowed from C. It includes decimal (e.g. 1234), octal (e.g. 02322), and hexadecimal (e.g. 0x4d2) integers, and characters enclosed in single quotes (e.g. 'a'). Letters, numbers and punctuation (except for the single-quote character and backslash, which must be escaped) are allowed in character constants. There is also the full set of escape sequences, as listed in table B.5.

<table>
<thead>
<tr>
<th>Escape</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n</td>
<td>Newline</td>
</tr>
<tr>
<td>\t</td>
<td>Horizontal tab</td>
</tr>
<tr>
<td>\v</td>
<td>Vertical tab</td>
</tr>
<tr>
<td>\b</td>
<td>Backspace</td>
</tr>
<tr>
<td>\r</td>
<td>Carriage return</td>
</tr>
<tr>
<td>\f</td>
<td>Form feed</td>
</tr>
<tr>
<td>\a</td>
<td>Bell</td>
</tr>
<tr>
<td>\</td>
<td>Backslash</td>
</tr>
<tr>
<td>?</td>
<td>Question mark</td>
</tr>
<tr>
<td>'</td>
<td>Single quote</td>
</tr>
<tr>
<td>&quot;</td>
<td>Double quote</td>
</tr>
<tr>
<td>\xhh</td>
<td>Hexadecimal byte</td>
</tr>
<tr>
<td>\ooo</td>
<td>Octal byte</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>Undefined value</td>
</tr>
<tr>
<td>.min</td>
<td>Smallest representable integer</td>
</tr>
<tr>
<td>.max</td>
<td>Largest representable integer</td>
</tr>
<tr>
<td>.sml</td>
<td>Size of smallest directly accessible memory unit, in addressing units (usually bytes)</td>
</tr>
<tr>
<td>.big</td>
<td>Size of largest directly accessible memory unit</td>
</tr>
<tr>
<td>.med</td>
<td>Size of intermediate memory unit (usually the same as .big)</td>
</tr>
</tbody>
</table>
B.2.14 Symbolic Constants

P provides a number of predefined symbolic constants with numeric values. These are listed in table B.6 on the facing page.
Appendix C

The WIM3 Intermediate Code

C.1 Introduction

WIM3\(^1\) is a register-based 3-address intermediate code. It is very simple, but it retains higher-level features such as an unbounded number of virtual registers and basic block structure; thus, although its instruction set is based on a common subset of modern RISC assembly languages (most notably the MIPS and DEC Alpha instruction sets [KH92, Sit92]), its ancestry can be traced back to A-2, Grace Hopper's 3-address symbolic language for the US Navy Mark I [RH53]. The orthogonal design of WIM3 makes it easy to work with while retaining a significant amount of generality.

C.2 Basic Blocks and Insns

WIM3 code is essentially a low-level representation of the program's flowgraph. It consists of a collection of basic blocks, each of which contains a sequence of instructions, or insns\(^2\). Jumps are not allowed except at the end of basic blocks, where a jump or return from procedure is mandatory. The order in which blocks appear in the final executable image is not specified. Conditional jumps have two possible destinations but since indirect jumps to register are allowed, basic blocks may have any number of successors (or predecessors, for that matter).

Each block belongs to a procedure; each procedure has a unique entry block, where its execution begins. We refer to the sets of all procedures and all basic blocks as Procs and Blocks respectively; the set of basic blocks corresponding to procedure entrypoints is $\text{EntryB}$, and $\text{FIRST}_B(p)$ is the entry block of procedure $p$. For any block $b$, we will denote the set of its predecessors as $\text{PRED}_B(b)$ and its set of successors as $\text{SUCC}_B(b)$.

We adopt a similar notation for instructions: $\text{Insns}$ is the set of all insns, and $\text{Insns}(p) \subseteq \text{Insns}$ the set of insns contained in block or procedure $p$, and $\text{Block}(i)$ the block containing insn $i$. $\text{FIRST}_I(b)$, $\text{LAST}_I(b) \in \text{Insns}(b)$ are the first and last insns in $b$ respectively. $\text{PRED}_I(i)$ and $\text{SUCC}_I(i)$ are the sets of predecessors and successors of insn $i \in \text{Insns}$ (these will be singleton sets within basic blocks; at block limits, they will consist of the last/first insns of all the predecessor/successor blocks respectively).

---

\(^1\) Named after Wim Mertens, a minimalist composer.

\(^2\) A term borrowed from the RTL intermediate language of gcc.
We also define, for each insn \(i\), sets \(\text{Ref}_V(i)\) and \(\text{Def}_V(i)\) corresponding to the vregs referenced and defined, respectively, by \(i\).

### C.3 Insn Operand Types

WIM3 insns have 3 operands (not all of which need be used). Each operand has a type and an associated value. The types available are shown in table C.1. (The Prefix column gives the prefixes of operands of the corresponding type in WIM3 code listings produced by the compiler.) A few points to note about WIM3 argument types are:

- Real CPU registers (\(hregs\)) and virtual registers (\(vregs\)) holding compiler-generated temporaries have the same type, but may be distinguished by their register numbers if necessary. This allows the target architecture’s procedure calling standard to be followed even before register assignment. In code listings, physical register numbers are prefixed by ‘$’, virtual registers corresponding to global variables by ‘g’, and all other virtuals by ‘R’.

- Register number 0 is assumed to be special: it ignores writes and always returns the number 0 when read.

- Since basic blocks may only be entered at the top, code labels are actually basic block identifiers.

- All variables (including globals) are assigned virtual registers by the front end, so any variable whose address may be taken can be identified by its virtual register.

- Dynamic vectors are allocated on the stack and their offset from the stack pointer is known at compile-time, but not before register allocation and assignment, so vectors are referred to by unique identifiers until the code generation pass. We will refer to the set of vector identifiers as \(\text{Vectors}\).

- Each P module has a collection of immutable static data, consisting of tables (read-only vectors) and character strings. Since their base addresses are not fixed until link time, we refer to them by unique identifiers from the set \(\text{Statics}\).

We refer to the value associated with the \(n\)th argument of insn \(i\) as \(\text{Arg}_V(i, n)\) and to its type as \(\text{Arg}_T(i, n)\). \(n\) takes values from the set \(\{1, 2, 3, j\}\); we use \(j\) to indicate the "false" branch destination of a conditional jump, for reasons to be explained later.

<table>
<thead>
<tr>
<th>Type</th>
<th>Prefix</th>
<th>Type description</th>
<th>Associated value</th>
</tr>
</thead>
<tbody>
<tr>
<td>reg</td>
<td>$, g, R</td>
<td>register</td>
<td>register number</td>
</tr>
<tr>
<td>con</td>
<td>#</td>
<td>compile-time numeric constant</td>
<td>constant value</td>
</tr>
<tr>
<td>bbl</td>
<td>L</td>
<td>code label</td>
<td>basic block number</td>
</tr>
<tr>
<td>str</td>
<td>s</td>
<td>static object (string or table)</td>
<td>static object number</td>
</tr>
<tr>
<td>adr</td>
<td>&amp;</td>
<td>address (Lvalue) of variable</td>
<td>register number</td>
</tr>
<tr>
<td>vec</td>
<td>v</td>
<td>stack offset of dynamic vector</td>
<td>vector number</td>
</tr>
<tr>
<td>fun</td>
<td>f</td>
<td>address of procedure</td>
<td>procedure name</td>
</tr>
</tbody>
</table>
### C.4 Insn Opcodes

The *opcode*, $\text{Op}_i$, of an insn $i$ specifies what the insn does with, or to, its arguments. There are 27 WIM3 opcodes, listed in table C.2. Some points to note are:

- Unless otherwise specified in the table, insn operands may have any type.
- Some insns (such as `add`) reference their second and third arguments and define the first. Other insns (such as `store`, store to memory) reference their arguments without altering their values. Insns of the latter category influence the state of the program through *side effects*—modification of memory contents, transfer of control, implicit assignment to registers, etc.
- In addition to the opcode and operands, each WIM3 insn contains a pointer to the source code statement that caused it to be generated. This allows filenames and line numbers to be reported when optimisation warnings are issued.
- There is no "copy register" insn; it is simulated by an addition with register 0 as the third argument.
- There is no bitwise negation insn; it is simulated by a NOR with register 0 as the third argument.
- The existence of special opcodes to access procedure arguments and spilt registers on the stack allows the compiler to defer stack layout decisions until the final code generation phase.
- Reflecting the structure of RISC jump instructions, conditional jump insns only contain one destination argument. This is technically a violation of the basic block property, since conditional jumps must be followed by unconditional jumps which only get executed if the condition fails. The compiler must treat such jump pairs atomically to avoid this; in particular, no insns should be inserted between them. We use $\text{ARG}_i(J)$ and $\text{ARGT}_i(J)$ to refer to the value and type of the "false" branch destination of a conditional jump insn $i$, thus avoiding the need to refer to its anomalous unconditional supplement.
### APPENDIX C. THE WIM3 INTERMEDIATE CODE

Table C.2: WIM3 opcodes. argn refers to the nth operand of the insn.

<table>
<thead>
<tr>
<th>Memory Access</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>load</strong></td>
<td>Load arg1 from address arg2 + arg3.</td>
<td></td>
</tr>
<tr>
<td><strong>larg</strong></td>
<td>Load procedure argument number arg2 from the stack into register arg1; arg2 is a numeric constant, arg3 is the stack pointer.</td>
<td></td>
</tr>
<tr>
<td><strong>stor</strong></td>
<td>Store arg1 to address arg2 + arg3.</td>
<td></td>
</tr>
<tr>
<td><strong>sarg</strong></td>
<td>Store arg1 on the stack as argument number arg2 for a procedure to be called later; arg2 is a numeric constant, arg3 is the stack pointer.</td>
<td></td>
</tr>
<tr>
<td><strong>spil</strong></td>
<td>Spill virtual register arg1 to the stack; after register assignment, arg3 is the real register holding arg1.</td>
<td></td>
</tr>
<tr>
<td><strong>unsp</strong></td>
<td>Reload virtual register arg1 from the stack; after register assignment, arg3 is the real register to use.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bitwise Operations</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>lsh</strong></td>
<td>Shift arg2 left by arg3 bits, put result in register arg1.</td>
<td></td>
</tr>
<tr>
<td><strong>rsh</strong></td>
<td>Shift arg2 right by arg3 bits, put result in register arg1.</td>
<td></td>
</tr>
<tr>
<td><strong>ash</strong></td>
<td>Shift and sign-extend arg2 right by arg3 bits, put result in register arg1.</td>
<td></td>
</tr>
<tr>
<td><strong>and</strong></td>
<td>Bitwise AND of arg2, arg3 in register arg1.</td>
<td></td>
</tr>
<tr>
<td><strong>orr</strong></td>
<td>Bitwise OR of arg2, arg3 in register arg1.</td>
<td></td>
</tr>
<tr>
<td><strong>nor</strong></td>
<td>Bitwise NOR of arg2, arg3 in register arg1.</td>
<td></td>
</tr>
<tr>
<td><strong>xor</strong></td>
<td>Bitwise XOR of arg2, arg3 in register arg1.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>add</strong></td>
<td>Add arg2 and arg3, put result in register arg1.</td>
<td></td>
</tr>
<tr>
<td><strong>sub</strong></td>
<td>Subtract arg3 from arg2, put result in register arg1.</td>
<td></td>
</tr>
<tr>
<td><strong>mul</strong></td>
<td>Multiply arg2 by arg3, put result in register arg1.</td>
<td></td>
</tr>
<tr>
<td><strong>div</strong></td>
<td>Divide arg2 by arg3, put result in register arg1.</td>
<td></td>
</tr>
<tr>
<td><strong>mod</strong></td>
<td>Divide arg2 by arg3, put remainder in register arg1.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Calls and Jumps</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>call</strong></td>
<td>Call a procedure at arg1 with arg2 arguments in registers and arg3 arguments in total (arg2 and arg3 are numeric constants).</td>
<td></td>
</tr>
<tr>
<td><strong>hack</strong></td>
<td>Do nothing; this insn always follows call to make life easier for liveness analysis.</td>
<td></td>
</tr>
<tr>
<td><strong>entr</strong></td>
<td>Do nothing; marks the entry to procedure arg1 which takes arg2 arguments in registers (arg2 is a numeric constant).</td>
<td></td>
</tr>
<tr>
<td><strong>retn</strong></td>
<td>Return from procedure arg1; may only appear as the last insn of a basic block.</td>
<td></td>
</tr>
<tr>
<td><strong>jeq</strong></td>
<td>Jump to arg1 (register or label) if arg2 and arg3 are equal.</td>
<td></td>
</tr>
<tr>
<td><strong>jne</strong></td>
<td>Jump to arg1 (register or label) if arg2 and arg3 are not equal.</td>
<td></td>
</tr>
<tr>
<td><strong>jlt</strong></td>
<td>Jump to arg1 (register or label) if arg2 is less than arg3.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Miscellaneous</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>slt</strong></td>
<td>Set register arg1 to one/zero if arg2 is/is not less than arg3, respectively.</td>
<td></td>
</tr>
<tr>
<td><strong>nop</strong></td>
<td>Do nothing at all.</td>
<td></td>
</tr>
</tbody>
</table>
Appendix D

P Tools Quick Reference

D.1 Introduction

The P compiler distribution includes the following programs:

- p, the P compiler.
- pi, the P library stub compiler.
- c2p, the C-to-P translator.

This user-oriented appendix contains brief descriptions of the programs and their options. Installation and configuration instructions can be found in the source distribution.

D.2 The Compiler

D.2.1 Introduction

The P compiler accepts a single source code filename on its command line and produces assembly, object or executable code from it. Its operation is governed by the run-time configuration file (if any) and options that may precede or follow the filename on the command line.

The compiler has been written with portability in mind; it will install and perform happily in any ISO C environment (although full functionality on non-Unix systems will require a very small porting effort). However, it can only produce code for the MIPS family of processors, and will require access to a MIPS assembler for the generation of executable code. The compiler and assembler may reside on different machines, as long as some filespace is shared between the two.

D.2.2 Command Line Options

The compiler accepts short options, which are preceded by a minus sign and can be bunched together according to the common Unix convention, and long options, preceded by a plus sign. The long options are:

+bitmap Produce conflict matrix graphs in XBM image format.
+davinci Produce parse graphs in a format acceptable to the davinci graph visualisation system.
+dead Show dead blocks in basic block listings.
+display Run a graph visualiser on every graph produced.
+distinct Do not attempt to reuse storage for identical read-only static objects (strings and tables).
+flowdot Produce flowgraphs in a format acceptable to the dot graph typesetter.
+flowdg Produce flowgraphs in a format acceptable to the dg graph visualisation system.
+help Display a help message and exit.
+pardot Produce parse graphs in a format acceptable to the dot graph typesetter.
+passes Specify (as a string following the option) the order in which compiler passes are executed. See the description of the PASSES entry in the run-time configuration file.
+quarter Produce quarter-size conflict matrix output.
+usedby Specify a comma-separated list of module names as the next argument; the module will be optimised assuming that its facilities are only used by the modules listed.

The following short options are accepted:
- A Produce a listing of the initial parse tree in text (default) or graphical format (as specified by the long options).
- B Produce a parse tree listing after every constant folding pass.
- C Generate a listing of the initial intermediate code in text format (default) or of the flowgraph in graphical format (as specified by long options).
- D As -C, for intermediate code after every uninitialised variable detection pass.
- E Produce a text listing of per-block value range tables after range computation.
- F Produce a listing of per-block vreg increment stacks after range computation (not described in this thesis).
- G Produce a listing of register spans after register allocation.
- H As -G, for spans after register stuffing.
- I Produce a concise listing of stuffed registers after register stuffing.
- J As -C, for intermediate code just before the register assignment pass.
- K As -G, for spans (with hregs assigned) after register assignment.
- Y Generate a listing of the conflict matrix in text (default) or graphical (see long options) form.
- Z As -C, for the final intermediate code.
Keep assembly code. The assembly file name is the source file name, with its .p extension replaced by .s. This is also necessary if the automatically invoked assembler resides on a different system.

Stop before invoking the assembler.

Stop before invoking the linker; object code is left in a file with a .o extension.

D.2.3 Configuration File

The main configuration file may be compiled into the compiler when the system is built (as a #include header file) or parsed and read at run time. The former results in a faster compiler; the latter allows radical configuration changes without the need to recompile the compiler. The parameters that may be changed from this file include:

SIZE_BIG The size of "large" data objects (those accessed by the P '!' operator) on the target machine.

SIZE_SML The size of "small" data objects on the target machine.

PCS Various parameters specifying the target architecture's Procedure Calling Standard. The first and last argument, caller-saved, callee-saved, procedure result, return address and stack pointer registers may be defined, as well as the total number of available registers and the temporary storage register used in stuffing. Two separate groups of caller- and callee-saved registers are provided to conform with MIPS conventions.

HREGBITS The size (in bits) of hard registers.

POINTERBITS The number of bits in an address, used to enable optimisations for 32-bit operating systems running on 64-bit hardware.

MAXBLOCKS The maximum number of basic blocks in a translation unit.

SMALLBLOCK The maximum size of blocks that will be duplicated to eliminate jumps.

HASHTABSZ The size of hash tables used in register allocation and value range computation.

IPATH The default search path for interface files.

APATH The default search path for annotation files.

EXTNS Various parameters specifying the standard extensions of interface, source, assembly, annotation and object files; may be changed on systems that use different conventions or do not allow dots in filenames (e.g. Acorn RISC OS).

DIRCHAR The filesystem's directory separator character (this would be '\\' on MS-DOS, '.' on RISC OS).

PASSES A string controlling the order in which compiler passes are applied. The passes are:

a Parser.

b Constant folding on the parse tree.
APPENDIX D. P TOOLS QUICK REFERENCE

c Intermediate code generation.
e Dead code elimination and jump threading.
f Liveness analysis.
g Elimination of dead computations.
h Unused variable detection.
i Uninitialised variable detection.
j Range computation and register allocation. Includes several applications of passes e-i.
l Conflict matrix construction. This is never used as the compiler does not allocate registers by colouring.
m No-op elimination.
n Limited form of jump threading.
o Register stuffing.
p Register assignment.
q Peephole optimisation.

The default PASSES string is 'abcefgijopmnq'. The PASSES mechanism does not control the back end (final code generation, assembly and linking).

D.3 The Library Stub Compiler

At the moment, there exists no dedicated P run-time library. User programs have to use the standard C library, which has two major failings as far as P programs are concerned:

1. It does not conform to P naming conventions (i.e. procedure names are not prefixed with the interface name), since C does not have an interface concept.

2. It does not define global variables containing the addresses of the procedures as required for typeless programming.

pi takes a P interface as input and prepares assembly source code declaring the appropriate global variables with the corresponding procedure addresses as initial values. Thus the interface stdio.i containing the names printf and scanf will result in an assembly code stub declaring global variables stdio.printf and stdio.scanf and assigning them the addresses of the procedures printf and scanf as initial values respectively.

The assembly stubs can be assembled and turned into a library using the system's standard tools (as and ar on Unix systems). This library is linked together with the object code, C library, and C run-time system to produce an executable program.

The same process can be used to interface any collection of C procedures to P programs, as long as an appropriate interface for them has been written.
D.4 The C to P Translator

This lex and yacc based translator is a work in progress. It works by bolting a P parse tree generator onto a C grammar and running the resulting parse tree through a P prettyprinter. At the moment, it cannot handle \texttt{typedef}, \texttt{struct} and \texttt{union} constructs. The \texttt{sizeof} operator is not implemented, and the implementation of the right shift operator does not handle signed quantities properly. Even programs that do not use any of these features have their semantics subtly altered; code that relies on the sizes of any of the types it uses will fail.
Bibliography


[Car71] Lewis Carroll. Through the Looking-Glass: and What Alice Found There. 1871.


Alonzo Church. The calculi of lambda-conversion. Annals of Mathematics Studies, 6, 1941.


[Mac86] David MacQueen. Modules for standard ML. In *Standard ML* [HMM86].


BIBLIOGRAPHY


This thesis was prepared using the \TeX\ typesetting system. The text was formatted using the \LaTeX\ macro package; the text fonts used belong to the Pandora family, and mathematical notation is set using the Euler typeface. Computer Modern Typewriter is used as the fixed-pitch font throughout. Figures were drawn using macros provided by purpose-written .sty files building on the facilities of the \Xy-pic package, except for certain graphs rendered in PostScript from automatically-generated compiler output by the dot graph layout program.