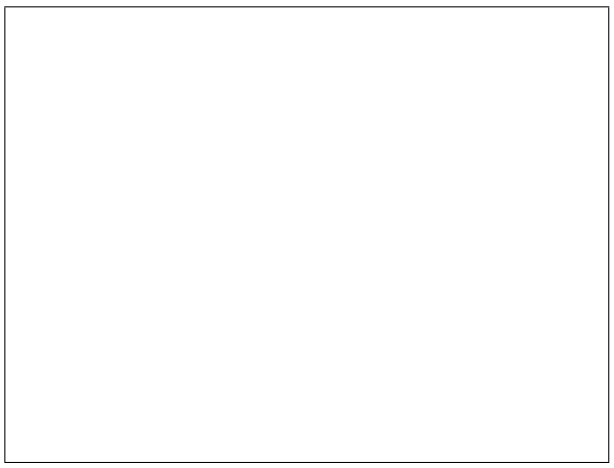
Static Single Assignment for Decompilation

by

Michael James Van Emmerik

B.E. (Hons), B.Sc (Comp. Sc.)



A thesis submitted for the degree of

Doctor of Philosophy

at

The University of Queensland

Submitted 3rd May 2007

School of Information Technology and Electrical Engineering

"Life would be much easier if I had the source code." [Jar04]

"Use the source, Luke!" [Tho96]

"*To extract the mythic essence, mere detail must become subservient to a deeper truth.*" Prospero, speaking to the Dive team in [Ega98].

The last quote could have been describing the process of decompilation: machine specific detail is discarded, leading to the essence of the program (source code), from which it is possible to divine a deeper truth (understanding of the program).

Statement of Originality

I declare that the work presented in the thesis is, to the best of my knowledge and belief, original and my own work, except as acknowledged in the text, and that the material has not been submitted, either in whole or in part, for a degree at this or any other university.

. . . . . .

(Mike Van Emmerik, candidate)

. . . . . .

(Professor Paul Bailes, principal advisor)

Acknowledgements

I would like to firstly thank my associate advisor, Cristina Cifuentes, for her inspiration, hard work finding grant money to pay me during the binary translation years (1997 to 2001), ideas, and suggestions. By extension, I'm indebted to Cristina's supervisors, Professors John Gough and Bill Caelli, who came up with the idea of using decompilation for security analysis, and as a PhD topic. My primary advisor, Prof. Paul Bailes, was also very helpful, particularly for showing me the value of writing thesis material as early as possible during candidature.

In the first years of the IBM PC and clones (early 1980s), Ron Harris and I would disassemble scores of Z80 and 8086 programs. Thanks, Ron; they were the best years! By the late 1980s, Glen McDiarmid was inspiring me (and the rest of the world) with a truly great interactive disassembler. Thanks also to Glenn Wallace, who in about 1992 started a pattern based decompiler, which we then both worked on, called "dc".

I am indebted to several people for their helpful discussions. Ilfak Guilfanov, author of IDA Pro, was one of these, and was kind enough to drive from a neighbouring country for one such discussion. Alan Mycroft's paper "Type-Based Decompilation" was an early inspiration for investigating type analysis [Myc99]. Two of Alan's students were also influential. Eben Upton pointed out to me that type analysis for aggregates is a different problem to analysis of the more elementary types. The other was Jeremy Singer, whose ideas on Static Single Information triggered many positive ideas.

The theory in this thesis has been tested on an open source decompiler platform called Boomerang. I am very grateful to Boomerang co-developer Trent Waddington, who wrote the pre-SSA data flow code, early preservation code (his "proof engine"), the idea of propagating `%flags`, early Statement class hierarchy, and much else. His code helped make it possible for me to test the theory without having to write a complete decompiler from scratch. He demonstrated what could be achieved with *ad hoc* type analysis, and also helped by participating in long discussions. I'd like to thank also those open source developers who helped test and maintain Boomerang, particularly Gerard Krol, Emmanuel Fleury and his students [BDMP05], Mike Melanson, Mike "tamlin" Nordell, and Luke "infidel" Dunstan. Thanks, guys, it's much easier to maintain a big project with more people.

Boomerang is based in part on the University of Queensland Binary Translator (UQBT), which was funded in large part by Sun Microsystems. UQBT benefited from the work of many students. In particular, I'd like to single out Doug Simon, for the early parameter identification work, and Shane Sendall, for the Semantic Specification Language and supporting code.

Email correspondents were also helpful with more general discussions about decompilation, particularly Jeremy Smith and Raimar Falke. Ian Peake showed how to implement the one-sentence summaries, while Daniel Jarrott was very helpful with LaTeX and LyX problems, and generally with discussions.

Gary Holden and Paul Renner of LEA Detection System, Inc. were the first clients to make use of Boomerang for commercial purposes. Their decompilation problem provided valuable real-world experience, and Boomerang developed significantly during this period.

I am indebted to Prof. Simon Kaplan for supporting my scholarship, even though the combined teaching and research plan did not work out in the long term.

Thanks to Jens Tröger and Trent Waddington for reviewing drafts of this thesis; I'm sure the readers will appreciate the improvements you suggested.

In any long list of acknowledgements, some names are inevitably left out. If you belong on this list and have not been mentioned, please don't feel hurt; your contributions are appreciated.

Last but by no means least, thanks to my wife Margaret and daughter Felicity for putting up with the long hours.

List of Publications

M. Van Emmerik and T. Waddington. Using a Decompiler for Real-World Source Recovery. In the *Working Conference on Reverse Engineering,* Delft, Netherlands, 9th-12th November 2004.

C. Cifuentes, M. Van Emmerik, N. Ramsey and B. Lewis. Experience in the Design, Implementation and Use of a Retargetable Static Binary Translation Framework. Sun Microsystems Laboratories, *Technical Report TR-2002-105*, January 2002.

C Cifuentes. and M. Van Emmerik. Recovery of Jump Table Case Statements from Binary Code. In *Science of Computer Programming*, 40 (2001): 171-188.

C. Cifuentes, T. Waddington, and M. Van Emmerik. Computer Security Analysis through Decompilation and High-Level Debugging. Decompilation Techniques Workshop, *Proceedings of the Eighth Working Conference on Reverse Engineering*, Stuttgart, Germany, October 2001. IEEE-CS Press, pp 375-380.

C. Cifuentes and M. Van Emmerik. UQBT: Adaptable Binary Translation at Low Cost. In *Computer* 33(3), March 2000. IEEE Computer Society Press, pp 60-66.

C. Cifuentes, M. Van Emmerik, D. Ung, D. Simon and T. Waddington. Preliminary Experiences with the Use of the UQBT Binary Translation Framework. In *Proceedings of the Workshop on Binary Translation*, NewPort Beach, Oct 16, 1999. Technical Committee on Computer Architecture Newsletter , IEEE-CS Press, Dec 1999, pp 12-22.

C. Cifuentes, M. Van Emmerik, and N. Ramsey. The Design of a Resourceable and Retargetable Binary Translator. In *Proceedings of the Sixth Working Conference on Reverse Engineering*, Atlanta, USA, October1999. IEEE-CS Press, pp 280-291.

C. Cifuentes and M. Van Emmerik. Recovery of Jump Table Case Statements from Binary Code. *Proceedings of the International Workshop on Program Comprehension*, Pittsburgh, USA, May 1999, IEEE-CS Press, pp 192-199.

M. Van Emmerik. Identifying Library Functions in Executable Files Using Patterns. In *Proceedings of the 1998 Australian Software Engineering Conference*, Adelaide, 9th to 13th November, 1998. IEEE-CS Press, pp 90-97.

Abstract

Static Single Assignment enables the efficient implementation of many important decompiler components, including expression propagation, preservation analysis, type analysis, and the analysis of indirect jumps and calls.

Source code is an essential part of all software development. It is so valuable that when it is not available, it can be worthwhile deriving it from the executable form of computer programs through the process of decompilation. There are many applications for decompiled source code, including inspections for malware, bugs, and vulnerabilities; interoperability; and the maintenance of an application that has some or all source code missing. Existing machine code decompilers, in contrast to existing decompilers for Java and similar platforms, have significant deficiencies. These include poor recovery of parameters and returns, poor handling of indirect jumps and calls, and poor to nonexistent type analysis. It is shown that use of the Static Single Assignment form (SSA form) enables many of these deficiencies to be overcome. SSA enables or assists with

- data flow analysis, particularly expression propagation;
- the identification of parameters and returns, without assuming ABI compliance;
- preservation analysis (whether a location is preserved across a call), which is needed for analysing parameters and return locations;
- type analysis, implemented as a sparse data flow problem; and
- the analysis of indirect jumps and calls.

Expression propagation is a key element of a decompiler, since it allows long sequences of individual instruction semantics to be combined into more complex, high level statements. Parameters, returns, and types are features of high level languages that do not appear explicitly in machine code programs, hence their recovery is important for readability and the ability to recompile the generated code. In addition, type analysis is either absent from existing machine code decompilers, or is limited to a relatively

simple propagation of types from library function calls. The analysis of indirect jumps and calls is important for finding all code in a machine code program, and enables the translation of important high level program elements such as switch statements, assigned gotos, virtual function calls, and calls through function pointers.

Because of these challenges, machine code decompilers are the most interesting case. Existing machine code decompilers are weak at identifying parameters and returns, particularly where parameters are passed in registers, or the calling convention is non standard. A general analysis of parameters and returns is demonstrated, using new devices such as Collectors. These analyses become more complex in the presence of recursion. The elimination of redundant parameters and returns are shown to be global analyses, implying that for a general decompiler, procedures can not be finalised until all other procedures are analysed.

Full type analysis is discussed, where the semantics of individual instructions, as well as information from library calls, contribute to the solution. A sparse, iterative, data flow based approach is compared with the more common constraint based approach. The former requires special functions to handle the multiple constraints that result from overloaded operators such as addition and subtraction. Special problems arise with aggregate types (arrays and structures), and address taking of variables.

Indirect branch instructions are often handled at instruction decode time. Delaying analysis until the program is represented in SSA form allows more powerful techniques such as expression propagation to be used. This results in a simpler, more general analysis, at the cost of having to throw away some results and restart some analyses. It is shown that this technique easily extends to handling Fortran assigned gotos, which can not be effectively analysed at decode time. The analysis of indirect call instructions has the potential for enabling the recovery of object oriented virtual function calls.

Many of the techniques presented in this thesis have been verified with the Boomerang open source decompiler. The goal of extending the state of the art of machine code decompilation has been achieved. There are of course still some areas left for future work. The most promising areas for future research have been identified as range analysis and alias analysis.

Contents

List of Figures

List of Tables

List of Algorithms

List of Abbreviations

ABI	Application Binary Interface
AMD	Advanced Micro Devices
ARA	Affine-Relation Analysis [BR04]
ARM	Advanced RISC Machines
ASI	Aggregate-Structure Identification
AST	Abstract Syntax Tree
AT&T	American Telephone and Telegraph Corporation (as: AT&T syntax vs Intel syntax)
AXP	appellation for DEC Alpha architecture
BB	Basic Block
BCPL	Basic Combined Programming Language
BSD	Berkeley Software Distribution
CCR	Condition Code Register
CDC	Class Device Context
CFG	Control Flow Graph
CISC	Complex Instruction Set Computer
CLI	Common Language Infrastructure
CLR	Common Language Runtime
CODASYL	Conference on Data Systems Languages
COINS	Compiler Infrastructure. `www.coins-project.org`
CPS	Continuation Passing Style
CPU	Central Processing Unit
CSE	Common Subexpression Elimination
CSP	Constraint Satisfaction Problem
dcc	de- C compiler
DCE	Dead Code Elimination
DEC	Digital Equipment Corporation
DFA	Data Flow Analysis
DFG	Dependence Flow Graph
DLL	Dynamically Linked Library

DOS	Disk Operating System (elsewhere also as Denial Of Service)
DML	Data Manipulation Language
DSP	Digital Signal Processing
du	definition-use (as: du-chain)
ELF	Executable and Linkable Format
EM64T	Extended Memory 64 Technology
EPIC	Explicitly Parallel Instruction Computing
FFT	Fast Fourier Transform
FLIRT	Fast Library Recognition Technique
FPGA	Field Programmable Gate Array
GCC	GNU Compiler Collection
GLB	Greatest Lower Bound
GNU	Gnu is Not Unix
GOT	Global Offset Table
GPL	General Public License
GSA	Gated Single Assignment form (rarely: GSSA)
GUI	Graphical User Interface
HLL	High Level Language
IA32	Intel Architecture, 32-bits. Also: x86
IBM	International Business Machines
ICT	Indirect Control Transfer
IDA	Interactive Disassembler (usually as IDA Pro)
ILP	Instruction-Level Parallelism
IR	Intermediate Representation (sometimes Internal Representation)
ISS	Instruction Set Simulation
JIT	Just In Time
KB	Kilo Byte (binary Kilo: 1024; sometimes called Kibi)
LHS	Left Hand Side (of an assignment)
LLVM	Low Level Virtual Machine. `llvm.org`
LNCS	Lecture Notes in Computer Science (published by Springer)
MGM	Metro-Goldwyn-Mayer
MFC	Microsoft Foundation Classes
MS-DOS	Microsoft Disk Operating System
MSIL	Microsoft Intermediate Language
MVS	Multiple Virtual Storage
NAN	Not A Number (an infinity, the result of an overflow or underflow, etc)
NOP	No Operation (machine code instruction to do nothing)
OO	Object Oriented

PC	Personal Computer
PE	Portable Executable (as: PE format)
PLM	Programming Language for Microcomputers (also as PL/M)
PPC	Power Performance Computing
PSW	Program Status Word
REC	Reverse Engineering Compiler
RET	Reverse Engineering Tool; occasionally Return (statement or instruction)
RHS	Right Hand Side (of an assignment)
RISC	Reduced Instruction Set Computer
RTL	Register Transfer Language (sometimes Register Transfer Lists)
RTTI	Runtime Type Information (sometimes Runtime Type Identification)
SC	Strong Component (also called a strongly connected component)
SCALE	Scalable Compiler for Analytical Experiments. `ali-www.cs.umass.edu/Scale`
SMC	Self Modifying Code
SPARC	Scalable Processor Architecture
SR	Status Register
SSA	Static Single Assignment
SSI	Static Single Information
SSL	Semantic Specification Language
STL	Standard Template Library
SUIF	Stanford University Intermediate Format. Also SUIF1 and SUIF2. `suif.stanford.edu`
TA	Type Analysis
ud	use-definition (as: ud-chain)
VAX	originally Virtual Address Extension
VDG	Value Dependence Graph
VFC	Virtual Function Call
VFT	Virtual Function Table (also virtual table or VT or virtual method table)
VSA	Value-Set Analysis
VT	Virtual Table (also virtual method table or VFT or virtual method table)
WSL	Wide Spectrum Language (here, a particular language by M. Ward and K. Bennett)
x64	64-bit versions of the x86 microprocessor family: AMD64 or EM64T
x86	8086 series microprocessor: 8086/80186/80286/80386/80486/Pentium/Core/AMD86. Also IA32

Glossary

There is currently vastly more literature devoted to compilers than to decompilers. As a result, certain commonly used terms such as "source code" carry a bias towards the forward engineering direction. A few terms will be used here with slightly special meanings.

- An **affine relation** is a relation where one location remains at a fixed offset or a fixed scaled offset from another location over an area of interest, usually a loop. For example, a loop index and a running array pointer are usually related this way.

- An **aggregate** is a structure (record) or array, as in the C Language Reference [IBM04]. Unions are not considered aggregates.

- The term "**argument**" is used here specifically to represent an actual argument in call statements, and the term "**parameter**" is used for formal or dummy parameters in callees.

- An **assembler** translates assembly language into object code, suitable for linking. Occasionally, the term refers to the assembler proper and linker, i.e. a translator from assembly language to machine code.

- **Assembly language** is machine specific source code with instruction mnemonics, labels, and optional comments. It is sometimes misused to mean "machine code".

- An **assignment** in a decompiler is similar to an assignment in a compiler: the value of a location (on the left of the assignment) is changed to become the result of evaluating an expression (on the right side of the assignment).

- A **basic block** is a block of statements or instructions that will execute sequentially, terminated by a branch, label, or a call.

- The term "**binary**" is used here to mean "executable", as in "Application Binary Interface" (ABI). Elsewhere it is sometimes used to mean "non text". Operators that take exactly two operands are also known as binary operators, forming binary expressions.

- The **Boomerang** decompiler is a machine code decompiler co-authored by the author of this thesis [Boo02]. It was written partly to test and demonstrate some of the ideas developed here.

- The **borrow** flag (also called a condition code bit or status register bit) is the bit used to indicate a borrow from one subtract operation to the next. In most processors, it is the same register bit as the carry flag; in some processors, the logical negation of the carry flag is the effective borrow flag.

- A call statement is said to be **bypassed** when a use is found to be defined by the call, but the call is later found to preserve the used location. (In the special case of the stack pointer, the effect of the procedure could be to have a constant value added to it.) The use is modified to become defined by the definition of the location which reaches the call.

- **Callers** are procedures that call **callees**. Callees are sometimes known as **called** procedures.

- The **carry** flag (also called a condition code bit or status register bit) is the bit used to indicate a carry (in the ordinary arithmetic sense) from one addition, shift, or rotate operation to the next. For example, after adding 250 to 10 in an 8-bit add instruction, the result is 4 with a carry (representing one unit of 256 that has to be taken into account by adding one to the high order byte of the result, if there is one). Adding 240 and 10 results in 250 with the carry flag being cleared.

- A **call graph** is the directed graph of procedures with edges directed from callers to callees. Self recursion causes cycles of length 1 in the call graph; mutual recursion causes cycles of longer length. Often, a call graph is connected, i.e. the entry point of the program reaches all procedures. Sometimes, as in a library, there will be multiple entry points, and the possibility exists that the call graph will not be connected.

- The **canonical form** of an expression is the most favoured form. For example, `m[esp`$_0$`-8]` is preferred to `m[ebp+8-12]`, where esp_0 is the value of the stack pointer register on entry to the procedure. With completely equivalent expressions such as `a+8` and `8+a`, one is arbitrarily chosen as canonical. The process of converting expressions to their canonical equivalents is **canonicalisation**. When expressions are canonicalised, they can be more readily compared for equality, and errors due to aliasing at the machine code level are reduced.

- A **childless** call is one where the callee does not yet have a data flow summary available, either because it is a recursive call, or because the call is indirect and not yet analysed. In the call graph, childless calls have no child nodes that have call summaries.

- A **collector** is an invention of this thesis, which collects either reaching definitions (a def collector), or live locations (a use collector).

- **Colocated variables** are variables that an optimising compiler has assigned the same address (memory location) or register. This is possible when the live variables of the ranges do not overlap. Such variables might have different types, necessitating their separation.

- CLI (**Common Language Infrastructure**) encompasses languages and tools that target the **CIL** (Common Infrastructure Language) . Microsoft's own implementation of this standard is the **.NET** framework, their CIL language is **MSIL** (Microsoft Intermediate Language), and their runtime engine is called the **CLR** (Common Language Runtime).

- A **compilation** is the result of running a compiler. Usually the output of a compiler is object code, however some compilers emit assembly code which has to be assembled to object code. Sometimes the term "compiler" includes the linking stage, in which case the output would be machine code. The term can therefore apply to the compiler proper, compiler and assembler, compiler and linker, or compiler, assembler and linker.

- **Condition codes** (also called **flags** or **status bits**) are a set of single bit registers that represent such conditions as the last arithmetic operation resulting in a zero result, negative result, overflow, and so on. These bits are usually collected into a **condition code register** or **flags register**. A **status register** often includes condition codes in its lower bits. Different input machines define different condition codes, though the "big four" of Zero, Sign, Overflow and Carry are usually present in some form.

- **Conditional preservation** is an analysis for determining whether a location is preserved in a procedure when that procedure is involved in recursion.

- A **constant** in the intermediate representation is a literal or immediate value. Constants have types; often the same bit pattern can represent several different things depending on the type. A constant could be any integer, floating point, string, or other fixed value, e.g. 99, 0x8048ca8, -5.3, or "hello, world". Pointers to procedures or global data objects are also constants with special types. There are also type constants, e.g. in the constraint $\mathcal{T}(x) = \alpha \,|\, \mathsf{int};$ the int is a type constant.

- Expressions representing parameters, arguments, defineds, and results have a different form in the caller **context** as opposed to the callee context. For example,

the first parameter of a procedure might take the form `m[esp`$_0$`+4]`, while in the caller context it might take the form `m[esp`$_0$`-20]`. Both expressions refer to the same physical memory word, but `esp`$_0$ (the stack pointer value on entry to the current procedure) depends on the context.

- **Continuation Passing Style** (CPS) is a style of programming with functional languages. It can also appear at the machine code level, which is more relevant here. In CPS machine code, the address of the machine code to continue with after a call is passed as a parameter to each procedure, and the procedure is jumped to rather than called. At the end of the procedure, the usual return instruction is replaced with an indirect jump to the address given by the continuation parameter.

- To **contract** a set of vertices from a graph, the set of vertices is replaced by a single vertex representing the contracted vertices. For example, nodes x, y, and z of a graph could be contracted by replacing them with a single vertex labelled $\{x, y, z\}$. The connectivity of the graph is maintained, apart from edges associated only with contracted vertices.

- A **Cygwin** program is one which calls Unix library functions, but using a special library, is able to run on a Windows operating system. A version of the GCC compiler is used to generate these executables.

- The process of **decoding** a procedure in executable, binary form is the transforming of the individual instructions to a raw intermediate representation, using the semantics of each individual instruction, instantiating operands if required.

- **Dead code** is code, usually an assignment, whose definition is unused. The elimination of dead code (dead code elimination, or DCE) does not affect the semantics of the output, and improves the readability.

- **Decompiled output** is the high level code generated by the decompiler. It is a form of "source" code, despite being an output.

- A **decompiler** could be designed to read either machine code, Java or other bytecode files, assembly language, or object code. Where not otherwise stated, a machine code decompiler is assumed. In other contexts, there are "decompilers" for Flash movies, database formats, and so on. Such usage is not considered here.

- The **defines** of a call statement are the locations it defines (modifies). Not necessarily all of these may end up being declared as results of the call.

- A **definition** is a statement that assigns a value to a location. Sometimes, as at a call, several locations can be assigned values at once. The term "definition" can also refer to a declaration of a variable in a program, e.g. in the decompiled output.

- A **definition-use** chain (also du-chain or define-use chain) is a data structure storing the uses for a particular definition of a location.

- A definition is said to **dominate** a use if for any path from the entry point to the use, the path includes the dominating definition.

- **Downcasting** is the process of converting an expression representing a pointer to an object of a base class to a pointer to an object of a derived class. The pointer thus moves down the class hierarchy. The reciprocal process is called upcasting. In some cases, downcasting or upcasting will result in the compiler emitting machine code, for example adding a constant to the pointer, which should be handled correctly by a decompiler.

- The **endianness** of an architecture is either little endian (little end first, at lowest memory address) or big endian (big end first). Data in a file will be represented in a particular endianness; if the file and architecture have opposite endianness, byte swapping will be necessary.

- The address of a local variable is said to **escape** the current procedure, if the address is passed as an argument to a procedure call, is the return expression of the current procedure, or is assigned to a global variable. When the address of a local variable escapes, it is difficult to determine when an aliased assignment is made to the local variable.

- An **executable file** is a general class of files that could be decompiled. Here, the term includes both machine code programs, and programs compiled to a virtual machine form. The term "native" distinguishes machine code executables from others, although "machine code" seems to be clearer than "native executable".

- An **expression** is as per a compiler: either a value, or a combination of an operator and (an) appropriate value or expression operand(s). Usually a function call will not be part of an expression in a decompiler, because of its initially unknown side effects.

- A **filter** is here used as an infinite set of locations to be intersected with potential parameters or returns. The effect is to remove certain types of location, such as

global variables, from sets intended to be used as parameters or returns. The filtered locations are of interest, but not as potential parameters or returns.

- The type **float** is the C floating point type, equivalent to real or shortreal types in some other languages. It is often assumed that the size of a float and the size of an integer are the same, as is true for most compilers.

- An idiom or **idiomatic** instruction **sequence** is a sequence of instructions whose overall semantics is not obvious from the semantics of the individual instructions. For example, three `xor` (exclusive or) instructions can be used to swap the values of two registers without requiring a third register or memory location.

- An **implicit definition** may be inserted into the IR of a program to provide a definition for locations such as parameters which do not have an explicit definition in the current procedure. This may be done to provide a consistent place (a location's definition) where type or other information is stored.

- An **implicit reference** may be inserted into the IR of a program to provide a use or reference to a location which is not explicitly used. An example is where the address of a memory location is passed as a parameter; there is no explicit use of the memory location. Implicit references may be required to prevent the definition of some locations from being incorrectly eliminated as dead code.

- An **induction variable** is a variable whose value is increased or decreased by a constant value in a loop. Sometimes, such variables are "induced" by indexing; for example, incrementing an index by 1 in a loop may increase the value of a temporary pointer generated by a compiler by 4.

- The **input program** is the machine code or assembly language program (etc.) that the decompiler reads. Terms such as "source program" create confusion.

- **itof** is the *i*nteger *to* *f*loat operator. For example, `itof(-5)` yields the floating point value `-5.00`. In practice, the operator may take a pair of precisions, e.g. `itof(-5, 32, 64)` to convert the 32-bit integer `5` to the 64-bit floating point value `-5.00`.

- A **Just In Time** (JIT) compiler is a piece of code that translates instructions designed for execution on a Virtual Machine (VM) to machine code on-the-fly as dictated by the current virtual program counter. Such a compiler achieves the same result as an interpreter for the VM, but usually has better performance.

- The term "**lattice**" usually refers to a lattice of types; see "Notation" on page xl.

- A **linker** is a program that combines one or more object files into an executable, machine code file.

- A location is considered **live** at a program point if defining the location at that point would affect the semantics of the program. The **liveness** of a location is the question of whether the location is live, e.g. "the liveness of x is killed by an unconditional assignment to x".

- The **live range** of a variable is the set of program points from one or more definitions to one or more uses, where the variable is live. If the program is represented in Static Single Assignment form, the live ranges associated with multiple definitions are united by ϕ-functions.

- A **local variable** is a variable located in the stack frame such that it is only visible (in scope) in the current procedure. Its address is usually of the form `sp`$\pm$`K`, where `sp` is the stack pointer register, and `K` is a constant. Stack parameters usually also have the same form, and are sometimes considered to be local variables.

- A **location** is a register or memory word which can be assigned to (it can appear on the left hand side of an assignment). It can be used as a value (it can appear on the right hand side of assignments, and in other kinds of statements). Temporaries, array elements, and structure elements are special cases of registers or variables, and are therefore also locations. A location in machine code corresponds to a variable in a high level language.

- **Machine code** (also called machine language) refers to native executable programs, i.e. instructions that could be executed by a real processor. Java bytecodes are executable but not machine code.

- When subtracting two numbers, the **minuend** is the number being subtracted from, i.e. *difference* = *minuend* - *subtrahend*.

- The **modifieds** of a procedure are the locations that are modified by that procedure. A subset of these become returns. A few special locations (e.g. the stack pointer register and program counter) which are modifieds are not considered returns.

- The term "**name**" is sometimes used in the aliasing sense, e.g. $*p$ and q can be different names (or aliases) for the same location if p points to q. In the context of the SSA form, the term is used in a different sense. Locations (including those not subject to aliasing, such as registers) are subscripted, effectively **renaming**

them or giving them different names, in order to create unique definitions for each location.

- **Object code** is the relocatable output of an assembler or compiler. It is used by some authors to mean "machine code", but is here used to refer to the contents of object files (`.o` or `.obj` files on most machines). These files contain incomplete machine code, relocation information, and enough symbolic information to be able to link the object file with others to form an executable machine code file.

- An **offset pointer** is a pointer to other than the start of a data object, obtained by adding an original pointer to an offset or displacement. Offset pointers frequently arise as a result of one or more array indexes having a non zero lower bound.

- The **original compiler** is the one presumed to have created the input executable program.

- An **original** pointer is a pointer to the start of a data object, e.g. the first element of an array (even if that represents a non zero index), or the first element of a structure.

- **Original source code** is the original, usually high level source code that the program was written in.

- An **overwriting statement** is an assignment of the form $x = f(x)$, e.g. `x = x+2` or `x = (z*x)-y`.

- **Parameters** are formal or dummy parameters in callees, as opposed to **arguments** which are actual arguments in calls.

- A **parameter filter** is a filter to remove locations, such as memory expressions, that cannot be used as a parameter to a procedure or argument to a call.

- **Pentium** refers to the Intel IA32 processor architecture. The term "x86" is more appropriate, since Pentium is no longer used by Intel for their latest IA32 processors, and other manufacturers have compatible products with different names.

- A location is **preserved** by a procedure if its value at the end of the procedure is the same as the value at the start of the procedure, regardless of the path taken in the procedure. A preserved location may be simply unused (even by procedures called by the current procedure). Most commonly it is saved near the start of the procedure and restored near the end. In rare cases, a location may be considered preserved if it always has a constant added to its initial value through

the procedure. For example, a CISC procedure where the arguments are removed by the caller might always increase the stack pointer register by 4, representing the return address popped by the return instruction (call instructions always subtract 4 from the stack pointer in those architectures).

- The term **"procedure"** is used interchangeably with the term **"function"**; whether a procedure returns a value or values or nothing at all is not known until quite late in the decompilation.

- "**Propagation**" is here used to mean expression propagation. Compilers often perform copy propagation, where only assignments of the form $x := y$ (where x and y are simple variables, not expressions) are propagated. This is because compilers usually do not want to increase complexity of expressions; they prefer simple expressions that match better with machine code instructions. Compilers sometimes propagate simple expressions such as $x+K$ where K is a constant. This is sometimes called forward substitution or forward propagation. Decompilers by contrast prefer more complex expressions, as these are generally more readable, and therefore propagate complete expressions such as $x := y+z*4$.

- **Range analysis** is an analysis that attempts to find the possible runtime values of locations, usually as a range or a set of ranges. The ranges may or may not be strided, that is, the possible values may be a minimum value plus a multiple of a stride, particularly for pointers. For function pointers or assigned goto pointers, this tends to be called value analysis.

- Some applications of decompilation require a **recompile** of the decompiled output. While some programs will not have been compiled in the first place, the term "recompiled" will still be used to indicate compiling of the source code generated by the decompiler. The ability to compile the decompiler's output is called "**recompilability**".

- The term "**record**" is sometimes used synonymously with "structure", i.e. an aggregate data structure with several members (possibly of different types).

- A procedure is said to be self **recursive** if it calls itself. It is said to be mutually recursive if it calls itself via other procedures. For example, a calls b, b calls c, and c calls a.

- **Relocation information** is information allowing the linker or loader to adjust various pointers inside object files and sometimes executable files so that they will correctly point to a data object in its final location after linking.

- In the SSA form, locations are **renamed** to effectively create unique definitions for each location. For example, two definitions of register `r8` could be renamed to $\texttt{r8}_1$ and $\texttt{r8}_2$.

- The **results** of a call statement are those defines that are used before being redefined after the call. In a decompiler, calls are treated as special assignment statements, where several locations can be the results of a call. For example, *a* is a result in *a := mycall(x, y)*; *a* and *b* are both results in *{a,b} := mycall(x, y, z)*.

- The **returns** from a procedure are the locations that are defined by that procedure and in addition are used by at least one caller before being redefined. Each return has a corresponding result in at least one call. In a high level program, usually only one value is returned, and the location (register or memory) where that value is returned is not important.

- A **return filter** is a filter to remove locations, such as global variables, that can not be used as a return location for a procedure.

- **Reverse compilation** and **decompilation** are here used interchangeably. Johnstone [JS04] prefers to use "decompilation" for the uncompiling of code produced by a compiler, and uses "reverse compilation" for the more general problem of rewriting hand-written code as high level source. Whether the original program was machine compiled or not should make little difference to a general decompiler.

- **Reverse engineering** is the process of identifying a system's components and their interrelationships, and creating representations of the system in another form or at a higher level of abstraction [CC90]. Decompilation is therefore one form of reverse engineering; there are also source to source forms such as architecture extraction. It could be argued that binary to binary translations are also reverse engineering. Popular usage and some authors consider only the process of extraction of information from a binary program.

- **Self Modifying Code** is code that makes changes to executable instructions as the program is running. Such code can be difficult to write such that the processor's instruction cache does not produce unwanted effects. It also makes the code difficult to decompile sensibly.

- A variable is said to be **shadowed** by a local variable in a procedure or nested scope if the local variable has the same name as the other variable, and the other variable would have been in the scope of the procedure. For the lifetime of the local variable, the shadowed variable is not visible to the procedure.

- The term "**signature**" has an unfortunate pair of conflicting meanings, both of which are used in decompilation. Usually, as used here, the term is a synonym for "prototype", in the sense of a declaration of a function with the names and types for its parameters, and also its return type. The other meaning is as a bit pattern used to recognise the binary form of the function as it appears statically linked into an application [VE98]. Library function recognition is not considered in detail here.

- The **signedness** of a variable or a type refers to whether the variable is signed or unsigned. This usually only applies to the integral types, sometimes including the character type. For example, the C type `char` is usually but not always regarded as signed.

- A **sink** in a directed graph is a vertex that has no outgoing edges.

- "**Source** code" is a term very firmly associated with a high level program representation, and is used here despite the fact that it is usually the output of a decompilation.

- The term "**sparse**" as applied to a data flow analysis implies a minimisation of the information stored about the program. Conventional flow-sensitive type analysis stores type information for each variable at each program point (each basic block or sometimes even each statement). By contrast, sparse type analysis stores type information for each variable, and if the program is represented in SSA form, a degree of flow sensitivity is achieved if the variable can be assumed to keep the same type from each definition to the end of its live range.

- **Static Single Assignment** (SSA) form is a program representation where each use of a location is reached statically by only one definition. When a program is represented in the SSA form, many important transformations are simplified.

- A **strongly connected component** is a maximal subgraph such that each node reaches all other nodes in the component, i.e. there is a path from every node to every other node.

- A **subarray** of a multidimensional array is the array implied by removing some indexes. For example, consider a C array declared as `int a[10][20]`. It is effectively an array of 10 subarrays, each of which are arrays of 20 integers. If `sizeof(int)` = 4, the expression `a[x][y]` is effectively a reference to the memory at address `((void*)&a) + x*80 + y*4`. The number 80 in this expression is the size of the subarray `a[x]`, which is an array of 20 integers.

- When subtracting two numbers, the **subtrahend** is the number being subtracted, i.e. *difference* = *minuend* - *subtrahend*.
- The type x is a **subtype** of type y ($x \sqsubset y$) if for every place that type y is valid, substituting x for y does not change the validity of the program.
- Locations are here labelled **suitable** if they meet the criteria of a parameter filter or a return filter.
- The terms "**target**" and "retargeting" are firmly associated with machines, and are best avoided. The decompiled output is often referred to as "source code" despite the inherent contradiction, or as "generated code".
- **Unreachable code** is code which can not be reached by any execution path of the program.
- A **use-definition** chain (also ud-chain or use-define chain) is a data structure which stores the definitions for a particular use of a location.
- A **value** is a location or a constant, i.e. an elementary item that can appear on the right hand side of an assignment, either by itself or combined with other values and operators into an expression. The leaves of expression trees are values. The term is sometimes also used to refer to one or more runtime values of a location.
- **Value analysis** is an analysis which attempts to find the set of possible runtime values of locations, e.g. function pointers. When run on ordinary variables, it tends to be called "range analysis", since the values of ordinary variables tends to be a range or set of ranges.
- The terms "**x64**" (sometimes "**x86_64**" , "**x86-64**", "**AMD64**", or "**EM64T**") refer to a modern version of the x86 processor family capable of running in 64-bit mode. Sometimes "EM64T" or "AMD64" are used to imply the slightly different Intel and AMD versions of this processor, respectively. Note that "IA64" refers to Intel's Itanium architecture, which has a completely different instruction set.
- "**x86**" is the term referring to the family of Intel and compatible processors which maintain binary instruction set compatibility to the original 8086 processor. The name came from the original numbering system of 8086 and 80186 through 80486. The term i686 is still used by some software to denote a modern member of this series, even though there has never been a processor numbered 80586 or 80686. Here, this term implies a 32-bit processor, which means at least an 80386. Also called "IA32" (Intel Architecture, 32 bits).

Notation

- **0x123C** represents the hexadecimal constant $123C_{16}$, as per the C language.
- **a[*exp*]** represents the address of *exp* in memory.
- $\texttt{loc}_n$ represents a version of the location `loc`. In the SSA form, all definitions of a location are considered distinct, so the subscript is used to differentiate them. The subscripts could be sequential integers, statement number of the defining statement, or other schemes could be used.
- **m[*exp*]** represents the memory value at address *exp*.
- $\phi(a_1, a_2)$ represents a **phi-function**. See section 4.1 for details. It should not be confused with $\emptyset$, which represents the empty or **null set**.
- $\mathcal{T}(e)$ is used to represent "the type of e" (where e is some expression). In the functional programming world, it is common to use $e :: \tau$ to say "e has the type τ", but commonly needed constraints of the type "e_1 has the same type as e_2" cannot readily be expressed with this notation. No standard has emerged: Tip et al. use [e] for the type of e [TKB03], Palsberg uses $[\![e]\!]$ [PS91], while Hendren uses T(e) [GHM00].
- Type constants such as int are printed in $\mathsf{sans\ serif\ font}$. Type variables (i.e. variables whose possible values are types) are written as lower case Greek letters such as α.
- $\top$ (top) represents the top element of a **lattice** of types or other data flow information, representing (for types) no type information. Analogously, $\bot$ (bottom) represents overconstraint, or conflicting type information. If the lattice elements are thought of as a set of possible types for a location, $\top$ represents the set of all possible types and $\bot$ the empty set. Type information progresses from the top of the lattice downwards. Authors in semantics and abstract interpretation use the opposite convention, i.e. types are only propagated *up* the lattice [App02].
- $\sqcap$ represents the **meet** operator; $a \sqcap b$ is the lattice node that is the greatest lower bound of a and b. Some authors use $\wedge$ for greater generality. A type lattice contains elements that are not comparable, hence the $\sqcap$ symbol is preferred because it implies a partial ordering.
- $\sqcup$ represents the **join** operator; $a \sqcup b$ is the lattice node that is the least upper bound of a and b. Some authors use $\vee$ for greater generality.

- $\sqsubset$ represents the **subtype** operator. It is written as $x \sqsubset y \Rightarrow X \subset Y \wedge x$, y are comparable (where X and Y are the sets of values that variables of type x and y could hold, respectively). Similarly $x \sqsubseteq y \Rightarrow X \subseteq Y \wedge x$, y are comparable.
- $\xi(\mathsf{array}(\alpha))$ indicates an **element** of an array whose elements are of type α.
- $\xi(\mathsf{structure\text{-}containing\text{-}}\alpha)$ indicates a structure **member** whose type is α.

Summary

Static Single Assignment enables the efficient implementation of many important decompiler components, including expression propagation, preservation analysis, type analysis, and the analysis of indirect jumps and calls. vii

Chapter 1

Introduction

Machine code decompilers have the ability to provide the key to software evolution: source code. Before their considerable potential can be realised, however, several problems need to be solved.

Computers are ubiquitous in modern life, and there is a considerable investment in the software needed to control them. Software requires constant modification throughout its life, to correct errors, improve security, and adapt to changing requirements. Source code, usually written in a high level language, is the key to understanding and modifying any program. Decompilers can generate source code where the original source code is not available.

Since the first compilers in the 1950s, the prospect of converting machine code into high level language has been intriguing. Machine code decompilers were first used in the 1960s to assist porting programs from one computer model to another [Hal62]. In essence, a decompiler is the inverse of a compiler, as shown in Figure 1.1.

In this figure, the compiler, assembler, and linker are lumped together for simplicity. Compilers parse source code into an intermediate representation (IR), perform various analyses, and generate machine code, as shown in the left half of Figure 1.1. Decompilers decode binary instructions and data into an IR, perform various analyses, and generate source code, as shown in the right half of the figure. Both compilers and decompilers transform the program from one form into another. Both also use an IR to represent the program, however the IR is likely to be more low level for a compiler and high level for a decompiler, reflecting the different target languages. The overall direction is from source to machine code (high level to low level) in a compiler, but from machine to source code (low level to high level) in a decompiler. Table 1.1 compares the features of source code and machine code.

Despite the complete reversal of direction, compilers and decompilers often employ

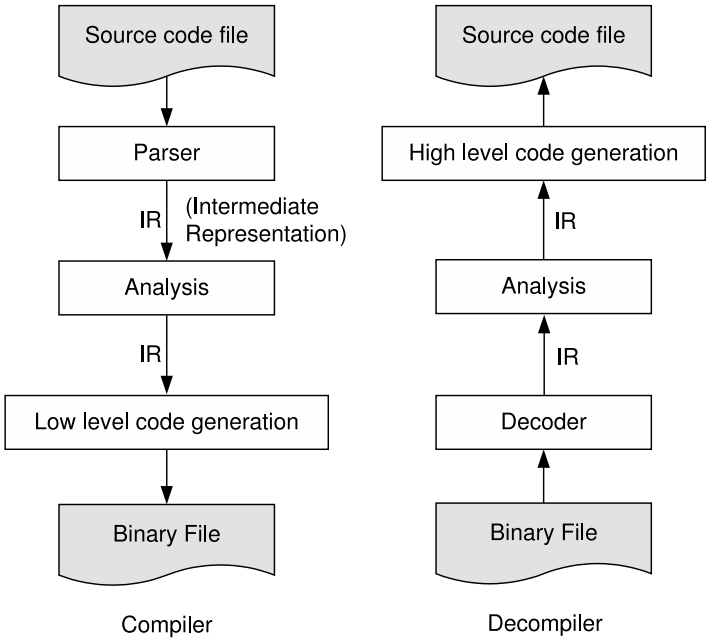


Figure 1.1: Symmetry between a compiler and a decompiler.

Table 1.1: Contrasts between source code and machine code.

Source code	Machine code
High level	Low level
Low detail	High detail
Complex expressions	Simple expressions
Machine independent	Machine dependent
More structure	Less structure
Comprehension aids	No comprehension aids

similar techniques in the analysis phases, such as data flow analysis. As an example, both compilers and decompilers need to apply algebraic identities such as $x \; or \; 0 = x$, even though a compiler may eventually emit an instruction such as `or r3,0,r5` to copy register `r3` to register `r5` if the target instruction set lacks a move instruction. Similarly, both compilers and decompilers need to know the definition(s) of a variable, whether a variable is live at a certain program point, and so on.

The compiler process of parsing the source code to an intermediate representation (IR) corresponds roughly with the decompiler process of decoding instructions into its intermediate representation. Similarly, low level code generation in a compiler corresponds roughly with high level code generation in a decompiler.

Compilation adds machine details to a program, such as which registers to use and exactly what primitive instructions to use to evaluate expressions, how to implement a loop, what address to use for a variable, and so on. Compilation removes comprehension

aids such as comments and meaningful names for procedures and variables; these aids will probably never be recovered automatically. Decompilation removes (abstracts) the machine dependent detail, and recovers information such as the nesting of loops, function parameters, and variable types from the instructions and data emitted by the compiler.

Section 1.1 shows the importance of source code, while Section 1.2 relates decompilers to other reverse engineering tools. Several of the applications for decompilation are enumerated in Section 1.3, indicating that many benefits would result from solving some of the current problems. The goal of this thesis is to solve several of these problems, in large part by using the Static Single Assignment form, as used in many optimising compilers. The current state of the art in machine code decompilation is poor, as will be shown in Section 1.4.

Section 1.5 discusses the problems faced by a variety of binary analysis tools: various existing and general classes of decompiler, and the ideal disassembler. It is found that a good decompiler has to solve about twice the number of fundamental problems of any existing binary analysis tool. Two existing machine code decompilers are included in the comparison, to show in greater detail the problems that need to be solved. Legal issues are briefly mentioned in Section 1.6, and Section 1.7 summarises the main goals of the research.

1.1 Source Code

Source code is so important to software development that at times it becomes worthwhile to derive it from the executable form of computer programs.

A program's source code specifies to a computer the precise steps required to achieve the functionality of the program. When that program is compiled, the result is an executable file of verbose, machine specific, and minute detail with the precise steps required to perform the steps of the program on the target machine. The executable version has the same essential steps, except in much greater detail. For example, the machine registers and/or memory addresses are given for each primitive step.

The source code and the machine code for the same program are equivalent, in the sense that each is a specification of how to achieve the intended functionality of the program. In other words, the program, in both source and machine code versions, conveys *how* to perform the program's function. Comments and other design documents can optionally, and to varying extents, convey *what* the program is doing, and *why*. The computer has no need for the *what* or the *why*, only the *how*.

Computers with different instruction sets require different details about how to perform the program; hence two executable files for the same program compiled for different instruction sets will in general be completely different. However, the essential components of how to perform the program are in all versions of the program: the original source code, infinitely many variants of the original source code that also perform the same functionality, and their infinitely many compilations. The task of a decompiler is essentially to find one of the variants of the original source code that is semantically equivalent to the machine code program, and hence to the original source code.

It could be argued that the output of a decompiler, typically C or C++ with generated variable names and few if any comments, is not "high level" source code. Despite this, decompiler output is easier to read than machine code, because

- source code is much more compact than any machine code representation;
- irrelevant machine details are not present, so the reader does not have to be familiar with the input machine; and
- high level (more abstract) source code features such as loops and conditionals are easier to grasp at a glance than code with compares, branches and other low level instructions.

Since the output of a decompiler is much more readable and compact than the machine code it started with, such output will be referred to as "high level language".

Source code does not necessarily imply readability, even though it is a large step in the right direction. Figure 1.2 shows the C source code for a program which is almost completely unreadable, even though it compiles on any C compiler. It is unlikely that any decompiler could produce readable source code for this program, since by design its operation is hidden. Source code comprehension (even with meaningful variable names and comments) is an active area of Computer Science.

What is desired is fully reverse engineered source code, such as shown in Figure 1.3. This was achieved using traditional reverse engineering techniques, starting with the source code of Figure 1.2. Techniques used include pretty printing (replacing white space), transforming ternary operator expressions (`?:`) into if-then-else statements, path profiling, and replacing most recursive function calls with conventional calls [Bal98]. From this it can be seen that low readability source code has an intrinsic value: it can be transformed to higher readability source code.

This example also illustrates the difference between decompilation (which might produce source code similar to that of Figure 1.2) and traditional reverse engineering (which might produce source code similar to that of Figure 1.3).

```
main(t,_,a)
char *a;
{return!0<t?t<3?main(-79,-13,a+main(-87,1-_,
main(-86, 0, a+1 )+a)):1,t<_?main(t+1, _, a ):3,main ( -94, -27+t, a
)&&t == 2 ?_<13 ?main ( 2, _+1, "%s %d %d\n" ):9:16:t<0?t<-72?main(_,
t,"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l,+,/n{n+\
,/+#n+,/#;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l q#'+d'K#!/\
+k#;q#'r}eKK#}w'r}eKK{nl]'/#;#q#n')(){}#}w'){){nl]'/+#n';d}rw' i;# ){n\
l]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#\
n'wk nw' iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c \
;;{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;\
#'rdq#w! nr'/ ') }+}{rl#'{n' ')# }'+}##(!!/")
:t<-50?_==*a ?putchar(a[31]):main(-65,_,a+1):main((*a == '/')+t,_,a\
+1 ):0<t?main ( 2, 2 , "%s"):*a=='/'||main(0,main(-61,*a, "!ek;dc \
i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-O;m .vpbks,fxntdCeghiry"),a+1);}
```

Figure 1.2: An obfuscated C program; it prints the lyrics for "The Twelve Days of Christmas" (all 64 lines of text). From [Ioc88].

```
void inner_loop(int count_day, int current_day) {
  if (count_day == FIRST_DAY) {
    print_string(ON_THE);           /* "On the " */
    /* twelve days, current_day ranges from -1 to -12 */
    print_string(-current_day);
    print_string(DAY_OF_CHRISTMAS); /* "day of Christmas ..." */
  }
  if (count_day < current_day)      /* inner iteration */
    inner_loop(count_day+1,current_day);
  /* print the gift */
  print_string(PARTRIDGE_IN_A_PEAR_TREE+(count_day-1));
}
```

Figure 1.3: Part of a traditional reverse engineering (from source code) of the Twelve Days of Christmas obfuscated program of Figure 1.2. From [Bal98].

Source code with low readability is also useful simply because it can be compiled. It could be linked with other code to keep a legacy application running, compiled for a different target machine, optimised for a particular machine, and so on. Where maintainability is required, meaningful names and comments can be added via an interactive decompiler, or in a post-decompilation process.

It could be said that source code exists at several levels:

- Well written, documented source code, written by the best human programmers. It contains well written comments and carefully chosen variable names for functions, classes, variables, and so on.

- Human written source code that, while readable, is not as well structured as the best source code. It is commented, but not as readable by programmers unfamiliar with the program. Functions and variables have names, but are not as well chosen as those in the best programs.

- Source code with few to no comments, mainly generic variable and function names, but otherwise no strange constructs. This is probably the best source code that a fully automatic decompiler could generate.

- Source code with few to no comments, mainly generic variable and function names, and occasional strange constructs such as calling an expression where the original binary program contained an indirect call instruction.

- Source code with no translation of the data section. Accesses to variables and data structures are via complex expressions such as `*(v3*4+0x8048C00)`. Most registers in the original program are represented by generic local variables such as `v3`. This is the level of source code that might be emitted by a static binary translator that emits C code, such as UQBT [CVE00].

- As above, but all original registers are visible.

- As above, but even the program counter is visible, and the program is in essence a large switch statement with one arm for the various original instructions. This is the level of source code emitted by Instruction Set Simulation techniques [MAF91].

1.2 Forward and Reverse Engineering

Decompilation is a form of reverse engineering, starting with an executable file, and ending where traditional reverse engineering starts (i.e. with source code that does not fully expose the design of a program).

Figure 1.4 shows the relationship between various engineering and reverse engineering tools and processes. Some compilers generate assembly language for an assembler, while others generate object code directly. However, the latter can be thought of as generating assembly language internally, which is then assembled to object code. Compilers for virtual machines (e.g. Java bytecode compilers) generate code that is roughly the equivalent of object code. The main tools available for manipulating machine code are:

- Disassemblers: these produce a simplified assembly language version of the program; the output may not be able to be assembled without modification. The

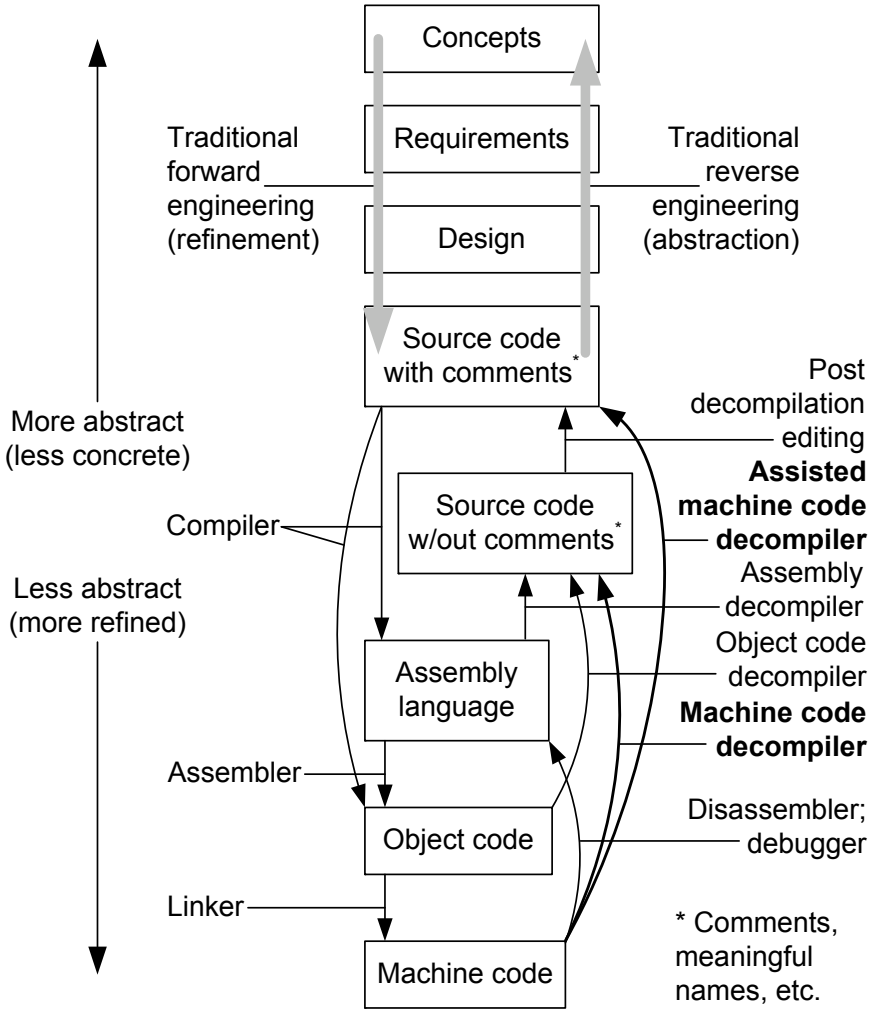


Figure 1.4: The machine code decompiler and its relationship to other tools and processes. Parts adapted from Figure 2 of [Byr92].

user requires a good knowledge of the particular machine's assembly language, and the output is voluminous for non trivial programs.

- Debuggers: conventional debuggers also operate at the simplified assembly language level, but have the advantage that the program is running, so that values for registers and memory locations can be examined to help understand the program's operation. Source level debuggers would offer greater advantages, but would require all the problems of static decompilation to be solved first.

- Decompilers: these produce a high level source code version of the program. The user does not have to understand assembly language, and the output is an order of magnitude less voluminous than that of a disassembler. The output will in general not be the same as the original source code, and may not even be in the same language. Usually, there will be few, if any, comments or meaningful variable

names, except for library function names. Some decompilers read machine code, some read object (pre-linker) code, and some read assembly language.

Clearly, a good decompiler has significant advantages over the other tools.

It is possible to imagine a tool that combines decompilation and traditional reverse engineering. For example, Ward has transformed assembly language all the way to formal specifications [War00]. However, reverse engineering from source code entails the recognition of "plans" which must be known in advance [BM00], whereas decompiling to source code from lower forms needs only to recognise such basic patterns as loops and conditionals. It therefore seems reasonable to view decompilation and traditional reverse engineering as separate problems.

1.3 Applications

Decompilers have many useful applications, broadly divided into those based on browsing parts of a program, providing the foundation for an automated tool, and those requiring the ability to compile the decompiler's output.

The first uses for decompilation were to aid migration from one machine to another, or to recover source code. As decompiler abilities have increased, software has become more complex, and the costs of software maintenance have increased, a wide range of potential applications emerge. Whether these are practical or not hinges on several key questions. The first of these is, "Is the output to be browsed, is an automated tool required, or should the output be complete enough to recompile the program?" If the output is to be recompiled, the next key question is "Is the user prepared to put significant effort into assisting the decompilation process, or will the automatically generated code be sufficient?" These questions lead to the graph of potential decompilation applications shown in Figure 1.5.

1.3.1 Decompiler as Browser

When viewed as program browsers, decompilers are useful tools that focus on parts of the input program, rather than the program as a whole.

For these applications, a decompiler is used to browse part of a program. Not all of the output code needs to be read or even generated. Browsing high level source code from a decompiler could be compared to browsing assembly language from a simple disassembler such as `objdump`.

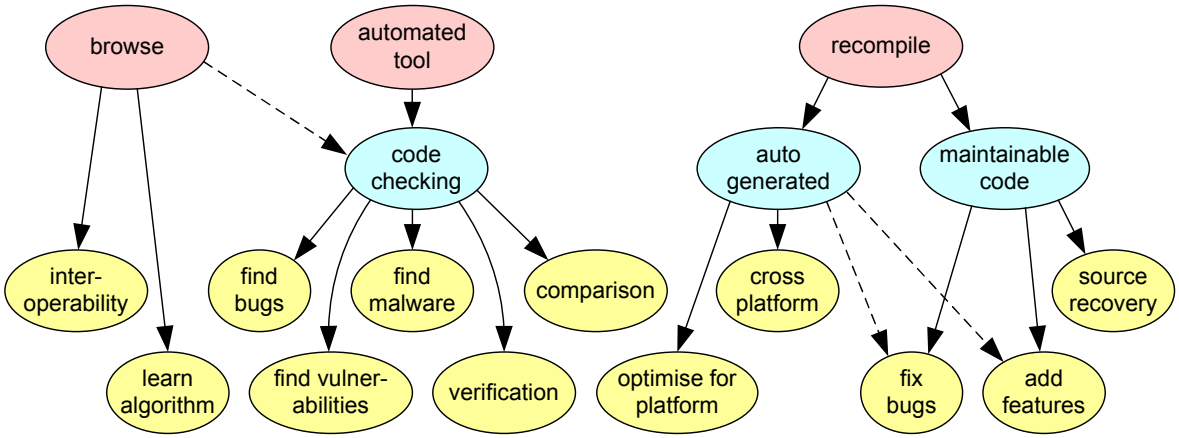


Figure 1.5: Applications of decompilation, which depend on how the output is to be used, and whether the user modifies the automatically generated output.

Interoperability

Interoperability is the process of discovering the design principles governing a program or system, for the purposes of enabling some other program or system to operate with it. These design principles are ideas, not implementations, and are therefore not able to be protected by patent or copyright. Reverse engineering techniques are lawful for such uses, as shown by Sega v. Accolade [Seg92] and again by Atari v. Nintendo [Ata92]. See also the Australian Copyright Law Review Committee report [Com95], and the IEEE position on reverse engineering [IU00].

Decompilation is a useful reverse engineering tool for interoperability where binary code is involved, since the user sees a high level description of that code.

Learning Algorithms

The copyright laws in many countries permit a person in lawful possession of a program to observe, study, or analyse the program in order to understand the ideas and principles underlying the program (adapted from [CAI01]). The core algorithm of a program occupies typically a very small fraction of the whole program. The lower volume of code to read, and the greater ease of understanding of high level code, confer an advantage to decompilation over its main alternative, disassembly.

Code Checking

Browsing the output of a decompiler could assist with the manual implementation of various forms of code checking which would normally be performed by an automatic tool, if available. This possibility is suggested by the left-most dotted line of Figure 1.1.

1.3.2 Automated Tools

Decompilation technology can be used in automated tools for finding bugs, finding vulnerabilities, finding malware, verification, and program comparison.

Tools that operate on native executable code are becoming more common, especially as the need for checking the security of applications increases.

Finding Bugs

Sometimes software users have a program which they have the rights to use, but it is not supported. This can happen if the vendor goes out of business, for example. Another scenario is where the vendor cannot reproduce a bug, and travelling to the user's premises is not practical. If a bug becomes evident under these conditions, decompilation may be the best tool to help an expert user to fix that bug. The only other alternative would be to use a combination of disassembly and a debugging tool. It will be difficult to find the bug in either case, but with the considerably smaller volume of high level code to work through, the expert with the decompiler should in general take less time than the expert working with disassembly. A tool with a decompiler integrated with a debugger would obviously be more useful than a stand alone debugger and decompiler.

The bugs could be fixed by patching the binary program, or source code fragments illustrating the bug could be sent to the vendor for maintenance. With more user effort, the bug could be fixed in the saved source code and the program is rebuilt, as described in subsection 1.3.4.

An automated tool could also search for sets of commonly found bugs, such as dereferencing pointers that could be null.

Finding Vulnerabilities

Where a security vulnerability is suspected, a large amount of code has to be checked. Again, the lower volume of output to check would in general make the decompiler a far more effective tool. It should be noted that the actual vulnerability will sometimes have to be checked at the disassembly level, since some vulnerabilities can be extremely machine specific. Others, such as buffer overflow errors, may be effectively checked using the high level language output of the decompiler. For the low level vulnerabilities, decompilation may still save considerable time by allowing the expert user to navigate quickly to those areas of the program that are most likely to harbour problems.

Finding Malware

Finding malware (behaviour not wanted by the user, e.g. viruses, keyboard logging, etc.) is similar in approach to finding vulnerabilities. Hence, the same comments apply, and the same advantages exist for decompilers.

Verification

Verification checks that the software build process has produced correct code, and that no changes have been made to the machine code file since it was created. This is important in safety critical applications [BS93].

Comparison

When enforcing software patents, or defending claims of patent infringements, it is sometimes necessary to compare the high level semantics of one piece of binary code with another piece of binary code, or its source code. If both are in binary form, these pieces of code could be expressed in machine code for different machines. Decompilation provides a way of abstracting away details of the machine language (e.g. which compiler, optimisation level, etc), so that more meaningful comparisons can be made. Some idiomatic patterns used by compilers could result in equivalent code that is very different at the machine code level, and the differences may be visible in the decompiled output. In these cases, decompilation may need to be accompanied by some source to source transformations.

1.3.3 Recompilable Code, Automatically Generated

The user may choose to accept the default, automatically generated output, which even if difficult to read still has significant uses.

Decompilers can ideally be used in round-trip reengineering: starting with a machine code program, they should be able to produce source code which can be modified, and a working program should result from the modified source code. This ideal can currently only be approached for Java and CLI (.NET) bytecode programs, but applications can be discussed ahead of the availability of machine code decompilers with recompilable output.

Automatically generated output will be low quality in the sense that there will be few meaningful identifier names, and few if any comments. This will not be easy code to

maintain, however, recompilable source code of any quality is useful for the following applications.

Optimise for Platform

Many Personal Computer (PC) applications are compiled in such a way that they will still run on an 80386 processor, even though only a few percent of programs would be running on such old hardware. Widely distributed binary programs have to be built for the expected worst case hardware. A similar situation exists on other platforms, e.g. SPARC V7 versus V9. There is a special case with Single Instruction Multiple Data (SIMD) instructions, where the advantage of utilising platform specific instructions is decisive, and vendors find that it is worth rewriting very small pieces of code several times for several different variations of instruction set. Apart from this exception, the advantages of using the latest machine instruction set are often not realised. A decompiler can realise those advantages by producing source code which is then fed into a suitable compiler, optimising for the exact processor the user has. The resulting executable will typically perform noticeably better than the distributed program.

Cross Platform

Programs written for one platform are commonly "ported" to other platforms. Obviously, source code is required, and decompilation can supply it. Two other issues arise when porting to other platforms: the issues of libraries and system dependencies in the programs being ported. The latter issue has to be handled by hand editing of the source code. The libraries required by the source program may already exist on the target platform, they may have to be rewritten, or a compatibility library such as Winelib [Win01] could be used.

With the introduction of 64-bit operating systems, another opportunity for decompilation arises. 32-bit drivers will typically not work with 64-bit operating systems. In these cases, either all drivers will have to be rewritten, or some form of compatibility layer provided. The compatibility layer, if provided, typically provides lower performance than native drivers, and of course will not provide the benefits of 64-bit addresses (i.e. being able to address more than 4GB of memory). There will be some hardware for which there is no vendor support, yet the hardware is still useful. In these cases, a 64-bit driver could be rewritten from decompiled source code for the 32-bit driver. This would allow maximum performance and provide full 64-bit addressing.

Machine code drivers are commonly the only place that low level hardware details are visible. As a result, porting drivers across operating systems when the associated

hardware is not documented may even be feasible with the aid of decompilation to expose the necessary interoperability information. Drivers also have the advantage of being relatively small pieces of code, which typically call a relatively small set of operating system functions.

Other Applications for Automatically Generated Code

Automatically generated code may be suitable for fixing bugs and adding features. More commonly, these would be performed on manually enhanced, maintainable code, as discussed below. The dotted lines near the right end of Figure 1.5 indicate this possibility.

1.3.4 Recompilable, Maintainable Code

If sufficient manual effort is put into the decompilation process, the generated code can be recompilable and maintainable.

In contrast with creating source code for the above cases, creating maintainable code requires considerable user input. The process could be compared to using a disassembler with commenting and variable renaming abilities. The user has to understand what the program is doing, enter comments and change variable names in such a way that others reading the resultant code can more easily understand the program. There is still a significant advantage over rewriting the program from scratch, however: the program is already debugged.

The result will probably not be bug free, since no significant program is. However the effort of writing and debugging a new program to the same level of bugs as the input program is often substantially larger than the effort of adding enough comments and meaningful identifiers via decompilation. (This assumes that the decompiler does not introduce more bugs than a manual rewrite.)

> *"It is easier to optimize correct code than to correct optimized code."* – Bill Harlan [Har97].

In decompilation terms, optimisation is actually the creation of the most readable representation of the program; the optimum program is the most readable one. The above quote, obviously aimed at the forward engineering process, applies equally to decompilation.

Decompilation for comprehension, including for maintenance, could be viewed as a program understanding problem where the only program documents available are end-user documents and the machine code program itself.

Fix Bugs

Once maintainable source code is available for an application, the user can fix bugs by making changes to the decompiled high level code.

It could be argued that fixing bugs is possible with automatically generated code, because the changes could be made in the absence of comments or meaningful names. However, it will be easier to effect the change and to maintain it (bug fixes need maintenance, like all code), if at least the code near the bug is well commented. This is the reason for the dotted line between "auto generated" and "fix bugs" in Figure 1.5.

Many programs ship without debugging information, and often link with third party code. Sometimes when the program fails in the field, a version of the failing program with debugging support turned on will not exhibit the fault. In addition, developers are reluctant to allow debug versions of their program into the field. Existing tools for this situation are inadequate. A high level debugger, essentially a debugger with a decompiler in place of the usual disassembler, could ease the support burden, even for companies that have the source code (except perhaps to third party code) [CWVE01].

Add Features

Similarly to the above, once maintainable source code is available, the user can add new features to the program. Also similarly to the above, it could be argued that this facility is also available to automatically generated code.

Source Code Recovery

Legacy applications, where the source code (or at least one of the source files) is missing, can be maintained by combining a decompiler with hand editing to produce maintainable source code. Variations include applications with source code but the compiler is no longer available, or is not usable on modern machines. Another variation is where there is source code, it is missing some of the desirable features of modern languages. An example of the latter is the systems language BCPL, which is essentially untyped. Sometimes there is source code, but it is known to be out of date and missing some features of a particular binary file. Finally, one or more source code files may be written in assembly language, which is machine specific and difficult to maintain.

1.4 State of the Art

The state of the art as of 2002 needed improvement in many areas, including the recovery of parameters and returns, type analysis, and the handling of indirect jumps and calls.

At the start of this research in early 2002, machine code decompilers could automatically generate source code for simple machine code programs that could be recompiled with some effort (e.g. adding appropriate `#include` statements). Statically compiled library functions could be recognised to prevent decompiling them, give them correct names, and (in principle) infer the types of parameters and the return value [VE98, Gui01]. The problem of transforming unstructured code into high level language constructs such as conditionals and loops is called structuring. This problem is largely solved; see [Cif94] Chapter 6 and [Sim97].

Unlike bytecode programs, machine code programs rarely contain the names of functions or variables, with the exception of the names of library functions and user functions dynamically invoked by name. As a result, converting decompiler output into maintainable code involves manually changing the names of functions and variables, and adding comments. At the time, few decompilers performed type recovery, so that the types for variables usually have to be manually entered as well. Enumerated types (e.g. `enum colour {red, green, blue};`) that do not interact with library functions can never be recovered automatically from machine code, since they are indistinguishable from integers.

Many pre-existing decompilers have weak analyses for indirect jump instructions, and even weaker analysis of indirect call instructions. When an indirect jump or call is not analysed, it becomes likely that the decompilation will not only fail to translate that instruction, but the whole separation of code and data could be incomplete, possibly resulting in large parts of the output being either incorrect or missing altogether.

Most decompilers attempted to identify parameters and returns of functions. However, there were often severe limitations on this identification (e.g. the decompiler might assume that parameters are always or never located in registers).

Existing decompilers can be used for a variety of real-world applications, if the user is prepared to put in significant effort to manually overcome the deficiencies of the decompiler. Sometimes circumstances would warrant such effort; see for example [VEW04]. However, such situations are rare. There would be far more applications for decompilation if the amount of effort needed to correct deficiencies in the output could be reduced. In summary, the main limitations of existing decompilers were:

- Type recovery was poor to non existent.
- There was poor translation of indirect jump instructions to switch/case statements.
- There was even poorer handling of indirect call instructions, to e.g. indirect function calls in C, or virtual functions in C++.
- The identification of parameters and returns made assumptions that were sometimes invalid.
- Some decompilers had poor performance, or failed altogether, on programs more than a few tens of kilobytes in size.
- Some decompilers produced output that resembled high level source code, but which would not compile without manual editing.
- Some decompilers produced incorrect output, e.g. through incorrect assumptions about stack pointer preservations.

Finally, meaningful names for functions and variables are not generated. Certainly, the original names are not recoverable, assuming that debug symbols are not present. Similarly, enumerated types (not associated with library functions) are not generated where a human programmer would use them. Unless some advanced artificial intelligence techniques become feasible, these aspects will never be satisfactorily generated without manual intervention.

1.5 Reverse Engineering Tool Problems

Various reverse engineering tools are compared in terms of the basic problems that they need to solve; the machine code decompiler has the largest number of such problems.

Reverse Engineering Tools (RETs) that analyse binary programs face a number of common difficulties, which will be examined in this section. Most of the problems stem from two characteristics of executable programs:

1. Some information is not present in some executable file formats, e.g. variable and procedure names, comments, parameters and returns, and types. These are losses of information, and require analyses to recover the lost information.

2. Some information is mixed together in some executable file formats, e.g. code and data; integer and pointer calculations. Pointers and offsets can be added together at compile time, resulting in a different kind of mixing which is more difficult to separate. These are losses of separation, and require analyses to make the separation, without which the representation of the input program is probably not correct.

By contrast, forward engineering tools have available all the information needed. Programming languages, including assembly languages, are designed with forward engineering in mind, e.g. most require the declaration of the names and types of variables before they are used.

There are some advantages to reverse engineering over forward engineering. For example, usually the reverse engineering tool will have the entire program available for analysis, whereas compilers often only read one module (part of a program) in isolation. The linker sees the whole program at once, but usually the linker will not perform any significant analysis. This global visibility for reverse engineering tools can potentially make data flow and alias analysis more precise and effective for reverse engineering tools than for the original compiler. This advantage has a cost: keeping the IR for the whole program consumes a lot of memory.

Another advantage of reverse engineering from an executable file is that the reverse engineering tool sees exactly what the processor will execute. At times, the compiler may make decisions that surprise the programmer [BRMT05]. As a result, security analysis tools increasingly work directly from executable files, and hopefully decompilers may be able to help in this area.

Table 1.2 shows the problems to be solved by various reverse engineering tools. The tools considered are:

- decompilers for Java and CLI/CIL bytecodes
- decompilers that read assembly language and object (pre-linker) code
- the ideal disassembler: it transforms any machine code program to reassemblable assembly language
- the ideal machine code decompiler: it transforms any machine code program to recompilable high level language
- the problems that were solved by two specific machine code decompilers, *dcc* [Cif96] and the Reverse Engineering Compiler (REC) [Cap98].

Table 1.2: Problems to be solved by various reverse engineering tools.

Problem	Machine code decompiler	*dcc* decompiler	REC decompiler	Object code decompiler	Assembly decompiler	Ideal disassembler	Java bytecode decompiler	CLI decompiler
Separate code from data	yes	some	some	some	no	yes	no	no
Separate pointers from constants	yes	no	no	easy	no	yes	no	no
Separate original from offset pointers	yes	no	no	easy	no	yes	no	no
Declare data	yes	no	no	yes	easy	yes	easy	easy
Recover parameters and returns	yes	yes	most	yes	yes	no	no	no
Analyse indirect jumps and calls	yes	no	no	yes	yes	yes	no	no
Type analysis	yes	no	no	yes	some	no	most local variables	no
Merge instructions	yes	yes	yes	yes	yes	no	yes	yes
Structure loops and conditionals	yes	yes	yes	yes	yes	no	yes	yes
Total	9	3½	3¼	6½	4½	5	2½	2

1.5.1 Separating Code from Data

Separating code from data is facilitated by data flow guided recursive traversal and the analysis of indirect jumps and calls, both of which are addressed in this thesis.

Recompilability requires the solution of several major problems. The first of these is the separation of code from data. For assembly, Java bytecode, and CLI decompilers, this problem does not exist. For the other decompilers, the problem stems from the fact that in most machines, data and code can be stored in the same memory space. The general solution to this separation has been proved to be equivalent to the halting problem [HM79]. The fact that many native executable file formats have sections with names such as .text and .data does not alter the fact that compilers and programmers often put data constants such as text strings and switch jump tables into the same section as code, and occasionally put executable code into data segments. As a result, the separation of code and data remains a significant problem.

There are a number of ways to attack this separation problem; a good survey can be found in [VWK+03]. The most powerful technique available to a static decompiler is the data flow guided recursive traversal. With this technique, one or more entry points are followed to discover all possible paths through the code. This technique relies on all paths being valid, which is unlikely to be true for obfuscated code. It also relies on the

ability to find suitable entry points, which can be a problem in itself. Finally, it relies on the ability to analyse indirect jump and call instructions using data flow analysis.

Part of the problem of separating code from data is identifying the boundaries of procedures. For programs that use standard call and return instructions this is usually straightforward. Tail call optimisations replace call and return instruction pairs with jump instructions. These can be detected easily where the jump is to the start of a function that is also the destination of a call instruction. However, some compilers such as MLton [MLt02] do not use conventional call and return instructions for the vast majority of the program. MLton uses Continuation Passing Style (CPS), which is readily converted to ordinary function calls. Decompilation of programs compiled from functional programming languages is left for future work, but it does not appear at present that finding procedure boundaries poses any particularly difficult additional problems.

1.5.2 Separating Pointers from Constants

The problem of separating pointers from constants is solved using type analysis.

The second major problem is the separation of pointers from constants. For each pointer-sized immediate operand of an instruction, a disassembler or decompiler faces the choice of representing the immediate value as a constant (integer, character, or other type), or as a pointer to some data in memory (it could point to any type of data). Since addresses are always changed between the original and the recompiled program, only those immediate values identified as pointers should change value between the original and recompiled programs.

This problem is solved using type analysis, as described in Chapter 5.

1.5.3 Separating Original from Offset Pointers

Separating original pointers from those with offsets added (offset pointers) requires range analysis.

Reverse engineering tools that read native executable files have to separate original pointers (i.e. pointers to the start of a data object) from offset pointers (e.g. pointers to the middle of arrays, or outside the array altogether). Note that if the symbol table was present, it would not help with this problem. Figure 1.6 shows a simple program illustrating the problem, in C source code and x86 machine code disassembled with IDA Pro [Dat98]. There is no need to understand much about the machine code, except to

note that the machine code uses the same constant (identified in this disassembly as str) to access the three arrays. Interested readers may refer to the x86 assembly language overview in Table 3.1 on page 63, but note that this example is in Intel assembly language style where the destination operand comes first.

```
#include <stdio.h>
int a[8] = {1, 2, 3, 4, 5, 6, 7, 8};
float b[8] =
  {1., 2., 3., 4., 5., 6., 7., 8.};
char* str = "collision!";
int main() {
  int i;
  for (i=-16; i < -8; i++)
    printf("%-3d ", (a+16)[i]);
  printf("\n");
  for (i=-8; i < 0; i++)
    printf("%.1f ", (b+8)[i]);
  printf("\n");
  printf("%s\n", str);
  return 0;
}
```

(a) Source code

```
mov eax, offset str
lea eax, [edx+eax]
mov eax, [eax]
push eax
push offset a3d  ; "%-3d "
call sub_8048324 ; printf

mov eax, offset str
lea eax, [edx+eax]
fld dword ptr [eax]
lea esp, [esp-8]
fstp [esp+24h+var_24]
push offset a_1f ; "%.1f "
call sub_8048324 ; printf

mov eax, str
push eax
call sub_8048304 ; puts
```

(b) Disassembly of the underlined source code lines. Intel syntax.

Figure 1.6: A program illustrating the problem of separating original and offset pointers.

In this program, there are two arrays which use negative indexes. Similar problems result from programs written in languages such as Pascal which support arbitrary array bounds. In this program, the symbol table was not stripped, allowing the disassembler to look up the value of the constants involved in the array fetch instructions. However, the disassembler assumes that the pointers are original, i.e. that they point to the start of a data object, in this case the string pointer `str`. The only way to find out that the first two constants are actually offset pointers is to analyse the range of possible values for the index variable `i` (register `edx` in the machine code contains `i*4`).

Note that decompiling the first two array elements as
`((int*)(str-16))[i]` and
`((float*)(str-8))[i]` respectively
would in a sense be an accurate translation of how the original program worked. It may under some circumstances even compile and run correctly, but it would be very poor code quality. It is difficult to read, but most importantly the compiler of the decompiled output is free to position the data objects of the program in any order and

with any alignment that may be required by the target architecture, probably resulting in incorrect behaviour.

Without a separation of original and offset pointers, it is not possible to determine which data object is accessed in memory expressions, so that not all definitions are known. In addition, when the address of a data object is taken, it is not known which object's address is taken, which could also lead to a definition not being identified. As will be shown in Chapter 3, underestimating definitions is unsafe.

The problems of separating pointers from constants and original pointers from offset pointers could be combined. The combined problem would be the problem of separating constants, original pointers, and offset pointers. These problems are not combined here because although they stem from the same cause (the linker adding together quantities that it is able to resolve into constants), the problems are somewhat distinct. Also, separating original from offset pointers requires an extra analysis that separating constants from pointers does not.

1.5.4 Tools Comparison

The problems faced by reverse engineering tools increases as the "abstraction distance" from source code to the input code increases; hence assembly language decompilers face relatively few problems, and machine code decompilers face the most.

The information and separation losses discussed above occur at different stages of the compilation process, as shown in Figure 1.7.

Figure 1.7 shows that no separations are lost in the actual compilation process (here excluding assembly and linking). In principle, procedure level comments could be present at the assembly language level (and even at the level of major loops, before and after procedure calls, etc.) Hence, only statement level detailed comments are lost at this stage. The entry "structured statements" indicates that several structured statements are lost, such as conditionals, loops, and multiway branch (switch or case) statements. However, the switch labels persist at the assembly language level. The original expressions from the source code are not directly present in the assembly language, however the elementary components (e.g. individual add and divide operations) are present in individual instructions. Some higher level types may still be present, e.g. structure member `Customer.address` may be accessed via the symbols `Customer` (address of the structure) and `address` (offset to the address field). An array of double precision floating point numbers starts with a label and has assembly language statements that reserve space for the array. The elementary type of the array elements is absent, however its name and size (in machine words or bytes, if not in elements) is given. This

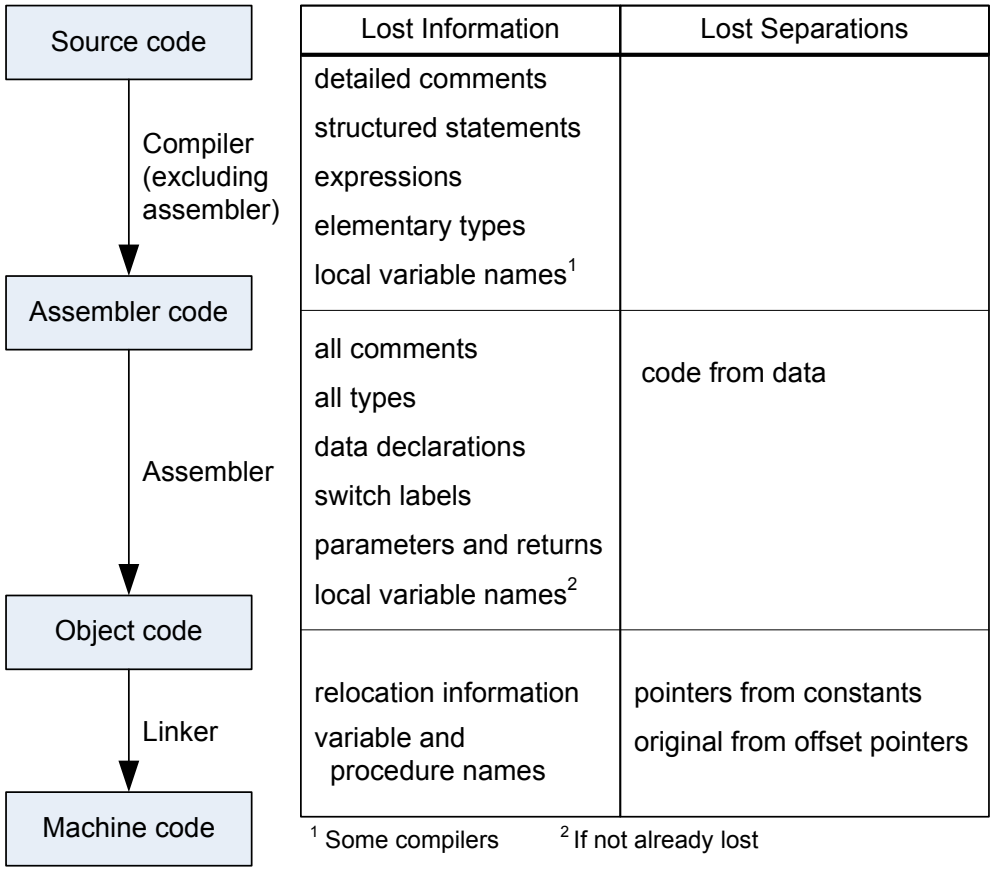


Figure 1.7: Information and separations lost at various stages in the compilation of a machine code program.

shows that decompiling from assembly language to source code is not as difficult, in terms of fundamental problems to be overcome, as decompilation or even disassembly from object or machine code.

After assembly, all comments, types, and data declarations are lost, as are switch labels. At this stage, the first separation is lost: code and data are mixed in the executable sections of the object code file. Even at the object code level, however, there are clues provided by the relocation information. For example, a table of pointers will have pointer relocations and no gaps between the relocations; this can not be executable code. However, a data structure with occasional pointers is still not readily distinguished from code. For this reason, the entry for "separate code from data" in the "object code decompiler" column is given as "some".

After linking, all variable and procedure names are lost, assuming the usual case that symbols and debug information is stripped. The very useful relocation information is also lost. It would be convenient for decompilation if processors had separate integer and pointer processing units, just as they now have separate integer and floating point units.

Since they do not, an analysis is required to separate integer constants from pointer constants. Object code provides ready clues via relocation information to make this analysis easy. From machine code, the analysis has to determine whether the result of a calculation is used as a memory address or not, and, if it is a memory pointer, whether the pointer is original or offset. These are more difficult problems that cannot be solved for all possible input programs.

1.5.4.1 Disassemblers

Machine code decompilers face four additional problems to those of an ideal disassembler; all are addressed in this thesis, or are already solved.

The problems encountered by an ideal decompiler as opposed to an ideal disassembler are:

- identifying parameters and returns, this is covered in Chapter 3;
- declaring types using type analysis, this is covered in Chapter 5;
- merging instruction semantics to form complex expressions; this is already solved by Java and CLI decompilers, and is facilitated by the data flow techniques of Chapter 3; and
- structuring loops and conditionals, which are already solved by Java and CLI decompilers.

In other words, if an ideal disassembler could be written, the techniques now exist to extend it to a full decompiler. Unfortunately, present disassemblers suffer from similar limitations to those of current decompilers.

1.5.4.2 Assembly Decompilers

Good results have been achieved for assembly language decompilers, however, they only face about half the problems of a machine code decompiler.

Some decompilers take as input assembly language programs rather than machine code; such tools are called assembly decompilers. Assembly decompilers have an easier job in several senses. An obvious advantage is the likely presence of comments and meaningful identifier names in the assembly language input. Other advantages include the nonexistence of the three separation problems of disassemblers (separating code from data, pointers from constants, and original from offset pointers), type analysis is much less

work, and declaring data is easy. Assembly language decompilers have the names and sizes of data objects, hence there is no problem separating original and offset pointers. Figure 1.8 shows the assembly language code for the underlined code of Figure 1.6(a), with the explicit labels `a`, `b`, and `str`.

```
movl $a+64, %eax
leal (%edx,%eax), %eax
movl (%eax), %eax
pushl %eax
pushl $.LC1 ; "%-3d "
call printf
```

```
movl $b+32, %eax
leal (%edx,%eax), %eax
flds (%eax)
leal -8(%esp), %esp
fstpl (%esp)
pushl $.LC2 ; "%.1f "
call printf
```

```
movl str, %eax
pushl %eax
call puts
```

Figure 1.8: Assembly language output for the underlined code of Figure 1.6(a), produced by the GCC compiler.

Assembly decompilers still have some of the major problems of decompilers as shown in Table 1.2: identifying function parameters and return values, merging instruction semantics into expressions, and structuring into high level language features such as loops. The symbols in assembly language typically make compound types such as arrays and structures explicit, but the types of the array or structure elements typically still requires type analysis, hence the entry "some" under type analysis for assembly decompilers in Table 1.2. Overall, assembly decompilers have slightly fewer problems to solve than an ideal disassembler, if each problem is assigned an equal weight.

The above considerations apply to normal assembly language, with symbols for each data element. Some generated assembly language, where the addresses of data items are forced to agree with the addresses of some other program, would be much more difficult to generate recompilable high level source code for. The assembly language output of a binary translator would be an example of such difficult assembly language. Applications for the decompilation of such programs would presumably be quite rare, and will not be considered further here.

1.5.4.3 Object Code Decompilers

Object code decompilers are intermediate in the number of problems that they face between assembly decompilers and machine code decompilers, but the existence of relocation information removes two significant separation problems.

A few decompilers take as input linkable object code (.o or .obj files). Object files are interesting in that they contain all the information contained in machine code files, plus more symbols and relocation information designed for communication with the linker. The extra information makes the separation of pointers from constants easy,

and original pointers from offsets pointers as well. Figure 1.9 shows the disassembly for the underlined code of Figure 1.6(a), starting from the object file, again with explicit references to the symbols `a`, `b`, and `str`. However, there are few circumstances under which a user would have access to object files but not the source files.

```
mov eax, (offset a+40h)
lea eax, [edx+eax]
mov eax, [eax]
push eax
push offset a3d ; "%-3d "
call printf
```

```
mov eax, (offset b+20h)
lea eax, [edx+eax]
fld dword ptr [eax]
lea esp, [esp-8]
fstp [esp+24h+var_24]
push offset a_1f ; "%.1f "
call printf
```

```
mov eax, str
push eax
call puts
```

Figure 1.9: Disassembly of the underlined code from Figure 1.6(a) starting with object code. Intel syntax. Compare with Figure 1.6(b), which started with machine code.

1.5.4.4 Virtual Machine Decompilers

Most virtual machine decompilers such as Java bytecode decompilers are very successful because their matadata-rich executable file formats ensure that they only face two problems, the solutions for which are well known.

Some decompilers take as input virtual machine executable files, e.g. Java bytecodes. Despite the fact that the Java bytecode file format was designed for programs written in Java, compilers exist which compile several languages other than Java to bytecodes (e.g. Component Pascal [QUT99] and Ada [Sof96]; for a list see [Tol96]). Like assembly decompilers, most virtual machine decompilers have an easier job than machine code decompilers, because of the extensive metadata present in the executable program files. For example, Java bytecode decompilers have the following advantages over their machine code counterparts:

- Data is already separated from code.
- Separating pointers (references) from constants is easy, since there are bytecode opcodes that deal exclusively with object references (e.g. *getfield*, *new*).
- Parameters and return values are explicit in the bytecode file.
- There is no need to decode global data, since all data are contained in classes. Class member variables can be read directly from the bytecode file.
- Type analysis is not needed for member variables and parameters, since the types are explicit in the bytecode file. However, type analysis is needed for most local

variables. In the Java Virtual machine, local variables are divided by opcode group into the broad types integer, reference, float, long, and double. There is no indication as to what subtype of integer is involved (boolean, short, or int), or what objects a reference refers to. The latter problem is the most difficult. It is also possible for a local variable slot to take on more than one type, even though this is not allowed at the Java source code level [GHM00]. This requires the different type usages of the local variable to be split into different local variables.

The following problems are common with machine code decompilers:

- Typing of most local variables;
- Merging instruction effects into expressions; and
- Structuring the code (eliminate `gotos`, generate conditionals, loops, break statements, etc).

Bytecode decompilers have one minor problem that most others do not: they need to flatten the stack oriented instructions of the Java (or CIL) Virtual Machine into the conventional instructions that real processors have. While this process is straightforward, it is sometimes necessary to create temporary variables representing the current top of stack (sometimes called stack variables). A few bytecode decompilers assume that the stack height is the same before and after any Java statement (i.e. values are never left on the stack for later use). These fail with optimised bytecode.

The advantages of virtual machine decompilers are due exclusively to the metadata present in their executable file formats. A decompiler for a hypothetical virtual machine whose executable files is not metadata rich would be faced with the same problems as a machine code decompiler.

At present, bytecode to Java decompilers are in an interesting position. The number of problems that they have to solve is small, so the barrier to writing them is comparatively small, many are available, and some perform quite well. A few of them perform type analysis of local variables. For this reason, several Java decompilers are reviewed in Section 2.4 on page 42.

1.5.4.5 Limitations of Existing Decompilers

The limitations of existing machine code decompilers include the size of the input program, identification of parameters and returns, handling of indirect jumps and calls, and type analysis.

Of the handful of machine code decompilers in existence at the start of this research in early 2002, only two had achieved any real success. These are the *dcc* decompiler [Cif96], and the Reverse Engineering Compiler (REC) [Cap98], both summarised in Section 2.1 on page 35. *Dcc* is a research compiler, written to validate the theory given in a PhD thesis [Cif94]. It decompiles only 80286 DOS programs, and emits C. REC is a non commercial decompiler recognising several machine code architectures and Binary File Formats, and produces C-like output. Considerable hand editing is required to convert REC output to compilable C code. Table 1.2 showed the problems solved by these decompilers, compared with the problems faced by various other tools, while Table 1.3 shows the limitations in more detail.

Table 1.3: Limitations for the two most capable preexisting machine code decompilers.

Problem area	dcc	REC
Handle large programs	No	Yes, but does not appear to perform much interprocedural analysis
Identification of parameters and returns	Yes, but only performed interprocedural analysis on registers and made assumptions about stack parameters	Inconsistent results
indirect jumps	IR level pattern matching; mentions need for "more idioms"	Inconsistent results
indirect calls	Assumes memory function pointer is constant; no recovery of higher level constructs	No recovery of higher level constructs
type analysis	Handles pairs of registers for long integers; no floats, arrays, structures, etc.	No floats, arrays, structures, etc.

1.5.5 Theoretical Limits and Approximation

Compilers and decompilers face theoretical limits which can be avoided with conservative approximations, but while the result for a compiler is a correct but less optimised program, the result for a decompiler ranges from a correct but less readable program to one that is incorrect.

Many operations fundamental to decompilation (such as separating code from data) are equivalent to the halting problem [HM79], and are therefore undecidable. By Rice's theorem [Ric53], all non-trivial properties of computer programs are undecidable [Cou99], hence compilers and other program-related tools are affected by theoretical limits as well.

A compiler can always avoid the worst outcome of its theoretical limitations (incorrect output) by choosing conservative behaviour. As an example, if a compiler can not prove that a variable is a constant at a particular point in a program when in reality it is, there is a simple conservative option: the constant propagation is not applied in that instance. The result is a program that is correct; the cost of the theoretical limit is that the program may run more slowly or consume more memory than if the the optimisation had been applied. Note that any particular compiler theoretical limitation can be overcome with a sufficiently powerful analysis; the theoretical limitation implies that no compiler can ever produce optimal output for *all possible* programs.

This contrasts with a decompiler which due to similar theoretical limitations cannot prove that an immediate value is the address of a procedure. The conservative behaviour in this case would be to treat the value as an integer constant, yet somehow ensure that all procedures in the decompiled program start at the same address as in the input program, or that there are jump instructions at the original addresses that redirect control flow to the decompiled procedures. If in fact the immediate value is used as a procedure pointer, correct behaviour will result, and obviously if the value was actually an integer (or used sometimes as an integer and elsewhere as a procedure pointer), no harm is done. Note that such a solution is quite drastic, compared to the small loss of performance suffered by the compiler in the previous example. To avoid this drastic measure, the decompiler will have to choose between an integer and procedure pointer type for the constant (or leave the choice to the user); if the incorrect choice is made, the output will be incorrect.

A similar situation will be discussed in Section 3.6 with respect to calls where the parameters cannot be fully analysed. A correct but less readable program can be generated by passing all live locations as parameters. In the worst case, a decompiler could fall back to binary translation techniques, whose output (if expressed in a language like C) is correct but very difficult to read. Since the original program is (presumed to be) correct, the decompiled program can always mimic the operation of the original program.

Where a decompiler chooses not to be completely conservative (e.g. to avoid highly unreadable output), it is important that it issues a comprehensive list of warnings about decisions that have been made which may prove to be incorrect.

1.6 Legal Issues

There are sufficient important and legal uses for decompilation to warrant this research, and decompilation may facilitate the transfer of facts and functional concepts to the public domain.

A decompiler is a powerful software analysis tool. Section 1.3 detailed many important and legal uses for decompilation, but there are obviously illegal uses as well. It is acknowledged that the legal implications of decompilation, and reverse engineering tools in general, are important. However, these issues are not considered here. For an Australian perspective on this subject, see [FC00] and [Cif01].

Noted copyright activist Lawrence Lessig has written:

> 'But just as gun owners who defend the legal use of guns are not endorsing cop killers, or free speech activists who attack overly broad restrictions on pornography are not thereby promoting the spread of child pornography, so too is the defence of p2p technologies not an endorsement of "piracy". ' [Les05]

The last part could be replaced with "so too is the defence of decompilation research not an endorsement of the violation of copyright or patent".

It is possible that decompilation may eventually help to attain the original goals of copyright laws. Temporary monopolies are granted to authors, to eventually advance the progress of science and art. From [Seg92]:

> "... the fact that computer programs are distributed for public use in object code form often precludes public access to the ideas and functional concepts contained in those programs, and thus confers on the copyright owner a de facto monopoly over those ideas and functional concepts. That result defeats the fundamental purpose of the Copyright Act - to encourage the production of original works by protecting the expressive elements of those works while leaving the ideas, facts, and functional concepts in the public domain for others to build on."

Unfortunately, as of 2005, the legal position of all technologies capable of copyright infringement is unclear. Prior to the MGM v Grokster, the landmark Sony Betamax case stated in effect that manufacturers of potentially infringing devices (in this case, Sony's Video Cassette Recorder) can not be held responsible for infringing users, as long as there are substantial non-infringing uses [Son84]. However, the ruling in MGM vs Grokster was:

> "We hold that one who distributes a device with the object of promoting its use to infringe copyright, as shown by clear expression or other affirmative steps taken to foster infringement, is liable for the resulting acts of infringement by third parties using the device, regardless of the device's lawful uses." [Gro05]

1.7 Goals

The main goals are better identification of parameters and returns; reconstructing types, and correctly translating indirect jumps and calls; all are facilitated by the Static Single Assignment form.

Many questions related to decompilation are undecidable. As a result, it is certainly true that not all machine code programs can successfully be decompiled. The interesting question is therefore how closely is it possible to approach an ideal machine code decompiler for real-world programs. The objective is to overcome the following limitations of existing machine code decompilers:

- Correctly and not excessively propagating expressions from individual instruction semantics into appropriate complex expressions in the decompiled output.
- Correctly identifying parameters and returns, without assuming compliance with the Application Binary Interface (ABI). This requires an analysis to determine whether a location is preserved or not by a procedure call, which can be problematic in the presence of recursion.
- Inferring types for variables, parameters, and function returns. If possible, the more complex types such as arrays, structures, and unions should be correctly inferred.
- Correctly analysing indirect jumps and calls, using the power of expression propagation and type analysis. If possible, calls to object oriented virtual functions should be recognised as such, and the output should make use of classes.

This thesis shows that the above items are all facilitated by one technique: the use of the Static Single Assignment (SSA) form. SSA is a representation commonly used in optimising compilers, but until now not in a machine code decompiler. SSA is discussed in detail in Chapter 4.

It is expected that the use of SSA in a machine code decompiler would enable at least simple programs with the following problems, not handled well by current decompilers, to be handled correctly:

- register and stack-based parameters that do not comply with the Application Binary Interface (ABI) standard,
- parameters that are preserved,
- parameters and returns should be recovered in the presence of recursion, without having to assume ABI compliance,
- basic types should be recovered,
- switch statements should be recognised despite code motion and other optimisations, and
- at least the simpler indirect call instructions should be converted to high level equivalents.

1.8 Thesis Structure

Following chapters review the limitations of existing machine code decompilers, and show how many of their limitations in the areas of data flow analysis, type analysis, and the translation of indirect jumps and calls are solved with the Static Single Assignment form.

This section gives an overview of the whole thesis. Throughout the thesis, one sentence summaries precede most sections. These can be used to obtain an overview of a part of the thesis. The index of summaries on page xliii is therefore an overview of the whole thesis.

Chapter 2 reviews the state of the art of various decompilers of machine code through to virtual machines. It reviews what has been done to date, with an emphasis on their shortcomings.

Data flow analysis is one of the most important components of a decompiler, as shown in **Chapter 3**. Data flow establishes the relationship between definitions and uses, which is important for expression propagation and eliminating condition codes from machine code. The data flow effects of call statements are particularly important; equations are defined to summarise the various elements. While traditional data flow techniques can solve the problems required by decompilers, they are complex.

Chapter 4 introduces the Static Single Assignment (SSA) representation; the use of SSA for decompilation forms the core of this thesis. SSA is a well known technique in compilation, used to perform optimisations that require data flow analysis. The advantages of SSA over other data flow implementations for the very important technique of propagation are demonstrated. A more convenient way of identifying parameters and returns is given. Also shown are the solutions to two other problems which are enabled by the use of SSA: determining whether a procedure preserves a location, and handling overlapped registers. Some problems caused by recursion are discussed. Methods for safely dealing with the inevitable imprecision of data flow information are also given.

In **Chapter 5**, a solution to the important problem of type analysis for decompilers is presented. While this problem has been implemented before as a constraint satisfaction problem, it can also be cast as a data flow problem. The use of a representation based on SSA is useful, since SSA provides a convenient pointer from every use of a location to its definition; the definition becomes a convenient place to sparsely store type information. After a brief comparison of the constraint based and data flow based implementations, details are presented for handling arrays and structures. Such compound types have not been handled properly in existing decompilers.

Chapter 6 describes techniques for analysing indirect jumps and calls. These instructions are the most difficult to resolve, and are not well handled by current decompilers. The solution is based on using the power of propagation and high level pattern matching. However, there are problems caused by the fact that before these instructions are analysed, the control flow graph is incomplete. When correctly analysed, indirect jumps can be converted to switch-like statements. Fortran style assigned gotos can also be converted to such statements. With the help of type analysis, indirect calls can be converted to class member function invocations.

Decompilation is a subject that is difficult to theorise about without actually testing the theory on actual programs. **Chapter 7** demonstrates results for various topics discussed throughout the other chapters.

Chapter 8 contains the conclusion, where the main contributions and future work are summarised.

Chapter 2

Decompiler Review

Existing decompilers have evaded many of the issues faced by machine code decompilers, or have been deficient in the areas detailed in Chapter 1. Related work has also not addressed these issues.

The history of decompilers stretches back more than 45 years. In the sections that follow, salient examples of existing decompilers and related tools and services are reviewed. Peripherally related work, such as binary translation and obfuscation, are also considered.

2.1 Machine Code Decompilers

A surprisingly large number of machine code decompilers exist, but all suffer from the problems summarised in Chapter 1.

Machine code decompilation has a surprisingly long history. Halstead [Hal62] reports that the Donnelly-Neliac (D-Neliac) decompiler was producing Neliac (an Algol-like language) code from machine code in 1960. This is only a decade after the first compilers. Cifuentes [Cif94] gives a very comprehensive history of decompilers from 1960 to 1994. The decompilation Wiki page [Dec01] reproduces this history, and includes history to the present. A brief summary of the more relevant decompilers follows; most of the early history is from Cifuentes [Cif94].

There was a difference of mindset in the early days of decompilation. For example, Stockton Gaines worries about word length (48 vs 36 was common), the representation of negative integers and floating point numbers, self modifying code, and side effects such as setting the console lights. Most of these problems have vanished with the exception of self modifying code, and even this is much less common now. Gaines also

has a good explanation of idioms [Gai65]. Early languages such as Fortran and Algol do not have pointers, which also marks a change between early and modern decompilation.

- D-Neliac decompiler, 1960 [Hal62], and Lockheed Neliac decompiler 1963-7. These produced Neliac code (an Algol-like language) from machine code programs. They could even convert non-Neliac machine code to Neliac. Decompilers at this time were pattern matching, and left more difficult cases to the programmer to perform manually [Hal70].

- C. R. Hollander's PhD thesis 1973. Hollander used a formal methodology, which while novel for the time, is essentially still pattern matching. Hollander may have been the first to use a combination of data flow and control flow techniques to improve the decompiler output [Hol73].

- The Piler System 1974. Barbe's Piler system was a first attempt to build a general decompiler. The system was able to read the machine code of several different machines, and generate code for several different high level languages. Only one input phase was written (for the GE/Honeywell 600 machine) and only two output phases were written (Fortran and COBOL) [Bar74].

- Hopwood's PhD thesis 1978. Hopwood describes a decompiler as a tool to aid porting and documentation. His experimental decompiler translated assembly language into an artificial language called MOL620, which features machine registers. This choice of target language made the decompiler easier to write, however the result is not really high level code. He also chose one instruction per node of his control flow graph, instead of the now standard basic block, so his decompiler wasted memory (relatively much more precious at that time). He was able to successfully translate one large program with some manual intervention; the resultant program was reportedly better documented than the original [Hop78].

- Exec-2-C, 1990. This was an experimental project by the company Austin Code Works, which was not completed. Intel 80286/DOS executables were disassembled, converted to an internal format, and finally converted to C. Machine features such as registers and condition codes were visible in the output. Some recovery of high level C (e.g. if-then, loops) was performed. Data flow analysis would be needed to improve this decompiler. The output was approximately three times the size of the assembly language file, when it should be more like three times smaller. It can be downloaded from [Dec01], which also contains some tests of the decompiler.

- 8086 C Decompiling System 1991-3. This work describes an Intel 8086/DOS to C decompiler . It has library function recognition (manually entered and compiler specific) to reduce unwanted output from the decompiler. It also has rule-based recognition of data types such as arrays and pointers to structures, though the papers give little detail on how this is done [FZL93, FZ91, HZY91].

- Cifuentes' PhD thesis "Reverse Compilation Techniques" 1994. Parameters and returns were identified using data flow analysis, and control flow analysis was used to structure the intermediate representation into compilable C code. Her structuring algorithm produces quite natural conditionals and loops, although in the rare case of an irreducible graph, a `goto` is generated. She demonstrated her work with a research decompiler called *dcc* [Cif96], which reads small Intel 80286/DOS programs [Cif94, CG95].
  The only types *dcc* could identify were 3 sizes of integers (8, 16, and 32 bit), and string constants (only for arguments to recognised library functions). Arrays were emitted as memory expressions (e.g. `*(arg2 + (loc3 << 1))`. *dcc* can be considered a landmark work, in that it was the first machine code decompiler to have solved the basic decompiler problems excluding the recovery of complex types.
  The *dcc* compiler was modified to read 32-bit Windows programs in 2002; see *ndcc* on page 36.

- Reverse Engineering Compiler (REC), 1998. This decompiler extends Cifuentes' work in several small ways, but the output is less readable since it generates a C-like output with registers (but not condition codes) still present. It is able to decompile executable files for several processors (e.g. Intel 386, Motorola 68K), and handles multiple executable file formats (e.g. ELF, Windows PE, etc). It is able to use debugging information in the input file, if present, to name functions and variables as per the original source code. Variable arguments to library functions such as `printf` are handled well. Complex types such as array references remain as memory expressions. Individual instruction semantics (occasionally in the form of `asm` statements) are visible in the decompiled output.
  REC is not open source software, however several binary distributions are available. The decompiler engine does not appear to have been updated after 2001, however a GUI frontend for Windows was released in 2005 [Cap98].

- The University of Queensland Binary Translator (UQBT), 1997-2001. This binary translator uses a standard C compiler as the "back end"; in other words, it emits C source code. The output is not intended to be readable, and in practice is very difficult to read. However, the output is compilable, so UQBT could be used for

optimising programs for a particular platform, or cross platform porting (Section 1.3.3). Work on UQBT was not completed, however it was capable of producing low level source code for moderate sized programs, such as the smaller SPEC [SPE95] benchmarks [CVE00, CVEU+99, UQB01].

- "Computer Security Analysis through Decompilation and High-Level Debugging", 2001. Cifuentes et al. suggested dynamic decompilation as a way to provide a powerful tool for security work. The main idea is that the security analyst is only interested in one small piece of code at one time, and so high level code could be generated "on the fly". One problem with traditional (static) decompilation is that it is difficult to determine the range of possible values of variables; by contrast, a dynamic decompiler can provide at least one value (the current value) with little effort [CWVE01].

- Type Propagation in IDA Pro Disassembler, 2001. Guilfanov describes the type propagation system in the popular disassembler IDA Pro [Dat98]. The types of parameters to library calls are captured from system header files. The parameter types for commonly used libraries are saved in files called type libraries. Assignments to parameter locations are annotated with comments with the name and type of the parameter. This type information is propagated to other parts of the disassembly, including all known callers. At present, no attempt is made to find the types for other variables not associated with the parameters of any library calls [Gui01].

- DisC, by Satish Kumar, 2001. This decompiler is designed to read only programs written in Turbo C version 2.0 or 2.01; it is an example of a compiler specific decompiler. With only one compiler generating the input programs, simpler pattern matching is more effective. DisC does not handle floating point instructions or perform type analysis, so strings are emitted as hexadecimal constants. It handles conditionals, switch statements, and loops reasonably well, although `for` loops are translated to `while` loops. It is an interesting observation that since most aspects of decompilation are ultimately pattern matching in some sense, the difference between simple pattern matching and general decompilers is essentially the generality of the patterns [Kum01b].

- *ndcc* decompiler, 2002. André Janz modified the *dcc* decompiler to read 32-bit Windows Portable Executable (PE) files. The intent was to use the modified decompiler to analyse malware. The author states that a rewrite would be needed to fully implement the 80386 instruction set. Even so, reasonable results were obtained, without overcoming *dcc*'s limitations [Jan02].

- The Anatomizer decompiler, circa 2002. K. Morisada released a decompiler for Windows 32-bit programs [Mor02]. On some Windows programs, it performs fairly well, recovering parameters, arguments, and returns. Conditionals and switch statements are handled well. When run on a Cygwin program, it failed to find arguments for a call to `printf`, possibly because the compiler translated it to a call to `Red_puts()`. Anatomizer does not appear to handle any floating point instructions or indirect calls. Arrays are left as memory expressions. Overlapping registers (e.g. `BL` and `EBX`) are treated as unrelated. No attempt was made to propagate constants (e.g. where `EBX` = 0 throughout a procedure). In some procedures where registers are used before definition, they are not identified as parameters. It is difficult to evaluate this tool much further, since no source code or English documentation is available, the program aborts on many inputs, and activity on the web site (which is in Japanese) appears to have ceased in 2005.

- "Analysis of Virtual Method Invocation for Binary Translation", 2002. Tröger and Cifuentes show a method of analysing indirect call instructions. If such a call implements a virtual method call and is correctly identified, various important aspects of the call are extracted. The technique as presented is limited to one basic block; as a result, it fails for some less common cases. The intended application is binary translation, however it could be used in a decompiler. An extension of the analysis is discussed in Section 6.3.1 [TC02].

- Boomerang, 2002-present. This is an open source decompiler, with several front ends (two are well developed) and a C back end. Boomerang has been used to demonstrate many of the techniques from this thesis. At the time of writing, it was just becoming able to handle larger binary programs. Its many limitations are mainly due to a lack of development resources. Chapter 7 contains results using this decompiler.

- Desquirr, 2002. This is an IDA Pro plug-in, written by David Eriksson as part of his Masters thesis. It decompiles one function at a time to the IDA output window. While not intended to be a serious decompiler, it illustrates what can be done with the help of a powerful disassembler and about 5000 lines of C++ code. Because a disassembler does not carry semantics for machine instructions, each supported processor requires a module to decode instruction semantics and addressing modes. The x86 and ARM processors are supported. Conditionals and loops are emitted as gotos, there is some simple switch analysis, and some identification of parameters and returns is implemented. [Eri02]

- Yadec decompiler, 2004. Raimar Falke submitted his Diploma thesis "Entwicklung

eines Typanalysesystem für einen Decompiler" (Development of a type analysis system for a decompiler) to the Technical University of Dresden. He implemented an i386 decompiler, based on an adaptation of Mycroft's theory [Myc99], and extended it to handle arrays. A file representing user choices is read to resolve conflicts [Fal04].

- Andromeda decompiler, 2004-5. This decompiler written by Andrey Shulga was never publicly released, but a GUI program for interacting with the IR generated by the decompiler proper is available from the website [And04]. The author claims that the decompiler is designed to be universal, although only an x86 front end and C/C++ back end are written at present. The GUI program comes with one demonstration IR file, and the output is extremely impressive. Unfortunately, it is not possible to analyse arbitrary programs with the available GUI program, so the correct types might for example be the result of extensive manual editing. As of March 2007, the web page has been inactive since May 2005.

- Hex-Rays decompiler for 32-bit Windows, 2007. At the time of writing, author Ilfak Guilfanov had just released a decompiler plugin for the IDA Pro Disassembler. The decompiler view shows one function at a time in a format very close to the C language. As with the assembler view, it is possible to click on a function name to jump to the view for that function. Small functions are translated in a fraction of a second. The sample code contained conditionals (including compound predicates with operators such as `||`) and loops (`for` and `while` loops, including `break` statements). Parameters and returns were present in all functions. There is an API to access the decompiler IR as a tree, allowing custom analyses to be added. The author stressed that the results are for visualisation, not for recompilation [Gui07a, Gui07b].

2.2 Object Code Decompilers

Object code decompilers have advantages over machine code decompilers, but are less common, presumably because the availability of object code without source code is low.

As noted earlier, the term "object code" is here used strictly to mean linkable, not executable, code. Object code contains relocation information and symbol names for objects that can be referenced from other object files, including functions and global variables.

- Schneider and Winger 1974. Here the contrived grammar for a compiler was inverted to produce a matching decompiler [SW74]. This works only for a particular compiler, and only under certain circumstances; it was shown in the paper that Algol 60 could not be deterministically decompiled. It generally fails in the presence of optimisations, which are now commonplace. However, this technique could be useful in safety critical applications [BS93].

- Decomp 1988. Reuter wrote a quick decompiler for a specific purpose (to port a game from one platform to another, without the original source code). Decomp was an object code decompiler, and produced files that needed significant hand editing before they could be recompiled. No data flow analysis was performed, so every function call required inspection and modification [Reu88].

2.3 Assembly Decompilers

The relatively small number of problems faced by assembly decompilers is reflected in their relatively good performance.

Some of the early decompilers (e.g. those by W. Sassaman [Sas66] and Ultrasystems [Hop78]) read assembly language, because there was a pressing need to convert assembly language programs (second generation languages) to high level languages (third generation languages). This is a somewhat easier task than machine code decompilation, as evidenced by the number of problems listed in Table 1.2 on page 18 ($4\frac{1}{2}$ versus 9).

- Housel's PhD thesis 1973. Housel describes a general decompiler as a series of mappings (transformations) with a "Final Abstract Representation" as the main intermediate representation. He uses concepts from compiler, graph, and optimisation theory. His experimental compiler was written in 5 person-months, and translated from a toy assembly language (MIXAL) to PL/1 code. Six small programs were tested, all from Knuth's *Art of Computer Programming Vol. 1* [Knu69]. Of these six, two were correct with no user intervention, and of all PL/1 statements generated, 12% needed manual intervention [Hou73, HH74].

- Friedman's PhD thesis 1974. Friedman describes a decompiler used to translate operating system minicomputer assembly language code from one machine to another. It highlighted the problems that such low level code can cause. Up to 33% of a module's assembly language required manual intervention on the first test (dissimilar machines). On a second test (similar machines) only 2% required intervention on average. The final program had almost three times the number of instructions [Fri74].

- Zebra 1981. Zebra was a prototype decompiler developed at the Naval Underwater Systems Center [Bri81]. It was another assembly decompiler, this time emitting another assembly language. The report concluded that decompilation to capture the semantics of a program was not economically practical, but that it was useful as an aid to the porting of programs.

- Simon's honours thesis "Structuring Assembly Programs", 1997. Simon proposed parenthesis theory as a higher performance alternative to interval graphs for structuring assembly language programs. He also found ways to improve the performance of interval based algorithms as used in the *dcc* decompiler. His conclusion was that with these improvements, there was no decisive advantage of one algorithm over the other; intervals produced slightly better quality, while parentheses gave slightly better performance [Sim97].

- Glasscock's diploma project "An 80x86 to C reverse compiler", 1998. This project decompiled 80x86 assembler to C. No mention is made of indirect jump or call instructions. Test programs were very simple, consisting of only integer variables, string constants used only for the printf library function, and no arrays or structures. The goal was only to decompile 5 programs, so even if-then-else conditionals were not needed. Data flow analysis was used to merge instruction semantics and eliminate unwanted assignments to flags (condition codes) [Gla98].

- Mycroft's Type-Based Decompilation, 1999. Mycroft describes an algorithm where machine instructions (in the form of Register Transfer Language, RTL) could be decompiled to C. RTL is at a slightly higher level than machine code; the author states that it is assumed that data and code are separated, and that procedure boundaries are known. It is not stated how these requirements are fulfilled; perhaps there is a manual component to preparation of the RTLs. He uses the SSA form to "undo register-colouring optimisations, whereby objects of various types, but having disjoint lifetimes, are mapped onto a single register".
  One of the drivers for Mycroft's work was a large quantity of BCPL source code that needed to be translated to something modern; BCPL is an untyped predecessor of C. He proposes the collection of constraints from the semantics of the RTL instructions, and unification to distill type information. Only registers, not memory locations, are typed. He considers code with pointers, structures, and arrays. Sometimes there is more than one solution to the constraint equations, and he suggests user intervention to choose the best solution. Mycroft describes his unification as Herbrand-based, with additional rules to repair occurs-check failures. No results are given in the paper [Myc99].

- Ward's FermaT transformation system, 1999. Ward has been working on program transformations for well over a decade. His FermaT [War01] system is capable of transforming from assembly language all the way up to specifications with some human intervention [War00]. He uses the Wide Spectrum Language WSL as his intermediate representation. Assembly language is translated to WSL; various transformations are applied to the WSL program, and finally (if needed) the WSL is translated to the output language. The FermaT transformation engine is released under the GPL license and can be downloaded from the author's web page [War01]. He also founded the company Software Migrations Ltd. In one project undertaken by this company, half a million lines of 80186 assembler code were migrated to a similar number of lines of maintainable C code. The migration cost about 10% of the estimated 65 person-months needed to manually rewrite the code [War04].

- University of London's "asm21toc'' reverse compiler, 2000. This assembly language decompiler for Digital Signal Processing (DSP) code was written in a compiler-compiler called `rdp` [JSW00]. The authors note that DSP is one of the last areas where assembly language is still commonly used. This decompiler faces problems unique to DSP processors, as noted in section 2.6.2, however decompilation from assembly language is considerably easier than from executable code. The authors "doubt the usefulness" of decompiling from binary files. See also [JS04].

- "Proof-Directed De-compilation of Low-Level Code", 2001. Katsumata and Ohori published a paper [KO01] on decompilation based on proof theoretical methods. The input is Jasmin, essentially Java assembly language. The output is an ML-like simply typed functional language. Their example shows an iterative implementation of the factorial function transformed into two functions (an equivalent recursive implementation). Their approach is to treat each instruction as a constructive proof representing its computation. While not immediately applicable to a general decompiler, their work may have application where proof of correctness (usually of a compilation) is required.
  In [Myc01], Mycroft compares his type-based decompilation with this work. Structuring to loops and conditionals is not attempted by either system. He concludes that the two systems produce very similar results in the areas where they overlap, but that they have different strengths and weaknesses.

2.4 Decompilers for Virtual Machines

Virtual machine specifications (like Java bytecodes) are rich in information such as names and types, making decompilers for these platforms much easier, however good type analysis is still necessary for recompilability.

As a group, the only truly successful decompilers to date have been those which are specific to a particular virtual machine standard (e.g. Visual Basic or Java Bytecodes). Executables designed to run on virtual machines typically are rich in information such as method names and signatures, which greatly facilitate their decompilation.

Because of the large number of Virtual Machine decompilers, only the more interesting ones will be described here. For more details, and a comparison of some decompilers, see [Dec01].

2.4.1 Java decompilers

Since Java decompilers are relatively easy to write, they first started appearing less than a year after the release of the Java language.

As indicated in Table 1.2 on page 18, the types of most local variables can only be found with type analysis. Some of the simpler Java decompilers, including some commercial decompilers, do not perform type analysis, rendering the output unsuitable for recompilation except for simple programs.

- McGill's Dava Decompiler. The Sable group at McGill University, Canada, have been developing a framework for manipulating Java bytecodes called Soot. The main purpose of Soot is optimisation of bytecodes, but they have also built a decompiler called Dava [Dec01] on top of Soot. With Dava, they have been concentrating on the more difficult aspects of bytecode decompilation. In [MH02], Miecznikowski and Hendren found that four commonly used Java decompilers were confused by peephole optimisation of the bytecode. The goal of their Dava decompiler is to be able to decompile arbitrary, verifiable bytecode to pure, compilable Java source code. Three problems have been overcome to achieve that goal: typing of local variables, generating stack variables, and structuring.
  Finding types for all local variables is more difficult than might be imagined. The process is largely a matter of solving constraints, but Gagnon et al. [GHM00] found that a three stage algorithm was needed. The first stage solves constraints; the second is needed when the types of objects created differs depending on the run-time control flow path. The third stage, not needed so far in their extensive

tests with actual bytecode, essentially types references as type Object, and introduces casts where needed.
Stack variables, in the context of Java bytecodes, arise in optimised bytecode; instead of saving such variables to locals, they are left on the stack to be used as needed. The decompiler has to create local variables in the generated source code for these stack variables, otherwise the code would not function. Optimisation of bytecodes can also result in reusing a bytecode local variable for two or more source code variables, and these may have different types. It is important for the decompiler to separate these distinct uses of one bytecode local variable into distinct local variables in the generated output.
Structuring is the process of transforming the unstructured control flow of bytecodes into readable, correct high level source code (Java in this case). For example, all gotos in the bytecode program must be converted into conditionals, loops, break or continue statements, or the like. Most decompilers do a reasonable job of this most of the time.
Miecznikowski and Hendren give examples where four other decompilers fail at all three of the above problems, and their own Dava decompiler succeeds [MH02]. Van Emmerik [Dec01] shows that one of the more recent Java decompilers (JODE, below) can also recover types well.
While Dava usually produced correct Java source code, it was often difficult to read. Some high level patterns and flow analysis were used in later work to improve the readability, even from obfuscated code [NH06]. Compound predicates (using the && and || operators) , rarely handled by decompilers, are included in the patterns.

- JODE Java Decompiler. JODE (Java optimise and decompile environment) is an open source decompiler and obfuscater/optimiser. JODE has a verifier, similar to the Java runtime verifier, that attempts to find type information from other class files. JODE is able to correctly infer types of local variables, and is able to transform code into a more readable format, closer to the way Java is naturally written, than early versions of Dava [Hoe00].

- JAD Java Decompiler. JAD is a freeware decompiler for non-commercial use, but source code is not available. Since it is written in C++, it is relatively fast [Kou99]. It is in the top three of nine decompilers tested in [Dec01], but is confused by some optimised code.

- JReversePro Java Decompiler. JReversePro is an open source Java decompiler. It seems immature compared with JODE, which is similarly open source. It fails

a number of tests that most of the other Java decompilers pass. However, it does attempt to type local variables. For example, in the Sable test from [Dec01], it correctly infers that the local variable should be declared with type `Drawable`, when most other decompilers use the most generic reference type, `Object` (and emit casts as needed) [Kum01a].

2.4.2 CLI Decompilers

MSIL is slightly easier to decompile than Java bytecodes.

The situation with Microsoft's CLI (.NET) is slightly different to that of Java. In CLI bytecode files, the types of all local variables are stored, as well as all the information that bytecode files contain. This implies that local variables can only be reused by other variables of the same type, so that no type problems arise from such sharing. No types analysis is needed. It is therefore possible to write a very good decompiler for MSIL.

- Anakrino .NET to C# Decompiler. This decompiler is released under a BSD-like license [Ana01]. As of this writing, the decompiler does not decompile classes, only methods, necessitating some hand editing for recompilation. It exited with a runtime fault for two of five tests from [Dec01].
- Reflector for .NET. Reflector is a class browser for CLI/.NET components and assemblies, with a decompiler feature. It decompiles to C#, Delphi-like and Visual Basic-like output. Full source code is available [Roe01].

2.5 Decompilation Services

A few commercial companies offer decompilation services instead of, or in addition to, selling a decompiler software license.

Despite the lack of capable, general purpose decompilers, a few companies specialise in providing decompilation services. It could be that they have a good decompiler that they are not selling to the public. It is more likely, however, that they have an imperfect decompiler that needs significant expert intervention, and they see more value to their company in selling that expertise than selling the decompiler itself and providing support for others to use it. This is similar to some other reverse engineering tools, whose results cannot be guaranteed. Only a few commercial decompilation services are summarised here; for more details, see [Dec01].

- Software Migrations Ltd [SML01]. Their services are "based on ground breaking research work first undertaken during the 1980's at the Universities of Oxford and Durham in England" (from their web page). They specialise in mainframe assembler comprehension and migration. See also the entry on the company's founder, Martin Ward, in Section 2.3 on page 40.

- The Source Recovery Company and ESTC. The Source Recovery Company [SRC96, FC99], and their marketing partner ESTC [EST01] offer a service of recovering COBOL and Assembler source code from MVS load modules. They use a low level pattern matching technique, but this seems to be suitable for the mainframe COBOL market. The service has been available for many years, and appears to be successful. It is possible that this and related services are relatively more successful than others in part because COBOL does not manipulate pointers.

- JuggerSoft [Jug05], also known as SST Global [SST03] and Source Recovery [SR02a] (not related to The Source Recovery Company above). This company sells decompilers and decompilation services, as well as source to source translators. They have a good collection of decompilers for legacy platforms. Their most recent decompiler for HP-UX produces C or C++ [SR02b]. Interestingly, this company guarantees success (provided the customer has enough money, presumably), and will write a custom decompiler if necessary. They claim that even this will be accomplished in "a fraction of the time" of rewriting.

- Dot4.com. This company offers the service of translating assembly language code to maintainable C. Some 90-95% is performed automatically using proprietary in-house tools. Comments and comment blocks are maintained from the assembly language code, and instructions are merged to produce expressions [Dot04].

- MicroAPL assembler to assembler translators. MicroAPL offer both tools and services for the translation of one assembly language to another. Several machines are supported, including 68K, DSP, 80x86, Z8K, PowerPC, and ColdFire cores [Mic97].

- Decompiler Technologies, formerly Visual Basic Right Back, offered a Visual Basic decompilation service from 2004-2006 [DT04]. A Windows C/C++ decompilation service was announced in mid 2005. Recompilability was not guaranteed; output was generated automatically using proprietary, not for sale decompilers and other tools. Manual enhancement of the output was offered as a separate service. In mid 2007, the Visual Basic decompilation service was resumed.

2.6 Related Work

This related work faces a subset of the problems of decompilation, or feature techniques or representations that appear to be useful for decompilation.

Several existing tools, analyses, or representations overlap with decompilation research. Each of these is considered below.

2.6.1 Disassembly

Disassembly achieves similar results to decompilation, and encounters some of the same problems as decompilation, but the results are more verbose and machine specific. No existing disassemblers automatically generate reassemblable output.

Disassemblers can be used to solve some of the same problems that decompilers solve. They have three major drawbacks compared to decompilers: their output is machine specific, their output is large compared to high level source code, and users require knowledge of a particular machine language. Decompilers are preferred over disassemblers, if available at the same level of functionality, for the same reasons that high level languages are preferred over assembly language. As an example, Java disassemblers are rarely used (except perhaps to debug Java compilers), because good Java decompilers are available.

The most popular disassembler is probably IDA Pro [Dat98]. It performs some automatic analysis of the input program, but also offers interactive commands to override the auto analysis results if necessary, to add comments, declare structures, etc. Separation of pointers from constants and original from offset pointers is completely manual. Ida Pro has the ability to generate assembly language in various dialects, however routine generation of assembly language for reassembly is discouraged [Rob02, Van03]. High level visualiser plugins are available, and a pseudocode view was being demonstrated at the time of writing.

2.6.2 Decompilation of DSP Assembly Language

This is a specialised area, where assembly language is still in widespread but declining use, and has several unique problems that are not considered here.

Current Digital Signal Processors (DSPs) are so complex that they rely on advanced compilers for performance. Previous generations relied heavily on assembly language for their performance. As a result, there is a need for conversion of existing assembly

language code to high level code, which can be achieved by assembly decompilation. Decompilation of DSP assembly language has extra challenges over and above the usual problems, including

- saturated arithmetic,
- sticky overflow,
- hardware support for buffers, including circular buffers, and
- more reliance on timing guarantees.

These issues are not considered here.

2.6.3 Link-Time Optimisers

Link-time Optimisers share some of the problems of machine code decompilers, but have a decided advantage because of the presence of relocation information.

Link-time optimisers share several problems with decompilers. Muth et al. [MDW01] found the following problems with the `alto` optimiser for the Compaq Alpha:

- It is not possible to assume Application Interface (ABI, essentially calling convention) compliance, since the compiler or linker could perform optimisations that ignores the conventions, as could hand written assembly language code.
- Memory analysis is difficult.
- Preserved registers (registers whose value is unaffected by a call to a particular procedure) are needed. This is covered in Section 4.3.
- Constant propagation is very useful for optimisations and for analysing indirect control transfer instructions. The authors found that an average of about 18% of instructions had operands and results with known constant values.

Analysis is easier at the object code (link-time) level, because of the presence of relocation information. Optimisers have a very simple fallback for when a proposed optimisation cannot be proved to be safe: do not perform the optimisation. `Alto` is able to force conservative assumptions for indirect control transfers by using the special control flow graph nodes $\mathsf{B}_{unknown}$ (for indirect branches) and $\mathsf{F}_{unknown}$ (for indirect calls), which have worst case behaviour (all registers are used and defined). This is

able to be kept reasonably precise because of the presence of relocation information. For example, there are control flow edges from $\mathsf{B}_{unknown}$ to only those basic blocks in the procedure whose addresses have relocation entries associated with them (implying that there is at least one pointer in the program to that code). Without relocation information, $\mathsf{B}_{unknown}$ would have edges to the beginning of all basic blocks, or possibly to all instructions.

2.6.4 Synthesising to Hardware

An additional potential application for decompilation is in improving the performance of hardware synthesised from executable programs.

Programs written for conventional computers can be partially or completely synthesised in hardware such as field programmable gate arrays (FPGAs). Hardware compilers can read source code such as C, or they can read executable programs. Better performance can be obtained if basic control and data flow information is provided, allowing the use of advanced memory structures such as "smart buffers" [SGVN05]. As noted in this paper, decompilation techniques can be used to automatically provide such information, or to provide source code that, even if not suitable for software maintenance, is adequate for synthesising to hardware.

The ability to synthesise to hardware is becoming more important with the availability of chips that incorporate a microprocessor and configurable logic on one chip. These chips allow portions of a program for embedded devices to be implemented in hardware, which can result in significant power savings [SGVV02].

2.6.5 Binary Translation

Static Binary Translators that emit C produce source code from binary code, but since they do not "understand the data", the output has very low readability.

The University of Queensland Binary Translator (UQBT) is an example of a binary translator that uses a C compiler as its back end. It is able to translate moderately sized programs (e.g. the SPEC CPU95 [SPE95] benchmark "go") into very low level C code [UQB01, CVE00, CVEU+99, BT01]. The semantics of each instruction are visible, and all control flow is via `goto` statements and labels. Parameters and returns are identified, although actual parameters are always in the form of simple variables, not expressions.

There is no concept of the data item that is being manipulated, apart from its size. To ensure that the address computed by this instruction will read the intended data,

binary translators have to force the data section of the input binary program to be copied exactly to the data section of the output program and at the same virtual address. In essence, the target program is replicating the bit manipulations of the source program. The data section is regarded as a group of addressable memory bytes, not as a collection of integers, floats, structures, and so on.

This is a subtle but important point. The compiler (if there was one) that created the input program inserted instructions to manipulate high level data objects defined in the source program code. The compiler has an intermediate representation of the data declared in the source program, as well as the imperative statements of the program. By contrast, a binary translator makes no attempt to recreate the intermediate representation of the data. It blindly replicates the bit manipulations from the source program, without the benefit of this data representation. It ends up producing a program that works, but only because the input program worked. It makes no attempt to analyse what the various bit manipulations do. In effect, the binary translator is relying on the symbol table of the original compiler to ensure that data items do not overlap. The source code emitted by such a translator is at the second or third lowest level, "no translation of the data section", as described in Section 1.1 on page 6.

This contrasts with the code quality that a decompiler should produce. A decompiler needs to recover a representation of the data as well as the imperative statements. Experienced programmers reading the decompiled output should be able to understand most of the program's behaviour, just as they would if they were to read the original source code with the comments blanked out. In other words, decompilers should strive for source code that is two levels higher up the list of Section 1.1.

The above discussion concerns static binary translation, where no attempt is made to execute the input program. Dynamic binary translation does run the program, and achieves two of the goals of decompilation. The first is the ability to run a legacy application with no source code on a new platform. The second is the ability to optimise a binary program to run with close to optimal performance on a particular platform (e.g. Dynamo [BDB00]). However, dynamic binary translation in the usual configuration does not allow maintenance of the program except by patching the executable file, and most programs require modification over their lifetimes. However, a modified dynamic binary translator might be able to find values for important pointers or other locations and emit information to aid a static decompilation.

2.6.6 Instruction Set Simulation

This is an automatic technique for generating source code which is like a static interpre-

tation of the input program inlined into a large source file, relying heavily on compiler optimisations for performance.

Instruction Set Simulation is a technique where each instruction of the original machine code program is represented by an inlined interpretation for that instruction. This is a static translation (however there are also dynamic versions, e.g. Shade [CK94]), and the result is source code, like the output of a binary translator, except that even the original program's program counter is visible. This is the lowest level of source code listed in Section 1.1. When macros are used, the "source code" resembles assembly language implemented in a high level language. As with binary translation, no attempt is made to decode the data section(s) [MAF91, Sta02].

2.6.7 Abstract Interpretation

Abstract interpretation research shows how desirable features such as correctness can be proved for decompilation analyses.

Most analyses in a decompiler compute a result that is incomplete in some sense, e.g. a subset of the values that a register could take. In other words, these analyses are abstract evaluations of a program. Cousot and Cousot [CC76] show that provided certain conditions are met, correctness and termination can be guaranteed. This result gives confidence that the various problems facing decompilers can eventually be overcome.

2.6.8 Proof-Carrying Code

Proof-carrying code can be added to machine code to enable enough type analysis to prove various kinds of safety.

Proof-carrying code is a framework for proving the safety of machine-language programs with machine-checkable proofs [AF00]. The proof-carrying code can be at the machine code level [SA01] or at the assembly language level [MWCG98]. While these systems work with types for machine or assembly language, they do not derive the types from the low level code. Rather, extra information in the form of checkable proofs and types are included with the low level code.

Presumably, decompilation of proof-carrying code would be easier than that of normal machine code, perhaps comparable to the decompilation of Java bytecode programs. Verifiable Java programs provide a measure of safety through a different means. It is interesting to observe that most attempts to increase program safety seem to increase the ease of decompilation.

2.6.9 Safety Checking of Machine Code

CodeSurfer/x86 uses a variety of proprietary tools to produce intermediate representations that are similar to those that can be created for programs written in a high-level language.

Some security checking tools read machine code directly, without relying on or in some cases trusting any annotations from the compiler.

In his University of Wisconsin-Madison thesis, Xu uses the concept of typestate checking, which encompasses the types and also the states (e.g. readable, executable) of machine code operands [Xu01]. He uses symbol table information (not usually available in real-world machine code programs) to identify the boundaries of procedures. Xu claims to be able to deal with indirect calls using "a simple intraprocedural constant propagation algorithm".

Also from the University of Wisconsin-Madison, Christodorescu and Jha discuss a Static Analyser for Executables (SAFE) [CJ03]. The binary is first disassembled with the IDA Pro disassembler. A plug-in for IDA Pro called the Connector, provided by GrammaTech Inc [Gra97], interfaces to the rest of the tool. Value-Set Analysis (VSA) is used to produce an intermediate representation, which is presented in a source code analysis tool called CodeSurfer (sold by GrammaTech Inc). CodeSurfer is a tool for program understanding and code inspection, including slicing and chopping. It appears that the executable loader used in this work was an early version that relied on IDA Pro to (possibly manually) analyse indirect jump and call instructions. They were able to detect obfuscated versions of four viruses with no false positives or negatives, where three commercial pattern-based virus detection tools all failed.

A later paper from the same university names the composite tool for analysis and inspection of executables as CodeSurfer/x86 [BR04]. The aim is to produce intermediate representations that are similar to those that can be created for a program written in a high-level language. Limitations imposed by IDA Pro (e.g. not analysing all indirect branches and calls) are mitigated by value-set analysis and also Affine Relations Analysis (ARA). This work appears to make the assumption that absolute addresses and offsets indicate the starting address of program variables. In other words, it does not perform separation of original and offset pointers (see Section 1.5.3). Various safety checks were demonstrated, and additional checks or visualisations can be implemented via an API. Memory consumption is high, e.g. 737MB for analysing `winhlp32.exe`, which is only 265KB in size, and this is while a "temporary expedient" for calls to library functions was used.

Starting in 2005, papers from the same university have been published, featuring a third

major analysis [BR05, RBL06]. Aggregate Structure Identification (ASI) provides the ability to recover record and array structure. A technique the authors call recency-abstraction avoids problems of low precision [BR06]; see also Section 4.6 on page 139. Decompilation is mentioned as a potential application for their tool, but does not appear to have been attempted in the currently published papers.

2.6.10 Traditional Reverse Engineering

Traditional reverse engineering increases high level comprehension, starting with source code, while decompilation provides little high level comprehension, starting with machine code.

Decompilation is a reverse engineering process, as defined by the generally accepted taxonomy [CC90], since it "creates representations of the system in another form or at a higher level of abstraction". It would seem reasonable to assume that decompilation therefore has much in common with traditional reverse engineering, but this is not the case. Traditional reverse engineering is performed on source code, usually with the benefit of some design documentation. From [CC90]:

> "The primary purpose of [traditional] reverse engineering a software system is to increase the overall comprehensibility of the system for both maintenance and new development."

Decompilation provides the basis for comprehension, maintenance and new development, i.e. source code, but any high-level comprehension is provided by the reader. Comprehension in decompilation is limited to such details as "this variable is incremented by one at this point in the program, and its type is unsigned integer". Traditional reverse engineering could add high-level comprehension such as "because the customer has ordered another book".

Baxter and Mehlich [BM00] make the following observation:

> "A single implementation may in fact represent any of several possible plans. An "INC Memory" instruction could implement the "add one" abstraction, the "store a One" (knowing a value is already zero) abstraction, the "Complement the LSB" abstraction, or several other possibilities."

A decompiler can readily deduce that a value is set to one by the combination of clearing and incrementing it; this is an example where decompilation provides a limited kind of

comprehension. It is a mechanical deduction, but quite useful even so. Such mechanical deductions can be surprising at times (see for example the triple xor idiom in Section 4.3). If sufficient identities are preprogrammed into a decompiler, it seems plausible that a "complement the LSB" comment could be inserted under appropriate circumstances (e.g. the LSB is isolated with an appropriate AND instruction, and the result is tested for zero or one in a conditional statement predicate). In most other circumstances, emitting `mem++` or the like seems appropriate, leaving the determination of the higher level meaning to the reader.

2.6.11 Compiler Infrastructures

Several compiler infrastructures exist with mature tool sets which, while intended for forward engineering, might be adaptable for decompilation.

Compilers are becoming so complex, and the need for performance is so great, that several compiler infrastructures have appeared in the last decade, offering the ability to research a small part of the compiler chain, and not having to write the whole compiler, visualisation tools, testing frameworks, and so on. None of these are designed with decompilation in mind, but they are all flexible systems. It is worthwhile therefore to consider whether one of these relatively mature infrastructures, with documented interfaces and debugged transformations already in place, might be able to be extended to become the basis for a machine code decompiler. Presumably, much less work would be required to get to the point where some code could be generated, compared to starting from scratch. Parts that could be used with little modification include the IR design, the C generators, simplification passes, translating into and out of SSA form, constant and copy propagation passes (which could be extended to expression propagation), dead code elimination, and probably many data flow analyses such as reaching definitions. Even the front ends could be used as a quick way to generate IR for experimentation.

2.6.11.1 LLVM

The Low Level Virtual Machine is an infrastructure for binary manipulation tools such as compilers and optimisers. It has also been used for binary to binary tools, such as binary translators. The IR of LLVM is interesting in that it is based on the Static Single Assignment form (SSA form). Memory objects are handled only through load and store instructions, and are therefore not in SSA form.

The LLVM intermediate representation is essentially a list of RISC-like three address instructions, with an encoding scheme that allows an infinite number of virtual registers

while retaining good code density. There are a few concessions for type and exception handling information, but fundamentally this is not suitable for representing the complex expressions needed for decompilation. It is possible that complex expressions could be generated only in the back ends, but this would duplicate the propagation code in all back ends. The propagation could be applied immediately before language specific back ends are invoked, but this would mean that the back ends require a different IR than the rest of the decompiler. Also, earlier parts of a decompiler (e.g. analysing indirect jump and call instructions) rely on the propagation having generated complex expressions. Figure 2.1 shows an overview of LLVM for multi-stage optimisation.

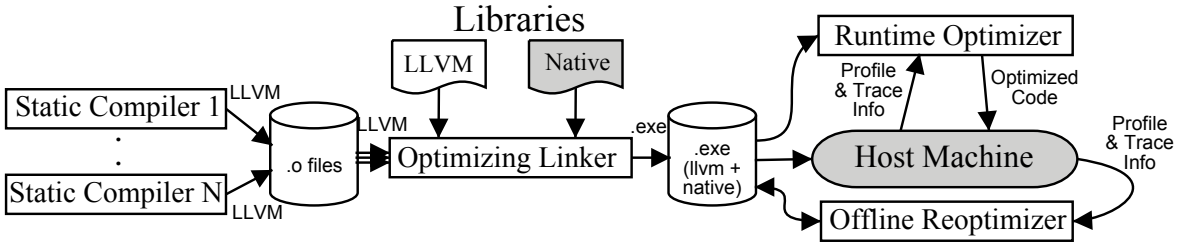


Figure 2.1: LLVM can be used to compile and optimise a program at various stages of its life. From [Lat02].

2.6.11.2 SUIF2

SUIF2 is version 2 of the Stanford University Intermediate Format, an "infrastructure designed to support collaborative research and development of compilation techniques" [ADH+01]. Figure 2.2 gives an brief overview. The SUIF2 IR supports high level program concepts such as expressions and statements, and it is extensible so that it should be able to handle e.g. SSA phi nodes. However, authors of a competing infrastructure state that "SUIF [17] has very little support of the SSA form" [SNK+03]. There are tools to convert the IR to C, front ends for various languages, and various optimisation passes. At any stage, the IR can be serialised to disk, or several passes can operate in memory. SUIF2 may be able to be extended to an IR suitable for decompilers.

2.6.11.3 COINS

COINS is a compiler infrastructure written in Java designed for research on compiler optimisations [SNK+03, SFF+05]. Figure 2.3 shows an overview. It has a high level and a low level IR, and C code can be generated from either. The low level IR (LIR) appears to be suitable for most compiler work, with a tree-based expression representation. LIR is not SSA based, but there is support to translate into and out of SSA form. Operands are typed, but currently the supported types are basically only a size and a

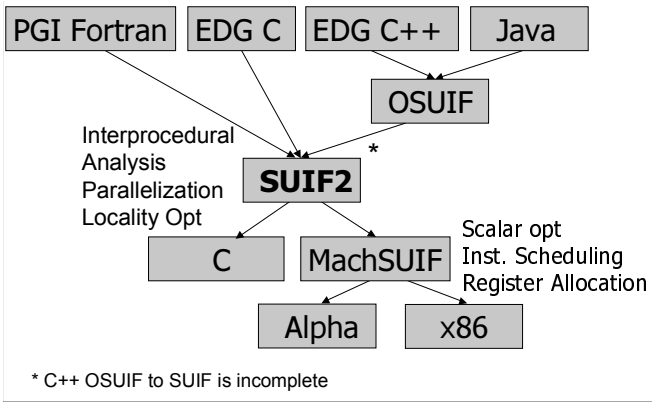


Figure 2.2: Overview of the SUIF2 compiler infrastructure. From [Lam00].

flag indicating whether the operand is an integer (sign is not specified) or floating point. There is some extensibility in the IR design, so COINS may be able to be expanded be become the basis of a decompiler IR.

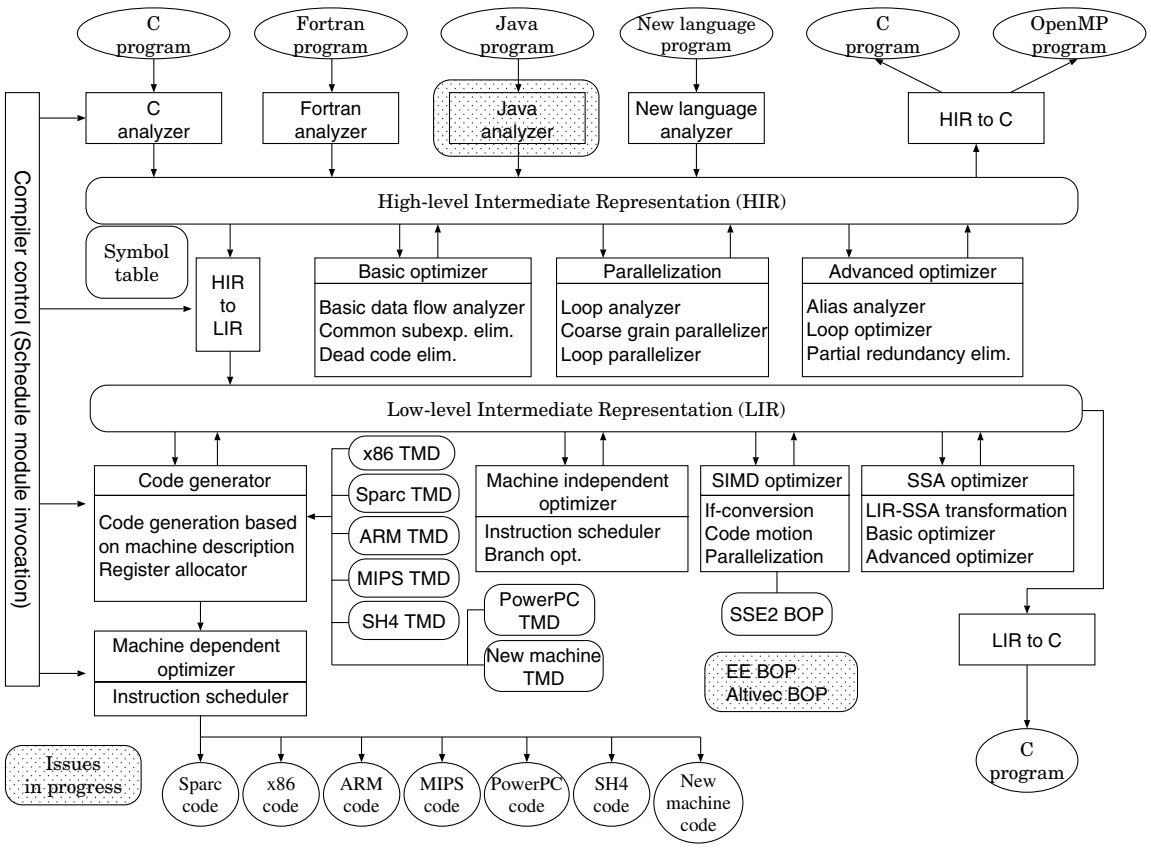


Figure 2.3: Overview of the COINS compiler infrastructure. From Fig. 1 of [SFF+05].

2.6.11.4 SCALE

"Scale is a flexible, high performance research compiler for C and Fortran, and is written in Java" [CDC+04, Sca01]. The data flow diagram of Figure 2.4 gives an overview. Again, there is a high and low level IR; the low level IR is in SSA form. Expressions in the SSA CFG are tree based, hence it may be possible to adapt this infrastructure for research in decompilation.

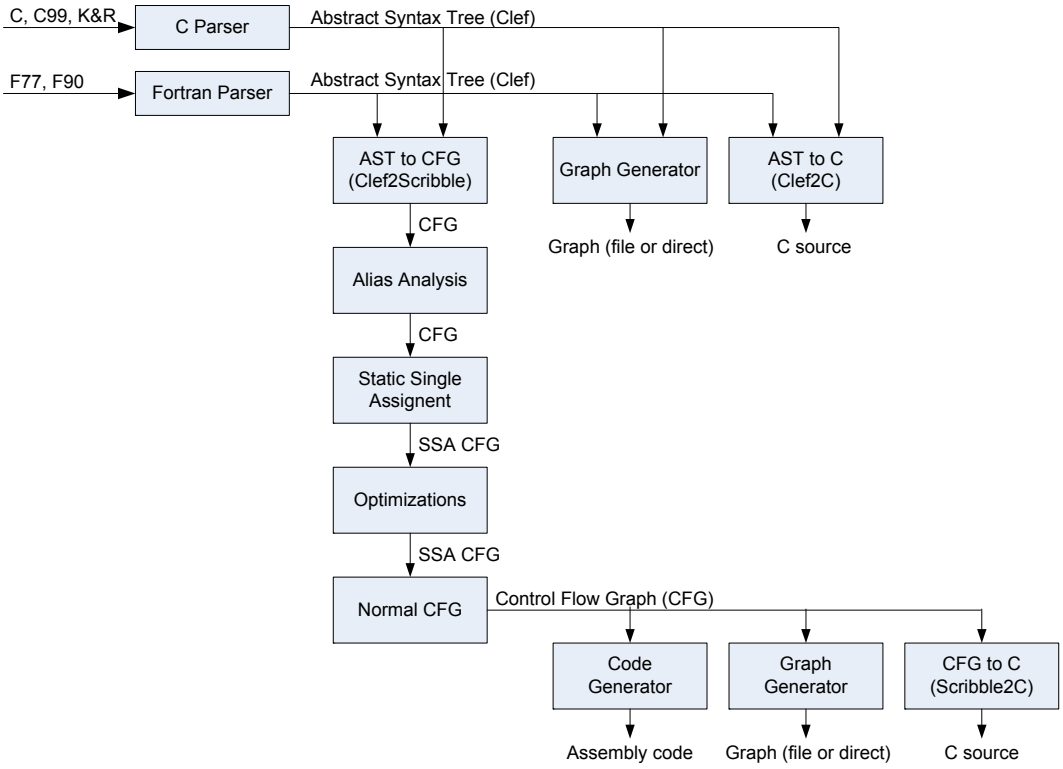


Figure 2.4: Scale Data Flow Diagram. From [Sca01].

2.6.11.5 GCC Tree SSA

As of GCC version 4.0, the GNU compiler collection includes the GENERIC and GIMPLE IRs, in addition to the RTL IR that has always been a feature of GCC [Nov03, Nov04]. RTL is a very low level IR, not suitable for decompilation. GENERIC is a tree IR where arbitrarily complex expressions are allowed. Compiler front ends can convert their language-dependent abstract syntax trees to GENERIC, and a common translator (the gimplifier) could convert this to GIMPLE, a lower version of GENERIC with three address representations for expressions. Alternatively, front ends can convert directly to GIMPLE. All optimisation is performed at the GIMPLE level, which is probably too low level for decompilation. GENERIC shows some promise, but few of the tools support it. Figure 2.5 shows the relationship of the various IRs.

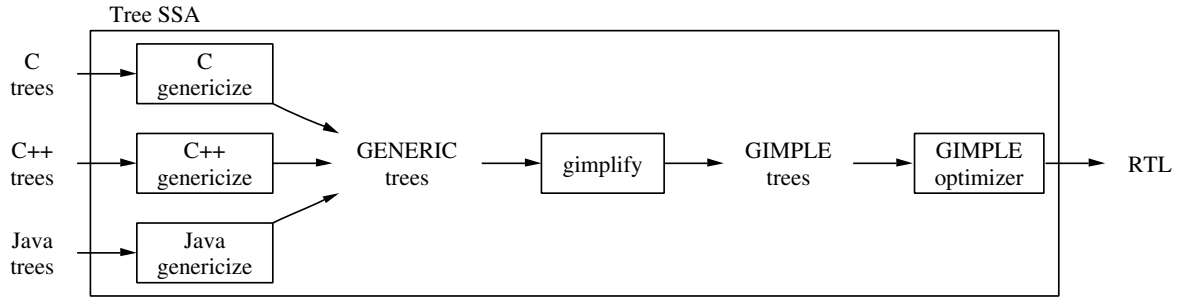


Figure 2.5: The various IRs used in the GCC compiler. From a presentation by D. Novillo.

There is a new SSA representation for aggregates called Memory SSA [Nov06] which is planned for GCC version 4.3 . While primarily aimed at saving memory in the GCC compiler, some ideas from this representation may be applicable to decompilation.

2.6.11.6 Phoenix

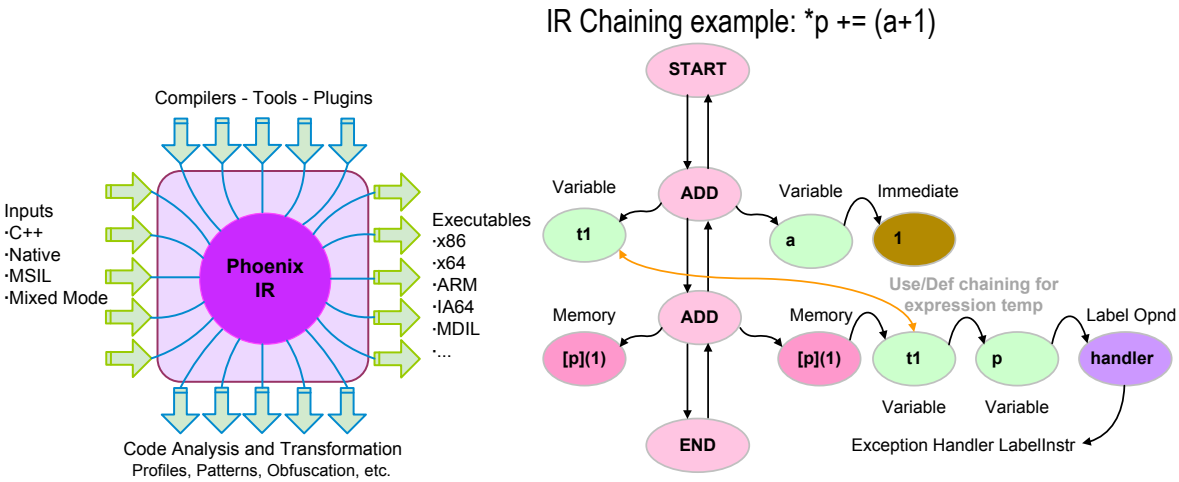


Figure 2.6: Overview of the Phoenix compiler infrastructure and IR. From [MRU07].

Phoenix is a compiler infrastructure from Microsoft Research, used by Microsoft as the basis of future compiler products. Figure 2.6 gives an overview, and an example of IR showing one line of source code split into two IR instructions. Phoenix has the ability to read native executable programs, but does not (as of this writing) come with a C generator. IR exists at three levels (high, medium, and low), plus an even lower level for representing data. Unfortunately, even at the highest level (HIR), expression operands can not be other expressions, only variables, memory locations, constants, and so on. The three main IR levels are kept as similar as possible to make it easier for phases to operate at any level. As a result, Phoenix would probably not be a suitable infrastructure for decompiler research [MS04].

2.6.11.7 Open64

Open64 is a compiler infrastructure originally developed for the IA64 (Itanium) architecture. It has since been broadened to target x86, x64, MIPS, and other architectures. The IR is called WHIRL, and exists at five levels, as shown in Figure 2.7.

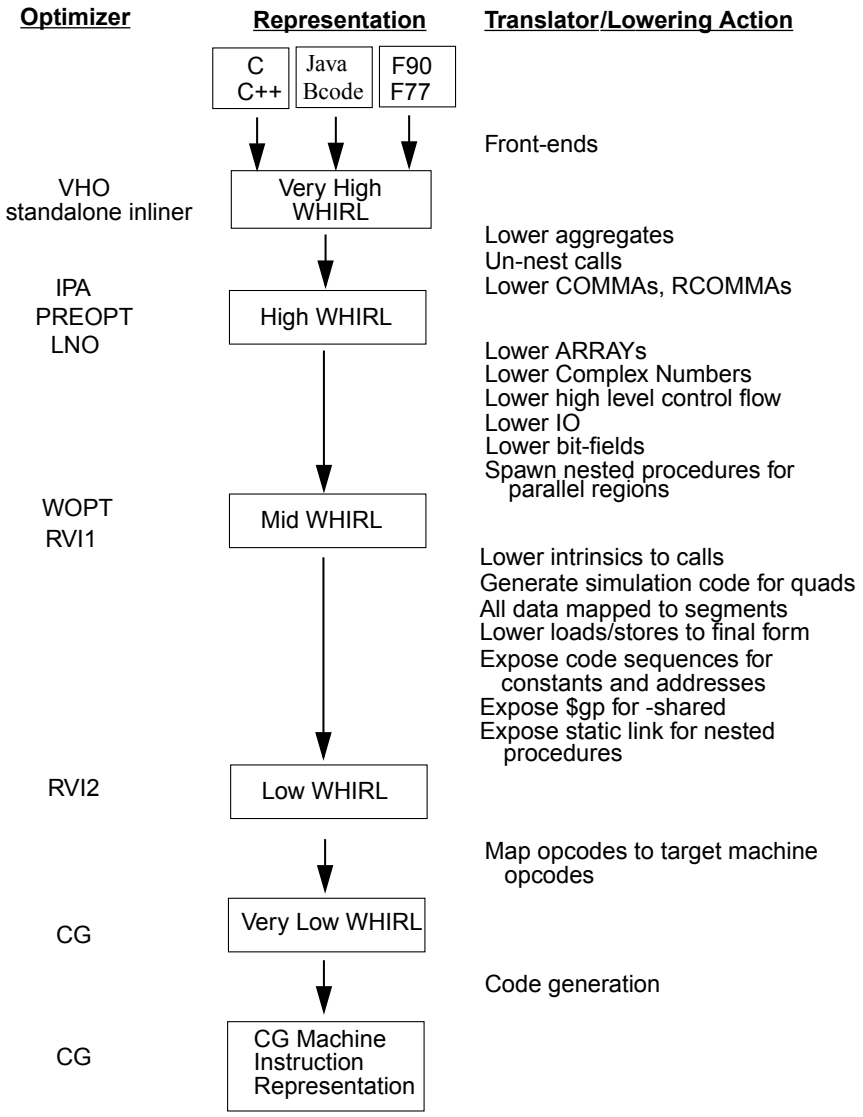


Figure 2.7: The various levels of the WHIRL IR used by the Open64 compiler infrastructure. From [SGI02].

At all IR levels, the IR is a tree, which is suitable for expressing high level language constructs. The higher two levels can be translated directly to C or Fortran by provided tools. Many of the compiler optimisations use the SSA form, hence there are facilities for transforming into and out of SSA form. This infrastructure would therefore appear to be suitable for decompilation research.

Open64 is free software licensed under the GPL, and can be downloaded from SourceForge [SF06]. See also [UH02].

2.6.12 Simplification of Mathematical Formulae

Decompilers can make use of research on the simplification of mathematical formulae to improve the output.

One of the transformations needed by compilers, decompilers, and other tools is the simplification of expressions, e.g. $x+0 \rightarrow x$. The general goal is to reduce terms to canonical or simpler forms. Dolzmann and Sturm [DS97] give ideas on what constitutes "simplest": few atomic formulae, small satisfaction sets, etc. They point out that some goals are contradictory, so at times a user may have to select from a set of options, or select general goals such as "minimum generated output size" or "maximum readability".

The field of decompilation probably has much to learn from such related work, to fine tune the output presented to users.

2.6.13 Obfuscation and Protection

Obfuscation and protection are designed to make the reverse engineering of code more difficult, including decompilation; in most cases, such protection prevents effective decompilation.

Obfuscation is the process, usually automatic, of transforming a program distributed in machine code form in such a way as to make understanding of the program more difficult. (There is also source code obfuscation [Ioc88], but at present this appears to be largely a recreational activity.)

A typical technique is to add code not associated with the original program, confusing the reverse engineering tool with invalid instructions, self modifying code, and the like. Branches controlling entry to the extra code are often controlled by opaque predicates, which are always or never taken, without it being obvious that this is the case. Another technique is to modify the control flow in various ways, e.g. turning the program into a giant switch statement.

"Protection" of executable programs is a term that encompasses various techniques designed to hide the executable code of a program. Techniques include encrypting the instructions and/or data, using obscure instructions, self modifying code, and branching to the middle of instructions. Attempts to thwart dynamic debuggers are also common, but these do not affect static decompilation except to add unwanted code to the output.

The result of decompiling an obfuscated or protected program will depend on the nature of the modifications:

- If compiler specific patterns of machine code are merely replaced with different but equivalent instructions, the result will probably decompile perfectly well with a general decompiler.
- Where extraneous code is inserted without special care, the result could well decompile perfectly with a general decompiler.
- Where extraneous code is inserted with special care to disguise its lack of reachability, the result will likely be a program that is correct and usable, but difficult to understand and costly to maintain unless most of the obfuscations are manually removed. Even though such removal could be performed at the source code level, any removal is likely to be both tedious and costly. Some obfuscations (e.g. creating branches to the middle of instructions) may cause a decompiler to fail to generate a valid representation of the program.
- Where self-modifying code is present, the output is likely to be incorrect.
- Where the majority of the program is no longer in normal machine language, but is instead encrypted, the result is likely to be only valid for the decrypting routine, and the rest will be invalid. Where several levels of encryption and/or protection are involved, the result is likely to be valid only for the first level of decryption.
- Where the majority of the program has been translated into instructions suitable for execution by a Virtual Machine (VM), and a suitable interpreter or Just In Time (JIT) compiler is part of the executable program, the result will likely be valid only for the VM interpreter or JIT compiler.

As a research area, obfuscation has only received serious attention since the mid 1990s, with the appearance of Java bytecodes and CLI.

Collberg et al. have published the seminal paper in this area, *"A Taxonomy of Obfuscating Transformations"* [CTL97]. Wroblewski's thesis [Wro02] presents a method for obfuscating binary programs.

The difficulties presented by most forms of executable program protection indicate that decompilation of such code is usually not feasible. Where decompilation of a heavily protected program is required (e.g. source code is lost, and the only available form of the program is a protected executable), the protection must first be removed by processes unrelated to decompilation.

Chapter 3

Data Flow Analysis

Static Single Assignment form assists with most data flow components of decompilers, assisting with such fundamental tasks as expression propagation, identifying parameters and return values, deciding if locations are preserved, and eliminating dead code.

Data flow analysis and control flow analysis are two of the main classes of analyses for machine code decompilers, as shown in Figure 3.1. Control flow analysis, where the intermediate representation of a program is converted to high level statements such as conditionals and loops, is a largely solved problem. However, data flow analysis, where instruction semantics are transformed into more complex expressions, parameters and return values are identified, and types are recovered, still has significant potential for improvement.

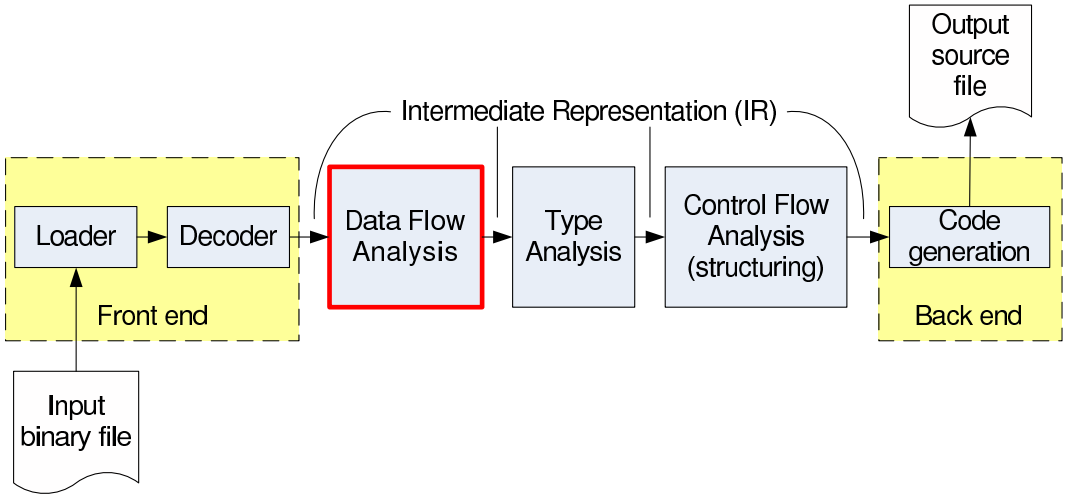


Figure 3.1: Basic components of a machine code decompiler.

The intermediate representation (IR) for a decompiler resembles that of a compiler. It consists of statements containing expressions. For example, there may be a branch

```
#include <stdio.h>
/*          n       n!          n(n-1)...
 * Calculate C = -------- = --------------- (n-r terms top and bottom)
 *            r  r!(n-r)!    (n-r)(n-r-1)...  */
int comb(int n, int r) {
   if (r >= n)
      r = n;              /* Make sure r <= n */
   double res = 1.0;
   int num = n;
   int denom = n-r;
   int c = n-r;
   while (c-- > 0) {
      res *= num--;
      res /= denom--;
   }
   int i = (int)res;      /* Integer result; truncates */
   if (res - i > 0.5)
      i++;                /* Round up */
   return i;
}
int main() {
   int n, r;
   printf("Number in set, n:  ");    scanf("%d", &n);
   printf("Number to choose, r:  "); scanf("%d", &r);
   printf("Choose %d from %d:  %d\n", r, n, comb(n, r));
   return 0;
}
```

Figure 3.2: Original source code for the combinations program.

```
80483b0: 55        push %ebp           ; Standard function prologue
80483b1: 56        push %esi           ; Save three registers
80483b4: 53        push %ebx
80483b5: 57        push %ecx
80483b6: 83 ec 08  sub  $0x8,%esp      ; Space for 8 bytes of locals
80483b9: 8b 55 08  mov  0x8(%ebp),%edx ; edx stores n and num
80483bc: 89 f9     mov  %ecx,%ebx      ; ebx = r
80483be: 2b 4d 08  sub  0x8(%ebp),%ebx ; ebx = r-n, carry set if r<n
80483c1: 19 c0     sbb  %eax,%eax      ; eax = -1 if r<n else eax = 0
80483c3: 21 c8     and  %ebx,%eax      ; eax = r-n if r<n else eax = 0
80483c5: 03 45 08  add  0x8(%ebp),%eax ; eax = r-n=r if r<n, else eax=n
```

Figure 3.3: First part of the compiled machine code for procedure *comb* of Figure 3.2.

Table 3.1: x86 assembly language overview.

Assembly language	Explanation
`%eax`, ... `%edx`, `%esi`, `%edi`, `%ebp`, `%esp`	Registers. `%esp` is the stack pointer; `%ebp` is the "base pointer", usually pointing to the top of the stack frame.
`%st`, `%st(1)`	Top of the floating point register stack (often implied), and next of stack
`(`*r*`)`	Memory pointed to by register *r*.
disp`(`*r*`)`	Memory whose address is *disp* + the value of register *r*.
`push` *r*	Push the register *r* to the stack. Afterwards, the stack pointer *esp* points to *r*.
`pop` *r*	Pop the value currently pointed to by *esp* to register *r*. Afterwards, *esp* points to the new top of stack.
`mov` *sr*`,`*dest*	Move (copy) the register or immediate value *sr* to register or memory *dest*.
`cmp` *sr*`,`*dr*	Compare the register *dr* with the register or immediate value *sr*. Condition codes (e.g. carry, zero) are affected.
`jle` *dest*	Jump if less or equal to address *dest*. Uses condition codes set by an earlier instruction.
`lea` *mem*`,`*rd*	Load the effective address of *mem* to register *rd*.
`jmp` *dest*	Jump to address *dest*. Equivalent to loading the program counter with *dest*.
`call` *dest*	Call the subroutine at address *dest*. Equivalent to pushing the program counter and jumping to *dest*.
`test` *rs*`,`*rd*	And register or memory *rd* to register or immediate value *rs*; result is not stored. Condition codes are affected.
`sub` *rs*`,`*rd*	*rd* := *rd* - *rs*. Condition codes are affected.
`sbb` *rs*`,`*rd*	rd := rd - rs - carry/borrow. Condition codes are affected.
`fidiv` *mem*	The value at the top of the floating point stack (FPTOS) is divided by the integer in memory location *mem*.
`leave`	Equivalent to `mov %ebp,%esp`; `pop %ebp`.
`ret`	Subroutine Return. Equivalent to `pop temp`; `jmp temp`
`dec` *rd*	Decrement register *rd*, i.e. *rd* := *rd*`-1`.
`nop`	No operation (do nothing).

statement with the expression `m[esp-8] > 0` representing the condition for which the branch will be taken, and another expression (often a constant) which represents the destination of the branch in the original program's address space. `esp` is the name of a register (the x86 stack pointer), and `m[esp-8]` represents the memory location whose address is the result of subtracting 8 from the value of the `esp` register. Register and memory locations are generally the only entities that can appear on the left hand side of assignments. When locations appear in the decompiled output, they are converted to local or global variables, array elements, or structure members, depending on their form and how they are used. Expressions consist of locations, constants, or operators

combining other expressions, locations, or constants. Operators consist of the usual high level language operators such as addition and bitwise or. There are also a few low level operators such as those which convert various sizes of integers to and from floating point representations.

Data flow analysis concerns the *definitions* and *uses* of locations. A definition of a location is where is it assigned to, usually on the left hand side of an assignment. One or more locations could be assigned to as a side effect of a call statement. A use of a location is where the value of the location affects the execution of a statement, e.g. in the right hand side of an assignment, or in the condition or destination of a branch. Just as compilers require data flow analysis to perform optimisation and good code generation, decompilers require data flow analysis for the various transformations between instruction decoding and control flow analysis.

To demonstrate the various uses of data flow analysis in decompilers, the running example of Figure 3.2 will be used for most of this and the next chapters. It is a simple program for calculating the number of combinations of r objects from a set of n objects. The combinations function is defined as follows:

$$ {}^{n}C_{r} = \binom{n}{r} = \frac{n!}{r!(n-r)!} = \begin{cases} \overbrace{\frac{n(n-1)...}{(n-r)(n-r-1)...}}^{(n-r)\ terms\ top\ and\ bottom} & if\ n > r \\ 1 & if\ n = r \\ undefined & if\ n < r \end{cases} $$

Figure 3.3 shows the first part of a possible compilation of the original program for the x86 architecture. For readers not familiar with the x86 assembly language, an overview is given in Table 3.1. The disassembly is shown in AT&T syntax, where the last operand is the destination, as used by the `binutils` tools such as `objdump` [FSF01].

Data flow analysis is involved in many aspects of decompilation, and is a well known technique. In compilers, it is frequently required for almost any form of optimisation [ASU86, App02]. Other tools such as link time optimisers also use data flow analysis [Fer95, SW93]. Early decompilers made use of data flow analysis (e.g. [Hol73, Fri74, Bar74]).

Section 3.1 shows how useful expression propagation is in decompilation. Uncontrolled propagation, however, can cause problems as Section 3.2 shows. Dead code elimination, introduced in Section 3.3, reduces the bulk of the decompiled output, one of the advantages of high level source code. One set of the machine code details that are eliminated includes the setting and use of condition codes; Section 3.3.1 shows how these

are combined. The x86 architecture has an interesting special case with floating point compares, discussed in Section 3.3.2. One of the most important aspects of a program's representation is the way that calls are summarised. Section 3.4 covers the special terminology related to calls, the necessity of changing from caller to callee contexts, and discusses how parameters, returns, and global variables interact when locations are defined along only some paths. The various elements of call summaries are enumerated in Section 3.4.4 in the form of equations. Section 3.5 considers whether data flow could or should be performed over the whole program at once, while Section 3.6 discusses safety issues related to the summary of data flow information. Architectures that have overlapped registers present problems that can be overcome with data flow analysis, as shown in Section 3.7.

3.1 Expression Propagation

Expression propagation is the most common transformation used by decompilers, and there are two simple rules, yet difficult to check, for when it can be applied.

The IR for the first seven instructions of Figure 3.3 are shown in Figure 3.4(a). Note how statement 2 uses (in `m[esp]`) the value of `esp` computed in statement 1.

Let `esp0` be a special register for the purposes of the next several examples. An extra statement, statement 0, has been inserted at the start of the procedure to capture the initial value of the stack pointer register `esp`. This will turn out to be a useful thing to do, and the next chapter will introduce an intermediate representation that will do this automatically.

Statement 0 can be propagated into every other statement of the example; this will not always be the case. For example, statement 1 becomes `esp := esp0 - 4`. Similarly, `m[esp]` in statement 2 becomes `m[esp0-4]`, as shown in Figure 3.4(b). Propagation is also called "forward propagation" or "forward substitution".

The rules for when propagation is allowed are well known. A propagation of x from a source statement s of the form $x := exp$ to a destination statement u can be performed if the following conditions hold (adapted from [ASU86]):

1. Statement s must be the only definition of x that reaches u.
2. On every path from s to u, there are no assignments to any location used by s.

For example if s is `edx := m[ebp+8]`, then there must be no assignments to `m[ebp+8]` or to `ebp` along any path from s to u. With traditional data flow analysis, two separate

```
         0 esp0 := esp          ; Save esp; see text
80483b0  1 esp := esp - 4
         2 m[esp] := ebp         ; push ebp
80483b1  3 ebp := esp
80483b3  4 esp := esp - 4
         5 m[esp] := esi         ; push esi
80483b4  6 esp := esp - 4
         7 m[esp] := ebx         ; push ebx
80483b5  8 esp := esp - 4
         9 m[esp] := ecx         ; push ecx
80483b6 10 tmp1 := esp
        11 esp := esp - 8        ; sub esp, 8
80483b9 13 edx := m[ebp+8]       ; Load n to edx
```

(a) Before expression propagation

```
         0 esp0 := esp
80483b0  1 esp := esp0 - 4
         2 m[esp0-4] := ebp
80483b1  3 ebp := esp0-4
80483b3  4 esp := esp0 - 8
         5 m[esp0-8] := esi
80483b4  6 esp := esp0 - 12
         7 m[esp0-12] := ebx
80483b5  8 esp := esp0 - 16
         9 m[esp0-16] := ecx
80483b6 10 tmp1 := esp0 - 16
        11 esp := esp0 - 24
80483b9 13 edx := m[esp0+4]      ; Load n to edx
```

(b) After expression propagation

```
80483b0  2 m[esp0-4] := ebp
80483b3  5 m[esp0-8] := esi
80483b4  7 m[esp0-12] := ebx
80483b5  9 m[esp0-16] := ecx
80483b9 13 edx := m[esp0+4]      ; Load n to edx
```

(c) After dead code elimination

Figure 3.4: IR for the first seven instructions of the combinations example.

analyses are required. The first analysis pass uses use-definition chains (ud-chains), based on reaching definitions, a forward-flow, any-path data flow analysis. The second one requires a special purpose analysis called "rhs-clear", a forward-flow, all-paths analysis [Cif94]. The next chapter will introduce an intermediate representation which makes the checking of these rules much easier.

Statement 3 also uses the value calculated in statement 1, so statement 1 is also propagated to statement 3. Statement 13 uses the value calculated in statement 3, and the

conditions are still met, so statement 3 can be propagated to statement 13. The result of applying all possible propagations is shown in Figure 3.4(b).

Note that the quantity being propagated in these cases is of the form `esp0-K`, where `K` is a constant; the quantity being propagated is an expression. This is *expression propagation*, and differs from the constant propagation and copy propagation performed in compilers. Compilers propagate statements of the form $x := y$, where x is a variable and y is either a constant or another variable, but not a more complex expression. Performing expression propagation tends to make the result more complex, copy propagation leaves the complexity the same, and constant propagation usually makes the result simpler. Compilers translate potentially complex expressions eventually to simple machine instructions. In other words, the overall progression is from the complex to the simple. Decompilers do the opposite; the progression is from the simple to the complex. As a result, decompilers need not restrict propagation to that of constants or simple variables (locations). Copy and constant propagation are also useful for decompilers, since propagating them tends to reduce the need for temporary variables.

Expression propagation, when applied repeatedly, tends to result in expressions that are in terms of the input values of the procedure. This is already evident in Figure 3.4(b), where the uses of seven different values for `esp` are replaced by constant offsets from one (`esp0`). This tendency is very useful in a decompiler, because memory locations end up in a canonical form, as shown in the following example.

```
80483d8 89 34 24 mov %esi,(%esp)     ; Copy denom to integer stack
80483db da 75 e8 fidiv -24(%ebp)     ; res /= denom
```

(a) Machine code

```
80483d8 55 m[esp] := esi
80483db 56 st := st /f (double)m[ebp-24]
```

(b) IR before propagation

```
80483d8 55 m[esp0-28] := esi
80483db 56 st := st /f (double)m[esp0-28]
```

(c) IR after propagation

Figure 3.5: Two machine instructions referring to the same memory location using different registers. `/f` is the floating point division operator, and `(double)` is the integer to floating point conversion operator.

Figure 3.5 shows two instructions from inside the while loop, which implement the division. In Figure 3.5(a), `m[esp]` is an *alias* for `m[ebp-24]`, however this fact is not

obvious. After repeated propagation, both memory expressions result in `m[esp0-28]`, as shown in Figure 3.5(c). As a result, both memory expressions are treated as representing the same memory location, as they should. Treating them as separate locations could result in errors in the data flow information. This form of aliasing is more prevalent at the machine code level than at the source code level.

3.2 Limiting Propagation

Although expression propagation is usually beneficial in a decompiler, in some circumstances limiting propagation produces more readable output.

Figure 3.6 shows decompiler output where too much propagation has been performed.

```
24 if (local26 >= maxTickSizeX - (maxTickSizeX < 0 ?  -1 :  0) >> 1) {
25   local26 = (maxTickSizeX - (maxTickSizeX < 0 ?  -1 :  0) >> 1) - 1;
```

Figure 3.6: Excessive propagation. From [VEW04].

The result is needlessly difficult to read, because a large expression has been duplicated. The reader has to try to find a meaning for the complex expression twice, and it is not obvious from the code whether the condition is the same as the right hand side of the assignment on line 25. Figure 3.7 shows how the problem arises in general.

```
x := large_expression          x := large_expression  // Dead code
if (x ...)                     if (large_expression ...)
   ...  x                         ...  large_expression
```

(a) Original intermediate code (b) Undesirable code after propagation

Figure 3.7: The circumstance where limiting expression propagation results in more readable decompiled code.

Common Subexpression Elimination (CSE) is a compiler optimisation that would appear to solve this problem. However, implementing CSE everywhere possible effectively reverses all propagations, as demonstrated by the results in Section 7.2.

Excessive propagation can be limited by heuristics, as shown below. In some cases, it may be desirable to undo certain propagations, e.g. where specified manually. For those cases, CSE could be used to undo specific propagations. While compilers perform CSE for more compact and faster machine code, decompilers limit or undo expression propagation to reduce the bulk of the decompiled code and improve its readability.

The problem with Figure 3.6 is that a large expression has been propagated to more than one use. Where this expression is small, e.g. `max-1`, this is no problem, and is probably easier to read than a new variable such as `max_less_one`.

Similarly, when a complex expression has only one use, it is also not a problem. However, where the result of a propagation would be an expression occupying several lines of code, it may be worth preventing the propagation. The threshold for such propagation limiting may vary with individual user's preferences.

The main problematic propagations are therefore those which propagate expressions of moderate complexity to more than one destination. Simple measures of expression complexity will probably suffice, e.g. the number of operators is easy to calculate and is effective.

Section 7.2 demonstrates that minimising such propagations is effective in improving the readability of the output in these cases.

3.3 Dead Code Elimination

Dead code elimination is facilitated by storing all uses for each definition (definition-use information).

Dead code consists of assignments for which the definition is no longer used; its elimination is called Dead Code Elimination (DCE). Dead code contrasts with unreachable code, which can be any kind of statement, to which there is no valid path from the program's entry point(s).

Propagation often leads to dead code. In Figure 3.4(b), half the statements are calculations of intermediate results that will not be needed in the decompiled output, and are therefore eliminated as shown in Figure 3.4(c). In fact, all the statements in this short example will ultimately be eliminated as dead code. DCE is also performed in compilers, where it is considered an optimisation. The main process by which the machine specific details of the executable program are removed in a decompilation is by a combination of expression propagation and dead code elimination. This removes considerable bulk and greatly increases the readability of the generated output. Hence, DCE is an important process in decompilation.

As the name implies, dead code elimination depends on the concept of *liveness* information. If a location is live at a program point, changing its value (e.g. by overwriting it or eliminating its definition) will affect the execution of the program. Hence, a live variable can not be eliminated. In traditional data flow analysis, every path from a definition to the end of the procedure has to be checked to ensure that on that path the location is defined before it is used. The next chapter will introduce an intermediate representation with the property that any use anywhere in the program implies that the location is live.

Some instructions typically generate effects that are not used. For example, a divide instruction usually computes both the quotient and the remainder in two machine registers. Most high level languages have operators for division and remainder, but typically only one or the other operation is specified. As a result, the divide and similar instructions often generate dead code.

3.3.1 Condition Code Combining

Expression propagation can be used to implement machine independent combining of condition code definitions and uses.

Another machine specific detail which must usually be eliminated is the condition code register, also called the flags register or status register. Individual bits of this register are called condition codes, flags, or status bits. Condition codes are a feature of machine code but not high level source code, so the actual assignments and uses of the condition codes should be eliminated. In some cases, it may be necessary for a flag, e.g. the carry flag as a local Boolean variable CF, to be visible. In general, such cases should be avoided if at all possible, since they represent very low level details that programmers normally don't think about, and hence do not appear in normal source code.

In the example IR of Figure 3.4, the instruction at address `80483b6`, a subtract immediate instruction, has the side effect of setting the condition codes. There is a third statement for this instruction (statement 12, omitted earlier for brevity) representing this effect, as shown in Figure 3.8(a).

As shown in Figure 3.8(b), the semantics implied by the SUBFLAGS macro are complex. Decompilers do not usually need the detail of the macro expansion. Other macro calls are used to represent the effect on condition codes after other classes of instructions, e.g. floating point instructions, or logical instructions such as a bitwise `and`. For example, add and subtract instructions affect the carry and overflow condition codes differently on most architectures.

In this case, none of the four assignments to condition codes are used. In a typical x86 program, there are many modifications of the condition codes that are not used; the semantics of these modifications are removed by dead code elimination. Naturally, some assignments to condition codes are not dead code, and are used by subsequent instructions.

Separate instructions set condition codes (e.g. compare instructions), and use them (e.g. conditional branch instructions). Specific instructions which use condition codes use different combinations of the condition code bits, e.g. one might branch if lower (carry

```
80483b6 10 tmp1 := esp
        11 esp := esp - 8            ; sub $8,%esp
        12 %flags := SUBFLAGS(tmp1, 8, esp)
```

(a) Complete semantics for the instruction using the SUBFLAGS macro call.

```
SUBFLAGS32(op1, op2, result) {
   %CF := op1 <u op2
   %OF := MSB(op1) ^ MSB(op2) ^ MSB(result) ^ %CF
   %NF := result <s 0
   %ZF := result == 0
};
```

(b) Definition for the SUBFLAGS macro. `%CF`, `%OF`, `%NF` and `%ZF` represent respectively the carry, overflow, negative, and zero flags; `MSB(`*loc*`)` represents the most significant bit of location *loc*; $<_u$ and $<_s$ are the unsigned and signed comparison operators.

Figure 3.8: The subtract immediate from stack pointer instruction from register `esp` from Figure 3.4(a) including the side effect on the condition codes.

set) or equal (zero set), while another might set a register to one if the sign bit is set. Hence the overall effect is the result of combining the semantics of the instruction which sets the condition codes, and the instruction which uses them. Often, these instructions are adjacent or near each other, but it is quite possible that they are separated by many instructions.

Data flow analysis makes it easy to find the condition code definition associated with a condition code use. It reveals which instructions combine at a particular condition code use, and hence the semantics of that use. This combination can be used to define the semantics of the instruction using the condition codes (e.g. the branch condition is $a \leq b$). After this is done, the condition code assignments are no longer necessary, and can be eliminated as dead code.

In a decompiler, modifications to the condition code register can be modelled by assignments to the `%flags` abstract location, as shown earlier in Figure 3.8(a). Figure 3.9(b) shows an assignment to the condition codes by a decrement instruction that is used by the following branch instruction.

In general, after an arithmetic instruction (e.g add, decrement, compare), there will be an assignment of the form

`%flags := SUBFLAGS(`*op1*`,` *op2*`,` *res*`)`

where *op1* and *op2* are the operands, and *res* is the result (sum or difference). Add-like instructions use the ADDFLAGS macro, which has slightly different semantics to the

```
80483e2 4B     dec  %ebx     ; Decrement counter and set flags
80483e3 7d ee  jge  80483d3  ; Continue while counter >= 0
```

(a) Machine code

```
80483e2 59 tmp1 := ebx
        60 ebx := ebx - 1
        61 %flags := SUBFLAGS(tmp1, 1, ebx)
80483e3 62 BRANCH 80483d3, conditions: ge, %flags
```

(b) IR before propagation

```
80483e2 59 tmp1 := ebx                              // Dead code
        60 ebx := ebx - 1
        61 %flags := SUBFLAGS(tmp1, 1, ebx)         // Dead code
80483e3 62 BRANCH 80483d3 if ebx >= 1
```

(c) IR after propagation

Figure 3.9: Combining condition codes in a decrement and branch sequence.

SUBFLAGS macro. A compare instruction is the same as a subtract instruction without the result being stored, however it is convenient to compute a result in a temporary location to pass as the last operand of the SUBFLAGS macro. After a logical instruction (`and`, `xor`, `test`) there will be an assignment of the form

```
%flags := LOGICALFLAGS(res)
```

where `res` is the result of the logical operation.

Branch statements contain an expression representing the high-level condition, essentially the expression that would be emitted within parentheses of an `if` statement when decompiling to C. For integer branches, this expression is initially set to `%flags`, so that expression propagation will automatically propagate the correct setting of the integer condition codes to the branch statement. Floating point branches initially set the condition to `%fflags` (representing the *floating point* flags), so that interleaved integer and floating point condition codes producers (e.g. compare instructions) and consumers (e.g. branch statements) will combine correctly.

At many stages of decompilation, expressions are *simplified*. For assignments, identities such as $x + 0 = x$ are applied. Combining condition code definitions and uses can be implemented as a special case of expression simplification. For example, there can be special modifications that convert

`BRANCH (SUBFLAGS(`*op1*`, `*op2*`, `*res*`), BRANCH_ULE)` to `BRANCH(`*op1* $<_u$ *op2*`)` and

`BRANCH (LOGICALFLAGS(`*op1*`, `*op2*`, `*res*`), BRANCH_MINUS)` to `BRANCH(`*res* `< 0)`.

Here, `BRANCH_ULE` is an enumeration representing branches with the condition *unsigned less*, and $<_u$ represents the *unsigned less than* operator.

The general method of using propagation to combine the setting and use of condition codes to form conditional expressions is due to Trent Waddington, who first implemented the method in the Boomerang decompiler [Boo02]. It is able to handle various control flow configurations, such as condition code producers and consumers appearing in different basic blocks.

There is a special case where the carry flag can be used explicitly, as part of a subtract with borrow instruction. Figure 3.10 from the running example shows a subtract with borrow instruction used in an idiomatic code sequence.

The essence of the sequence is to perform a compare or subtract which affects the condition codes, followed by a subtract with borrow of a register from itself. The carry flag (which also stores borrows from subtract instructions) is set if the first operand of the compare or subtract is unsigned less than the second operand. The result is therefore either zero (if no carry/borrow) or -1 (if carry/borrow was set). This value can be `and`ed with an expression to compute conditional values without branch instructions, or by adding one to the result, to compute C expressions such as `a = (b < c)` (`a` is set to 1 if `b<c`, else 0).

In the example of Figure 3.10, `ebx` is set to `r-n`, and the carry flag is set if `r<n` and cleared otherwise. Register `eax` is subtracted from itself with borrow, so `eax` becomes -1 if `r<n`, or 0 otherwise. After `eax` is `and`ed with `ebx`, the result is `r-n` if `r<n`, or 0 otherwise. Finally, `n` is added to the result, giving `r` if `r<n`, or `n` otherwise.

The carry flag can be treated as a use of the `%flags` abstract location, so that appropriate condition code operations will propagate their operands into the `%CF` location. During propagation, a special test is made for the combination of propagating `%flags` = `SUBFLAGS(`*op1*`,` *op2*`,` *res*`)` into `%CF`. The branch condition is set to *op1* $<_u$ *op2*. The result is correct, compilable, but difficult to read code as shown in Figure 3.10(e). Readability could be improved with an extra simplification rule of the form

`((0 - (`*a*$<_u$*b*`) & (`*a*`-`*b*`)) ⇒ (`*a*$<_u$*b*`) ? (`*a*`-`*b*`) : 0`

in conjunction with

`(`*p* `?` *x* `:` *y*`) + `*c* ⇒ *p* `?` *x*`+`*c* `:` *y*`+`*c* .

The above technique is not suitable for multi-word arithmetic, which also uses the carry flag. Multi-word arithmetic could be handled by high level patterns.

The reader may question why the semantics for flags are not substituted directly into expressions for the branch conditions directly to derive the high level branch condition from first principles each time. For example, the instruction commonly known as

```
            if (r > n)
               r = n;              /* Make sure r <= n */
```

(a) Source code

```
80483b9 8b 55 08  mov  0x8(%ebp),%edx   ; edx stores n and num
80483bc 89 fb     mov  %ecx,%ebx        ; ebx = r
80483be 2b 5d 08  sub  0x8(%ebp),%ebx   ; ebx = r-n, carry set if r<n
80483c1 19 c0     sbb  %eax,%eax        ; eax = -1 if r<n else = 0
80483c3 21 d8     and  %ebx,%eax        ; eax = r-n if r<n else = 0
80483c5 03 45 08  add  0x8(%ebp),%eax   ; eax = r-n=r if r<n, else = n
```

(b) Machine code

```
80483b9 13 edx := m[ebp + 8]
80483bc 14 ebx := ecx
80483be 15 tmp1 := ebx
        16 ebx := ebx - m[ebp + 8]
        17 %flags := SUBFLAGS(tmp1, m[ebp + 8], ebx)
80483c1 18 tmp1 := ...
        19 eax := 0 - %CF
        20 %flags := SUBFLAGS(...)
80483c3 21 eax := eax & ebx
        22 %flags := LOGICALFLAGS(eax)
80483c5 23 tmp1 := eax
        24 eax := eax + m[ebp + 8]
        25 %flags := ADDFLAGS(...)
```

(c) IR before propagation

```
80483b9 13 edx := m[esp0 + 4]
80483bc 14 ebx := ecx
80483be 15 tmp1 := ecx
        16 ebx := ecx - m[esp0+4]
        17 %flags := SUBFLAGS32(ecx, m[esp0+4], ebx)
80483c1 18 tmp1 := eax
        19 eax := 0 - (edi <u m[esp0+4])
        20 %flags := SUBFLAGS(...)
80483c3 21 eax := (0 - (ecx <u m[esp0+4])) & ebx
        22 %flags := LOGICALFLAGS(...)
80483c5 23 tmp1 := ...
        24 eax := ((0 - (ecx <u m[esp0+4])) & ebx) + m[esp0+4]
        25 %flags := ADDFLAGS( ...  )
```

(d) IR after propagation

Figure 3.10: Code from the running example where the carry flag is used explicitly.

```
edx = param1;                    // edx := n
ebx = param2 - param1;           // ebx := r-n
eax = 0 - (param2 < param1);     // eax := r<n ? -1 : 0
eax = (eax & ebx) + param1;      // eax := r<n ? r : n
esi = param1 - eax;              // esi := n-r if n<r
   ...
   do{ ...
      st = st * (float)edx_1 / (float)esi;    // Use esi as denom
```

(e) Generated C code.

Figure 3.10: (continued). Code from the running example where the carry flag is used explicitly.

"branch if greater or equals" actually branches if the sign flag is equal to the overflow flag. It is possible to substitute the expressions for %NF and %OF from Figure 3.8(b) to derive the condition `op1` $\geq_s$ `op2`, but the derivation is difficult. The distilled knowledge that the combination of `op1 - op2` (assuming flags are set) and JGE implies that the branch is taken when op1 $\geq_s$ `op2` is essentially "caching a difficult proof".

Once propagation of the `%flags` abstract location is performed, the assignment to `%flags` is unused, so that standard dead code elimination will remove it. In summary, expression propagation makes it easy to combine condition code setting and use. Special propagation or simplification rules extract the semantics of the condition code manipulations. Finally, dead code elimination removes machine dependent condition codes and the `%flags` abstract location.

3.3.2 x86 Floating Point Compares

Expression propagation can also be used to transform away machine details such as those revealed by older x86 floating point compare instruction sequences.

Modern processors in the x86 series are instruction set compatible back to the 80386 and even the 80286 and 8086, where a separate, optional numeric coprocessor was used (80387, 80287 or 8087 respectively). Because the CPU and coprocessor were in separate physical packages, there was limited access from one chip to the other. In particular, floating point branches were performed with the aid of the `fnstsw` (floating point store status word) instruction, which copied the floating point status word to the `ax` register of the CPU. Machine code for the floating point compare of Figure 3.2 is shown in Figure 3.11.

The upper half of the `16-bit` register `ax` is accessible as the single byte register, `ah`. The floating point status bits are masked off with the `test $0x45,%ah` instruction.

```
int i = (int) res;     /* Integer result; truncates */
if (res - i > 0.5)
   i++;                /* Round up */
```

(a) Source code

```
; res-i is on the top of the floating point stack
08048405 d9 05 60...flds   0x8048568 ; Load 0.5
0804840b d9 c9      fxch   %st(1)    ; Exchange:  TOS=0.5
0804840d da e9      fucompp          ; Floating compare 0.5 to res-i
0804840f df e0      fnstsw %ax       ; Store FP condition codes in ax
08048411 f6 c4 45   test   $0x45,%ah ; Test all FP flags
08048414 75 01      jne    8048417   ; Jump if integer nz ⟺ if FP ≥
```

(b) Disassembly

Figure 3.11: 80386 code for the floating point compare in the running example.

The expression propagated into the branch is `(SETFFLAGS(`*op1*`,` *op2*`, res) & 0x45) != 0`. Similar logic to the integer propagation is used to modify this to *op1* $>=_f$ *op2*, where $>=_f$ is the floating point greater-or-equals operator. The result is a quite general solution that eliminates most of the machine dependent details of the above floating point compare code. The `test $0x45,%ah` is often replaced with sequences such as `and $0x45,%ah`; `xor $0x40,%ah` or `and $1,ah`. Some compilers emit a `jp` instruction (jump if parity of the result is even) after a `test` or `and` instruction. This can save a compare instruction. For example, after `test $0x41,%ah` and `jp` *dest*, the branch is not taken if the floating point result is less (result of the test instruction is 0x01) or equal (0x40), but not uncomparable (0x41). (Two floating point numbers are uncomparable if one of them is a special value called a NAN (Not A Number).) The exact sequence of instructions found depends on the compiler and the floating point comparison operator in the original source code.

3.4 Summarising Calls

The effects of calls are best summarised by the locations modified by the callee, and the locations used before definition in the callee.

If data flow analysis is performed on the whole program at once, the effects of called procedures are automatically incorporated into their callers; Section 3.5 will consider this possibility. Assuming that the whole program is not analysed at once, a summary of some kind is required of the effects of calls.

Information has to be transmitted from a caller to and from a callee (e.g. the number

and types of parameters), and this information may change as the decompilation proceeds. This raises some issues with respect to the context of callers and callees, which will be considered after the terminology definitions.

Finally, the various call-related quantities of interest can be concisely defined by a set of data flow equations.

3.4.1 Call Related Terminology

The semantics of call statements and their side effects necessitates terminology that extends terms used by the compiler community.

Call instructions, and the call statements that result from them in a decompiler's intermediate representation, are complex enough to warrant some special terminology. By comparing the source and machine code in Figures 3.2 and 3.3, the instructions at addresses 80483b9 and 80483bc are seen to load **parameters** from the stack and from a register respectively. To prevent confusion, a strict distinction is made between the formal parameters of a procedure and actual **arguments** of a call statement. Instructions not shown compute a value in register `edx`, copy it to register `eax`, and the caller uses the value in register `eax`.

Since both `edx` and `eax` are modified by the procedure, they are called **modifieds**. The value or values returned by a procedure are called **returns** (used as a noun). It is possible at the machine language level that more than one value is returned by a procedure, hence the plural form. Unlike the return *values* of high level languages, at the machine code level, returns are assignments. Thus, there are **return locations** (usually registers) and **return expressions** (which compute the return values). In this program, only the value in `eax` is used, so only `eax` is called a **return** of the procedure *comb* and a **result** of the call to it. If and only if a call has a result will there be either an assignment to the result in the decompiled output, or the call will be used as a parameter to another call.

Figure 3.12 shows IR for part of the running example, illustrating several of the terms defined above.

The concept of parameters and returns as special features of a procedure is almost entirely nonexistent at the machine code level. The two loads from the parameters are similar to a load from a local variable and a register to register copy. Assignments to return locations are indistinguishable from ordinary assignments. The fact that register `eax` is used by some callers, and register `edx` is not used by any callers, is not visible at all by considering only the IR of the called procedure. The locations used to store

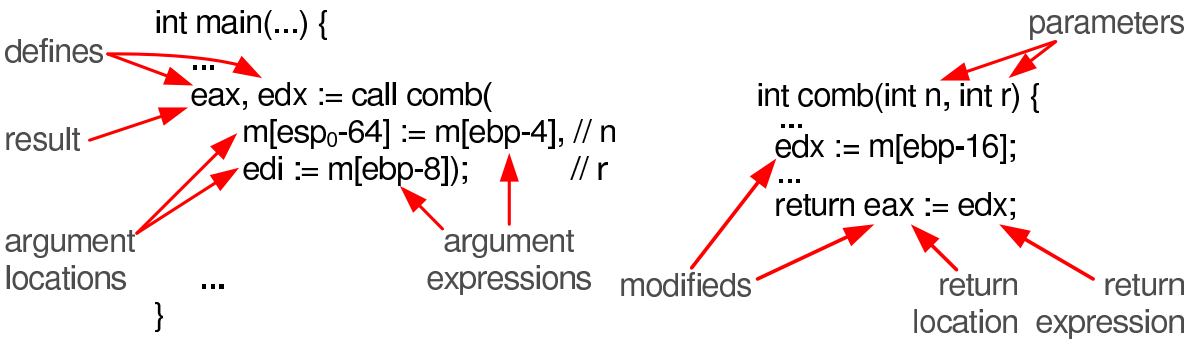


Figure 3.12: Intermediate representation illustrating call related terminology.

parameters and return locations are a matter of convention, called the Application Binary Interface (ABI); there are many possible conventions. While most programs will follow one of several ABI conventions, some will not follow any, so a decompiler should not assume any particular ABI.

Most of the time, a location that is used before it is defined implies that it is a parameter. However, in the example of Figure 3.3, registers `esi` and `ebx` are used by the `push` instructions, yet they are not parameters of the procedure, since the behaviour of the program does not depend on the value of those registers at the procedure's entry point. The registers `esi` and `ebx` are said to be *preserved* by the procedure.

Register `ecx` is also preserved, however the procedure does depend on its value; in other words, `ecx` is a preserved parameter. Care must be taken to separate those locations used before being defined which are preserved, parameters, or both. This can be achieved by a combination of expression propagation and dead code elimination.

For example, consider statements 5, 7 and 9 of Figure 3.4. These save the preserved registers, and are ultimately dead code. They are needed to record the fact that *comb* preserves `esi`, `ebx` and `ecx`, but are not wanted in the decompiled output. When the preservation code is eliminated, the only remaining locations that are used before definition will be parameters.

Reference parameters are an unusual case. Consider for example a procedure that takes only `r` as a reference parameter. At the machine code level, its address is passed in a location, call it `ar`. In a sense, the machine code procedure has two parameters, `ar` (since the address is used before definition) and `r` itself, or `*ar` (since the referenced object and its elements are used before being defined). However, the object is by definition in memory. There is no need to consider the data flow properties of memory objects; the original compiler (or programmer) has done that. Certainly it is important to know whether `*ar` is an array or structure, so that its elements can be referenced appropriately, but this can be done by defining `ar` with the appropriate type (e.g. as being of type Employee* rather than void*). It is quite valid for the IR to keep the

parameter as a pointer until very late in the decompilation. It could be left to the back end or even to a post decompilation step to convert the parameter to an actual reference, since not all languages support references (e.g. C).

3.4.2 Caller/Callee Context

Memory and register expressions are frequently communicated between callers and callee(s); the difference in context requires some substitutions to obtain expressions which are valid in the other context.

When the callee for a given caller is known, the parameters are initially the set of variables live at the start of the callee. These locations are relative to the callee, and not all can be used at the caller without translation to the context there. Figure 3.13 shows the call from *main* to *combs* in the running example.

```
804847e 47 ecx := m[esp0-12]          ; copy r
8048481 48 esp := esp0 - 64
        49 m[esp0-64] := m[esp0-8]    ; copy n
8048489 51 esp := esp0 - 68
        52 m[esp0-68] := %pc          ; Save program counter
        54 CALL comb
```

(a) Caller (`main`)

```
80483b9 13 edx := m[esp0+4]       ; load n
80483bc 14 ebx := ecx             ; load r
```

(b) Callee (`comb`)

Figure 3.13: Caller/callee context.

The first argument, n, is passed in memory location `m[esp0-64]`, while r is passed in register `ecx`. In the callee, parameter n is read from `m[esp0+4]`, while r is read from register `ecx`. (The memory location `m[esp0]` holds the return address in an x86 procedure). Note that these memory expressions are in terms of `esp0`, the stack pointer on entry to the procedure, but the two procedures (*main* and *comb*) have different initial values for the stack pointer. Without expression propagation to canonicalise the memory expressions in the callee, the situation is even worse, since parameters could be accessed at varying offsets from an ever changing stack pointer register, or sometimes at offsets from the stack pointer register and at other times at offsets from the frame pointer register (if used).

Since it will be necessary to frequently exchange location expressions between callers and callees , care needs to be taken that this is always possible. For example, if register `ecx` in *main* is to be converted to a local variable, say *local2*, it should still be accessible as the

original register `ecx`, so that it can be matched with a parameter in callees such as *comb*. Similarly, if `ecx` in *comb* is to be renamed to *local0*, it should still be accessible as `ecx` for the purposes of context conversion. Similar remarks apply to memory parameters such as `m[esp0-64]` in *main* and `m[esp0+4]` in *comb*. One way to accomplish this is to not explicitly rename a location to an expression representing a local variable, but rather maintain a mapping between the register or memory location and the local variable name to be used at code generation time (very late in the decompilation). Most of the time the IR is used directly, but at code generation time and possibly in some IR display situations the mapped name is used instead.

Note that while arguments and the values returned by procedures can be arbitrary expressions, the locations that they are passed in are usually quite standardised. For example, parameters are usually passed in registers or on the stack; return values are usually located in registers. Hence, as long as registers are not renamed, the main problem is with the stack parameters, which are usually expressions involving the stack pointer register.

Consider argument *n* in main in Figure 3.13. It has the memory expression `m[esp0-64]`. Suppose we know the value of the stack pointer at the call to *comb*; here it is $\texttt{esp}_{call}$ = `esp0-68` (in the context of *main*; in other words, the value of the stack pointer at the call is 68 bytes less than the initial value at the start of *main*). This is reaching definitions information; it turns out to be useful to store this information at calls (for this and several other purposes). It is assumed that if the caller pushes the return address, this is part of the semantics of the caller, and has already happened by the time of the call. The call is then purely a transfer of control, and control is expected to return to the statement following the call.

Hence, `esp0` (at the start of *comb*) = $\texttt{esp}_{call}$`+68`. Substituting this equation into the memory expression for `n` in main (`m[esp0-64]`) yields `m[`$\texttt{esp}_{call}$ `+ 68 - 64]` = `m[`$\texttt{esp}_{call}$ `+ 4]`. Since the value for the stack pointer at the call is the same as the value at the top of *comb*, this is the memory expression for `n` in terms of `esp0` in *comb*, in other words, in the context of *comb* this expression is `m[esp0+4]`, just as the IR in Figure 3.13(b) indicates. Put another way, it is possible to generate the expression (here `m[esp0+4]`) that is equivalent to the expression valid in *main* (here `m[esp0-64]`) for a stack location, so that it is possible to search through the IR for a callee to find important information. A similar substitution can be performed to map from the callee context to the caller context. When the number of parameters in a procedure changes, machine code decompilers need to perform this context translation to adjust the number of actual arguments at each call.

3.4.3 Globals, Parameters, and Returns

Three propositions determine how registers and global variables that are assigned to along only some paths should be treated.

Figure 3.14 shows parts of the running example extended with some assignments and uses of global variables. It shows a potential problem that arises when locations are assigned to on some but not all paths of a called procedure.

```
int g; register int edi; /* Global memory and register */
int comb(int n, int r) {
   ...
   int c = n-r;
   if (c == 0) {
      g = 7; edi = 8;    /* Possibly modify globals */
   } ...
}
int main() {
   g=2; edi=3;            /* Assign to globals */
   ...comb(n, r);
   printf("g = %d, edi = %d\n", g, edi);
}
```

Figure 3.14: Potential problems caused by locations undefined on some paths through a called procedure.

In this example, `g` and `edi` are global variables, with `edi` being a register reserved for this global variable. Both `g` and `edi` are found to be modified by *comb*. If the assumption is made that locations defined by a call kill reaching definitions and livenesses, the definitions of `g` and `edx` in `main` no longer reach the use in the `printf` call. As a result, the definitions are dead and would be removed as dead code. This is clearly wrong, since if `n`≠`r` and `c`≠`0`, then the program should print "`g = 2, edi = 3`" rather than undefined values.

One of these global variables is a register, represented in the decompiled output as a local variable. It could also be represented by a global variable, but at the machine code level, there are few clues that the register behaves like a global variable. The two ways that the modification of a local variable can be communicated to its callers are through the return value, or through a reference parameter. A reference parameter is logically equivalent to the location being both a parameter and a return of the procedure. In a decompiler, it is convenient to treat parameters as being passed by value, so that arguments of procedure calls are pure uses, and only returns define locations at a call statement.

Location `edi` therefore becomes a return of procedure *comb*. Since *comb* already returns `i`, this will be the second return for it. Local variable `edi` is not defined before use on the path where `n=r`, in other words the value returned in `edi` by `comb` depends on the value before the call, hence `edi` also becomes a parameter of *comb*.

Conveniently, making `edi` an actual argument of `comb` means that the definition of `edi` is no longer unused (it is used by the one of the arguments of the call to `comb`).

Unfortunately, this solution is not suitable for the global variable, `g`. In the example, there would be code similar to `g = comb(g, ...)`, and the parameter would shadow the global inside the procedure `comb`. This has the following problems:

- It is inefficient allocating a parameter to shadow the global variable, copying the global to the parameter, from the parameter to the return location, and from the return location back to the global.
- Manipulating global variables in this fashion is unnatural programming; programmers do not write code this way, so the result is less readable.
- The global may be aliased with some other expression (e.g. another reference parameter, or the dereference of a pointer). When the aliased location is assigned to, the global will change, but not the parameter.

The problem can be solved by adopting a policy of never allowing globals to become parameters or returns, and never eliminating definitions to global variables. The latter is probably required for alias safety in any case. The above discussion can be summarised in the following propositions:

Proposition 3.1: *Locations assigned to on only some paths become parameters and returns.*

Proposition 3.2: *Global variables are never parameters or returns.*

Proposition 3.3: *Assignments to global variables should never be eliminated.*

Using these extra constraints causes the definitions for `g` and `edi` of Figure 3.14 to remain, although for different reasons. The program decompiles correctly, as shown in Figure 3.15.

The decompiled program does not retain `edi` as a global register. It could be argued that a decompiler should recognise that such variables are global variables. The only clue at the machine code level that a register is global would be if it is not defined before being passed to every procedure that takes it as a parameter, perhaps initialised

```
int g;     /* Global memory */
int comb(int n, int r, int* edi) {
   ...
   int c = n-r;
   if (c == 0) {
      g = 7; *edi = 8;    /* Possibly modify ‘‘globals’’ */
   } ...
}
int main() {
   g=2;                    /* Initialise the global variable */
   int edi=3;              /* edi is a local variable now */
   ...comb(n, r, &edi);    /* Pass edi ‘‘by reference’’ */
   printf("g = %d, edi = %d\n", g, edi);
}
```

Figure 3.15: Solution to the problem of Figure 3.14.

in a compiler generated function not reachable from *main*. In the above example, this does not apply since the parameter is defined before the only call to *comb*.

Decompilation can sometimes expose errors in the original program. For example, a program could use an argument passed to a procedure call after the call, assuming that the argument is unchanged by the call. This could arise through a compiler error, or a portion of the program written in assembly language. The decompiled output will make the parameter a reference parameter, or add an extra return, which is likely to make the error more visible than in the original source code. In a sense, there is no such thing as an error in the original program as far as a decompiler is concerned; its job is to reproduce the original program, including all its faults, in a high level language.

3.4.4 Call Summary Equations

The calculation of call-related data flow elements such as parameters, defines, and results can be concisely expressed in a series of data flow equations.

Proposition 3.2 states that global memory variables are not suitable as parameters and returns. This proposition motivates the definition of *filters* for parameters and returns:

$$\textit{param-filter} = \{\textit{non-memory}\ \ \textit{expressions} + \textit{local-variables}\} \tag{3.4}$$

$$\textit{ret-filter} = \{\textit{non-memory}\ \ \textit{expressions}\} \tag{3.5}$$

These filters can be thought of as sets of expressions representing locations, which can be intersected with other sets of locations to remove unsuitable locations. Locations

that satisfy the filter criteria (i.e. are not filtered out) are referred to as *suitable*. Local variables are not filtered from potential parameters for the following reason. Some parameters are passed on the stack; architectures that pass several parameters in registers pass subsequent parameters, if any, on the stack. Stack locations used for parameters are equivalent to local variables in the caller. They are both located at offsets (usually negative) from the stack pointer value on entry to the procedure (`esp0` in the running example). A machine code decompiler will typically treat local variables and stack locations used to pass procedure arguments the same, and eliminate assignments to the latter as dead code. For simplicity, no distinction will be made between the two types of variable here.

The main data flow sets of interest concerning procedures are as follows.

- Potential parameters of a procedure p are live variables at the procedure entry. They are *potential* parameters, because preserved locations often appear to be parameters when they are not. Such locations will be removed from the parameters with a later analysis, described in Section 4.3.

$$\textit{init-params}(p) = \textit{live-on-entry}(p) \cap \textit{param-filter} \tag{3.6}$$

- Initial argument locations of a call c are the potential parameters of the callee, translated to the caller context as discussed in Section 3.4.2:

$$\textit{argument-locations}(c) = \textit{params}(\textit{callee}) \tag{3.7}$$

  Here *params* will be either *init-params,* or *final-params* if available. Note that while an argument location may be a register or memory location, the actual argument may be any expression such as a constant or a sum of two locations. Arguments are best represented as a kind of assignment, with a left hand side (the argument location) and a right hand side (the actual argument expression). The argument location has to be retained for translating to and from the callee context, as mentioned earlier. By contrast, parameters are represented by only one location, the register or memory location that they are passed in.

- *Modifieds* of a procedure are the definitions reaching the exit, except for preserved locations. This exception is explained in detail in Section 4.3. Modifieds are a superset of returns: not all defined locations will be used before definition in any caller.

$$\mathit{modifieds}(p) = \textit{reach-exit}(p) - \mathit{preserveds}(p) \tag{3.8}$$

- *Defines* of a call are the definitions reaching the exit of the callee:

$$defines(c) = modifieds(callee) \tag{3.9}$$

- *Results* of a call are the suitable defines that are also live at the call:

$$results(c) = defines(c) \cap \textit{ret-filter} \cap live(c) \tag{3.10}$$

- *Returns* of a procedure are the suitable locations modified by the procedure, which are live at some call or calls to that procedure:

$$\textit{return-locations}(p) = modifieds(p) \cap \textit{ret-filter} \cap \bigcup_{c\ calls\ p} live(c) \tag{3.11}$$

  Returns, like arguments, are best represented as assignments, with a left hand side containing the return location, and the right hand side being the actual return expression. Results, like parameters, are represented by only one expression, the location that they are "passed" in.

In all but the first case, context translation from the callee to the caller or vice versa is required.

Potential return locations are based on reaching definitions at the exit of the procedure. It might be thought that available definitions (statements whose left hand side is defined on all paths) should be used instead of reaching definitions, since it seems reasonable that a return value should be defined along all paths in a procedure. However, as shown in the example of Figures 3.14 and 3.15, suitable locations that are defined along only some paths become returns and parameters, or equivalently reference parameters.

The above equations assume that a callee procedure can be found for each call. In some cases, this is not true, at least temporarily. For example, an indirect call instruction may not yet be resolved, or recursion may prevent knowledge of every procedure's callees. In these cases, the conservative approximation is made that the callee uses and defines all locations. For such *childless calls*, denoted as *cc*:

- Argument locations are all definitions reaching the call:

$$\textit{argument-locations}(cc) = reach(cc) \tag{3.12}$$

- Defines and results are all locations live at the call:

$$defines(cc) = results(cc) = \textit{live-at-call}(cc) \tag{3.13}$$

where *live-at-call(cc)* is the set of locations live at the childless call statement *cc*.

3.4.5 Stack Pointer as Parameter

The stack pointer, and occasionally other special pointers, can appear to be a parameter to every procedure, and is handled as a special case.

The stack pointer register is used before it is defined in all but the most trivial of procedures. For all other locations, use before definition of the location (after dead code is eliminated) implies that the location is a parameter of the procedure. In the decompiled output, the stack pointer must not appear as a parameter, since high level languages do not mention the stack pointer. It is constructive to consider why the stack pointer is an exception to the general parameter rule, and whether there may be other exceptions.

The value of the stack pointer on entry to a procedure affects only one aspect of the program: which set of memory addresses are used for the stack frame (and the stack frames for all callees). By design, stack memory is not initialised, and after the procedure exits, the values left on the stack are never used again. The stack is therefore a variable length array of temporary values. The value of the stack pointer at any instant is not important; only the relative addresses (offsets) within the stack array are important, to keep the various temporary values separated from one another, preventing unintended overwriting.

The program will run correctly with any initial stack pointer value, as long as it points to an appropriately sized block of available memory. In this sense, the program does not depend on the value of the stack pointer. This is the essence of the exception to the general rule.

Some programs reference initialised global memory through offsets from a register. One example of this are Motorola 68000 series programs which use register `A5` as the pointer to this data. Another example are x86 programs using the ELF binary file format. These programs use the `ebx` register to point to a data section known as the Global Offset Table (GOT). These special registers would also appear to be parameters of all procedures, yet the specific value of these registers is unimportant to the running of the program, and should not appear in the decompiled output. In these cases, however, it is probably better for the front end of the decompiler to assign an arbitrary value to these special registers. Doing this ensures that all accesses to scalar global variables are of the form `m[K]` where `K` is a constant. If this is done, the question of whether the special registers should be removed as parameters disappears: propagation and simplification will transform all the scalar memory addresses to the `m[K]` form.

It might be considered that giving the stack pointer an initial arbitrary value would avoid the need for any exception. However, when a procedure is called from different

points in the program, it can be passed different values for that procedure's initial stack pointer. Worse, in a program with recursion, the value of the stack pointer depends on the dynamic behaviour of the program. Rather than causing the stack pointer to take on simple constant values, such a scheme would, for many programs, generate extremely complex expressions for each stack memory reference.

In summary, the stack pointer is the only likely exception to the general rule that a location used before definition after dead code elimination in a procedure implies that the location is a parameter.

3.5 Global Data Flow Analysis

Decompilers could treat the whole program as one large, global (whole-program) data flow problem, but the problems with such an approach may outweigh the benefits.

Some analyses are approximate until global analyses can be performed over the IR for all procedures. In the running example, the procedure *comb* modifies registers `eax` and `edx`. It is not possible to know whether these modified registers are returning a value until all callers are examined. Imagine that *comb* is used in a much larger program, and there are a hundred calls to it. No matter what order the callers are processed, it could be the case that the last caller processed will turn out to be the first one to use the result computed in (e.g.) register `edx`. Other analyses also require the IR for all procedures. For example, type analysis is global in the sense that a change to the type of a parameter or return can affect callers or callees respectively.

Since the IR for all procedures has to be available for a complete decompilation, one approach would be to treat the whole program as a single whole-program data flow problem, with interprocedural edges from callers to callees, and from the exit of callees back to the basic block following the caller. The solution of data flow problems is usually iterative, so a global data flow analysis would automatically incorporate definitions and uses from recursive callees.

As noted by Srivastava and Wall [SW93], a two phase algorithm is required, to prevent impossible data flow (a kind of "leakage" of data flow information from one call to another). So-called "context insensitive" interprocedural data flow ignores this problem, and suffers a lack of precision as a result [YRL99]. Figure 3.16 shows a program with two procedures similar to *main* in the running example (*main1* and *main2*) calling *comb*, which is the same as in the running example. This program illustrates context sensitivity.

Two callers are passing different arguments, say `comb(6, 4)` and `comb(52, 5)`. The

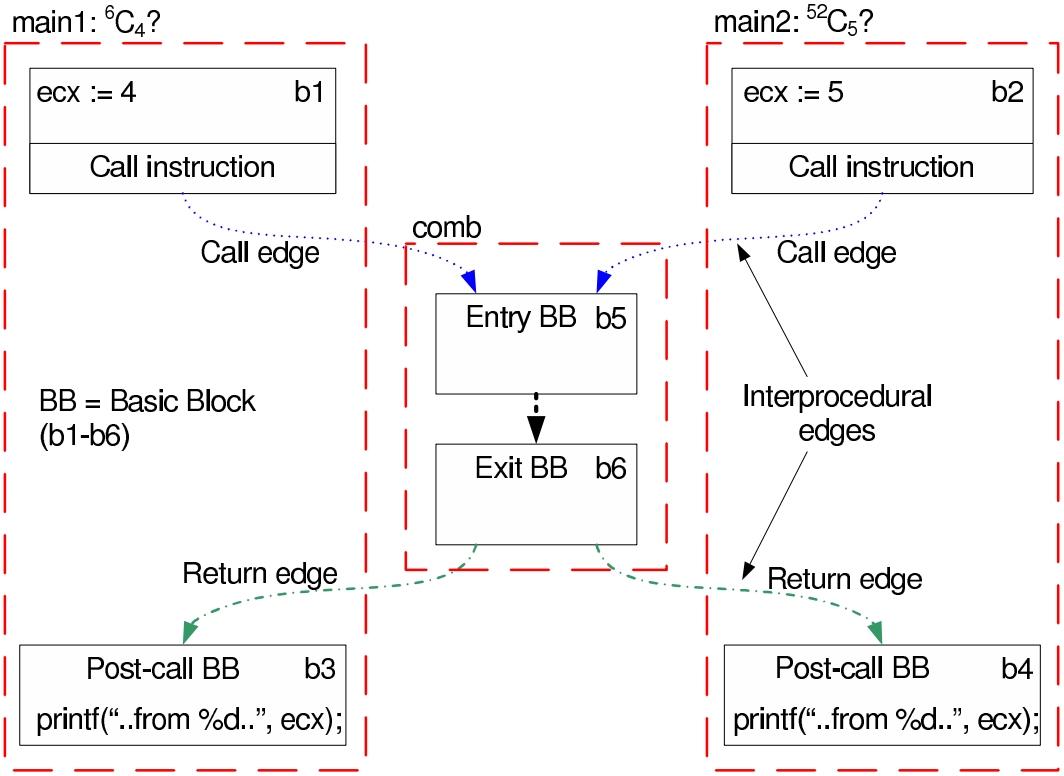


Figure 3.16: Interprocedural data flow analysis. Adapted from [SW93].

second argument, passed in register `ecx`, is 4 for the first caller, and 5 for the second. Without the two stage approach, all interprocedural edges are active at once, and it is possible for data flow information to traverse from basic block *b1* (which assigns 4 to `ecx`), through procedure *comb*, to basic block *b4*, which prints among other information the value of `ecx` to caller 2, which should see only the definition in b2 reaching that use (with the value 5). Similarly, data flow information from *b2* can imprecisely flow to *b3*.

The solution to this lack of precision requires that the data flow be split into two phases. Before the start of the first phase, some of the interprocedural edges are disabled. Between the first and second phases, these edges are enabled and others are disabled. In [SW93], only backward-flow problems were considered, but the technique can be extended to forward-flow problems as well.

Factors not favouring global data flow analysis include:

- Resources: the interprocedural edges consume memory, and their activation and deactivation consumes time. In addition, there are many more definitions for call parameters, since they are defined at every call. Every parameter will typically have from two to several or even hundreds of calls. Special locations such as the stack pointer will typically have thousands of definitions.

- There is no need for the two phase algorithm in the non-global analysis, with its detail, and time and space costs.
- Since some programs to be compiled require a large machine to execute them, and decompilation requires of the order of 20 times as much memory as the original executable, it would be useful to be able to split the problem of decompiling large programs so that parts of it can be run in parallel with cluster or grid computing. This is more difficult with the global approach.
- Procedures are inherently designed to be separate from their callers; the non-global approach seems more natural.
- The number of definitions for a parameter can become very large if a procedure is called from many points in the program. So there can be problems scaling the global approach to large programs.

Factors that do not significantly favour one approach over the other include:

- Potential parameters are readily identified in the non-global approach as those that have no definition. However, with the global approach, parameters could be identified as locations that have some or all definitions in procedures other than the one they are used in.
- The saving and restoration of preserved locations become dead code when considering procedures in isolation. This effectively solves the problem of preserved locations becoming extra parameters and returns, although recursion causes extra problems (Sections 4.3 and 4.4). With the global approach, definitions only used outside the current procedure (returns are used by the return statement in the current procedure) could be eliminated as dead code.

Factors favouring the global approach include:

- Simplicity: the data flow problems are largely solved by one, albeit large, analysis.
- There is no need to summarise the effects of calls, and any imprecision that may be associated with the summaries.
- There are problems that result from recursion, which do not arise with the global approach. However, these are solvable for the non-global approach, as will be shown in Section 4.4.
- Information has to be transmitted from the callee context to the caller context, and vice versa, in the non-global approach. Section 3.4.2 showed that this is easily performed.

3.6 Safe Approximation of Data Flow Information

Whether over- or under-estimation of data flow information is safe depends on the application; for decompilers, it is safe to overestimate both definitions and uses, with special considerations for calls.

Some aspects of static program analysis are undecidable or difficult to analyse, e.g. whether one branch of an `if` statement will ever be taken, or whether one expression aliases with another. As a result, there will always be a certain level of imprecision in any data flow information. Where no information is available about a callee, the lack of precision of data flow information near the associated call instructions could temporarily be extensive, e.g. every location could be assumed to have been assigned to by the call.

Since approximation is unavoidable, it is important to consider safety, in the sense of ensuring that approximations lead at worst to lower readability, and not to incorrect output (programs whose external behaviour differs from that of the input binary program).

Less readable output is obviously to be preferred over incorrect output. In many cases, if the analyses are sophisticated enough, low quality output will be avoided, as temporarily prevented transformations could later be found to be safe, excess parameters can be removed by a later analysis, and so on. Where information is incomplete, approximations are necessary to prevent errors. Errors usually cannot be corrected, but approximations can be refined.

The main question to be answered in this section is whether an over- or under-estimate of elementary data flow operations (e.g. definitions, parameters, etc.) is safe. For example, definitions over which there is some uncertainty could be counted as actual definitions (over-estimating definitions), or not (under-estimating definitions). Definitions could also be divided into three categories: must-define, does-not-define, and may-define. Using the three categories is more precise, but consumes more memory and complicates the analyses.

An expression can only be safely propagated from a source statement to a destination statement if there are no definitions to locations that are components of the source statement (Section 3.1). If definitions are underestimated (the algorithm is not aware of some definitions), then a propagation could be performed which does not satisfy the second condition. Hence, underestimating definitions is unsafe for propagation. By contrast, overestimating definitions could prevent some propagations from occurring, or at least be delayed until later changes are made to the IR of the program. This will potentially cause lower quality decompilations.

Definitions that reach the exit of a procedure are potential return locations. Underestimating definitions would cause some such definitions to not become returns, resulting in some semantics of the original program that are not contained in the decompiled output. Overestimation of definitions may cause extra returns, which is safe.

Locations that are live (used before being defined) are generally parameters of a procedure (an exception will be discussed later). Underestimating uses would result in too few parameters for procedures, hence underestimating uses is unsafe. Overestimating uses may result in overestimating livenesses, which may result in unnecessary parameters in the generated output; this is less readable but safe.

Dead code elimination depends on an accurate list of uses of a definition. If uses are overestimated, then some statements that could be deleted would be retained, resulting in lower quality but correct decompilations. If uses are underestimated, definitions could be eliminated when they are used, resulting in incorrect output. Hence, underestimation of uses is also unsafe for dead code removal.

Overall, an overestimate of definitions and uses is safe, and an underestimation of either is unsafe. However, overestimation of one quantity can lead to underestimation of another. In particular, definitions kill liveness (the state of being live), leading to potentially underestimated uses. Figure 3.17(a) shows a definition and use with a call between them; the call has an imprecise analysis which finds that x may be defined by the call when in reality it is not; it is approximated to be definitely defined (its definitions are overestimated). The arrows represent the liveness of x "flowing" from the use to the definition. This situation arises commonly when it is not known whether a location is preserved or not, either because the preservation analysis has failed, or because the analysis can not be performed yet (e.g. as a result of recursion).

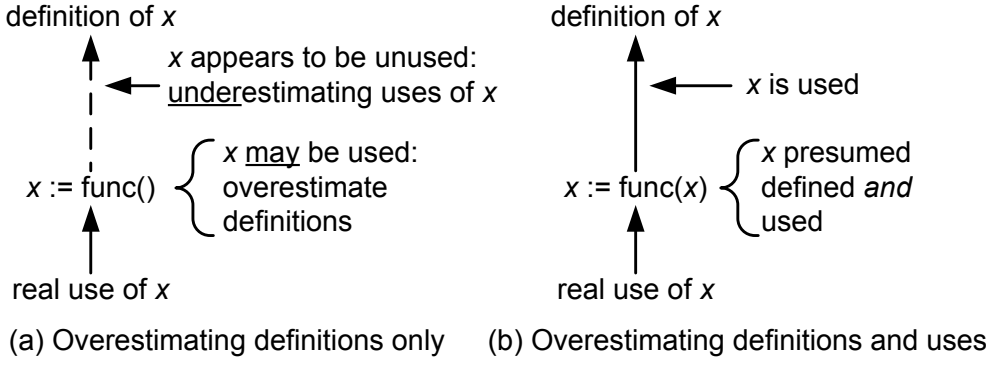


Figure 3.17: Transmission of liveness from use to definition through a call.

With the data flow as shown in Figure 3.17(a), the definition of x appears to be unused, hence it appears to be safe to eliminate the definition as dead code. In reality, x is used and the removal would be incorrect. The solution to this problem is as follows: inside

calls, for every non-global location whose definition is overestimated (such as x in the figure), its uses must be also be overestimated. In Figure 3.17(b), x is an argument of the call as well as a return, hence the original definition of x is still used, and it will not be eliminated.

This is similar to the problem of Figure 3.14; in that example, the location was known to be defined but only along some paths. If x is a global variable, the above would not apply, and the uses of x would in fact be underestimated above the call. However, by Proposition 3.3, definitions of globals are never eliminated, hence the underestimation is safe. Global variables must not be propagated past calls to procedures that may define them, but this is guaranteed by the fact that such globals appear in the modifieds list for the call. However, these modifieds never become returns because of the return filter of Equation 3.5.

Parameters act as definitions when there is no explicit definition before a use in a procedure. As shown in Figure 3.18, a similar situation exists with procedure parameters, and the solution is the same: overestimate the uses as well as the definitions in the call by making x an argument as well as a return of the call to *func*.

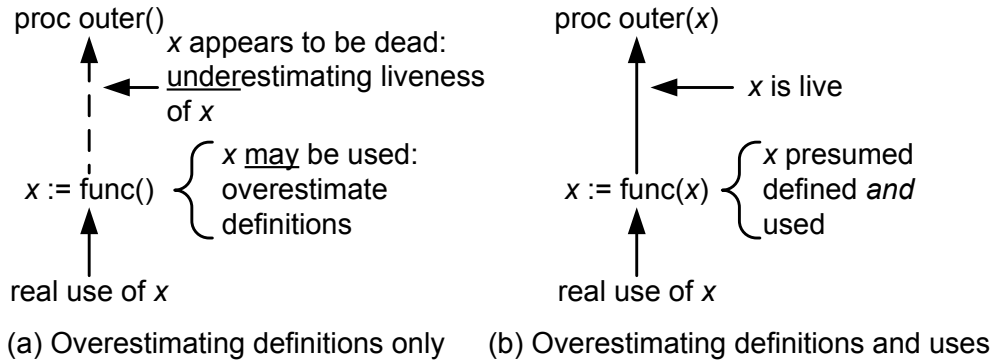


Figure 3.18: Transmission of liveness from use to parameter through a call.

3.7 Overlapped Registers

Overlapped registers are difficult to handle effectively in the intermediate representation, but representing explicit side effects produces a correct program, and dead code elimination makes the result readable.

CISC architectures (Complex Instruction Set Computers) often provide the ability to operate on operands of various sizes, e.g. 8-/16-/32- and even 64-bit values. Figure 3.19 shows the x86 register `eax`, whose lower half can be accessed as the register `ax`, and both halves of `ax` can be accessed as `al` (low half) and `ah` (high half). In the

x64 architecture, all these registers overlap with the 64-bit `rax` register. Three other registers (`ebx`, `ec`*x*, and `edx`) have similar overlaps, while other registers have overlaps only between the 32-bit register and its lower half (e.g. `ebp` and `bp`).

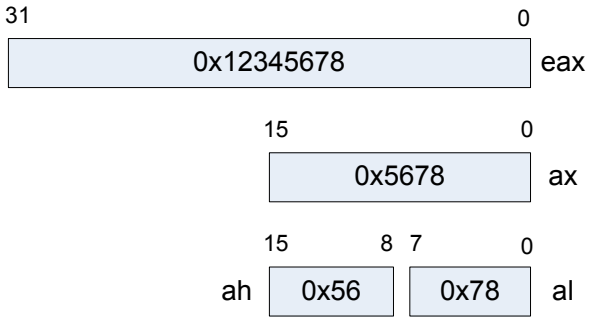


Figure 3.19: Overlapped registers in the x86 architecture.

The Motorola 68000 family of processors has a similar situation, although only the lower parts (lower 16 bits, or lower 8 bits) can be accessed. In the 6809 and 68HC11 series, both halves of the D (double byte) register can be accessed as the A (upper) and B (lower) accumulators.

Some RISC architectures have a similar problem. For example, SPARC double precision registers use pairs of single precision registers, and typical 32-bit SPARC programs use pairs of 32-bit loads and stores when loading or storing a 64-bit floating point register. However, this problem is different, in that the smaller register is the machine word size, and bit manipulation such as shifting and masking cannot be performed on floating point registers.

Since there are distinct names for the overlapped registers at the assembly language level, it seems natural to represent these registers with separate names in the intermediate representation (IR). However, this leads to a kind of aliasing at the register level; treating the registers as totally independent leads to errors in the program semantics.

The alternative of representing all subregisters with the one register in the IR also has problems. For example, `al` and `ah` may be used to implement two distinct 8-bit variables in the original source program. When accessing the variable assigned to `ah`, awkward syntax will result, e.g. `local3 >> 8` or `local3 & 0xFF`. Even worse, the data flow for the two variables will become conflated, so that assignment to one variable will appear to kill the other.

Another possibility is to declare registers in the decompiled output using overlapped source code constructs such as C unions or Fortran `common` statements. However, such output is quite unnatural, and represents low level information that does not belong in the decompiled output. Such a scheme is more suitable for a binary translator that uses a compiler as its back end (e.g. [CVE00]).

The best solution seems to be to treat the overlapped registers as separate, but to make the side effects explicit. This was first suggested for the Boomerang decompiler by Trent Waddington [Boo02]. Most such side effects can be added at the instruction decoding level, which is source machine dependent already. (An exception are results from call statements, which are not explicit at the assembly language level, and which may be added later in the analysis.) After any assignment to an overlapped register, the decoder can emit assignments representing the side effects. Considering the example register in Figure 3.19, there are four cases to consider:

- After an assignment to `eax`:
  - `ax := truncate(eax)`
  - `al := truncate(eax)`
  - `ah := eax@[8:15]` *`/* Select bits 8-15 from eax */`*
- After an assignment to `ax`:
  - `eax := (eax & 0xFFFF0000) | ax`
  - `al := truncate(ax)`
  - `ah := ax@[8:15]`
- After an assignment to `al`:
  - `eax := (eax & 0xFFFFFF00) | al`
  - `ax := (ax & 0xFF00) | al`
- After an assignment to `ah`:
  - `eax := (eax & 0xFFFF00FF) | ah ≪ 8`
  - `ax := (ax & 0x00FF) | ah ≪ 8`

Many of these have the potential of leading to complex generated source code, but dead code elimination will remove almost all such code. For example, where `al` and `ah` are used as independent variables, the larger registers `ax` and `eax` will not be used, so dead code elimination will simply eliminate the complex assignments to `ax` and `eax`. When operands are constants, the simplification process reduces something like `0x1234567@8:15` to `0x56` or plain decimal 86. Where the complex manipulations are not eliminated, the original source program must in fact have been performing complex manipulations, so the complexity is justified.

3.7.1 Sub-fields

Sub-fields present similar problems to that of overlapped registers.

Some high level languages, such as C, allow the packing of several structure fields into one machine word. Despite the fact that the names of such fields are related, in data flow terms, the variables are completely independent. As mentioned above, current data flow analyses will not treat them as independent, leading to less than ideal generated code. There is no reason that compilers for languages such as Pascal which support range variables (e.g. `0..255` or `100..107`) could not pack two or more such variables into a machine word. Such variables will also not be handled properly by a standard data flow analysis.

The solution to this problem is left for future work.

The PowerPC architecture has a 32-bit register containing 8 fields of four condition codes (8 condition code registers, named *cr0* - *cr7*). Instructions using the condition codes register specify which condition codes set they will refer to. This poses a similar problem, except that the way that the condition code register is represented in a decompiler is a design decision. In this case, it would appear to be better to implement the 32-bit condition code register as eight separate registers.

3.8 Related Work

The related work confirms that the combination of expression propagation and dead code elimination is a key technique in decompilation.

Johnstone and Scott describe a DSP assembly language to C reverse compiler called `asm21toc` [JS04]. They find that the combination of propagation and dead code elimination is also useful for their somewhat unusual application (DSP processors have some unusual features compared to ordinary processors). They do not appear to use any special IR; they parse the assembly language to produce low-level C, perform data flow analyses using standard techniques (presumably similar to those of [ASU86]), and use reductions to structure the some high level constructs. Their output typically requires editing before it can be compiled. They found that some 85% of status (flags) assignments could be eliminated, and typically 40% of other assignments.

Chapter 4

SSA Form

Static Single Assignment form assists with most data flow components of decompilers, including such fundamental tasks as expression propagation, identifying parameters and return values, deciding if locations are preserved, and eliminating dead code.

Chapter 3 listed many applications for data flow analysis, several of which were somewhat difficult to apply. This chapter introduces the Static Single Assignment form (SSA form), which dramatically reduces these difficulties. The SSA form also forms the basis for type analysis in Chapter 5 and the analysis of indirect jumps and calls in Chapter 6.

The SSA form is introduced in Section 4.1, showing that the implementation of many of the data flow operations in a machine code decompiler is simpler. Some problems that arise with the propagation of memory expressions are discussed in Section 4.2, while Section 4.3 introduces the important concept of preserved locations. The analysis of these is complicated by recursion, hence Section 4.4 discusses the problems caused by recursion, and presents solutions. At various points in the process of decompilation, it is convenient to store a snapshot of definitions or uses, and these are provided by collectors, discussed in Section 4.5. Sections 4.6 and 4.7discuss related work and representations, while Section 4.8 summarises the contributions of this chapter.

4.1 Applying SSA to Registers

SSA form vastly simplifies expression propagation, provides economical data flow information, and is strong enough to solve problems that most other analysis techniques cannot solve.

The Static Single Assignment (SSA) form is a form of intermediate representation which maintains the property that each variable or location is defined only once in

the program. Maintaining the program in this form has several advantages: analyses are simpler in this form, particularly expression propagation. For most programs, the size of the SSA representation is roughly linear in the size of the program, whereas traditional data flow information tends to be quadratic in memory consumption.

In order to make each definition of a location unique, the variables are "renamed". For example, three definitions of register `edx` could be renamed to $\texttt{edx}_1$, $\texttt{edx}_2$, and $\texttt{edx}_3$. The subscripts are often assigned sequentially as shown, but any renaming scheme that makes the names unique would work. In particular, the statement number or memory address of the statement could be used. SSA form assumes that all locations are initialised at the start of the program (or program fragment or procedure if analysing in isolation). The initial values of locations are usually given the subscript 0, e.g. $\texttt{edx}_0$ for the initial value of `edx`.

Since each use in a program has only one definition, a single pointer can be used from each use to the definition, effectively providing reaching definitions information (the one definition that reaches the use is available by following the pointer). Much of the time, the original definition does not have to be literally renamed; it can be assumed that each definition is renamed with the address of the statement as the "subscript". (For convenience, display of the location could use a statement number as the subscript). Sassa et al. report that renaming the variables accounts for 60-70% of the total time to translate into SSA form [SNK+03]. Implicit assignments are required for parameters and other locations that have uses but no explicit assignments; these are comparatively rare. This representation is used in the Boomerang decompiler [Boo02].

The algorithms for transforming ordinary code into and out of SSA form are somewhat complex, but well known and efficient [BCHS98, App02, Mor98, Wol96]. Figure 4.1 shows the main loop of the running example and its equivalent in SSA form. Statements irrelevant to the example have been removed for clarity (e.g. assignments to unused temporaries, unused flag macros, and unused assignments to the stack pointer).

Note how the special variable `esp0` instroduced earlier is now automatically replaced by the initial value of the stack pointer, $\texttt{esp}_0$, and there is no need to explicitly add an assignment from `esp0` to `esp`. The initial value for other locations is also handled automatically, e.g. $\texttt{st}_0$.

Assignments with *ϕ-functions* are inserted at the beginning of basic blocks where control flow merges, i.e. where a basic block has more than one in-edge. A ϕ-function is needed for location a only if more than one definition of a reaches the start of the basic block. The top of the loop in Figure 4.1 is an example; locations $\texttt{m[esp}_0\texttt{-28]}$, `st`, `edx`, `esi`, and `ebx` are all modified in the loop, and also have definitions prior to the loop, but $\texttt{esp}_0$ itself requires no ϕ-function. $\texttt{st}_1$ `:=` $\phi(\texttt{st}_0, \texttt{st}_3)$ means that the value of $\texttt{st}_1$,

```
loop1:
80483d8 46 m[esp] := edx
80483d9 47 st := st *f (double)m[esp]
80483dc 49 edx := edx - 1              // Decrement numerator
80483dd 51 m[esp] := esi
80483e0 52 st := st /f (double)m[ebp-24]) // m[ebp-24] = m[esp]
80483e6 57 esi := esi - 1              // Decrement denominator
80483e7 60 ebx := ebx - 1              // Decrement counter
80483e8 62 goto loop1 if ebx >= 0
80483f5 65 m[esp] := (int)st
80483fb 67 edx := m[esp]
```

(a) Original program. `*f` is the floating point multiply operator, and `/f` denotes floating point division.

```
loop1:
      m[esp0-28]1 := φ(m[esp0-28]0, m[esp0-28]3)
      st1 := φ(st0, st3)
      edx1 := φ(edx0, edx2)
      esi1 := φ(esi0, esi2)
      ebx1 := φ(ebx0, ebx2)
   46 m[esp0-28]2 := edx1
   47 st2 := st1 *f (double)m[esp0-28]2
   49 edx2 := edx1 - 1                 ; Decrement numerator
   51 m[esp0-28]3 := esi1
   52 st3 := st2 /f (double)m[esp0-28]3
   57 esi2 := esi1 - 1                 ; Decrement denominator
   60 ebx2 := ebx1 - 1                 ; Decrement counter
   62 goto loop1 if ebx2 >= 0
65 m[esp0-20]1 := (int)st3
67 edx3 := m[esp0-20]1
```

(b) SSA form.

Figure 4.1: The main loop of the running example and its equivalent SSA form.

treated as a separate variable from $\texttt{st}_0$ and $\texttt{st}_3$, takes the value $\texttt{st}_0$ if control flow enters via the first basic block in-edge (falling through from before the loop), and the value $\texttt{st}_3$ if control flow enters through the second in-edge (from the branch at the end of the loop). In general, a ϕ-function at the top of a basic block with n in-edges will have n parameters.

A relation called the dominance frontier efficiently determines where ϕ-functions are necessary: for every node n defining location a, every basic block in the dominance frontier of n requires a ϕ-function for a. The dominance frontier for a program can be calculated in practically linear time. If required, ϕ-functions can be implemented (made executable) by inserting copy statements. For example, $\texttt{st}_1 := \phi(\texttt{st}_0, \texttt{st}_3)$ could be implemented by inserting $\texttt{st}_1 := \texttt{st}_0$ just before the loop (i.e. at the end of

the basic block which is the first predecessor to the current basic block), and inserting $\texttt{st}_1$:= $\texttt{st}_3$ at the end of the loop (which is the end of the basic block which is the second predecessor to the current basic block). In many instances the copy statements will not be necessary, as will be shown in a later section.

The main property of a program in SSA form is that each use has effectively only one definition. If in the pre-SSA program a use had multiple definitions, the definition will be a ϕ-function that has as operands the original definitions, or other ϕ-functions leading them. A secondary property is that definitions dominate all uses. That is, any path that reaches a use of a location is guaranteed to pass through the single definition for that location. For example st is defined at statement 52 in Figure 4.1(a), after a use in statement 47. However, in the equivalent program in SSA form in Figure 4.1(b), there is a ϕ-function at the top of the loop which dominates the use in statement 47. One of the operands of that ϕ-function references the definition in statement 52 (i.e. $\texttt{sp}_3$).

4.1.1 Benefits

The SSA form makes propagation very easy; initial parameters are readily identified, preservations are facilitated, and SSA's implicit use-definition information requires no maintenance.

Both requirements for expression propagation (Section 3.1 on page 65) are automatically fulfilled by programs in SSA form. The first condition (that s must be the only definition of x to reach u) is satisfied because definitions are unique, and the second condition (that there be no definitions of locations used by s) is satisfied because definitions dominate uses. For example, consider propagating statement

s: $\texttt{st}_2$:= $\texttt{st}_1$ *f (double)$\texttt{m[esp}_0\texttt{-28]}_2$ into

u: $\texttt{st}_3$:= $\texttt{st}_2$ /f (double)$\texttt{m[esp}_0\texttt{-28]}_3$.

Since the program is in SSA form, any path from the entry point to statement s must pass through the unique definitions for $\texttt{st}_1$, $\texttt{esp}_0$, and $\texttt{m[esp}_0\texttt{-28]}$, the three subscripted locations on the right hand side. Because all definitions are unique, any path from s to u cannot contain another definition of $\texttt{st}_1$, $\texttt{esp}_0$, or $\texttt{m[esp}_0\texttt{-28]}$.

If the left hand side of the statement being propagated is a memory expression or an array expression, there is a similar requirement for subscripted locations in the address or index expressions on the left hand side. For example, if propagating

s2: $\texttt{m[esp}_0\texttt{-28]}_3$:= $\texttt{esi}_1$ into

u: $\texttt{st}_3$:= $\texttt{st}_2$ /f (double)$\texttt{m[esp}_0\texttt{-28]}_3$,

there can be no assignment to $\texttt{esp}_0$ or $\texttt{esi}_1$ between *s2* and *u*, and *s2* must be the only definition of $\texttt{m[esp}_0\texttt{-28]}$ to reach *u*. These are again automatically satisfied, except that the SSA form has no special way of knowing whether expressions other than precisely $\texttt{m[esp}_0\texttt{-28]}$ alias to the same memory location. For example, if $\texttt{m[ebp}_1\texttt{-4]}$ aliases with $\texttt{m[esp}_0\texttt{-28]}$ and there is an assignment to $\texttt{m[ebp}_1\texttt{-4]}$, the assumption of definitions being unique is broken. This issue will be discussed in greater detail later.

After statement 46 is propagated into statement 47, the IR is as shown in Figure 4.2. Note that the propagation from statement 47 to statement 52 is one that would not normally be possible, since there is a definition of `edx` between them. In SSA terms, however, the definition is of $\texttt{edx}_2$, whereas $\texttt{edx}_1$, treated as a different variable, is used by statement 47. The cost of this flexibility is that when the program is translated back out of SSA form, an extra variable may be needed to store the old value of `edx` ($\texttt{edx}_1$).

```
46 m[esp0-28]2 := edx1
47 st2 := st1 *f (double)edx1
49 edx2 := edx1 - 1              ; Decrement numerator
51 ...
52 st3 := st2 /f (double)m[esp0-28]3
```

Figure 4.2: A propagation not normally possible is enabled by the SSA form.

As a result, once a program or procedure is in SSA form, most propagations can be performed very easily. Most uses can be replaced by the right hand side of the expression indicated by the subscript. In practice, some statements are not propagated e.g. ϕ-functions and call statements. ϕ-functions must remain in a special position in the control flow graph. Call statements may have side effects, but could be propagated under some circumstances. If the resultant statement after a propagation still has subscripted components, and those definitions are suitable for propagation, the process can be repeated. For example, the result of the substitution from *s* to *u* is

$\texttt{st}_3\ \texttt{:= (st}_1\ \texttt{*f (double)m[esp}_0\texttt{-28]}_2\texttt{) /f (double)m[esp}_0\texttt{-28]}_3$,

which can have the two memory expressions propagated, resulting in

$\texttt{st}_3\ \texttt{:= (st}_1\ \texttt{*f (double)edx}_1\texttt{) /f (double)esi}_1$.

Note that this propagation has removed the machine code detail that the floating point instructions used require the operands to be pushed to the stack; these instructions can not reference integer registers such as `edx` directly. All three uses in this expression refer to ϕ-functions, so no further propagation is possible.

Since expression propagation is simple in SSA form, analyses relying on them (e.g. condition code elimination) benefit indirectly.

While SSA form inherently provides only use-definition information (the one definition for this use, and other definitions via the ϕ-functions), definition-use information (all uses for a given definition) can be calculated from this in a single pass. The IR can also be set up to record definition-use information as well as or instead of use-definition information upon translation into SSA form (so the arcs of the SSA graph point from definition to uses). Definition-use information facilitates transformations such as dead code elimination and translating out of SSA form.

The algorithm for transforming programs into SSA form assumes that every variable is implicitly defined at the start of the program, at an imaginary "statement zero". This means that when a procedure is transformed into SSA form, all its potential parameters (locations used before being defined) appear with a subscript of zero, and are therefore easily identified.

The implicit use-definition information provided by the SSA form requires no maintenance as statements are modified and moved. There is never an extra definition created for any use, and a definition should never be deleted while there is any use of that location. By contrast, definition-use information changes with propagation (some uses are added, others are removed), and when dead code is eliminated (uses are removed).

Deciding whether a location is preserved (saved and restored, or never modified) by a procedure is also facilitated by the way that the SSA form separates different versions of the same location. Firstly, it is easy to reason with equations such as "$\texttt{esi}_{99}$ = $\texttt{esi}_0$", rather than having to distinguish the definition of `esi` that reaches the exit, the original value of `esi`, and all its intermediate definitions. The equation can be simplified to true or false by manipulation of the equation components. Secondly, starting with the definition reaching the exit, preceding definitions can be accessed directly without searching all statements in between.

With the aid of some simple identities, SSA form is able to convert some sequences of complex code into a form which greatly assists its understanding. One example is the triple xor idiom, shown in Section 4.3.

Finally, the SSA form is essential to the operation and use of collectors, discussed in Section 4.5.

From the above, it is clear that the SSA form is very useful for decompilers.

4.1.2 Translating out of SSA form

The conversion from SSA form requires the insertion of copy statements; many factors affect how many copies are needed.

The Static Single Assignment form is not directly suitable for converting to high level source code, since ϕ-functions are not directly executable. Φ-functions could effectively be executed by inserting one copy statement for each operand of the ϕ-function. Every definition of every original variable would then effectively be the definition of a separate variable. This would clutter the decompiled output with variable declarations and copy statements, making it very difficult to read. Hence, the intermediate representation has to be transformed out of SSA form in a different way.

Several mapping policies from SSA variables to generated code variables are possible. For example, the original names could be ignored, and new variables could be allocated as needed. However, this discards some information that was present in the input binary - the allocation of original program variables to stack locations and to registers. While optimising compilers do merge several variables into the one machine location, and sometimes allocate different registers to the same variable at different points in the program, they may not do this, especially if register pressure is low. Attempting to minimise the number of generated variables could lead to the sharing of more than one unrelated original variable in one output variable. While the output program would be more compact, it could become more confusing than other options.

The simplest mapping policy is to keep the original allocation of program variables to memory or register locations as far as possible. In most cases, the implementation of this policy is to simply remove the subscripts and ϕ-functions. However, there are two circumstances where this is not possible.

The first is where the binary program uses the location in such a way that different types will have to be declared in the output program. In the running example, register location `edx` is used to hold three original program variables: `num`, `n`, and `i`. All of these were declared to be the same type, `int`, but if one was of a different fundamental type (e.g. `char*`), then the location `edx` would have to be split into two output variables, each with a different type.

The second circumstance where subscripts cannot simply be removed is that there may be an overlap of live ranges between two or more versions of a location in the current (transformed) IR. When two names for the same original variable are live at the same point in the program, one of them needs to be copied to a new local variable. Figure 4.3 shows part of the main loop of the running example, after the propagations of Section 4.1.1. Note how the multiply operation, previously before the decrement of `edx`, now follows it, and the version of `edx` used ($\texttt{edx}_1$) is the version before the decrement.

The seven columns at the left of the diagram represent the live ranges of the SSA variables. None of the various versions of the variables overlap, except for $\texttt{edx}_1$ and $\texttt{edx}_2$, indicated by the shaded area. When transforming out of SSA form, a temporary

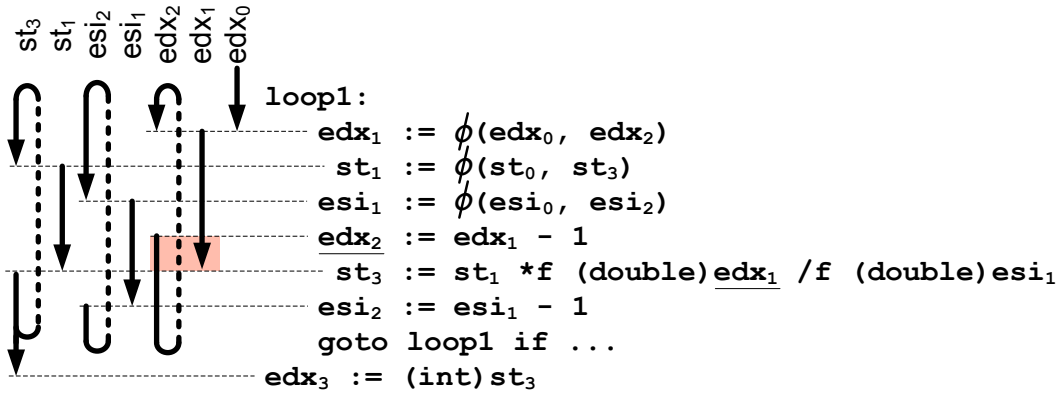


Figure 4.3: Part of the main loop of the running example, after the propagation of the previous section and dead code elimination.

variable is required to hold either $\texttt{edx}_1$ or $\texttt{edx}_2$. Figure 4.4(a) shows the result of replacing $\texttt{edx}_1$ with a new local variable, `local1`.

```
loop1:
   local1 := edx
   edx := edx - 1
   st := st *f (double)local1
      /f (double)esi
   esi := esi - 1

   goto loop1 if ...
edx := (int) st
```

(a) Replacing $\texttt{edx}_1$ with `local1`

```
loop1:
   local2 := edx - 1

   st := st *f (double)edx
      /f (double)esi
   esi := esi - 1
   edx := local2
   goto loop1 if ...
edx := (int) st
```

(b) Replacing $\texttt{edx}_2$ with `local2`

Figure 4.4: The IR of Figure 4.3 after transformation out of SSA form.

It is also possible to replace $\texttt{edx}_2$ with `local2`, as shown in Figure 4.4(b). In this case, the definition for $\texttt{edx}_1$ became ϕ`(edx0, local2)`, which is implemented by inserting the copy statement `edx := local2` at the end of the loop.

Note that in this second case, it is now possible to propagate the assignment to `local2` into its only use, thereby removing the need for a local variable at all. In addition, the combination of the SSA form and expression propagation has achieved code motion (i.e. the decrement of edi has moved past the multiply (as in the original source code) and past the divide (which it was not in the original source code). Unfortunately, this is not always possible, as shown in the next example. Figure 4.5 shows the code of Figure 4.3, with the loop condition now `num > r` instead of $\texttt{c} \neq 0$, and unlimited propagation as before. Register $\texttt{ecx}_0$ holds the value of parameter `r`.

Expression propagation has again been unhelpful, changing the loop condition from the original $\texttt{edx}_2$ `> r` to $\texttt{edx}_1$`-1 > r`. Now it is no longer possible to move the definition of

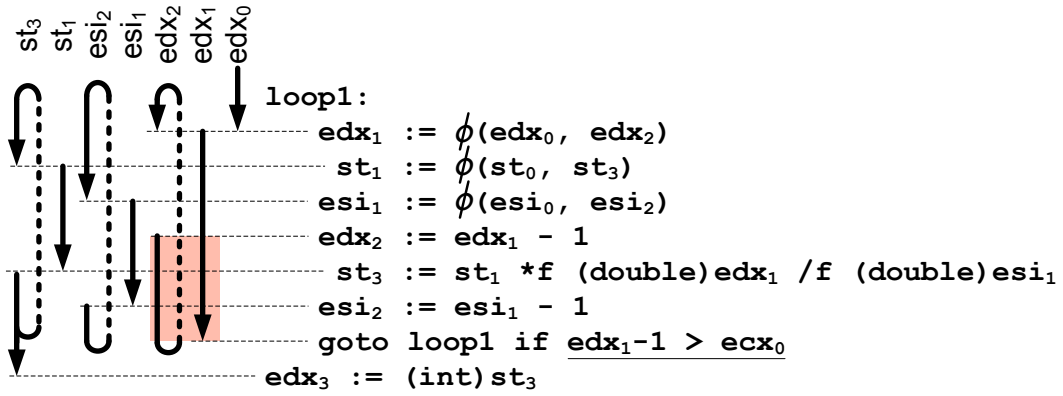


Figure 4.5: The code of Figure 4.3, with the loop condition optimised to `num>r`.

edx_2 to below the last use of edx_1, since edx_1 is used in the loop termination condition. Figure 4.6 shows two possible transformations out of SSA form for the code of Figure 4.5.

```
loop1:
   local1 := edx
   edx := local1-1
   st := st *f (double)local1
      /f (double)esi
   esi := esi - 1

   goto loop1 if local1-1 > ecx
edx := (int) st
```

(a) Replacing edx_1 with `local1`

```
loop1:
   local2 := edx-1

   st := st *f (double)edx
      /f (double)esi
   esi := esi - 1
   local1 := edx
   edx := local2
   goto loop1 if local1-1 > ecx
edx := (int) st
```

(b) Replacing edx_2 with `local2`, then of necessity edx_1 with `local1`

Figure 4.6: Two possible transformations out of SSA form for the code of Figure 4.5.

Note that the second case could again be improved if expression propagation is performed on `local2`. Unfortunately, at this point, the program is no longer in SSA form, so propagation is no longer as easy as it was. Such propagations within the transformation out of SSA form are only needed across short distances (within a basic block). Hence it may be practical to test the second condition for propagation by testing each statement between the source and destination. This condition (see Section 3.1) states that components of the expression being propagated should not be redefined on the path from the source to the destination.

In this case, not propagating into the loop expression would have avoided the problem. Techniques for minimising the number of extraneous variables and copy statements will

be discussed in a following section. Optimising the decision of which of two versions of a variable to replace with a new variable, and effectively performing code motion, is equivalent to sequentialising data dependence graphs, and has been shown to be NP complete [Upt03].

The above examples show that the number of copy statements and extra local variables is affected by many factors. In the worst case, it is possible for a ϕ-function with n operands to require n copy statements to implement it, if every operand has been assigned a different output variable. Section 4.1.3 will discuss how to minimise the copies.

4.1.2.1 Unused Definitions with Side Effects

Unused but not eliminated definitions with side effects can cause problems with the translation out of SSA form, but there is a simple solution.

Figure 4.7 shows a version of the main loop of the running example where an assignment to $\texttt{st}_2$, even though unused, is not eliminated before transformation out of SSA form. This can occur with assignments to global variables, which should never be eliminated.

```
loop1:
   st1 := φ(st0, st3)
   edx1 := φ(edx0, edx2)
   esi1 := φ(esi0, esi2)
   st2 := st1 *f (double)edx1      ; Unused definition
   edx2 := edx1 - 1                ; Decrement numerator
   st3 := st1 *f (double)edx1 /f (double)esi1
   esi2 := esi1 - 1                ; Decrement denominator
   goto loop1 if ...
edx3 := (int) st3
```

Figure 4.7: A version of the running example where an unused definition has not been eliminated.

The definition of $\texttt{st}_2$ no longer has any uses, so it has no live range, and hence does not interfere with the live ranges of other versions of `st`. As a result, considering only live ranges, translation to the incorrect code of Figure 4.8 is possible.

This program is clearly wrong; in terms of the original program variables, `res *= num` is being performed twice for each iteration of the loop, where the original program performs this step only once per iteration. The solution to this problem is to treat definitions of versions of variables as interfering with any existing liveness of the same variable, even if they cannot themselves contribute new livenesses. The result is to cause one or other assignment to `st` to assign to a new variable.

```
loop1:
   st := st *f (double)edx       ; Unused code with side effects
   local1 := edx
   edx := edx - 1
   st := st *f (double)local1 /f (double)esi
   esi := esi - 1
   goto loop1 if ...
edx := (int) st
```

Figure 4.8: Incorrect results from translating the code of Figure 4.7 out of SSA form, considering only the liveness of variables.

4.1.2.2 SSA Back Translation Algorithms

Several algorithms for SSA back translation exist in the literature, but one due to Sreedhar et al. appears to be most suitable for decompilation.

Because of the usefulness of the SSA form, and the cost of transforming out of SSA form, decompilers benefit from keeping the intermediate representation in SSA form for as long as possible.

The process of translating out of SSA form is also known as SSA back translation in the literature. The original algorithm for translation out of SSA form by Cryton et al. [CFR+91] has been shown to have several critical problems, including the so-called "lost copy" problem. Solutions to the problem have been given by several authors; the salient papers are by Briggs et al. [BCHS98] and Sreedhar et al. [SDGS99]. Sassa et al. attempted to improve on these solutions, but concluded that the algorithm by Sreedhar et al. produces fewer copies except in rare cases [SKI04].

While the application of these algorithms to decompilation remains the subject of future work, it would appear that the algorithm by Sreedhar et al. offers the best opportunity for decompilation. Sreedhar uses both liveness information and the interference graph to choose where to insert copy statements to implement the semantics of ϕ-functions. (Two locations interfere with each other if their live ranges overlap.) He then uses a coalescing algorithm to coalesce pairs of variables (use the same name). Unique to Shreedhar's algorithm is the ability to safely coalesce the variables of copy statements such as `x=y` where the locations interfere with each other, provided that the coalesced variable does not cause any new interference.

Figure 4.9 illustrates the coalescing phase. Figure 4.9(a) shows a program in SSA form. Since none of the operands of the ϕ-functions interfere with each other, the subscripts can be dropped as shown in Figure 4.9(b). The standard coalescing algorithm, due to Chaitin, can not eliminate the copy `x=y` since they interfere with each other [Cha82].

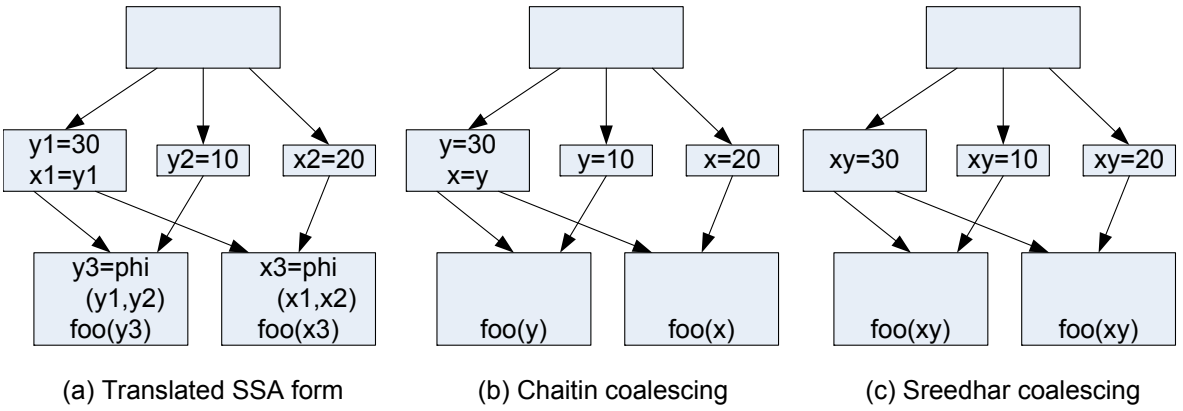


Figure 4.9: Sreedhar's coalescing algorithm. Adapted from [SDGS99].

Although the variables `x1` and `y1` interfere with each other, Sreedhar's algorithm is able to identify that if `x1` and `y1` are coalesced, the resulting variable does not cause any new interferences. Figure 4.9(c) shows the example code after `x1`, `x2`, `y1`, and `y2` are coalesced into one variable `xy`, and the copy statement `x1=y1` is removed.

4.1.2.3 Allocating Variables versus Register Colouring

Translation out of SSA form in a decompiler and register colouring in a compiler have some similarities, but there are enough significant differences that register colouring cannot be adapted for use in a decompiler.

The role of the register allocator in a compiler, usually implemented by a register colouring algorithm, is to allocate registers to locations in the IR. The number of registers is fixed, and depends on the target machine. It is important that where live ranges overlap, distinct registers are allocated.

One of the main tasks when translating out of SSA form in a decompiler is to allocate local variables to locations in the SSA-based IR. The number of local variables has no fixed limit, however for readability, it is important to use as few local variables as possible. It is important that where live ranges overlap, distinct local variables are allocated.

From the above two paragraphs, it can be seen that there are similarities and differences between register colouring and the translation out of SSA form. The similarities include:

- Resources are allocated to locations in the IR.
- These resources should be allocated sparingly.
- Where live ranges of locations in the IR overlap, an edge is inserted in an interference graph.

- For both compilers and decompilers, ϕ-functions indicate sets of locations that it would be beneficial to allocate the same resource to. These relations could be recorded either by a different kind of edge in the interference graph, or edges in a different graph (e.g. a "unites" graph).

The differences include:

- Compilers allocate machine registers, while decompilers allocate local variables.
- The number of registers is fixed, but the number of local variables has no fixed limit.
- Spilling to memory may be required in a compiler when a given graph cannot be coloured with K colours, where K is the number of available machine registers. There is no equivalent concept in a decompiler, although the necessity of allocating new local variables in some circumstances has a superficial similarity.
- For a compiler, a small set of interference graph nodes are pre-coloured, because some registers must be used for parameter passing or the return value, or because of peculiarities of the target machine architecture (e.g. the result of a multiplication appears in `edx:eax`). For a decompiler, assuming that the original mapping from locations to local variables is to be preserved as far as possible, the graph is initially completely pre-coloured.
- For a compiler, the interference graph is always consistent, but until finished it is not fully coloured. For a decompiler, the graph is always fully coloured but possibly inconsistent; the task is not to colour the graph, but to remove the inconsistencies caused by overlapping live ranges.
- For a compiler, it is simplest to implement the ϕ-functions with copy statements, and attempt to minimise the copy statements with the coalescing phase of the register colouring algorithm. The ϕ-functions or copy statements guide the allocation of registers to IR locations. For a decompiler, ϕ-functions are used to guide the allocation of new local variables if needed.
- While a compiler attempts to coalesce nodes in the interference graph (thereby allocating the same register to a group of assignments), a decompiler must at times split a group of nodes that currently uses the same local variable, so that some of the members of the group no longer use the same local variable as the others. The aim of coalescing is to reduce copy statements; the cost of splitting is to introduce more copy statements.

The differences are significant enough that only interference graphs and the broad concept of colouring can be adapted from register colouring to allocating local variables in a decompiler.

4.1.3 Extraneous Local Variables

While SSA and expression propagation are very helpful in a decompiler, certain patterns of code induce one or more extraneous local variables, which reduces readability.

Consider the IR fragment of Figure 4.1 from the running example. After exhaustive expression propagation, the result is as shown in Figure 4.10.

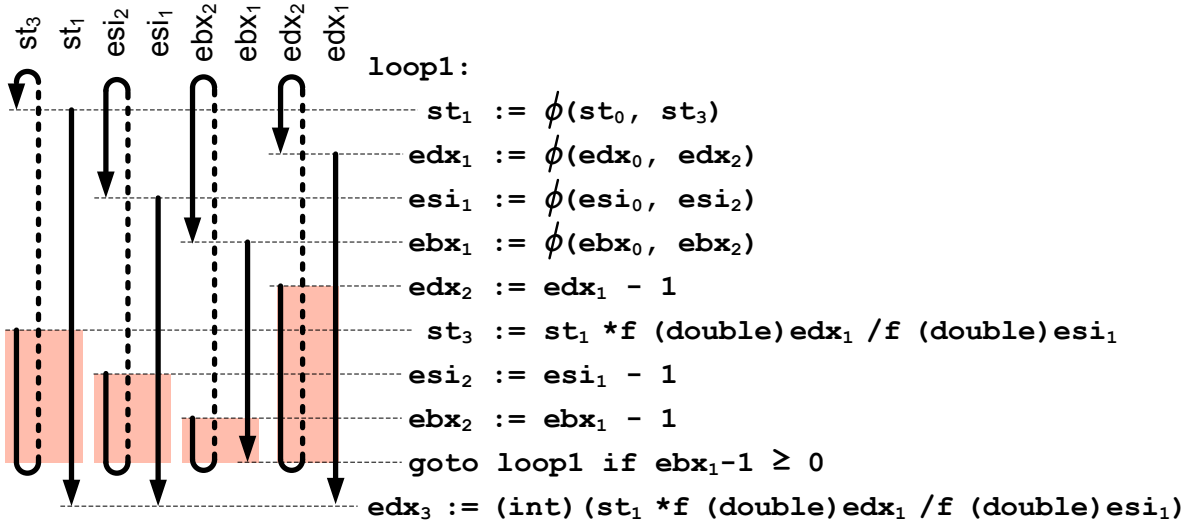


Figure 4.10: The code from Figure 4.1 after exhaustive expression propagation, showing the overlapping live ranges.

Note the multiple shaded areas indicating an overlap of live ranges of different versions of the same variable. When propagating $\texttt{st}_2$ of Figure 4.1, which uses $\texttt{edx}_1$ (via $\texttt{m[esp}_0\texttt{-28]}_2$), past a definition of $\texttt{edx}_2$, `local8` is allocated, representing $\texttt{edx}_2$. However, now the ϕ-function for $\texttt{edx}_1$ requires a copy, since $\texttt{edx}_1$ can now become $\texttt{edx}_0$ or `local8` depending on the path taken. This necessitates a copy of $\texttt{edx}_2$ at the end of the loop into `local12`, and back to $\texttt{edx}_1$ (allocated to `local11`) at the start of the loop. Similarly, the live range overlaps of `ebx`, `esi`, and `st` (`st` is the register representing the top of the floating point stack) necessitate copies to variables `local10`, `local17`+`local9`, and `local7` respectively. The resultant generated code is obviously very hard to read, due to six extra assignment statements in the main loop, as shown in Figure 4.11.

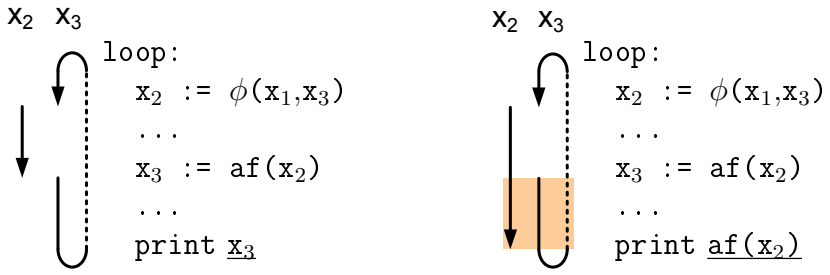
As well as the extra local variables, the loop condition is changed by one (`local10 >= 1` compared with the original `c > 0`). In other programs, this can be more obvious,

```
do {
   local11 = local12;
   local10 = local15;
   local17 = local13;
   local7 = local18;
   local8 = local11 - 1;
   local18 = local7 * (float)local11 / (float)local17;
   local9 = local17 - 1;
   local15 = local10 - 1;
   local12 = local8;
   local13 = local9;
} while (local10 >= 1)
local14 = (int)(local7 * (float)local11 / (float)local17);
```

Figure 4.11: Generated code from a real decompiler with extraneous variables for the IR of Figure 4.10. Copy statements inserted before the loop are not shown.

e.g. `i < 10` becomes `old_i < 9`. The floating point multiply and divide operations are repeated after the loop, adding more unnecessary volume to the generated code. Finally, the last iteration of the loop, computing the result in `local18`, is unused. As a result of all these effects, the output program is potentially less efficient than the original. A very good optimising compiler used on the decompiled output might be able to attain the same efficiency as the original program.

The code where these extraneous local variables are being generated contains statements such as x_3 `:= af(`x_2`)` inside a loop, where `af` is an arithmetic function, not a function call. Examples include `x := x - 1` and `y := (x*y) + z`. Propagation of statements like this inside a loop will always cause problems, as shown in Figure 4.12(a).



(a) Before propagation. (b) After propagation of x_3.

Figure 4.12: Live ranges for x_2 and x_3 when x_3 `:= af(`x_2`)` is propagated inside a loop.

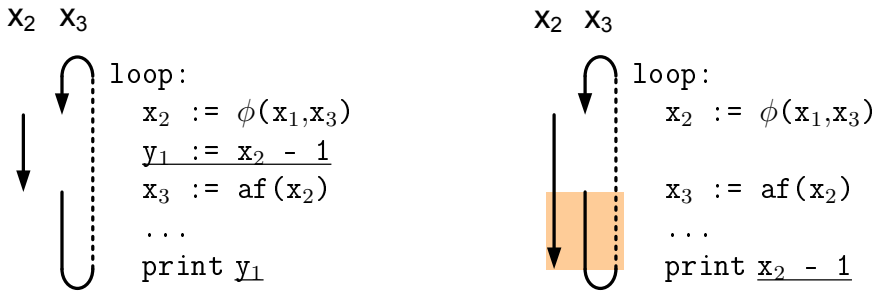
Note that either x_2 or x_3 is live throughout the loop. If x_3 is propagated anywhere in the loop, e.g. to the print statement, x_2 will be live after the assignment to x_3, as shown in Figure 4.12(b). The assignment to x_3 cannot be eliminated, since it has side effects;

in other words, x_3 is used by the ϕ-function, hence it is never unused. Overlapping live ranges for the different versions of the same base variable will result in extra variables in the decompiled output, as noted above. Only assignments of this sort, where the same location appears on the left and right hand sides, have this property. These assignments are called *overwriting statements*.

Note that the opportunity for propagation in this situation is unique to renaming schemes such as the SSA form. In normal code, overwriting statements cannot be propagated, since a component of the right hand side (here x_2) is modified along all paths from the source to the destination. Propagation of this sort in SSA form is possible only as a result of the assumption that different versions of the same variable are distinct.

Propagating overwriting statements where the definition is not in a loop is usually not a problem. The reason is that the propagated expression usually becomes eliminated as dead code, so that the new version of `x` does not interfere with other versions of `x`. When inside a loop, the new version of `x` is always used by the ϕ-function at the top of the loop.

The problem also occurs when propagating expressions that "carry" a component of the overwriting statements across the overwriting statement. Figure 4.13 shows an example where the non-overwriting statement `y := x-1` is propagated across an overwriting statement such as `x := x+1`.



(a) Before propagation. (b) After propagation of y_1.

Figure 4.13: Live ranges for x_2 and x_3 when y_1`:= x2- 1` is propagated across an overwriting statement inside a loop.

Whether an overwriting statement is inside a loop can be determined by whether there is a component of the right hand side that is defined by a ϕ-function one of whose operands is the location defined by the statement. Φ-functions reference earlier values of a variable; for a location such as x_3 in 4.13 to appear as an operand of a ϕ-function that defines one of its components, it must be in a loop.

Proposition 4.1: *Propagating components of overwriting statements past their definitions inside a loop leads to the generation of extraneous local variables when the code*

Algorithm 1 Preventing extraneous local variables in the decompiled output due to propagation of components of overwriting statements past their definitions.

```
/* Initial: r is a subscripted location that could be propagated into. */
/* Returns: true if this location should be propagated into */
function shouldPropagateInto(r)
  src := the assignment defining r
  for each subscripted component c of the RHS of r
    if the definition for c is a φ-function ph then
      for each operand op of ph
        opdef := the definition for op
        if opdef is an overwriting statement and either
          (opdef = src) or
          (there exists a CFG path from src to opdef and
          from opdef to the statement containing r) then
            return false
  return true
```

is transformed out of SSA form.

In Figure 4.13(a), $\mathtt{x}_2$ is a component of the assignment to $\mathtt{y}_1$. Because x_2 appears in an overwriting statement between the definition of y_1 and its use, and the definition of the overwriting statement (x_3) is an operand of the ϕ-function defining x_2, the overwriting statement is in a loop. Hence, propagating $\mathtt{y}_1$ past the overwriting statement (defining $\mathtt{x}_3$) would lead to an extraneous variable in the decompiled output.

Algorithm 1 gives the detail of how to prevent extraneous local variables as discussed in this section. It has been implemented in the Boomerang decompiler; for results see Section 7.3 on page 230.

4.2 Applying SSA to Memory

As a result of alias issues, memory expressions must be divided into those which are safe to propagate, and those which must not be propagated at all.

Memory expressions are subject to aliasing, in other words, there can be more than one way to refer to the location, effectively creating more than one name for the same location. The meaning of the term "name" here is different to the one used in the context of SSA renaming or subscripting. Aliasing can occur both at the source code level and at the machine code level.

At the source code level, the names for a location might include g (a global variable), $*p$ (where p is a pointer to g), and *param1* (where *param1* is a reference parameter that is in at least some cases supplied with the actual argument g).

At the machine code level, all these aliases persist. Reference parameters appear as call-by-value pointers at the machine code level, but this merely adds another * (dereference) to the aliasing name. In addition, a global variable might have the names `m[10 000]` (the memory at address 10 000), `m[r1]` where `r1` is a machine register and `r1` could take the value 10 000, `m[r2+r3]` where `r2+r3` could sum to 10 000, `m[m[10 008]]` where `m[10 008]` has the value `10 000`, and so on. There will also be several equivalents each for **p* (e.g. `*edx`) and *param1*.

These problems exist regardless of the intermediate representation of the program. As a result of these additional aliasing opportunities, aliasing is more common at the machine code level than at the source code level.

The main causes of machine code aliasing are the manipulation of the stack and frame pointers, and the propensity for compilers to use registers as internal pointers to objects (pointers inside multi-word data items, not pointing to the first word of the item). Internal pointers are particularly common with object oriented code, since objects are frequently embedded inside other objects.

The following sections will detail the problems and offer some solutions.

4.2.1 Problems Resulting from Aliasing

Alias-induced problems are more common at the machine code level, arising from internal pointers, a lack of preservation information, and other causes.

Figure 4.14 illustrates the problem posed by internal pointers with a version of the running example.

Statement 3, which causes one register to become a linear function of another register containing a pointer, provides an opportunity for memory aliasing. By inspection, the assignments of statements 14 and 15 are to the same memory location, but the memory expressions appear to be different in Figure 4.14(b). When the memory variables are placed into SSA form, as shown in Figure 4.14(c), they are treated as independent variables. As a result, the wrong value (from statement 3) is propagated, and the decompilation is incorrect.

In this example, statements 2 and 3 have not been propagated into statements 13-15 for some reason; perhaps they were propagated but too late to treat the memory expressions of statements 14 and 15 as the same loation. If they had been propagated, the last two memory expressions would have been $\texttt{m[eax}_1\texttt{ + 4]}$, and the problem would not exist. This is an example where too little propagation causes problems.

```
int main() {
   pair<int>* n_and_r = new pair<int>;
   ...
   printf("Choose %d from %d:  %d\n", n_and_r->first,
      n_and_r->second, comb(n_and_r));
```

(a) Original source code

```
 1 eax1 := malloc(8)    // eax1 points to n_and_r and n_and_r.n
 2 esi2 := eax1         // esi1 points to n_an_r and n_and_r.n
 3 edi3 := eax1 + 4     // edi3 points to n_and_r.r
 ...
13 m[esi2] := exp1
14 m[esi2+4] := exp2
15 m[edi3] := exp3
26 printf ..., m[esi2], m[esi2+4], ...
```

(b) Original IR with the call to `comb` inlined

```
 1 eax1 := malloc(8)
 2 esi2 := eax1
 3 edi3 := eax1 + 4
 ...
13 m[esi2]13 := exp1
14 m[esi2+4]14 := exp2
15 m[edi3]15 := exp3
26 printf ..., m[esi2]13→exp1 , m[esi2+4]14→exp2, ...
```

(c) Incorrect SSA form

Figure 4.14: A version of the running example using pairs of integers.

Alias-induced problems can also arise from a lack of preservation information. Preservation analysis will be discussed more fully in Section 4.3. Figure 4.15(a) shows code where a recursive call (statement 30) prevents knowledge of whether the stack frame register (`ebp`) is preserved by the call.

Preservation analysis requires complete data flow analysis, and complete data flow analysis requires preservation analysis for the callee (which, because of the recursion, is the current procedure, and has not yet had its data flow summarised). As a result of the lack of preservation knowledge, the frame pointer is conservatively treated as being defined by the call. Later analysis reveals that `ebp` is not affected by the call, i.e. $\texttt{ebp}_{30} = \texttt{ebp}_3 = \texttt{esp}_0\texttt{-4}$, and the memory expression in statement 79 is in fact the parameter n as read at statement 14 (i.e. $\texttt{m[esp}_0\texttt{+4]}_0$). In statement 79, the memory expression $\texttt{m[ebp}_{30}\texttt{+8]}_{30}$ therefore becomes $\texttt{m[esp}_0\texttt{+4]}_0$. The problem is to ensure that no attempt is made to propagate to or from the memory location $\texttt{m[ebp}_{30}\texttt{+8]}_{30}$ until its address expression is in some sense "safe" to use. Another issue is that the data flow analysis of the whole procedure is not complete until it is known that the memory locations of

```
int rcomb(int n, int r) {
   if (n <= r)
      return 1;
   double res = rcomb(n-1, r);
   res *= n;
   res /= (n-r);
   return (int)res;
}
```

(a) Source code.

```
14 eax14 := m[esp0+4]0          ; n
23 m[esp0-48]23 := m[esp0+8]    ; argument r
25 m[esp0-52]25 := eax14-1      ; argument n-1
30 eax30, ebp30, m[ebp30+8]30 := CALL rcomb(...)    ; Recurse
   ...
79 st79 := st76 *f (double) m[ebp30+8]30   ; res *= n
```

(b) IR after expression propagation but before preservation analysis.

Figure 4.15: A recursive version *comb* from the running example, where the frame pointer (`ebp`) has not yet been shown to be preserved because of a recursive call.

statements 14 and 79 are in fact the same location.

4.2.2 Subscripting Memory Locations

If memory locations are propagated, care must be taken to avoid alias issues; however, not propagating memory locations is by itself not sufficient.

The problems with correctly renaming memory expressions leads to the question of whether memory expressions should be renamed at all. For example, the Low Level Virtual Machine (LLVM, [LA04]) compiler optimisation framework uses SSA as the basis for its intermediate representation, however "memory locations in LLVM are not in SSA form" (not subscripted). To address this question, consider Figure 4.16.

Figure 4.16(a) shows the main loop of the running example with one line added; this time variables `c` and `denom` are allocated to local memory. Figure 4.16(b) shows the program in SSA form, ignoring the alias implications of statement 82. The program is correctly represented, except for one issue. If `c` and `*p` could be aliases, the program is incorrect. Hence, care must be taken if the address of `c` is taken, and could be assigned to `p`.

```
c = n-r;
*p = 7;    /* Debug code */
while (c-- > 0) {
   res *= num--;
   res /= denom--;
}
```

(a) Source code.

```
80 c80 := n0 - r0
81 esi81 := p0
82 m[esi81] := 7  // *p = 7
84 dl84 := (n0-r0 > 0)
85 c85 := c80 - 1
86 goto after_loop if dl84 = 0
                  → n0-r0 <= 0
loop:
```

(b) Subscripting and propagating memory expressions.

```
80 c := n - r;
81 esi81 := p
82 m[esi81] := 7 // *p = 7
84 dl84 := (c > 0)
85 c := c - 1
86 goto after_loop if dl84 = 0
                        → c <= 0
loop:
```

(c) Error: No propagation of memory expressions.

Figure 4.16: The running example with a debug line added.

Figure 4.16(c) shows the result of propagating only non-memory expressions; the `while` loop now tests `c` after the decrement, which is incorrect.

Note that although no definition of a memory location was propagated, `dl` (an 8-bit register, the low byte of register `edx`) was propagated, and "carried" the old value of `c` (from statement 84) across a definition of `c` (statement 85). LLVM appears to avoid this problem by making memory locations accessible only through special `load` and `store` bytecode instructions; all expressions are in terms of virtual registers, never memory locations directly. Load and store instructions are carefully ordered so that the original program semantics are maintained. As a result, expressions such as `c` $\leq$ `0` (where `c` represents a memory location) are not even expressible. (They would load `c` into a virtual register and compare that against 0). Decompilers cannot avoid converting some memory expressions (e.g. `m[sp-K]`) to variables, but the fact that a variable originated from memory expressions can be recorded.

One solution would be to not allow propagation of any statements that contain (or contained in the past) any memory expression components on the right hand side. However, this has the drawback of making programs difficult to read. For example, consider a program fragment with the following original source code:

```
a[i] = b[g] + c[j];    // g is a global variable
```

The IR and the generated code, if not propagating non-local memory expressions, would be similar to

```
r1 := m[G]                          r1 = g;
r2 := m[B+r1]                       r2 = b[r1];
r3 := m[C+rj]  # rj holds j         r3 = c[j];
r4 := r1 + r3
m[A+ri] := r4  # ri holds i         a[i] = r2+r3;
```

Obviously, it is desirable to propagate memory expressions at least "short distances", so that these lines can be combined. Naturally, this must be done with alias safety.

4.2.3 Solution

A solution to the memory expression problem is given, which involves propagating only suitable local variables, and delaying their subscripting until after propagation of non-memory locations.

The solution to the problem of conservatively propagating memory locations has several facets. Firstly, only stack local variables whose addresses do not escape the current procedure are propagated at all. Such local variables are where virtual registers in a compiler's IR are allocated, apart from those that are allocated to registers. All other memory expressions, including global variables, array element accesses, and dereferences of pointers, are not propagated using the usual method.

Naturally, with a suitable analysis of where the address of a local variable is used, the restriction on propagating local variables whose address escapes the local procedure could be relaxed. Also, where the address is used inside the current procedure, care must be taken not to propagate a local variable across an aliased definition. Escape analysis is much more difficult in a decompiler than in a compiler. For example, to take the address of local variable `foo` at the source code level, the expression `&foo` or equivalent is always involved. If the address of `foo` at the machine code level is `sp-12` (where `sp` is the stack pointer register), then an expression such as `sp-12` may be involved, or it may actually be used to find the address of the variable at `sp-4`, or the expression `sp-24` might be used to "forge" the address of `foo` by adding 12 to that expression. The latter is particularly likely if `foo` happens to be a member of a structure whose address is `sp-24`. Escape analysis is the subject of future work.

For those memory locations that are identified as able to be propagated, their subscripting and hence propagation is delayed until the first round of propagations and preservation analysis is complete. This requires a Boolean variable, one for each procedure, which is set after these propagations and analyses. Before this variable is set, no attempt is made to subscript or create ϕ-functions for memory locations. After the

Boolean is set, the subscripting and ϕ-function inserting routines are called again. Only local variable locations identified as suitable for propagation are subscripted.

For propagation over "short distances" to avoid temporary variables as noted at the end of the last section, a crude propagation could be attempted one statement at a time, checking each component propagated for alias hazards associated with the locations defined in the statement being propagated across. This would allow propagation of array elements, as per the example at the end of the last section.

Figure 4.17 shows the result of applying the solution of this section to the problem shown in Figure 4.14.

```
 1 eax1 := malloc(8)
 2 esi2 := eax1
 3 edi3 := eax1 + 4
  ...
13 m[eax1]13 := exp1
14 m[eax1+4]14 := exp2
15 m[eax1+4]15 := exp3
26 printf ..., m[eax1]13→exp1 , m[eax1+4]15→exp3
```

Figure 4.17: The example of Figure 4.14 with expression propagation before renaming memory expressions.

Now, expression propagation has replaced $\texttt{esi}_2$+4 and $\texttt{edi}_3$ in statements 13-15 with the more fundamental expressions involving $\texttt{eax}_1$. As a result, when SSA subscripting is re-run, the two assignments in statements 14 and 15 are treated correctly as assignments to the same location, and the correct result is printed.

4.3 Preserved Locations

The ability to determine whether a location is preserved by a procedure call is important, because preserved locations are an exception to the usual rule that definitions kill other definitions.

In data flow analysis, a definition is considered to be killed by another definition of the same location. There is a change of the expression currently assigned to the location defined in the first definition, which normally means that this first value is lost. However, this is not always the case. A common example is in a procedure, where callee-save register locations are saved near the start of the procedure, and restored near the end. The register is said to be *preserved*. Between the save and the restore, the procedure uses the register for various purposes, usually unrelated to the use of that register in its callers.

Whether a location is preserved or not is often specified in a convention called the application binary interface (ABI). However, there are many possible ABIs, and the one in use can depend on the compiler and the calling convention of a particular procedure. In addition, a compiler may change the calling convention for calls that are not visible outside a compilation module, e.g. to use more registers to pass paremeters. Assembler code or binary patches may also have violated the convention. This is particularly important if the reason for decompiling is security related. Hence, the safest policy is not to make ABI assumptions, hence *preservation analysis* is required.

A `pop` instruction, or load from stack memory in a RISC-like architecture, which restores the register is certainly a definition of the register. The unrelated definitions of the register between the save and the restore are certainly killed by the restore. However, any definitions reaching the start of the procedure in reality reach the end of the procedure. Since the save and restore are on all possible paths through the procedure, definitions available at the start of the procedure are also available at the end.

Figure 4.18 shows the effect of ignoring the restoration of a variable.

```
a := 5
x := a
call proc1
y := a
```

(a) Input program

```
x := 5
proc1();
y := 5;
```

(b) Ideal decompilation

```
x := 5;
a := proc1(5);
y := a;
```

```
a := 5;
x := 5;
proc1(a); // ref param
y := a;
```

(c) Decompilations resulting from ignoring the restoration of variable `a` in `proc1`

Figure 4.18: The effect of ignoring a restored location. The last example uses a call-by-reference parameter.

Either the decompiled output has extra returns, or extra assignments. More importantly, the propagation of constants and other transformations which improve the output (e.g. `y := 5`) are prevented.

It may be tempting to reason that the saving and restoring of registers at the beginning and end of a procedure is a special case, deserving special treatment, as in [SW93]. However, compilers may spill (save) registers at any time, and assembler code could save and restore arbitrary locations (such as local or global variables). Treating `push` and `pop` instructions as special cases in the intermediate representation is adequate for architectures featuring such instructions (e.g. the *dcc* [Cif94] compiler does this), but does not solve the problem for RISC machine code.

Saving of locations could occur without writing and reading memory, as shown in Figure 4.19. In this contrived example, an idiomatic code sequence for swapping two registers without using memory or a temporary register has been used.

```
b := xor a, b        ; b = a_old ^ b_old   (^ is xor op.)
a := xor a, b        ; a = a_old ^ (a_old ^ b_old) = b_old
b := xor a, b        ; b = b_old ^ (a_old ^ b_old) = a_old
a := 5               ; define a (new live range)
print(a)             ; use a
b := xor a, b        ; swap...
a := xor a, b        ; ... a and b ...
b := xor a, b        ; ...  again
ret                  ; return to caller
```

Figure 4.19: Pseudo code for a procedure. It uses three `xor` instructions to swap registers `a` and `b` at the beginning and end of the procedure. Effectively, register `a` is saved in register `b` during the execution of the procedure.

By itself, expression propagation cannot show that the procedure of Figure 4.19 preserves variable `a`. The first statement could be substituted into the next two, yielding `a=b` and `b=a` except that `a` is redefined on the path from the first to the third statement. A temporary location is required to represent a swap as a sequence of assignments. The combination of expression propagation and SSA form can be used to analyse this triplet of statements, as shown in Figure 4.20.

```
b1 = a0 ^ b0        ; b1 = a0 ^ b0
a1 = a0 ^ b1        ; a1 = a0 ^ (a0 ^ b0) = b0
b2 = a1 ^ b1        ; b2 = b0 ^ (a0 ^ b0) = a0
a2 = 5              ; define a (new live range)
print(a2)           ; use a                                  print(5)
b3 = a2 ^ b2        ; b3 = 5 ^ a0
a3 = a2 ^ b3        ; a3 = 5 ^ (5 ^ a0) = a0: preserved
b4 = a3 ^ b3        ; b4 = a0 ^ (5 ^ a0) = 5 : overwritten   b4 = 5
ret                 ; return to caller                       return b4
```

(a) Intermediate representation (b) After prop. & DCE

Figure 4.20: Pseudo code for the procedure of Figure 4.19 in SSA form. Here it is obvious (after expression propagation) that `a` is preserved, but `b` is overwritten.

In this example, it is clear that `a` is preserved (the final value for a_3 is a_0), but `b` is overwritten (the final value for b_4 is `5`, independent of b_0). Effectively, b_1 is used to save and restore a_0 (although combined in a reversible way with b_0). `b` is a potential return, despite the fact that the assignment is to the variable `a`. If no callers use `b`, it can be removed from the set of returns, but not from the set of parameters. Figure 4.21 shows the procedure of Figure 4.19 with two extra statements.

With propagation and dead code elimination, it is obvious that `a` and `b` are now parameters of the procedure (since they are variables with zero subscripts).

```
b = xor a, b
c = b
a = xor a, b
b = xor a, b
a = 5
print(a)
print(c)
b = xor a, b
a = xor a, b
b = xor a, b
ret
```

(a) Original code

```
b1 = a0 ^ b0
c1 = a0 ^ b0
a1 = b0
b2 = a0
a2 = 5
print(5)
print(a0 ^ b0)
b3 = 5 ^ a0
a3 = a0
b4 = 5
ret
```

(b) SSA form

```
print(5)
print(a0 ^ b0)


b4 = 5
return b4
```

(c) After propagation and DCE

Figure 4.21: The procedure of figure 4.19 with two extra statements.

The above example illustrates the power of the SSA form, and how the combination of SSA with propagation and dead code elimination resolves many issues.

All procedures can trivially be transformed to only have one exit, and a definition collector (see Section 4.5) can be used to find the version of all variables reaching the exit. This makes it easy to find the preservation equation for variable `x`: it is simply $x_0{=}x_e$, where e is the definition reaching the exit, provided by the definition collector in the return statement. With some manipulation and a lot of simplification rules, this equation can be resolved to true or false. This algorithm for determining the preservation of locations using the SSA form was first implemented in the Boomerang decompiler by Trent Waddington [Boo02].

In the absence of control flow joins, expression propagation will usually make the decision trivial, as in Figure 4.21, where the preservation equation for `a` is $a_3{=}a_0$, and the definition reaching the exit is the assignment a_3 `:=` a_0. Substituting the assignment into the equation yields $a_0{=}a_0$, which simplifies to true. At the control flow joins, there will be ϕ-functions; the equation is checked for each operand of the ϕ-function.

Figure 4.22 shows an example with a ϕ-function.

The preservation equation is $ebx_0{=}ebx_2$. Substituting the assignment ebx_2 `:=` ebx_1`-1` into the equation yields $ebx_0{=}ebx_1$`-1` . It is not yet impossible that `ebx` is preserved; for example, if the definition for ebx_1 happened to be ebx_0`+1`, `ebx` would indeed be preserved by procedure `comb`. However, the definition in the example is the ϕ-function ϕ`(`ebx_0`, `ebx_2`)`. The current equation $ebx_0{=}ebx_1$`-1` is therefore split into two equations, one for each operand of the ϕ-function, substituting into ebx_1 of the current equation. The resultant equations are $ebx_0{=}ebx_0$`-1` and $ebx_0{=}ebx_2$`-1`, both of which have to simplify to true for `ebx` to be analysed as preserved by `comb`. In this case, the first equation evaluates to false, so there is no need to evaluate the second. Note that

```
comb: ...
loop:
   ebx1 := φ(ebx0, ebx2)
   ...
   ebx2 := ebx1 - 1
   goto loop1 if ebx2 >= 0
...
return {Reaching definitions:  ebx2, ...}
```

Figure 4.22: A version of the running example with the `push` and `pop` of `ebx` removed, illustrating how preservation analysis handles ϕ-functions.

if the second equation had to be evaluated, there is a risk of infinite recursion that has to be carefully avoided, since $\texttt{ebx}_2$ is defined in terms of $\texttt{ebx}_1$ and $\texttt{ebx}_1$ is partly defined in terms of $\texttt{ebx}_2$. Maintaining a set of locations already processed is sufficient. It is possible to encounter "ϕ loops" that involve several ϕ-functions, and which are not obvious from examination of the program's IR [BDMP05].

Section 7.5 on page 234 describes the implementation of an equation solver in the Boomerang decompiler which uses the above techniques. The presence of recursion complicates preservation analysis, as will be shown in Section 4.4.2.

4.3.1 Other Data Flow Anomalies

Decompilers need to be aware of several algebraic identities both to make the decompiled output easier to read, and also to prevent data flow anomalies.

Just as preserved locations appear to be used when for all practical purposes they are not, there are several algebraic identities which appear to use their parameters, but since the result is a constant, do not in reality use them. These include:

x - x = 0	x ^ x = 0 (^ = xor)	x | ~x = -1
x & 0 = 0	x & ~x = 0	x | -1 = -1

To these could be added $x \div x = 1$ and $x \bmod x = 0$ if it is known that $x \neq 0$. For each of these, a naive data flow analysis will erroneously conclude that x is used by the left hand side of each of these identities. The exclusive or and subtract versions are commonly used by compilers to set a register to zero. Decompilers need to be aware of such identities to prevent needless overestimating of uses. The constant result is shorter than the original expression, thereby making the decompiled output easier to read. Often the constant result can combine with other expressions to trigger more simplifications. This problem is readily solved by replacing appropriate patterns by their

constant values before data flow analysis is performed. These changes can be made as part of the typical set of "simplifications" that can be performed on expressions, such as $x + 0 = x$.

4.3.2 Final Parameters

Final parameters are locations live on entry after preservation analysis, dead code elimination, and the application of identities.

Equation 3.6 on page 84 states that the initial parameters are the locations live on entry at the start of the procedure after intersecting with a parameter filter. Two exceptions have been seen. Firstly, preserved locations are often not parameters, although they could be. Figure 4.23 shows how propagation and dead code elimination combine to expose whether preserved location *loc* is used as a parameter or not.

$\texttt{save}_1$:= $\texttt{loc}_0$	$\texttt{save}_1$:= $\texttt{loc}_0$	
...	...	...
use($\texttt{loc}_0$) // If present	use($\texttt{loc}_0$) // If present	use($\texttt{loc}_0$) // If present
...	...	...
$\texttt{loc}_{99}$:= $\texttt{save}_1$	$\texttt{loc}_{99}$:= $\texttt{loc}_0$	
return	return	return
(a) Initial	(b) After propagation	(c) After DCE

Figure 4.23: Analysing preserved parameters using propagation and dead code elimination.

Secondly, locations involved in one of the above identities are never parameters. For example, if the first use of register `eax` in a procedure appears to be used in a register which performs an exclusive-or operation on `eax` with itself, e.g. `eax` := `eax` ^ `eax`, then `eax` is not a parameter. Application of the identity $x \hat{} x = 0$ solves this problem.

Hence, the equation for final parameters is in fact the same as that for initial parameters (Equation 3.6), but parameters are only final after propagation, dead code analysis, and the application of identities.

4.3.3 Bypassing Calls

The main reason for determining whether locations are preserved is to allow them to bypass call statements in caller procedures.

Figure 4.24 shows the IR for a fragment of the recursive version of the running example in a little more detail than that of Figure 4.15.

```
int rcomb(int n, int r) {
   if (n <= r)
      return 1;
   double res = rcomb(n-1, r);
   res *= n;
   res /= (n-r);
   return (int)res;
}
```

(a) Source code.

```
 3 ebp_3 := esp_0 - 4
11 goto to L2 if eax_7 > m[r28_0 + 8]
12 m[esp_0 - 32] := 1
13 GOTO L2
L1:
14 eax_14 := m[esp_0+4]_0          ; n
23 m[esp_0-48]_23 := m[esp_0+8]    ; argument r
25 m[esp_0-52]_25 := eax_14-1      ; argument n-1
30 eax_30, ebp_30, m[ebp_30+8]_30 := CALL rcomb  ; Recurse
            Reaching definitions: ... esp= esp_0-56, ebp := esp_0-4, ...
79 st_79 := st_76 *f (double) m[ebp_30+8]_30   ; res *= n
L2:
191 ebp_191 := φ(ebp_3, ebp_30)
184 esp_184 := ebp_191 + 8
185 return
```

(b) Before bypassing. Many statements are removed for simplicity, including those which preserve and restore `ebp`.

```
...
30 eax_30 := CALL rcomb          ; Recurse
79 st_79 := st_76 *f (double) m[esp_0+4]_0 ; res *= n
L2:
191 ebp_191 := esp_0 - 4         ; φ-function now an assignment
184 esp_184 := esp_191 + 8 → esp_0 + 4 ; esp is preserved
185 return
```

(c) After bypassing.

Figure 4.24: Part of the IR for the program of Figure 4.15.

Note that the preservation of the stack pointer `esp` depends on the preservation of another register, the frame pointer `ebp`. The fact that the call is recursive is ignored for the present; the effect of recursion on preservation will be addressed soon. For the purposes of this example, it will be assumed that after preservation analysis, it is determined that `ebp` is unchanged and `esp` is incremented by 4. (The addition of 4 comes about from the fact that the stack is balanced throughout the procedure except

for the return statement at the end, which pops the 32-bit (4-byte) return address from the stack. X86 call statements have a corresponding decrement of the stack pointer by 4, where this return address is pushed. Hence, incrementing `esp` by 4 is the x86 equivalent of preservation.)

The ϕ-function now has two parameters, $\texttt{ebp}_3$ and $\texttt{ebp}_{30}$. $\texttt{Ebp}_{30}$ is the value of `ebp` after the call; since `ebp` was found to be preserved by the call, all references to $\texttt{ebp}_{30}$ can be replaced by the value that reaches the call, which is $\texttt{esp}_0\texttt{-4}$ (reaching definitions are stored in calls by a collector; see Section 4.5). Hence, both operands of the ϕ-function evaluate to $\texttt{esp}_0\texttt{-4}$. There is no longer any need for the ϕ-function, so it is replaced by an assignment as shown. Effectively, the value of `ebp` has *bypassed* the call (it is as if the call did not exist) and also bypassed the ϕ-function caused by the control flow merge at L2.

4.4 Recursion Problems

Recursion presents particular problems when determining the preservation of locations, and the removing of unused parameters and returns.

The presence of recursion in a program complicates the ordering of many decompilation analyses. The most important analysis affected by recursion is preservation analysis. Following subsections discuss how to determine the best order to process procedures in when they are involved with recursion, and how to perform the preservation analysis. A final subsection discusses the problem of removing unused parameters and returns, which is also complicated by recursion.

Decompilers have the whole input binary program available at once, unlike compilers, which typically see source code for only one of potentially many source modules. A decompiler therefore potentially has access to the IR of the whole program at once. Definitions and uses in a procedure depend on those in all child procedures, if any, with the exception that a procedure's returns depend on the liveness of all callers. As has been mentioned earlier, this must be done after all procedures are analysed. It makes sense therefore to process procedures in a depth first ordering (of the call graph, the directed graph of procedures, where a callee is a child of a caller). In a depth first ordering, child nodes are processed before parent nodes, so the callers have the data flow summary of callees (as per Section 3.4) available to them. The main summary information stored in callees that is of interest to callers are the modifieds set, possibly stored in the return statement of the callee, and the set of live locations at the callee entry point.

However, it is only possible to access the summary information of every callee if the call graph is a tree, i.e. there are no cycles in the call graph introduced by recursive procedures. Many programs have at least a few recursive procedures, and sometimes there is mutual recursion, which causes larger cycles in the call graph. Figure 4.25 shows an example of a program call graph with many cycles; there is self recursion and mutual recursion.

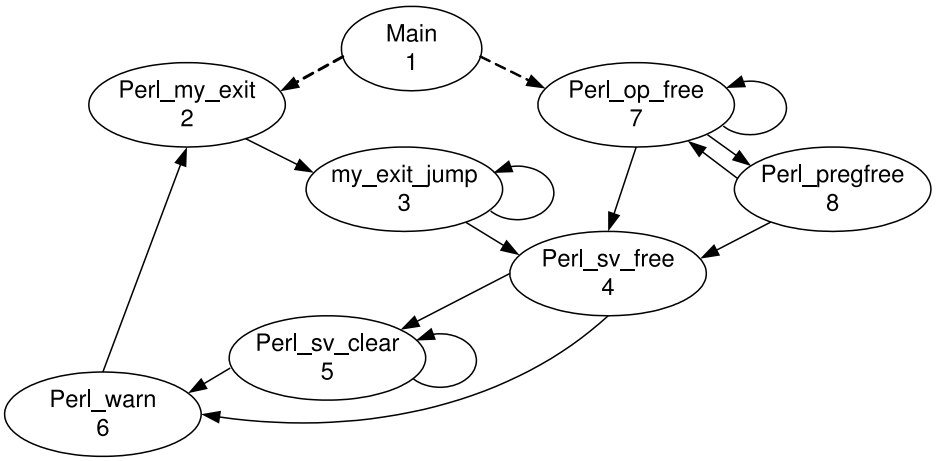


Figure 4.25: A small part of the call graph for the 253.perlbmk SPEC CPU2000 benchmark program.

From this program fragment, several cycles are evident, e.g.

$3 \rightarrow 3$ (direct recursion), $7 \rightarrow 8 \rightarrow 7$ (mutual recursion),

$2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 2$ (a long mutual recursion cycle), and

$2 \rightarrow 3 \rightarrow 4 \rightarrow 6 \rightarrow 2$.

These cycles imply that during analysis of these procedures, an approximation has to be made of the definitions and uses made by the callees in recursive calls. Since unanalysed indirect calls also have no callee summary available, they are treated similarly. Both types of calls will be referred to as childless calls. Section 3.6 concluded that it is safe to overestimate the definitions and uses of a procedure when there is incomplete information about its actual definitions and uses. Therefore, it can be assumed that all definitions reaching a recursive call are used by the callee (all locations whose definitions reach the call are considered live), and all live locations at the call are defines (defined by the callee), according to Equations 3.12 and 3.13 respectively on page 85. In effect, all childless calls (calls for which the callee is not yet analysed, including recursive calls and unanalysed indirect calls) become of this form:

```
<all> := CALL childless_callee(<all>)
```

where `<all>` is a special location representing all live locations (the leftmost `<all>`) and all reaching definitions (the rightmost `<all>`).

This is a quite coarse approximation, but it is safe. Preservation analysis is the main analysis to be performed considering together all the procedures involved in mutual recursion. In effect, preservation and bypassing refine the initially coarse assumption that everything is defined in the callee. For example, in Figure 4.24, the initial assumption that register `ebp` is defined at the recursive call is removed, once it is determined by preservation analysis that `ebp` is preserved by the call, and is therefore effectively not defined by the call.

Similarly, before preservation analysis, it is assumed that register `ebx` is used by the procedure `rcomb`, i.e. it is live at the call to `rcomb`. In reality, `ebx` is pushed at the start of `rcomb` and popped at the end, and not used as a parameter. In other words, it is not actually used by the call, so that definitions of `ebx` before the call not used except by the call are in reality dead code. After preservation analysis and dead code elimination, the real parameters of `rcomb` can be found (recall that preserved locations appear to be parameters until after dead code elimination). Similarly to the situation with definitions, the coarse assumption of all locations being used by the recursive call can be refined after preservation analysis and call bypassing.

All the procedures involved in a recursion cycle can have the usual data flow analyses (expression propagation, preservation analysis, call bypassing, etc) applied with the approximation of childless calls using and defining everything, which will result in a summary for the procedures (locations modified and used by the procedures).

4.4.1 Procedure Processing Order

An algorithm is given for determining the correct order to process procedures involved in mutual recursion, which maximises the information available at call sites.

Figure 4.26 shows another call graph. In this example, i should be processed before h, and h before c. Procedures involved in independent cycles such as *f-g-f* can be processed independently of the other cycles. Both f and g are processed together using the approximations mentioned above, and the data flow analyses are repeated until there is no change.

However, procedures such as j and k, while part of only one cycle, must be processed with the larger group of nodes, including b, c, d, e, j, k, and l. The difference arises because f and g depend only on each other, and so once processed together, all information about f and g is known before c is processed. Nodes j and k, however, are not

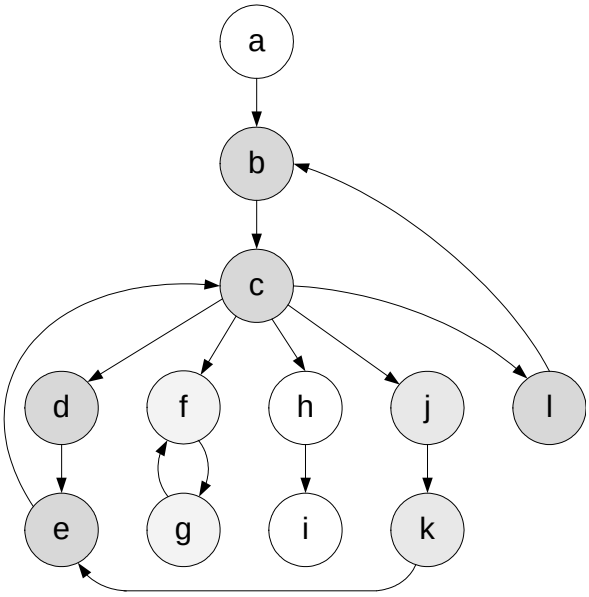


Figure 4.26: A call graph illustrating the algorithm for finding the correct ordering for processing procedures.

complete until e is complete; e depends on c, which in turn depends on j and k, but also l, which depends on b as well. The order of processing should be as follows: visit a, b, c, d, e, f and g; group process f and g; visit h and i; process i; process h; visit j, k, l and b; group process b, c, d, e, j, k, and l; process a.

The procedures that have to be processed as a group are those involved in cycles in the call graph, or sets of cycles with shared nodes. For these call graph nodes, there is a path from one procedure in the group to all the others, and from all others in the group to the first. In other words, each recursion group forms a strongly connected component of the call graph. In the example graph, there is a path from c to f, but none from f to c (the call graph is directed), so f is not part of the strongly connected component associated with c. Nodes l and e are part of the same strongly connected component, because there is a path from each to the other (via their cycles and the shared node c).

Algorithm 2 gives a general algorithm for finding strongly connected components in a graph. It has been shown that this algorithm is linear in the size of the graph [Gab00]. Decompilers typically do not store call graphs directly, but the algorithm can be adapted for decompilation by only notionally contracting vertices in the graph that are associated with cycles.

Algorithm 3 shows an algorithm for finding the recursion groups and processing them in the correct order.

When the child procedure does not cause a new cycle, the recursive call to *decompile*

Algorithm 2 General algorithm for finding strongly connected components in a graph. From Algorithm 10.1.5 of [GY04].

Input: directed graph $G = (V, E)$.
Output: strongly connected components of G.
Repeat until G has no more vertices:
  grow a depth-first path P until a sink or a cycle is found.
    sink s: mark $\{s\}$ as a strongly connected component and delete s from P and G.
    cycle C: contract the vertices of C.

performs a depth first search of the call graph. In the example call graph of Figure 4.26, procedure i is decompiled first, followed by h. When the algorithm recurses back to c, cycles are detected. For example, when child f of c is examined, no new cycles are found, so f is merely visited. When child g of f is examined, the first and only child f is found to have been already visited. At this point, *path* contains *a-b-c-f-g*. The fact that f is in *path* indicates a new cycle. The set *child*, initially empty, is unioned with the set of all procedures from f to the end of path (i.e. f and g). As a result, g is not decompiled yet, and *decompile* returns with the set $\{f, g\}$. At the point where the current procedure is f, the condition at Note 1 is true, so that f and g are analysed as a group. (The group analysis will be described more fully in Section 4.4.2 below.) Note that the cycle involving c and six other procedures is still being built. Assuming that c's children are visited in left to right order, a, b, c, d, and e have been visited but otherwise not processed.

When j's child e is considered, it has already been visited, but is not part of *path* (which then contains *a-b-c-j-k*). However, e has already been identified as part of a cycle *c-d-e-c*, so *c->cycleGrp* contains the set $\{c, d, e\}$. The first element of *path* that is in this set is c, so all procedures after c to the end of path (i.e. $\{j, k\}$), are added to *child*, which becomes $\{c, d, e, j, k\}$. Finally, when l's child b is found to be in path, the recursion group becomes $\{b, c, d, e, j, k, l\}$. Each of the procedures involved in this cycle have their *cycleGrp* members point to the same set.

4.4.2 Conditional Preservation

A method for extending the propagation algorithm to mutually recursive procedures is described.

Consider first a procedure with only self recursion, i.e. a procedure which calls itself directly. When deciding whether a location is preserved in some self recursive procedure p, there is the problem that at the recursive calls, preservation for location l will usually depend on whether l is preserved by the call. There may be copy or swap instructions such that l is preserved if and only if some other location m is preserved.

Algorithm 3 Finding recursion groups from the call graph.

```
/* Initial: cycleGrp is a member variable (an initially empty set of procedure pointers)
   path is an initially empty list of procedures, representing the call path from the
     current entry point to the current procedure, inclusive.
   Returns: a set of procedure pointers representing cycles associated with the current
     procedure and all its children */
function decompile(path)      /* path initially empty */
   child := new set of proc pointers   /* child is a local variable */
   append this proc to path
   for each child c called by this proc
      if c has already been visited but not finished then
         /* have new cycle */
         if c is in path then
            /* this is a completely new cycle */
            child := child ∪ {all procs from c to end of path inclusive}
         else
            /* this is a new branch of an existing cycle */
            child := c->cycleGrp
            find first element f of path that is in cycleGrp
            insert every proc after f to the end of path into child
         for each element e of child
            child := child ∪ e->cycleGrp
            e->cycleGrp := child
      else
         /* no new cycle */
         tmp := c->decompile(path)    /* recurse */
         child := child ∪ tmp
         set return statement in call to that of c
   if child empty then
      child := earlyDecompile()
      removeUnusedStatments()     /* Not involved in recursion */
   else
      /* is involved in recursion */
      find first element f in path that is also in cycleGrp
      if f = this proc then      /* Note 1 */
         recursionGroupAnalysis(cycleGrp)
         /* Do not add these procs to the parent's cycles */
         child := new set of proc pointers
   remove last element (= this proc) from path
   return child
```

This will be considered below. The location l will be preserved by the call if the whole procedure preserves l, but the whole procedure's preservation depends on many parts of the procedure, including preservation of l at the call. This is a chicken and egg problem; the infinite recursion has to be broken by making some valid assumptions about preservations at recursive calls.

Preservation is an all-paths property; a location is preserved by a procedure if and only if it is preserved along all possible paths in the procedure. When the preservation depends on the preservation of l in p at a recursive call c, the preservation at c will have to succeed for the overall preservation to succeed. A failure along any path will result in the failure of the overall preservation. Figure 4.27 shows a simplified control flow graph for the recursive version of the running example program, which illustrates the situation.

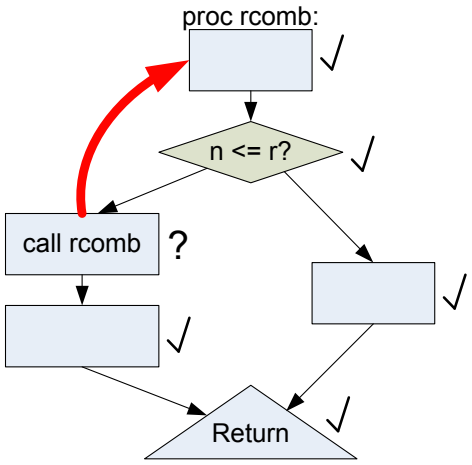


Figure 4.27: A simplified control flow graph for the program of Figure 4.15.

The ticked basic blocks are those which are on a path for which a particular location is known to be preserved. In some cases, several of the blocks have to combine, e.g. with a save of the location in the entry block and a restoration in the return block. However, the result is the same. The only block with a question mark is the recursive call. If that recursive call preserves the location, then the location is preserved for the overall procedure. However, the recursive call does not add any new control flow paths; it will only lead to ticked blocks or the recursive call itself. The call itself does not prevent any locations from being preserved, so the location truly is preserved.

Hence, for the purpose of preservation analysis, the original premise can safely be assumed to succeed until shown otherwise. In effect, the problem "is l preserved in recursive procedure p" is answered by asking the question "assuming that l is preserved by calls to p in p, is the location l preserved in p?". This assumption breaks the infinite recursion, so that the problem can be solved.

Consider now the situation where the preservation of l in p depends on the preservation of some other location m in p. This happens frequently, for example, when a stack pointer register is saved in a frame pointer register, so that preservation of the stack pointer in p depends on preservation of the frame pointer in p. It is not safe to assume that given the premise that l is preserved in p, m must automatically be preserved in

p; this has to be determined by a similar analysis.

The analysis recurses with the new premise, i.e. that both l and m are preserved. The difference now is that a failure of either part of the premise (i.e. if either l or m are found not to be preserved) causes the outer premise to fail (i.e. l is not preserved). There may be places where the preservation of m in p requires the preservation of l in p; if so, this can safely be assumed, as described above. There may also be places where the preservation of m in p depends on the preservation of m in p, which would lead to infinite recursion again. However, given that the current goal is to prove that m is preserved in p, by a similar argument to the above, it is safe to make that assumption, breaking the infinite recursion.

In order to keep track of the assumptions that may safely be made, the analysis maintains a set of required premises. This set could be thought of as a stack, with one premise pushed at the beginning of every analysis, and popped at the end. This stack is unique to the question of the preservation of l in p; other preservation questions require a separate stack. Each individual assumption (e.g. that m is preserved in p) is not valid until the outer preservation analysis is complete. In other words, each premise is a necessary but not sufficient condition for the preservation to succeed. Since the number of procedures and locations involved in mutual recursion is finite, this analysis will eventually terminate.

Consider now mutual recursion, where p calls q and possibly other procedures, one or more of which eventually calls p. When the preservation analysis requires that m is preserved by q, this premise is added to the stack of required premises, as above. The difference now is that the elements of the stack have two components: the location that is assumed preserved, and the procedure to which this assumption applies. In all other respects, the algorithm is the same.

Note how the preservation analysis requires the intermediate representation for each of the procedures involved in the mutual recursion to be available. In the example of Figure 4.26, when checking for a location in procedure f, there will at some point be a call to g, so the IR for g will need to be available. To prevent another infinite recursion, the analyses are arranged as follows. When Algorithm 3 finds a complete recursion group, "early" analyses are performed separately on each of the procedures in the recursion group. This decodes each procedure, inserting definitions and uses at childless calls as described earlier. Once each procedure has been analysed to this stage, "middle" analyses are performed on each procedure with a repeat until no change for the whole group of middle analyses. Preservation analysis the main analysis applied during middle analyses. When this process is complete, each procedure in the recursion group is finished off with "late" analyses, involving dead code elimination and the removal of

unused parameters and returns.

The repeat until no change aspect of middle analyses suggests that perhaps preservation analysis could be viewed as a fixedpoint dataflow algorithm. The fit is not a good one, but two possible lattices for this process are shown in Figure 4.28.

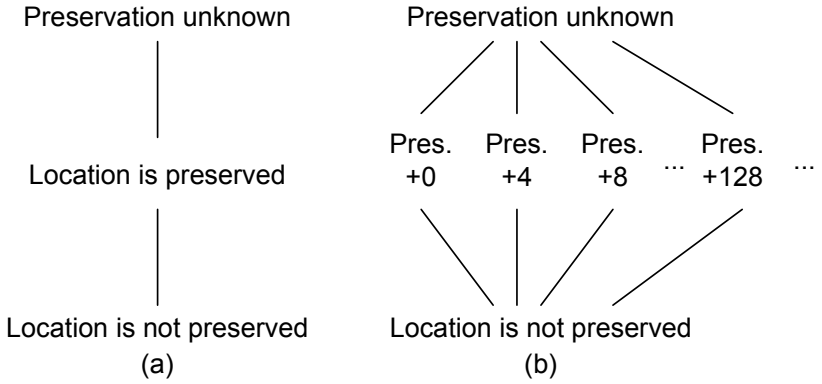


Figure 4.28: Two possible lattices for preservation analysis.

Figure 4.28(b) is remiscent of the textbook lattices for constant propagation as a fixed-point data flow transformation, or Fig. 1 of [Moh02]. "Pres. +0" indicates that the location of interest has been found to be preserved along some paths with nothing added to it; "Pres. +4" indicates that it is preserved but has 4 added to it, and so on. If along some path the location is found not to be preserved, or to be preserved by adding a different constant than the current state indicates, the location's state moves to the bottom state (not preserved).

Section 7.5.1 on page 235 demonstrates the overall preservation analysis algorithm (not lattice based) in practice, using the Boomerang decompiler.

4.4.3 Redundant Parameters and Returns

The rule that a location which is live at the start of a function is a parameter sometimes breaks down in the presence of recursion, necessitating an analysis to remove redundant parameters. An analogous situation exists for returns.

After dead code elimination, the combination of propagation and dead code elimination will have removed uses and definitions of preserved locations that are not parameters. As a result, no locations live at the entry are falsely identified as parameters due to saves of preserved locations. However, there may still be some parameters whose only use in the whole program is to supply arguments to recursive calls. Consider the example program in Figure 4.29.

```
int, int a(p, q) {
   ...                /* No use of p , q , r or s*/
   r, s := b(p, q);   /* b is always called by a */
   ...                /* No use of p , q , r or s */
   return r, s;
}
int,int b(p, q) {
   if (q > 1)        /* q is used other than in a call to a */
      r, s := a(p, q-1);
   else
      ...            /* No use of p , q , r , or s */
      print(r);      /* Use r */
   return r, s;
}
```

Figure 4.29: Example program illustrating that not all parameters to recursive calls can be ignored.

Procedures a and b are mutually recursive, and the recursion is controlled entirely in b. The question is what algorithm to use to decide whether any of the parameters of a or b are redundant and can therefore be removed. By inspection, p is a redundant parameter of both a and b, but q is not. Parameter p is used only to pass an argument to the call to b, and in b it is used only to pass an argument to the call to a. By removing the parameter p from the definitions of a and b and the calls to them, the program is simplified and becomes more readable. By contrast, q is used gainfully inside b, so it is also needed in a (to pass an argument to the call to b).

Note that by considering procedure a in isolation, there is no apparent difference in the redundancy or otherwise of p and q. Hence, for each procedure involved in recursion, each parameter has to be considered separately, recursive calls have to be followed, and only locations whose only uses in the whole program are as arguments to calls to procedures in the current recursion group can be considered redundant. For example, consider procedure a in Figure 4.29. It currently has two parameters, p and q. When a recursive call is encountered, the analysis needs to consider the callee (here b) and if necessary all its callees, for calls to a. During this analysis, parameter p is only used as an argument to a call to a, so it is redundant. However, q is used in the `if` statement, so it is not redundant.

A symmetrical situation exists with redundant returns. Considering procedure a in isolation, there is no apparent difference in the redundancy or otherwise of r and s. However, while s is used only by return statements in recursive procedures (the return statements in a and b), r is used gainfully (in the print statement in b). Hence, s is a redundant parameter but r is not. For each procedure involved in recursion, each return

component has to be considered separately (e.g. in `return c+d`, consider components c and d separately), recursive calls have to be followed, and only return components defined by calls to other procedures in the current cycle group that are otherwise unused can be considered redundant.

Once the intermediate representation for all procedures is available, (this is a "late" analysis, in the sense of Section 4.4.2), unused returns can be removed by searching through all callers for a liveness (use before definition) for each potential return location. Livenesses from arguments to recursive calls or return statements in recursive calls are ignored for this search. When the intersection of all considered livenesses is empty, the return can be removed from the callee. Removing the returns removes some uses in the program, so that some statements and some parameters may become unused as a result.

For example, when a newly removed return is the only use of a define in a call statement, it is possible that the last use of that define has now been removed; this will happen if no other call to that callee uses that location. Hence the unused return analysis has to be repeated for that callee. In this way, changes resulting from unused returns propagate *down* the call graph. When parameters are removed, the arguments of all callers are reduced. When arguments are reduced, uses are removed from the caller's data flow. As a result of this, the changes initiated from removing unused parameters propagate *up* the call graph. Hence, there is probably no best ordering to perform the removal of unused parameters and returns.

Section 7.6 on page 240 demonstrates this algorithm in practice, using the Boomerang decompiler.

4.5 Collectors

Collectors, a contribution of this thesis, extend the sparse data flow information provided by the Static Single Assignment form in ways that are useful for decompilers, by taking a snapshot of data flow information already computed.

The SSA form provides use-def information (answering the question "what is the definition for this use?") and if desired also def-use information (answering the question "what are the uses of this definition?"). Sometimes reaching definitions are required, which answers "what locations reach this program point, and what are their definitions?". The SSA form by itself can answer the second part of the question, but not the first part.

During the conversion of a program to SSA form, however, this information is available, but is discarded during conversion (the required information is popped off a stack).

Decompilers only need reaching definitions at two sets of program points: the end of procedures, and at call sites. Live variables are also needed at calls, and this information is also available during conversion. Despite this, SSA is generally considered in the literature to be unsuitable for backwards data flow information such as live variables [CCF91, JP93].

These observations are the motivation for *definition collectors*, which capture reaching definitions, and *use collectors*, which capture live variables. The collectors store information available during the standard variable renaming algorithm, when it iterates through all uses in a statement. In effect, collectors are exceptions to the usually sparse storage of data flow information in SSA form. The additional storage requirements are modest because they are only needed at specific program points: the end of procedures, and at call statements, as shown in Figure 4.30.

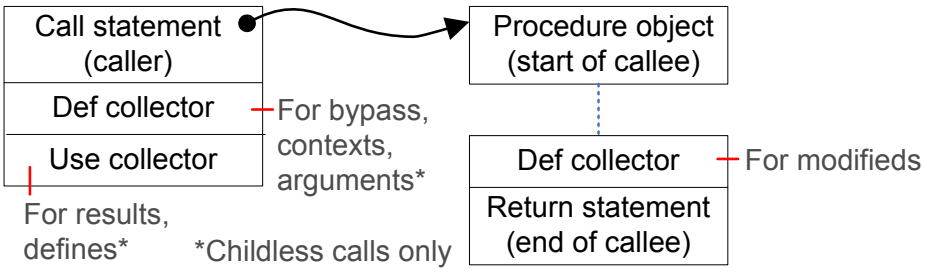


Figure 4.30: Use of collectors for call bypassing, caller and callee contexts, arguments (only for childless calls), results, defines (also only for childless calls), and modifieds.

Algorithm 4 shows the algorithm for renaming variables, as per [App02], modified for updating collectors, and for subscripting with pointers to defining statements rather than consecutive integers.

Where the algorithm reads "if can rename a", it is in the sense given in Section 4.2.3, i.e. a is not a memory location, or it is a suitable memory location and memory locations are being propagated for the current procedure.

Recall that at childless calls (call statements whose destination is unknown, or has not had complete data flow analysis performed to summarise the definitions and uses), all locations are assumed to be defined and used. To implement this successfully, Appel's algorithm is extended. When a childless call is encountered, a definition by the call is pushed to all elements of *Stacks* (note that *Stacks* is an array of stacks, one stack for each location seen so far). This slightly complicates removal of these elements, adding the requirement, absent in Appel's algorithm, that the statements be considered in reverse order at *Note 1*.

Algorithm 4 Renaming variables, with updating of collectors. Adapted from [App02].

```
Rename(n) =
  for each statement S in block n
    for each use of some location x in S
      if cannot rename x          /* Can memory expression can be renamed yet? */
        continue;
      if x is subscripted
        if x refers to a call c
          add x to use collector of c
      else                        /* x is not subscripted */
        if Stacks[x] is empty
          if Stacks[all] is empty
            def := null;          /* No definition, i.e. x_0 */
          else
            def := Stacks[all].top
        else
          def := Stacks[x].top
        if def is a call statement c
          add x to use collector in c
        replace the use of x with x_def in S
      if S is a call or return statement
        update definition collector in S
      for each definition of some location a in S
        push pointer to S onto Stacks[a]
      if S is a childless call
        for each location l in Stacks
          push pointer to S onto Stacks[l]
    for each successor Y of block n    /* Successor in the CFG */
      suppose n is the jth predecessor of Y
      for each φ-function in Y
        suppose the jth operand of the φ-function is a
        if Stacks[a] is empty then def := null
        else def := Stacks[a].top
        replace the jth operand with a_def
    for each child X of n              /* child in the dominator tree */
      Rename(X)
    for each statement S in block n in reverse order   /* Note 1 */
      for each definition of some location a in S
        if can rename a
          pop Stack[a]
      if S is a childless call
        for each location l in Stacks    /* Pop all definitions due to childless calls */
          if Stacks[l].top = S
            pop Stacks[l]
```

Collectors complicate the handling of statements containing them slightly. For example, when performing a visitor pattern on a call statement, which contains two collectors, should the expressions in the collectors visited? At times, this is wanted, but not at other times. As an example, it is desirable to propagate into the expressions on the right hand side of definition collectors, so in this case the right hand sides of definition collectors are treated as uses. However, when deciding if a definition is dead, these uses should be ignored.

4.5.1 Collector Applications

Collectors find application in call bypassing, caller/callee context translation, computing returns, and initial arguments and defines at childless calls.

As covered in Section 4.3.3, the definitions of locations reaching calls are needed for call bypassing. The definition collector in call statements provides this expression.

Recall from Section 3.4.2, an expression in the caller context such as m[esp_0-64] is converted to the callee context (as `m[`esp_0`+4]`) using the value esp_{call} of the stack pointer at the call statement. The definition collector in the call collects reaching definitions for every location, so it is simple to compute $\texttt{esp}_{call}$.

At childless calls, the arguments are initially all definitions that reach the call. These are provided by the definition collector at the call. Similarly, defines at a childless call are initially all locations live at the call. These are provided by the use collector at the call.

Call results are computed late in the decompilation process. For each call, the results are the intersection of the live variables just after the call (in the use collector of the call) and the defines of the call (which are the modifieds of the callee translated to the context of the caller). The final returns of the callee depend on the union of livenesses (obtained from the use collectors in the callers) over all calls. Thus, if a procedure defines a location that is never used by any caller, it is removed from the returns of the callee.

4.6 Related Work

With suitable modifications to handle aggregates and aliases well, the SSA form obviates the need for the complexity of techniques such as recency abstraction.

Balakrishnan and Reps propose "recency-abstraction" as a way of analysing heap-allocated storage in executable programs [BR06]. Figure 4.31 reproduces an example

from that paper.

```
void foo() {
  int **pp, a;
  while(...)  {
    pp =
      (int*)malloc(sizeof(int*));
    if(...)
      *pp = &a;
    else {
      // No initialization of *pp
    }
    **pp = 10;
  }
}
```

(a)

```
void foo() {
  int **pp, a, b;
  while(...)  {
    pp =
      (int*)malloc(sizeof(int*));
    if(...)
      *pp = &a;
    else {
      *pp = &b;
    }
    **pp = 10;
  }
}
```

(b)

Figure 4.31: The weak update problem for malloc blocks. From Fig. 1 of [BR06].

The problem is that for representations without the single assignment property, information about assignments such as *pp= in the example are summary nodes; they must summarise all assignments to `*pp` in the program. To ensure soundness, value analysis must always over-approximate the set of possible values, so the summary nodes are initialised for all variables to the special value $\top$, representing all possible values. All updates to summary nodes must be weak updates, so the values of the nodes never change. If the information for `*pp` was initialised to $\emptyset$ (the empty set), information such as "`*pp` points to `a`" can be established, but in the case of Figure 4.31(a) the fact that `*pp` is not initialised along some paths has been lost, and this is the very kind of situation that the authors want to capture.

The authors use the elaborate recency-abstraction to overcome this problem while maintaining soundness. However, ignoring alias issues for a moment, single assignment representations such as the SSA form only have one assignment to each variable (or virtual variable, such as `*pp` in the example). In such a system, all updates can be strong, since the one and only assignment (statically) to each variable is guaranteed to change it to the value or expression in the assignment. For example, let $\texttt{pp}_1$ be the variable assigned to at the malloc call. $\texttt{pp}_1$ always points to the last heap object allocated at this site. Following $(*\texttt{pp}_1)_1$= `&a` and $(*\texttt{pp}_1)_2$ = `&b`, the two versions of $(*\texttt{pp}_1)$ always point to `a` and `b` respectively.

Information about each variable can be initialised to $\top$ as required for soundness. Following a heap allocation, all elements of the structure (in this case there is only one, *pp), are assigned $\top$ (e.g. $(*\texttt{pp}_1)_0 = \top$). In Figure 4.31(a), the value for `*pp` that is

assigned the value 10 will be the result of a ϕ-function with $(*pp_1)_1$ and $(*pp_1)_0$ as operands; the value sets for these are `&a` and $\top$ respectively. The resultant variable will have a value set of $\top$ (i.e. possibly undefined). In Figure 4.31(b), the equivalent value for `*pp` will be the result of a ϕ-function with $(*pp_1)_1$ and $(*pp_1)_2$, which have values sets `&a` and `&b` respectively. The result will have the value set {`&a`, `&b`} (i.e. `*pp` could point to `a` or to `b`). Figure 4.32 shows the example of Figure 4.31(a) in SSA form.

```
void foo() {
  int **pp, a;
  while(...)  {
    pp1 = (int*)malloc(sizeof(int*));
    *(pp1)0 = ⊤;          // Value set = {⊤}
    if(...)
      (*pp1)1 = &a;       // Value set = {&a}
    else {
      // No initialization of *pp
    }
    (*pp1)2= φ((*pp1)0, (*pp1)1);   // Value set = {⊤}
    *(*pp1)2 = 10;
  }
}
```

Figure 4.32: The code of Figure 4.31(a) in SSA form.

This considerable advantage comes at a cost; information is stored about every version of every variable. While the SSA form solves the simple example above, as usually applied in compilers the SSA form needs to be extended to handle aggregates well. More importantly, variables such as $*pp_1$ are only uniquely assigned to if no other expression both aliases to $*pp_1$ and is assigned to. If the SSA form can be extended to accommodate the complications of aliasing and aggregates, it shows great promise for obviating the need for techniques such as recency-abstraction.

4.7 Other Representations

Many alternative intermediate representations exist, especially for optimising compilers, but few offer real advantages for decompilation.

Many intermediate representations (IRs) for program transformations have been suggested, mainly to facilitate optimisations in compilers. The Static Single Assignment form (SSA form) is one such IR, and there have been several extensions to the basic SSA form.

The Gated Single Assignment form (GSA form) [TP95] is an extension of the SSA form designed to make possible the interpretation of programs in that form. This is not

required in a decompiler, since copy statements can be added to make the program executable. However, it is possible that GSA or similar forms might reduce the number of copy assignments, and hence increase readability.

4.7.1 Value Dependence Graph

The VDG and other representations abstract away the control flow graph, but the results for decompilation are not compelling.

There is a series of intermediate representations that are based on the idea of abstracting away the Control Flow Graph (CFG). A notable example is the Value Dependence Graph (VDG) [WCES94]. The key difference between VDG and CFG-based IR is that VDG is a parallel representation that specifies a partial order on the operations in the computation, whereas the CFG imposes an arbitrary total order. The authors claim that the CFG, for example by naming all values, "gets in the way of analysing the underlying computation".

The running example in their paper shows seven different types of optimisations, all facilitated by their representations, which is impressive. However, many optimisations such as loop invariant code motion do not apply to decompilation; the best position for an assignment in source code is the position that maximises readability. One optimisation is "name insensitive global common subexpression elimination". Decompilers based on the IR described in this chapter are already inherently "name insensitive", in the sense that variables such as *acopy* (a copy of variable a) in their example are automatically replaced with the original definition by the process of expression propagation. Similarly, "code motion into the arm of an `if` statement" is automatically performed by expression propagation.

VDG does reveal opportunities for parallelisation that might be useful for future decompilers that emit source code in some language that expresses parallelism directly. However, it could be argued that parallelisation opportunities are the job of the compiler, not the programmer, and therefore do not belong in source code.

4.7.2 Static Single Information (SSI)

Static Single Information, an extension of SSA, has been suggested as an improved intermediate representation for decompilation, but the benefits do not outweigh the costs.

Using the SSI form, as opposed to SSA form, improves a decompiler's type analysis in very rare circumstances. One such case involves a pair of load instructions which are

hoisted ahead of a compare and branch. Figure 4.33 shows the source and machine code of an example.

```
#define TAG_INT 1
#define TAG_PTR 2
typedef struct {
  int tag;
  union {
    int a;
    int *p;
  } u;
} tagged_union;
```

```
int f(tagged_union *tup) {
  int x;
  if (tup->tag == TAG_INT)
    x = tup->u.a;
  else
    x = *(tup->u.p);
  return x;
}
```

(a) Source code

```
   // Calling conventions:  argument passed in register r0,
   // result returned in register r0
// Expected output from a    // Possible output from an optimising
// non-optimising compiler   // compiler
f:                           f:
  LDW r1 ← [r0]                LDW r1 ← [r0]
  CMP r1, #1                   LDW r0 ← [r0,#4]   // note that the load
  BNE label                    CMP r1, #1         // has been hoisted
  LDW r0 ← [r0,#4]             BEQ return         // above the compare
  JMP return                   LDW r0 ← [r0]      // and branch
label:                       return:
  LDW r0 ← [r0,#4]             RET
  LDW r0 ← [r0]
return:
  RET
```

(b) Machine code

Figure 4.33: An example program from [Sin03].

The problem is that the hoisted load instruction loads either a pointer or an integer, depending on the path taken in the `if` statement. In the non-hoisted version, while `r0` is sometimes used as a pointer and as an integer, there are separate definitions for the two different types. With its concept of having only one definition of a variable, SSA is very good at separating the different uses of the register, and sensible output can be emitted, with no type conflicts. The decompiled program will require casts or unions, but so did the original program.

In the compilation with the hoisted load, the one (underlined) load instruction (and therefore one definition) produces either an integer or a pointer, and there is a path in the program where that value is returned. In essence, the machine code conflates the two loads, and uses extra information (the tag field) to perform the right operations on the result. When the value loaded is a pointer, control always proceeds to the last

load instruction, which dereferences the pointer and loads an integer. It appears that the program could take the other path, thereby returning the pointer, but this never happens, due to the design of the program.

By itself, the SSA form combined with any type analysis is not powerful enough to infer that the type of the result is always an integer. This is indicated by the type equation for the return location $\mathrm{r0}_d$ of Figure 4.34(a), which is $\mathcal{T}(\mathtt{r0}_d) = \alpha \vee \alpha*$. This equation could be read as "the type of variable $\mathrm{r0}_d$ is either an alpha, or a pointer to an alpha". (α is a type variable, i.e. a variable whose values are types like float or int*. It appears here because until the return location is actually used, there is nothing to say what type is stored at offset 4 in the structure; as soon as the return value is used as an integer, α is assigned the value int.)

```
LDW r1a←m[r0a]
LDW r0b←m[r0a+4]
CMP r1a, #1

   LDW r0c←m[r0b]
r0d←φ(r0b, r0c)

T(r0d) = α ∨ α*
```

(a) SSA form

```
LDW r1a←m[r0a]
LDW r0b←m[r0a+4]
CMP r1a, #1
   r0x,r0y←σ(r0b)
   LDW r0c←m[r0y]
r0d←φ(r0x, r0c)

T(r0d) = α
```

(b) SSI form

```
LDW r1a←m[r0a]
LDW r0b←m[r0a+4]
CMP m[r0a], #1

   LDW r0c←m[m[r0a+4]]
r0d←φ(r0b, r0c)

T(r0d) = α
```

(c) SSA form with propagation

Figure 4.34: IR of the optimised machine code output from Figure 4.33.

As with all three representations, $\mathcal{T}(\mathtt{r1}_a) = \mathsf{int}$ and $\mathtt{r0}_a$ has the type struct{int, union{α, α*}}*, in other words, $\mathtt{r0}_a$ points to a structure with an int followed by a union of an α and a pointer to an α.

When the program is converted to SSI form, as shown in Figure 4.34(b), the decompiler spends extra time and space splitting the live range of every live variable at every control flow split (branch or switch instruction). For example, $\mathtt{r0}_b$ is split into into $\mathtt{r0}_x$ and $\mathtt{r0}_y$, gambling that something different will happen to the split variables to justify this extra effort. In this case the gamble succeeds, because $\mathrm{r0}_y$ is used as a pointer while $\mathrm{r0}_x$ remains used only as an integer. This allows each renamed variable to be typed separately, and the final type for $\mathrm{r0}_d$ is α, as it should be. Hence this program is one case where the SSI form has an advantage over the SSA form.

However, this advantage disappears with propagation, as shown in Figure 4.34(c). While SSI splits all live variables at every control flow split, propagation removes the use of $\mathtt{r0}_b$ as a temporary pointer value, placing the memory indirection directly where it is used. In other words, the single "memory of operator" from the single load instruction is copied into two separate expressions, neatly undoing the hoisting, and neatly

avoiding the consequent typing problem. The variables $\texttt{r0}_b$, $\texttt{r0}_c$, and most importantly $\texttt{r0}_d$ all have a consistent type, α, as expected.

Once again, the combination of the SSA form and expression propagation is found to be quite useful for a decompiler. The extra overhead of the SSI form (the time and space to create the σ-functions) has not been found to be useful for decompilation in this case.

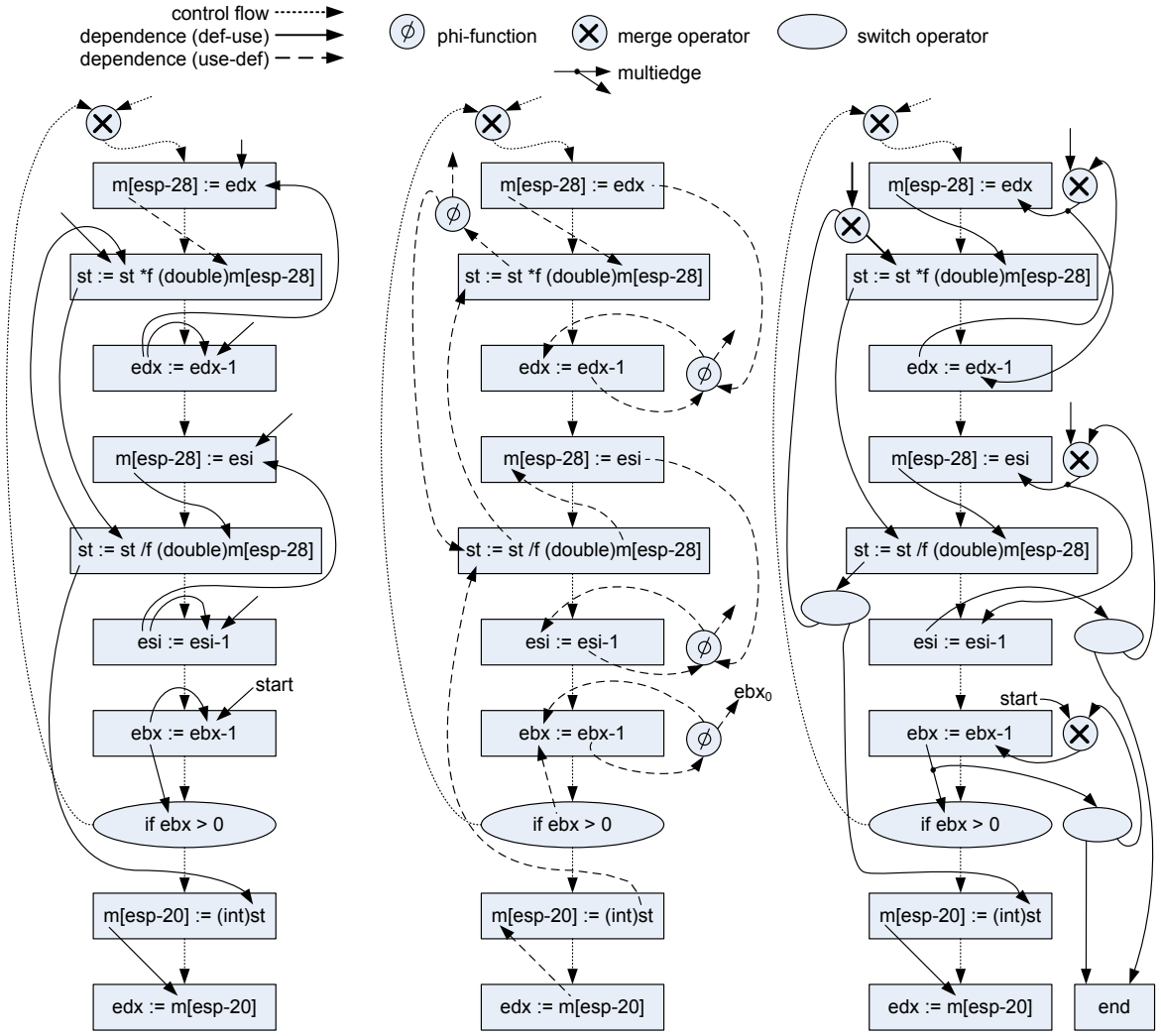
One area where the extra information could potentially be useful for a decompiler is in the analysis of array sizes. For example, if an array index takes the values 0 through 9 in a loop, then the version of the index variable in the loop can be shown to take the range 0-9, while the same variable outside the loop could take other values (typically only one value, 10, after the loop exits). This information could potentially be used to correctly emit the initialised values for an initialised array. However, decompilers need to be able to do the same thing even if the index variable has been optimised away, effectively deducing the existence of an induction variable. The SSA form appears to be adequate for this purpose [Wol92].

4.7.3 Dependence Flow Graph (DFG)

The Dependence Flow Graph shows promise as a possible augmentation for the Static Single Assignment form in machine code decompilers.

The Dependence Flow Graph (DFG) can be thought of as an extension of the Static Single Assignment form [JP93]. In this form, the emphasis is on recording dependencies, both from uses to definitions and vice versa. Figure 4.35 shows the main loop of the running example in various intermediate representations.

Note that the arrows are reversed for the SSA form, since in that form, the most compact encoding of the dependencies is for each use to point to the unique definition. This encoding is used in [KCL+99]. However, the dependencies could be recorded as a list of uses for each definition, as per the earlier SSA papers (e.g. [CFR+91, WZ91]), or both. In the DFG literature, dependencies are described as "pairs of nodes", so in the DFG form, uses could point to definitions (sometimes via merge operators), and/or definitions could point to their uses (sometimes via switch operators or multiedges). Switch operators differ from multiedges in the DFG form in that there is a second input (not shown in Figure 4.35) that carries information about the uses at the target of the switch operator. For example, at the true output of the switch operator for `ebx`, it is known that `ebx>0`. For the other switch operators, the switch predicate does not give directly useful information, although it may be possible for example to deduce that



(a) CFG with def-use chains (b) SSA form (c) DFG form

Figure 4.35: A comparison of IRs for the program of Figure 4.1. Only a few def-use chains are labelled, for simplicity. After Figure 1 of [JP93].

`esi` is always greater than zero, so that a divide by zero error or exception will never occur in the loop.

The presence of switch operators and multiedges makes the DFG more suitable for backwards data flow analyses than SSA form.

The original control flow graph still exists in the DFG form, so no extra step is necessary to construct a CFG for high level code generation, which would normally require one. The DFG form can also readily be converted to a dataflow graph for execution on a so-called dataflow machine.

The availability of uses for each definition, albeit at some cost for maintaining this information, may make it suitable as a decompiler intermediate representation.

4.8 Summary

The Static Single Assignment form has been found to be a good fit for the intermediate representation of a machine code decompiler.

It has been demonstrated that data flow analysis is very important for decompilation, and that the Static Single Assignment form makes most of these considerably easier. The propagation of memory expressions has been shown to be difficult, but adequate solutions have been given.

Preservation analysis was found to be quite important, and in the presence of recursion, surprisingly difficult. An algorithm has been given which visits the procedures involved in recursion in the correct order, and correctly analyses the preservations.

Definition and use collectors have been introduced to take a snapshot of already computed data flow information that is of particular importance in decompilation.

Finally, an overview of related intermediate representations has been presented.

Overall, the SSA form has been found to be a good fit for the intermediate representation of a machine code decompiler. The main reasons are the extreme ease of propagation, compact storage of use-definition information, ease of reasoning for preservation analysis, simplicity of dead code elimination, and a general suitability for identifying parameters and returns.

The SSA form is conventionally used only on scalar variables. Although numerous attempts have been made to extend it to handle arrays and structures, an extension that works well for for machine code programs has yet to be found.

Chapter 5

Type Analysis for Decompilers

The SSA form enables a sparse data flow based type analysis system, which is well suited to decompilation.

Compilers perform type analysis to reject invalid programs early in the software development process, avoiding more costly correction later (e.g. in debugging, or after deployment). Some source languages such as C and Java require type definitions for all variables. For these languages, the compiler checks the mutual consistency of statements which have type implications at compile time in a process called *type checking*. Since much type information is required by the compiler of the decompiled source code, and no explicit type information exists in a typical machine code program, a machine code decompiler has considerable work to do to recover the types of variables. This process is called *type analysis* for decompilers.

Other languages such as Self, C/C++ and Smalltalk require few type definitions, variables can hold different types throughout the program, and most type checking is performed at runtime. Such languages are dynamically type checked. Static (compile time) type analysis is often still performed for these languages, to discover what types a variable could take at runtime for optimisation. This process, called *type inferencing* or *type reconstruction*, is harder for a compiler than type checking [KDM03]. Since C-like languages are more common, this more difficult case will not be considered further here.

The type analysis problem for decompilers is to associate each piece of data with a high-level type. The program can reasonably be expected to be free of type errors, even though some languages such as C allow casting from one type to almost any other. Various different pieces of data require typing: initialised and uninitialised global data, local variables, heap allocated variables, and constants.

Type analysis is conceptually performed after data flow analysis and before control flow

analysis, as shown in Figure 5.1.

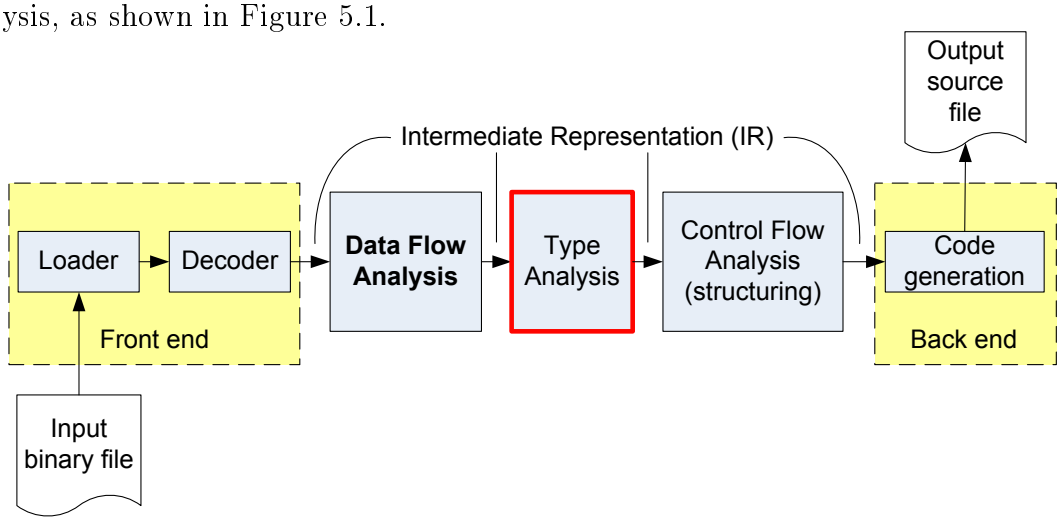


Figure 5.1: Type analysis is a major component of a machine code decompiler.

Type analysis, even more so than data flow analysis, is vastly more heavily discussed in the literature from the point of view of compilers. Detailed consideration is given here on the nature of types in machine code programs, and the requirements for types in the decompiler output.

The type information present in machine code programs can be regarded as a set of constraints to be solved, or a set of operations to be processed by something similar to data flow analysis. This chapter presents a sparse data flow based type analysis based on the Static Single Assignment form (SSA form). Since type information can be stored with each definition (and constant), there is ready access to the current type from each use of a location. The memory savings from this sparse representation are considerable.

Addition and subtraction instructions, which generate three constraints each, require a different approach in a data based type analysis system. Similarly, arbitrary expressions deserve special attention in such a system, as there is nowhere to store types for intermediate expression results.

In any type analysis system for machine code decompilers, there is a surprisingly large variety of memory expressions that represent data objects from simple variables to array and structure members. These will be enumerated in detail for the first time.

Section 5.1 lists previous work that forms the basis for this chapter. Section 5.2 introduces the nature of types from a machine code point of view, and why they are so important. The sources of type information are listed in Section 5.3. Constants require typing as well as locations, as shown in Section 5.4. Section 5.5 discusses the limitations of solving type analysis as a constraint satisfaction problem. The special requirements of addition and subtraction instructions are considered in Section 5.6, and revisited in Section 5.7.4. An iterative data flow based solution is proposed in Section 5.7, and the

benefits of an SSA version are given in Section 5.7.2. The large number of memory location patterns is enumerated in Section 5.8. Decompilers need to process data as well as code, and this process goes hand in hand with type analysis, as discussed in Section 5.9. Section 5.10 mentions some special types useful in type analysis, Section 5.11 discusses related work, and Section 5.12 lists work that remains for the future. Finally, Section 5.13 summarises the contributions of the SSA form to type analysis.

5.1 Previous Work

The work of Mycroft and Reps et al. have some limitations, but they laid the foundation for type analysis in machine code decompilation.

Mycroft's paper [Myc99] was the first to seriously consider type analysis for decompilation. He recognised the usefulness of the SSA form to begin to "undo register-colouring optimisations", i.e. to separate colocated variables. He notes the problem caused by array indexing where the compiler generates more than one instruction for the array element access. Section 5.5.1 discusses this limitation.

Mycroft's work has a number of other limitations, e.g. global arrays do not appear to have been considered. However, the work in this chapter was largely inspired by Mycroft's paper.

Reps et al. describe an analysis framework for x86 executables [RBL06, BR05]. Their goal is to produce an IR that is similar to that which could be produced from source code, but with low-level elements important to security analysis. They ultimately use tools designed to read source code for browsing and safety queries. Figure 5.2 gives an overview.

One of the stated goals is type information, but the papers do not spell out where this information comes from, apart from a brief mention of propagating information from library functions. Their papers mention decompilation as a potential application, but it is not one of their primary areas of interest.

Three main analyses are used: value set analysis (VSA), a form of value analysis; affine relation analysis (ARA), and aggregate structure identification (ASI). VSA finds an overapproximation of the values that a location could take at a given program point. ARA is a source code analysis modified by the authors for use with executable programs. It finds relationships between locations, such as index variables and running pointers. ASI recovers the structure of aggregates such as structures and arrays, including arrays of structures.

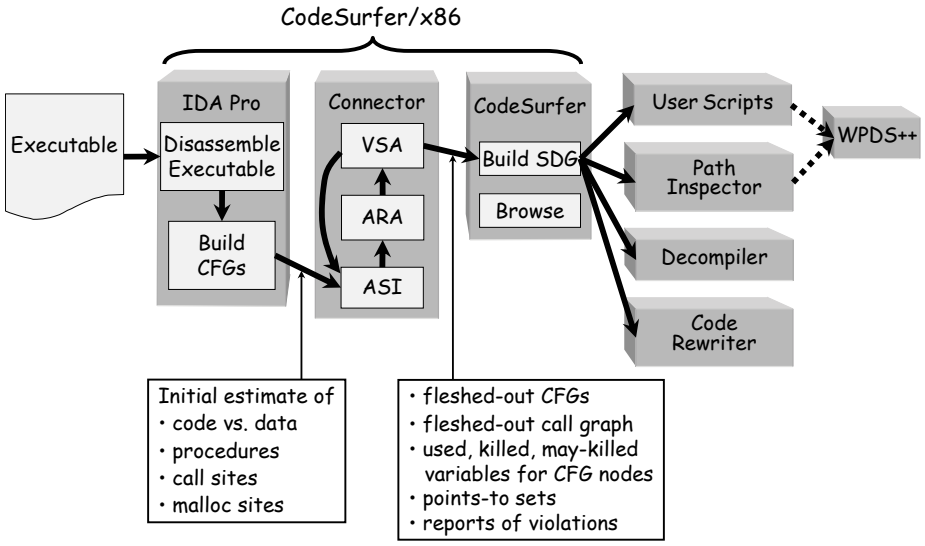


Figure 5.2: Organisation of the CodeSurfer/x86 and companion tools. From [RBL06].

The authors report results that are reasonable, particularly compared to the total failure of most tools to analyse complex data structures in executable programs. However, only 55% of virtual functions were analysed successfully [BR06], and 60% of the programs tested have one or more virtual functions that could not be analysed [BR06, BR05]. 72% of heap allocated structures were analysed correctly (i.e. the calculated structure agreed with the debugging information from the compiler), but for 20% of the programs, 12% or less were correct [BR05]. As will be shown later, the form of memory expressions representing accesses to data items, particularly to aggregate elements, is surprisingly complex. In particular, it is not clear that their analyses can separate original from offset pointers (Section 1.5.3 on page 19). It is hoped that an analysis that takes into account all the various possible memory expression forms will be able to correctly analyse a larger proportion of data accesses, while also finding types for each data item.

5.2 Type Analysis for Machine Code

Type information encapsulates much that distinguishes low level machine code from high level source code.

The nature and uses of types will be considered in the next sections from a machine code point of view.

5.2.1 The Role of Types

Types are assertions; they partition the domain of program semantics, and partition the data into distinct objects.

When a programmer declares variables to be of type integer or class Customer in a statically type checked language, assertions are made about what values the variables can take, and what operations can usefully be performed on them. Part of the domain of program semantics applies to variables of type integer (e.g. a shift left by 2 bits), and others (e.g. a call to method `getName()`) apply to objects of type class Customer. When a program is found to be applying the wrong semantics to a variable, it can be shown that the program violates the typing rules of the language, so an error message can be issued at compile time.

Types include information about the size of a data object. Types partition the data sections from blocks of bytes to sets of distinct objects with known properties.

In a machine code program, type declarations usually have been removed by the compiler. (An exception is debugging information, inserted during software development, which may still be present in some cases.) When decompiling the program, therefore, the fact that a variable is at some point shifted left by two is a strong indication that it should be typed as integer and not as class Customer.

Similarly, when four bytes of data are used as integer, those four bytes become distinct from other data objects in the data sections. It is unlikely that these four bytes are part of another data object, a string for example, that happens to precede it in the data section. In this way, the decompiler can effectively reconstruct the mapping from addresses to types present in the symbol table of the original compiler.

The view of types as assertions leads to the idea of generating constraints for each variable whenever it is used in a context that implies something about its type. Not all uses imply full type information. For example, a 32-bit copy (register to register move) instruction only constrains the size of the type, not whether it is integer, float, or pointer. The above assumes that pointers are 32 bits in size. Programs generally use one size for pointers, which can be found by examining the binary file format. The examples in this chapter will assume 32-bit pointers; obviously other sizes can be accommodated.

Some operations imply a basic type, such as integer, but no detail of the type, such as its *signedness* (signed versus unsigned). The shift left operator is an example of this. Operators such as shift right arithmetic imply the basic type (integer), and the signedness (in this case, signed). The partial nature of some type information leads to the notion of a hierarchy or lattice of types. The type unsigned short is more precise than

short integer (where the signedness is not known), which is more precise than size16, which in turn is more precise than $\top$ (no type information at all).

In object oriented programs, an important group of types are the class types. It is desirable to know, for example, whether some pointer is of type Customer* or type Employee*. Such type information does not come directly from the semantics of individual instructions, but from class hierarchy analysis.

A program expressed in a high level language must contain some type declarations to satisfy the rules of the language. However, it is possible to not use the type system except at the most trivial level. Consider for example a binary translator which makes no attempt to comprehend the program it is translating, and copies the data section from the original program as one monolithic block of unstructured bytes. Suppose also that it uses the C language as a target machine independent assembler; the UQBT binary translator operates this way [CVE00, CVEU+99, UQB01]. Such "low level" C emitted by the translator will contain type-related constructs, since the C language requires that each variable definition is declared with a type, but all variables are declared as integer, and casts are inserted as needed to reproduce the semantics of the original program. Memory is referenced by expressions of the form `*(<type>*)(<address expression>)`.

Code such as this is essentially a low level disassembly expressed in a high level language. One of the main features absent from this kind of code is real type information for the data.

5.2.2 Types in High Level Languages

Types are essential to the expression of a program in high level terms: they contribute readability, encapsulate knowledge, separate pointers from numeric constants, and enable Object Orientation.

From the above discussion, it is clear that types are an essential feature of code written in high level languages, and without them, readability will be extremely poor.

The separation of pointers and numeric constants is implicit in deciding types for variables. An immediate value in an instruction (which in a disassembly could represent the address of some structure in memory, an integer number, or a floating point number) is clearly a pointer or constant once assigned a type. Recall that in Section 1.5, separating pointers from numeric constants is one of the fundamental problems of reverse engineering from machine code.

One of the differences between machine code programs and their high level language equivalents is that features such as null pointer checks and array bounds checks are

usually implicitly applied by the compiler. In other words, they are present at the machine code level, but a good decompiler should remove them. Type analysis is needed for this removal; for a null pointer check to be removed, the analysis must find that a variable is a pointer. Similarly, arrays and array indexes must be recognised before array bounds checks can be removed. It could be argued that if the null pointer or array bounds checks are removed by a decompiler but were present in the original source code, that the decompiled program is still correct.

Recursive types are types where part of a type expression refers to itself. For example, a linked list may have an element named `Next` which is a pointer to another element of the same linked list. Care must be taken when dealing with such types. For example, a simple algorithm for printing a type as a string is as follows: when encountering a pointer, emit a star and recurse with the child of the pointer type. This works for non-recursive types, but fails for the linked list example (it will attempt to emit an infinite series of stars). This problem is of course not unique to decompilers.

5.2.3 Elementary and Aggregate Types

While elementary types emerge from the semantics of individual instructions, aggregate types must at times be discovered through stride analysis.

Most machine instructions deal with elementary (simple) types. Such types include integers and floating point numbers. Enumerations and function pointers are represented at the machine code level as integers. Aggregate types are combinations of elementary types: arrays, structures, and unions of elementary types.

The other main class of types (other than elementary and aggregate types) are the pointers. These will be considered in more detail in the sections that follow.

Aggregate types are usually handled at the machine code level one element at a time. The few machine instructions which deal with aggregate types, such as block move or block set instructions, can be broken down into more elementary instructions in a loop. While machine instruction semantics will often determine the type of an elementary data item, aggregate types can generally only be discovered as emerging from the context of elementary instructions, e.g. elementary operations performed in a loop, or some elementary operation performed at a fixed offset from a pointer.

Figure 5.3 shows source and machine code that uses elementary and aggregate types. Note how on a complex instruction set such as the Intel x86, objects of elementary types are accessed with simple addressing modes such as `m[r1]`, while the aggregate types use more complex addressing modes such as `m[r1+r2*S]` (addressing mode `(r1,r2,S)`) and `m[r1+K]` (`r1` and `r2` are registers, and `S` and `K` are constants).

```
void process(int& i,
  float& f, int* a, s* ps) {
  i += 1;

  f += 2.;

  a[i] += 3;

  ps->b += 4;
```

(a) Source code

```
mov  0x8(%ebp),%edx   ; pi
mov  (%edx),%eax      ; i
inc  %eax             ; i+1
mov  %eax,(%edx)      ; save, eax=i
mov  0xc(%ebp),%ecx   ; pf
fld  0x402000         ; 2.0
fadd (%ecx)           ; f+2.0
fstp (%ecx)           ; store
mov  0x10(%ebp),%ebx  ; pa
add  $3,(%ebx,%eax,4) ; a[i] += 3
mov  0x14(%ebp),%esi  ; ps
add  $4,4(%esi)       ; ps->b += 4
```

(b) Machine code

Figure 5.3: Elementary and aggregate types at the machine code level.

5.2.4 Running Pointers

While running pointers and array indexing are equivalent in most cases, running pointers pose a problem in the case of an initialised array.

C programmers are aware that it is possible to access an array using either indexing or by incrementing a pointer through the array. Generally, pointers are more efficient than indexing; as a result, a program written with indexing may be converted by an optimising compiler to equivalent code that manipulates pointers. This implies that a decompiler is free to represent array handling code either way. Users may prefer one representation over the other. It could be argued that the representation can safely be left to a runtime option or an interactive user choice. Alternatively, one representation could always be produced, and a post decompilation transformation phase used if the other representation is desired.

The use of running pointers on arrays has an important implication for the recovery of pre-initialised arrays. Simplistic type analysis expecting to see indexing semantics for arrays (e.g. [Myc99]) may correctly decompile the code section of a program, but fail to recover the initial values for the array. Figure 5.4 illustrates the problem.

In Figure 5.4(a), the type analysis discovered an array, and in order to declare the array properly, is prompted to analyse the size of the array. If it is in a read-only section of the binary file, its size and type can be used to declare initial values for the array (not shown in the example). In Figure 5.4(b), the type of p is char*, and the type of the constants is also char*. The fact that address 10 000 is used as a char may prompt one char to be declared as a character, and if in a read-only section of the binary file, it may be given an initial value. Note that there is no prompting to declare the other nine values as type char. Worse, the constant 10 010 is used as type char*, which may

```
char a[10];
int i=0;
while (i<10) {
  process(a[i]);
  ++i;
}
```

(a) Using an indexed array

```
char* p;
p = (char*) 10000;
while (p < (char*) 10010) {
  process(*p);
  ++p;
}
```

(b) Using a running pointer

Figure 5.4: The decompilation of two machine code programs processing an array of ten characters.

prompt the type analysis to declare the object at address 10 010 to be of type char, when in fact this is just the next data item in the data section after array `a`. That object could have any type, or be unused. This demonstrates that whenever constant `K` is used with type `Y*`, it does not always follow that location `K*` (i.e. `m[K]`) is used with type `Y`.

Programs written in the style of 5.4(a) can be transformed by compilers to machine code programs in the style of 5.4(b).

The size of the array in Figure 5.4(a) is relatively easy to determine. In other cases, it may be difficult or impossible to determine the length of the array. For example, a special value may be used to terminate the loop. The loop termination condition could be arbitrarily complex, necessitating in effect that the program be executed to find the terminating condition. Even executing the program may not determine the full size of the array, since for each execution, only part of the array may be accessed. In these cases, type analysis may determine that there is an initialised array, but fail to compute the length of the array.

```
typedef struct {
int i;
float f;
} s;
s as[10];
s* ps = as;
while (ps < as+10) {
  processInt(ps->i);
  processFloat(ps->f);
  ++ps;
}
```

(a) Original program

```
void* p = (void*) 10000;
while (p < (void*) 10080) {
  processInt(*(int*)p); p += 4;
  processFloat(*(float*)p);
  p += 4;
}
```

(b) Decompilation

Figure 5.5: A program referencing two different types from the same pointer.

Another case is where an array of structures is traversed with running pointers, as shown in Figure 5.5(a). Here the structure contains one integer and one float; inside the loop, the pointer references alternately an integer and a float. The original compiler has rearranged the code to increment the pointer by the size of the integer and the size of the float after each reference, to make the machine code more compact and efficient. As shown in Figure 5.5(b), the type of the pointer is now `void*`, since it is used as two incompatible pointer types.

Programs such as shown in Figure 5.4(b) and 5.5(b) are less readable but correct (assuming suitable casts and allocating memory at the appropriate addresses) if the array is not initialised. However, the only way to make them correct if the array is initialised is to use binary translation techniques: force the data to the original addresses, and reverse the data sections if the endianness of source and target machines is different. Needless to say, the results are far from readable, and translation to machines with different pointer sizes (among other important characteristics) is simply not feasible.

To avoid this extreme unreadability, it is necessary to analyse the loops containing the pointer based code, and find the range of the pointer(s).

Pointer range analysis for decompilers is not considered here; analyses such as the value set analysis of Reps et al. may be suitable [RBL06].

5.3 Sources of Type Information

Type information arises from machine instruction opcodes, from the signatures of library functions, to a limited extent from the values of some constants, and occasionally from debugging information

One of the best sources of type information in a machine code program is the set of calls to library functions. Generally, the signature of a library function is known, since the purpose of a library function is to perform a known and documented function. A signature consists of the function name, the number and type of parameters, and the return type (if any) of the function. This information can be stored in a database of signatures derived from parsing header files, indexed by the function name (if dynamically linked) or by an identifier obtained by pattern matching statically linked code [FLI00, VE98].

A limited form of type information comes from the value of some constants. In many architectures, certain values can be ruled out as pointers (e.g. values less than about 0x100). One of the major decisions of type analysis is whether a constant is a pointer or not, so this information can be valuable.

A third source of type information is the semantics of individual machine instructions. All machine instructions will imply a size for each non-immediate operand. Registers have a definite size, and all memory operations have a size encoded into the opcode of the instruction. For instructions such as some load, store and move instructions, there is no information about the type other than the size, and that the type of the source and destination are related by $\mathcal{T}(dest) \geq \mathcal{T}(src)$. ($\mathcal{T}(x)$ denotes the type of x. The inequality only arises with class pointers; the destination could point to at least as many things as the source). The same pointer-sized move instruction could be used to move a pointer, integer, floating point value, etc. As a result, there needs to be a representation for a type whose only information is the size, e.g. `size16` for any 16-bit quantity.

In some machine code programs, there is runtime type information (sometimes called RTTI or runtime type identification). This is typically available where the original program used the C++ `dynamic_cast` operation or similar. Even if the original source code did not use such operations explicitly, the use of certain libraries such as the Microsoft Foundation Classes (MFC) may introduce RTTI implicitly [VEW04]. When RTTI is present in the input program, the names and possibly also the hierarchy of classes is available. Class hierarchy provides type information; for example if class A is a parent of class B, then $\mathcal{T}(A) > \mathcal{T}(B)$.

Finally, it is possible that the input program contains debug information or symbols. Debugging information is usually switched on during program development to aid with debugging. It is possible that in some circumstances, debugging information will still be present in the machine code available to a decompiler. When debug information is available in the input program, the names of all procedures are usually available, along with the names and types of parameters. The types of function returns and local variables may also be available, depending on the details of the debug information present.

Most type information travels from callee to caller, which is convenient for decompilation because most of the other information (e.g. information about parameters and returns) also travels from callee to caller. However, some type information travels in the opposite direction. There is symmetry between arguments/parameters and returns/results, with values sent from argument to parameter and return to result. The direction that type information from the sources discussed above travels depends on whether the library function, constant, machine instruction, *etc.* resides in the caller or callee.

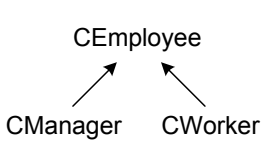
Consider the simple example of a function that returns the sum of its two 32-bit (pointer-sized) parameters. The only type information known about the parameters of this

function, taken in isolation, is both parameters and the return are of size 32 bits, and not of any floating point type. In other words, several combinations of pointer and integer are possible. If types are found for either of the callers' actual arguments or result, type information will flow from caller to callee.

This example also highlights a problem with type analysis in general: there may be several possible solutions, all valid. The programs for adding a pointer and an integer, an integer and a pointer, and two integers, could all compile to identical machine code programs. This problem is most acute with small procedures, however, since the chance of encountering a use that implies a specific type increases as larger procedures are considered.

Type analysis is considered a bidirectional problem [KDM03]. In other words, if implemented as a data flow problem, neither forward or reverse traversal of the flow graph will result in dramatically improved performance. The bidirectionality has its roots in the fact that most type defining operations affect a destination (a definition) and some operands (uses). The type effects of definitions flow forward to uses of that definition; the type effects of uses flow backwards to definition(s) of those uses. Library function calls similarly have bidirectional effects. Call by value actual arguments to the library calls are uses of locations defined earlier (type information flows backwards). Return value types and call by reference arguments are definitions whose type effects flow forwards. As a result, type analysis is one of few problems that can be expressed in data flow terms and is truly bidirectional.

Comparison operations result in type constraints that may not be intuitive. The equality relational operators ($=$ and $\neq$) imply only that the types of the variables being compared are comparable. (If two types are comparable, it means that they are compatible in the C sense; `int` and `char*` are not compatible so they can't be compared.) The type of either operand could be greater or equal to the type of the other operand, since the equality relations are commutative, as shown in Figure 5.6.




```
void AddEmployee(CEmployee* emp, CManager* mgr) {
  if (emp == mgr) ...
  ...
  if (mgr != emp) ...
```

Figure 5.6: Pointer equality comparisons.

By contrast, the non-commutative relational operators such as $\leq$ imply an array of uniformly typed objects, since high level languages do not allow assumptions about the relative addresses of other objects. Figure 5.7(b) shows an example where the constant `a+10` should be typed as `int*`, even though the same numeric constant may be

used elsewhere as the address of the `float f`. These operators therefore imply that the types of the operands are equal.

```
int a[10] = {1, 2, ...};
float f[10];
int i=0;
while (i < 10) {
  process(a[i]); ...
```

(a) Original C

```
int a[10] = {1, 2, ...};
float f[10];
int* p=a;
while (p < a+10) {
  process(*p); ...
```

(b) Optimised effective code

Figure 5.7: A pointer ordering comparison.

Some relational operators imply a signedness of the operands (e.g. signed less than or equal). Signedness mismatches of integral operands are often ignored by compilers, hence signed integers, unsigned integers, and integers of unknown sign should not be regarded as distinct types. Rather, signedness of an integer type is a property, and the final declared signedness of a variable can be decided by a heuristic such as the statically most commonly used variant.

Usually, a different comparison operator is used when comparing integers and floating point numbers, so the basic type is implied by these operators. Fixed point numbers will often be manipulated by integer operators (e.g. comparison operators, add and subtract), with only a few operations (e.g. fixed point multiply) identifying the operands as fixed point. It may therefore be possible to promote integers to fixed point numbers, i.e. fixed-point $\sqsubset$ integer.

5.4 Typing Constants

Constants have types just as locations do, and since constants with the same numeric value are not necessarily related, constants have to be typed independently.

In high level languages, constants have implied types. For example, the type of `2.0` is double; `'a'` is char, `0x61` is integer, and so on. However, in decompilation, this does not apply. Constants require typing just as locations do. Depending on the type of the constant, the same immediate value (machine language constant) might end up emitted in the decompiled code as `1084227584`, `5.000000`, `&foo`, `ENUM_40A00000`, or possibly other representations. It is important to realise that every constant has to be typed independently. For example, in the following statement, each of the constants with value 1000 could be a different type:

$$\texttt{m[}1000_1\texttt{]} := \texttt{m[}1000_2\texttt{]} + 1000_3$$

The three solutions are:

- $\mathcal{T}(1000_1) = \mathsf{int}^*$ and $\mathcal{T}(1000_2) = \mathsf{int}^*$ and $\mathcal{T}(1000_3) = \mathsf{int}$
- $\mathcal{T}(1000_1) = \alpha^{**}$ and $\mathcal{T}(1000_2) = \alpha^{**}$ and $\mathcal{T}(1000_3) = \mathsf{int}$
- $\mathcal{T}(1000_1) = \alpha^{**}$ and $\mathcal{T}(1000_2) = \mathsf{int}^*$ and $\mathcal{T}(1000_3) = \alpha^*$

where each α represents an arbitrary type, but each appearance of α implies the same arbitrary type as the other αs. In the third example, 1000_3 is a pointer to an α, which is added to an integer (found at memory address 1000), yielding a pointer to an α, which overwrites the integer that was temporarily at location 1000. Casts must have been used in the original program for the third solution to be valid, and will be needed in the decompiled output.

Note also that the value in a register can be an intermediate value that has no type, e.g. in the following SPARC code:

```
sethi 0x10 000, %o0
add %o0, 0x500, %o1   // %o1 used later as char*, value 0x10 500
add %o0, 0x600, %o2   // %o2 used later as integer, value 0x10 600
```

The value `0x10 000` in register `%o0` has no standard type, although it appears to be used as part of two different types, `char*` and `integer`. It must not appear in the decompiled output. The intermediate value is a result of a common feature of RISC machines: since RISC instructions are usually one word in length, two instructions are needed to produce most word-length constants. Constant propagation and dead code elimination, both facilitated by the SSA form, readily solve this problem by removing the intermediate constants.

5.5 Type Constraint Satisfaction

Finding types for variables and constants in the decompiled output can be treated as a constraint satisfaction problem; its specific characteristics suggest an algorithm that makes strong use of constraint propagation.

Assigning types to decompiled variables can be considered to be a Constraint Satisfaction Problem (CSP) [VH89, Bar98]. The domain of type variables is relatively small:

- elementary types (enumerated in Section 5.2.3)
- pointer to α (where α is any type, including another pointer)
- an array of α (where α is any type, including another array)
- a structure or class
- an enumerated type

Figure 5.8 shows a program fragment in source code, machine code (with SSA transformation), the resulting constraints, and the solutions to the constraints.

In the second instruction, the register `r1a` (first SSA version of register `r1`) is set to 0. Since zero could be used as an integer and but also as a NULL pointer, the constraints are that r1a could be an integer (*t1a = int*) or a pointer to something, call it α_1 (*t1a = ptr*(α_1)). The constraints generated by the add instruction are more complex, reflecting the fact that there are three possibilities: *pointer + integer = pointer*, *integer + pointer = pointer*, and *integer + integer = integer* respectively. The constraints are normally solved with a standard constraint solver algorithm, but parts of a simple problem such as that of Figure 5.8 can be solved by eye. For example, the constraint for the load instruction has only one possibility, *t0b = ptr(mem(0: t2a)* (meaning that `r0b` points to a structure in memory with type `t2a` at offset 0 from where `r0b` is pointing). This can be substituted into the constraints for the instruction with the label `3F2:`, so the types for `t0a` and `t0c` must be the same as this.

Continuing the constraint resolution, two solutions are found. In most cases, there would not be more than one solution.

Because constants are typed independently (previous section), and expressions (including constants) have to be expressed in terms of constraints, it is necessary to differentiate constants in some way. For example, constants could be subscripted much as SSA variables are renamed, e.g. 1000_3 as has already been seen. The type of each version of the constant 1000 needs to be treated as a separate variable.

Type constants (values for type variables which are constant) are common. For example, an `xor` instruction implies integer operands and result; a `sqrt` instruction implies floating point operands and result; an `itof` instruction implies integer operand and floating point result. A library function call results in type constants for each parameter (if any) and the result (if any).

Choosing a value for a type variable (in CSP terminology) is often implicit in the constraint. For example, the add instruction of Figure 5.8, `add r2a,r1b,r1c`, results in the constraints

```
int f(struct A *x)
{  int r=0;
   for (; x !=0; x = x->t1)
      r += x->hd;
   return x;
}
```

(a) Original C program

Destination is last operand
`r0a`, `r0b` are SSA versions of register `r0`
t1a = type of `r1a`
mem(n:t) means a pointer offset n bytes from pointer t

(b) Legend

```
f:
      mov  r0,r0a
      mov  #0,r1a
      cmp  #0,r0a
      beq  L4F2
L3F2: mov  φ(r0a,r0c),r0b
      mov  φ(r1a,r1c),r1b
      ld.w 0[r0b],r2a
      add  r2a,r1b,r1c

      ld.w 4[r0b],r0c
      cmp  #0,r0c
      bne  L3F2
L4F2: mov φ(r1a,r1c),r1d
      mov  r1d,r0d
      ret
```

(c) SSA machine code

$tf = t0 \rightarrow t99$
$t0 = t0a$
$t1a = int \vee t1a = ptr(\alpha_1)$
$t0a = int \vee t0a = ptr(\alpha_2)$
$t0b = t0a,\ t0b = t0c$
$t1b = t1a,\ t1b = t1c$
$t0b = ptr(mem(0 : t2a)$
$t2a = ptr(\alpha_3),\ t1b = int,\ t1c = ptr(\alpha_3) \vee$
$t2a = int,\ t1a = ptr(\alpha_4),\ t1c = ptr(\alpha_4) \vee$
$t2a = int,\ t1b = int,\ t1c = int$
$t0b = ptr(mem(4 : t0c))$
$t0c = int \vee t0c = ptr(\alpha_5)$
$t1d = t1a,\ t1d = t1c$
$t0d = t1d$
$t99 = t0d$

(d) Constraints

Solving the above constraints leads to the following "failure":
$t0c = t0b = ptr(mem(4: t0c)) = ptr(mem(0 : t2a)$
This is "repaired" by using `struct G { `$t2a$ `m0; `$t0c$ `m4; ...}` i.e.
$t0c = ptr(mem(0 : t2a,\ 4 : t0c)) = ptr($`struct G`$)$
Continuing the process yields the following two solutions:
$t0 = t0a = t0b = t0c = ptr($`struct G`$)$
$t99 = t1a = t1b = t1c = t1d = t2a = t0d =$ `int`
$tf = ptr($`struct G`$) \rightarrow$`int`
and the "parasitic" solution:
$t0 = t0a = t0b = t0c = ptr($`struct G`$)$
$t2a =$ `int`
$t99 = t1a = t1b = t1c = t1d = t2a = t0d = ptr(\alpha_4)$
$tf = ptr($`struct G`$) \rightarrow ptr(\alpha_4)$

Parasitic solutions are unlikely with larger programs, and Mycroft suggests ways to avoid them in his paper.

Figure 5.8: A simple program fragment typed using constraints. From [Myc99].

$$t2a = ptr(\alpha_3),\ t1b = int,\ t1c = ptr(\alpha_3) \vee$$
$$t2a = int,\ t1a = ptr(\alpha_4),\ t1c = ptr(\alpha_4) \vee$$
$$t2a = int,\ t1b = int,\ t1c = int$$

Here, the constraints are expressed as a disjunction (or-ing) of conjuncts (some terms and-ed together; the commas here imply and). Finding values for $t2a$, $t1b$, and $t1c$ happen simultaneously, by choosing one of the three conjuncts. The choice is usually made by rejecting conjuncts that conflict with other constraints. Hopefully, at the end of the process, there is one set of constraints that represents the solution to the type analysis problem.

Some constraint solvers are based on constraint propagation (e.g. the "forward checking" technique). Others rely more on checking for conflicts (e.g. "simple backtracking" or "generate and test"). Taken together, the above factors (small domain size, constants and equates being common, and so on) indicate that constraint propagation will quickly prune branches of the search tree that will lead to failure. Therefore, it would appear that constraint propagation techniques would suit the problem of type constraint satisfaction as it applies to decompilation better than the other group of techniques.

A weakness of constraint-based type analysis algorithms is that some such algorithms are incomplete, i.e. a given algorithm may find one or more solutions, or prove that the constraints cannot be solved (prove they are inconsistent), or they may not be able to do either.

5.5.1 Arrays and Structures

Constraint-based type analysis requires extra rules to handle arrays correctly.

Mycroft proposed a constraint based type analysis system for decompilation of machine code programs (in Register Transfer Language form or RTL) [Myc99]. He generates constraints for individual instructions, and solves the constraints to type the variables of the program being decompiled. He assumes the availability and use of double register addressing mode instructions to signal the use of arrays. For example, from Section 4 of his paper,

instruction	generated constraint
`ld.w (r5)[r0],r3`	*t0=ptr(array(t3)), t5 = int* $\vee$ *t0 = int, t5=ptr(array(t3))*

However, some machine architectures do not support two-register indexing, and even if they did, a compiler may for various reasons decide to perform the addition separately

instruction	generated constraint
`add r5,r0,r1`	$t5 = ptr(\alpha) \wedge t0 = int \wedge t1 = ptr(\alpha) \quad \vee$ $t5 = int \wedge t0 = ptr(\alpha) \wedge t1 = ptr(\alpha) \quad \vee$ $\underline{t5 = int \wedge t0 = int \wedge t1 = int}$
`ld.w 0[r1],r3`	$t1 = ptr(mem(0{:}\ t3) \quad \wedge$ $(t5 = int \wedge t0 = ptr(t3) \quad \vee \quad t5 = ptr(t3) \wedge t0 = int)$

Figure 5.9: Constraints for the two instruction version of the above. Example from [Myc99].

to the load or store instruction. Hence, the above instruction may be emitted as two instructions, with constraints as shown in Figure 5.9.

The last (underlined) conjunct for the first instruction is immediately removed since `r1` is used as a pointer in the second instruction. The final constraints are now in terms of $ptr(t3)$, rather than $ptr(array(t3))$. These are equivalent in the C sense, but the fact that an array is involved is not apparent. In other words, considering individual instructions by themselves is not enough (in at least some cases) to analyse aggregate types. Either some auxiliary rule has to be added outside of the constraint system, or expression propagation can be used in conjunction with a high level type pattern. (In Section 4.3 of his paper, Mycroft seems to suggest considering pairs of instructions to work around this problem.) This is another area where expression propagation and simplification as described in Chapters 3 and 4 could be used.

5.6 Addition and Subtraction

Compilers implicitly use pointer-sized addition instructions for structure member access, leading to an exception to the general rule that adding an integer to a pointer of type α yields another pointer of type α*.*

Pointer-sized addition and subtraction instructions are a special case for type analysis, since they can be used on pointers or integers. Mycroft [Myc99] states the type constraints for the instruction `add a,b,c` (where `c` is the destination, i.e. `c := a + b`) to be:

$$\begin{aligned}
&\mathcal{T}(a) = ptr(\alpha), \mathcal{T}(b) = int, \quad \mathcal{T}(c) = ptr(\alpha) \vee \\
&\mathcal{T}(a) = int, \quad \mathcal{T}(b) = ptr(\alpha'), \mathcal{T}(c) = ptr(\alpha') \vee \\
&\mathcal{T}(a) = int, \quad \mathcal{T}(b) = int, \quad \mathcal{T}(c) = int
\end{aligned}$$

where again $\mathcal{T}(x)$ represents "the type of x" (Mycroft uses tx) and $ptr(\alpha)$ represents a pointer to any type, with the type variable α representing that type. Mycroft uses

the C pointer arithmetic rule, where adding an integer to a variable of type α* will always result in a pointer of type α*. However, the C definition does not always apply at the machine code level. Compilers emit add instructions to implement the addition operator in source programs, but also for two other purposes: array indexing and structure member access. For array indexing, the C pointer rules apply. The base of an array of elements of type α is of type α*, the (possibly scaled) index is of type integer, and the result is a pointer to the indexed element, which is of type α*.

However, structure member access does not follow the C rule. Consider a structure of type Σ with an element of type ϵ at offset `K`. The address of the structure could be type Σ* (a pointer to the structure) or type ϵ_0* (a pointer to the first element ϵ_0 of the structure), depending on how it is used. At the machine code level, Σ and $\Sigma.\epsilon_0$ have the same address and are not distinguishable except by the way that they are used. `K` is added to this pointer to yield a pointer to the element ϵ, which is of type ϵ*, in general a different type to Σ* or ϵ_0*.

Unfortunately, until type analysis is complete, it is not known whether any particular pointer will turn out to be a structure pointer or not. Figure 5.10 gives an example.

```
void foo(void* p) {
  m[p] := ftoi(3.00);  // Use p as int*
  ...
  m[p+4] := itof(-5);  // Use p as pointer to struct with float at offset 4
```

Figure 5.10: A program fragment illustrating how a pointer can initially appear not be be a structure pointer, but is later used as a structure pointer.

Initially, parameter p is known only to be a pointer. After processing the first statement of the procedure, p is used with type int*, and so $\mathcal{T}$(p) is set to int*. In the second statement (in general, any arbitrary time later), it is found that p points to a struct with an int at offset 0 and a float at offset 4. If the C rule was used that the sum of p (then of type int*) and 4 results in a pointer of the same type as p, then in the second statement p is effectively used as both type int* and type float*. This could lead to p being declared as a union of int* and float*.

Note that the exception to the otherwise general C rule involves only the addition of a structure pointer and a *constant* integer; compilers know the offset of structure members and keep this information in the symbol table for the structure. Hence if the integer is not a constant, the types of the result and the input pointer can be constrained to be equal, i.e. Mycroft's constraints apply.

To avoid this problem, Mycroft's constraints could be rewritten as follows:

$$\mathcal{T}(a) = ptr(\alpha),\ \ \mathcal{T}(b) = \textit{var-int},\ \ \mathcal{T}(c) = ptr(\alpha) \vee$$

$$\mathcal{T}(a) = \textsf{void*}, \quad \mathcal{T}(b) = \textit{const-int}, \mathcal{T}(c) = \textsf{void*} \vee$$
$$\mathcal{T}(a) = \textit{var-int}, \quad \mathcal{T}(b) = \textit{ptr}(\alpha\text{'}), \quad \mathcal{T}(c) = \textit{ptr}(\alpha\text{'}) \vee$$
$$\mathcal{T}(a) = \textit{const-int}, \mathcal{T}(b) = \textsf{void*}, \quad \mathcal{T}(c) = \textsf{void*} \vee$$
$$\mathcal{T}(a) = \textit{int}, \quad \mathcal{T}(b) = \textit{int}, \quad \mathcal{T}(c) = \textit{int}$$

Here, void*, represents a variable known to be a pointer, but the type pointed to is unknown (output) or is ignored (input). Where there is a conflict between void* and an α*, α* would be used by preference. Note that this is already suggesting a hierarchy of types, with void* being preferred to no type at all, but any α* preferred to void*.

Subtraction from or by pointers is not performed implicitly by compilers, so Mycroft-like constraints for the pointer-sized subtract operation can be derived. For $c := a$ - b:

$$\mathcal{T}(a) = \textit{ptr}(\alpha), \mathcal{T}(b) = \textit{int}, \quad \mathcal{T}(c) = \textit{ptr}(\alpha) \vee$$
$$\mathcal{T}(a) = \textit{ptr}(\alpha), \mathcal{T}(b) = \textit{ptr}(\alpha), \mathcal{T}(c) = \textit{int} \vee$$
$$\mathcal{T}(a) = \textit{int}, \quad \mathcal{T}(b) = \textit{int}, \quad \mathcal{T}(c) = \textit{int}$$

5.7 Data Flow Based Type Analysis

Type analysis for decompilers where the output language is statically type checked can be performed with a sparse data flow algorithm, enabled by the SSA form.

Types can be thought of as sets of possible values for variables. The smaller the set of possible values, the more precise and useful the type information. These sets form a natural hierarchy: the set of all possible values, the set of all possible 32-bit values, the set of signed or unsigned 32-bit integers, the set of signed 32-bit integers, a subrange of the signed integers, and so on. The effect of machine instructions is often to restrict the range of possible values, e.g. from all 32-bit values to 32-bit integers, or from 32-bit integers to unsigned 32-bit integers.

This effect suggests that a natural way to reconcile the various restrictions and constraints on the types of locations is to use an iterative data flow framework [MR90, KU76]. The data that are flowing are the type restrictions and constraints, and the results are the most precise types for the locations in the program given these restrictions and constraints.

Data flow based type analysis has some advantages over the constraint-based type analysis outlined in Section 5.5. Since there are no explicit constraints to be solved outside the normal intermediate representation of the program, there is no need to distinguish constants that happen to have coinciding values. Constraints are difficult to

solve, sometimes there is more than one solution, and at other times there is no solution at all. By contrast, the solution of data flow equations is generally quite simple. Two exceptions to this simplicity are the integer addition and subtraction instructions; as will be shown, these are more complex than other instructions, but the complexity is moderate.

5.7.1 Type Lattices

Since types are hierarchical and some type pairs are disjoint, the relationship between types forms a lattice.

Types can be thought of as sets of possible values, e.g. the type integer could be thought of as the infinite set of all possible integers. In this chapter, the two views will be used interchangeably. The set of integers is a superset of the set of unsigned integers (counting numbers). There are more elements in the set of integers than the set of unsigned integers, or integer $\supset$ unsigned integer. The set of integers is therefore in a meaningful way "greater than" the set of unsigned integers; hence there is an ordering of types.

The ordering is not complete, however, because some type pairs are disjoint, i.e. they cannot be compared sensibly. For example, the floating point numbers, integers, and pointers are not compatible with each other at the machine code level, and so are considered to be mutually incompatible (i.e. not comparable). None of these elementary types implies more information than the others. To indicate that the ordering is partial, the square versions of the usual set operator symbols are used: $\subset$, $\supset$, $\cap$, and $\cup$ become $\sqsubset$, $\sqsupset$, $\sqcap$, and $\sqcup$ respectively. Hence int $\sqsupset$ unsigned int or "int is a supertype of unsigned int".

These types are incompatible at the machine code level despite the fact that the mathematical number 1027 is an integer, it is also a real number, and it could be used as the address of some object in memory. Floating point numbers have a different bit representation to integers, and are therefore used differently. Pointer variables should only be used by load and store instructions (including complex instructions incorporating loads and/or stores), or compare/test instructions. Integer variables should similarly be used by integer instructions (integer add, shift, bitwise or), and floating point variables should only be used by floating point instructions (floating point add, square root, etc).

Two exceptions to this neat segregation of types with classes of instructions are the integer add and subtract instructions. In most machines, these instructions can be used on integer variables and also pointer variables. Section 5.6 discussed the implications of this exception in more detail. It is the fact that objects of different types usually

must be used with different classes of instructions that makes it so important that type errors are avoided.

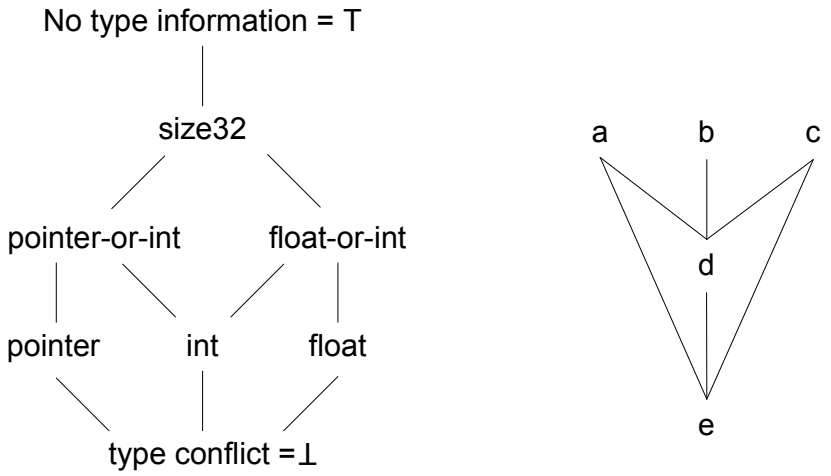
In decompilation, types additional to those used in computer languages are considered, such as size32 (a 32-bit quantity, whose basic type is not yet known, yet the size is known), or the type pointer-or-integer. These are temporary types that should be eliminated before the end of the decompilation. As a contrived example, consider a location referenced by three instructions: a 32-bit `test` of the sign bit, an integer `add` instruction, and an arithmetic shift right. Before the `test` instruction, nothing is known about the location at all. The type system can assign it a special value, called $\top$ (top). $\top$ represents all possible types, or the universal set U, or equivalently, no type information (since everything is a member of U). After the `test` instruction, all that is known is that it is a 32-bit signed quantity, so the type analysis can assign it the temporary type signed32. Following the add instruction, it is known to be either a pointer or an integer: pointer-or-int. An arithmetic shift right instruction applies only to integer variables, so the location could be assigned the type int.

The type hierarchy so far can be considered an ordered list, with $\top$ at one end, and int at the other end. It is desirable for the type of a location to move in one direction (with one exception, described below), towards the most constrained types (those with fewer possible values, away from $\top$). It is also possible that later information will conflict with the currently known type. For example, suppose that a fourth instruction used the location as a pointer. Logically, this could be represented by another special value $\bot$ (bottom), representing overconstraint or type conflict, or the empty set $\emptyset$ of possible values. This could occur as a result of a genuine inconsistency in the original program (e.g. due to a union of types, or a typecast), or from some limitation of the type analysis system.

In practice, for decompilation, it is preferable never to assign the value $\bot$ to the types of variables; this is equivalent to giving up on typing the location altogether. Rather, decompilers would either retain the current type (int) or assign the new type (the pointer), and conflicting uses of the location would be marked in such a way that a cast or union would be emitted into the decompiled output. (If a cast or union is not allowed in the output language, a warning comment may be the best that can be done. Some languages are more suitable as the output language of a decompiler than others.)

Continuing the example of the location referenced by three instructions, the list of types can be represented vertically, with $\top$ and types size32, pointer-or-int, and int in that order extending downwards, as shown in Figure 5.11(a).

Since another instruction (e.g. a square root instruction) in place of the add instruction would have determined the location to be of type float instead, there is a path from



(a) (b)

Figure 5.11: Illustrating the greatest lower bound.

size32 to float, and no path from int to float, since these are incompatible.

So far, when a location is used as two types a and b, the lower of the two types in the lattice becomes the new type for the location. However, consider if a location had been used with types pointer-or-int and float-or-int. (The latter could come about through being assigned a value known not to be valid for a pointer). The resultant type cannot be either of the types it has been used with so far (pointer-or-int or float-or-int), but should in fact be the new type int, which is the greatest type less than both pointer-or-int and float-or-int. In other words, the general result of using a location with types a and b is the *greatest lower bound* of a and b, also known as the *meet* of a and b, written $a \sqcap b$.

Note the similarity with the set intersection symbol $\cap$. The result of meeting types a and b is basically $a \cap b$ where a, b, and the result are thought of as sets of possible values. For example, if the current type is ?signed-int (integer of unknown sign), it could be thought of as the set {sint, uint}. If this type is met with unsigned-int ({uint}), the result is {sint, uint} $\cap$ {uint} = {uint} or unsigned-int.

Figure 5.11(b) shows why it is the *greatest* lower bound that is required. When meeting types a and c, the result should be d or e, which are lower bounds of a and c. In fact, it should be d, the greatest lower bound, since there is no justification (considering only the meet operation of a and c) for selecting the more specialised type e. For example, if the result is later met with b, the result has to be d, since types b and e are not comparable in the lattice (meaning that they are not compatible types in high level languages).

Figure 5.12 shows a more practical type lattice, showing the relationship of the numeric

types.

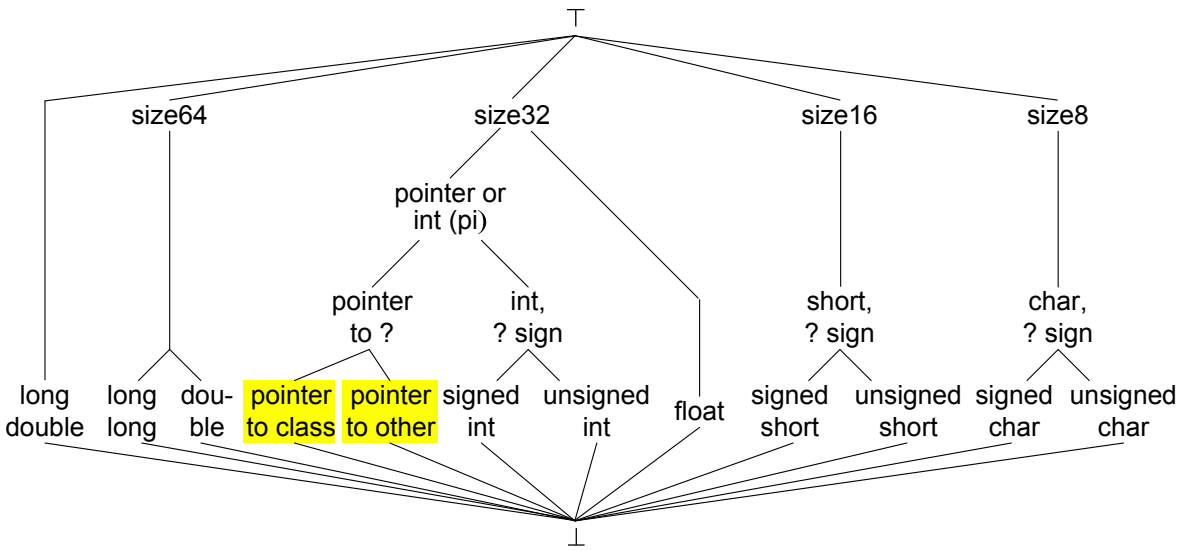


Figure 5.12: A simplified lattice of types for decompilation.

Earlier it was mentioned that when considering the types of locations in a program, the types do not always move from top to bottom in the lattice. The exception concerns pointers to class objects in an object oriented language such as C++.

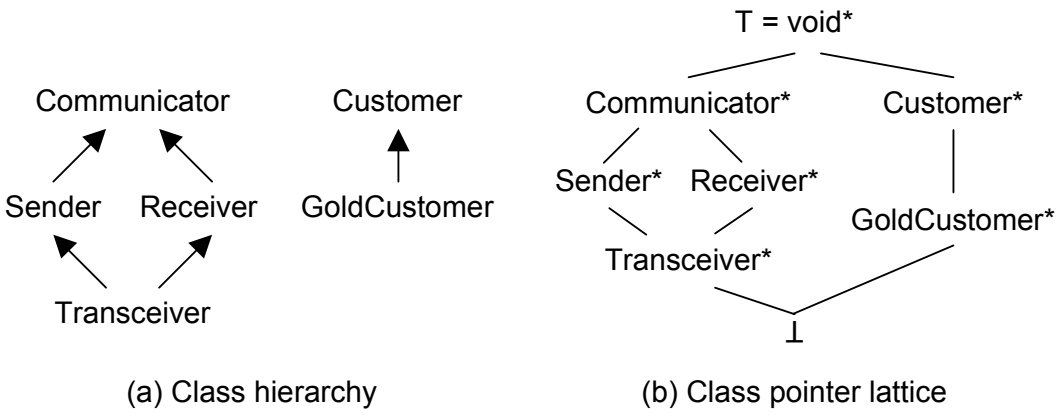


Figure 5.13: Class pointers and references.

Consider the example class hierarchy and lattice shown in Figure 5.13. The similarity between the class hierarchy and the hierarchy of pointers to those classes is evident. A pointer could be assigned the type Sender* in one part of the program, and the type Receiver* in another part. If there is a control flow merge from these two parts, the pointer will have been used as both Sender* and Receiver*. The rules so far would result in the type Transceiver*, which is a pointer to a type that has multiple inheritance from classes Sender and Receiver. However, this may be overkill, if the program is only referring to those methods and/or members which are common to Senders and

Receivers, i.e. methods and members of the common ancestor Communicator. Also, multiple inheritance is relatively uncommon, so in many programs there is no class that inherits from more than one class, and to generate one in the decompiled output would be incorrect.

This example illustrates that sometimes the result of relating two types is higher up the lattice than the types involved. In these cases, relating types α* and β* results in the type $(\alpha \sqcup \beta)$*, where $\sqcup$ is the *join* operator, and $\alpha \sqcup \beta$ results in the *least upper bound* of α and β. Note the similarity to the set union operator symbol $\cup$.

This behaviour occurs only with pointers and references to class or structure objects. For pointers to other types of objects, the types of the pointer or reference and the variable so referenced have to correspond exactly in type.

There is an additional difference brought about by class and structure pointers and references. In a copy statement such as $a := b$, if a and b are not such pointers or references, then the type of a and b are both affected by each other; the type of both is the type of a met with the type of b. However, with $p := q$ where p and q are both class or structure pointers or references, p may point to a larger set of objects than q, and this broadening of the type of p does not affect q. Hence after such an assignment, the type of p becomes a pointer to the join of p's base type and q's base type, but the type of q is not affected. Only assignments of this form have this exception, and only when the variables involved are pointers or references to classes or structures.

For the purposes of type analysis, procedure parameters are effectively assigned to by all corresponding actual argument expressions at every call to the procedure. The parameter could take on values from any of the actual argument expressions. For parameters whose types are not class or structure pointers or references, the types of all the arguments and that of the parameter have to be the same. However, if the type of the parameter is such a pointer or reference, the type of the parameter has to be the join of the types of all of the corresponding arguments.

For more detail on the subject of lattices, see [DP02].

5.7.2 SSA-Based Type Analysis

The SSA form links uses to definitions, allowing a sparse representation of type information in assignment nodes.

In the beginning of Section 5.7, it was stated that the decompiler type recovery problem can be solved as a data flow problem, just as compilers can implement type checking for statically checked languages that way. Traditional data flow analysis, often implemented

using bit vectors, can be used [ASU86]. However, this involves storing information about all live variables for each basic block of the program (or even each statement, depending on the implementation). Assuming that the decompiler will generate code for a statically typed language, each SSA variable will retain the same type, so that a more sparse representation is possible. Each use of an SSA location is linked to its statically unique definition, so the logical place to store the type information for variables is in the assignment statement associated with that definition. For parameters and other locations which have no explicit assignment, an implicit assignment can be inserted into the intermediate representation.

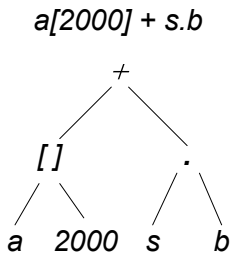
This framework is sparse in the sense that type information is located largely only where needed (one type variable per definition and constant, instead of one per variable and constant per basic block).

The SSA form of data flow based type analysis is not flow sensitive, in the sense that the computed type is summarised for the whole program. However, since the type for the location is assumed to be the same throughout the program, this is not a limitation. Put another way, using the SSA form allows the same precision of result with a less expensive flow insensitive implementation.

5.7.3 Typing Expressions

In a sparse intermediate representation, the types of subexpressions are not stored, so these have to be calculated on demand from the expression leaves.

Early in the decompilation process, assignments are often in a simple form such as $c := a \odot b$, where $\odot$ is a binary operator, c is a location, and a and b are locations or constants. However, some instructions are more complex than this, and after propagation, expressions can become arbitrarily complex. Figure 5.14 shows a simple example.



The leaves of the expression tree are always locations or constants (except for the destination of an assignment, which is always a location). Locations and constants have type variables associated with them in a sparse type analysis system for a decompiler. However, there is no place to store the type for a subexpression, such as *a[2000]* or *s.b,* unless all subexpressions store a type and the sparseness is lost.

Figure 5.14: Typing a simple expression.

In the example, the only type information known from other statements is that a is an array of character pointers (i.e. a currently has type char*[]). Type analysis for the

expression starts at the bottom of the expression tree in a process which will be called *ascend-type*. In this example the algorithm starts with subexpression *a[2000]*. The array-of operator is one of only a few operators where the type of one operand (here *a*, not *2000*), if known, affects the result, and the type of the result, if known, affects that operand. Since the type of *a* is known, the type of *a[2000]* can be calculated; in this case it is char*. This subexpression type is not stored; it is calculated on demand. The integer addition operator is a special case, where if one operand is known to be a pointer, the result is a pointer type, because adding a pointer and an integer results in a pointer. Hence the type of the overall expression is calculated to be void*. (Adding an integer to a char* does not always result in another char*, hence the result has type void*). Again, this type is not stored. No type information is gained from *s*, *b*, or *s.b*, since the types of *s* and *b* are currently unknown.

Next, a second phase begins, which will be called *descend-type*. Now type information flows *down* the expression tree, from the root to the leaves. In order to find the type that is pushed down the expression tree to the right of the addition operator, ascend-type is called on its left operand. This will result in the type char*, as before. This type, in conjunction with the type for the result of the addition, is used to find the type for the right subexpression of the add. Since pointer plus integer equals pointer, the type found is integer. The structure membership operator, like the array-of operator, can transmit type information up or down the expression tree. In this case, it causes the type for *b* to be set to integer, and the type for *s* to be set to a structure with an integer member.

When the process is repeated for the left subexpression of the add node, the result is type void*, which implies a type void*[] for *a*. However when this is met with the more precise existing type char*[], the type of *a* remains as char*[]. The type for the constant *2000* is set to integer (array indexes are always integer).

In the example above, the initial type for location *a* came from type information elsewhere in the program. In contrast to locations, constants have no connection to other parts of the program, as shown in Section 5.4. As a result, constants are typed only by the second phase, descend-type.

In general, type information has to be propagated up the expression tree, then down again, in two separate passes.

Table 5.1 shows the type relationships between operands(s) and results for various operators and constants. Most operators are in the first group, where operand and result types are fixed. For the other, less common operators and constants, the full cycle of ascend-type and descend-type is required.

In the example of Figure 5.14, the type for a[2000] was calculated twice; once during

Table 5.1: Type relationships for expression operators and constants.

Operators	Type relationships
*, /, itof, < etc	Fixed types for operands and results
memory-of, address-of, array-of, structure membership	Type information flows up or down the tree
?: (ternary operator)	Type information flows up the tree (from the related operands) and down (from the boolean operand)
integer + and -	Types for any two operands implies the type for the other; if one operand is a pointer, it implies types for the others
constants	Types flow down to constants, but constraints such as "cannot be float" or "cannot be pointer" could flow up the tree.

ascend-type, and again for descend-type. For more complex expressions, descend-type may call ascend-type on a significant fraction of the expression tree many times. Figure 5.15 (a) shows a worst-case example for an expression with four levels of binary operators. At the leaves, checking the type of the location (if present) could be considered one operation. This expression tree has 16 leaf nodes, for a total of 16 operations. One level up the tree, the type of the parent nodes is checked, using information from the two child nodes, for a total of three operations. There are eight such parent nodes, for a total of 24 operations at this level. Similarly, at the top level, 31 operations (almost 2^{h+1} where h is the height of the expression tree) are performed by descend-type.

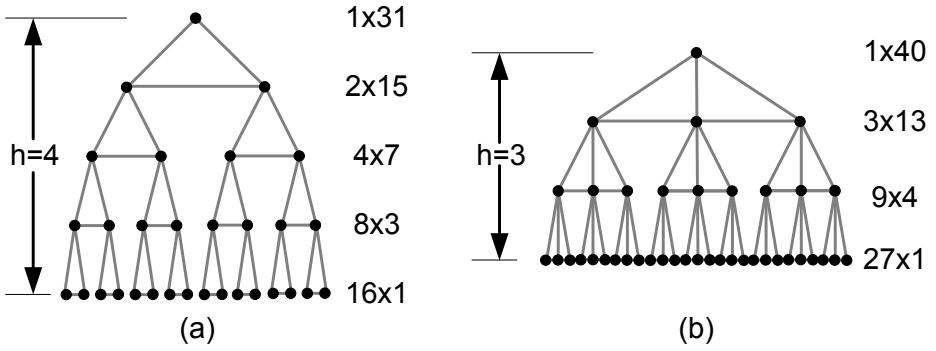


Figure 5.15: Complexity of the ascend and descend type algorithms.

Figure 5.15(b) shows a tree with all ternary operators (e.g. the C `?:` operator). Such a tree would never be seen in a real-world example, but it illustrates the worst-case complexity of the descend-type algorithm. Here, of the order of 3^{h+1} operations are performed. This potential cost offsets the space savings of storing type information sparsely (only at definitions, not uses).

5.7.4 Addition and Subtraction

The types inferred from pointer-sized addition and subtraction instructions require special type functions in a data flow based type analysis.

Section 5.6 showed a modification of Mycroft's constraints that take into account the exception caused by adding constants to structure pointers. In order to represent Mycroft's constraints in a data flow based analysis, some extra types and operators are required. Let π pointer-or-integer or higher in the lattice. It is hoped that all occurrences of π will eventually be replaced with a lower (more precise) type such as integer or a specific pointer type. The integer is understood to be the same size as a pointer in the original program's architecture. In the lattice of Figure 5.12, π could have the values pointer-or-integer, size32, or $\top$. The data flow equations associated with `add a, b, c` , (i.e. `c=a+b`), with the structure pointer exception, can be restated as:

$$\mathcal{T}(a) = \Sigma_a(\mathcal{T}(c), \mathcal{T}(b)) \quad (5.1)$$

$$\mathcal{T}(b) = \Sigma_a(\mathcal{T}(c), \mathcal{T}(a)) \quad (5.2)$$

$$\mathcal{T}(c) = \Sigma_s(\mathcal{T}(a), \mathcal{T}(b)) \quad (5.3)$$

where Σ_a and Σ_s (a stands for addend or augend, s for sum) are special functions defined as follows:

Σ_a	$\beta*$	$\mathcal{T}(c) =$ *var-int*	var-π	*const-*π
$\alpha*$	*var-int*	$\perp$	*var-int*	*var-int*
$\mathcal{T}$*(other)* =*var-int*	$\beta*$	*int*	*var-*π	*var-*π
const-int	*void**	*int*	*var-*π	-
var-π	π	π	*var-*π	*var*π
const-π	π	π	*var-*π	-

Σ_s	$\alpha*$	$\mathcal{T}(a) =$ *var-int*	*const-int*	*var-*π	*const-*π
$\beta*$	$\perp$	$\beta*$	*void**	$\beta*$	*void**
$\mathcal{T}(b) =$ *var-int*	$\alpha*$	*int*	*int*	*var-*π	*var-*π
const-int	*void**	*int*	*int*	*var-*π	-
var-π	$\alpha*$	*var-*π	*var-*π	*var-*π	*var-*π
const-π	*void**	*var*π	-	*var-*π	-

For brevity, ptr(α) is written as $\alpha*$. The type variables $\mathcal{T}(a)$, $\mathcal{T}(b)$, and $\mathcal{T}(c)$ are initialised to *var-*π or *const-*π if the operands are locations or constants respectively.

As an example, consider `p = q+r` with `q` known to have type `char*`, and the type of `r` is wanted. Since the type of `p` is not known yet, the type of `p` remains at its initial value of *var-π*. `p`, `q`, and `r` are substituted for `c`, `a`, and `b` respectively of equation 5.2. This equation uses function Σ_a, defined above in the left table. Since $\mathcal{T}(c)$ = var-π, the third column is used, and since $\mathcal{T}(other) = \alpha*$ with $\alpha =$ `char`, the first row is used. The intersection of the third column and first row contains *var-int*, so $\mathcal{T}(b) = \mathcal{T}(r) =$ `int`.

Similarly, the data flow equations associated with `sub a, b, c` (i.e. `c=a-b`) can be restated as:

$$\mathcal{T}(a) = \Delta_m(T(c), T(b)) \tag{5.4}$$
$$\mathcal{T}(b) = \Delta_s(\mathcal{T}(c), \mathcal{T}(a)) \tag{5.5}$$
$$\mathcal{T}(c) = \Delta_d(\mathcal{T}(a), \mathcal{T}(b)) \tag{5.6}$$

where Δ_m, Δ_s and Δ_d (m stands for *m*inuend (item being subtracted from), s for *s*ubtrahend (item being subtracted), and d for *d*ifference (result)) are special functions defined as follows:

Δ_m	$\beta*$	$\mathcal{T}(c) =$ *int*	π
$\alpha*$	$\perp$	$\alpha*$	$\alpha*$
$\mathcal{T}(b) = int$	$\beta*$	*int*	π
π	$\beta*$	*int*	π

Δ_s	$\beta*$	$\mathcal{T}(c) =$ *int*	π
$\alpha*$	*int*	$\alpha*$	π
$\mathcal{T}(a) = int$	$\perp$	*int*	*int*
π	*int*	π	π

Δ_d	$\alpha*$	$\mathcal{T}(a) =$ *int*	π
$\beta*$	*int*	$\perp$	*int*
$\mathcal{T}(b) = int$	$\alpha*$	*int*	π
π	π	*int*	π

As noted earlier, compilers do not use subtraction for referencing structure members, so this time there is no need to distinguish between *var-int* and *const-int*, or *var-π and const-π*.

5.8 Type Patterns

A small set of high level patterns can be used to represent global variables, local variables, and aggregate element access.

A sequence of machine instructions for accessing a global, local, or heap variable, array element, or structure element will in general result in a memory expression which can be expressed in the following normal form:

$$\texttt{m}[\left[\ \texttt{sp}_0\ \middle|\ \texttt{pl}\ \right] + \sum_{j=1}^{n} \texttt{S}_\texttt{j} * \texttt{ie}_\texttt{j} + \texttt{K}] \tag{5.7}$$

where

- `m[`x`]` represents memory at address x.
- All but one of the terms of the sum inside `m[...]` could be absent.
- $\left[a \middle| b\right]$ indicates optionally a or b but not both.
- $\texttt{sp}_0$ represents the value of the stack pointer register at the start of the procedure.
- `pl` is a nonconstant location used as a pointer (although nonconstant, it could take one of several constant values at runtime).
- The $\texttt{S}_j$ are scaling constants, some of which could be unity.
- Here $*$ represents multiplication.
- The $\texttt{ie}_j$ are nonconstant integer expressions that do not include the stack pointer register, and are known not to be of the form `x+C` where `x` is a location and `C` is a constant. Constants could appear elsewhere in an $\texttt{ie}_j$, e.g. it could be `4*r1*r2`.
- `K` is a constant (an integer or pointer constant).

Where necessary, the distributive law is applied, e.g. `m[4*(r1+1000)]` is transformed to `m[r1*4+4000]` before attempting to match to the above equation. Expression propagation, simplification, and canonicalisation, as discussed in Chapter 3, can be used to prepare the intermediate representation for high level pattern matching.

The sum of the terms inside the `m[...]` must be a pointer expression by definition. Pointers cannot be added, adding two or more integers does not result in a pointer, and the result of adding a pointer and an integer is a pointer. It follows that exactly one of the terms is a pointer, and the rest must be integers. Since $\texttt{sp}_0$ is always a pointer, $\texttt{sp}_0$ and `pl` cannot appear together, and if $\texttt{sp}_0$ or `pl` are present, `K` cannot be a pointer.

It could be argued that since `pl` or K could be negative, all three of $\texttt{sp}_0$, `pl`, and `K` could be present, with two pointers being subtracted from each other, resulting in a

constant. However, such combinations would require the negation of a pointer, which is never seen in real programs.

Initially, it may not be possible to distinguish `pl` from an $\texttt{ie}_j$ with $\texttt{S}_j$=1, so temporary expressions such as $\texttt{m[l}_1\texttt{+K]}$ or $\texttt{m[l}_1\texttt{+l}_2\texttt{+K]}$ may be needed, until it becomes clear which of $\texttt{l}_1$, $\texttt{l}_2$ and `K` is the pointer.

When present, $\texttt{sp}_0$ indicates that the memory expression represents a stack allocated variable, array element, or structure element.

Table 5.2: Type patterns and propositions discussing them.

Pattern	Proposition
$\texttt{ie}_j$ present	5.8
`K` present	5.9
`m[K]`	5.10
$\texttt{m[sp}_0\texttt{+K]}$	5.12
`m[l+K]`	5.13
`m[ie+K]`	5.14
`m[pl+K]`	5.15
$\texttt{m[sp}_0\texttt{+ie+K]}$	5.17

Pattern	Proposition
$\texttt{m[l}_1\texttt{+l}_2\texttt{+K]}$	5.18
`m[ie+pl+K]`	5.19
$\texttt{m[ie}_1\texttt{+ie}_2\texttt{+K]}$	5.20
`m[ie+pl]`	5.21
$\texttt{m[S}_1\texttt{*ie}_1\texttt{+...+S}_n\texttt{*ie}_n\texttt{+K]}$	5.22
$\texttt{m[S}_1\texttt{*ie}_1\texttt{+...+S}_n\texttt{*ie}_n\texttt{+pe+K]}$	5.23
$\texttt{m[S}_1\texttt{*l}_1\texttt{+...+S}_n\texttt{*l}_n\texttt{+K]}$	5.24
other	5.25

Table 5.2 shows a range of patterns that may be found in the intermediate representation after propagation and dead code elimination, and the propositions which discuss them over the next several pages. It is evident that there are a lot of possible patterns, and distinguishing among them is not a trivial task. Most of this distinguishing is left as future work.

Proposition 5.8: ($\texttt{ie}_j$ present) *Adding a non-constant integer expression to a pointer implies an array access.*

An array is an aggregation of uniform elements, and can be indexed. A structure is an aggregation of any kind of elements, not necessarily identical in type, so that indexing is not possible. Arrays can therefore be indexed with a constant or variable index expression; index expressions are of integer type. Structure elements can only be accessed with constant offsets.

The only place where a non-constant integer expression can appear is therefore as an array index. Hence, when present, the $\texttt{ie}_j$ indicate array indexing, and the overall memory expression references an array element. For an array element access with m dimensions n of which are non-constant, there will be at least n such terms. (One or more index expressions could be of the form `j+k` where `j` and `k` are locations, hence n is a minimum.)

For single dimension arrays whose elements occupy more than one byte, there will be a scale involved. The scaling may be implemented as a shift operation, an add to self, or as a multiply by a constant, but canonicalisation of expressions will change all these into multiply operations. Multidimensional arrays are usually implemented as arrays of subarrays, so the higher order index expressions are scaled by the size of the subarray. The sizes of arrays and subarrays are constant, so the S_j will be constant for any particular array. Variations in S_j between two array accesses indicate either that different arrays are being accessed, or that what appears to be scaling is in at least one case part of the index expressions.

Proposition 5.9: : (`K` present) *When present, the constant `K` of Equation 5.7 could represent the sum of the following:*

(a) The address of a global array, global structure, or global variable (sp_0 and `pl` not present). This component of `K` may be bounded: the front end of the decompiler may be able to provide limits on addresses that fall within the read-only or read/write data sections.

(b) The (possibly zero) offset from the initial stack pointer value to a local variable, array, or structure (sp_0 present).

(c) One or more possibly scaled constant array index(es).

(d) Constants arising from the constant term(s) in array index expressions. For example, if in a 10×10 array of 4-byte integers, the index expressions are `a*b+4` and `c*d+8`, the expression for the offset to the array element is `(a*b+4)*40 + (c*d+8)*4`, which will canonicalise to `40*a*b + 4*c*d + 192`. In the absence of other constants, `K` will then be 192, which comes in part from the constants 4 and 8 in the index expressions, as well as the size of the elements (S_2=4) and the size of the subarray (S_1=40).

(e) Offsets arising from the lower array bound not being zero (for example, `a: array [-20 .. -11] of real`). Where more than one dimension of a multidimensional array has a lower array bound that is non zero, several such offsets will be lumped together. In the C language, arrays always start at index zero, but it is possible to construct pointers into the middle of arrays or outside the extent of arrays, to achieve a similar effect, as shown in Figure 5.16. Note that in Figure 5.16(b), it is possible to pass `a0+100` to another procedure, which accesses the array `a0` using indexes ranging from -100 to -91.

(f) Structure member offset of a variable, array, or structure inside a parent structure. Where nested structures exist, several structure member offsets could be lumped

together to produce the offset from the start of the provided object to the start of the structure member involved. For example, in `s.t.u[i]`, `K` would include the offsets from the start of `s` to the start of `u`, or equivalently the sum of the offsets from the start of `s` to the start of `t` and the start of `t` to the start of `u`.

```
var a:array[-20 ..  -11] of
real;
a[-20] := 0;
process(a);
```

(a) Pascal language.

```
double a0[10], *a =
a0+20;
a[-20] = 0;
process(a);
```

(b) C language.

Figure 5.16: Source code for accessing the first element of an array with a nonzero lower index bound.

Many combinations are possible; e.g. options (d) and (f) could be combined if an array is accessed inside a structure with at least one constant index, e.g. `s.a[5]` or `s.b[5, y]` where `s` is a structure, `a` is an array, and `b` is a two-dimensional array.

The $\texttt{ie}_j$ terms represent the variable part of the index expressions, with the constant part of index expressions split off as part of K, as shown above at option 5.9(d). To save space, statements such as "`ie` represents the variable part of the index expression" will be shortened to "`ie` represents the index expression", with the understanding that the constant parts of the index expressions are actually lumped in with other terms into `K`.

It is apparent that many patterns could be encountered in the IR of a program to be typed, and these make different assertions about the types of the various subexpressions involved. The following propositions summarise the patterns that may be encountered.

Proposition 5.10: *`m[K]` represents a global variable, a global structure member, or a global array element with constant index(es), possibly inside a global structure.*

`K` is the sum of options 5.9(a) and 5.9(c)-(f). As noted in section 3.4.4 on page 83, it is assumed that architectures where global variables are accessed as offsets from a register reserved for this purpose, that register is initialised with a constant value by the decompiler front end to ensure that all global variable accesses (following constant propagation) are of this form. Since this pattern can represent either a global variable, structure element, or an array element with fixed offset(s), elementary types can be promoted to structure or array elements. To fit this into the lattice of types concept, this observation could, by abusing terminology slightly, be expressed as $\xi(\mathsf{array}(\mathsf{int})) \sqsubseteq \mathsf{int}$. The notation $\xi(\mathsf{array}(\alpha))$ here denotes "an element of an array whose elements are of type α". This relation could be read as "the type 'element of an array of int' is a subtype of or equal to type 'variable of int' " (the former, occuring less frequently, is in a sense

more constrained than the latter). The same applies for structure elements; an element of a structure σ whose type is β is written $\xi(\sigma)(\beta)$, and where the structure type is not known or not specified, as $\xi(\text{structure-containing-}\beta)$. These thoughts lead to the following proposition:

Proposition 5.11: *ξ(array(α)) $\sqsubseteq \alpha$, and ξ(structure-containing-α) $\sqsubseteq \alpha$.*

The above proposition will be improved below. It should be noted that the types α and ξ(array(α)) are strictly speaking the same types, so the $\sqsubseteq$ relation is really $=$ (equality), but expressing the various forms of α in this way allows these forms to become part of the lattice of types. When various triggers are found (e.g. K is used as a pointer to an array elsewhere in the program), the type associated with a location can be moved down the lattice (e.g. from α to ξ(array(α))). This makes the handling of array and structure members more uniform with the overall process of type analysis.

Proposition 5.12: *m[sp_0 + K] represents a stack-based local variable, local structure member, or local array element with constant index(es), possibly inside a local structure.*

K represents the sum of options 5.9(b)-(f).

Proposition 5.13: *m[l + K] represents an aggregate (array or structure) element access.*

Here, l is a location, which could be an ie representing an unscaled array index, or a pl representing a pointer to an aggregate. If l is used elsewhere as an integer or a pointer, this expression can be refined to one of the following patterns. Since l+K is used as a pointer, and adding two terms implies that one of the two terms is an integer and the other a pointer, then if the value of K is such that it cannot be a pointer, K must be an integer and l a pointer. As pointed out in Section 5.4, constants are independent, so other uses of the same constant elsewhere do not affect whether l or K is a pointer. Hence, the only factors affecting whether l or K is the pointer in m[l + K] are whether l is used elsewhere as a pointer or integer, and whether K has a value that precludes it from being a pointer.

Proposition 5.14: *m[ie + K] represents a global array element access.*
Here ie is known to be used as an integer. K represents the sum of options 5.9(a) and 5.9(c)-(f), and ie is the possibly scaled index expression. There could be other index expressions for a multidimensional array, all constant.

Proposition 5.15: *`m[pl + K]` represents a structure element access or array element access with constant index(es).*

Here `pl` is a pointer to the array or structure (global, local, or heap allocated), and `K` represents the sum of options 5.9(c)-(f). For example, `m[pl + K]` could represent `s.m` where `s` is a variable of type structure (represented by `pl`), and `m` is a member of `s` (`K` represents the offset from the start of `s` to `m`). It could also represent `a[C]`, where `a` is a variable of type array (represented by `pl`), and C is a constant with the value $\frac{\texttt{K}}{\texttt{sizeof(a[0])}}$. It is not possible to distinguish these two representations unless a nonconstant array access is found elsewhere.

```
struct {
  COLOUR c1;
  COLOUR c2;
  COLOUR c3;
} colours;
colours.c1 = Red;
colours.c2 = Green;
colours.c3 = Blue;
process(colours.c2);
```

(a) Structure.

```
COLOUR colours[3];
colours[1] = Red;
colours[2] = Green;
colours[3] = Blue;
process(colours[2]);
```

(b) Array.

Figure 5.17: Equivalent programs which use the representation `m[pl + K]`.

The two representations for `m[pl + K]` are equivalent, as shown in Figure 5.17. Hence, neither representation is incorrect. The structure representation seems more natural, hence this pattern could be considered a structure reference unless and until an array reference with a nonconstant index is found. In either case, since an array reference with a nonconstant index could be found elsewhere at any time, the array element is in a sense more constrained than the structure element. Again abusing terminology, this can be expressed as $\xi(\text{array}(\alpha)) \sqsubseteq \xi(\text{structure containing } \alpha$.

Combining this with Proposition 5.11 and noting that an array could be contained in a structure results in the following proposition:

Proposition 5.16:
$\xi(\text{structure-containing-}\xi(\text{array}(\alpha))) \sqsubseteq \xi(\text{array}(\alpha)) \sqsubseteq \xi(\text{structure-containing-}\alpha) \sqsubseteq \alpha$.

This leads to the lattice fragment shown in Figure 5.18.

The above example illustrates the point that the lattice of types for decompilation is based at least in part not on the source language or how programs are written, but on how information is uncovered during type analysis.

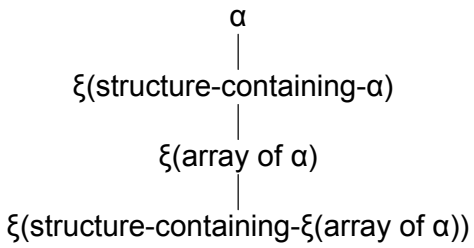


Figure 5.18: A type lattice fragment relating structures containing array elements, array elements, structure members, and plain variables.

Proposition 5.17: *m[sp_0 + ie + K] represents a local array element access.*

`ie` is the index expression, and K lumps together options 5.9(b)-(f).

There is no pattern `m[`sp_0`+pl+K]` since sp_0 and `pl` are both pointers, and pointers are assumed never to be added.

The pattern `m[`l_1`+`l_2`+K]` where l_1 and l_2 are locations is interesting because both locations could match either `ie` or `pl`. Until other uses of l_1 or l_2 are found that provide type information about the locations, it is not known which of the locations or `K` is a pointer.

Proposition 5.18: *m[l_1 + l_2 + K] represents an array element access.* If either l_1 or l_2 is used elsewhere as a pointer, or K has a value that cannot be a pointer and either l_1 or l_2 is used elsewhere as a pointer or integer, this expression can be refined to the following pattern:

> **Proposition 5.19:** *m[ie + pl + K] represents an array element access.*
> `ie` is an index expression, `pl` is a pointer to the array (global, local, or heap allocated), and *K* represents the sum of options 5.9(c)-(f).

If both l_1 and l_2 are used elsewhere as integers, then Proposition `5.18` can be refined to the following pattern:

> **Proposition 5.20:** *m[ie_1 + ie_2 + K] represents a global array element access.*
> The non-constant index expression is `ie`$_1$`+ie`$_2$, and K represents the sum of options 5.9(a) and (c)-(f).

Another special case of the pattern in Proposition `5.18` is when `K=0`. One of l_1 and l_2 must be an integer, and the other a pointer. From proposition 5.8, this implies that the integer expression represents array indexing. If one of l_1 or l_2 are used elsewhere as a pointer or integer, this pattern can be refined to the following pattern.

Proposition 5.21: *m[ie + pl] represents an array element access.*
`ie` is the possibly scaled index expression, and `pl` is a pointer to the array.

Proposition 5.22: *m[S_1*ie_1 + S_2*ie_2 + ... + S_n*ie_n + K] represents an m-dimensional global array element access.* Here the ie_j are the index expressions, $S_1..S_n$ are scaling constants, and K lumps together options 5.9(a) and 5.9(c)-(f). If there are index expressions with more than one location, then m could be less than n. If there are constant index expressions, then m could exceed n. These two factors could cancel out or not be present, in which case $m = n$.

Proposition 5.23: *m[pl + S_1*ie_1 + S_2*ie_2 + ... + S_n*ie_n + K] represents an m-dimensional array element access.*

Here pl points to the array or structure containing the array, the ie_j are the index expressions, S_1.. S_n are scaling constants, and K lumps together options 5.9(a) and 5.9(c)-(f).

Proposition 5.24: *m[sp_0 + S_1*l_1 + S_2*l_2 + ... + S_n*l_n + K] represents an m-dimensional local array element access.*

Here the l_i are the index expressions, $S_1..S_n$ are scaling constants, and K lumps together options 5.9(b)-(f).

Proposition 5.25: *Other expressions are assumed to be indirections on pointer expressions.*

Examples in the C language include `(*p)[i]` and `*(ptrs[j])`, where the above expressions have been applied to part of the overall expression to yield the array indexes.

5.9 Partitioning the Data Sections

Decompilers need a data structure comparable to the compiler's symbol table (which maps symbols to addresses and types) to map addresses to symbols and types.

Section 5.2.3 on page 155 noted that aggregate types are usually manipulated with a series of machine code instructions, rather than individual instructions. For example, to sum the contents of an array, a loop is usually employed. If a structure has five elements of elementary types, there will usually be at least five separate pieces of code to access all the elements. In other words, aspects of aggregate types such as the number of elements and the total size emerge as the result of many instructions, not individual instructions. This contrasts with the type and size of elementary types, which are usually available from a single instruction.

Hence a data structure is required to build up the "picture" of how the data section is composed. This process could be thought of as partitioning the data section into the various variables of various types. This data structure is in a sense the equivalent of the symbol table in the compiler or assembler which allocated addresses to data originally. In a compiler or assembler, the symbol table is essentially a map from a symbolic name to a type and a data address. In a decompiler, the appropriate data structure, which could be called a *data map*, is a map from data address to a symbolic name and type.

At least two address spaces need to be considered: the global address space, containing global variables, and the stack local address space, containing local variables. There is also the heap address space, where variables, usually aggregates, are created with a language keyword such as `new` or a call to a heap allocating library function such as `malloc`. However, allocations for heap objects are usually for one object at a time, and the addresses by design do not overlap. A separate data map would be used for each address map.

5.9.1 Colocated Variables

Escape analysis is needed to determine the validity of separating colocated variables.

As a space saving optimisation, compilers may allocate more than one variable to the same address. Once this has been determined to be safe, such colocation has little cost for a compiler; it merely has some entries in the symbol table that have the same or overlapping values for the data addresses. For a decompiler, however, the issue is more complex. The data address does not uniquely identify a data map entry, as shown in Figure 5.19.

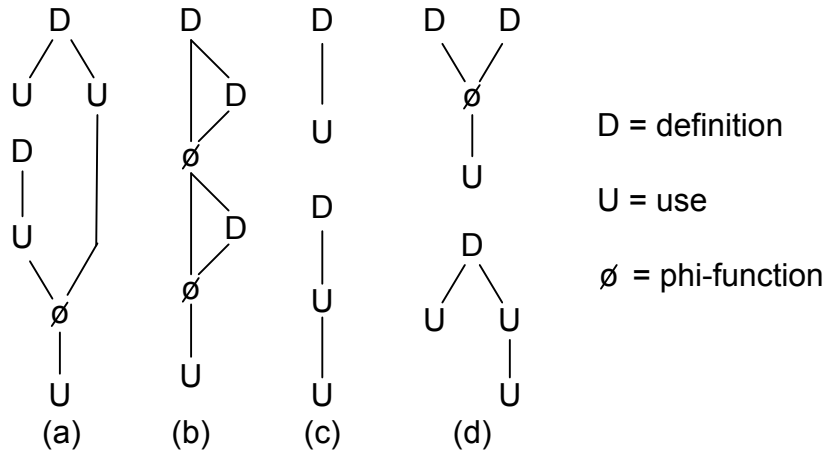


Figure 5.19: Various configurations of live ranges for one variable.

In Figure 5.19(a) and (b), although there are multiple definitions for the variable, the

definitions and uses of the variable are united by ϕ-functions. In cases (c) and (d), there is more than one live range for the variable, made obvious by the breaks in data flow from one live range to the next. Where there is more than one live range for a variable and the types of the variable are different in the live ranges, two or more variables must be emitted, as the compiler has clearly colocated unrelated variables, and each named variable can have only one type.

Although more than one name must be generated, there is still the option of uniting the addresses of those names (e.g. with a C union), or separating them out as independent variables, which the compiler of the generated code could assign to different addresses.

In cases where the types for the various live ranges agree, however, it is not possible to decide in general whether the compiler has colocated unrelated variables that happen to have the same type, or if the original source code used one variable in several live ranges. An example of where the latter might happen is where an array index is re-used in a second loop in the program. Always separating the live ranges into multiple variables could clutter the generated source code with excess variable declarations. Never separating the live variables into separate variables could result in confusion when the variable is given a meaningful name. The name that is meaningful for one live range could be incorrect for other live ranges. This is a case where an expert user may profitably override the default decompiler behaviour. Despite the potential for lessened readability, there is no case here where incorrect output is unavoidable.

Sometimes the intermediate representation will effectively be taking the address of a variable or aggregate element. At the high level, this may be explicit, as with the `&` unary operator in C, or implicitly, as in Java when an object is referenced (references are pointers at the bytecode level). This taking of the address may be far away from the eventual use of that variable or aggregate element. The type patterns for the address of a variable or aggregate element are as per Section 5.8, but with the outer `m[...]` operation removed. In some cases, these will be very common expressions such as `K` or `ie + K`. Clearly, not all such patterns represent the address of a global variable, or the address of an aggregate element.

These patterns are therefore employed after type analysis, and are only used when type analysis shows the pattern (excluding the `m[...]`) is a pointer type. Obviously, the patterns of Section 5.8 with the `m[...]` guarantee that the inner expression is used as a pointer.

When the address of a variable is taken, the variable is referenced, but it is not directly defined or used. Data flow information ultimately comes from definitions or uses, but when the reference is passed to a library function, the usage information is usually condensed to *constant reference* or *non-constant reference* of a specified type. A constant

reference implies that the location being referenced is used but not defined. A non constant reference could imply various definition and use scenarios, so the conservative summary is "may define and may use". The purpose of the library function may carry more data flow information than its prototype. For example, the *this* parameter of `CString::CString()` (the constructor procedure for a string class) does not use the referenced location (`*this`) before defining it. This extra information could help separating the live ranges of variables whose storage is colocated with other objects. While it may be tempting to use types to help separate colocated variables, the possibility that the original program used casts makes types less reliable for this purpose. In the case of the CString constructor, a new live range is always being started, but this fact is not captured in the prototype for the constructor procedure.

```
mov $5,-16(%esp)     ; Define
print -16(%esp)      ; Use as int
lea -16(%esp),%edi   ; Take the address, → edi
call proc1
...
mov $proc2,-16(%esp); Define
call -16(%esp)       ; Use as proc*
...
mov $'a',-16(%esp)   ; Define
putchar -16(%esp)    ; Use as char
...
process((%edi))      ; Use saved address (as char)
```

(a) Machine code

```
int i;
void (*p)(void);
char c, *edi;
i=5;
print(i);
edi = &c;
proc1();
...
p = proc2;
(*p)();
...
c='a';
putchar(c);
...
process(*edi);
```

(b) Decompiled code

Figure 5.20: A program with colocated variables and taking the address.

Taking the address of a variable that has other variables colocated can cause an ambiguity as to which object's address is being taken. For example, consider the machine code program of Figure 5.20(a). The location `m[esp-16]` has three separate variables sharing the location. Suppose that it is known that `proc1` and `proc2` preserve and do not use `edi`. The definition of `m[esp-16]` with the address of `proc2` completely kills the live range of the integer variable, yet the address taken early in the procedure turns out to be used later in the program, at which point only the char definition is live. Definitions of `m[esp-16]` do not kill the "reach" of the separate variable `edi`, which continues to hold the value `esp-16`. It is the interpretation of what `esp-16` represents that changes with definitions of `m[esp-16]`, from `&i` to `&proc2` to the address of a character. Type information about `edi` is linked to the data flow information about

`m[esp-16]` by the address taking operation.

In this case, if there were no other uses of `edi` and no other instructions took the address of `m[esp-16]`, the three variables could be separated in the decompiled output, as shown in Figure 5.20(b). However, if the address of `m[esi-16]` escaped the procedure (e.g. `edi` is passed to `proc1`), this separation would not in general be safe. For example, it may not be known whether the procedure the address escapes to (here `proc1`) defines the original variable or not. If it only used the original variable, then it would be used as an integer, and the reference is to `i` in Figure 5.20(b). However, it could define and use the location with any type. Finally, it could copy the address to a global variable used by some procedure called before the variable goes out of scope. In this case, the type passed to proc1 depends on which of the colocated variables is live when the location is actually used. (The compiler would have to be very smart to arrange this safely, but it is possible.) Hence if the address escapes, the conservative approach is to declare the three variables as a union, just as the machine code in effect does. Such *escape analysis* is commonly performed by compilers to ensure the validity of optimisations; decompilers require it to ensure the validity of separating colocated variables.

If `proc1` in the example of Figure 5.20 was a call to a library function to which `m[esp-16]` was a non-constant reference parameter, the same ambiguity arises. The type of the parameter will be a big clue, but because of the possibility of casts, using type information may lead to an incorrect decompilation. Hence, while library functions are an excellent source of type information, they are not good sources of the data flow information that can help separate colocated variables. This could be a case where it is useful to at least partially decompile a library function.

Colocated variables are a situation where one original program address represents more than one original variable. Figure 1.6 on page 20 showed a program where three arrays are accessed at the machine code level using the same immediate values (one original pointer and two offset pointers). This is a different situation, where although the three variables are located at different addresses, the same immediate constant is used to refer to these with indexes of different ranges. It illustrates the problem of how the constant K of Equation 5.7 could have many components, in particular that of Proposition 5.9(e).

Figure 5.21 shows the declation of three nested structures. The address of the first element could refer to any of `large`, `large.medium`, `large.medium.small`, or `large.medium.small.a`. The types of each of these is distinct, so that type analysis can be used to decide which of the possible references is required.

```
struct {
    struct {
        struct {
            int a;
            ...
        } small;
        ...
    } medium;
    ...
} large;
```

Figure 5.21: Nested structures.

5.10 Special Types

A few special types are needed to cater for certain machine language details, e.g. upper(float64).

In addition to the elementary types, aggregates, and pointers, a few special types are useful in decompilation. There is an obvious need for no type information ($\top$ in a type lattice, or the C type void), and possibly overconstraint ($\bot$ in a type lattice). Many machines implement double word types with pairs of locations (two registers, or two word sized memory locations). It can therefore be useful to define upper(τ) and lower(τ) where τ is a type variable, to refer to the upper or lower half respectively of τ. For larger types, these could be combined, e.g. lower(upper(float128)) for bits 64 through 95 of a 128-bit floating point type. For example, size32 would be compatible with upper(float64). Pairs of upper and lower types can be coalesced into the larger type in appropriate circumstances, e.g. where a double word value is passed to a function at the machine language level in two word sized locations, this can be replaced by one variable (e.g. of type double).

5.11 Related Work

Most related work is oriented towards compilers, and hence does not address some of the issues raised by machine code decompilation.

Using iterative data flow equations to solve compiler problems has been discussed by many authors, starting with Allen and Cocke [All70, AC72] and also Kildall [Kil73]. The theoretical properties of these systems were proved by Kam and Ullman [KU76].

Khedker, Dhamdhere and Mycroft argued for a more complex data flow analysis framework, but they attempted to solve the more difficult problem of type inferencing for dynamically typed languages [KDM03].

Guilfanov [Gui01] discussed the problem of propagating types from library functions through the IR for an executable program. However, he did not attempt to infer types intrinsic in instructions.

Data flow problems can be solved in an even more sparse manner than that enabled by the SSA form, by constructing a unique evaluation graph for each variable [CCF91]. However, this approach suffers from the space and time cost of generating the evaluation graph for each variable, and some other tables required by this framework.

Guo et al. reported on a pointer analysis for assembly language, which they suggested could be extended for use at run-time [GBT+05]. In their treatment of addition instructions, they assumed that the address expressions for array and structure element accesses will be of the form $r2 + i \times l + c$, where i is a non-constant (index) value, l is the size of the array elements, and c is a constant. This contrasts with the more complex expression of Equation 5.7. The latter is more complex mainly to express multidimensional arrays, e.g. `a[x][y]` as e.g. `m[r1 * 40 + r2 * 4] + 8`. However, the latter could be normalised slightly differently to `m[(r1 * 10 + r2) * 4] + 8`, representing `a[x*10+y]`, where 10 is the number of elements in the row. It seems possible that machine code arrays could be analysed like this, and converted back to `a[x][y]` or `a[x, y]` format at a later stage.

5.12 Future Work

While good progress has been made, much work remains before type analysis for machine code decompilers is mature.

Most of the ideas presented in this chapter have been at least partially implemented in the Boomerang decompiler [Boo02]. The basic principles such as the iterative data flow based solution to the type analysis problem work well enough to type simple programs. However, experimental validation is required for several of the more advanced aspects of type analysis, including

- the more unusual cases involving pointer-sized add and subtract instructions (Section 5.7.4 on page 177);
- splitting the value of K into its various possible origins (Proposition 5.9 on page 181), which will probably require range analysis for pointers and index variables;
- separating colocated variables and escape analysis (Section 5.9.1 on page 187); and

- partitioning the data sections (Section 5.9 on page 186).

Range analysis of pointers is also important for initialised aggregate data, as noted in Section 5.2.4.

The current handling of arrays assumes a C-like "array of arrays" implementation of multidimensional arrays. It should be possible to modify the high level patterns to accommodate other implementations.

Object oriented languages such as C++ introduce more elements to be considered, such as member pointers, class hierarchies, and so on. Some of these features are discussed in the next chapter.

5.13 SSA Enablements

Expression propagation, enabled by the SSA form, combines with simplification to prepare memory expressions for high level pattern analysis, and the SSA form allows a sparse representation for type information.

The high level patterns of Section 5.8 on page 178 require the distinction of integer constants from other integer expressions. This is readily achieved by the combination of expression propagation and simplification that are enabled by the Static Single Assignment form. These also eliminate partial constants generated by RISC compilers, which would have been awkward to deal with in any type analysis system.

The SSA form also allows a sparse representation of type information at the definitions of locations. One type storage per SSA definition contrasts with the requirement of one type storage per live variable per basic block, as would be required by a traditional bit vector based iterative framework.

Chapter 6

Indirect Jumps and Calls

While indirect jumps and calls have long been the most problematic of instructions for reverse engineering of executable files, their analysis, facilitated by SSA, yields high level constructs such as switch statements, function pointers, and class types.

When decoding an input executable program, indirect jump and call instructions are the most problematic. If not for these instructions, a recursive traversal of the program from all known entry points would in most cases visit every instruction of the program, thereby separating code from data [VWK+03]. This assumes that all branch instruction targets are valid, there is no self modifying code, and the program is "well behaved" in the sense of call and return instructions doing what they are designed to do.

Indirect jumps and calls are decoded after loading the program file, and before data flow analysis, as shown in Figure 6.1.

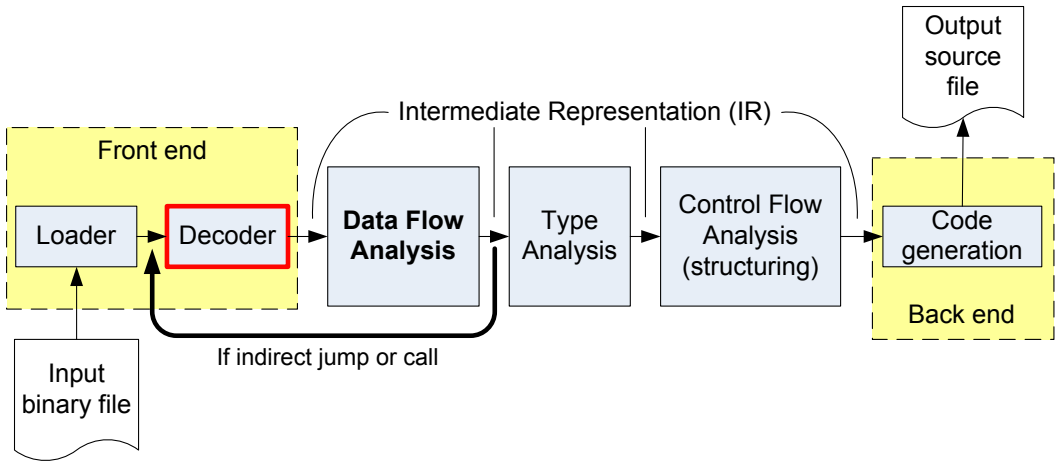


Figure 6.1: Decoding of instructions in a machine code decompiler.

It is interesting to note that it is *indirect* jump and call instructions that are most problematic when reconstructing the control flow graph, and it is *indirect* memory

operations (e.g. `m[m[`x`]]`) which include the most problems (in the form of aliases), in data flow analysis [CCL+96].

The following sections describe various analyses, most facilitated by the static single assignment form (SSA form), which convert indirect jumps and calls to the appropriate high level constructs.

6.1 Incomplete Control Flow Graphs

Special processing is needed since the most powerful indirect jump and call analyses rely on expression propagation, which in turn relies on a complete control flow graph (CFG), but the CFG is not complete until the indirect transfers are analysed.

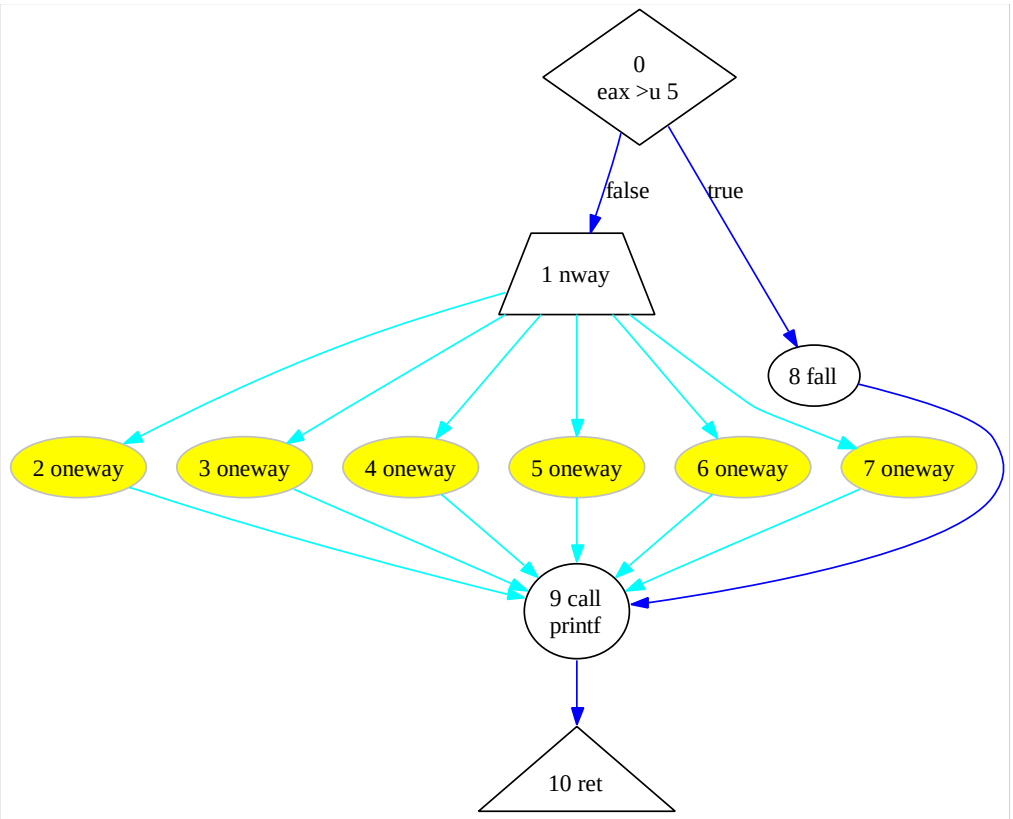
The analysis of both indirect jumps and indirect calls share a common problem. It is necessary to find possible targets and other relevant information for the location(s) involved in the jump or call, and the more powerful techniques such as expression propagation, and value and/or range analysis have the best chance of computing this information. These techniques rely heavily on data flow analyses, which in turn rely on having a complete CFG. Until the analysis of indirect jumps and calls is completed, however, the CFG is not complete. This type of problem is often termed a *phase ordering* problem. While initially it would appear that this chicken and egg problem cannot be resolved, consider that each indirect jump or call can only depend on locations before the jump or call is taken, and consequently only instructions from the start of the procedure to that jump or call instruction need be considered.

Figure 6.2 illustrates the problem. It shows a simple program containing a switch statement, with one print statement in each arm of the switch statement (including the default arm, which is executed if none of the switch cases is selected). Part (b) of the figure shows the control flow graph. Before the n-way branch is analysed, the greyed basic blocks are not part of the graph (they are code that is not yet discovered). As a result, the data flow logic is able to deduce that the definition of the print argument in block 8 (the string `"Other!"`) can be propagated into the print statement in block 9. In fact, basic blocks 8 and 9 are not yet separate at this stage. However, one of the rules for safe propagation is that there are no other definitions of the components of the right hand side of the assignment to be propagated which reach the destination (Section 3.1 on page 65). Once the n-way branch is analysed, however, it is obvious that this propagation is invalid.

One way to correct this problem would be to force conservative behaviour (in this case, not propagating), by somehow anticipating the possibility of in-edges to block 8/9.

```
int main(int argc) {
  switch(argc) {
    case 2:  printf("Two!\n");  break;    case 3:  printf("Three!\n"); break;
    case 4:  printf("Four!\n"); break;    case 5:  printf("Five!\n");  break;
    case 6:  printf("Six!\n");  break;    case 7:  printf("Seven!\n"); break;
    default:printf("Other!\n");break;
  }
  return 0;
}
```

(a) C Source code for the example program.



(b) Control Flow Graph for the above program. Shaded basic blocks are not discovered until the n-way (switch) block is analysed.

Figure 6.2: Example program with switch statement.

However, nothing in the decoded instructions indicates such a possibility. It should be noted that restricting the propagation too much will result in the failure of the n-way branch analysis.

Another way would be to store undo information for propagations, so that propagations found to be invalid after an indirect branch could be restored. At minimum, the original expression before propagation would need to be stored, possibly in the form of the original SSA reference to its definition. Dead code elimination is not run until much

later, so the original definition is guaranteed to still exist.

The simplest method, costly in space and time, is to make a copy of the procedure's IR just after decoding, and the data flow for the whole procedure is restarted after every indirect jump (of course, making use of the new jump targets).Where nested switch statements occur, several iterations of this may be needed; in practice, more than one or two such iterations would be required. This is the approach taken by the Boomerang decompiler [Boo02]. It causes a loop in the top level of the decompiler component graph (as shown in Figure 6.1).

Simpler techniques not requiring propagation could be used for the simpler cases, to reduce the number of times the expensive restarts are required.

Indirect calls that are not yet resolved have to assume the worst case preservation information. It is best to recalculate the data flow information for these calls after finding a new target, so that less pessimistic propagation information can be used. Whether the analysis of the whole procedure needs to be restarted after only indirect calls are analysed depends on the design of the decompiler.

6.2 Indirect Jump Instructions

Indirect jump instructions are used in executable programs to implement switch (case) statements and assigned goto statements, and tail-optimised calls.

Compilers often employ indirect jump instructions (branches) to implement switch or case statements, unless the number of cases is very small, or the case values are sparse. They can also be used to implement assigned goto statements, and possibly exception handling [SBB+00]. Finally, any call at the end of a procedure can be tail-call optimised to a jump. It is assumed that jumps whose targets are the beginning of functions have been converted to call statements followed by return statements.

6.2.1 Switch Statements

Switch statements can conveniently be analysed by delaying the analysis until after expression propagation.

The details of the implementations of switch statements are surprisingly varied. Most implementations store direct code pointers in a jump table associated with the indirect jump instruction. Some store offsets in the table, usually relative to the start of the table, rather than the target addresses themselves. This is presumably done to minimise

the number of entries in the relocation tables in the object files; relocation table entries are not visible in executable files.

One approach to recognising these branch table implementations is to slice backwards from the indirect jump until one of a few "normal forms" is found [CVE01].

Table 6.1: High level expressions for switch expressions in Boomerang.

Form	High level pattern	Table entry
A	`m[<`*expr*`>*4 + T]`	Address
O	`m[<`*expr*`>*4 + T] + T`	Offset
R	`%pc + m[%pc + <`*expr*`>*4 + k]`	Offset
r	`%pc + m[%pc + <`*expr*`>*4 - k] - k`	Offset

Table 6.1 shows the high level expressions for various forms of switch statement for the Boomerang decompiler. `T` is a constant representing the address of the table; `k` is a small constant (in the range 8 to a few thousand). The expressions assume a 32-bit architecture, with a table size of 4 bytes. `%pc` represents the program counter (Boomerang is not very precise about what point in the program `%pc` represents, but it does not need to be).

These expressions are very similar to those from Figure 7 of [CVE01], which builds up the expression for the target of the indirect branch by slicing backwards through the program at decode time. This approach was used initially in the Boomerang decompiler, however, the variations seen in real programs caused the code to become very complex and difficult to maintain. Since a decompiler needs to perform constant and other propagation, it seems natural to use this powerful technique instead of slicing, despite the necessity of restarting analyses. By delaying the analysis of indirect branches until after the IR is in SSA form and expression propagation has been performed, the expression for the destination of the call appears in the IR for the indirect branch with no need for any slicing or searching.

This delayed approach has the advantage that the analysis can bypass calls if necessary, which is not practical when decoding. There is no need to follow control flow edges, and the analysis automatically spans multiple basic blocks if necessary. Figure 6.3 shows the IR for the program of Figure 6.2.

The expression in statement 11 (`m[((eax`$_4$ `- 2) * 4) + 0x8048934]`) matches form A, with *expr* = `eax`$_4$ `- 2` and `T = 0x8048934`. The number of cases is found by searching the in-edge of the switch basic block for 2-way basic blocks. This is slicing in a sense, but it is simpler because the branch expression has been propagated into the condition of the branch instruction at the end of the 2-way basic block. In this case, the

```
 1 m[esp0 - 4] := ebp0
 3 ebp3 := esp0 - 4
 4 eax4 := m[esp0 + 4]0
 6 eax6 := eax4 - 2
10 BRANCH to 0x804897c if (eax4 - 2) >u 5
11 CASE [m[((eax4 - 2) * 4) + 0x8048934]]
```

Figure 6.3: IR for the program of Figure 6.2.

branch expression is `(eax4 - 2) >u 5` where `(eax4 - 2)` matches ***expr***. The number of cases is readily established as six (branch to the default case if ***expr*** is greater than 5).

6.2.1.1 Minimum Case Value

Where there is a minimum case value, expression propagation, facilitated by the SSA form, enables a very simple way to improve the readability of the generated switch statement.

In Figure 6.2, there is a minimum switch case value of 2. Typical implementations subtract 2 from the switch variable before comparing the result against the number of case values, including any gaps. Hence the comparison in block 0 is against 5, not 7. Without checking the case expression, this would result in the program of Figure 6.4.

```
int main(int argc) {
  switch(argc-2) {
    case 0:  printf("Two!\n");  break;    case 1:  printf("Three!\n"); break;
    case 2:  printf("Four!\n"); break;    case 3:  printf("Five!\n");  break;
    case 4:  printf("Six!\n");  break;    case 5:  printf("Seven!\n"); break;
    default:printf("Other!\n");break;
  }
  return 0;
}
```

Figure 6.4: Output for the program of Figure 6.2 when the switch expression is not checked for subtract-like expressions.

This program, while correct, is less readable than the original. There is a simple check that can be performed to increase readability: if the switch expression is of the form ℓ-K where ℓ is a location and K is an integer constant, simply add K to all the switch case values, and emit the switch expression as ℓ instead of ℓ-K. This simple expedient shows the power of expression propagation, canonicalisation, and simplification, which are facilitated by the SSA form. It saves a lot of checking of special cases, following possibly multiple in-edges to find a subtract statement, and so on. This improvement can be trivially extended to handle the case of ℓ+K.

In the example of Figure 6.3, ℓ is $\texttt{eax}_4$ and `K` is 2. The generated output is essentially as per the original source code.

6.2.1.2 No Compare for Maximum Case Value

There are three special cases where an optimising compiler does not emit the compare and branch that usually sets the size of the jump table.

Occasionally, the maximum value of a switch value is evident from the switch expression. Examples include ℓ`%K`, ℓ`&(N-1)`, and ℓ`|(-N)`, where K is any nonzero integer constant (e.g. 5), and N is an integral power of 2 (e.g. 8). For example, ℓ`|(-8)` will always yield a value in the range -1 to -8. In these cases, an optimising compiler may omit the comparison against the maximum switch case value, if it is redundant (i.e. the highest valued switch case is present).

Rarely, a compiler may rely on a range restriction on a procedure parameter that is known to be met by all callers. In these cases, range analysis will be needed to find the number of entries in the jump table.

6.2.2 Assigned Goto Statements

Fortran assigned goto statements can be converted to switch statements, removing an unstructured statement, and permitting a representation in languages other than Fortran.

Figure 6.5 shows a simple program containing an assigned goto statement. Many modern languages do not have a direct equivalent for this relatively unstructured form of statement, including very expressive languages such as ANSI C. The similarity to the switch program in Figure 6.2 suggests that the switch statement could be used to express this program. For n assignments to the goto variable, there are n possible destinations of the indirect jump instruction, which can be expressed with a switch statement containing n cases. The case items are somewhat artificial, being native target addresses of the indirect jump instruction.

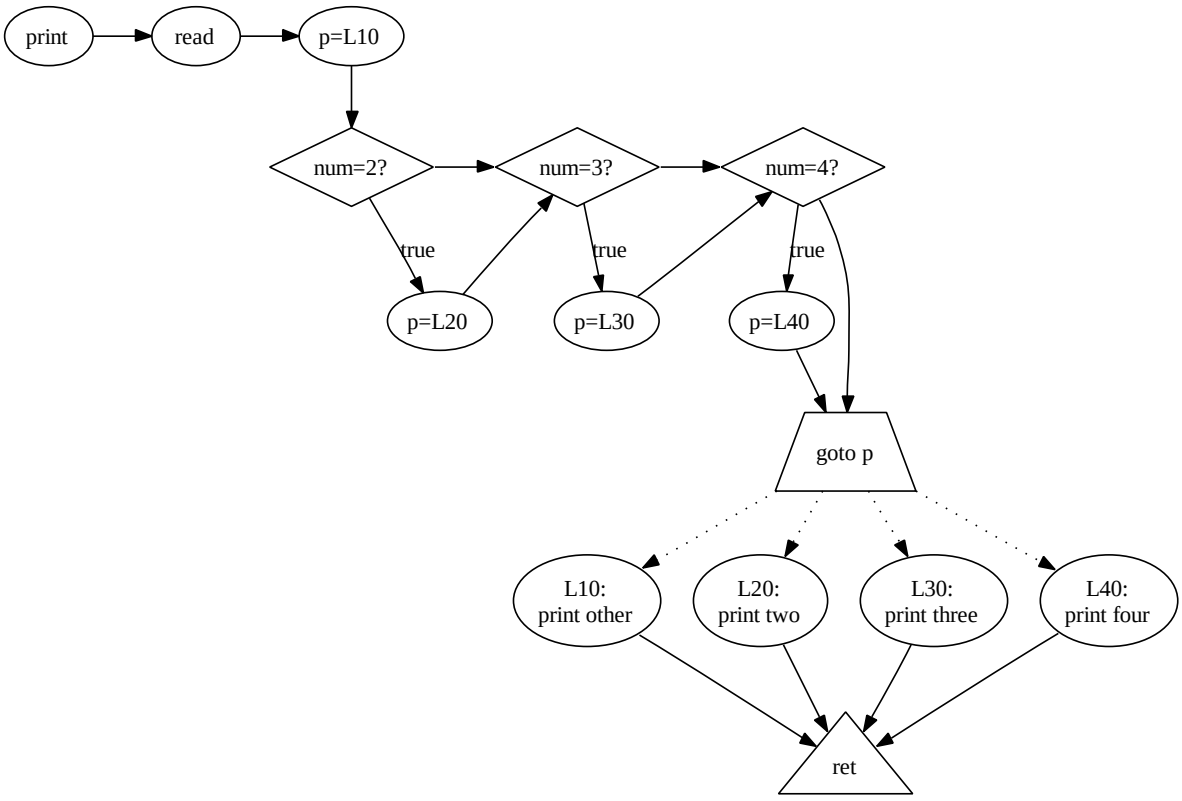
SSA form enables the task of finding of these targets efficiently. Figure 6.6 shows the detail. In effect, the ϕ-statements form a tree, with assignments to constants (represented here by the letter L and their Fortran labels) at the leaves. It is therefore straightforward to find the list of targets for the indirect jump instruction.

```
program asgngoto
    integer num, dest
    print*, 'Input num:'
    read*, num
    assign 10 to dest
    if (num .eq.  2) assign 20 to dest
    if (num .eq.  3) assign 30 to dest
    if (num .eq.  4) assign 40 to dest
    * The computed goto:
    goto dest, (20, 30, 40)
10  print*, 'Input out of range'
    return
20  print*, 'Two!'
    return
30  print*, 'Three!'
    return
40  print*, 'Four!'
    return
    end
```

(a) Fortran source code.

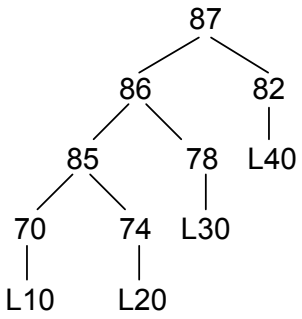
```
70 local2 := 134514528
73 if (local1 ≠ 2) goto L1
74 local2 := 134514579
L1:
85 local2 :=
       φ{local2_70, local2_74)
77 if (local1 ≠ 3) goto L2
78 local2 := 134514627
86 local2 :=
       φ{local2_85, local2_78)
81 if (local1 ≠ 4) goto L3
82 local2 := 134514675
L3:
87 local2 :=
       φ{local2_86, local2_82)
84 goto local2_87
```

(b) Intermediate representation.



(c) Control flow graph.

Figure 6.5: A program using an assigned goto.



Statement 84 of Figure 6.5(b) shows the case statement location to be subscripted with 87. Statement 87 contains its definition, a phi statement. Effectively, this line says that the location defined in statement 87 was defined at either statement 86 or 82. Statement 82 assigns the `goto` variable (`location2`) with a constant.

Figure 6.6: Tree of ϕ-statements and assignments to the goto variable from Figure 6.5.

Figure 6.7 shows the resultant decompiled program. The output is not ideal, in that aspects of the original binary program (the values of the labels) are visible in the decompiled output. However, the computed goto is represented in the output language, and is correct.

```
local2 = 0x8048760;      /* assign 10 to dest */
if (local1 == 2) {       /* if (num .eq.  2) */
  local2 = 0x8048793;    /* assign 20 to dest */
}
if (local1 == 3) {       /* if (num .eq.  3) */
  local2 = 0x80487c3;    /* assign 30 to dest */
}
if (local1 == 4) {       /* if (num .eq.  4) */
  local2 = 0x80487f3;    /* assign 40 to dest */
}
switch(local2) {         /* goto dest, (20, 30, 40) */
  case 0x8048760:        /* Label 10 */
    ...                  /* 'Out of range' */
  case 0x8048793:        /* Label 20 */
    ...                  /* 'Two' */
```

Figure 6.7: Decompiled output for the program of Figure 6.5. Output has been edited for clarity.

6.2.2.1 Other Indirect Jumps

Indirect jump instructions that do not match any known pattern have long been the most difficult to translate, but value analysis combined with the assigned goto switch statement variant can be used to represent them adequately.

The above assigned goto analysis has generated code which would be suitable for an indirect jump instruction which does not match any of the high level patterns; all that

is needed is the set of possible jump instruction targets. There may be cases where the jump instruction targets are not available via a tree of ϕ-statements as is the case in the program of Figure 6.5. An analysis that provides an approximation of the possible values for a location (e.g. the value set analysis (VSA) of Reps et al. [BR04, RBL06]) would enable such indirect jump instructions to be correctly represented in the decompiled output. The imprecision of the value analysis may cause some emitting of irrelevant output, or omission of required output, but this is still a considerable advance over not being able to handle such jump instructions at all.

6.2.2.2 Other Branch Tree Cases

Branch trees or chains are found in switch statements with a small number of cases, and subtract instructions may replace the usual compare instructions, necessitating some logic to extract the correct switch case values.

Branch trees are also found in other situations, such as a switch statement with a small number of cases. Figure 6.8 shows a program fragment with such an example. In this case, with only three cases, there isn't a tree as such, but the implementation has an interesting difference, as shown in Figure 6.9.

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, ...)
{ ...
  switch (message) {
    case WM_COMMAND:                    // WM_COMMAND = 273
      wmId = LOWORD(wParam); ...
    case WM_PAINT:                      // WM_PAINT = 15
      hdc = BeginPaint(hWnd, &ps); ...
    case WM_DESTROY:                    // WM_DESTROY = 2
      PostQuitMessage(0); ...
    default:
      return DefWindowProc(hWnd, message, wParam, lParam);
  }
}
```

Figure 6.8: Source code for a short switch statement with special case values.

Although the three switch values are 2, 15, and 273, the values compared to are 2, 13, and 258. This is because instead of three compare and branch instruction pairs, the compiler has chosen to emit three subtract and branch pairs. (Perhaps the reasoning is that the differences between cases are usually less than the case values themselves, and the x86 target has more compact instruction forms for smaller immediate values).

The decompiler has already transformed the relational expression `param5-2 == 0` to `param5 == 2` for the first comparison. Care needs to be taken to ensure that similar

```
if (param5 == 2) {                  // 2 = WM_DESTROY
  PostQuitMessage(0);
} else {
  if (param5 - 2 == 13) {           // 13 = WM_PAINT - WM_DESTROY
    BeginPaint(param4, &param2); ...
  } else {
    if (param5 - 15 == 258) {       // 258 = WM_COMMAND - WM_PAINT
      if ((param6 & 0xffff) == 104) {...
      }
    }
  } else {
    DefWindowProcA(param4, param5, param6, param7);
  }
}
```

Figure 6.9: Direct decompiled output for the program of Figure 6.8.

transformations are applied to the other two relational expressions, so that the true switch values are obtained. Yet again, these manipulations are easier after expression propagation, with no need to trace backwards through the intermediate representation.

6.2.3 Sparse Switch Statements

Sparse switch statements usually compile to a branch tree which can be transformed into a switch-like statement for greater readability.

```
int main() {
  int n; printf("Input a number, please:  "); scanf("%d", &n);
  switch(n) {
    case 2:  printf("Two!\n"); break;
    case 20:  printf("Twenty!\n"); break;
    case 200:  printf("Two Hundred!\n"); break;
    case 2000:  printf("Two thousand!\n"); break;
    case 20000:  printf("Twenty thousand!\n"); break;
    case 200000:  printf("Two hundred thousand!\n"); break;
    case 2000000:  printf("Two million!\n"); break;
    case 20000000:  printf("Twenty million!\n"); break;
    case 200000000:  printf("Two hundred million!\n"); break;
    case 2000000000:  printf("Two billion!\n"); break;
    default:           printf("Other!\n");
  }
  return 0;
}
```

Figure 6.10: Source code using a sparse switch statement.

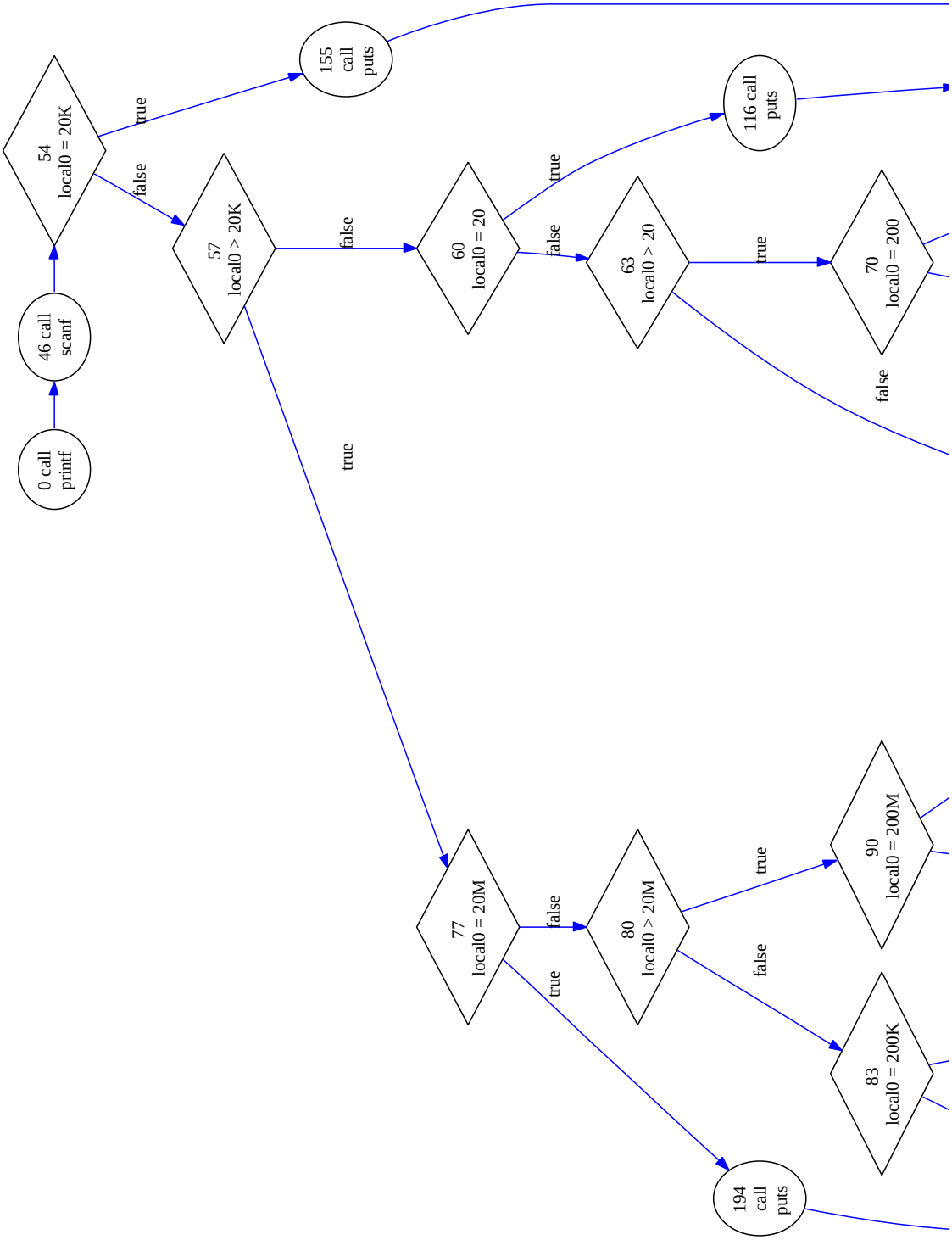


Figure 6.11: Control Flow Graph for the program of Figure 6.10 (part 1).

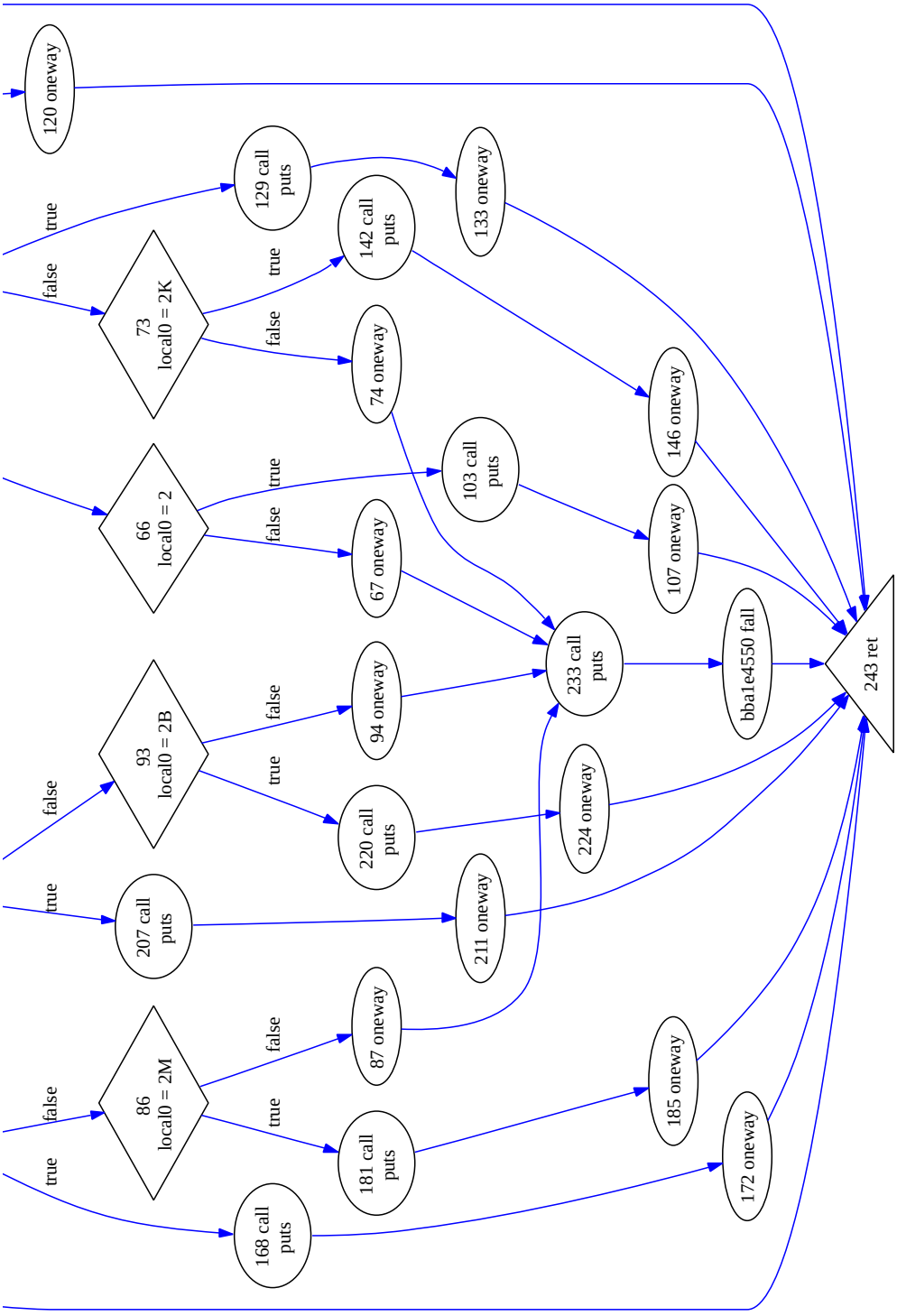


Figure 6.11: Control Flow Graph for the program of Figure 6.10 (part 2).

While Figure 6.6 shows a tree of values leading to an indirect jump instruction, trees also feature in the control flow graph of sparse switch statements. Compilers usually emit a tree of branch instructions to implement sparse switch statements. Figure 6.10 shows C source code for such a program, and Figure 6.11 shows its control flow graph.

Note that no indirect jump instruction is generated; a jump table would be highly inefficient. In this case, almost two billion entries would be needed, with only ten of these actually used. The compiler emits a series of branches such that a binary search of the sparse switch case space is performed. Without special transformations, the decompiled output would be a series of highly nested conditional statements. This would be correct, but less readable than the original code with a switch statement.

It should be possible to recognise this high level pattern in the CFG. There are no control flow edges from other parts of the program to the conditional tree, and no branches outside what will become the switch statement. The follow node (the first node that will follow the generated switch statement) is readily found as the post dominator of the nodes in the conditional tree. The switch cases are readily found by searching the conditional tree for instructions comparing with a constant value. Range analysis would be better, since it would allow for ranges of switch cases (e.g. `case 20: case 21: ... case 29: print("Twenty something")`).

The lcc compiler can generate a combination of branch trees and possibly short jump tables [FH91, FH95]. Again, it should be possible to recognise this high level pattern in the CFG and recreate close to the original source code.

An older version of the Sun C compiler, for sparse switch statements, compiled in a simple hash function on the switch variable, and the jump table consisted of a pointer and the hash key (switch value). A loop was required to account for collisions. Several different simple hash functions have been observed; the compiler seemed to try several hash functions and to pick the one with the best performance for the given set of switch values. Suitable high level patterns can transform even highly unusual code like this into normal switch statements [CVE01].

6.3 Indirect Calls

Indirect calls implement calls through function pointers and virtual function calls; the latter are a special case which should be handled specially for readability.

Compilers emit indirect call instructions to implement virtual function calls (VFCs), and calls through function pointers (e.g `(*fp)(x)`, where `fp` is a function pointer, and `x` is a passed argument). Similarly to the implementation of switch statements using

jump tables, virtual function calls are usually implemented using a few fixed patterns, with some variation amongst compilers. There is an extra level of indirection with VFCs, because all objects of the same class share the same virtual function table (VFT or VT). The VT is often stored in read-only memory, since it never changes; the objects themselves are usually allocated on the heap or on the stack.

Indirect function calls that do not match the patterns for VFCs can be handled as indirect function pointer calls. In fact, VFT calls could be handled as indirect function pointer calls, but there would be needless and complex detail in the generated code. For example, a call to method 3 of object `*obj` might be generated as `(obj->VT[3])(x)`, where again `x` is the argument to the call, compared to the expected `obj->method3(x)`.

6.3.1 Virtual Function Calls

In most object oriented languages, virtual function calls are implemented with indirect call instructions, which present special challenges to decompilers.

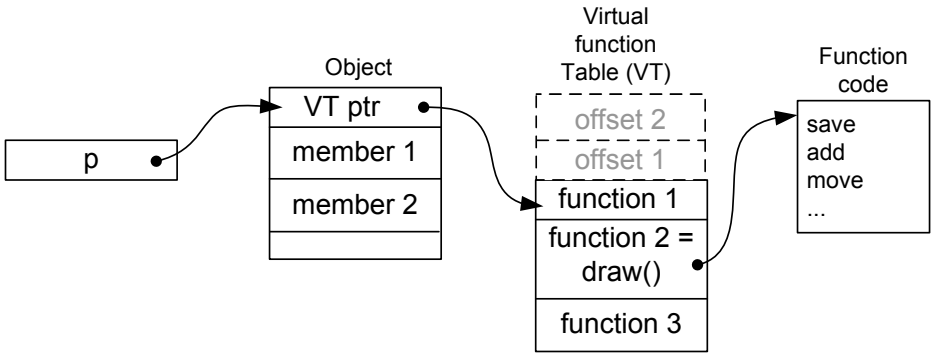


Figure 6.12: Typical data layout of an object ready to make a virtual call such as `p->draw()`.

It is common in object oriented programs to find function calls whose destination depends on the run-time class of the object making the call. In languages like C++, these are called virtual function calls. The usual implementation includes a table of function pointers, called the virtual function table, VFT, VT, or virtual method table, as a hidden member variable. Figure 6.12 shows a typical object layout. The VT pointer is not necessarily the first member variable. Figure 6.13 shows a typical implementation of such a call in x86 assembly language.

Where multiple inheritance is involved, the VT may include information about how to cast a base class pointer to a derived class pointer (a process known as "downcasting"). Such casting usually involves adding an offset to the pointer, but the required constant

```
% eax has pointer to object
mov (%esi),%eax     % Load VT pointer to eax
mov %esi,(%esp)     % Copy ‘‘this’’ to TOS (first parameter)
call *0x4(%eax)     % Call member function at offset 4
```

Figure 6.13: Implementation of a simple virtual function call (no adjustment to the “this” (object) pointer.

often depends on the current (runtime) type of the original pointer. Sometimes this constant is stored at negative offsets from the VT pointer, as shown dotted in figure 6.12. Figure 6.14 shows an implementation.

```
% eax has pointer to object
mov %esi,%ebx       % Copy object to ebx
mov (%esi),%eax     % Load VT pointer to eax
mov -12(%eax),%ecx  % Load offset to ecx
add %ecx,%ebx       % Add offset to object
mov (%ebx),%eax     % Load VT for ancestor object
mov %ebx,(%esp)     % Copy adjusted this as first
parameter
call *(%eax)        % Call member function (offset 0)
```

Figure 6.14: Implementation of a more complex function call, with an adjustment to “this”, and using two virtual function tables (VTs).

Tröger and Cifuentes [TC02] report that static analysis can identify virtual function calls using high level patterns similar to those used for switch statements. It can determine the location the object pointer is read from, the offset to the VT, and the offset to the method pointer. To find the actual method being called, however, requires a run-time value of an object pointer. As an example, the analysis might reveal that the object pointer resides in register `esi` at a particular instruction, that the VT is at offset 0 in the object, and the function pointer is at offset 8 in the VT. To find one of the methods actually being called, the analysis needs to know that the object pointer could take on the value allocated at the call to `malloc` at address 0x80487ab. The authors imply that this analysis is possible only in a dynamic tool (such as a dynamic binary translator), since only in a dynamic tool would such run-time information be available in general.

However, once the object pointer and offset of the VT pointer in the object is found from the above analysis, finding the VT associated with the object is essentially equivalent to performing value analysis (like range analysis but expecting sets of singleton values rather than ranges of values) on the VT pointer member. In the compiler world, this is called type determination. Pande and Ryder [PR94, PR96] prove that this is NP-hard, but provide an approximate polynomial solution. Readability can be improved if all

such object pointer values can be found. Finding all such objects allows examination of all possible targets, eliminating arguments that are not used by any callees, and allowing precise preservation analysis. The latter prevents all live locations from having to be passed as reference parameters (or as parameters and returns) of the call, as discussed in Section 3.4.3 on page 81. Value analysis may be able to take advantage of trees of ϕ-statements in the SSA form, as discussed in Section 6.2.2 on assigned gotos.

Once the arguments and object pointer are determined, the indirect call instruction can be replaced by a suitable high level IR construct. The rest of the code implementing the indirect call will then be eliminated as dead code, leaving only the high level call (e.g. `employee->calculate_pay()`) in the decompiled output, as required.

If the above could be achieved for most object values, then it would be possible to identify most potential targets of virtual calls. This would enable decoding of instructions reachable only via virtual functions, achieving more of the goal of separating code from data.

6.3.1.1 Data Flow and Virtual Function Target Analyses

Use of the SSA form helps considerably with virtual function analysis, which is more complex than switch analysis, by mitigating alias problems, and because SSA relations apply everywhere.

While the techniques of Tröger and Cifuentes [TC02] can be used to find the object pointer, VT pointer offset, and method offset, it does not find the actual VT associated with a given class object. Finding the VT associated with the object is equivalent to finding a class type for the object. In some cases, the VT will even have a pointer to the original class name in the executable [VEW04]. This section considers two analyses for finding the targets of virtual function calls, the first without using the SSA form, and the second using the SSA form.

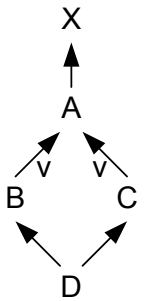


Figure 6.15 shows a C++ program using shared multiple inheritance. The class hierarchy is at left. It uses shared multiple inheritance in the sense that class `A`, which is multiply inherited, is shared by classes `B` and `C` (as opposed to being replicated inside both `B` and `C`). In the source code, the underlined `virtual` keywords make this choice. Machine code for this program is shown in Figure 6.16. Unfortunately, simpler examples do not contain aliases, and hence do not illustrate the point that SSA form has considerable advantages in practical use.

Consider the call at address `804884c`, implementing `c->foo()` in the source code. Suppose first this call is being analysed instruction by instruction without the SSA form or

```
#include <iostream>
class X {
public:
  int x1, x2;
  X(void) { x1=100; x2=101; }
  virtual void foo(void) {
    cout << "X::foo(" << hex << this << ")" << endl; }
};
class A: public X {
public:
  int a1, a2;
  A(void) { a1=1; a2=2; }
  virtual void foo(void) {
    cout << "A::foo(" << hex << this << ")" << endl; }
};
class B: public virtual A {
public:
  int b1, b2;
  B(void) { b1=3; b2=4; }
  virtual void bar(void) {
    cout << "B::bar(" << hex << this << ")" << endl; }
};
class C: public virtual A {
public:
  int c1, c2;
  C(void) { c1=5; c2=6; }
  virtual void foo(void) {
    cout << "C::foo(" << hex << this << ")" << endl; }
};
class D: public B, public C {
public:
  int d1, d2;
  D(void) { d1=7; d2=8; }
  virtual void foo(void) {
    cout << "D::foo(" << hex << this << ")" << endl; }
  virtual void bar(void) {
    cout << "D::bar(" << hex << this << ")" << endl; }
};
int main(int argc, char *argv[]) {
  D* d = new D();    d->foo();
  B* b = (B*) d;     b->bar();
  C* c = (C*) d;     c->foo();
  A* a = (A*) b;     a->foo();
  ....
```

Figure 6.15: Source code for a simple program using shared multiple inheritance.

```
80487a9:  push $0x38
80487ab:  call operator_new           // Allocate 56 bytes
80487b0:  mov  %eax,%esi              // esi points to base
80487b2:  lea  0x24(%esi),%eax        // eax is 36 bytes past base
80487b5:  lea  0x10(%esi),%ebx        // ebx is 16 bytes past base
80487b8:  movl $0x64,0x24(%esi)
80487bf:  movl $0x8049c40,0x8(%eax)
80487c6:  movl $0x65,0x4(%eax)
80487cd:  movl $0x1,0xc(%eax)
80487d4:  movl $0x2,0x10(%eax)
80487db:  movl $0x8049c30,0xc(%esi)
80487e2:  movl $0x3,0x4(%esi)
80487e9:  movl $0x8049c20,0x8(%eax)
80487f0:  movl $0x4,0x8(%esi)
80487f7:  mov  %eax,0x10(%esi)
80487fa:  movl $0x5,0x4(%ebx)
8048801:  movl $0x6,0x8(%ebx)
8048808:  mov  %eax,(%esi)
804880a:  movl $0x8049c00,0xc(%esi)
8048811:  movl $0x7,0x1c(%esi)
8048818:  movl $0x8049c10,0x8(%eax)
804881f:  movl $0x8,0x20(%esi)
8048826:  mov  %eax,(%esp)
8048829:  call *0x8049c18             // d->foo()
804882f:  mov  0xc(%esi),%eax
8048832:  mov  %esi,(%esp)
8048835:  call *0x8(%eax)             // b->bar()
8048838:  xor  %eax,%eax
804883a:  test %esi,%esi
804883c:  sete %al
804883f:  lea  0xffffffff(%eax),%edi
8048842:  and  %ebx,%edi
8048844:  mov  (%edi),%eax
8048846:  mov  0x8(%eax),%edx
8048849:  mov  %eax,(%esp)
804884c:  call *0x8(%edx)             // c->foo()

804884f:  add  $0x10,%esp
8048852:  xor  %ebx,%ebx
8048854:  test %esi,%esi
8048856:  je   804885a
8048858:  mov  (%esi),%ebx
804885a:  sub  $0xc,%esp
804885d:  mov  0x8(%ebx),%eax
8048860:  push %ebx
8048861:  call *0x8(%eax)             // a->foo() (via B)
```

Figure 6.16: Machine code for the start of the `main` function of Figure 6.15.

the benefit of any data flow analysis. By examining the preceeding three instructions, it is readily determined that the object pointer is `eax` at instruction 8048846, the VT pointer offset is 8, and the method offset is also 8.

A simplified algorithm for determining the value of the VT pointer associated with the call is as follows. Throughout the analysis, one or more expression(s) of interest is (are) maintained. Analysis begins with the expression of interest set to the expression for the VT pointer, in this case `m[eax+8]` (with `eax` taking the value it has at instruction 8048846). First, analysis proceeds backwards through the program. For each assignment to a component of an expression of interest (e.g. `eax` is a component of `m[eax+8]`), the right hand side of the assignment is substituted into the component. For example, when the instruction `mov(%edi),%eax` is encountered, the value of `eax` is no longer of interest, since it is overwritten at this instruction, and cannot affect the indirect call from that point back. The new expression of interest is found by replacing the left hand side of the assignment (here `eax`) with the right hand side of the assignment (here `m[edi]`), resulting in a new expression of interest of `m[m[edi]+8]`. For simplicity, consider the `and` instruction at `8048842` to be a move instruction; it implements a null-preserved pointer, discussed below. This phase of the algorithm terminates when the expression of interest would contain the result of a call to the operator new library function, or the start of the procedure is reached.

Now a second phase begins where analysis proceeds forwards from this point. In this phase, when an assignment is reached that produces an affine relation for one of the components, the analysis continues with an extra expression of interest involving new locations. For example, if the expression of interest is ...`m[eax+16]`... and the assignment `esi := eax` is encountered, the new expressions of interest are ...`m[eax+16]`... and ...`m[esi+16]`... . If the assignment was instead `ebx := eax+8`, an expression for `eax` is derived (here `eax := ebx-8`), and this is substituted into the existing expression. In this case, the resulting expressions of interest would be ...`m[eax+16]`... and ...`m[ebx+8]`... . An assignment such as `ebx := eax+8` is called an affine relation because it "marries" the locations `eax` and `ebx` to each other. Such affine related locations, if involved in memory expressions, produce aliases. This phase of the analysis terminates successfully when an expression of interest is a constant, or unsuccessfully if the starting instruction is reached.

Table 6.2 shows this process in detail for the program of Figures 6.15 and 6.16. When the first phase of the analysis winds back to the call to operator new, this is equivalent to `m[m[`*base*`+16]+8]`, were *base* is the start of the memory allocated for the object. However, the analysis ignored instruction `80487f7: mov %eax, 16(%esi)`, because at the time it was not known that `m[ebx]` and `m[esi+16]` were aliases. This is the reason

Table 6.2: Analysis of the program of Figure 6.16.

Instruction considered	Expression(s) of interest
`8048846` (start)	`m[eax+8]`
`8048844 mov (%edi),%eax`	`m[m[edi]+8]`
`8048842 mov %ebx,%edi` (effectively)	`m[m[ebx]+8]`
`8048835 call *0x8(%eax)`	`m[m[ebx]+8]`
`8048829 call *0x8(%eax)`	`m[m[ebx]+8]`
`80487b5 lea 16(%esi),%ebx`	`m[m[esi+16]+8]`
`80487b0 mov %eax,%esi`	`m[m[eax+16]+8]`
`80487ab call operator_new`	`m[m[eax+16]+8]`
`80487b0 mov %eax,%esi`	`m[m[eax+16]+8] m[m[esi+16]+8]`
`80487b2 lea 36(%esi),%eax`	`m[m[esi+16]+8] m[m[eax-20]+8]`
`80487b5 lea 16(%esi),%ebx`	`m[m[esi+16]+8] m[m[eax-20]+8] m[m[ebx]+8]`
`80487f7 mov %eax,16(%esi)`	`m[eax+8]`
`8048818 mov $8049c10,8(%eax)`	`0x8049c10`

for the second phase of the algorithm. Further forward progress reaches instruction `8048818: movl $0x8049c10,8(%eax)`, which gives the address of the VT for this indirect call. However, the instructions at addresses `80487bf` and `80487e9` also match the final expression of interest, and these can only be reached by changing analysis direction yet again. (In this case, they are not needed, as they are overwritten by the assignment at `8048818`, but this may not always be the case.) Virtual calls with the same VT pointer call methods of the same class, thereby giving information about the classes that methods belong to. It is now a simple matter of looking up the VT at offset 8 (the method pointer offset, here coincidentally the same as the VT pointer offset) to find the target of the call.

Note that in this example, there is only one possible target for the call, so the call could have been optimised to a direct call if the compiler had optimised better. The more common case is that there will be several possible objects pointed to, and each of these could have different types, and hence different VT pointer values. The simplified algorithm needs to be extended to take this into account, and since object pointers are commonly passed as parameters to procedures, the analysis needs to be interprocedural as well. Since global objects are created in special initialisation functions that are specific to the executable file format, this code needs to be examined as well.

Note also that for this example, the standard preservations apply (e.g. `esi` and `ebx` are preserved by the calls at 8048829 and 8048835). Alternatively, each call could be processed in sequence, strictly checking each preservation as targets are found. However, this will only strictly be correct if for every call, all possible targets are discovered, since

any undiscovered target might be an exception to the preservation information based only on discovered targets.

This example provides a taste of the complexity caused by aliases, which are common in machine code implementing virtual calls.

```
080487ab 18  eax18 := call operator_new()
080487b0 19  esi19 := eax18
080487b2 20  eax20 := eax18 + 36
080487b5 21  ebx21 := eax18 + 16
080487b8 22  m[eax18 + 36]22 := 100
080487bf 23  m[eax18 + 44]23 := 0x8049c40
080487c6 24  m[eax18 + 40]24 := 101
080487cd 25  m[eax18 + 48]25 := 1
080487d4 26  m[eax18 + 52]26 := 2
080487db 27  m[eax18 + 12]27 := 0x8049c30
080487e2 28  m[eax18 + 4]28 := 3
080487e9 29  m[eax18 + 44]29 := 0x8049c20
080487f0 30  m[eax18 + 8]30 := 4
080487f7 31  m[eax18 + 16]31 := eax18 + 36
080487fa 32  m[eax18 + 20]32 := 5
08048801 33  m[eax18 + 24]33 := 6
08048808 34  m[eax18]34 := eax18 + 36
0804880a 35  m[eax18 + 12]35 := 0x8049c00
08048811 36  m[eax18 + 28]36 := 7
08048818 37  m[eax18 + 44]37 := 0x8049c10
0804881f 38  m[eax18 + 32]38 := 8
08048826 39  m[esp0 - 44]39 := eax18 + 36
08048829 43  eax43 := call __thunk_36_foo__1D()
               Reaching definitions: ... ebx43=eax18+16, esi43=esi19, ...
0804882f 44  eax44 := m[esi43 + 12]?
08048832 45  m[esp43]45 := esi43
08048835 49  eax49 := call m[eax44 + 8]()
               Reaching definitions: ... ebx49=ebx43, esi49=esi43, ...
08048838 50  eax50 := 0
0804883c 54  eax54 := esi49 = 0
0804883f 56  edi56 := (0 | eax54) - 1
08048842 57  edi57 := ((0 | eax54) - 1) & ebx49
08048844 59  eax59 := m[edi57]?
08048846 60  edx60 := m[eax59 + 8]?
08048849 61  m[esp49]61 := eax59
0804884c 65  eax65 := call m[edx60 + 8]()
```

Figure 6.17: IR for the program of Figures 6.15 and 6.16.

Consider now the alternative of waiting until data flow analysis has propagated, canonicalised, and simplified expressions in the IR, and further that the IR is based on the SSA

form, as shown in Figure 6.17. For reasons of alias safety, propagation of memory expressions have not been performed at this stage. (If expression propagation could have been performed with complete alias safety, which may be possible some day, the call at statement 65 would be `eax := call m[0x8049c18]`, which means that no analysis is needed to find the target of the call.

Table 6.3: Analysis of the program of Figure 6.17.

Statement considered	Expression(s) of interest
60 (start)	$m[eax_{59}+8]+8]$
59 $eax_{59} := m[edi_{57}]_?$	$m[m[edi_{57}]+8]$
57 $edi_{57} := ebx_{49}$ (effectively)	$m[m[ebx_{49}]+8]$
49 call ... $ebx_{49}=ebx_{43}$	$m[m[ebx_{43}]+8]$
43 call ... $ebx_{43}=eax_{18} + 16$	$m[m[eax_{18}+16]+8]$
31 $m[eax_{18} + 16]_{31} := eax_{18} + 36$	$m[eax_{18}+44]$
37 $m[eax_{18} + 44]_{37} := 0x8049c10$	0x8049c10

Table 6.3 shows the equivalent analysis with these assumptions. In the SSA form, assignments such as $esi_{19} := eax_{18}$ are valid everywhere that they could appear. Unfortunately, although aliasing is mitigated (witness all the memory expressions that are expressed in terms of eax_{18} in Figure 6.16), aliasing can still exist because of statements such as $eax_{59} := m[edi_{57}]_?$, where the data flow analysis has not been able to determine where $m[edi_{57}]$ has been assigned to. Hence, with SSA form, there is no need for strict direction reversals as with the first algorithm, but sometimes more than one statement has to be considered. Even so, the advantage of the SSA form version is apparent by comparing Tables 6.2 and 6.3.

6.3.1.2 Null-Preserved Pointers

Compilers sometimes emit code that preserves the "nullness" of pointers that are offsets from other pointers, necessitating some special simplification rules.

In Figure 6.16, the instructions at addresses 8048838-8048842 guarantee that a pointer into a memory allocation that may have failed will be NULL if that allocation failed. For example, if the return value from `operator new` at address 80487ab returns NULL, then the value of `esi` at address 804883a will be zero. If so, the `sete` (set if equal) instruction will set register `al` (and hence `eax`) to 1, which will set `edi` to 0 at 804883f. When this is `and`ed with `ebx` at 8048842, the result will be 0 (NULL). Any nonzero value returned from `operator new` will cause the `sete` instruction to assign 0 to `eax`, -1 to `edi`, and after the `and` instruction, `edx` will have a copy of `ebx` (which has the result of the call to `operator new` plus 16).

Without special attention, these constructs result in expressions such as $\texttt{edi}_{57}$ `:= ((0 | (esi`$_{49}$`=0)) - 1) & ebx`$_{49}$. This could be overcome with a simplification such as $((x = 0) - 1)\,\&\,y \rightarrow y$. This construct could occur in the original program's source code, so this simplification rule should be restricted to this sort of analysis.

The code sequence at 8048852-8048856 of Figure 6.16 implements a similar construct for memory references inside the potentially invalid memory block. (The idea seems to be to make sure that any fault should happen in user code, not in compiler generated code). In this case, the simplification $(x = 0)\,?\,0 : y \rightarrow y$ could be used, again restricted to this analysis. Note that the branch in this sequence causes extra basic blocks that would not otherwise be present, and this simplification avoids the complexity of following more than one in-edge for the basic block currently being analysed.

6.3.2 Recovering the Class Hierarchy

Value analysis on the VT pointer member, discussed in earlier sections, allows the comparison of VTs which may give clues about the original class hierarchy.

When decompiling object oriented programs, it is desirable to be able to recreate as closely as possible the class hierarchy of the original source code in the decompiler output. If at a virtual function call the associated object could be of type A or B, then A and B must be related in the class hierarchy of the original program. This is valuable information that comes from performing value analysis on the VT pointer fields of objects associated with VFCs.

If class B is derived from class A, it generally follows that the VT for B has at least the same number of methods as the VT for A. In other words, if the VT for B is a superset of the VT for A, then B extends A, or B is a subtype of A. However, some of the methods of B could override the methods of A, and in addition B may have no new virtual methods that A does not. In fact, B could override all the methods of A, in which case it is not possible to determine from the VTs alone whether B is derived from A or A from B. In this case, the size of the object itself (which indicates the number of data members) could be used to determine which is the superclass. In the case that the objects are of equal size, then in reality it does not matter which is declared as the parent, at least as far as compilable output is concerned.

As mentioned in [VEW04], it is sometimes possible to obtain class hierarchy information by a compiler specific analysis of the executable program.

6.3.3 Function Pointers

Value analysis could be applied to function pointers, which should yield a subset of the set of possible targets, avoiding readability reductions that these pointers would otherwise cause.

Indirect call instructions that do not implement virtual function calls are most likely the result of applying function pointers, e.g. `(*fptr)(a, b)` in the C language.

Value analysis applied to a function pointer would yield the set of possible targets. As mentioned above in relation to virtual function calls, the precise solution to this is an NP-hard problem. Hence, approximate solutions (a subset of all possible targets) are all that can be expected.

As with virtual function pointers, if target information is not available, the possible consequences are missing functions, excess arguments for the pointer indirect calls, and a lack of dead code elimination (since there is no preservation information).

6.3.3.1 Correctness

Imprecision in the list of possible targets for indirect calls leads to one of the few cases where correct output is not possible in general.

Since missing functions obviously imply an incorrect output, the inability to find the complete set of targets for indirect calls is one of the very few reasons why decompiled output cannot be correct in the general case. Recall that for most other limitations of decompilation, it is possible to emit correct but less readable code.

While the missing functions could be found manually or by using dynamic techniques, there must still be at least one error in the code for each missing function, since the original source code must have assigned the address of the function to some variable or data structure. If the pointer analysis was able to identify this value as a function pointer, then the function would not be missing. So somewhere in the decompiled code is the use of a constant (function pointers are usually constants) in an expression, and this constant will be expressed as an integer when it should be the name of a function. When the decompiled output is recompiled, the integer will no longer point to a valid function, and certainly not to the missing function, so the program will not have the semantics of the original program.

In order for this incorrect output to occur, the necessary conditions are a function that is reachable only through one or more function pointers, and the value analysis has to fail to find that value for the function pointer.

6.3.4 Splitting Functions

A newly discovered indirect call, or occasionally a branch used as a call, could end up pointing to the middle of an existing function; this situation can be handled by splitting the function.

When either virtual function analysis or function pointer analysis is successful, a set of possibly new function targets is available. An indirect branch could be used as a tail call optimised call to another function, so it is also possible for an indirect branch to lead to the discovery of a new function. The tail call optimisation replaces a call and return sequence with a branch; this saves time, uses fewer instructions, and most importantly saves stack space. In highly recursive programs, the tail call optimisation can be very significant.

Occasionally, the newly discovered function target could be in the middle of an existing function. Balakrishnan and Reps treat this as an error and merely report a message [BR04]. The function may however be split into two, with a tail call at the end of the first part to preserve the semantics. This will obviously not be practical if there are loops crossing the split point.

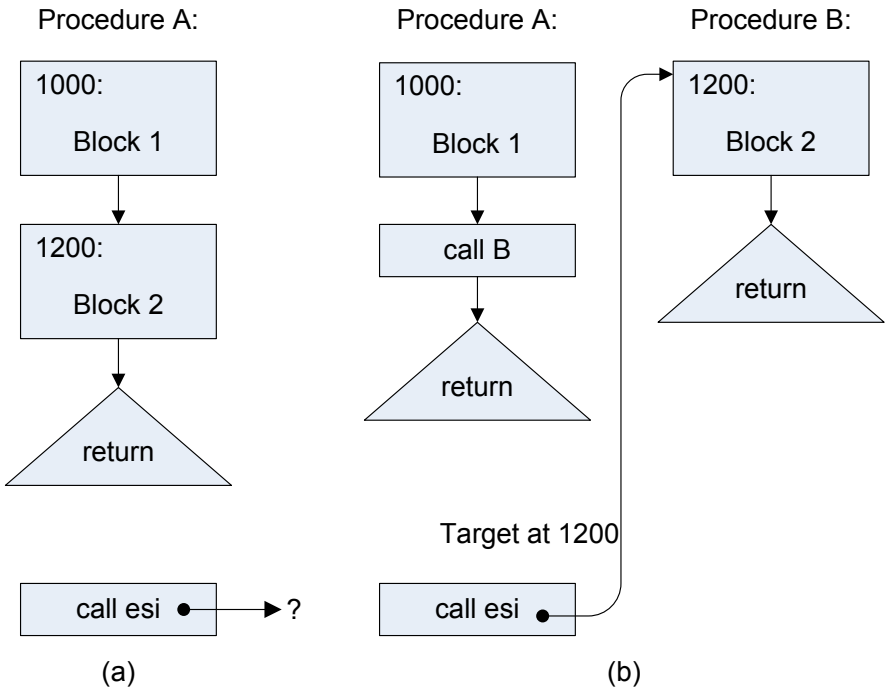


Figure 6.18: Splitting a function due to a newly discovered call target.

Figure 6.18 shows an example of a function being split. In part (a), the indirect call instruction has not yet been analysed, and procedure A consists of block1 and block2. In part (b), a new call target of 1200 has been found, which is in the middle of the

existing procedure A, and only one control flow edge crosses the address 1200. If the address 1200 was in the middle of a basic block, it could be split, as routinely happens during decoding when new branch targets are discovered.

The instructions from address 1200 to the end of the function become a new procedure B as shown. In order that the semantics of procedure A are maintained, a call to B is inserted at the end of block1, and a return statement is inserted after the call. Thus, after executing block 1 in A, block 2 is executed, control returns to the return statement in A, and A returns as normal to its caller. This is effectively the reverse of the tail call optimisation.

6.4 Related Work

De Sutter et al. observe a similar problem to the phase ordering problem of Section 6.1 when constructing the ICFG (Interprocedural Control Flow Graph) from sets of object files [SBB+00]. They solve it by using a so-called hell node in the ICFG, which unknown control flows from indirect jumps or calls lead to. They construct control flow edges also from the hell node to all possible successors of indirect jumps or calls. For their domain, where relocation information is available, "all possible" successors is a reasonably bounded set. However, for decompilation, relocation information is generally not available, so the hell node approach is not suitable.

Tröger and Cifuentes implemented a way of analysing virtual function calls to find the object pointer, virtual table pointer offset, and method offset [TC02]. They relied on dynamic techniques to find actual values for the object pointer. They used a simple algorithm that was limited to the basic block the the virtual call is in, and did not have the advantages of expression propagation. Expression simplification was used, but only a few special cases were considered.

Vinciguerra et al. surveyed the various techniques available to disassemblers for finding the code in an executable program [VWK+03]. The same techniques are used in the decoder component of a machine code decompiler. The problem of indirect jumps and calls is cited as the "problem which has dominated much of the work on C/C++ disassembly tools". Slicing and "data flow guided" techniques are mentioned, but not with the advantages of expression propagation. The authors also mentioned that the effect of limitations in this process produce, at best, a less readable but correct program.

Reps et al. perform security analysis on x86 executables [BR04]. To cope with indirect jumps and calls, they use a combination of the heuristics in IDA Pro and their Value Set Analysis (VSA). VSA has already been mentioned as being needed for the solution

of several problems in type analysis and the analysis of indirect jumps and calls. They do not split functions when a new target is in the middle of an existing function, but presumably this could be done using IDA Pro's API.

Harris and Miller describe a set of analyses that split a machine code executable into functions [HM05]. They claim that compiler independent techniques are not necessary, because in practice "simple machine and compiler dependent methods have proved effective at recovering jump table values". They do not appear to analyse indirect calls, relying instead on various heuristics to find procedures in the gaps between regions reachable with recursive traversal.

Chapter 7

Results

Several techniques introduced in earlier chapters were verified with a real decompiler, and show that good results are possible with their use.

Many of the analyses described in earlier chapters have been tested on the Boomerang decompiler [Boo02]. Simpler machine code decompilers, such as REC, can be relied on to produce passable output on programs of almost any size, but Boomerang specialises on correct, recompilable output for a wide range of small programs. Boomerang's test suite consists of some 75 small programs, of which about a dozen fail for various reasons.

The results in this chapter relate to material in earlier chapters. Section 7.1 describes an industry case study which confirms the limitations of current decompilers stated in Chapter 1. Chapter 3 showed how useful expression propagation is for decompilation, but how unlimited propagation can in some circumstances lead to poor readability. Section 7.2 shows that common subexpression elimination is not the answer, but a simple heuristic is quite effective. The usefulness of the SSA form is demonstrated in Chapter 4, although in certain circumstances involving overwriting statements in a loop, extraneous variables can be emitted. Section 7.3 gives the results of applying Algorithm 1 to an example where many extraneous variables were being generated.

Section 4.3.2 showed how a preserved location could be a parameter; Section 7.4 presents an example and results. Preservation analysis is shown in detail in Section 7.5. In the presence of recursion, preservation analysis is much more complex, as discussed in Section 4.4.2. Section 7.5.1 gives detailed results of applying the theory of Section 4.4.2 to its example. Recursion also complicates the elimination of redundant parameters and returns, as Section 4.4.3 indicated. Section 7.6 presents an example which shows good results.

7.1 Industry Case Study

When a Windows program was decompiled for a client, the deficiencies of existing decompilers were confirmed, and the importance of recovering structures was highlighted.

Chapter 1 discussed the problems faced by machine code decompilers, and Chapter 2 reviewed the limitations of existing decompilers. This section confirms the limitations of existing decompilers in an industrial setting.

The author of this thesis and a colleague were asked to attempt a partial decompilation of a 670KB Windows program [VEW04]. Source code was available, but it was for a considerably earlier version of the program. The clients were aware of the limitations of machine code decompilation.

While there were a few surprising results, in general the case study confirmed the findings of Chapters 1 and 2. At the beginning of the study, Boomerang was broadly comparable in capabilities with other existing machine code decompilers. Type analysis was *ad hoc*, and structures were not supported. The parameter recovery logic assumed that all parameters were passed on the stack, which was often not the case in the C++ Windows executable to be decompiled.

As a result of this study, the salient problems facing machine code decompilers were confirmed to be

- weak or non-existent type analysis;
- lack of support for structures and compound types in general;
- incomplete recovery of parameters and returns; and
- poor handling of indirect control transfers, particularly indirect call instructions.

One speculation from the paper ([VEW04]) has since been overturned. Section 5.2 of the paper, referring to the problem of excessive propagation, states that "Some form of common subexpression elimination could solve this problem". As shown below, this is not the case, and an alternative solution is given.

7.2 Limiting Expression Propagation

Common Subexpression Elimination does not solve the problem of excessive expression

propagation; the solution lies with limiting propagation of complex expressions to more than one destination.

Section 3.2 on page 68 states that Common Subexpression Elimination (CSE) does not solve the problem of excessive propagation, but preventing propagation of complex expressions to more than one destination does. Results follow that verify this statement.

Figure 7.1 shows part of the sample output of Figure 8 in [VEW04]. It shows two cases where excessive propagation has made the output less readable than it could be: the `if` statement of lines 18-21, and the `if` statement of line 24.

```
13      if ((*(char*)(local11 + 68) & 1) != 0) {
14        local26 = (/* opTruncs/u */ (int) (sizePixelsPerTick_cx *
15          -2004318071 >> 32) + sizePixelsPerTick_cx >> 4) + (/* opTruncs/u */
16          (int) (sizePixelsPerTick_cx * -2004318071 >> 32) +
17          sizePixelsPerTick_cx >> 4) / -2147483648;
18        if ((/* opTruncs/u */ (int) (sizePixelsPerTick_cx *
19          -2004318071 >> 32) + sizePixelsPerTick_cx >> 4) + (/* opTruncs/u */
20          (int) (sizePixelsPerTick_cx * -2004318071 >> 32) +
21          sizePixelsPerTick_cx >> 4) / -2147483648 < 2) {
22            local26 = 2;
23        }
24        if (local26 >= maxTickSizeX - (maxTickSizeX < 0 ?  -1 :  0) >> 1) {
25          local26 = (maxTickSizeX - (maxTickSizeX < 0 ?  -1 :  0) >> 1) - 1;
26        }
```

Figure 7.1: Output from [VEW04], Figure 8.

Value numbering was used to generate a table of subexpressions whose values were available on the right hand sides of assignments. Because of the simple nature of machine instructions, almost all subexpressions are available at some point in the program. CSE was performed before any dead code was eliminated, so all propagated expressions still had their originating assignments available. A C++ map was used in place of the usual hash table, to avoid issues arising from collisions. Statements were processed in order from top to bottom of the procedure. Figure 7.2 shows the results of applying CSE to the code of Figure 7.1.

Line 10 is now simplified as desired (it corresponds to lines 18-21 of Figure 7.1). Similarly, lines 16 and 17 are simplified (corresponding to lines 24 and 25 respectively of Figure 7.1). However, there are now far too many local variables, and the semantics of individual instructions is again evident. What is desired for ease of reading is propagation where possible, except in cases where an already complex expression (as measured e.g. by the number of subexpressions) would be propagated to more than one destination. In other words, it is better to prevent excessive propagation, rather than trying to repair excessive propagation with CSE.

```
1  tmp = *(int*)(local64 + 56) & 1;
2  if (tmp != 0) {
3    tmp = local84 * 0x88888889;
4    local143 = (unsigned int) tmp >> 32;
5    local143 += local84;
6    local83 = local143 >> 4;
7    local144 = local83 / 0x80000000;
8    local143 = local83 + local144;
9    local142 = local83 + local144;
10   if (local143 < 2) {
11     local142 = 2;
12   }
13   local143 = maxTickSizeX < 0 ?  -1 :  0;
14   local144 = maxTickSizeX - local143;
15   local144 = local144 >> 1;
16   if (local142 >= local144) {
17     local142 = local144 - 1;
18   }
```

Figure 7.2: Common subexpression elimination applied to the same code as Figure 7.1.

The propagation algorithm of Boomerang has been updated as follows. Before each propagation pass, a map is now created of expressions that could be propagated to. The map records a count of each unique expression that could be propagated to (an SSA subscripted location). In the main expression propagation pass, before each expression is propagated from the right hand side of an assignment to a use, this map is consulted. If the number recorded in the map is greater than 1, and if the complexity of the expression to be propagated is more than a given threshold (set with a command line parameter), the expression is not propagated. The complexity is estimated as the

```
1  if ((*(int*)(local42 + 56) & 1) != 0) {
2    local80 = (unsigned int) (local46 * 0x88888889 >> 32) + local46 >> 4;
3    local80 += local80 / 0x80000000;
4    local79 = local80;
5    if (local80 < 2) {
6      local79 = 2;
7    }
8    local81 = maxTickSizeX;
9    local81 = local81 - (local81 < 0 ?  -1 :  0) >> 1;
10   if (local79 >= local81) {
11     local79 = local81 - 1;
12   }
```

Figure 7.3: Propagation limited to below complexity 2, applied to the same code as Figure 7.1.

```
1 if ((*(int*)(local45 + 56) & 1) != 0) {
2   local83 = (unsigned int) local49 * 0x88888889 >> 32;
3   local83 = (local83 + local49 >> 4) + (local83 + local49 >> 4)/0x80000000;
4   local82 = local83;
5   if (local83 < 2) {
6     local82 = 2;
7   }
8   local84 = maxTickSizeX - (maxTickSizeX < 0 ?  -1 :  0);
9   if (local82 >= local84 >> 1) {
10    local82 = (local84 >> 1) - 1;
11  }
```

Figure 7.4: Propagation limited to below complexity 3, applied to the same code as Figure 7.1.

```
1 if ((*(int*)(local44 + 56) & 1) != 0) {
2   local82 = (unsigned int) (local48 * 0x88888889 >> 32) + local48;
3   local82 = (local82 >> 4) + (local82 >> 4) / 0x80000000;
4   local81 = local82;
5   if (local82 < 2) {
6     local81 = 2;
7   }
8   local83 = maxTickSizeX - (maxTickSizeX < 0 ?  -1 :  0) >> 1;
9   if (local81 >= local83) {
10    local81 = local83 - 1;
11  }
```

Figure 7.5: Propagation limited to below complexity 4, applied to the same code as Figure 7.1.

number of operators in the expression (binary and ternary operators, and the "memory of" operator).

Performing this limited expression propagation on the same code from Figure 7.1 results in the output shown in Figures 7.3 - 7.5. The difference between these is that the first prevents propagation of expressions of complexity 2 or more, the second with expressions of complexity 3 or more, and the last with expressions of complexity 4 or more. The `if` statements are again simplified, but complex expressions are retained for the other statements, for maximum readability. Although the number of lines of code only reduces from 14 to 11 or 12, the readability of the code is considerably improved. Readability was measured with three metrics: the character count (excluding multiple spaces, newlines, and comments), the Halstead difficulty metric, and the Halstead program length metric [Hal77]. The Halstead program length metric is the sum of the number of operators and operands in the program.

Metrics based on control flow complexity (e.g. McCabe Cyclomatic Complexity [McC76]) will not find any difference due to expression propagation or the lack thereof, since the

control flow graph is not affected by propagation. Table 7.1 shows the results for the selected metrics. The Halstead metrics were calculated with a public domain program [Uti92].

Table 7.1: Complexity metrics for the code in Figures 7.1 - 7.5.

	Character count	Halstead Difficulty	Halstead Length
Unlimited propagation (Figure 7.1)	595	28.3	142
CSE (Figure 7.2)	464	28.1	116
Limited propagation (limit = 2, Figure 7.5)	329	22.8	93
Limited propagation (limit = 3, Figure 7.5)	357	23.8	111
Limited propagation (limit = 4, Figure 7.5)	342	22.7	107

Varying the expression depth limit causes minor variations in the output, as can be seen by inspecting the decompiled output of Figures 7.3 - 7.5. The metrics vary in an unpredictable way with this limit, because of a kind of interference effect. For example, preventing a propagation at one statement will lead to an intermediate statement of low complexity, but this statement may now be able to be propagated to a later statement. If the early propagation succeeded, the later propagation may have failed. Whether the overall metrics improve or not ends up depending on chance factors. While the effect of the propagation limit is not very large (the Halstead Difficulty varies from 22.7 to 28.3, a 25% difference), it does suggest that propagation could be one area where user input will be required for the most readable result (according to the tastes of the reader).

The low value for the Halstead length metric in the CSE case is because of the extreme re-use of each subexpression; nothing is calculated more than once. The fact that there are many local variables and assignments is not reflected in this metric; the left hand sides of each assignment are ignored. This metric is therefore not measuring the distraction and difficulty of dealing with a needlessly large number of local variables, and hence this metric does not show the dramatic difference that either the character count or Halstead Difficulty show for limited propagation.

As a further example of excessive propagation, consider the program fragment in Figure 7.6. As shown in Figure 7.7, the compiler uses an idiomatic sequence involving the subtract with carry instruction. Output from the Boomerang decompiler without expression limiting is shown in Figure 7.8. Boomerang has special code to recognise the use of the carry flag in this idiom, but not to emit the conditional assignments that the idiom represents. The result is completely incomprehensible code, but it is at least correct. Figure 7.9 shows the same code with expression propagation limiting in effect. While still difficult to comprehend, the result is much better laid out.

```
int test(int i) {
  if (i < -2) i = -2;
  if (i > 3) i = 3;
  printf("MinMax result %d\n", i);
}
```

Figure 7.6: Original C source code for function `test` in the Boomerang SPARC `minmax2` test program. The code was compiled with Sun's compiler using `-xO2` optimisation.

```
mov -0x2, %g1
mov -0x1, %g4
sra %o0, 0x1f, %o5
subcc %g1, %o0, %g2
subx %g4, %o5, %g4
and %g2, %g4, %g2
sethi %hi(0x10400), %g3
sub %g1, %g2, %g1
add %g3, 0x34c, %o0
sra %g1, 0x1f, %g2
subcc %g1, 0x3, %g1
subx %g2, 0x0, %g2
and %g1, %g2, %g1
add %g1, 0x3, %o1
mov %o7, %g1
call printf
mov %g1, %o7
```

Figure 7.7: Disassembled SPARC machine code for the program fragment of Figure 7.6. Note the `subcc` (subtract and set condition codes) and `subx` (subtract extended, i.e. with carry) instructions.

```
void test(int param1) {
  printf("MinMax result %d\n",
    (-5 - (-2 - param1 & -1 - (param1 >> 31) +
    (0xfffffffe < (unsigned)param1)) &
    (-2 - (-2 - param1 & -1 - (param1 >> 31) +
    (0xfffffffe < (unsigned)param1)) >> 31) -
    ((unsigned)(-2 - (-2 - param1 & -1 - (param1 >> 31) +
    (0xfffffffe < (unsigned)param1))) < 3)) + 3);
  return;
}
```

Figure 7.8: Decompiled output for the program fragment of Figure 7.6, without expression propagation limiting.

```
void test(int param1) {
  int local0; // r4
  int local1; // r1
  int local2; // r2
  local0 = -1 - (param1 >> 31) + (0xfffffffe < (unsigned)param1);
  local1 = -2 - (-2 - param1 & local0);
  local2 = (local1 >> 31) - ((unsigned)local1 < 3);
  printf("MinMax result %d\n", (local1 - 3 & local2) + 3);
  return;
}
```

Figure 7.9: Decompiled output for the program fragment of Figure 7.6, but with expression propagation limiting.

7.3 Preventing Extraneous Local Variables

When the techniques of Section 4.1.3 are applied to the running example, the generated code is significantly more readable.

Figure 4.11 on page 111 showed output from an early version of Boomerang which had no logic for limiting extraneous variables. It is reproduced in Figure 7.10 with the local variables renamed as far as possible the same names as the original registers. Variable `st` represents the stack top register of the x86 processor. This is done to facilitate comparison with later output, which does this automatically. There are 10 statements inside the loop.

```
do {
   edx_1 = edx;
   ebx_1 = ebx;
   esi_1 = esi;
   st_1 = st;
   edx_2 = edx_1 - 1;
   st = st_1 * (float)edx_1 / (float)esi_1;
   esi_2 = esi_1 - 1;
   ebx = ebx_1 - 1;
   edx = edx_2;
   esi = esi_2;
} while (ebx_1 >= 1)
st_2 = (int)(st_1 * (float)edx_1 / (float)esi_1);
```

Figure 7.10: A copy of the output of Figure 4.11 with local variables named after the registers they originated from.

Figure 7.11(a) shows the output from a more recent version of Boomerang, which incorporates the expression limiting heuristic (with expression depth limited to 3), and makes better choices about which version of variable to allocate to a new variable. Already, it is a vast improvement. In this version, there are 5 statements inside the loop.

Figure 7.11(b) shows output from Boomerang using the `-X` (experimental) flag. This flag uses Algorithm 1 in Section 4.1.3 to minimise extraneous variables by not propagating overwriting statements in a loop. A side effect is that the assignment to `st` is split onto two lines, so there are still five statements in the loop, but no extraneous variables. With support for the C post decrement operator (e.g. `edx--`), output similar to the original source code would be possible.

As of April 2007, the Boomerang source code implementing the algorithm of Section 4.1.3 is flawed, using the invalid concept of "dominance numbers". It will fail in some

```
do {
   edx_1 = edx;
   edx = edx_1 - 1;
   st = st * (float)edx_1 / (float)esi;
   esi = esi - 1;
   ebx = ebx - 1;
} while (ebx >= 0);
edx = (int)st;
```

(a) Standard switches

```
do {
   st = st * (float)edx;
   edx = edx - 1;
   st = st / (float)esi;
   esi = esi - 1;
   ebx = ebx - 1;
} while (ebx >= 0);
edx = (int)st;
```

(b) With dominator heuristics

Figure 7.11: The code of Figure 4.11 with limited propagation.

cases, although the flaw does not affect the example.

7.4 Preserved Parameters

Preserved locations appear to be parameters, when usually they are not, but sometimes a preserved location can be a parameter. Propagation and dead code elimination in combination solve this problem.

Figure 7.12 shows the x86 assembly language for a simple function that takes one parameter and returns a result, when the parameter is a register that is saved, used as a parameter, overwritten, and restored.

```
# parameter is passed in register %ebx
twice:
   push %ebx        # save ebx
   mov  %ebx, %eax  # copy parameter to return register, %eax
   addl %eax, %eax  # double eax
   mov  $55, %ebx   # usually preserved registers are overwritten
   pop  %ebx        # restore ebx
   ret
```

Figure 7.12: Assembly language source code for part of the Boomerang `restoredparam` test.

Figure 7.13 shows the intermediate representation for the program, after expression propagation, but before dead code elimination. Note how the save instruction becomes dead code, the restore instruction becomes a null statement, and the use of the parameter in statement 3 has no definition (hence the zero subscript).

After dead code elimination, the only statement remaining is the return statement. The decompiled output is correct, as shown in Figure 7.14.

```
 1 esp := esp0 - 4              // Decrement the stack pointer
 2 m[esp0-4] := ebx0            // Save ebx (result no longer used)
 3 eax := ebx0                  // Copy parameter to result register
 4 tmp1 := ebx0
 5 eax := ebx0 + ebx0           // Double result (also in return statement)
 6 %flags := ADDFLAGS32( ... )
 7 ebx := 55                    // Overwrite ebx (not used)
 8 ebx := ebx0                  // Restore ebx (not used)
 9 esp := esp0
10 %pc := m[esp0]0
11 esp := esp0 + 4
12 RET eax := ebx0 + ebx0       // Return result
```

Figure 7.13: Intermediate representation for the code of Figure 7.12, just before dead code elimination.

```
int twice(int param1) {
    return param1 +
param1;
}
```

Figure 7.14: Boomerang output for the code of Figure 7.12. The parameter and return are identified correctly.

Figure 7.15 shows the source code for a Fibonacci function, and Figure 7.16 shows the disassembly of an MS-DOS compilation of it. It is a modified version of a test program for the *dcc* decompiler [Cif94]. Where the original program saved the *SI* register with a `PUSH` instruction and then loaded the parameter from the stack, this version passes the parameter in *SI* yet still `PUSH`es and `POP`s the register. The register *SI* is therefore both preserved and a parameter. The resultant program still runs on a 32-bit personal computer, despite the age of the instruction set (from 1978), and performs the same operation as the original program in fewer instructions (not counting `NOP`s). It is therefore a valid program that machine code decompilers should be able to generate valid source code for.

```
unsigned fib(x) /* compute fibonacci number recursively */
int x;
{
    if (x > 2)
        return (fib(x - 1) + fib(x - 2));
    else
        return (1);
}
```

Figure 7.15: Original source code for a Fibonacci function. From [Cif94].

```
035B 55     PUSH BP
035C 8BEC   MOV BP,SP
035E 56     PUSH SI
035F 909090 NOP         ; Was MOV SI,[BP+4]
0362 83FE02 CMP SI,+02
0365 7E1C   JLE 0383
0367 4E     DEC SI      ; Was MOV AX,SI
0368 9090   NOP         ; Was DEC AX
036A 90     NOP         ; Was PUSH AX
036B E8EDFF CALL 035B
036E 5E     POP SI      ; Was POP CX; get original parameter
036F 56     PUSH SI
0370 50     PUSH AX
0371 83C6FE ADD SI,-02  ; Was MOV AX,SI
0374 90     NOP         ; Was ADD AX,-2
0375 90     NOP         ; Was PUSH AX
0376 E8E2FF CALL 035B
0379 90     NOP         ; Was POP CX
037A 8BD0   MOV DX,AX
037C 58     POP AX
037D 03C2   ADD AX,DX
037F EB07   JMP 0388
0381 EB05   JMP 0388
0383 B80100 MOV AX,0001
0386 EB00   JMP 0388
0388 5E     POP SI
0389 5D     POP BP
038A C3     RET
```

Figure 7.16: Disassembly of the modified Fibonacci program adapted from [Cif94].

As shown in Figures 7.17 and 7.18(a), the *dcc* and REC decompilers do not produce valid source code. Both decompilers do not identify any parameters, although REC passes two arguments to one call and none to the other. Both emit invalid C code for `POP` instructions.

The Boomerang decompiler uses propagation and dead code elimination to identify the preserved parameter. The result of decompiling the 32-bit equivalent[1] of the program of Figure 7.16 is shown in Figure 7.18(b). This program also demonstrates the removal of unused parameters and returns, in the presence of recursion.

[1]The 32-bit version was not completely equivalent. The 32-bit version originated from the source code of Figure 7.19. The two minor differences cause fib(0) to return 0, as is usually considered correct.

```
int proc_1 ()
/* Takes no parameters.
 * High-level language prologue code.
 */
{
    int loc1;
    int loc2; /* ax */
    if (loc1 > 2) {
        loc1 = (loc1 - 1);
        POP loc1
        loc1 = (loc1 + 0xFFFE);  /* 0xFFFE = -2 */
       loc2 = (proc_1 () + proc_1 ());
    }    else { ...
```

Figure 7.17: Output from the *dcc* decompiler for the program of Figure 7.16.

```
L0000025b()
{
    /* unknown */ void si;
    if(si <= 2) {
        ax = 1;
    } else {
        si = si - 1;
        (restore)si;
        si = si - 2;
        dx = L0000025B(
          L0000025B(), si);
        (restore)ax;
        ax = ax + dx;
    }
}
```

(a) REC

```
int fib(int param1) {
   __size32 eax;
   __size32 eax_1; // eax30
   int local2;     // m[esp - 12]
   if (param1 <= 1) {
      local2 = param1;
   } else {
      eax = fib(param1 - 1);
      eax_1 = fib(param1 - 2);
      local2 = eax + eax_1;
   }
   return local2;
}
```

(b) Boomerang (32-bit equivalent)

Figure 7.18: Output from the REC and Boomerang decompilers for the program of Figure 7.16 and its 32-bit equivalent respectively.

7.5 Preservation

Most components of the preservation process are facilitated by the SSA form.

The Boomerang machine code decompiler has an equation solver. It was added when it was found necessary to determine whether registers are preserved or not in the presence of recursion. Figure 7.21 shows the output of Boomerang's solver, finding that `esi` (an x86 register) is preserved in the recursive Fibonacci function of Figures 7.19 (source code) and 7.20 (IR).

```
int fib (int x)
{
   if (x > 1)
      return (fib(x - 1) + fib(x - 2));
   else return (x);
}
```

Figure 7.19: Source code for a slightly different Fibonacci function.

It begins with the premise $\texttt{esi}_{35}$ = $\texttt{esi}_0$ (i.e. esi as last defined is the same as esi on entry; the return statement contains information about which locations reach the exit). Various rules are applied to the current equation: propagation into the left hand side, adding constants to both sides, using the commutation property of equality (swap left and right sides), and so on. For example on the second line, the LHS is $\texttt{esi}_{35}$ (esi as defined at line 35). In Boomerang, the subscript on esi is actually a pointer to the statement at line 35, so again the propagation is very easy. The RHS has esi_0, and the proof engine actually searches for statements (such as statement 5) which save the value of $\texttt{esi}_0$. On the fifth line, m[] (the memory-of operator) is removed from both sides (i.e. to prove that m[a] = m[b], it is sufficient to show that a = b). When the left hand side could be propagated to by a ϕ-function, then the current equation has to be proved for each operand. In the example, the current equation becomes $\texttt{ebp}_{46}$ = $(\texttt{esp}_4\texttt{+4})$. Statement 46 is a ϕ-function (46: ebp := $\phi(\texttt{ebp}_{26}, \texttt{ebp}_3)$). In effect, this expression represents two separate equations to be solved: $\texttt{ebp}_{26}$ = $(\texttt{esp}_4\texttt{+4})$, and $\texttt{ebp}_3$ = $(\texttt{esp}_4\texttt{+4})$. (Control flow could be such that ebp could be defined at statement 26, or at statement 3, so it is necessary to prove both subproblems in order to prove the premise, i.e. that esi is preserved through the whole function.) Each of these equations is proved separately, thereby proving the premise.

Without the SSA form, it would be necessary to separate various versions of each location, such as "esi on entry", "esi at the procedure exit", and "esi after statement 26".

7.5.1 Conditional Preservation Analysis

Preservation analysis in the presence of mutual recursion is complex, as this example shows.

A test program was written to exercise the conditional preservation analysis of Section 4.4.2 on page 130. Figure 7.22 shows the call graph for the program, deliberately designed to copy Figure 4.26 on page 129. An outline of the test program's source code is given in Figure 7.23.

```
08048798  1 m[esp_0 - 4] := ebp_0
          2 esp := esp_0 - 4
08048799  3 ebp := esp_0 - 4
0804879b  4 m[esp_0 - 8] := esi_0
          5 esp := esp_0 - 8
0804879c  6 m[esp_0 - 12] := ebx_0
          7 esp := esp_0 - 12
0804879d  8 ebx := m[esp_0 + 4]
080487a0  9 tmp1 := ebx_8 - 1
080487a3 11 BRANCH 0x80487c0, ebx_8 <= 1
080487a5 12 eax := ebx_8 - 1
080487a8 13 m[esp_0 - 16] := ebx_8 - 1
         14 esp := esp_0 - 16
080487a9 15 esp := esp_0 - 20
         18 <all> := CALL fib(<all>)
080487ae 19 esi := eax_18
080487b0 20 eax := ebx_18 - 2
080487b3 21 m[esp_0 - 20] := ebx_18 - 2
         22 esp := esp_0 - 20
080487b4 23 esp := esp_0 - 24
         26 <all> := CALL fib(<all>)
080487b9 27 tmp1 := eax_26
         28 eax := eax_26 + esi_26
080487bb 30 GOTO 0x80487c2
080487c0 31 eax := ebx_8
00000000 43 eax := φ(eax_28, eax_31)
         44 ebx := φ(ebx_26, ebx_8)
         45 esp := φ(esp_26, esp_7)
         46 ebp := φ(ebp_26, ebp_3)
         47 esi := φ(esi_26, esi_0)
         48 tmp1 := φ(tmp1_27, tmp1_9)
080487c2 32 esp := ebp_46 - 8
080487c5 33 ebx := m[ebp_46 - 8]
         34 esp := ebp_46 - 4
080487c6 35 esi := m[ebp_46 - 4]
         36 esp := ebp_46
080487c7 37 esp := ebp_46
         38 ebp := m[ebp_46]
         39 esp := ebp_46 + 4
080487c8 41 esp := ebp_46 + 8
         42 RET eax := eax_43
```

Figure 7.20: IR for the Fibonacci function of Figure 7.19.

The global variable *res* is used to ensure that each procedure is called the correct number of times. Each procedure increments *res* by a prime number, so that if the program outputs the correct total, the control flow is very likely correct. Global variables with

attempting to prove esi is preserved by fib
$esi_{35} = esi_0$
$m[esp_{34}] = esi_0$
$m[esp_{34}] = m[esp_4]$
$esp_{34} = esp_4$
$(esp_{32} + 4) = esp_4$
$esp_{32} = (esp_4 + -4)$
$esp_{32} = (esp_4 - 4)$
$(ebp_{46} - 8) = (esp_4 - 4)$
$ebp_{49} = ((esp_4 - 4) + 8)$
$ebp_{49} = (esp_4 + 4)$
found $\phi(ebp_{26}, ebp_3)$ prove for each
  proving for $ebp_{26} = (esp_4 + 4)$
  $ebp_{26} = (esp_4 + 4)$
   using proven (or induction) for fib ebp = ebp
  $ebp_{18} = (esp_4 + 4)$
   using proven (or induction) for fib ebp = ebp
  $ebp_3 = (esp_4 + 4)$
  $esp_1 = (esp_4 + 4)$
  $(esp_0 - 4) = (esp_4 + 4)$
  $esp_0 = ((esp_4 + 4) + 4)$
  $esp_0 = (esp_4 + 8)$
  $(esp_4 + 8) = esp_0$
  $esp_4 = (esp_0 + -8)$
  $esp_4 = (esp_0 - 8)$
  $(esp_1 - 4) = (esp_0 - 8)$
  $esp_1 = ((esp_0 - 8) + 4)$
  $esp_1 = (esp_0 - 4)$
  $(esp_0 - 4) = (esp_0 - 4)$
  true
  proving for $ebp_3 = (esp_4 + 4)$
  $ebp_3 = (esp_4 + 4)$
  $esp_1 = (esp_4 + 4)$
  $(esp_0 - 4) = (esp_4 + 4)$
  $esp_0 = ((esp_4 + 4) + 4)$
  $esp_0 = (esp_4 + 8)$
  $(esp_4 + 8) = esp_0$
  $esp_4 = (esp_0 + -8)$
  $esp_4 = (esp_0 - 8)$
  $(esp_1 - 4) = (esp_0 - 8)$
  $esp_1 = ((esp_0 - 8) + 4)$
  $esp_1 = (esp_0 - 4)$
  $(esp_0 - 4) = (esp_0 - 4)$
  true
true

The problem
The premise
Propagate into the left hand side (LHS)
Use `5: m[esp`$_4$`]:=esi`$_0$ on RHS
Remove `m[]` from both sides
Propagate into LHS
Subtract 4 from both sides
a + -b → a - b
Propagate into LHS again
Add 8 to both sides
Simplify
Note `46: ebp :=` ϕ`(ebp`$_{26}$`, ebp`$_3$`)`
First ϕ-function operand
New premise
We have earlier proved that ebp is preserved
Therefore can propagate across call
There is another recursive call
Propagate across this call
Propagate into LHS
Propagate into LHS
Add 4 to both sides
Simplify
Commute
Subtract 8 from both sides
a + -b → a - b
Substitute LHS
Add 4 to both sides
Simplify
Substitute LHS
LHS = RHS!
Second ϕ-function operand
New premise
Substitute LHS
Substitute LHS
Add 4 to both sides
Simplify
Commute
Subtract 8 from both sides
a + -b → a - b
Substitute LHS
Add 4 to both sides
Simplify
Substitute LHS
LHS = RHS
Now proved for each ϕ-function operand

Figure 7.21: Debug output from Boomerang while finding that `esi` (register `esi`) is preserved (saved and restored).

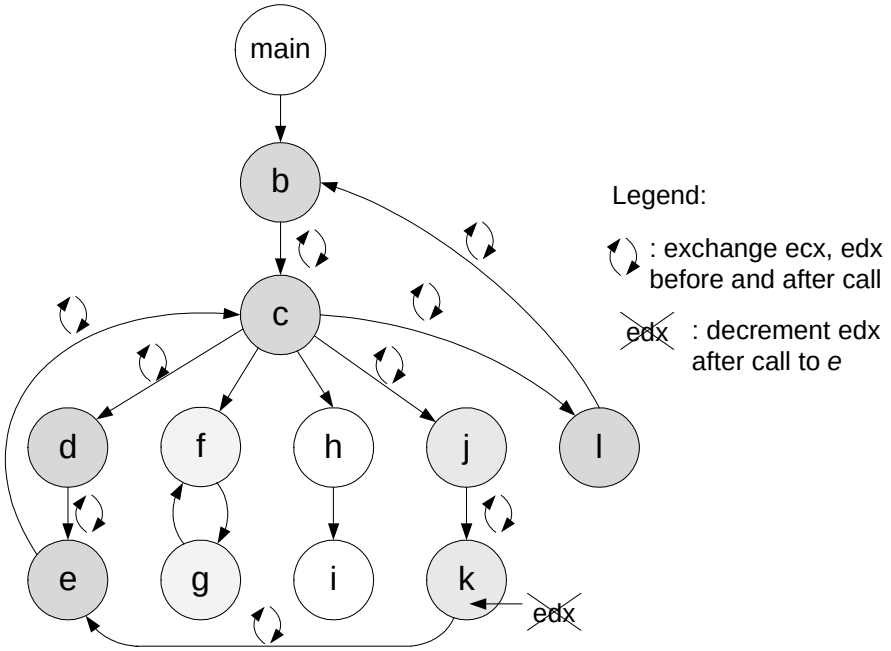


Figure 7.22: Call graph for the Boomerang test program `test/pentium/recursion2`.

```
/* Global affected as a side effect from all calls */
int res = 0;
/* Number of times to traverse b to c edge */
int b_c = 3;
int c_d = 3;
...
int l_b = 3;
int main(int argc) {
  b();
  printf("ecx is %d, edx is %d\n", 0, 0);
  printf("res is %d\n", res);
  return 0;
}
void b() {
  if (--b_c >= 0) c();
  res += 2;
}
```

```
void c() {
  if (--c_d >= 0) d();
  if (--c_f >= 0) f();
  if (--c_h >= 0) h();
  if (--c_j >= 0) j();
  if (--c_l >= 0) l();
  res += 3;
}
...
void k() {
  if (--k_e >= 0) e();
  res += 27;
}
void l() {
  if (--l_b >= 0) b();
  res += 29;
}
```

Figure 7.23: An outline of the source code for the program `test/pentium/recursion2` from the Boomerang test suite.

names of the form $x\_\ y$ are used to control the recursion from procedure x to procedure y. All such globals are initialised to 3; the program outputs "res is 533". To make this test more difficult for the conditional preservation analysis, there are instructions to exchange the contents of machine registers *ecx* and *edx* at the points indicated on

Figure 7.22, and a single instruction to decrement register *edx* is placed in procedure *k* after the call to *e*. This results in *edx* changing value, but also register *ecx*, since there is a path in the call graph where there is an odd number of exchange instructions (*c*-*d*-*e*-*c*).

The first cycle detected is *f*-*g*-*f*, when *g*'s child *f* is found in *path*. The first preservation to succeed is `esp = esp+4`. As with most x86 procedures, the stack pointer for *g* is preserved in the base pointer register (`ebp`). Because of the control flow join at the end of *g*, there is a ϕ-statement defining the final value of `ebp`, so the proof has to succeed for both paths. One path is through the call to f, so the preservation of `ebp` in g depends on the preservation of `ebp` in *f*. *f* has a similar structure to g, so again there is a ϕ-statement, the proof has to succeed for both paths, and now the preservation for `ebp` in *f* depends on the preservation of `ebp` in *g*. Note that the algorithm has now come almost full circle: `esp` in *g* depends on `ebp` in *g*, `ebp` in *g* depends on `ebp` in *f*, `ebp` in *f* depends on `ebp` in *g*. Since preservation that depends only on recursive calls can be assumed to succeed, `ebp` is indeed assumed to succeed in *g* (note: this is not yet proven, it is a *conditional* result). No other problems are found with this preservation, so finally `esp = esp+4` is proven. Note that the intermediate conditional results (`ebp=ebp` in *g* and *f*) are not stored, since at the time when the proof function exits, it was not known if the outer proof would succeed or fail. This is illustrated in the next example.

The final example is the preservation `ecx=ecx` in *b*. Recall that there are instructions to exchange the value of these registers before and after calls in the *c*-*j*-*k*-*e*-*c*, *b*-*c*-*l*, and *c*-*d*-*e*-*c* cycles. Due to these, preservation of `ecx` in b depends on `edx` in *c*, which depends on `ecx` in *d*, which depends on `edx` in *e*, which depends on `ecx` in *c*. Note that while the algorithm is considering procedure *c* again, it requires a different preservation this time, so the process continues. `ecx` in *c* depends on `edx` in *d*, which depends on `ecx` in *e*, which depends on `edx` in *c*. Finally, there is a required premise that is already being tested. As a result, the preservation of `edx` in c can be conditionally assumed. As will be shown soon, this does not turn out to be true. In similar vein, the preservations `ecx` in *e* and `edx` in *d* are assumed to be conditionally proven. However, c has other calls, so `ecx` in *c*, which depended on `edx` in *d* and has conditionally passed, now depends on `edx` in *j*, which depends on `ecx` in *k*, which depends on `edx` in *e*. This is another premise, which is conditionally assumed to succeed, and similarly `edx` for *j*. There is still one more call in *c*, to *l*, so `ecx` in *c* also depends on `edx` in *l*. This depends on `ecx` in *b*, the original premise, and so conditionally succeeds. Finally, `ecx` in *c* succeeds conditionally, leading to conditional successes for `edx` in *e* and `ecx` in *d*. The algorithm is now considering `edx` in c, which depends on ecx in j, and in turn `edx` in k. This depends on `ecx` in *e*, which has been calculated and conditionally proven before, but

is calculated anew. After the call to k is the decrement of edx, so one critical path of the whole proof finally fails, since edx_0 = edx_0 - 1 cannot be proven. As a result, ecx is assumed to be assigned to in the call to c, ecx remains a parameter and return of b, c, and all the other procedures involved in the cycle. Similarly, edx is found not to be preserved, and so remains a parameter and return of those functions also.

```
//__size32 b(__size32 param1, __size32 param2) {
pair       b(__size32 param1, __size32 param2) {
   __size32 ecx;
   __size32 edx_1;   // edx22
   pair pr;
   b_c = b_c - 1;
   if (b_c >= 0) {
      //ecx = c(param2, param1); /* Warning:  also results in edx_1 */
      pr    = c(param2, param1);
      ecx = pr.first; edx_1 = pr.second;
      param2 = ecx;
      param1 = edx_1;
   }
   res += 2;
   //return param1; /* WARNING: Also returning:  edx := param2 */
   pr.first = param1; pr.second = param2;
   return pr;
}
```

Figure 7.24: The code generated for procedure b for the program test/pentium/recursion2. The Boomerang -X option was used to remove extraneous variables, as discussed in Section 7.3. The code has been modified by hand (underlined code) to return more than one location.

Figure 7.24 shows an outline of the code generated for procedure b. Since Boomerang did not at the time handle more than one return from a procedure, some hand editing of the output was necessary. The final program compiled and ran identically to the original.

7.6 Redundant Parameters and Returns

The techniques of Section 4.4.3 successfully remove redundant parameters and returns in a test program, improving the readability of the generated code.

Figure 7.25 shows assembly language code for the program of Figure 7.19. A few instructions were modified so that registers ecx and edx were used and defined in the function. Without the techniques of Section 4.4.3 on page 134, and assuming that

```
fib:
    pushl %ebp
    movl  %esp, %ebp
    pushl %ebx
    subl  $4, %esp
    cmpl  $1, 8(%ebp)
    jle   .L2
    movl  8(%ebp), %eax
    decl  %eax
    subl  $12, %esp
    pushl %eax
    call  fib
    addl  $16, %esp
    push  %edx            # Align stack, and make a ...
    push  %edx            # ...  use of %edx (but it is dead code)
    push  %eax            # Save intermediate result
    movl  8(%ebp), %eax
    subl  $2, %eax
    pushl %eax
    call  fib
    pop   %edx            # Remove argument, assign to edx
    pop   %ecx            # Get intermediate result, assign to ecx
    addl  $8, %esp
    addl  %eax, %ecx      # Add the two intermediate results
    movl  %ecx, -8(%ebp)
    jmp   .L4
.L2:
    movl  8(%ebp), %eax
    movl  %eax, -8(%ebp)
.L4:
    movl  -8(%ebp), %eax
    movl  -4(%ebp), %ebx
    leave
    ret
```

Figure 7.25: Assembler source code for a modification of the Fibonacci program shown in Figure 7.19. The underlined instructions assign values to registers `ecx` and `edx`.

`push` and `pop` instructions are not simply ignored, these two registers become extra parameters and returns of the `fib` function, as shown by the IR in Figure 7.26.

Although the assignment to `edx` at statement 44 is never actually used, it appears to be used by the ϕ-function in statement 68, which is in turn used by the return statement. The `edx` return from fib is used to pass an argument to the call at statement 43, and the edx parameter is used by the ϕ-function in statement 68.

```
void fib(int param1_0, int ecx_0, /*signed?*/int edx_0)
00000000 0 param1_0 := -
         0 ecx_0 := -
         0 edx_0 := -
080483bb 11 BRANCH 0x80483e8, condition param1_0 <= 1
080483c5 24 {eax_24, ecx_24, edx_24, esp_24} := CALL
fib(param1_0-1, ecx_0, edx_0)
080483d7 43 {eax_43, esp_43} := CALL fib(param1_0-2, ecx_24, edx_24)
080483dc 44 edx_44 := param1_0 - 2
080483e1 52 ecx_52 := eax_24 + eax_43
080483e3 54 m[esp_0-12] := eax_24 + eax_43
080483e6 55 GOTO 0x80483ee
080483eb 57 m[esp_0-12] := param1_0
00000000 67 ecx_67 := φ(ecx_52, ecx_0)
         68 edx_68 := φ(edx_44, edx_0)
         78 m[esp_0-12]_78 := φ(m[esp_0-12]_54, m[esp_0-12]_57)
080483f5 65 RET eax := m[esp_0-12]_78, ecx := ecx_67, edx := edx_68
```

Figure 7.26: IR part way through the decompilation of the program of Figure 7.25.

Similar comments apply to the assignment to ecx at statement 52, although in this case, that assignment is gainfully used. Along the path where $\texttt{param1}_0 > 1$, ecx is defined before use, and along the path where $\texttt{param1}_0 \leq 1$, it is only used in the ϕ-function in statement 67, which is only ultimately used to pass a redundant parameter to `fib`.

The techniques of Section 4.4.3 were able to determine that `ecx` and `edx` were redundant parameters and returns, resulting in the generated code of Figure 7.27.

```
int fib(int param1) {
   int eax;
   int eax_1;     // eax_30
   int local2;    // m[esp - 12]
   if (param1 <= 1) {
      local2 = param1;
   } else {
      eax = fib(param1 - 1);
      eax_1 = fib(param1 - 2);
      local2 = eax + eax_1;
   }
   return local2;
}
```

Figure 7.27: Generated code for the program of Figures 7.25 and 7.26. Redundant parameters and returns have been removed.

Chapter 8

Conclusion

8.1 Conclusion

The solutions to several problems with existing machine code decompilers are facilitated by the use of the SSA form.

The main problems for existing machine code decompilers were found to be the identification of parameters and returns, type analysis, and the handling of indirect jumps and calls.

The IR used by a decompiler has a profound influence on the ease with which certain operations can be performed. In particular, the propagation of registers is greatly facilitated by the SSA form, and propagation is fundamental to transforming the semantics of many individual instructions into the complex expressions typical of source code. In fact, unlimited propagation leads to expressions that are too complex, but appropriate heuristics can produce expressions of appropriate complexity.

Propagating assignments of condition codes into their uses (e.g. conditional branches) is a special case, where the combination of condition code and branch (or other) condition results in specific semantics.

Once expressions are propagated, the original definitions become dead code, and can be eliminated. This leads to the second fundamental transformation of machine code to high level language: reducing the size of the generated output. The established techniques of control flow structuring introduce high level constructs such as conditional statements and loops.

The next major decompilation step is the accurate recovery of call statements: arguments and parameters, returns and results. One of the main inputs to this process is preservation analysis: deciding whether a particular register or memory location is

modified by the target of a call, or is preserved. In the main, this is solved by standard data flow analysis, but the combination of indirect calls and recursion causes problems, since the results of some calls will not be known when needed. An algorithm for solving this problem is given.

While the SSA form makes the propagation of registers very easy, it is not safe to propagate all memory expressions because of the possibility of aliasing. Compilers can sidestep this issue by never moving load and store operations, although some will attempt to move them when it can be proved to be safe. The cost of not moving a memory operation is small for a compiler; a possible optimisation opportunity is lost. For decompilers, the cost of not being able to propagate memory expressions is a considerable loss of readability. In particular, all stack local variables, including arrays, should be converted to variables in the decompiled output. Without this conversion, the stack pointer will be visible in the decompiled output, and the output will be complex and may not be portable. A solution to the problem is given, but the solution ignores some aliasing possibilities.

Few if any existing decompilers attempt type recovery (type analysis) to any serious degree. This thesis outlines a type analysis implemented with an iterative, data flow based framework. The SSA form provides convenient, sparse type information storage with each definition and constant. A problem strongly related to type analysis is the partitioning of the data sections (global, local, and heap allocated) into objects of distinct types. This problem is made more complex by the widespread practice of colocating variables in the same register or memory location. Compilers use a surprisingly large number of different memory expression patterns to access objects of elementary types, and the elements of aggregate types. These patterns are explored in detail, but the resolution of some of the more complex patterns remains for future work. In particular, the interpretation of the constant `K` associated with many memory expressions is complicated by factors such as offset pointers and arrays with nonzero lower bounds.

The usual way to handle indirect jump and call instructions is to perform a pattern-based local search near the instruction being analysed, usually within the basic block that the indirect instruction terminates. This search is performed at decode time, before any more powerful techniques are available. The logic is that without resolving the indirect jump or call, the control flow graph is incomplete, and therefore the later analyses will be incorrect. While this is true, it is possible to delay analysis of these indirect instructions until the more powerful analyses are available. The analysis of the whole procedure has to be restarted after the analysis of any indirect branches, incorporating the new indirect branch targets. This step may be repeated several times if the indirect branches are nested. The same may be required after indirect calls. The

power of expression propagation, facilitated by the SSA form, can be used to improve the analysis, e.g. it may well succeed even when several basic blocks have to be analysed. The resolution of VFT pointers is shown to be simpler using this technique also.

Many of the techniques introduced in this thesis have been verified by a working machine code decompiler.

8.2 Summary of Contributions

This thesis advances the state of the art of machine code decompilation through several key contributions.

The first two chapters established the limitations of current machine code decompilers. Table 1.2 compared the fundamental problems of various tools related to machine code decompilers. Figure 1.7 identified where the losses and separations that cause these problems arise.

Chapter 3 demonstrated significant advantages for the SSA form as a decompiler IR, by simplifying expression propagation, recovery of parameters and returns, the analysis of indirect branches and calls, and enabling a sparse data flow based type analysis.

The SSA implementation of a pointer from each use to its definition saves having to rename the original definition in many cases. Compilers as well as decompilers could make use of this contribution.

The power of expression propagation and importance to the overall decompilation process was shown, along with the necessity of limiting some propagations.

In a machine code decompiler, summarising the effects of calls is important. Section 3.4 defined appropriate terminology and equations for deriving the required summary information.

While compilers can combine translation out of SSA form with register allocation, in decompilers, it is important to minimise the number of generated variables. Sections 4.1.2 and 4.1.3 showed how this can be done.

Although the SSA form makes it easy to propagate registers, the propagation of memory expressions is more difficult because of the possibility of aliasing. It is important to propagate as many classes of memory expressions as is safe, to improve the readability of the generated code. Section 4.2 identified the issues and gave a solution.

Unless it is assumed that all procedures follow the ABI specification, preservation analysis is important for correct data flow information. Section 4.3 defined the issues and showed how the SSA form simplifies the problem. Section 4.4 showed how recursion

causes extra problems, and gave a solution for preservation analysis in the presence of recursion.

Chapter 5 gave the first application of an iterative, data flow based type analysis for machine code decompilers. In such a system, addition and subtraction are not readily handled, but Section 5.7.4 provided a solution.

The memory expression patterns representing accesses to aggregate elements are surprisingly complex. These were explored in detail in Section 5.8.

Chapter 6 showed how the analysis of indirect jumps and calls can be delayed until more powerful analyses are available, and the implications of working with an incomplete control flow graph.

Section 6.2.2 showed how Fortran style assigned gotos can be analysed and represented in a language such as C.

In Section 6.3.1, it was shown that less work is involved in the identification of VFT pointers when using an IR such as the SSA form.

8.3 Future Work

While the state of the art of decompilation has been extended by the techniques described in this thesis, work remains to optimise an SSA-based IR for decompilers that handles aggregates well, and supports alias and value analyses.

At the end of Chapter 3 it was stated that an IR similar to the SSA form, but which stores and factors def-use as well as use-def information, could be useful for decompilation; the dependence flow graph (DFG) may be one such IR.

In several places, the need for value (range) analysis has been mentioned. This would help with function pointers and their associated indirect call instructions, interpreting the meaning of `K` in complex type patterns, and would allow the declaration of initial values for arrays accessed with running pointers.

Alias analysis is relatively common in compilers, and would be very beneficial for decompilers. However, working from source code (including from assembly language) has one huge advantage: when a memory location has its address taken, it is immediately obvious from the source code. This makes escape analysis practical; it is possible to say which variables have their addresses escape the current procedure, and it is safe to assume that no other variables have their addresses escape. At the machine code level, however, it is not obvious whether an expression represents the address of an object, and even if this is deduced by analysis, it is not always clear which object's address is

taken. Hence, for alias and/or escape analysis to be practical for machine code decompilation, it probably needs to be combined with value analysis. These analyses would also help with the problem of separating colocated variables.

One area that emerged from the research that remains for future work is the possibility of partitioning the decompilation problem so that parts can be run in parallel threads or on multiple machines. Most of the work of decompiling procedures not involved in a recursion group is independent of the decompilation of other procedures. It seems likely that following this stage, interprocedural analyses may be more difficult to partition effectively.

For convenience, a standard stack pointer register is usually assumed. In many architectures it is possible to use almost any general purpose register as a stack pointer, and a few architectures do not specialise one register at all as the stack pointer. In these cases, the ABI will specify a stack pointer, but it will be a convention not mandated by the architecture. Programs compiled with a nonstandard stack pointer register are likely to produce essentially meaningless output, since the stack pointer is so fundamental to most machine code. The ability to be "stack pointer register agile" could therefore be a significant design challenge, left for future work.

With the beginnings of the ability to decompile C++ programs comes the question of how to remove automatically generated code, such as constructors, destructors, and their calls. Presumably, these functions could be recognised from the sorts of things that they do, and their calls suppressed. The fallback is simply to declare them as ordinary functions, and to not remove the calls. The user or a post decompilation program could remove them if desired. Other compiler generated code such as null pointer checking, debug code such as initialising stack variables to 0xCCCCCCCC and the like, are potentially more difficult to remove automatically, since they could have been part of the original source code.

There is also the question of what to do with exception handling code. Most of the time, very little exception specific code is visible in the compiled code by following normal control flow; often there are only assignments to a special stack variable representing the current exception zone of the procedure. These assignments appear to be dead code. Finding the actual exception handling code is compiler specific, and hence will require support from the front end.

Structuring of control flow has been considered to be a solved problem. However, it is possible that some unusual cases, such as compound conditionals (e.g. `p && q`) in loop predicates, may require some research to handle effectively.

In decompilation, the issue of phase ordering (what order to apply transformations) is very significant; almost everything seems to be needed before everything else. There

may be some research opportunities in designing decompiler components in such a way that these problems can be minimised.

As of 2007, there is an open source decompiler (Boomerang) that is capable of decompiling small machine code programs into readable, recompilable source code. More than one source architecture is supported. The goal of a general machine code decompiler has been shown to be within reach, although much work remains.

Bibliography

[AC72] F. Allen and J. Cocke. Graph theoretic constructs for program control flow analysis. Technical Report RC 3923 (17789), IBM T.J. Watson Research Center, 1972.

[ADH+01] G. Aigner, A. Diwan, D. Heine, M. Lam, D. Moore, B. Murphy, and C. Sapuntzakis. An overview of the SUIF2 compiler infrastructure. Technical report, Computer Systems Laboratory, Stanford University, 2001. Retrieved Apr 2007 from `http://suif.stanford.edu/suif/suif2/doc-2.2.0-4`.

[AF00] A. Appel and A. Felty. A semantic model of types and machine instructions for proof-carrying code. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 2000.

[All70] F. Allen. Control flow analysis. *SIGPLAN Notices*, 5(7):1–19, July 1970.

[Ana01] Anakrino web page, 2001. Retrieved Mar 2007 from `http://www.saurik.com/net/exemplar`.

[And04] Andromeda decompiler web page, 2004. Retrieved Mar 2007 from `http://shulgaaa.at.tut.by`.

[App02] A. Appel. *Modern compiler implementation in Java*, chapter 19. Cambridge University Press, 2002.

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*, chapter 10. Adison-Wesley, 1986.

[Ata92] Atari Games Corp. v. Nintendo, 1992. 975 F.2d 832 (Fd. Cir. 1992).

[Bal98] T. Ball. Reverse engineering the twelve days of christmas, 23rd December 1998. Retrieved June 2006 from `http://research.microsoft.com/~tball/papers/XmasGift/final.html`.

[Bar74] P. Barbe. The Piler system of computer program translation. Technical report, Probe Consultants Inc., September 1974. See also [VE03].

[Bar98] R. Barták. On-line guide to constraint programming, 1998. Retrieved Mar 2004 from `http://kti.ms.mff.cuni.cz/~bartak/constraints/`.

[BCHS98] P. Briggs, K. Cooper, T. Harvey, and L. Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software – Practice and Experience*, 28(8):859–881, July 1998.

[BDB00] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transperent dynamic optimization system. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 2000.

[BDMP05] L. Bigum, H. Davidsen, L. Mikkelsen, and E. Pedersen. Detection and avoidance of ϕ-loops, 2005. Project report, Aalborg University. Retrieved Mar 2006 from `http://www.cs.aau.dk/library/cgi-bin/detail.cgi?id=1117527263`.

[BM00] I. Baxter and M. Mehlich. Reverse engineering is reverse forward engineering. *Science of Computer Programming*, 36:131–147, 2000.

[Boo02] Boomerang web page. BSD licensed software, 2002. Retrieved May 2005 from `http://boomerang.sourceforge.net`.

[BR04] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Proceedings of the International Conference on Compiler Construction*, pages 5–23. Springer, April 2004. LNCS 2985.

[BR05] G. Balakrishnan and T. Reps. Recovery of variables and heap structure in x86 executables. Technical Report #1533, University of Wisconsin Madison, July 2005.

[BR06] G. Balakrishnan and T. Reps. Recency-abstraction for heap-allocated storage. In *13th Annual Static Analysis Symposium*, pages 221–239. Springer, 2006. LNCS 4136.

[Bri81] D. Brinkley. Intercomputer transportation of assembly language software through decompilation. Technical report, Naval Underwater Systems Center, October 1981.

[BRMT05] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. WYSINWYX: What You See Is Not What You eXecute. In *Proceedings of Verified Software: Theory, Tools and Experiments*, 2005.

[BS93] J. Bowen and V. Stavridou. Safety-critical systems formal methods and standards. *IEE/BCS Software Engineering Journal*, 8(4):176–187, July 1993.

[BT01] BinaryTranslation wiki page, 2001. Retrieved May 2005 from `http://www.program-transformation.org/Transform/BinaryTranslation`.

[Byr92] E. Byrne. A conceptual foundation for software re-engineering. In *Proceedings of the International Conference on Software Maintenance*, pages 226–235. IEEE-CS Press, 1992.

[CAI01] Canadian Association for Interoperable Systems submission on copyright issues, 2001. Retrieved Mar 2002 from `http://strategis.ic.gc.ca/SSG/rp00326e.html`.

[Cap98] G. Caprino. REC - Reverse Engineering Compiler. Binaries free for any use, 1998. Retrieved Jan 2003 from `http://www.backerstreet.com/rec/rec.htm`.

[CC76] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the 2nd International Symposium on Programming*, pages 106–130, Paris, France, 1976.

[CC90] E. Chikofsky and J. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7:13–17, January 1990.

[CCF91] J. Choi, R. Cryton, and J. Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages*, pages 55–66. ACM Press, 1991.

[CCL+96] F. Chow, S. Chan, S. Liu, R. Lo, and M. Streich. Effective representation of aliases and indirect memory operations in SSA form. In *Proceedings of the International Conference on Compiler Construction*, pages 253–267. Springer, April 1996. LNCS 1060.

[CDC+04] R. Chowdhury, P. Djeu, B. Cahoon, J. Burrill, and K. McKinley. The limits of alias analysis for scalar optimizations. In *Compiler Construction*, pages 24–38. Springer, 2004. LNCS 2985.

[CFR+91] R. Cryton, J. Ferrante, B. Rosen, M. Wegman, and F. Zadek. Efficiently computing static single assignment form and the control dependence graph.

ACM Transactions on Programming Languages and Systems, 13(4):451–490, October 1991.

[CG95] C. Cifuentes and K.J. Gough. Decompilation of binary programs. *Software – Practice and Experience*, 25(7):811–829, 1995.

[Cha82] G. Chaitin. Register allocation and spilling via graph coloring. *SIGPLAN Notices*, 17(6):98–105, June 1982. Proceedings of the ACM SIGPLAN Symposium on Compiler Construction.

[Cif94] C. Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, School of Computing Science, July 1994. Also available Apr 2007 from `http://www.itee.uq.edu.au/~cristina/dcc/decompilation_thesis.ps.gz`.

[Cif96] C. Cifuentes. The dcc decompiler. GPL licensed software, 1996. Retrieved Mar 2002 from `http://www.itee.uq.edu.au/~cristina/dcc.html`.

[Cif01] C. Cifuentes. Reverse engineering and the computing profession. *Computer*, 34(12):136–138, Dec 2001.

[CJ03] M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th USENIX Security Symposium*, pages 169–186. Published online at `http://www.usenix.org`, August 2003.

[CK94] B. Cmelik and D. Keppel. Shade: A fast instruction-set simulator for execution profiling. In *ACM SIGMETRICS conference on Measurement and Modelling of Computer Systems*, volume 22. ACM Press, May 1994.

[Com95] Australian Copyright Law Review Committee. Report on computer software protection, 1995. Retrieved Mar 2002 from `http://www.law.gov.au/clrc`, Chapter 10, Exceptions to Exclusive Right.

[Cou99] P. Cousot. Directions for research in approximate system analysis. *ACM Computing Surveys*, 31(3es), September 1999.

[CTL97] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, The University of Auckland, 1997.

[CVE00] C. Cifuentes and M. Van Emmerik. UQBT: Adaptable binary translation at low cost. *Computer*, 33(3):60–66, March 2000.

[CVE01] C. Cifuentes and M. Van Emmerik. Recovery of jump table case statements from binary code. *Science of Computer Programming*, 40:171–188, 2001.

[CVEU+99] C. Cifuentes, M. Van Emmerik, D. Ung, D. Simon, and T. Waddington. Preliminary experiences with the use of the UQBT binary translation framework. In *Proceedings of the Workshop on Binary Translation, Newport Beach 16th Oct 1999*, pages 12–22. Technical Committee on Computer Architecture Newsletter, IEEE-CS Press, Dec, 1999.

[CWVE01] C. Cifuentes, T. Waddington, and M. Van Emmerik. Computer security analysis through decompilation and high-level debugging. In *Proceedings of the Working Conference on Reverse Engineering*, pages 375–380, Stuttgart, Germany, 2001. IEEE-CS Press.

[Dat98] DataRescue. IDA Pro, 1998. Retrieved Jan 2003 from `http://www.datarescue.com/idabase`.

[Dec01] DeCompilation wiki page, 2001. Retrieved May 2005 from `http://www.program-transformation.org/Transform/DeCompilation`.

[Dot04] Dot4, Inc. *Assembler to "C" Software Migration Tool*, 2004. Retrieved Jan 2004 from `http://www.dot4.com/literature`.

[DP02] B. Davey and H. Priestley. *Introduction to Lattices and Order*. Cambridge Press, 2002. ISBN 0521784514.

[DS97] A. Dolzmann and T. Sturm. Simplification of quantifier-free formulae over ordered fields. *J. Symbolic Computation*, 24:209–231, 1997.

[DT04] Decompiler Technologies web page, 2004. Retrieved Mar 2007 from `http://www.decompiler.org`.

[Ega98] G. Egan. Short story "The Planck Dive", February 1998. Retrieved Nov 2005 from `http://gregegan.customer.netspace.net.au/PLANCK/Complete/Planck.html`.

[Eri02] D. Eriksson. Desquirr web page, 2002. Retrieved Jul 2005 from `http://desquirr.sourceforge.net/desquirr`.

[EST01] Essential systems technical consulting, 2001. Retrieved Feb 2003 from `http://www.essential-systems.com/resource/index.htm`.

[Fal04] R. Falke. Entwicklung eines Typanalysesystem für einen Decompiler (Development of a type analysis system for a decompiler), 2004. Diploma thesis, in German; retrieved Jan 2005 from `http://risimo.net/diplom.ps`.

[FC99] L. Freeman and C. Cifuentes. An industry perspective on decompilation. In *Proceedings of the International Conference on Software Maintenance*. IEEE-CS Press, 1999. Only the abstract is printed form; full text published at `http://doi.ieeecomputersociety.org/10.1109/ICSM.1999.10009`.

[FC00] A. Fitzgerald and C. Cifuentes, editors. *Going Digital 2000*, chapter 3, pages 37–70. Prospect Media Pty, Sydney, Australia, 2nd edition, February 2000. Pegging out the Boundaries of Computer Software Copyright: The Computer Programs Act and the Digital Agenda Bill.

[Fer95] M. Fernández. Simple and effective link-time optimization of Modula-3 programs. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 103–115. ACM Press, 1995.

[FH91] C. Fraser and D. Hanson. A retargetable compiler for ANSI C. *SIGPLAN Notices*, 26(10):29–43, October 1991.

[FH95] C. Fraser and D. Hanson. *A Retargetable C Compiler: Desigm and Implementation*. Addison-Wesley, Boston, MA, USA, 1995.

[FLI00] IDA Pro FLIRT web page, 2000. Retrieved Mar 2007 from `http://www.datarescue.com/idabase/flirt.htm`.

[Fri74] F. Friedman. *Decompilation and the Transfer of Mini-Computer Operating Systems*. PhD dissertation, Purdue University, Computer Science, August 1974.

[FSF01] Free Software Foundation, Boston, USA. *GNU Binutils*, 2001. Retrieved Apr 2007 from `http://www.gnu.org/software/binutils`.

[FZ91] C. Fuan and L. Zongtian. C function recognition technique and its implementation in 8086 C decompiling system. *Mini-Micro Systems*, 12(11):33–40,47, 1991. Chinese language.

[FZL93] C. Fuan, L. Zongtian, and L. Li. Design and implementation techniques of the 8086 C decompiling system. *Mini-Micro Systems*, 14(4):10–18,31, 1993. Chinese language.

[Gab00] H. Gabow. Path-based depth-first search for strong and biconnected components. *Inf. Process. Lett.*, 74(3-4):107–114, 2000.

[Gai65] R. Stockton Gaines. On the translation of machine language programs. *Comm. ACM*, 8(12):736–741, December 1965.

[GBT+05] B. Guo, M. Bridges, S. Triantafyllis, G. Ottoni, E. Raman, and D. August. Practical and accurate low-level pointer analysis. In *Proceedings of the International Symposium on Code Generation and Opmimization*, pages 291–302. IEEE Press, 2005.

[GHM00] E. Gagnon, L. Hendren, and G. Marceau. Efficient inference of static types for Java bytecode. In *Static Analysis Symposium*, pages 199–219. Springer, June 2000. LNCS 1824.

[Gla98] P. Glasscock. An 80x86 to C reverse compiler. Technical report, University of Cambridge Computer Laboratory Library, 1998. DIP98/14.

[Gra97] GrammaTech Inc, 1997. Retrieved Nov 2004 from `http://www.grammatech.com`.

[Gro05] Metro-Goldwyn-Mayer Studios Inc. et al. v. Grokster, Ltd., et al., 2005. 545 U.S. 125 S.Ct. 2764, 2770 (2005), retrieved Nov 2005 from `http://news.bbc.co.uk/1/shared/bsp/hi/pdfs/supreme_court_mgm_grokster_27_06_05.pdf`.

[Gui01] I. Guilfanov. A simple type system for program reengineering. In *Proceedings of the Working Conference on Reverse Engineering*, pages 357–361, Stuttgart, Germany, 2001. IEEE-CS Press.

[Gui07a] I. Guilfanov. Blog: Decompilation gets real, April 2007. Retrieved Apr 2007 from `http://hexblog.com/2007/04/decompilation_gets_real.html`.

[Gui07b] I. Guilfanov. Hex-rays home page, 2007. Retrieved Aug 2007 from `http://www.hex-rays.com`.

[GY04] J. Gross and J. Yellen, editors. *Handbook of Graph Theory*, chapter 10. CRC Press, 2004.

[Hal62] M. Halstead. *Machine-independent computer programming*, chapter 11, pages 143–150. Spartan Books, 1962. See also `http://www.program-transformation.org/Transform/NeliacDecompiler`.

[Hal70] M. Halstead. Using the computer for program conversion. In *Datamation*, pages 125–129, May 1970.

[Hal77] M. Halstead. *Elements of Software Science, Operating, and Programming Systems Series*, volume 7. Elsevier, 1977.

[Har97] B. Harlan. A tirade against the cult of performance, 1997. Retrieved Aug 2007 from `http://billharlan.com/pub/papers/A_Tirade_Against_the_Cult_of_Performance.html`.

[HH74] B. Housel and M. Halstead. A methodology for machine code decompilation. In *Proceedings of the 1974 annual conference (ACM'74)*, pages 254–260. ACM Press, 1974.

[HM79] R. Horspool and N. Marovac. An approach to the problem of detranslation of computer programs. *The Computer Journal*, 23(3):223–229, 1979.

[HM05] L. Harris and B. Miller. Practical analysis of stripped binary code. *ACM SIGARCH Computer Architecture News*, 33(5):63–68, December 2005.

[Hoe00] J. Hoenicke. Java Optimize and Decompile Environment, 2000. Retrieved Jan 2003 from `http://sourceforge.net/projects/jode`.

[Hol73] C. Hollander. *Decompilation of Object Programs*. PhD dissertation, Stanford University, Computer Science, January 1973.

[Hop78] G. Hopwood. *Decompilation*. PhD dissertation, University of California, Irvine, Computer Science, 1978.

[Hou73] B. Housel. *A Study of Decompiling Machine Languages into High-Level Machine Independent Languages*. PhD dissertation, Purdue University, Computer Science, August 1973.

[HZY91] L. Hungmong, L. Zongtian, and Z. Yifen. Design and implementation of the intermediate language in a PC decompiler system. *Mini-Micro Systems*, 12(2):23–28,46, 1991. Chinese language.

[IBM04] International Business Machines Corporation. *XL C Language Reference V7.0*, 2004. Retrieved Aug 2005 from `scv.bu.edu/SCV/Archive/IBM/language7.pdf`.

[Ioc88] The International Obfuscated C Code Contest web page, 1988. Retrieved Aug 2004 from `http://www.ioccc.org`, file `phillipps.c`.

[IU00] IEEE-USA. Opposing adoption of the Uniform Computer Information Transactions Act (UCITA) by the states, 2000. Retrieved Jan 2003 from `http://www.ieeeusa.org/forum/POSITIONS/ucita.html`.

[Jan02] André Janz. Experimente mit einem Decompiler im Hinblick auf die forensische Informatik, 2002. Retrieved Aug 2005 from `http://agn-www.informatik.uni-hamburg.de/papers/doc/diparb_andre_janz.pdf`.

[Jar04] J. Jarrett, 2004. tagline, Electric Vehicle Discussion List, retrieved June 2004 from `http://autos.groups.yahoo.com/group/ev-list-archive/messages`.

[JP93] R. Johnson and K. Pingali. Dependence-based program analysis. In *Proceedings of the '93 SIGPLAN Conference on Programming Language Design and Implementation*, pages 78–89. ACM Press, June 1993.

[JS04] A. Johnstone and E. Scott. Suppression of redundant operations in reverse compiled code using global dataflow analysis. In *Software and Compilers for Embedded Systems: 8th International Workshop, SCOPES 2004*, Amsterdam, Netherlands, September 2004. Springer. LNCS 3199.

[JSW00] A. Johnstone, E. Scott, and T. Womack. Reverse compilation for Digital Signal Processors: a working example. In *Proceedings of the Hawaii International Conference on System Sciences*. IEEE-CS Press, January 2000.

[Jug05] JuggerSoft, 2005. (Also known as SST GLobal and Source Recovery.) Retrieved Apr 2006 from `http://juggersoft.com`.

[KCL+99] R. Kennedy, S. Chan, S. Liu, R. Lo, P. Tu, and F. Chow. Partial redundancy elimination in SSA form. *ACM Transations on Programming Languages and Systems*, 21(3):627–676, May 1999.

[KDM03] U. Khedker, D. Dhamdhere, and A. Mycroft. Bidirectional data flow analysis for type inferencing. *Computer Languages, Systems & Structures*, 29(1-2):15–44, 2003.

[Kil73] G. Kildall. A unified approach to global program optimization. In *Conference Record of the 1st ACM Symposium on Principles of Programming Languages*, pages 194–206. ACM Press, January 1973.

[Knu69] D. Knuth. *The Art of Computer Programming, Vol. 1.* Addison-Wesley, 1969.

[KO01] S. Katsumata and A. Ohori. Proof-directed de-compilation of low-level code. In *European Symposium on Prigramming*, pages 352–366. Springer, 2001. LNCS 2028.

[Kou99] P. Kouznetsov. JAD - the fast JAva Decompiler, 1999. Retrieved Jan 2003 from `http://kpdus.tripod.com/jad.html`.

[KU76] J. Kam and J. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, January 1976.

[Kum01a] K. Kumar. JReversePro - Java decompiler, 2001. Retrieved Jan 2003 from `http://jrevpro.sourceforge.net`.

[Kum01b] S. Kumar. DisC - decompiler for TurboC, 2001. Retrieved Feb 2003 from `http://www.debugmode.com/dcompile/disc.htm`.

[LA04] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the International Symposium on Code Generation and Optimization*, page 75, Palo Alto, California, 2004. IEEE CS Press.

[Lam00] M. Lam. Overview of the SUIF system, 2000. Presentation from PLDI 2000, retrieved Apr 2007 from `http://suif.stanford.edu/suif/suif2/doc-2.2.0-4/tutorial/suif-intro.ps`.

[Lat02] C. Lattner. LLVM: An infrastructure for muli-stage optimization. Master's thesis, University of Illinois at Urbana-Champaign, 2002. Also available Apr 2007 from `http://llvm.org/pubs/2002-12-LattnerMSThesis.html`.

[Les05] L. Lessig. Brief for Creative Commons as Amicus Curiae in support of respondents (MGM v Grokster), 2005. Retrieved Mar 2005 from `http://www.eff.org/IP/P2P/MGM_v_Grokster/20050301_cc.pdf`.

[MAF91] C. Mills, S. Ahalt, and J. Fowler. Compiled instruction set simulation. *Software – Practice and Experience*, 21(8):877–889, August 1991.

[McC76] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, December 1976.

[MDW01] R. Muth, S. Debray, and S. Watterson. `alto`: A link-time optimizer for the Compaq alpha. *Software – Practice and Experience*, 31(1):67–101, 2001.

[MH02] J. Miecznikowski and L. Hendren. Decompiling Java bytecode: Problems, traps and pitfalls. In *Proceedings of the International Conference on Compiler Construction*, pages 111–127, Grenoble, France, April 2002. Springer. LNCS 2304.

[Mic97] MicroAPL. MicroAPL: Porting tools and services, 1997. Retrieved Feb 2003 from `http://www.microapl.co.uk`.

[MLt02] MLton web page. BSD-style licensed software, 2002. Retrieved Jun 2006 from `http://www.mlton.org`.

[Moh02] M. Mohnen. A graph-free approach to data-flow analysis. In *Compiler Construction*, pages 185–213. Springer, 2002. LNCS 2304.

[Mor98] R. Morgan. *Building an Optimizing Compiler*. Digital Press, 1998. ISBN 155558179X.

[Mor02] K. Morisada. A decompilation introduction of Japanese, 2002. Retrieved Oct 2005 from `http://jdi.at.infoseek.co.jp` (Japanese); see also `http://www.program-transformation.org/Transform/AnatomizerDecompiler` (English).

[MR90] T. Marlowe and B. Ryder. Properties of data flow frameworks. *Acta Informatica*, 28:121–163, 1990.

[MRU07] Microsoft Research and the University of Virginia. *Phoenix Summer Workshop Presentation*, August 2007. Retrieved Aug 2007 from `https://connect.microsoft.com/Downloads/Downloads.aspx?SiteID=214`.

[MS04] Microsoft Corporation. *Phoenix Home Page*, 2004. Retrieved Aug 2007 from `https://connect.microsoft.com/Phoenix`.

[MWCG98] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 85 – 97. ACM Press, January 1998.

[Myc99] A. Mycroft. Type-based decompilation. In *8th European Symposium on Programming*. Springer, March 1999. LNCS 1576.

[Myc01] A. Mycroft. Comparing type-based and proof-directed decompilation. In *Proceedings of the Working Conference on Reverse Engineering*, pages 362–367, Stuttgart, Germany, 2001. IEEE-CS Press.

[NH06] N. Naeem and L. Hendren. Programmer-friendly decompiled Java. Technical Report SABLE-TR-2006-2, Sable Research Group, McGill University, March 2006. Retrieved Jul 2006 from `http://www.sable.mcgill.ca/publications/techreports/#report2006-2`.

[Nov03] D. Novillo. Tree SSA - A new optimization infrastructure for GCC. In *Proceedings of the GCC Developers Summit*, pages 181–194, June 2003.

[Nov04] D. Novillo. Design and implementation of tree SSA. In *Proceedings of the GCC Developers Summit*, pages 119–130, June 2004.

[Nov06] D. Novillo. Memory SSA - A unified approach for sparsely representing memory operations, 2006. Preliminary draft. Retrieved Apr 2007 from `http://people.redhat.com/dnovillo/pub/mem-ssa.pdf`.

[PR94] H. Pande and B. Ryder. Static type determination and aliasing for C++. In *Proceedings of the USENIX C++ Technical Conference*, pages 85–97, 1994. A longer version is available as Rutgers University Technical Report lcsr-tr-236, and empirical results from lcsr-tr-250-a.

[PR96] H. Pande and B. Ryder. Data-flow-based virtual function resolution. In *Static Analysis: Third International Symposium (SAS'96)*, pages 238–254. Springer, Sept 1996. LNCS 1145.

[PS91] J. Palsberg and M. Schwartzbach. Object-oriented type inference. *SIGPLAN Notices*, 26(11):146–161, 1991. Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA).

[QUT99] Queensland University of Technology. *Component Pascal on the JVM*, 1999. Retrieved Feb 2003 from `http://www.citi.qut.edu.au/research/plas/projects/cp_files/cpjvm.html`.

[RBL06] T. Reps, G. Balakrishnan, and J. Lim. Intermediate-representation recovery from low-level code. In *Proceedings of the ACM/SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipilation (PEPM'06)*, pages 100–111. ACM Press, January 2006. Invited talk.

[Reu88] J. Reuter, 1988. Public domain software; retrieved Feb 2003 from `ftp://ftp.cs.washington.edu/pub/decomp.tar.Z`.

[Ric53] H. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematics Society*, 74(2):358–366, 1953.

[Rob02] J. Roberts. Answer to forum topic "How do I translate EXE -> ASM -> EXE with IDA Pro?", 2002. Retrieved Mar 2005 from `http://www.datarescue.com/ubb/ultimatebb.php?ubb=get_topic;f=1;t=000429`.

[Roe01] L. Roeder. Lutz roeder's programming.net, 2001. Retrieved Sep 2004 from `http://www.aisto.com/roeder/dotnet`; offline as of April 2005.

[SA01] K. Swadi and A. Appel. Typed machine language and its semantics, July 2001. Retrieved Oct 2005 from `http://www.cs.princeton.edu/~appel/papers/tml.pdf`.

[Sas66] W. Sassaman. A computer program to translate machine language into Fortran. In *Proceedings SJCC*, pages 235–239, 1966.

[SBB+00] B. De Sutter, B. De Bus, K. De Bosschere, P. Keyngnaert, and B. Demoen. On the static analysis of indirect control transfers in binaries. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, Nevada, USA*, pages 1013–1019, June 2000.

[Sca01] Scale, a scalable compiler for analytical experiments, 2001. Retrieved Apr 2007 from `http://ali-www.cs.umass.edu/Scale`.

[SDGS99] V. Sreedhar, R. Dz-Ching Ju, D. Gilles, and V. Santhanam. Translating out of static single assignment form. In *Proceedings of the 6th International Symposium on Static Analysis*, pages 194–210. Springer, 1999. LNCS 1694.

[Seg92] Sega Entertainment Ltd. v. Accolade, Inc, 1992. 977 F.2d 1510 (9th Cir. 1992).

[SF06] *Open64 | The Open Research Compiler*, 2006. Retrieved Aug 2007 from `http://www.open64.net`.

[SFF+05] M. Suzuki, N. Fujinami, T. Fukuoka, T. Watanabe, and I. Nakata. SIMD optimization in COINS compiler infrastructure. In *Proc. 8th International Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems (IWIA2005)*, pages 131–140. IEEE Press, January 2005.

[SGI02] Silicon Graphics, Inc. *WHIRL Intermediate Language Specification*, 2002. Retrieved Aug 2007 from `http://www-rocq.inria.fr/~pop/doc/whirl.html`.

[SGVN05] G. Stitt, Z. Guo, F. Vahid, and W. Najjar. Techniques for synthesizing binaries to an advanced register/memory structure. In *Proceedings of the ACM/SIGDA symposium on Field-programmable gate arrays*, pages 118–124. ACM Press, 2005.

[SGVV02] G. Stitt, B. Grattan, J. Villarreal, and F. Vahid. Using on-chip configurable logic to reduce embedded system software energy. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 143–151. IEEE Press, 2002.

[Sim97] D. Simon. Structuring assembly programs. Honours thesis, The University of Queensland, School of ITEE, 1997. Also available Apr 2007 from `http://www.itee.uq.edu.au/~cristina/students/doug/dougStructuringAsmThesis97.ps`.

[Sin03] J. Singer. Static single information improves type-based decompilation, 2003. Unpublished. Was retrieved Sep 2004 from `http://www.cl.cam.ac.uk/~jds31/research/ssidecomp.pdf`.

[SKI04] M. Sassa, M. Kohama, and Y. Ito. Comparison and evaluation of back translation algorithms for Static Single Assignment form. In *IPSI*, Prague, Czech Republic, December 2004.

[SML01] Software Migrations, Ltd. *The Assembler Comprehension and Migration Specialists*, 2001. Retrieved Feb 2003 from `http://www.smltd.com`.

[SNK+03] M. Sassa, T. Nakaya, M. Kohama, Fukuoka T., and M. Takahashi. Static Single Assignment form in the COINS compiler infrastructure. In *SSGRR 2003w*, number 54, January 2003.

[Sof96] SofCheck Inc. Applet Magic, 1996. Retrieved Feb 2003 from `http://www.appletmagic.com`.

[Son84] Sony Corp. of America v. Universal City Studios, 1984. 464 U.S. 417 (2005), retrieved Nov 2005 from `http://www.eff.org/legal/cases/betamax/`.

[SPE95] Standard Performance Evaluation Corporation. *CPU95 benchmark*, 1995. Retrieved Mar 2002 from `http://www.spec.org/osg/cpu95/`.

[SR02a] Source Recovery, also known as JuggerSoft and SST Global, 2002. Retrieved Apr 2006 from `http://www.sourcecovery.com`, formerly `sourcerecovery.com`.

[SR02b] Source Recovery's HP-UX C/C++ Decompiler, 2002. Retrieved Feb 2004 from `http://www.sourcecovery.com/abstract.htm`.

[SRC96] The Source Recovery Company, 1996. Retrieved July 2004 from `http://www.source-recovery.com`.

[SST03] SST GLobal, 2003. (Also known as JuggerSoft and Source Recovery.) Retrieved Apr 2006 from `http://www.sstglobal.com`.

[Sta02] Static recompilers. Yahoo Tech Group, 2002. Retrieved Apr 2007 from `http://tech.groups.yahoo.com/group/staticrecompilers`.

[SW74] V. Schneider and G. Winiger. Translation grammars for compilation and decompilation. *BIT*, 14:78–86, 1974.

[SW93] A. Srivastava and D. Wall. A practical system for intermodule code optimization at link-time. *Journal of Programming Languages*, 1(1):1–18, March 1993. Also available as WRL Research Report 92/06; retrieved Apr 2004 from `citeseer.nj.nec.com/srivastava92practical.html`.

[TC02] J. Tröger and C. Cifuentes. Analysis of virtual method invocation for binary translation. In *Proceedings of the Working Conference on Reverse Engineering*, pages 65–74, Richmond, Virginia, 2002. IEEE-CS Press.

[Tho96] D. Thorpe. *Delphi Component Design*. Addison Wesley Longman, 1996.

[TKB03] F. Tip, A. Kiezun, and D. Bäumer. Refactoring for genearalization using type constraints. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 13–26. ACM Press, October 2003.

[Tol96] R. Tolksdorf. Programming languages for the Java Virtual Machine, 1996. Retrieved Feb 2003 from http://flp.cs.tu-berlin.de/~tolk/vmlanguages.html.

[TP95] P. Tu and D. Padua. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 414–423. ACM Press, 1995.

[UH02] *Overview of the Open64 Compiler Infrastructure*, November 2002. Retrieved Aug 2007 from `http://www2.cs.uh.edu/~dragon/Documents/open64-doc.pdf`.

[Upt03] E. Upton. Optimal sequentialization of gated data dependency graphs is NP-complete. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'03)*. CSREA Press, June 2003.

[UQB01] UQBT web page. BSD licensed software, 2001. Retrieved Apr 2002 from `http://www.itee.uq.edu.au/~cristina/uqbt.html`.

[Uti92] Unix-C.Utils, 1992. Retrieved Nov 2005 from `http://ftp.unicamp.br/pub/unix-c/utils/metrics.tar.Z`.

[Van03] P. Vandevenne. Answer to forum topic "Help with IdaPros generated ASM files required", 2003. Retrieved Apr 2006 from `http://www.datarescue.com/cgi-local/ultimatebb.cgi?ubb=get_topic;f=1;t=000495;p=0#000001`.

[VE98] M. Van Emmerik. Identifying library functions in executable files using patterns. In *Proceedings of the Australian Software Engineering Conference*, pages 90–97, Adelaide, Australia, Nov 1998. IEEE-CS Press.

[VE03] M. Van Emmerik. The PILER Decompilation System, 2003. Retrieved Jul 2003 from `http://www.program-transformation.org/Transform/PilerSystem`.

[VEW04] M. Van Emmerik and T. Waddington. Using a decompiler for real-world source recovery. In *Proceedings of the Working Conference on Reverse Engineering*, pages 27–36. IEEE-CS Press, November 2004. Extended version available from `http://www.itee.uq.edu.au/~emmerik/experience_long.pdf`.

[VH89] P. Van Hentenyck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[VWK+03] L. Vinciguerra, L. Wills, N. Kejriwal, P. Martino, and R. Vinciguerra. An experimentation framework for evaluating disassembly and decompilation tools for C++ and Java. In *Proceedings of the Working Conference on Reverse Engineering*, pages 14–23, Victoria, Canada, Nov 2003. IEEE-CS Press.

[War00] M. Ward. Reverse engineering from assembler to formal specifications via program transformations. In *Proceedings of the Working Conference on Reverse Engineering*, Brisbane, Australia, 2000. IEEE-CS Press.

[War01] M. Ward. The FermaT transformation system, 2001. Retrieved Feb 2003 from `http://www.dur.ac.uk/martin.ward/fermat.html`.

[War04] M. Ward. Pigs from sausages? Reengineering from assembler to C via FermaT transformations. *Science of Computer Prgramming Special Issue: Transformations Everywhere*, 52(1-3):213–255, 2004.

[WCES94] D. Weise, R. Crew, M. Ernst, and B. Steensgaard. Value depenence graphs: Representation without taxation. In *Conference Record of the 21st Annual ACM Symposium on Principles of Programming Languages*, pages 297–310. ACM Press, January 1994.

[Win01] Winelib web page, 2001. Retrieved Mar 2005 from `http://winehq.org/site/winelib`.

[Wol92] M. Wolfe. Beyond induction variables. In *SIGPLAN Conference on PLDI*, pages 162–174. ACM Press, 1992.

[Wol96] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.

[Wro02] G. Wroblewski. *General Method of Program Code Obfuscation*. PhD dissertation, Wroclaw University of Technology, Institute of Engineering Cybernetics, 2002. Retrieved Dec 2002 from `http://www.mysz.org/publications.html`.

[WZ91] M. Wegman and F. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

[Xu01] Z. Xu. *Safety-Checking of Machine Code*. PhD thesis, University of Wisconsin-Madison, 2001.

[YRL99] J. Yur, B. Ryder, and W. Landi. An incremental flow- and context-sensitive pointer aliasing analysis. In *Proceedings of the International Conference on Software Engineering*, pages 442–451, 1999.

Index