Software Transformation:
Applications, Tools, Challenges, and Program Representation

Eric Schulte
eschulte@grammatech.com

Michael McDougall
mcdougall@grammatech.com

Dave Melski
melski@grammatech.com

September 19, 2016

Abstract

We review the many applications, tools, and challenges for the automated analysis and transformation of software. This review covers both existing tools and techniques as well as emerging applications. We describe the general space of program representation, analysis, and transformation identifying particularly important special cases. We conclude with motivation for a program representation which may be used generally across different program transformation tools. We discuss desirable properties of such a representation to support general program transformation.

1 Introduction

Program transformation is so general a term as to include the entire field of software engineering. We can refine transformation into more restrictive categories, see Figure 1. Starting with *program translation*, rewriting a program from one representation to another, e.g. translating an ARM executable to x64, without changing its operational semantics or functional properties. Beyond *program translations*, *program transmutation* includes changes to a program's operational semantics which preserve functional properties. Finally fully general *program transformation* includes changes to a program's functional properties as well as non-functional properties and representation.

In section 2 we will begin with an analysis of the space of program representations and program translations between representations. We will identify common challenges including the generally undecidable problem of translating back *up* the space of program representations. In section 3 we will identify many potential use cases for program transformation, discussing specific interesting examples. In section 4 we will then review existing tools supporting program analysis and transformation, including those based on LLVM [14], CMU's Binary Analysis Platform (BAP) and GrammaTech's CodeSurfer®. Finally, in section 5 we suggest a particular program representation as the foundation upon which to build an ecosystem of tools for program transformation.

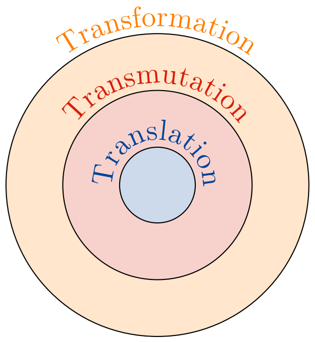

Figure 1: Restricted subsets of program transformations.

2 Program Representation and Translation

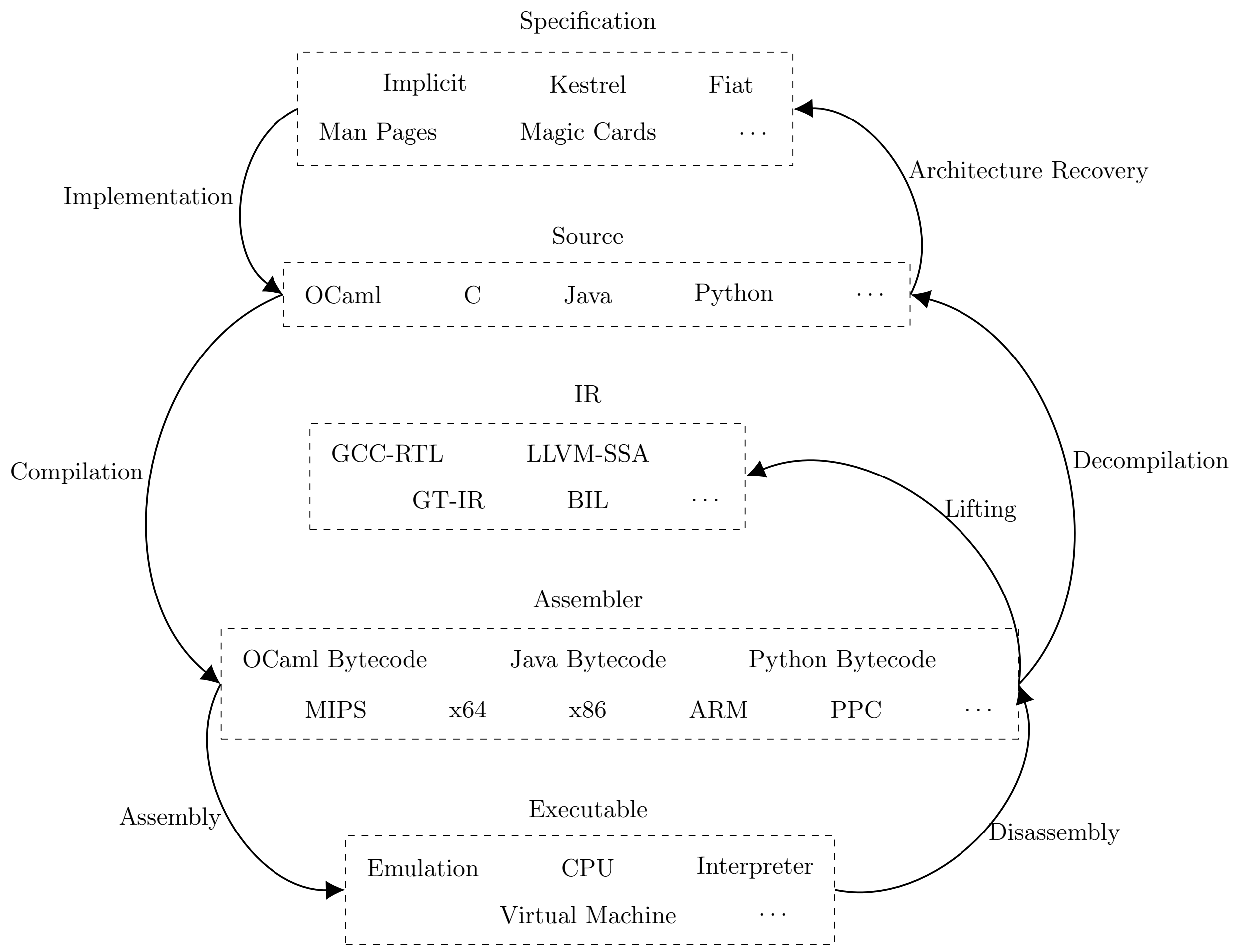

Figure 2: Space of program representations.

Program representations may loosely be grouped into the hierarchy shown in Figure 2. Moving down the hierarchy representations lose generality and gain additional operational specificity. Traditional software engineering tools and techniques such as compilers and assemblers perform such *descending* translations. General program translation also requires *ascending* translations. *Ascending* translations pose many challenges, some of which are generally undecidable. We discuss specific challenges below.

Symbol/literal disambiguation. One of the most significant problems encountered when disassembling program executables is determining whether numeric literals in the executable are constants or are addresses. A correct determination is critical as constants must be held constant by program translations while addresses must be changes to a symbolic form which permits translations which alter program layout. While this determination is generally undecidable, in practice it is often possible to correctly classify every literal in even large programs. When high confidence classification is impossible there are a number of possible mitigations of increasing severity.

1. *Pinning sections*, e.g. *.rodata*, of a program executable in memory is one way to ensure that untranslated program addresses, e.g. references to data, remain functional.

2. *Pinning targets* of control flow in memory provides similar benefits for untranslated addresses.
3. Dynamically *translating* all control flow targets from addresses in the original executable to corresponding addresses in the new executable provides the same benefit for code references without restricting layout in memory.
4. Retaining memory layout as much as possible and limiting changed code to either *trampolines* or to small size-constant differences alleviates most of this analysis burden.

Non-0-based arrays. A particularly difficult instance of literal disambiguation is Non-0-based arrays. In these cases the address used as the base of the array is never dereferenced directly but only when combined with, possibly large, offsets. This is troublesome because the address may lay outside of any valid memory region so heuristics which attempt to classify literals based on their values, or which attempt to collect referents based on addresses will fail.

Object delimitation in memory. The delineation of objects in global memory, in the heap, and on the stack is a difficult problem for binary analysis. However, modification to program data and code which builds and uses the stack requires this information to identify and potentially move objects.

Figure 2 hides many details. For example the lifting of executables to IR must handle multiple dimensions along which executables may differ. Knowledge of and customization to each of the following is necessary for reliable executable-to-executable rewriting.

ISA. Including x86, x64, MIPS ARM, ARM-thumb, ..., and specific versions of each.

Linking. Static or dynamic.

Operating System (environment). Windows, Linux, Mac, RTOS, ... all provide different runtimes (available system calls, error handling, etc...) in which binaries are executed.

Executable Format. ELF, PE, a.out, ...

Original Source Language. Determines non-code binary information including debug, relocation, dynamic linking, and error handling structures.

Compiler and compiler flags. Compilers use different *idioms* which when recognized provide useful information to binary lifters.

While program translation may be beneficial in its own right, for e.g. porting programs between execution environments by lifting to the be re-compiled and re-assembled, most often translation between program representation is done to perform some other transformation. In the next section we discuss uses of program transformation.

3 Applications of transformation

Common program transformations include optimization, hardening, obfuscation, diversification, reduction, and conglomeration. Each of these may be performed along any of the translations shown in Figure 2, as well as by translations within representations e.g., source-to-source transformations. The most common of these, the first two, are readily performed by compilers. With link time optimization and dead code elimination, reduction and conglomeration are also commonly performed by compilers. We discuss some of these transformations identifying particularly relevant representation translations below. GrammaTech has current or past funded research applying binary program transformation to support each of these following application areas.

3.1 Optimization

Commonly performed by compilers, optimization may also be performed along other translation paths. For example *super-optimization* [16], *stochastic super-optimization* [17], and *post-compiler optimization* [18] apply assembler-to-assembler transformations for optimization. Translating optimization opens up many additional opportunities for optimization, some of which we detail below.

Layer collapsing. One area where compiler optimizations are particularly limited is in the optimization of code which crosses separate compilation units, such as library boundaries. This leads to executables with potentially multiple layers of library invocations which perform redundant or unnecessary indirection, input transformation, and environment customization. Executable lifting has the opportunity to unify the many components of an executable into a single IR which may then be optimized to remove and combine redundant indirection.

Partial evaluation. Transformation of executables increases the ability of *end users* to transform programs. End users are in a special position in that they are free to customize the application to their particular use case. This presents opportunities not only for reduction (described below) but also for partial evaluation. One motivating example would be the evaluation of an application, e.g. a web server, which reads a configuration file to set program options, and to enable and disable functionality. Partial evaluation of the web server against the configuration file allows the transformation to perform traditional compiler transforms, e.g. constant propagation and dead code elimination, with the configuration data present. This also allows the memoization of startup operations.

Conglomeration. In some cases it may be useful to conglomerate multiple programs. This allows common functionality to be refactored saving space and potentially reducing the overhead of loading multiple different programs into memory. One example of this type of optimization could be the conglomeration of an executable with its shared libraries removing the need to dynamically load separate executables on startup.

Optimizations may generally be usefully applied alongside other transformations. For example transformations for hardening applied over the LLVM tool chain often tout the benefits of subsequent application of the existing LLVM optimization passes to transformed IR.

3.2 Hardening

Program hardening generally refers to the removal of behavior which violates a program's specification. This may be divided into two types of hardening (1) behavior which violates universal specifications, e.g. don't overrun buffers, and (2) behavior which violates a programs specific specification such as bugs.

Blanket defenses. Blanket defenses address (1) above and may be applied to rewrite whole programs to ensure uniform compliance to universal specifications. These include techniques of memory protection and control flow isolation.

Point repairs. Point repairs address (2) above and may be applied to specific locations in a binary at which bugs or vulnerabilities have been found. General fuzzing techniques have proven a popular technique of identifying point vulnerabilities in binaries.

3.3 Reduction.

Executables often include unused functionality. In addition to increasing the size of executables, with additional code and data, and slowing the runtime of executables, with additional control flow, unused functionality increases the attack surface. A common example would be PDF readers which include frequently exploited JavaScript engines. Another frequent contributor to application bloat is shared libraries which may frequently be included into a program which intends to use only a small fraction of their total functionality.

End users may wish to remove unused functionality, but typically only have access to executables. Thus reduction transformations which are applied along executables to executables translations are particularly motivating.

4 Techniques and tools

Program synthesis from specification. Although a number of tools have been developed for the synthesis of programs from specifications most are applicable only to narrow domains. A small selection of examples are discussed below.

- **Fiat** The Fiat library for the Coq proof assistant uses stepwise refinement to compile specifications into executable implementations [6].
- **Automated Program Transforms (APT).** The Kestrel Institute is building APT [11], a tool which seeks to automate the synthesis of program implementations from high level specifications leveraging the ACL2 theorem prover [12].
- **Macho.** Man pages provide implicit specifications of the programs they document. Using natural language techniques along with synthesis leveraging a large code database the MACHO tool has been able to synthesize text processing tools from man pages [5].
- **Latent predictor networks.** In recent work a team at Google has trained neural networks to synthesize python classes defining cards for the *Magic the Gathering* card game using the cards themselves as the specification. This approach treats synthesis as a translation problem and innovates by conditioning traditional sequence to sequence translation techniques on specific information allowing the technique to leverage both unstructured natural language and structured information from the cards [15].

Binary Analysis Platform (BAP) The BAP binary analysis platform provides an open infrastructure for the analysis and verification of program executables [2]. BAP works by first lifting the executable, using tooling based on either IDA Pro [8] or LLVM [14], to an intermediate binary intermediate layer or BIL representation. Analysis plugins may then be written to process the BIL representation.

CodeSurfer GrammaTech's CodeSurfer platform provides support for analyzing and rewriting program executables [1]. CodeSurfer similarly works by first lifting executables to an internal IR. This IR provides both assembler-agnostic information about the functional behavior of lifted code, as well as assembler specific information about the code's original representation. Programs are transformed at the IR level, and the subsequent modified IR may then be pretty printed to assembler and linked to form a transformed executable.

Diablo Diablo is a link-time binary rewriter for whole-program optimization [20]. Diablo requires that programs be compiled with an altered version of GCC instrumented to output additional information to aid in program lifting. Diablo targets whole-program optimizations to optimize programs speed and compaction.

LLVM lifting tools SecondWrite [19], Fracture [13], and McSema [7] are binary rewriting frameworks which work by first lifting program executables to the LLVM IR. Programs may then be transformed and recompiled using traditional LLVM compiler passes. Additionally the RevNIC tool uses dynamic analysis to lift binary code to LLVM IR [3], and the RevGen tool performs the same lift statically [4].

Zipper The University of Virginia's Zipper is another framework for binary rewriting [10]. Zipper first performs program analysis populating a database of program information. This database is used to rewrite the program, however, unlike most other frameworks discussed herein Zipper pins control flow targets rather than performing a full re-assembly round trip.

5 Rewriting Program Representation

After reviewing a number of prominent uses and tools for program transformation, the intermediate program representation emerges as a critical element. Functional and non-functional program transformations are typically written as IR-to-IR passes. Similarly, tools for executable-to-executable rewriting universally work in two passes, first lifting the original binary to an IR and second generating transformed output leveraging information in the IR.

The LLVM project demonstrates by example the power that a well designed IR can have to unify and accelerate a research community [14]. The benefits in the ability of separate projects and researchers to re-use each other's analysis, transformations, and supporting tooling provides a very large productivity boost. In fact, the utility of the LLVM compiler framework has led to the development of multiple general program transformation tools targeting the LLVM infrastructure, even though LLVM was only designed to support compilation (a specific instance of program transformation).

The program transformation community should seek to identify the characteristics of a intermediate program representation desirable to support *general* program transformation (both *ascending* and *descending* translations). The LLVM IR may be a good choice, however the static single assignment nature of LLVM's IR may not be well suited to as a target IR for lifting executable programs. We posit the following desirable attributes as a plausible beginning to this list.

Low level memory model. The memory model used by the IR should be compatible with common machine code memory models, including a global address space, stack, and heap.

Explicit environment representation. In addition to code, program executables encode information about their execution environment including memory layouts and permissions, error handling, and dynamically loaded libraries. This should be represented in program IR.

Control flow graph. Representation as a traversable control flow graph.

Types. The IR should support type decorations similar to those applied to program ASTs parsed from source code. This will enable automated reasoning and the proving over program IR.

Concrete decorations. The IR should support concrete decorations to preserve information lost in lifting. For example, decorations indicating the specific instructions used when lifting machine code to a more general functional IR.

Functional semantics. The IR should encode functional semantics in such a form as to enable traditional compiler analysis as well as projection into abstract domains for abstract analysis.

Operational semantics. The IR should be *executable*, this supports dynamic checking of translations, as well as techniques of dynamic analysis such as micro-execution [9].

References

[1] Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. Codesurfer/x86 – a platform for analyzing x86 executables. In *International Conference on Compiler Construction (CC)*, volume 3443 of *LNCS*, pages 250–254. Springer, 2005/04/04/April 4 2005. Edinburgh, UK.

[2] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. Bap: A binary analysis platform. In *Computer Aided Verification*, pages 463–469, 2011.

[3] Vitaly Chipounov and George Candea. Reverse engineering of binary device drivers with revnic. In *Proceedings of the 5th European conference on Computer systems*, pages 167–180. ACM, 2010.

[4] Vitaly Chipounov and George Candea. Enabling sophisticated analyses of× 86 binaries with revgen. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 211–216. IEEE, 2011.

[5] Anthony Cozzie, Murph Finnicum, and Samuel T King. Macho: Programming with man pages. In *Proceedings of the 2011 Workshop on Hot Topics in Operating Systems (HotOS 2011)*, 2011.

[6] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. Fiat: Deductive synthesis of abstract data types in a proof assistant. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 689–700, 2015.

[7] Artem Dinaburg and Andrew Ruef. Mcsema: Static translation of x86 instructions to llvm. In *ReCon 2014 Conference, Montreal, Canada*, 2014.

[8] Chris Eagle. *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler.* No Starch Press, 2008/// 2008.

[9] Patrice Godefroid. Micro execution. In *Proceedings of the 36th International Conference on Software Engineering*, pages 539–549. ACM, 2014.

[10] J. D. Hiser, Anh Nguyen-Tuong, Michele Co, BenjaminD Rodes, Matthew Hall, C. Coleman, John C. Knight, and J. W. Davidson. A framework for creating binary rewriting tools. In *Proceedings of the 2014 Tenth European Dependable Computing Conference*, pages 142–145, 2014/// 2014. Washington, DC.

[11] Kestrel Institute. Apt: Automated program transformations. http://www.kestrel.edu/home/projects/apt/.

[12] Matt Kaufmann and J. Strother Moore. An industrial strength theorem prover for a logic based on common lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, 1997.

[13] Draper Labs. Fracture: an architecture-independent decompiler to llvm ir.

[14] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[15] Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomas Kocisky, Andrew Senior, Fumin Wang, and Phil Blunsom. Latent predictor networks for code generation. *arXiv preprint arXiv:1603.06744*, 2016.

[16] H. Massalin. Superoptimizer: a look at the smallest program. *ACM SIGARCH Computer Architecture News*, 15(5):122–126, 1987.

[17] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *Proceedings of the eighteenth international conference on Architectural Support for Programming Languages and Operating Systems.* ACM, 2013.

[18] Eric Schulte, Jonathan Dorn, Stephen Harding, Stephanie Forrest, and Westley Weimer. Post-compiler software optimization for reducing energy. In *Proceedings of the eighteenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 639–652. ACM, 2014.

[19] Matthew Smithson, Khaled ElWazeer, Kapil Anand, Aparna Kotha, and Rajeev Barua. Static binary rewriting without supplemental information: Overcoming the tradeoff between coverage and correctness. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 52–61. IEEE, 2013.

[20] Ludo Van Put, Dominique Chanet, Bruno De Bus, Bjorn De Sutter, and Koen De Bosschere. Diablo: a reliable, retargetable and extensible link-time rewriting framework. In *Proceedings of the Fifth IEEE International Symposium on Signal Processing and Information Technology, 2005.*, pages 7–12. IEEE, 2005.