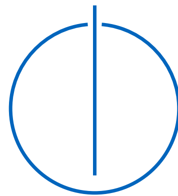
TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK
Lehrstuhl für Sprachen und Beschreibungsstrukturen



Binary Analysis using On-Demand Tabulation of Function Summaries

Julian Kranz

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK
Lehrstuhl für Sprachen und Beschreibungsstrukturen

Binary Analysis using On-Demand Tabulation of Function Summaries

Julian Kranz

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:
Prof. Tobias Nipkow, Ph.D.

Prüfer der Dissertation:
1. Prof. Dr. Helmut Seidl
2. Prof. Dr. Antoine Miné

Die Dissertation wurde am 16.01.2019 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 10.04.2019 angenommen.

Abstract

Program analysis tries to recover properties of software in order to find flaws or to understand the ideas behind a software system. Modern software is typically written in a high-level programming language and then translated into machine instructions by a compiler. An analysis can either work on the source code written by the programmer directly or use the compiled binary as input. Analyzing the source code has the advantage of being able to gather information from abstractions and control structures of the respective programming language. For example, the type of a variable constrains the set of operations that may modify its value. However, source code level analysis also has drawbacks. First and foremost, the source may not be available. This is the case, e.g., when analyzing proprietary systems or malware. In addition, source code analysis relies on the semantics specification of the programming language. This specification, however, may not be exact or may intentionally not cover all possible program constructs (as is the case for C and C++). Last but not least, the compiler itself may contain bugs resulting in a divergence between the language specification and the actual program behaviour. As a result, binary program analysis has recently gained attention.

This work addresses two main challenges of binary program analysis. Due to the complexity of modern processors (consider, e.g., the Intel x86 architecture) writing an analysis for a specific machine architecture is cumbersome and not portable. Thus, the binary has to be decoded and translated into an analysis-friendly architecture-independent intermediate representation (IR). Implementing a decoder is itself error-prone because general-purpose programming languages lack constructs required for an intuitive decoder specification. Therefore, we present a DSL that offers built-in syntax for instruction decoders. We also show how we compile our specifications into naturally-looking C code which can be easily understood and debugged by a human reader. The second challenge of binary analysis is scalability. Binary analysis has to deal with sized inputs resulting from code size increases by large factors during compilation and translation into an IR. A common means of dealing with the problem of analyzing a large program is breaking it up into smaller chunks – e.g. functions – and analyzing these in isolation. However, modularity may lead to a precision loss that is not acceptable. As a remedy, we present an analysis methodology that tabulates functions for certain properties of calling contexts on-demand. This approach allows us

to reach the necessary performance while not giving up contextual information where it is required.

We demonstrate our analysis algorithm using benchmarks gathered from an implementation that is based on our DSL for binary disassembly and that implements the modular analysis using on-demand tabulation of function summaries.

Zusammenfassung

Programmanalyse ist darum bemüht, Eigenschaften von Software zu rekonstruieren, um auf diese Weise Fehler zu entdecken oder die Ideen hinter einem Softwaresystem zu verstehen. Moderne Software wird üblicherweise in einer höheren Programmiersprache geschrieben und anschließend von einem Compiler in Maschinen-Instruktionen übersetzt. Eine Analyse kann entweder direkt mit dem Quelltext arbeiten, den der Programmierer verfasst hat, oder das kompilierte Binärprogramm als Eingabe verwenden. Den Quelltext zu analysieren hat den Vorteil, die Möglichkeit zu haben, Informationen aus den Abstraktionen und Kontrollstrukturen der jeweiligen Programmiersprache gewinnen zu können. Zum Beispiel beschränkt der Typ einer Variablen die Menge an Operationen, die ihren Wert verändern können. Allerdings hat die Analyse von Quelltext auch Nachteile. Zuallererst ist möglich, dass der Quelltext nicht verfügbar ist. Dies ist zum Beispiel der Fall, wenn ein proprietäres System oder Schadsoftware analysiert werden soll. Außerdem muss sich eine Analyse des Quelltextes auf die Spezifikation der Semantik der Programmiersprache verlassen. Diese Spezifikation kann allerdings ungenau sein oder sogar bewusst einige Programmkonstrukte nicht abdecken (dies ist bei C und C++ der Fall). Schließlich kann auch der Compiler selbst Fehler enthalten, was zu einer Divergenz zwischen der Spezifikation der Sprache und dem tatsächlichen Verhalten des Programms führt. Aus diesen Gründen hat die Analyse auf Binärebene an Wichtigkeit gewonnen.

Diese Arbeit beschäftigt sich mit zwei zentralen Herausforderungen von Binäranalyse. Wegen der Komplexität moderner Prozessoren (man bedenke, z.B., die x86-Achitektur von Intel) ist umständlich und nicht portabel, eine Analyse für eine spezifische Maschinen-Architektur zu entwerfen. Aus diesem Grund muss der Binärcode zunächst dekodiert und in eine analysefreundliche und nicht von der Architektur abhängige Zwischendarstellung (IR) übersetzt werden. Die Implementierung des Dekodierers selbst ist fehleranfällig, da Allzweck-Programmiersprachen keine passenden Konstrukte mitbringen, um Dekodierer intuitiv zu spezifizieren. Aus diesem Grund stellen wir eine domänenspezifische Sprache (DSL) vor, eine eingebaute Syntax für Instruktionsdekodierer anbietet. Wir erklären außerdem, wie wir die resultierenden Spezifikationen in natürlich aussehenden Quelltext der Programmiersprache C übersetzen, der von einem menschlichen Leser leicht verstanden und auf Fehler geprüft werden kann. Die zweite Herausforderung von Binäranalyse ist die Skalierbarkeit.

Binäranalyse muss mit umfangreichen Eingaben umgehen können, die sich aus starken Vergrößerungen des Codes während der Kompilierung bzw. der Übersetzung in eine IR ergeben. Eine übliche Herangehensweise, um mit dem Problem der Analyse großer Programme umgehen zu können, ist es, diese in kleinere Teile – z.B. Funktionen – herunterzubrechen und diese Teile in Isolation zu analysieren. Allerdings kann Modularität zu einem Präzisionsverlust führen, der nicht akzeptabel ist. Als Abhilfe hierfür stellen wir eine Analysemethodik vor, die Funktionen nach bestimmten Eigenschaften der Aufruf-Kontexte nach Bedarf tabelliert. Dieser Ansatz erlaubt es uns, die nötige Performanz zu erreichen, ohne dabei Informationen über den Kontext zu verlieren, wo ebendiese nötig sind.

Wir demonstrieren unseren Analyse-Algorithmus durch Messungen anhand einer Implementierung, die auf unserer DSL zur Disassemblierung von Binärcode aufbaut und die unsere modulare Analyse umsetzt, die bedarfsorientiert Funktionsabstraktionen tabelliert.

Acknowledgments

I would like to give thanks to Axel Simon. He officially was my supervisor until January 2015 and continued to assist me throughout my time as a doctoral candidate. This work would have not been possible without his constant support over the past years.

Contents

Abstract ii

Zusammenfassung iv

Acknowledgments vi

I. Introduction 1

1. Introduction 2

II. The GDSL Toolkit: An Architecture-Independent Framework for Machine Code Disassemblers 9

2. GDSL: The Generic Decoder Specification Language 10

2.1. General Language Overview 12
2.1.1. Endianness Configuration 15
2.2. Decoding x86 Prefixes 16
2.3. Evaluation 18
2.3.1. Performance 19
2.3.2. Correctness 20
2.4. Related Work 20

3. Semantics Translation using RReil 24

3.1. RReil Intermediate Representation 24
3.2. The Generic Decoder Specification Language (GDSL) 25
3.3. Writing Semantics using GDSL 26
3.3.1. An Example Intel Instruction 26
3.3.2. Generating RReil Statements using GDSL Monadic Functions . . 27
3.3.3. The Translator 28
3.4. Optimizing the RReil Code 29
3.4.1. Liveness Analysis and Dead Code Elimination 29
3.4.2. Forward Expression Substitution 30

3.5. Empirical Evaluation 31
3.6. Future Work 32

4. Verification of the Decoder and the Translator 34
4.1. Automatic Generation of End-to-End Tests 34
4.2. Generation of x86 Machine Instructions 35
4.3. Execution of the Generated Instruction 36
4.4. Test Results and Error Conditions 38

5. Compiling GDSL to C 40
5.0.1. Heap-Allocation and Avoidance of Garbage Collection 40
5.0.2. Unboxing of Polymorphic Values and Closures 41
5.0.3. Transformation of Monadic Functions 42
5.1. Lowering GDSL to Core 43
5.2. Conversion to an Imperative Language 48
5.2.1. Translating Monadic Sequences 58
5.3. Optimizing the Intermediate Representation Imp 60
5.3.1. Simplifying Imp 60
5.3.2. Removing Monadic Actions 62
5.3.3. Unboxing by Type Inference 65
5.4. Implementation 70
5.5. Experimental Evaluation 71
5.6. Related Work 73
5.7. Conclusion 75

III. Scalability Through Modular Analysis 77

6. Modular Analysis of Executables using On-Demand Heyting Completion 78
6.1. Preliminary Definitions 81
6.1.1. Abstract Interpretation of the Collecting Semantics 83
6.2. Modular Program Semantics 86
6.2.1. Abstract Interpretation of the Relational Semantics 86
6.2.2. Abstract Semantics of Memory Accesses 88
6.2.3. Application of Function Summaries 90
6.2.4. Computing a Fixpoint of the Abstract Relational Semantics . . . 101
6.3. On-Demand Heyting Completion 103
6.3.1. Extracting Refinement Information using Herbrand Terms 103
6.3.2. Specializing Summaries with Herbrand Terms 106
6.3.3. Combining Specialized Function Summaries 107

6.3.4. Heyting Completion . . . 109
6.4. Implementation . . . 110
6.5. Related Work . . . 112
6.5.1. Conclusion . . . 113

IV. Analysis Implementation and Evaluation **115**

7. The *Summy* Analysis Tool **116**
7.1. Getting Started . . . 116
7.2. Running the Analyzer . . . 117
7.3. Output of the Driver Tool . . . 118
7.4. RReil Code Optimization . . . 122
7.5. Fixpoint Computation . . . 125
7.5.1. Additional Narrowing Iteration . . . 129
7.6. Implementation of Memory Regions . . . 130
7.6.1. Handling of Conflicting Accesses . . . 130
7.6.2. Further Ideas for Improvement . . . 131
7.7. Evaluation . . . 131

V. Conclusion **135**

List of Figures **138**

List of Tables **141**

Bibliography **142**

Part I.

Introduction

1. Introduction

Program analysis tries to recover properties of software programs in order to find flaws or understand the ideas behind a software system. The latter particularly applies to malware as there is no documentation available that describes its functionality. Finding flaws is particularly relevant for strengthening the security of systems since software nowadays is immensely important for every part of modern life and, thus, exposes a massive attack surface.

In general, an analysis can work with different input program representations: It can either analyze the source code of a program in some higher level programming language or it can analyze the machine instructions which were generated by a compiler from the source of a program. Generally, the task of the analyzer is significantly simpler when dealing with source code because programming languages are geared towards readability and understandability which can also be exploited by the analysis tool. For example, programming languages offer high-level control flow structures and help in componentization of large system, both of which help during program analysis. As a matter of fact, programming languages are sometimes designed with certain types of program analyses in mind (e.g. automatic lifetime deduction for references in Rust [58]) in order to allow for meaningful error messages and aid refactoring during development.

However, analyzing the source code also has drawbacks. First, the source code needs to be available which is not always the case. This may be because of software licensing or because the authors of the software are unwilling to allow the analysis of their software, e.g. in case of malware. Second, the analysis of source code needs to rely on the specified semantics of the programming language. However, the definition of the semantics may be vague, implementation defined, or even undefined for some language constructs (and, indeed, many C and C++ programs rely on undefined behaviour [62]). Here, the semantics implemented by the compiler needs to be approximated which can be a very laborious task, in particular in the presence of different compilers and optimization settings that can affect the behaviour of the generated program. As a result, binary program analysis is a necessity.

Binary program analysis uses the CPU-specific instructions of a program as an input. Thus, it directly processes the code as seen by the machine and only has to rely on the semantics specification of the silicon manufacturer. However, there is a multitude of

hardware architectures, some of which include hundreds of instructions with complex semantics. It is therefore desirable to separate the analysis algorithm itself from both the complexity of machine instructions and the concrete hardware architecture. A common approach for this is to first translate the input program into some intermediate representation (IR) that is common to all supported architectures. The analysis then only needs to implement transformers for the semantic statements of the IR. In such a design, new hardware can easily be supported by the analyzer by providing a suitable decoder and semantics translator. In this work, we use a refined version of the REIL IR [19] called RReil (Relational Reverse Engineering Intermediate Language) [48] which is geared towards binary analysis by having a very small and simple set of semantic statements while offering flexibility through, e.g., allowing accesses at arbitrary bit offsets into registers.

Decoding machine instructions and translating them into an IR seems to be a simple and straight forward software engineering problem at first sight. However, instruction sets are built for interpretation by machines and are – in some well-known cases such as Intel x86 – extended oftentimes during their lifespan. As a result, software implementations of decoders tend to contain a huge number of rules which makes them hard to understand and maintain. Even though a large number of decoding tools already existed [1], none of them seemed to be a perfect fit for our requirements: We needed a decoding tool that allows for an easy specification of machine decoders that closely follows the layout of the documentation of the chip manufacturer as we deem this to be the best way to keep the code readable and, thus, extensible and maintainable. We therefore developed the GDSL (Generic Disassembler Specification Language) toolkit that offers a functional ML-like DSL that is specifically designed for instruction decoder specifications. In this work, we present the current state of the GDSL project and future directions.

Figure 1.4 contains the software lifecycle from the viewpoint of binary program analysis. First, source code is written. This source code is then compiled to machine code. Later, the machine code is decoded and translated into an IR which is used by an analysis tool to recover properties of the software. As an example, consider the C++ code in Fig. 1.1. The code defines two structures `A` and `B` where `B` inherits from `A`. Line 15 uses dynamic function binding in order to invoke the function `f()` in structure `B`. Figure 1.2 shows the x86 assembly code that results from compiling the code with version 6 of `clang` from the LLVM project using the `-O1` optimization flag. The figure shows the assembly code for all relevant functions. Each line within a function begins with the address of the instruction in memory which is followed by the bytes that encode the instruction. Finally, each line contains a string representation of the instruction in AT&T assembly syntax. Note that mangled function names have been replaced by more readable identifiers. The main function commences by calling the

```
struct A {
  virtual int f() {
    return 99;
  }
};

struct B : public A {
  virtual int f() {
    return 42;
  }
};

int main(void) {
  A *a = new B();
  return a->f();
}
```

Figure 1.1.: Example code that uses dynamic function binding.

memory allocation function in line 4 which returns a pointer to the object in register `%rax`. Next, `main()` calls the constructor of structure `B` in line 8 which expects a pointer to the object it is constructing in register `%rdi`. The constructor of structure `B`, in turn, calls the constructor of structure `A` which initializes the virtual table pointer in line 23. The pointer to the virtual table is replaced by a pointer to the virtual table of the subclass, i.e. the virtual table of `B`, in line 18. This way, when tail-calling the first entry in the virtual table in line 12, control dispatches to the subclass version of `f`, namely `B::f()`. Note that the two lines of C++ code in `main()` translated to 11 assembly instructions. Further note that the instructions are encoded using a varying number of bytes. This is caused by the rather complex encoding rules of the Intel x86 architecture.

Figure 1.3 contains a string representation of the RReil code that is generated by the GDSL toolkit for the main function in Fig. 1.2. RReil allows for arbitrary bit sizes and offsets. For example, line 9 writes the 32 bit value 0 to `DI` at offset 32. Note that a single machine instruction is translated into a number of RReil statements, resulting in another increase in code size compared to the x86 machine code. Also note that the shown RReil code is not optimal; for example, the register `IP` is incremented multiple times without being read in between. Such updates could be merged together.

The increase in size of the source code becomes a challenge to the scalability of a

```
0000000000400600 <main>:
  400600: 53                    push   %rbx
  400601: bf 08 00 00 00        mov    $0x8, %edi
  400606: e8 f5 fe ff ff        callq  400500 <memory_alloc>
  40060b: 48 89 c3              mov    %rax, %rbx
  40060e: 48 c7 03 00 00 00 00  movq   $0x0, (%rbx)
  400615: 48 89 df              mov    %rbx,%rdi
  400618: e8 13 00 00 00        callq  400630 <B_constructor>
  40061d: 48 8b 03              mov    (%rbx), %rax
  400620: 48 89 df              mov    %rbx, %rdi
  400623: 5b                    pop    %rbx
  400624: ff 20                 jmpq   *(%rax)

0000000000400630 <B_constructor>:
  400630: 53                    push   %rbx
  400631: 48 89 fb              mov    %rdi, %rbx
  400634: e8 17 00 00 00        callq  400650 <A_constructor>
  400639: 48 c7 03 18 07 40 00  movq   $0x400718, (%rbx)
  400640: 5b                    pop    %rbx
  400641: c3                    retq

0000000000400650 <A_constructor>:
  400650: 48 c7 07 68 07 40 00  movq   $0x400768,(%rdi)
  400657: c3                    retq

0000000000400660 <B_f>:
  400660: b8 2a 00 00 00        mov    $0x2a, %eax
  400665: c3                    retq

0000000000400670 <A_f>:
  400670: b8 63 00 00 00        mov    $0x63, %eax
  400675: c3                    retq
```

Figure 1.2.: Assembly code for the C++ program in Fig. 1.1

```
 1  push %rbx
 2    IP =:64 (IP + 1)
 3    T0 =:64 B
 4    SP =:64 (SP - 8)
 5    *[64]SP =:64 T0
 6  mov $0x8, %edi
 7    IP =:64 (IP + 5)
 8    DI =:32 8
 9    DI.32 =:32 0
10  callq 400500
11    IP =:64 (IP + 5)
12    SP =:64 (SP - 8)
13    *[64]SP =:64 IP
14    goto [CALL] [64](IP + -267)
15  mov %rax, %rbx
16    IP =:64 (IP + 3)
17    B =:64 A
18  movq $0x0, (%rbx)
19    IP =:64 (IP + 7)
20    *[64]B =:64 0
21  mov %rbx, %rdi
22    IP =:64 (IP + 3)
23    DI =:64 B
```

```
24  callq 400630
25    IP =:64 (IP + 5)
26    SP =:64 (SP - 8)
27    *[64]SP =:64 IP
28    goto [CALL] [64](IP + 19)
29  mov (%rbx), %rax
30    IP =:64 (IP + 3)
31    T0 =:64 *[64]B
32    A =:64 T0
33  mov %rbx, %rdi
34    IP =:64 (IP + 3)
35    DI =:64 B
36  pop %rbx
37    IP =:64 (IP + 1)
38    T0 =:64 *[64]SP
39    SP =:64 (SP + 8)
40    B =:64 T0
41  jmpq *(%rax)
42    IP =:64 (IP + 2)
43    T1 =:64 *[64]A
44    goto [JUMP] [64]T1
```

Figure 1.3.: Translated RReil code for the main function in Fig. 1.2.

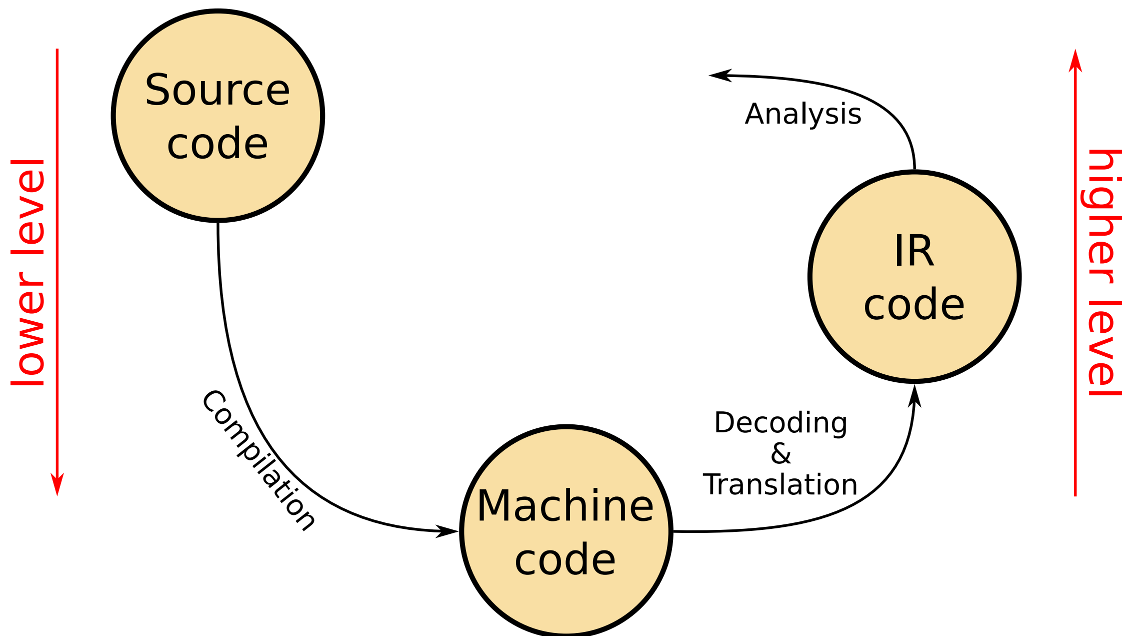


Figure 1.4.: Software lifecycle from the viewpoint of binary analysis.

program analysis. Figure 1.3 illustrates this for our example code. In general, compiling a single line of C code results in around 10 x86 machine instructions on average. Each of these machine instructions is then translated into an IR program which requires around four IR statements (already taking optimizations into account) per instruction. As a result, scalability needs careful consideration during development of a binary analysis tool. In this work, we discuss an analysis algorithm that uses on-demand tabulation of function summaries. Here, the analyzer initially processes each function in isolation, thus keeping the analysis modular. However, this modularity may lead to a severe precision loss – consider, for example, a function that expects a pointer to a structure of type `A` as parameter and invokes the function `f()` on it. In this case, a fully modular analysis would need to make worst-case assumptions about the target of the call. Our analyzer recognizes such situations. It tabulates the function for each concrete function pointer value passed to the function and makes use of the now fixed target during summary computation. This way, we compromise on modularity in order to keep the analysis as precise as necessary, however without sacrificing scalability.

Again consider the code in Fig. 1.1 and Fig. 1.2. In the C++ code, the static type of the variable a constrains the set of possible targets of the virtual call in line 15 (the call can either dispatch to `A::f()` or `B::f()`). In the assembly code, on the other hand, line 12 alone gives no clue as to where the tail-call dispatches. Indeed, an analysis has to infer

the effects of the constructor calls and apply them to the state of the object pointed to by `%rax` in order to build the inter-procedural control flow graph. The analysis algorithm presented in this work is able to do that.

In summary, we make the following contributions:

- We describe a functional ML-like language called GDSL that is geared towards the implementation of instruction decoders and semantics translators. We provide a compiler from GDSL to C, but other back-end languages are possible.
- We evaluated the language design by implementing a decoder for all 897 (as of September 2012) instructions of the Intel x86 architecture and offering semantics translations for about half of those instructions.
- We show how we use GDSL to implement basic optimizations on the produced IR. This allows us to provide optimizations that are independent of both the machine architecture of the translated instructions and the target language of the DSL compiler.
- We present a structure preserving compilation of our functional language GDSL to C that results in code that resembles manually written C code. This enables the developer of the DSL program to use an off-the-shelf C debugger to debug the DSL code.
- We present an analysis framework for programs in executable format that makes use of on-demand tabulation of function summaries in order to achieve both scalability and sufficient precision. We demonstrate our framework using an analysis that reconstructs the control flow and call graph.

Parts of this work have been published in the following papers:

1. *GDSL: A Generic Decoder Specification Language for Interpreting Machine Language* [1]
2. *GDSL: A Universal Toolkit for Giving Semantics to Machine Language* [33]
3. *The GDSL Toolkit: Generating Frontends for the Analysis of Machine Code* [55]
4. *Structure-Preserving Compilation: Efficient Integration of Functional DSLs into Legacy Systems* [34]
5. *Modular Analysis of Executables Using On-Demand Heyting Completion* [36]

Part II.

The GDSL Toolkit: An Architecture-Independent Framework for Machine Code Disassemblers

2. GDSL: The Generic Decoder Specification Language

The reconstruction of assembler instructions from an input (byte) sequence that comprises the program is the first step towards binary program analysis. The second step is to map each statement to a meaning which may be a value-, timing- or energy semantics, etc., depending on the goal of the analysis. Both aspects are commonly addressed by writing an architecture-specific decoder and a translator to some internal representation expressed in the implementation language of the analysis. The goal of our work is to build an infrastructure to specify decoders and translations to semantics using a domain specific language (DSL) that can be compiled into the programming language of the analysis tools. To this end, we present GDSL and motivate its design by the task of specifying decoders for Intel x86.

The incentive for creating a DSL to specify decoder and semantics of assembler instructions was a discussion at a Dagstuhl seminar on the analysis of executable code. Here, it was realized that many research groups implemented prototype analyses using an architecture-specific decoder and a hand-written semantics interpretation. Besides duplication of work, these approaches are usually incomplete, are bound to one architecture and are hard to maintain since their representation of instructions is geared towards a specific project. In the presence of steadily increasing instruction sets and the need to adapt an analysis to new targets such as virtual machines contained in malware, maintainability and simplicity of decoder specifications is of increasing importance.

To this end, it is desirable to group instructions logically or, when converting a manufacturer's manual, in alphabetical order; we call this mnemonic-centric specification. For the sake of efficiency, however, a decoder must make a decision based on the next value from the input sequence (opcode-centric dispatch) which precludes testing opcode patterns one after the other. While a classic scanner generator like `lex` can convert a mnemonic-centric specification to an opcode-centric decoder, it allows and encourages overlapping patterns. Consider the following `lex` scanner specification:

```
1  while|do|switch|case   { printf("keyword %s", yytext); }
2  [a-zA-Z][a-zA-Z0-9]*   { printf("ident %s", yytext); }
```

Opcode	Instruction	Description
00 /r	ADD r/m8,r8	Add r8 to r/m8.
28 /r	SUB r/m8,r8	Subtract r8 from r/m8.

Table 2.1.: Two typical instructions in the Intel x86 manual.

Here, the patterns for the keywords and the identifier are overlapping: the input `while` matches both rules. In this case, `lex` uses the rule that appears first in the specification file. Thus, a keyword is returned. Overlapping patterns are desirable in a scanner specification since they improve readability and conciseness. In an instruction decoder, however, overlapping patterns are undesirable since the sequence in which the rules are written starts to matter which, in turn, precludes a mnemonic-centric specification. Hence, a DSL for maintainable decoder specifications must provide a concise way of writing non-overlapping patterns to exactly match an instruction.

Another challenge is the processing of non-constant bits of an instruction that are used to specify parameters. Since parameter bits often follow recurring patterns, an abstraction mechanism is required to keep the specification concise. For example, the mod/rm-byte in Intel x86 instructions follows many opcodes and determines which register or memory addressing mode to use. Table 2.1 shows an excerpt of the Intel manual where the first column shows the two bytes that together form an instruction. The second byte `/r` is the mod/rm-byte that determines which 8-bit registers r8 and which pointer or register r/m8 stand for. Within our decoder specification language, we define functions `r/m8`[1] and `r8` to generate the arguments of an instruction. The contents of the mod/rm-byte are read by a sub-decoder named `/r` that stores the read byte in an internal decoder state. This sub-decoder can be re-used in the decoder for ADD and SUB:

```
1  val main [0x00 /r] = binop ADD r/m8 r8
2  val main [0x28 /r] = binop SUB r/m8 r8
```

Here, the decoder `main` is declared as reading `0x00` (resp. `0x28`) from the input before running the sub-decoder `/r`. The `binop` function is a simple wrapper that executes functions `r/m8` and `r8` (which access the values stored by `/r`) and applies the results to the passed-in constructor (here `ADD` and `SUB`). By using sub-decoders such as `/r` that communicate via the internal state, our `main` decoder comes very close to the specification in Table 2.1 which is a simplified excerpt from the Intel manual.

[1]We allow / as part of an identifier to accommodate the Intel nomenclature.

Since our DSL is an ML-like functional language, it is powerful enough to describe all parts of a decoder, even `r/m8` and `r8` that are often hand-coded primitives in other decoder frameworks. This comprehensive approach enables users to add instructions that have not been anticipated in the original design of `/r`. In summary, GDSL improves over existing approaches as follows:

- Its abstraction mechanisms enable the definition of instruction decoders that are very close to the syntax used in manufacturer's manuals, thereby ensuring maintainability even by the end users of the decoder framework.
- Our specification is type checked during compilation and overlapping patterns are detected. This ensures high fidelity of the resulting decoder, especially in the presence of mistakes in the manufacturer's manuals.
- The DSL is flexible enough to accommodate a variety of architectures. Due to its general nature, it is possible to add translations from native instructions to some abstract semantics, which will enable binary analysis tools to analyse code for any architecture that is described with our framework. In Chap. 3, we describe how we use GDSL for a translation to our IR called RReil.
- We provide a prototype compiler that generates C code which is competitive with other decoders. The specifications can be translated to other languages or used for other purposes (e.g. test generation) by writing a new backend.

After the next section presents the design of GDSL, Sect. 2.2 illustrates its expressiveness by detailing the decoding of Intel prefixes. Section 2.3 presents an evaluation of our implementation before Sect. 2.4 presents related work.

2.1. General Language Overview

This section discusses the design of GDSL by illustrating the use of the various syntactic constructs. The general idea is that the decoder specification is an executable functional program that consumes the input sequence and produces a heap containing the abstract syntax tree (AST) that represents the recognized instruction. After the AST in the heap has been processed, the heap can be reused for decoding the next instruction, thereby avoiding the need for a garbage collector or for allocating memory with each instruction.

The grammar of GDSL is shown in Fig. 2.1 on page 22. In the following, we refer to single productions using their name in the right column of the table. A file consists of a sequence of definitions given by **Decl**.

Line *type decl.* shows the production for algebraic data types that introduce (or extend) the type `t-id` with constructors `con`. As in ML, each constructor takes zero or one argument, allowing the definition of enumerations such as `type register = AX | BX | CX | DX` or AST nodes such as `type op = Reg of register | Mem of {size : int, reg : op} | Imm8 of [8]`. Here, the argument to the `Mem` constructor is a record while `Imm8` takes a bit vector of 8 bits, written `[8]`. Bit vectors and `int` are the only basic data types with singleton bit-vectors acting as Booleans. Abbreviations for complex types can be introduced using the syntactic construct in line *type abbrev.*. Line *action type* contains the production for the type of a monadic action. A monadic action has a result type (first non-terminal **Type**) and transforms a state of type t to a state of type t'. The `export` keyword states which of the decoders, functions and constants are publicly visible to the client code. In line 1 of Fig. 2.2 on page 23, we export an action `decode` that produces a result of type `instr`. It requires a state of sub-type of `{}` and produces a new state which has a type that is a super-type of `{}`.

Productions *function decl.*, *decoder decl.*, and *guarded decoder decl.* introduce functions, decoders and decoders with guards, respectively. Functions and decoders differ in that functions take arguments and have exactly one definition whereas decoders read from the implicit input stream and definitions with the same name augment each other. Consider the decoder snipped in Fig. 2.2. Here, `binop` and `r/m8` in lines 10 and 20 are functions taking three and no arguments, respectively. In contrast, lines 16, 18 and 25 define decoders whose right-hand-side is evaluated if the token sequence in the square brackets matches the current input. Tokens can be specified in three ways (Production *single token* in Fig. 2.1): either as a hexadecimal number (c.f. the first token of `main`), as a call to another decoder (c.f. the second token of `main`) or as a bit pattern (as used in the `/r` and `/0` decoders). Bit patterns, in turn, are enclosed in ticks and are given by Productions *bit patterns*, *pattern binding*, and *pattern var*:

- strings of 0,1,. (c.f. `000` in `/0`); the dot acts as a wildcard; a set of bit strings can be specified by separating them using a vertical bar, e.g. `00|01|10`.
- as above, with a leading variable separated by `@`; the variable is bound to the actual bits in the input; for instance, `/0` could have been written as follows:

```
1  val /0 ['mod:2 reg@000 rm:3'] =
2    update @{mod=mod, reg/opcode=reg, rm=rm}
```

- a variable with a width in bits; the notation `v:3` is syntactic sugar for `v@...`; examples are `mod`, `reg` and `rm` in the decoders `/r` and `/0`.

The semantics of "calling" another decoder within a token sequence is that the pattern of the called decoder is substituted where it appears and that its body is

prepended to the right-hand-side of the decoder. For instance, `main [0x80 /0]` is translated internally as follows:

```
val main [0x80 'mod:2 000 rm:3'] = do
  update @{mod=mod, reg/opcode='000', rm=rm};
  binop ADD r/m8 imm8
end
```

After inlining sub-decoders, an equivalent function without decoder syntax can easily be derived using a `consume8` primitive that reads one byte from the input stream. Here, all patterns of the rules of `main` are transformed into the following combined function:

```
val main = do
  byte1 <- consume8;
  case byte1 of
    '0x80' : do
      byte2 <- consume8;
      case byte2 of
        'mod:2 000 rm:3' : do
          update @{mod=mod, reg/opcode='000', rm=rm};
          binop ADD r/m8 imm8
        ...
    '0x00' : ...
```

During this translation, overlapping patterns are detected. For rules of varying lengths, a prefix length n is determined as the minimum number of bits that all rules read. Each rule is examined in turn by taking its prefix of size n and removing the matched patterns from an initial set of all patterns of size n. If this fails, i.e. the pattern to remove is no longer contained in the set, the rule is reported to overlap with a previous rule. Consider, for example, a prefix of size 2 which leads to the initial set {`'00'`, `'01'`, `'10'`, `'11'`} and two overlapping rules `'0.'` and `'00'`. For the first rule, the patterns `'00'` and `'01'` are removed from the initial set, resulting in the set {`'10'`, `'11'`}. For the second rule, the pattern `'00'` would need to be removed; however, `'00'` is not contained in the set of remaining patterns. As a result, the second rule is reported to overlap with a previous rule. Note that the actual implementation has to represent the set of remaining patterns more efficiently in order not to require an exponential amount of memory.

Once it is known that rules do not overlap, they can be re-arranged for more efficient pattern matching. Specifically, we repeatedly identify those bit positions that contain

the most constant bits (i.e. 0 or 1 but not wildcards) and generate a switch-statement for those bits. The results are nested `switch`-statements that mostly have consecutive cases and are therefore translated by the compiler into efficient tables.

The bodies of functions and decoders are given by the **Expr** productions. Here, productions *binding*, *bifurcation*, *conditional*, *function appl.*, *constants*, and *constructor/var* give the standard constructs found in a functional language with **Expr Expr** in line *function appl.* denoting function application. Our language allows the creation of compound values using records which are collections of field names bound to a value. Production *record constant* allows the construction of new records (used in line 13 of Fig. 2.2 on page 23). The value of a field `foo` is extracted using `$foo` which itself is a function. Thus, `$foo {foo=7}` evaluates to 7. Analogously, `@{foo=x}` is a function taking a record and setting the field `foo` to `x`. For instance, `@{bar='110'} {foo=7}` evaluates to `{bar='110', foo=7}`.

In order to allow for an internal state, each decoder is a monad, a concept borrowed from the pure functional language Haskell [44]. A monad is an abstract type containing a function from an input state to an output state and a result. The motivation for monads is to chain together computations that operate on a state without requiring side-effects in the language. Production *monadic seq.* details the `do`-statement which threads together monadic actions whose result can be bound to an identifier. A monadic action is a normal function that has a monadic type, i.e. expects the monadic state as parameter and produces a new, possibly modified state as result. The result of the `do`-statement is that of the last action. Production *basic actions* presents the three basic monadic actions of our language: `update f` applies `f` to the internal state (and is usually a record update); `query f` returns the result of applying `f` to the internal state (and is usually a record field selector); and `return x` that returns `x` as a result (this is a shorthand for `let val f _ = x in query f`).

The internal state can also be accessed using guards. Guards allow an additional case distinction based on the global state after a matching decoder rule has been selected. An example of guards can be seen in Fig. 2.2. Here, the first guard of `$opndsz`, `$rexw`, and `otherwise` in lines 27f that evaluates to `'1'` determines which right-hand-side is evaluated. Guards are functions taking the internal state as argument. Thus, `opndsz` and `rexw` are record fields in the internal state and `otherwise` is a function always returning `'1'`.

2.1.1. Endianness Configuration

When reading from the input stream, we have to take its instruction word size and endianness into account. For example, assume an architecture that uses 32 bit instruction words in big endian mode. Here, the first byte of an instruction at address *i* is not

found at address i, but at an offset of three, i.e. at $i + 3$. We allow the configuration of the endianness through the `endianness` primitive as can be seen in line 6 of Fig. 2.2 on page 23. The primitive expects a bit vector as parameter. The integer value of this bit vector is XOR'ed with the offset when reading the instruction stream. For little endian architectures, we use the vector 0. For big endian architectures, we use a vector ending in n one bits for an architecture word size of $8 * (n + 1)$. As an example, assume we want to read the second byte of the 4th instruction using the aforementioned 32-bit big endian architecture. The $4th$ instruction can be found at offset $3 * 4 = 12$ (3 preceding instructions of size 32 bits, i.e. 4 bytes, each). Thus, we want to access the 13th (counting from zero) byte of the instruction stream when not taking endianness into account. In order to find a proper endianness configuration vector for the architecture, we need to determine an n such that $8 * (n + 1)$ equals the word size, i.e. 32. It follows that n is 3 and the respective vector is $0b11$. XOR'ing the offset 13 with the vector `0b11` gives us $13 \oplus 11_2 = 14$. Indeed, given that the instruction spans 4 bytes starting from offset 12, this is the 2nd last byte, i.e. the byte we want to read.

2.2. Decoding x86 Prefixes

One challenge in decoding x86 instructions is the correct handling of prefixes: they either serve to modify the following instruction or they are part of the following opcode (a so-called mandatory prefix). In the latter case, other prefixes are allowed between the mandatory prefix and the actual opcode. For example, both instruction sequences **67 f3** `45 0f 7e d1` and **f3 67** `45 0f 7e d1` encode movq xmm10, xmm9 where `67` is an ADDRSZ prefix and `f3` is a REPNE prefix, but used here as mandatory prefix to extend the opcode `0f 7e`. Moreover, `45` is another "standard" REX prefix and `d1` the mod/rm byte. Confusingly, the REX prefix must immediately precede the opcode, otherwise it is ignored.

Since the mandatory prefix extends the opcode, there may be multiple instructions with the same opcode but different mandatory prefixes. For example, the instructions mulss, mulsd, and mulpd share the same opcode, here `0f 59`, but have different mandatory prefixes, namely `f2`, `f3`, and `66`, respectively. If multiple mandatory prefixes are present, the order in which these prefixes occur and dominance rules become important. For example, while the last occurrence of `f2` and `f3` determines the mandatory prefix, an occurrence of `66` is only recognized as mandatory prefix if `f2` and `f3` cannot start an instruction. A correct decoder recognizes[2]:

[2]Some of these instructions contain illegal prefixes; however, remember that we want our decoder to be able to decode all instructions accepted by Intel processors.

f3 **f2** 0f 59 ff	mulsd xmm7, xmm7	Mandatory prefix: **0xf2**
66 f3 **f2** 0f 59 ff	mulsd xmm7, xmm7	Mandatory prefix: **0xf2**
66 f2 **f3** 0f 59 ff	mulss xmm7, xmm7	Mandatory prefix: **0xf3**
66 0f 59 ff	mulpd xmm7, xmm7	Mandatory prefix: **0x66**
f2 66 0f 59 ff	mulsd xmm7, xmm7	Mandatory prefix: **0xf2**

Mandatory prefixes can easily be handled in GDSL by using different decoders, depending on the last relevant prefix. We decode prefixes using the following decoders that encode a state machine with start state “prefixes”:

```
1  val prefixes [0x66] = p/66
2  val prefixes [0xf2] = p/f2
3  val prefixes [0xf3] = p/f3
4  val prefixes [] = main
5  val p/66 [0x66] = p/66
6  val p/66 [0xf2] = p/66/f2
7  val p/66 [0xf3] = p/66/f3
8  val p/66 [] = after /66 main
9  val p/f3 [0x66] = p/66/f3 #f3 dominates 66
10 val p/f3 [0xf2] = p/f3/f2
11 val p/f3 [0xf3] = p/f3
12 val p/f3 [] = after /f3 main
13 val p/f3/f2 [0x66] = p/66/f3/f2 #f3/f2 dominates 66
14 val p/f3/f2 [0xf2] = p/f3/f2
15 val p/f3/f2 [0xf3] = p/f2/f3
16 val p/f3/f2 [] = after /f2 (after /f3 main)
17 ... #analogous for p/f2, p/66/f2, p/66/f3, p/f2/f3,
18     #               p/66/f3/f2, p/66/f2/f3
19 val /66 [] = continue #no match, continue with next decoder
20 val /f2 [] = continue #no match, continue with next decoder
21 val /f3 [] = continue #no match, continue with next decoder
22 val /66 [0x0f 0x59 /r] = binop MULPD xmm xmm/m128
23 val /f2 [0x0f 0x59 /r] = binop MULSD xmm xmm/m64
24 val /f3 [0x0f 0x59 /r] = binop MULSS xmm xmm/m32
25 val main [...] = ...
```

The entry point that is exported to the user is `prefixes`. When reading the sequence `f3 f2 0f 59 ff`, it dispatches to `p/f3` which itself reads `f2` and enters the decoder `p/f3/f2`. Since the next byte `0f` has no match in `p/f3/f2`, the expression `after /f2`

`(after /f3 main)` is executed. The `after` function calls the decoder `/f2` and, if it fails, continues with `(after /f3 main)`. The latter expression runs `f3` and, if this decoder fails, runs `main`. On our example byte sequence, the `/f2` decoder succeeds in consuming the remaining bytes `0f 59 ff` and returns the `mulsd` instruction. By construction of the prefix decoders, at most four decoder calls can fail to make progress, that is be unable to decode at least one further byte: one prefix decoder, plus one call to `/66`, `/f2`, and `/f3`, respectively. Thus, the adherence of the prefix rules only adds a small constant to the runtime complexity of the decoder.

Note that the example code does not contain the logic for recording which prefixes have been seen and are not mandarory, i.e. need to be taken into account to *configure* the decoded instruction. In pratice, this makes the rules a little harder to read:

```
1  val p/f2/f3 [] = after (with-f2 /f3) (
2    after (with-f3 /f2) (with-f2 (with-f3 /)))
```

Here, the `with-XX` calls encapsulate the calls to the decoders and set the respective prefix bits in the global state before calling the decoder passed as parameter. After the call to the decoder, the prefix bits are reset in the state so that they do not interfere with further decoder calls. Observe that `after` and `continue` can be defined directly within GDSL:

```
1  val after fst snd = do update @{cont=snd}; fst end
2  val continue = do decoder <- query $cont; decoder end
```

Here, `after` stores its argument `snd` in the decoder state and executes the decoder `fst`. The `continue` function retrieves the stored decoder and dispatches to it. Note that the main decoder has no default rule calling `continue` and, thus, fails in case no pattern matches. This completes the design of our prefix decoders.

2.3. Evaluation

Our GDSL decoder handles all 897 (as of September 2012) instructions of the Intel x86 instruction set; we also offer decoders for AVR, MSP430, and a substantial subset of ARMv7. In this section we compare the performance and correctness of the Intel x86 decoder.

Framework	**Time**	**#Instrs**	**p/f2/f3**	**p/66/f2/f3**	REX
BeaEngine	238ms	672207	–	–	–
distorm	204ms	671991	–	–	–
GDSL	673ms	671991	√	√	√
IDA Pro	/	/	√	–	√
libopcodes	309ms	671991	–	–	–
metasm	4m21s	/	–	–	√
udis86	705ms	673965	–	–	–
xed2	338ms	671991	√	√	√

Table 2.2.: Evaluation of different disassembler frameworks.

2.3.1. Performance

We compare the performance of our generated code with several existing disassembler projects. Table 2.2 shows the running time for a linear sweep disassembly of a binary consisting of 671991 instructions in the *.text* segment. The size of the *.text* segment was 3032027 bytes. The binary is one of our earlier decoders and is a statically linked x86_64 executable for Linux. Due to linking `libc` statically, it included several *SSE* and *VEX* instructions. We used *BeaEngine* [6], *distorm* [18], *IDA Pro* [28], *libopcodes* as shipped in a Debian package [38], *metasm* [41], *udis86* [60], and the *xed2* disassembler library that comes with the *pintool* [63] package. We ran all tests on an Intel Core i7 on Linux in 64-bit mode. The discrepancy in the number of decoded instructions for *BeaEngine* and *udis86* is due to incorrectly decoded instructions which subsequently results in decoding further incorrect instructions due to different offsets.

We included the *metasm* package to complete the comparison with a disassembler not written in C. A possible reason for the results of the *metasm* package being slower is that it does not only do a linear sweep but also resolves symbols and does some control-flow analyses using the decoded instructions. Similarly, we were unable to run a linear-sweep disassembly using *IDA Pro*.

As can be seen from Table 2.2, the generated C code of GDSL is comparable in speed, being about 3 times slower than the fastest hand-written library. Since decoding is unlikely to be a bottleneck in program analysis, we deem the performance acceptable. However, Chapter 5 describes how we apply further optimizations during code generation that help us not only to improve the readability of the generated C code, but also allow us to leverage existing optimizations found in off-the-shelf C compilers.

2.3.2. Correctness

Due to the complications of decoding byte sequences that contain prefix bytes, we compared the various disassemblers for correctness. Table 2.2 features three columns, labelled **p/f2/f3**, **p/66/f2/f3**, and REX, which test various prefix combinations as described in Sect. 2.2: **p/f2/f3** states if the order of f2 and f3 is honoured, **p/66/f2/f3** states if additionally 66 loses its mandatory prefix status once f2 or f3 was read, and REX states if this prefix is correctly ignored if not immediately preceding the opcode. A tick indicates a correct decoder.

According to the Intel manual, adding arbitrary prefixes may result in unpredictable behavior for certain instructions. We created byte sequences whose behaviour is unpredictable according to the manual and verified that an Core i7 processor executes them as if the superfluous prefixes were absent. While it could be argued that decoding sequences that are marked with unpredictable behavior is undesirable for program analysis, such sequences are routinely emitted by the *gcc* compiler which inserts prefixes in front of nop and ret instructions for alignment purposes. As an example, consider the following 14-byte padding sequence that occurred in our test binary:

```
666666662e0f1f840000000000:
  nop WORD PTR cs:[rax+rax*1+0x0]
```

Here, four 66 prefixes precede a segment override prefix 2e before a nop opcode f1 f8 follows which takes an elaborate argument. Furthermore, malware may add spurious prefixes as additional code obfuscation technique. Thus, a decoder has to recognize more than what the manual recommends.

On the contrary, certain applications, such as the search for gadgets (byte sequences that form a specific instruction), require that a decoder only recognizes instructions common to all processors. Our GDSL language can use guards from barring certain instructions from being recognized. Certain aspects, such as the difference between 32-bit and 64-bit mode can be implemented using different prefix decoders (the REX prefix is a normal instruction in 32-bit mode). We believe that an open-source implementation of a decoder is likely to converge to a decoder that is correct under all such configurations.

2.4. Related Work

Most decoder libraries for the Intel x86 instructions generate or use tables for mapping opcodes to instructions, however, the decoding of prefixes and arguments is usually hand-coded [38, 6, 18, 41]. One notable exception is *SLED* [45], a specification language

for encoding and decoding, which is a comprehensive specification language similar to GDSL. *SLED* specifies mnemonics using opcode-centric tables, thereby assigning fixed values to mnemonics. Besides mnemonics, it is possible to define pattern variables that associate names with sequences of bits. The mnemonics and pattern variables are then used to define an instruction. The fields of a pattern variable in such a definition can be specialized using constraints. Since these constraints are rather generic, it is not clear to which extent they can check if the resulting instruction definitions overlap (i.e. that the intersection of the constraint set of one rule with the constraint set of another rule is empty) and, thus, how often it can be avoided that constraints must be tested in sequence in order to find a matching pattern. Their approach is similar to regular expression matching, but without allowing repetition. Since the x86 instruction set allows for multiple and identical prefixes in many, but not arbitrary sequences, certain prefixed instructions are difficult to specify. In particular, the padding example using a `nop` in Sect. 2.3.2 is difficult to specify using *SLED* due to the inability to specify repetition. In fact, to our understanding, the specification given in [45] for x86 would not accept any instruction with superfluous prefixes. Even then, the ability of *SLED* to decode and encode instructions requires the specification to be bi-directional and therefore becomes relatively hard to understand and to maintain.

Another approach was taken by Fox et al. [23]. In their work they describe a formal model of the complete ARMv7 instruction set encoded in the HOL4 proof system [56]. The model directly operates on word sequences, as even the decoding logic is specified in the proof system. Besides mere decoding logic, a full semantics of the ARMv7 instruction set is also provided whose fidelity against an ARMv7 implementation was proved. Since the direct use of the decoder that is written in the HOL4 proof system is difficult, a provably correct translation to GDSL would be desirable.

The next chapter addresses the specification of semantics for which many intermediate representations have been suggested [11, 5, 48]. The expressed goal of GDSL is to also specify how a processor instruction can be translated to an intermediate representation that describes its semantics. Using a common framework can help to make the various intermediate representations comparable and usable in various analysis frameworks. Recently, Reps et al. have proposed to compile an abstract transformer for each processor instruction in order to obtain a more efficient analysis [39]. Future work will address how a different backend to our compiler can follow this setup.

Our implementation of GDSL is available at the website `https://github.com/gdslang/gdsl-toolkit`. It is written in SML/NJ v110.74 and released under a BSD license.

```
Decl      ::= export id : Type                                     export decl.
            | type id = con ( of Type )?( | con ( of Type )?)*     type decl.
            | type id = Type                                       type abbrev.
            | val id  id* = Expr                                   function decl.
            | val id [ TokPat* ] = Expr                            decoder decl.
            | val id [ TokPat* ] ( | Expr = Expr )+                guarded dec.
TokPat    ::= hex-num | id | ' BitPat* '                           single token
Type      ::= int | | num | | id
            | { field : Type ( , field : Type )* }
            | ( TypeList ) -> Type
            | S Type < Type => Type >                              action type
TypeList  ::= ϵ
            | Type ( , Type )*
BitPat    ::= BitStr ( | BitStr )*                                 bit patterns
            | id @ BitStr ( | BitStr )*                            pattern binding
            | id : num                                             pattern var
BitStr    ::= ( 0 | 1 | . )+                                       bit string
Action    ::= id ← Expr ; Action                                   monadic stmts
            | Expr ; Action
            | Expr
Cases     ::= CasePat : Expr ; Cases | ε                           body of case
CasePat   ::= ' BitStr ' | num
            | con id | con
Expr      ::= let  Decl in Expr end                                binding
            | if  Expr then Expr else Expr                         bifurcation
            | case Expr of Cases end                               conditional
            | Expr Expr+                                           function appl.
            | " string " | ' ( 0 | 1 )* ' | num                    constants
            | con | id                                             constructor/var
            | { } | { field = Expr ( , field = Expr )* }           record constant
            | @ { field = Expr ( , field = Expr )* }               record update
            | $ field                                              record selector
            | do Action end                                        monadic seq.
            | query Expr | update Expr | return Expr               basic actions
```

Figure 2.1.: The GDSL language grammar without monadic and generic types; we use the well-known POSIX syntax for regular expressions [29].

```
export decode : () -> S instr <{} => {}>

type instr = ADD of {op1:op, op2:op}

val decode = do
  endianness '';
  main
end

val binop cons giveOp1 giveOp2 = do
   operand1 <- giveOp1;
   operand2 <- giveOp2;
   return (cons {op1=operand1, op2=operand2})
end

val /r ['mod:2 reg:3 rm:3'] =
  update @{mod=mod, reg/opcode=reg, rm=rm}
val /0 ['mod:2 000 rm:3'] =
  update @{mod=mod, reg/opcode='000', rm=rm}
val r/m8 = do  #similar for r8, r/m16, r16, ...
   r <- query $rm;
   return (case r of '000': Reg AL | '001': Reg BL )
end

val main [0x80 /0] = binop ADD r/m8 imm8
val main [0x00 /r] = binop ADD r/m8 r8
val main [0x01 /r] | $opndsz = binop ADD r/m16 r16
                   | $rexw = binop ADD r/m64 r64
```

Figure 2.2.: Specification for decoding the Intel ADD instruction.

3. Semantics Translation using RReil

In the last chapter of this work, we presented how we use the GDSL language to specify instruction decoders. The specification of such decoders is the first step towards binary program analysis, followed by the translation of the instructions into the representation used by the analyzer; that is, semantics needs to be associated with the instructions. Recall that the challenge here is one of scalability: a single line in a high-level language is translated into several assembler instructions. Each native instruction, in turn, is translated into a number of semantic primitives. These semantic primitives are usually given as an *intermediate representation* (IR) and are later evaluated over an abstract domain [15] tracking intervals, value sets, taints, etc. In order to make the evaluation of the semantic primitives more efficient, a *transformer-specification language* (TSL) was recently proposed that compiles the specification of each native instruction directly into operations (transformers) over the abstract domain [39], thus skipping the generation of an IR. These tailored transformers are then optimized by a standard compiler. Our toolkit follows the more traditional approach of generating an IR that an analysis later interprets over the abstract domains. In contrast to the TSL approach, we perform optimizations on the IR program that represents a complete basic block rather than on a single native instruction. We show that the semantics of instructions can be simplified considerably when taking the surrounding instructions into account which highlights the optimization potential of using an IR.

3.1. RReil Intermediate Representation

Many intermediate representations for giving semantics to assembler instructions exist, each having its own design goals such as minimality [5, 19], mechanical verifiability [23], reversibility [45], or expressivity [5, 48]. Our own RReil IR [48], presented in Fig. 3.1, was designed to allow for a precise numeric interpretation. For instance, comparisons are implemented with special tests rather than expressed at the level of bits which is common in other IRs [19, 23, 39].

stmts	::=	ε | **stmt** ; **stmts**	**var**	::=	*id* | *id* . *int*
stmt	::=	**var** = : *int* **expr**	**addr**	::=	**linear** : *int*
	|	**var** = : *int* [**addr**]	**linear**	::=	*int* · **var** + **linear** | *int*
	|	[**addr**] = : *int* **expr**	**sexpr**	::=	**linear** | *arbitrary*
	|	*if* (**sexpr**) { **stmts** } *else* { **stmts** }		|	**linear cmp** : *int* **linear**
	|	*while* (**sexpr**) { **stmts** }	**expr**	::=	**sexpr**
	|	*cbranch* **sexpr** ? **addr** : **addr**		|	**linear bin linear**
	|	*branch* (*jump* | *call* | *ret*) **addr**		|	*sign-extend* **linear** : *int*
	|	(**var** : *int*)* = " *id* " (**linear** : *int*)*		|	*zero-extend* **linear** : *int*
cmp	::=	$\leq_s$ | $\leq_u$ | $<_s$ | $<_u$ | $=$ | $\neq$	**bin**	::=	*and* | *or* | *xor* | *shr* | ...

Figure 3.1.: The syntax of our RReil (Relational Reverse Engineering Language) IR. The construct ": *int*" denotes the size in bits whereas ". *int*" in the **var** rule denotes a bit offset. The statements are: assignment, read from address, write to address, conditional, loop (both only used to express the semantics within a native instruction), conditional branch, unconditional branch with a hint of its original purpose, and a primitive "*id*".

3.2. The Generic Decoder Specification Language (GDSL)

We developed a domain specific language called GDSL that is best described as a functional language with ML-like syntax. It features bespoke pattern matching syntax for specifying instruction decoders. Dependability of GDSL programs is increased by a sophisticated type inference [54] that eliminates the need of specifying any types. The algebraic data types and a special infix syntax facilitates the specification of instruction semantics and program optimizations.

The GDSL toolkit contains a compiler for GDSL as well as decoders, semantic translations and optimizations written in GDSL. The benefit of specifying optimizations in GDSL is that they can be re-used for any input architecture since they operate only on RReil. Besides a few instruction decoders for 8-bit processors, the toolkit provides a partial ARMv7 decoder and an Intel x86 decoder for 32- and 64-bit mode that handles all 897 (as of September 2012) Intel instructions. In terms of translations into RReil, we provide semantics for 457 instructions. Of the 440 undefined instructions, 228 are floating point instructions that we currently do not handle since our own analyzers cannot handle floating point computations. Many of the remaining undefined instructions would have to be treated as primitives as they modify or query the internal CPU state or because they perform computations whose RReil semantics is too cumbersome to be useful (e.g. encryption instructions).

a)

```
val sem-cmovcc insn cond = do
  size <- sizeof insn.opnd1;
  dst <- lval size insn.opnd1;
  dst-old <- rval size insn.opnd1;
  src <- rval size insn.opnd2;

  temp <- mktemp;
  mov size temp dst-old;

  _if cond _then
    mov size temp src;

  write size dst (var temp)
end
```

b)

```
t0 =:32 B
if (ZF) {
  t0 =:32 A
} else {
}
B =:32 t0
B.32 =:32 0
```

Figure 3.2.: The translator function a) and a translation result b)

3.3. Writing Semantics using GDSL

As a pure, functional language with algebraic data types and a state monad, GDSL lends itself for writing translators in a concise way as illustrated next.

3.3.1. An Example Intel Instruction

The following GDSL example shows the translation of the Intel instruction `cmov`. The instruction copies the contents of its source operand to its destination operand if a given condition is met. The instruction contains a condition (which is part of the opcode) and two operands, one of which can be a memory location. The translation of the instruction instance `cmovz ebx, eax` (using the Intel x86 architecture with the 64 bit extension) into RReil is shown in Fig. 3.2b). In order to illustrate the translation, we first detail the output of the GDSL decoder which is a value of the algebraic data type `insn` that is defined as follows:

```
type insn = #an x86 instruction
   CMOVZ of {opnd1: opnd, opnd2: opnd}
 | ... #other instruction definitions omitted
```

Thus, the `CMOVZ` constructor carries a record with two fields as payload. Both fields are

of type opnd and, for instance, carry a register or a memory location:

```
1 type opnd = #an x86 operand
2    REG of register
3  | MEM of memory
4  | ... #immediates, scaled operands and operands with offsets omitted
```

Note that all variants (here REG and MEM) implicitly contain information about the access size. In the example above, the instruction cmovz ebx, eax is represented by CMOVZ {opnd1 = REG EBX, opnd2 = REG EAX} where EAX has a size of 32-bits. The following section details helper functions that operate on opnd values.

3.3.2. Generating RReil Statements using GDSL Monadic Functions

Each semantic translator function generates a sequence of RReil statements. The sequence is stored inside the state of a monad. An RReil statement is added to the sequence by calling a GDSL monadic function which builds the abstract syntax tree of the statement. In order to explain the example in Fig. 3.2, we detail the GDSL functions for assignment, called mov, and conditional:

- `val mov sz dst src = ...`
  The mov function generates the RReil assignment statement `dst =:sz src` that copies the RReil expression src to the RReil variable dst.

- `val _if cond _then stmts = ...`
  This function generates the RReil statement `if (cond) { stmts } else {}`. The special mix-fix notation `_if cond _then stmts` is a call to a mix-fix function whose name `_if _then` is a sequence of identifiers that each commence with an underscore.

We further require the following functions that operate on x86 operands of type opnd. They are necessary to translate x86 registers, memory locations, or immediate values, that are encoded in the x86 operand, into RReil:

- `val sizeof x86-operand = ...`
  The sizeof function returns the size of an x86 operand in bits; here, `sizeof (REG EBX) = 32`.

- `val lval size x86-operand = ...`
  The lval function turns an x86 operand into an RReil left hand side expression, that is, either *var* or *[addr]*. Here, `lval 32 (REG EBX)` yields the RReil register *B* that contains the 32 bits of the Intel EBX register.

- `val rval size x86-operand = ...`
  The `rval` function turns an x86 operand into an RReil **expr**. In the example, `rval 32 (REG EAX)` yields the RReil register A.

- `val write size destination source = ...`
  The `write` function emits all statements necessary to write to an x86 operand. The operand is specified using the *destination* parameter; it is the return value of an associated call to `lval`. In Fig. 3.2b) lines 6 through 7 originate from the call to `write`.

Finally, the `mktemp` function is used to allocate a temporary variable.

3.3.3. The Translator

The translator function for `cmovz ebx, eax` is shown in Fig. 3.2a). The **do ... end** notation surrounding the function body is used to execute each of the enclosed monadic functions in turn. The decoded Intel instruction is passed-in using the *insn* parameter; the condition is determined by the caller depending on the actual mnemonic. The condition is a one-bit RReil expression. In the `cmovz ebx, eax` example, it is `ZF` which corresponds to the zero-flag.

The translation itself starts with a code block that is very common in instruction semantics: The operation's size is determined by looking at the size of one operand (line 2) and the respective operands are prepared for reading (using the `rval` monadic function) and writing (using the `lval` monadic function). Next, a new temporary RReil register is allocated and initialized to the current value of the destination operand (lines 7 and 8). This completes all preparations; the actual semantics of the instruction is implemented by the code lines 10 through 11. The condition is tested and, if it evaluates to *true*, the source operand is copied to the destination operand. It is important to note that the conditional is not evaluated at translation time, but that it is part of the emitted code. Finally, the (possibly) updated value of the temporary RReil register is written to the corresponding Intel register by code line 13.

One might think that the instruction pointlessly reads the source operand and writes the destination operand in case the condition evaluates to *false*. It is, however, necessary since the writeback can also cause further side effects that still need to occur, even if no data is copied. This is exemplified in Fig. 3.2b): since the instruction uses a 32 bit register in 64 bit mode, the upper 32 bits of the register are zeroed even if the lower 32 bits are unchanged (see line 7). The additional code is emitted by the write function.

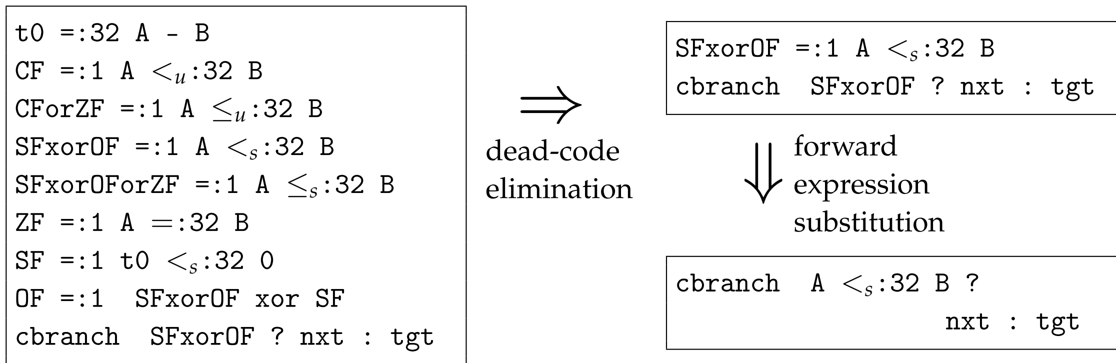


Figure 3.3.: Translation of the native Intel instructions `cmp eax, ebx; jl tgt` into RReil and applying optimizations. Here, `CForZF`, `SFxorOF`, `SFxorOForZF` are *virtual flags*, that is, translation-specific variables whose value reflect what their names suggest [48]. Note that this example is idealized since the removed flags may not actually be dead.

3.4. Optimizing the RReil Code

The design of RReil also allows for an effective optimization of the IR [48] which is illustrated in Fig. 3.3. The example shows the typical code bloat when translating two native instructions where the first sets many flags of which the second only evaluates one. Implementing these optimizations in GDSL is not only concise but also avoids the need to re-implement them in the individual analyses. The next sections consider two optimizations, both of which are implemented in the toolkit.

3.4.1. Liveness Analysis and Dead Code Elimination

The optimization strategy we implement is a dead-code elimination using a backwards analysis on the RReil code. To this end, we first need to obtain a set of live variables to start with. A simple approach assumes that all variables are live at the end of the block. This has the drawback that assignments to variables that are always overwritten in the succeeding blocks cannot be removed. We address this problem by refining the live-set for basic blocks that do not jump to computed addresses: Specifically, we infer the live variable set of the immediately succeeding blocks and use this live set as start set, thereby removing many more assignments to dead variables. We perform a true liveness analysis [47], that is, we refrain from marking a variable as live if it is used in the right-hand side of an assignment to a dead variable. For the body of while loops, however, this approach would require the calculation of a fixpoint. Since `while` loops are used rarely by our translator and since their bodies show little potential for optimization, a more conservative notion of liveness is used that does not require

a fixpoint computation. This approach marks a variable as live even if it used in an assignment to a dead variable. With this strategy, the dead code elimination takes linear time in the size of the basic block. See Fig. 3.3 for an example of how our liveness analysis helps in reducing the size of the RReil code.

3.4.2. Forward Expression Substitution

In addition to the liveness analysis, our analyzer also features a forward expression substitution pass [35]. This pass is responsible for the last step in Fig. 3.3 where the computation of `SFxorOF` is forward-substituted into the `cbranch` statement. While this seems to be a minor improvement on first sight, it actually reduces the complexity of subsequent analyses considerably. This is because without this transformation, an additional domain is required in the static analysis tool to recover the relation between the flag `SFxorOF` and the two registers `A` and `B`. Additionally, the forward substitution allows the removal of temporaries.

In the future, we plan to perform further optimizations. In particular, we plan better support for architectures like ARM where most instructions may be executed conditionally, depending on a processor flag. Compilers use this feature to translate small bodies of conditionals without jumps. Consider a conditional whose body translates to two native instructions $i_1;\ i_2$ that are executed if f holds. These are translated into the RReil statements

```
1  if (f) {
2    i1
3  }
4  if (f) {
5    i2
6  }
```

which ideally should be simplified as follows:

```
1  if (f) {
2    i1
3    i2
4  }
```

Without this optimization, a static analysis will compute a join of the unrelated states of the *then*- and *else*-branches of the first *if*-statement. The thereby incurred loss of precision is particularly problematic for the TSL approach since each instruction is executed on a single domain that, in general, will not be able to join two states without loss of precision.

3.5. Empirical Evaluation

We measured the impact of our dead-code elimination on a linear-sweep disassembly of standard Unix programs. Each basic block, that is, a sequence of Intel instructions up to the next jump, is translated into semantics. Table 3.1 presents our experimental results where the 'fac' column denotes the size of the RReil code (in 1000 lines of code, 'kloc') in relation to the native x86 disassembly ('nat. kloc'). Here, column 'translation' shows that, without optimizations, about six RReil statements are generated for each Intel instruction. The colums 'single', 'intra', and 'inter' show how the size of the RReil code reduces due to our optimizations and the time required to do so. Performing liveness analysis and dead code elimination on the semantics of a single instruction reduces the size by about 14% (column 'single'). Applying these optimizations on basic blocks reduces the size by about one third (column 'intra'). The 'inter' column shows the result of the optimizations as per Sect. 3.4.1: for basic blocks ending in a direct jump, the (one or two) blocks that are branched-to are translated and their set of live variables is computed. Using this refined liveness set, the dead code elimination removes between 40% and 60% of the RReil code relative to the non-optimized translation. Thus, with the information of the neighboring basic blocks, our RReil semantics is roughly 3 times larger than the x86 disassembly.

In order to compare our translation into RReil with the TSL approach [39] where a bespoke abstract transformer is generated for each native instruction, again consider column 'single' of Table 3.1. Since this column shows the reduction when considering the semantics of a single instruction, it provides an estimate of how many abstract transformers in a TSL translation a standard compiler can remove due to dead code elimination. While the TSL translations are optimized in other ways, it is questionable if this can rival the effect of removing not 14%, but around 50% of instructions, as our inter-basic block analysis does.

The GDSL compiler emits C code that closely resembles handwritten C programs. As a consequence, the resulting C code is easy to debug and allows the GDSL compiler to rely on the optimizations implemented by off-the-shelf C compilers. Indeed, the optimizations should be fast enough to re-apply them on-the-fly each time a basic block is analyzed. Even then, future work will address the elimination of bottlenecks in both, the GDSL compiler and optimizations written in GDSL.

Given these benefits, we hope that our open-source GDSL toolkit becomes an attractive front-end for any analysis targeting executable programs.

3.6. Future Work

Future work will extend our toolkit with decoders and translations for other architectures. While the toolkit currently already contains a partial ARMv7 decoder and semantics translator, it would be particularly interesting to mechanically translate the verified bit-level ARM semantics [23] into RReil. Moreover, given that an analysis that features a GDSL front-end can handle any architecture specified in GDSL, we hope for contributions from the community to further extend the range of architectures that GDSL offers. GDSL would also lend itself for defining semantics besides the RReil value semantics, namely energy or timing semantics.

In the long run, we hope that the GDSL toolkit will become the preferred choice for analyzing machine code, thereby replacing proprietary decoders (such as the popular xed2 decoder from Intel's PIN toolkit [63]) that are often equipped with a minimal, application-specific semantics covering only a few instructions.

prog.	nat.	translation			opt. single			
	kloc	**kloc**	**time**	**fac**	**kloc**	**time**	**red**	**fac**
bash	144	907	1.0s	6.3	778	5.1s	14%	5.4
cat	7	39	0.0s	5.9	34	0.2s	15%	5.0
echo	3	15	0.0s	5.6	13	0.1s	14%	4.8
less	21	152	0.1s	7.3	131	0.7s	14%	6.3
ls	15	106	0.1s	6.9	90	0.5s	16%	5.8
mkdir	7	45	0.0s	6.5	37	0.2s	16%	5.4
netstat	15	86	0.1s	5.6	75	0.4s	12%	4.9
ps	13	68	0.1s	5.3	57	0.4s	16%	4.5
pwd	3	19	0.0s	5.6	16	0.1s	14%	4.8
rm	8	47	0.0s	6.0	41	0.2s	14%	5.2
sed	9	54	0.1s	6.3	45	0.3s	16%	5.3
tar	50	317	0.3s	6.4	270	1.6s	15%	5.4
touch	8	47	0.0s	6.3	41	0.2s	14%	5.4
uname	3	15	0.0s	5.6	13	0.1s	14%	4.8
Xorg	346	2080	2.3s	6.0	1803	10.6s	13%	5.2
	opt. intra				**opt. inter**			
	kloc	**time**	**red**	**fac**	**kloc**	**time**	**red**	**fac**
bash	640	3.7s	30%	4.4	454	9.1s	50%	3.2
cat	28	0.2s	30%	4.1	21	0.4s	46%	3.2
echo	11	0.1s	29%	4.0	8	0.1s	46%	3.0
less	105	0.6s	31%	5.1	61	1.4s	60%	2.9
ls	66	0.4s	38%	4.3	49	1.0s	54%	3.2
mkdir	29	0.2s	35%	4.2	21	0.4s	53%	3.1
netstat	63	0.3s	26%	4.2	53	0.7s	39%	3.5
ps	45	0.3s	33%	3.5	40	0.6s	41%	3.1
pwd	14	0.1s	27%	4.1	11	0.2s	43%	3.2
rm	33	0.2s	30%	4.2	25	0.4s	47%	3.2
sed	37	0.2s	31%	4.3	28	0.5s	49%	3.2
tar	215	1.3s	32%	4.3	161	3.1s	49%	3.2
touch	31	0.2s	34%	4.1	23	0.5s	51%	3.1
uname	11	0.1s	28%	4.1	8	0.1s	45%	3.1
Xorg	1408	8.4s	32%	4.1	1067	20.9s	49%	3.1

Table 3.1.: Evaluating the reduction of the RReil code size due to dead-code optimization. The overall running time is the sum of the translation time plus the time for one of the optimizations. All measurements were obtained on an Intel Core i7 running at 3.40Ghz.

4. Verification of the Decoder and the Translator

Our current definition of the semantics translator has been written by hand in GDSL using the documentation offered by the chip manufacturer, e.g. Intel for the x86 architecture. The manufacturers usually offer a mixture of a textual description in combination with some more or less formally defined pseudo code to define the semantics of the processor instructions. As discussed in Sect. 2.4, there is a model of the ARMv7 architecture which has been formally proven; however, translating this model into GDSL is future work. Note that the definition of the translator is particularly error-prone because the description in the documentation is sometimes vague and can easily be misunderstood.

4.1. Automatic Generation of End-to-End Tests

A practical approach to finding and avoiding bugs in software is testing. A downside of testing is that tests need to be written and maintained. Plus, a manually crafted test can only detect a wrong implementation, but is unable to detect a mistake that originates from misunderstanding the original specification, i.e. the instruction set manual in this case. In addition, manually crafting tests for each machine instruction is a huge amount of work. In order to address these problems, we decided to automatically generate test cases for our decoding and translation pipeline. To this end, we came up with the following approach:

1. We start by generating an arbitrary machine instruction.
2. We decode and translate the instruction using GDSL.
3. We feed the resulting RReil code into an interpreter that executes it.
4. We execute the instruction on a physical processor.
5. We compare the effect of the interpretation and the execution on the physical processor.

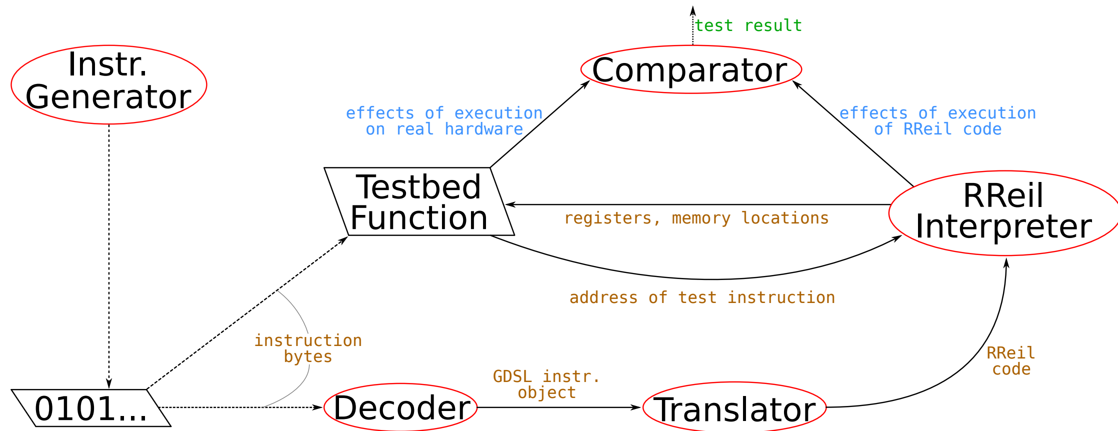


Figure 4.1.: Overview of the end-to-end GDSL test framework.

This approach allows us to to test all aspects of the decoder and the translator as only a valid decoding and translation will allow us to observe the same effects on both the physical hardware and the interpreter. Our testing tool is shipped with GDSL; however, since it contains architecture-specific parts, it currently only supports Intel x86. As part of our development roadmap, we are planning to port the tester to all our supported architectures as well as factor out architecture-specific code into sub-modules so that adding further architectures gets easy and straight forward.

An overview of our testing framework is given in Fig. 4.1. The following sections discuss the most important design decisions we made and challenges we faced during the implementation. The testing framework has initially been described in [32].

4.2. Generation of x86 Machine Instructions

The first step towards automatic testing is to obtain suitable machine instructions. As pointed out above, we currently only support the Intel x86 architecture. Generally, we considered the following approaches:

1. **Random bytes**: We could generate a random sequence of bytes and use it as input to the decoder.

2. **Rule-based generation**: We could use a set of rules derived from the architecture manual to generate valid machine instructions.

3. **Using exiting executables**: We could scan through existing executables and gather instructions from them.

For architectures that feature fixed-size instructions of small size, like ARM, the first approach is most thorough: one can generate and test every possible instruction. However, x86 instructions have a length of up to 15 bytes, making it impossible to systematically generate all instructions. A possible solution to this problem is using rules from the manual to generate valid instructions only while refraining from generating all possible immediate values within instructions (second approach). However, implementing such a generator is cumbersome and error-prone because of the size and complexity of the architecture (think of, for example, the rules regarding legitimate prefix combinations and orders) and its documentation. Thus, we have chosen a hybrid approach that takes the most important rules into account but does not guarantee to generate valid instructions only. For example, our generator generates prefix bytes only before the opcode, but it does not make sure that generated prefixes are in the correct order or are valid in combination with the given opcode.

The last approach, that is taking instructions from existing binaries, does not suffice on its own since it does not cover instructions that are not generated by standard compilers. It would, however, be an interesting addendum as it would make sure that our tests cover the most prominent instruction types and, thus, make translation bugs for these common cases less likely. Such a generator is currently not implemented in our testing framework.

4.3. Execution of the Generated Instruction

After generating an instruction, it needs to be executed on both an RReil interpreter and the physical hardware as a reference. To this end, we first decode and translate the instruction using GDSL. If the decoding fails, we consider the instruction as invalid and abort the test[1]. If the decoding and the semantics translation succeeds, the generated RReil code is passed to the RReil interpreter. In addition, we allocate memory for the dynamically generated *testbed function* that will later execute the instruction on the processor. Using this allocation we compute the memory address of the instruction which is used by the interpreter as contents of the `RIP` register (the program counter). During the execution of the RReil code by the interpreter, we record register and memory accesses. In case values are read from registers or memory, suitable random values are used as input and stored together with the access type (i.e. read or dereference access). Write accesses update the stored values to the values that are written. Using the data from the execution of the RReil code, we derive a set of registers and memory locations that are accessed during the execution of the instruction

[1] An alternative would be to execute the instruction on the processor while expecting an illegal instruction exception.

access type	recorded data	remark
read register	register name, initial value	initial value needs to be written to register before execution
write register	register name	register contents need to be compared after execution
read memory	memory address, initial value	memory needs to be mapped; initial value needs to be written to memory before execution
write memory	memory address	memory needs to be mapped; memory contents need to be compared after execution
jump target	target address	memory needs to be mapped; jump recorder code needs to be written to the mapped memory

Table 4.1.: Handling of accesses during RReil code interpretation.

on the processor. We use this data to initialize the read registers and map memory addresses from which the processor is expected to read, write, or execute. Memory regions that are used as jump targets additionally need to be initialized with machine code that records the realization of the jump and subsequently returns to the origin of the jump. Table 4.1 contains a summary of data we record during the interpretation of the RReil program and how we use that data later on.

Note that our approach relies on knowledge from the execution of the RReil code to prepare the environment for the execution of the instruction on the processor. This is problematic since the RReil code generation is under test and may, thus, contain bugs. In order to cope with this, we randomly select additional registers which take part in the test, i.e. are inspected after the test and are expected not to change. In case, for example, the RReil generator generates an empty RReil program it is, thus, likely that the test fails because one of these additional registers is modified by the instruction under test when executed on the physical hardware. In addition, note that if the instruction accesses additional memory it is likely that the corresponding address is not mapped which is again detected.

As an example, consider the instruction `add rbx, rax`. The instruction adds the 64-bit registers `rbx` and `rax`, saving the result to `rbx`. For this instruction, the tester generates the following output describing the data collected from the interpretation and the initialization of the read registers.

```
Read registers:
Register B: ffffffffffffffff
Register A: ffffffffffffffff
Register FLAGS: 0000000000000884
Written registers:
Register FLAGS: 00000000000008d5
Register B: ffffffffffffffff
Dereferenced registers: none
------------------
Register IP: 00007f06c1e2105c [defined:ffffffffffffffff]
Register FLAGS: 0000000000040084 [defined:0000000000244cd5]
Register A: 4b5d000137b47f92 [defined:ffffffffffffffff]
Register B: ec936e00e0ff4d52 [defined:ffffffffffffffff]
```

The read and written bits are given as a bit mask. Within the flags register, only certain bits (i.e. flags) are read and / or written. Note that the tester also stores a so called domain for each register; the domain stores which bits have a defined value. Only bits that have a defined value after an operation according to the reference manual need to be compared for equality at the end of the test.

4.4. Test Results and Error Conditions

There are a number of possible test outcomes that indicate different errors of the decoder or semantics translator. Note that the instruction is executed in a separate process in order to limit the effects its execution might have on the tester if the effects of the instruction are predicted wrongly (e.g., the instruction might overwrite arbitrary regions of memory). The error types and their causes are discussed in the following.

Decoding error The instruction under test cannot be decoded. Since the instructions generated by our simplified instruction generator are invalid with a probability of around 50% [32], we abort the test in case of a decoding error. In order to confirm the correctness of decoder, it would be helpful to check whether the instruction generates the correct exception if executed on the processor. However, this is, currently not implemented.

Interpreter error The RReil interpreter is unable to execute the RReil code produced by the GDSL translator. This indicates a bug in the semantics translation. For example, an interpreter error occurs if an undefined value is stored to memory or if selecting a branch of a conditional statement depends on an undefined value.

Signal `SIGSEGV` **during execution** The process executing the instruction under test on the physical hardware receives a *segmentation fault* signal. This signal indicates an invalid memory access. This is most likely caused by an invalid or incomplete semantics translation as the execution of the corresponding RReil code failed to predict an access to the memory address causing the signal.

Signal `SIGILL` **during execution** The process executing the instruction under test on the physical hardware receives an *illegal instruction* signal. This indicates an error in the decoder because an instruction has been successfully decoded by GDSL but has been rejected by the processor. In addition, illegal instruction signals can occur if an instruction requires a more recent hardware version or additional chip features.

Signal `SIGALARM` **during execution** The process executing the instruction under test on the physical hardware receives an *alarm* signal. This means the execution of the instruction has timed out. Even though unlikely, some instructions can take a long time or never terminate (e.g. a self-branch). We detect such situations by configuring a timeout for the execution of one second.

If none of the above errors occurs, the execution results are compared. If the values in the registers and memory locations match, the instruction passes the test.

5. Compiling GDSL to C

Advances in programming languages are difficult to transfer into an existing product as rewriting existing software using a modern language is usually not cost effective. Furthermore, since interfacing with modern languages is usually problematic, programmers fall back to using legacy programming languages for development: besides the technical challenges of marshaling data and transferring ownership of memory, there are practical maintenance problems in building, debugging, porting and profiling multi-language projects. An interesting alternative for reaping some benefits of higher-level languages is to develop a domain-specific language (DSL) that is translated into the legacy programming language [22]. One example are Cobol programs that nowadays feature embedded SQL statements that a pre-processor translates into the corresponding Cobol primitives. A general framework for creating such extensions is provided by the Xtext framework [7] that targets the Java programming language, thereby providing access to the garbage-collected heap and the introspection facilities of Java. In this chapter, we address the challenge of bridging a wider semantic gap, namely from the pure functional language GDSL to the C language so that we can neither rely on a garbage-collected heap nor on run-time type information.

Language features of GDSL such as boxed values, polymorphic functions, flexible records (where fields can be added to an existing record), monads, and closures thus have to be implemented explicitly using C data structures which generally clobber the logic of the generated code with boilerplate implementation details. The observation in this work is that these features are often not used to their full extent in which case they can be compiled into C code that resembles hand-written code. Based on this observation, we combined the following three optimizations into one highly efficient translation scheme.

5.0.1. Heap-Allocation and Avoidance of Garbage Collection

We solve the lack of a garbage-collected heap by equipping each translated DSL program with a micro-runtime that provides a heap. During the evaluation of a DSL function, objects may be allocated on this heap. The returned result may also contain pointers to the DSL heap. Once the C program has processed the result (possibly but not necessarily by copying the result to `malloc`'d memory regions), a function

`reset_heap()` can be called that discards all allocated data on the DSL heap. This operation can be as simple as setting the `free`-pointer to the beginning of the DSL heap and is thus a very cheap way of reclaiming the memory. Moreover, since no actual garbage collector is needed that traces the reachable set, the objects on the heap do not have to be tagged to distinguish pointers from data. Hence, an object stored on the heap has the same C type as when stored in a stack-allocated variable. As a consequence, code accessing the heap resembles hand-written C code. Moreover, the initialization of newly allocated objects is cheaper and the memory consumption is lower compared to garbage-collected languages that use tagged heap objects.

A consequence is that a long-running DSL program has to be broken down into a number of smaller, isolated tasks between which the heap is reclaimed so that the overall memory usage remains reasonable. These smaller tasks need to be designed so that they do not communicate heap-allocated data among one another. While this may seem as a major restriction, it fits many uses of DSLs [20, 24].

5.0.2. Unboxing of Polymorphic Values and Closures

GDSL allows for parametric polymorphism [10] where functions and data types can be defined that are agnostic to the data they operate on, such as a list reversal function. Translating such a function into a single C function requires that polymorphic values are represented uniformly, which usually means that they are represented by a pointer to the heap where the actual, variable-sized value resides. Values stored on the heap are called *boxed*. The efficiency of a program can be greatly improved by *unboxing* values that have a fixed type [37]. For instance, computing `x+(y+z)` without unboxing would allocate the intermediate result i of `y+z` on the heap before it is read from the heap to compute `x+`i. An unboxing optimization avoids this unnecessary allocation step.

The type information necessary to perform unboxing is usually obtained in the front-end through type checking/inference and must be retained up to the unboxing optimization pass which complicates the compiler [49]. Also, a polymorphic function is often only used in a restricted context. For instance, consider sorting a list of tuples `xs : (int*char) list` by the first component in ML:

```
1 let fun cmp (xKey, xVal) (yKey, yVal) = xKey <= yKey
2   in sort cmp xs
```

Here, `cmp` is polymorphic in the second tuple element which therefore cannot be unboxed even if it is only used for the type `char`. Indeed, it has been proposed to specialize polymorphic functions for the sake of more unboxing [8]. Our type-refinement optimization unboxes all variables that are used monomorphically such

as `cmp` above. For example, if all the lists in a program range over the same type, the payload of the list constructor `Cons` is unboxed. Moreover, it does not require any type information from the front-end and thus also works for dynamically typed DSLs.

Since the monomorphic types used in our optimization can be different from the types inferred in the front-end, we are able to infer additional information on function types and record types. Specifically, a function type carries a tag that indicates if it represents only trivial closures, that is, closures that only contain a function pointer but no arguments. We unbox these closures by passing the address of the function instead of passing the pointer to the heap-allocated closure. With respect to record types, note that our language features flexible records where fields can be added to and extracted from an existing record. These records are realized in the runtime by heap-allocating a linked list of field/value pairs. For readability and for the sake of calling functions with record arguments from C, it is desirable to use C `structs` instead. To this end, the optimization infers a set of fields (and their types) and whether these fields always exist. Any variable of record type whose fields always exist is turned into a C `struct`.

5.0.3. Transformation of Monadic Functions

A monadic function f is of type $\alpha_1 \times \ldots \times \alpha_n \to M\ \alpha$ where $M\ \alpha$ is itself an abbreviation for $s \to s \times \alpha$, that is, a function from state s to a new state s and the result α. Thus, the evaluation of `f a1 ... an` yields yet another function which is a closure. Since creating, returning and invoking a closure for each monadic function is hard to read and expensive, we transform the type of f to $s \times \alpha_1 \times \ldots \times \alpha_n \to s \times \alpha$ if all of its call sites are monadic, that is, they all supply $\alpha_1 \ldots \alpha_n$ as well as s. Due to this transformation, most monadic functions are translated into normal C functions and many closures are avoided.

In summary, the presented translation scheme contributes the following novelties:

- It illustrates how structure preserving translation can turn a high-level DSL language into readable C code (or other legacy languages), thereby making it possible to re-use existing debuggers, profilers and other tools for C.
- A front-end independent unboxing optimization is proposed and evaluated that is able to unbox closures and flexible records.
- Our evaluation shows that the generated code can be optimized by off-the-shelf C compilers, yielding code that is as fast as a hand-written C library.

GDSL serves as a reference language throughout the chapter. During the translation, GDSL is first transformed into an intermediate language called Core. Core is a

functional language similar to GDSL but without the domain-specific decoder syntax. Core is then translated into an imperative language Imp. Imp is designed to be suitable for our optimizations and also to be easily translatable into C. The C code can then be handed to an off-the-shelf C compiler.

The chapter is organized as follows: Section 5.1 presents the lowering of GDSL to Core. Section 5.2 illustrates how Core translates to Imp. Section 5.3 discusses the optimization of Imp. Our implementation, described in Sect. 5.4, is evaluated in Sect. 5.5. Related work is presented in Sect. 5.6 before Sect. 5.7 concludes.

5.1. Lowering GDSL to Core

The GDSL language described in the previous chapters is a domain specific language that offers a special syntax geared towards the specification of instruction decoders. Here, we introduce a language called Core which corresponds to GDSL without the decoder syntax. A GDSL program can easily be translated into Core. To understand this, consider the example decoder for Intel x86 instructions in Fig. 5.1 on page 44.

In the figure, we declare an algebraic data type to represent instructions. Then we define two decoders `decode` and `/r` which use the special pattern syntax in `[...]` to indicate that they read from the input byte stream. This pattern syntax is desugared to the Core code in Figure 5.2 on page 45. Here, the `consume8` function is used to extract a byte from the input stream over which a `case`-statement dispatches. The pattern `0x00` is translated into bit-pattern `'00000000'` which is a built-in data type containing the bit-string and its size. The use of the sub-decoder `/r` in both patterns is translated by inlining the body of the sub-decoder starting from lines 5 and 15, before the right-hand-side of the decode rules are evaluated. Note that because the patterns of sub-decoders can contain constant parts, their patterns need to be added to the patterns of the calling decoders and their bodies need to be prepended to the right-hand-side of the matching decode rules. Inside the code of `/r`, another byte is read and its bits are extracted using a built-in function `slice`.

The desugared code of `decode` and `/r` in Fig. 5.2 and the remaining code in Fig. 5.1 now only use language constructs from Core whose grammar is defined in Fig. 5.6 on page 48.

The translation scheme and the optimizations developed in this chapter allow a translation of Core into the C code shown in Fig. 5.4 and 5.5 that is sufficiently similar to allow the user of the DSL to debug or profile the program at the C level (note that we have added some additional comments in the listing for clarification; manually added comments are in purple). Note that this close correspondence is only possible if the program does not exploit the whole flexibility of the source language (which is usually

```
1  #declare a datatype for instructions
2  type insn = ADD of {opnd1:opnd,opnd2:opnd}
3            | INC of {opnd1:opnd} #increment
4
5  #define decoding rules for streams starting with 0x00 and 0x01; the
6  #latter queries the $opndsz flag of the monadic state
7  val decode [0x00 /r] = binop ADD r/m8 r8
8  val decode [0x01 /r] = do
9     opndsz <- query $opndsz;
10    if opndsz then binop ADD r/m16 r16
11              else binop ADD r/m32 r32
12 end
13
14 #a sub-decoder reading one byte and storing 2 (mod), 3 (reg/opcode)
15 #and 3 (rm) bits of it in the monadic state
16 val /r ['mod:2 reg/opcode:3 rm:3'] =
17   update @{mod=mod, reg/opcode=reg/opcode, rm=rm}
18
19 #a function that generates two-argument instructions; giveOpX are
20 #monadic functions, cons is an instruction constructor
21 val binop cons giveOp1 giveOp2 = do
22   op1 <- giveOp1; op2 <- giveOp2;
23   return (cons {opnd1=op1, opnd2=op2})
24 end
25
26 #a typical function passed as argument to binop: it queries the value
27 #of the reg/opcode field in the monadic state and returns an AST in
28 #the form of an algebraic data type, here a register REG XX
29 val r16 = do
30    r <- query $reg/opcode;
31    case r of
32       '000': return (REG AX)
33     | '001': return (REG CX)
34     | '010': return (REG DX)
35     | '011': return (REG BX)
36    end #other cases omitted
37 end
```

Figure 5.1.: A minimal decoder for Intel x86 instructions.

```
val decode = do
  tok <- consume8; #read byte from stream
  case tok of #make pattern matching explicit
    '00000000' : do
      # inlined code of /r
      tok <- consume8;
      rm <- slice tok 0 3;
      reg/opcode <- slice tok 3 3;
      mod <- slice tok 6 2;
      update @{mod=mod, reg/opcode=reg/opcode, rm=rm};
      # end of inlined code of /r
      binop ADD r/m8 r8
    end
   | '00000001' : do
      # inlined code of /r
      tok <- consume8;
      rm <- slice tok 0 3;
      # ... (remaining code of /r omitted)
      opndsz <- query $opndsz;
      if opndsz
        then binop ADD r/m16 r16
        else binop ADD r/m32 r32
    end
  end
end
```

Figure 5.2.: Desugaring the decoders to Core. We omit code handling pattern match failures.

```
val decode = do
  opndsz <- query $opndsz;
  if opndsz
    then binop ADD r/m16 r16
    else binop ADD r/m32 r32
end
```

Figure 5.3.: Example Core code which is based on lines 19–22 from Fig. 5.2.

```
typedef struct { /* struct for parameter of ADD */
  obj_t opnd1;
  obj_t opnd2;
} struct1_t;

typedef struct { /* a variant of datatype insn */
  int_t tag;
  struct1_t payload;
} con_struct1_t;

static inline /* constructor function ADD */
  obj_t constructor_ADD(struct1_t arg_of_ADD) {
  return alloc_con_struct1((con_struct1_t) {1/* ADD */, arg_of_ADD});
}

obj_t decode() { /* decode */
  int_t tok, rm, reg_slash_opcode, mod;
  tok = consume8();
  switch (slice(tok, 0, 8)) {
    case 0: /* '00000000' */ {
      /* start of sub-decoder /r */
      tok = consume8(); /* get the next token */
      rm = slice(tok, 0, 3); /* cut out lower 3 bits */
      reg_slash_opcode = slice(tok, 3, 3);
      mod = slice(tok, 6, 2);
      state.mod = mod;
      state.reg_slash_opcode = reg_slash_opcode;
      state.rm = rm;
      /* end of sub-decoder /r */
       return binop(constructor_ADD, r_slash_m8, r8);
      };
      break;
    case 1: /* '00000001' */ {
      /* ... (inlined sub-decoder /r omitted) */
      /* query mondic state */
      int_t opndsz = state.opndsz;
      if (opndsz) return binop(constructor_ADD, r_slash_m16, r16);
```

Figure 5.4.: The C code of the decoders (part 1). Some lines and variable declarations are rearranged for presentational purposes.

```
38        else return binop(constructor_ADD, r_slash_m32, r32);
39      };
40      break;
41    };
42  }
43
44  static obj_t constructor_REG(int_t arg_of_REG) {
45    return alloc_con_int((con_int_t) {3/* REG */, arg_of_REG});
46  }
47
48  static obj_t r16() { /* r16 */
49    int_t r = state.reg_slash_opcode;
50    switch (slice(r, 0, 3)) {
51      case 0: /* '000' */ {
52         return constructor_REG(CON_AX); };
53         break;
54      case 1: /* '001' */ {
55         return constructor_REG(CON_CX); };
56         break;
57      case 2: /* '010' */ {
58         return constructor_REG(CON_DX); };
59         break;
60      case 3: /* '011' */ {
61         return constructor_REG(CON_BX); };
62         break;
63    };
64  }
65
66  /* binop expects a pointer to the constructor function as parameter */
67  static obj_t binop(obj_t (*cons)(struct1_t),
68      obj_t (*giveOp1)(), obj_t (*giveOp2)()) {
69    obj_t op1 = giveOp1();
70    obj_t op2 = giveOp2();
71    return /* build parameter, call constructor */
72      cons((struct1_t){.opnd1=op1,.opnd2=op2});
73  }
```

Figure 5.5.: The C code of the decoders (part 2). Some lines and variable declarations are rearranged for presentational purposes.

```
Core    ::= Func*
Func    ::= val id id* = Expr
Action  ::= id ← Expr ; Action
          | Expr ; Action
          | Expr
Cases   ::= Pat : Expr ; Cases | ε
Pat     ::= ' ( . | 0 | 1 )* ' | num
          | con id | con
Expr    ::= let Func* in Expr end
          | if Expr then Expr else Expr
          | case Expr of Cases end
          | Expr Expr+
          | { ( field = Expr )* }
          | @ { ( field = Expr )* }
          | $field
          | query Expr | update Expr
          | do Action end
          | " string " | ' ( 0 | 1 )* ' | num
          | con | id
```

Figure 5.6.: The input language Core.

the case). For instance, none of the arguments to `binop` is a partially applied function. Due to this, the address of a C function can be passed as argument as done in line 30, 37, and 38, rather than a pointer to a heap-allocated closure. Also, the arguments of the `REG` constructor in lines 51-61 are preprocessor-defined constants instead of pointers to heap allocated constructors which would be required if one of the registers had an argument. In order to apply these ideas we translate Core to an imperative language Imp.

5.2. Conversion to an Imperative Language

This section details the conversion from the functional language Core to a language we call Imp whose grammar is given in Fig. 5.7. Imp is imperative in that the body of a function is a **Block** that contains a sequence of statements **Stmt**. Imp is meant to be lowered directly to C. As such, record operations are no longer first class citizens but expressions that require a record as argument. Moreover, Core makes all allocations

$$
\begin{array}{lll}
\textbf{Imp} & ::= & \textbf{TLDecl}^{*} \\
\textbf{TLDecl} & ::= & \underline{id}\ \underline{(}\ \textbf{Decl}^{*}\ \underline{)}\ \underline{:}\ \textbf{Type}\ =\ \underline{id}\ \underline{where}\ \textbf{Block} \\
 & | & \underline{constructor}\ \underline{id}\ \underline{(}\ \textbf{Decl}\ \underline{)} \\
 & | & \underline{closure}\ \underline{cid}\ \underline{[}\ \textbf{Decl}^{*}\ \underline{]}\ \underline{(}\ \textbf{Decl}^{*}\ \underline{)}\ \underline{:}\ \textbf{Type}\ \underline{\rightarrow}\ \underline{id} \\
\textbf{Decl} & ::= & \textbf{Type}\ \underline{id} \\
\textbf{Stmt} & ::= & (\underline{id}\ \underline{=})?\ \textbf{Expr}\ \underline{;} \\
 & | & \underline{if}\ \textbf{Expr}\ \underline{then}\ \textbf{Block}\ \underline{else}\ \textbf{Block}\ \underline{;} \\
 & | & \underline{case}\ \textbf{Expr}\ \underline{of}\ \textbf{Cases}\ \underline{;} \\
\textbf{Block} & ::= & (\ \textbf{Decl}\ \underline{;}\ )^{*}\ \textbf{Stmt}^{*} \\
\textbf{Cases} & ::= & \underline{num}^{+}\ \underline{:}\ \textbf{Block}\ \underline{;}\ \textbf{Cases}\ \mid\ \varepsilon \\
\textbf{Expr} & ::= & \underline{id}\ \mid\ \underline{num}\ \mid\ \underline{''}\,string\,\underline{''} \\
 & | & \textbf{Expr}\ \underline{(}\ \textbf{Expr}^{*}\ \underline{)} \\
 & | & \underline{*}\ \textbf{Expr}\ \underline{(}\ \textbf{Expr}^{+}\ \underline{)} \\
 & | & \underline{\{}\ (\ \underline{field}\ \underline{=}\ \textbf{Expr}\ )^{*}\ \underline{\}} \\
 & | & \underline{@}\ \underline{\{}\ (\ \underline{field}\ \underline{=}\ \textbf{Expr}\ )^{*}\ \underline{\}}\ \textbf{Expr} \\
 & | & \underline{\$}\ \underline{field}\ \textbf{Expr} \\
 & | & \underline{box}\ \underline{(}\ \textbf{Type}\ \underline{,}\ \textbf{Expr}\ \underline{)} \\
 & | & \underline{unbox}\ \underline{(}\ \textbf{Type}\ \underline{,}\ \textbf{Expr}\ \underline{)} \\
 & | & \underline{gen\text{-}closure}\ \underline{cid}\ \underline{[}\ \textbf{Expr}^{*}\ \underline{]} \\
 & | & \underline{\lambda s \rightarrow}\ \textbf{Block}\ \textbf{Expr} \\
 & | & \underline{exec}\ \textbf{Expr}
\end{array}
$$

Figure 5.7.: The intermediate language Imp. Note that the non-terminal **Type** is defined in Fig. 5.18.

```
1 val f x =
2   let
3     val g y = x + y
4   in
5     g
6   end
7 val six = (f 1) 5
8 val seven = six + 1
```

Figure 5.8.: A Core example program that uses a closure.

```
1 closure g_cl[obj x](int y): obj -> g
2 g(int x int y) : int = gRes where
3   gRes = +(x y);
4 f(obj x) : obj = fRes where
5   fRes = gen-closure g_cl[x];
6 six() : int = sixRes where
7    sixRes = *f(1) (5);
8 seven() : obj = sevenRes where
9    sevenRes = +(six() 1);
```

Figure 5.9.: The optimized Imp program generated from the code in Fig. 5.8.

on the heap explicit. For values, $box(t, e)$ returns a pointer to a freshly allocated heap region containing e while $unbox(t, p)$ returns the value pointed to by p. In both cases, the type $t \in \underline{int}, \underline{obj}, \ldots$, determines the size of the object. A function f is boxed by creating a closure on the heap using $\textit{gen-closure}\ f_{cl}[a_1 \ldots a_k]$ and unboxed by invoking the closure using $*p(a_{k+1} \ldots a_n)$ (note the star: a normal function call is written $p(\ldots)$). Here, a_i with $i \leq k$ is a value from the environment in which f was defined. We illustrate this using the Core program in Fig. 5.8. The corresponding Imp code is shown in Fig. 5.9. Note that the optimized code is shown for better readability. For example, our unboxing optimization described in Sect. 5.3.3 has been applied; therefore, only the closure itself has to live on the heap while all integers are passed by value. After introducing the details of the translation scheme, we will discuss the unoptimized output of the translator.

```
typedef struct {
  int_t (*func)(obj_t, int_t);
  int_t arg1;
} closure_t;

static int_t g(int_t x, int_t y) {
  return x + y;
}

static int_t g_closure(obj_t v, int_t y) {
  closure_t* c = (closure_t*) v;
  return g(c->arg1, y);
}

static obj_t gen_closure(int_t (*func)(obj_t, int_t), int_t arg1) {
  closure_t* closure = (closure_t*) alloc(sizeof(closure_t));
  *closure = (closure_t){func, arg1};
  return (obj_t) closure;
}

static obj_t f(int_t x) {
  return gen_closure(&g_closure, x);
}

static int_t invoke_closure(obj_t closure, int_t arg1) {
  return ((struct {int_t (*func)(obj_t, int_t);}*) closure)
    ->func(closure, arg1);
}

static int_t six() {
  return invoke_closure(f(1), 5);
}

static int_t seven() {
  return six() + 1;
}
```

Figure 5.10.: Simplified C code for a Core example that uses closures.

The Core program defines a function f that locally defines g. Since g refers to the variable x from the environment, the Imp program defines g to take x in addition to its parameter y. The value g returned by f in the Core program is translated by computing a closure that contains a pointer to g_{cl} as well as x where g_{cl} is a special declaration that is later translated to code invoking g with the arguments in the passed-in closure. Specifically, the call `f(1)` in line 7 of the Imp program returns an address of a closure on the heap, say a_{cl}, that contains a pointer to g_{cl} and an integer 1. The invocation $*a_{cl}$ `(5)` calls g_{cl} and passes a_{cl} and the argument `5` to it. The C code emitted for g_{cl} in line 1 in Fig. 5.9 then invokes g with the argument 1 from the closure and the actual argument 5.

The corresponding C code can be seen in Fig. 5.10. The function `g` is defined in line 6 and invoked in line 12 by the invocation function `g_closure()` which extracts additional parameters from the closure and uses them to call `g`. The function `f` generates the closure in line 22 by calling the closure generator function `gen_closure`. Note that `f` provides the argument `x` from the context in which the closure is constructed. The second argument to `g`, in contrast, is provided at the closure invocation site in line 31.

The translation scheme for all Core constructs (except `case`) is formalized in Fig. 5.13, 5.14, and 5.15 which are discussed in the following. Note that we assume all identifiers to be unique; this can easily be ensured by renaming shadowing declarations. There is a schema $[\![\cdot]\!]^C$ for translating a list of declarations, for functions and contants $[\![\cdot]\!]^F$, for blocks $[\![\cdot]\!]^B_\rho$, for expressions $[\![\cdot]\!]^E$ and for monadic sequences $[\![\cdot]\!]^S$, some of which add a top-level declaration specified using a call to the **declare** primitive. A Core program is a list of declarations d that is translated by calling $[\![d]\!]^C$; the result of the translation is returned as side effect. $[\![\cdot]\!]^C$ branches to $[\![\cdot]\!]^F$ in order to translate functions and constants; here, the translation of a function f creates a closure declaration f_{cl} and requires that all occurrences of f in the generated program are replaced by the expression that generates a closure of f. This is done by declaring the closure f_{cl} and generating a substitution. The substitution is collected by $[\![\cdot]\!]^C$ and applied after translating all functions on the same level, thereby allowing all functions within one declaration list to access each other. Constants are represented by functions in Core; however, they do not require a closure in Imp. Instead, each usage site of a constant that is computed from the Core expression e is replaced by a call to a function that computes the value of the constant from the free variables in e.

$$
\begin{aligned}
&\mathit{applySubst}^{\mathbf{B}} : (\underline{id} \rightarrow \mathbf{Expr}) \times \mathbf{Block} \rightarrow \mathbf{Block}\\
&\mathit{applySubst}^{\mathbf{B}}(\sigma,\ \vec{decls}\ stmt_1\ \ldots\ stmt_n) =\\
&\quad \mathit{return}\ \vec{decls}\ \mathit{applySubst}^{\mathbf{S}}(\sigma,\ stmt_1)\ \ldots\ \mathit{applySubst}^{\mathbf{S}}(\sigma,\ stmt_n));
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{applySubst}^{\mathbf{S}} : (\underline{id} \rightarrow \mathbf{Expr}) \times \mathbf{Stmt} \rightarrow \mathbf{Stmt}\\
&\mathit{applySubst}^{\mathbf{S}}(\sigma,\ id\ \underline{=}\ expr) =\\
&\quad \mathit{return}\ id\ \underline{=}\ \mathit{applySubst}^{\mathbf{E}}(\sigma,\ expr)
\end{aligned}
$$

$$
\mathit{applySubst}^{\mathbf{S}}(\sigma,\ stmt) = \ \ldots \textbf{(* apply other kind of statement *)}
$$

$$
\begin{aligned}
&\mathit{applySubst}^{\mathbf{E}} : (\underline{id} \rightarrow \mathbf{Expr}) \times \mathbf{Expr} \rightarrow \mathbf{Expr}\\
&\mathit{applySubst}^{\mathbf{E}}(\sigma,\ id) =\\
&\quad \mathit{return} \begin{cases} id & id \notin \mathrm{dom}(\sigma)\\ \sigma(id) & \mathit{otherwise} \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{applySubst}^{\mathbf{E}}(\sigma,\ e\,\underline{(}\,e_1 \ldots e_n\,\underline{)}\,) =\\
&\quad \mathit{return}\ \mathit{applySubst}^{\mathbf{E}}(\sigma,\ e)\,\underline{(}\,\mathit{applySubst}^{\mathbf{E}}(\sigma,\ e_1) \ldots \mathit{applySubst}^{\mathbf{E}}(\sigma,\ e_n)\,\underline{)}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{applySubst}^{\mathbf{E}}(\sigma,\ \lambda id \mapsto block\ expr) =\\
&\quad \mathit{return}\ \lambda id \mapsto \mathit{applySubst}^{\mathbf{B}}(\sigma,\ block)\ \mathit{applySubst}^{\mathbf{E}}(\sigma,\ expr)
\end{aligned}
$$

$$
\mathit{applySubst}^{\mathbf{E}}(\sigma,\ expr) = \ \ldots \textbf{(* apply other kind of expression *)}
$$

$$
\begin{aligned}
&\mathit{applySubst}^{\mathbf{C}} : (\underline{id} \rightarrow \mathbf{Expr}) \times \mathbf{Cases} \rightarrow \mathbf{Cases}\\
&\mathit{applySubst}^{\mathbf{C}}(\sigma,\ expr) = \ \ldots \textbf{(* apply to cases *)}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{applySubst}^{\mathbf{T}} : (\underline{id} \rightarrow \mathbf{Expr}) \times \mathbf{TLDecl} \rightarrow \mathbf{TLDecl}\\
&\mathit{applySubst}^{\mathbf{T}}(\sigma,\ id\ \vec{params}\ \underline{:}\ type = f\ \underline{where}\ block) =\\
&\quad \mathit{return}\ \mathit{applySubst}^{\mathbf{B}}(\sigma,\ block)
\end{aligned}
$$

Figure 5.11.: Application of substitutions on an Imp AST node.

The translation scheme uses the two functions $\mathit{applySubst}^{\mathbf{B}}$ and $\mathit{applySubst}^{\mathbf{E}}$ to substitute function calls with calls to newly generated closures and uses of constants with calls to the functions that compute their values. The functions expect an Imp AST node as second parameter. The most relevant part of its implementation can be seen in Fig. 5.11. The $\mathit{applySubst}^{*}$ functions recursively traverse the argument node. Whenever an id is encountered which is in the domain of the substitution (first argument), the substitution is applied. Note that shadowing of declarations is not possible by virtue of requiring that all variables have different names. The function returns the new Imp AST with the substituted values in place.

Consider again the Core code in Fig. 5.8. The result of applying the translation scheme

```
 9  closure g_cl[obj x](obj y): obj -> g
10  g(obj x obj y) : obj = gRes where
11    gRes = box(int, +(unbox(int, x), unbox(int, y)));
12  closure f_cl[](obj x): obj -> f
13  f(obj x) : obj = fRes where
14    fRes = gen-closure g_cl[x];
15  six() : obj = sixRes where
16     sixRes = *(*(gen-closure f_cl[](box(int, 1))))(box(int, 5));
17  seven() : obj = sevenRes where
18     sevenRes = box(int, +(unbox(int, six()), unbox(int, box(int, 1))));
```

Figure 5.12.: The non-optimized Imp program generated from the code in Fig. 5.8 using the translation scheme in Fig. 5.13, 5.14, and 5.15.

can be seen in Fig. 5.12. The translation commences with the first top level declaration, i.e. the declaration of `f`. Here, it continues with the declaration of `g` which is reached via the translation scheme for `let` expressions. The function translation scheme declares a closure `closure g_cl[obj x](obj y): obj -> g` and returns a tuple consisting of the translated body of `g` as first element and the substitution [`g/gen-closure g_cl[x]`] as second element. The function `g` is subsequently declared (the call to *applySubst*$^{\mathbf{T}}$ does not change its body) and the substitution is returned to the `let` translator. Here, the body of `f` is first translated to code that returns `g`. Applying the substitution [`g/gen-closure g_cl[x]`] then yields the code in line 14. This concludes the `let` translation; thus, the function translation of `f` is finished by declaring the closure for `f` (line 12) and returning the function `f` together with the substitution [`f/gen-closure f_cl[]`]. Back on the top level of the translation, the translation continues with the functions `six` and `seven`. Both are functions taking zero arguments; thus, the constant translator is used here. As a result, no closure declarations are generated. The constant generator generates the substitutions [`six/six()`] and [`seven/seven()`], respectively. Before the bodies of `f`, `six`, and `seven` are emitted, the substitutions from the sub-translators are applied, replacing `f` with `gen-closure f_cl[]` (see line 16) and `six` with `six()` (see line 18).

The schema for translating e into a basic block $[\![e]\!]^B_v$ takes an additional variable v to which the result of evaluating e is assigned to. The translation of an expression $[\![e]\!]^E$, in turn, returns a tuple $\langle \vec{s}, e' \rangle$ where $\vec{s}$ is a sequence of statements that need to be executed in order to compute e whose result is given by the Imp-expression e'. As an example, consider applying the translation of the conditional in Fig 5.13 on lines 20–22 of Fig. 5.2.

The translation first computes the code of the two branches (given by $[\![e_t]\!]^B_{ite}$ and $[\![e_e]\!]^B_{ite}$) so that the result is stored in the freshly created variable *ite*. Based on these two blocks, the tuple $\langle s,\ ite\rangle$ is returned where s is the single Imp-statement `if opndz then ite = ...; else ite = ...;`.

Recall that the record selector $*foo* in Imp requires an argument, e.g. $*foo r*, so that it can be emitted as the C code `r.foo`. The translation therefore replaced $*foo* with a closure of *select-foo* which is a newly generated function that extracts this field from a record. Creating named functions instead of the traditional approach of using an anonymous function $\lambda r \rightarrow \$foo\ r$ [31] has the advantage that a duplicate generation of *select-foo* is easily avoided by checking whether a function with that name already exists. Moreover, the resulting C code is easier to read. One speciality is the translation of algebraic data types: a constructor that takes no arguments is translated as a pointer to an integer that stores a tag identifying the represented variant. A constructor that does take an argument is translated into a special top-level *constructor* function that heap-allocates an integer for its tag followed by the payload.

translation of a declaration list: translate functions and constants, declare them as new top-level entities, and return substitutions for accesses to them

$[\![\underline{val}\ f_1\ \vec{x_1}\ \underline{=}\ e_1\ \ldots\ \underline{val}\ f_m\ \vec{x_m}\ \underline{=}\ e_m]\!]^C = \sigma$

$\quad$ *let fs* $= \emptyset$ *and* $\sigma = \emptyset$;

$\quad$ *for* $\langle f\ \underline{(}\ \vec{x}\ \underline{)}\ \underline{:}\ \underline{obj}\ \underline{=}\ e\ \underline{where}\ b,\ \sigma' \rangle \in \{ [\![\underline{val}\ f_1\ \vec{x_1}\ \underline{=}\ e_1]\!]^F,\ \ldots,\ [\![\underline{val}\ f_m\ \vec{x_m}\ \underline{=}\ e_m]\!]^F \}$:

$\quad\quad \sigma = \sigma \circ \sigma'$; $fs = fs \cup \{ f\ \underline{(}\ \vec{x}\ \underline{)}\ \underline{:}\ \underline{obj}\ \underline{=}\ e\ \underline{where}\ b \}$;

$\quad$ *for* $f\ \underline{(}\ \vec{x}\ \underline{)}\ \underline{:}\ \underline{obj}\ \underline{=}\ e\ \underline{where}\ b \in fs$:

$\quad\quad$ **declare** $applySubst^{\mathbf{T}}(\sigma,\ f\ \underline{(}\ \vec{x}\ \underline{)}\ \underline{:}\ \underline{obj}\ \underline{=}\ e\ \underline{where}\ b)$

function translation: explicitly add environment variables c_i to parameter list, translate body, declare closure *cls*, generate substitution of f with new instance of *cls*

$[\![\underline{val}\ f\ x_1 \ldots x_n\ \underline{=}\ e]\!]^F = \langle f(\underline{obj}\ c_1 \ldots \underline{obj}\ c_k\ \underline{obj}\ x_1 \ldots \underline{obj}\ x_n)\ \underline{:}\ \underline{obj}\ \underline{=}\ f_r\ \underline{where}\ [\![e]\!]^B_{f_r},$
$\quad\quad [f / gen\text{-}closure\ f_{cl}[c_1 \ldots c_k]] \rangle$

$\quad$ **declare** $closure\ f_{cl}\ \underline{[}\ \underline{obj}\ c_1 \ldots \underline{obj}\ c_k\ \underline{]}\ \underline{(}\ \underline{obj}\ x_1 \ldots \underline{obj}\ x_n\ \underline{)}\ \underline{:}\ \underline{obj}\ \underline{\rightarrow}\ f\ where\ c_i \in free(e)$

contant translation: a Core constant is a function without arguments; we generate a function that computes the value of the constant from the free variables

$[\![\underline{val}\ f\ \underline{=}\ e]\!]^F =$

$\quad \langle f(\underline{obj}\ c_1 \ldots \underline{obj}\ c_k) : \underline{obj}\ \underline{=}\ f_r\ \underline{where}\ [\![e]\!]^B_{f_r},\ [f / f\underline{(}c_1 \ldots c_k\underline{)}] \rangle,\ c_i \in free(e)$

block translation: translate expression e into block, assigning the result of e to x

$[\![e]\!]^B_x = \underline{obj}\ v_1\ \underline{;}\ \ldots\ \underline{obj}\ v_n\ \underline{;}\ \vec{s}\ x = e'\ \underline{;}\ where\ \langle \vec{s},\ e' \rangle = [\![e]\!]^E$ and $\{v_1, \ldots v_n\} = free(e)$

let translation: evaluate b, assign result to x, evaluate the body e

$[\![\underline{let}\ decls\ \underline{in}\ e\ \underline{end}]\!]^E = \langle applySubst^{\mathbf{B}}(\sigma,\ \vec{s}_e),\ applySubst^{\mathbf{E}}(\sigma,\ e') \rangle\ where$

$\quad \sigma = [\![decls]\!]^C$ and $\langle \vec{s}_e,\ e' \rangle = [\![e]\!]^E$

if translation: evaluate condition e_c by executing $\vec{s}_c$, fetch result from heap, compute value of then or else branch into the fresh variable *ite*, return *ite* as result

$[\![\underline{if}\ e_c\ \underline{then}\ e_t\ \underline{else}\ e_e]\!]^E =$

$\quad \langle \vec{s}_c\ \underline{if}\ \underline{unbox}\ \underline{(}\ \underline{int}\ e'_c\ \underline{)}\ \underline{then}\ [\![e_t]\!]^B_{ite}\ \underline{else}\ [\![e_e]\!]^B_{ite}\underline{;},\ ite \rangle \wedge \langle \vec{s}_c, e'_c \rangle = [\![e_c]\!]^E \qquad ite$ fresh

closure invocation: evaluate function expression and arguments; invoke function closure

$[\![e_0\ e_1 \ldots e_n]\!]^E = \langle \vec{s}_0 \ldots \vec{s}_n,\ \underline{*}e'_0\underline{(}\,e'_1 \ldots e'_n\underline{)} \rangle\ where\ \langle \vec{s}_i,\ e'_i \rangle = [\![e_i]\!]^E$ for $i = 0, n$

Figure 5.13.: Translation scheme from Core to Imp (part 1).

record constant: evaluate field expressions e_i before returning the record value $\underline{\{}\, f_0 \,\underline{=}\, e'_0 \ldots f_n \,\underline{=}\, e'_n \underline{\}}$, set fields f_i to resulting values

$[\![\underline{\{}\, f_0 \,\underline{=}\, e_0 \ldots f_n \,\underline{=}\, e_n \underline{\}}]\!]^E = \langle \vec{s}_1 \ldots \vec{s}_n,\ \underline{\{}\, f_0 \,\underline{=}\, e'_0 \ldots f_n \,\underline{=}\, e'_n \underline{\}} \rangle$ *where* $\langle \vec{s}_i,\ e'_i \rangle = [\![e_i]\!]^E$

record update: evaluate new field values e'_i, generate an update function $update\text{-}f_1 \cdots f_n$ and its closure upd_{cl}; return a new closure containing the field values

$[\![\underline{@\{}\, f_0 \,\underline{=}\, e_0 \ldots f_n \,\underline{=}\, e_n \underline{\}}]\!]^E = \langle \vec{s}_1 \ldots \vec{s}_n,\ \underline{gen\text{-}closure}\ upd\underline{[}\, e'_0 \ldots e'_n \underline{]} \rangle;\ \langle \vec{s}_i,\ e'_i \rangle = [\![e_i]\!]^E$

declare $\underline{closure}\ upd_{cl}\underline{[}\ \underline{obj}\ c_1 \ldots \underline{obj}\ c_n \underline{]}\ \underline{(}\ \underline{obj}\ r \underline{)}\ \underline{:}\ \underline{obj}\ \underline{\rightarrow}\ update\text{-}f_1 \cdots f_n$

declare $update\text{-}f_1 \cdots f_n \underline{(}\, c_1 \ldots c_n\ r \underline{)}\ \underline{:}\ \underline{obj}\ \underline{=}\ r'\ \underline{where}\ r'\ \underline{=}\ \underline{@\{}\, f_1 \,\underline{=}\, c_1 \ldots f_n \,\underline{=}\, c_n \underline{\}}\ r$

record selector function: generate a function $select\text{-}f$ and its closure; return a new closure as result

$[\![\underline{\$}\, f]\!]^E = \langle \emptyset,\ \underline{gen\text{-}closure}\ sel_{cl}\underline{[\ ]} \rangle$

declare $\underline{closure}\ sel_{cl}\underline{[\ ]}\ \underline{(}\ \underline{obj}\ r \underline{)}\ \underline{:}\ \underline{obj}\ \underline{\rightarrow}\ select\text{-}f$

declare $select\text{-}f \underline{(}\ \underline{obj}\ r \underline{)}\ \underline{:}\ \underline{obj}\ \underline{=}\ r'\ \underline{where}\ r'\ \underline{=}\ \underline{\$}\, f\ r$

query translation: invoke function e on the monadic state and return the result

$[\![\underline{query}\ e]\!]^E = \langle \vec{s},\ \underline{\lambda}\, \mathrm{s} \underline{\rightarrow}\, \underline{*}e' \underline{(}\,\mathrm{s}\underline{)} \rangle$ *where* $\langle \vec{s},\ e' \rangle = [\![e]\!]^E$

update translation: destructively set the monadic state to $\underline{*}e'\underline{(}\,\mathrm{s}\underline{)}$ and return a dummy value

$[\![\underline{update}\ e]\!]^E = \langle \vec{s}\ \mathrm{s}\ \underline{=}\ \underline{*}e'\underline{(}\,\mathrm{s}\underline{)}\,\underline{;},\ \underline{\lambda}\, \mathrm{s}\ \underline{\rightarrow}\ \underline{box}\ \underline{(}\ \underline{int}\ \ \underline{0}\ \underline{)} \rangle$ *where* $\langle \vec{s}, e' \rangle = [\![e]\!]^E$

do translation: return the empty statement sequence and the expression $\underline{\lambda}\, \mathrm{s}\ \underline{\rightarrow}\ \vec{d}\,\underline{;}\ \vec{s}\ e$ that evaluates to a closure containing declarations $\vec{d}$, statements $\vec{s}$, and the resulting expression e of the monadic sequence

$[\![\underline{do}\ seq]\!]^E = \langle [],\ \underline{\lambda}\, \mathrm{s}\ \underline{\rightarrow}\ \vec{d}\,\underline{;}\ \vec{s}\ e \rangle$ *where* $\langle \vec{d},\ \vec{s},\ e \rangle = [\![seq]\!]^S$

translation of assignment within monadic sequence: add declaration of x to assignments $\vec{d}$, evaluate e and apply it to the global state, assign the result to x; append translation of remaining sequence

$[\![x\ \underline{\leftarrow}\ e\,\underline{;}\ seq]\!]^S =$

$\langle \vec{d}\ \underline{obj}\ x,\ \vec{s}_e\ x\ \underline{=}\ exec\ e'\,\underline{;}\ \vec{s},\ \bar{e} \rangle$ *where* $\langle \vec{s}_e,\ e' \rangle = [\![e]\!]^E \wedge \langle \vec{d},\ \vec{s},\ \bar{e} \rangle = [\![seq]\!]^S$

translation of expression within monadic sequence: evaluate e, apply it to the global state, append translation of remaining sequence

$[\![e\,\underline{;}\ seq]\!]^S = \langle \vec{d},\ \vec{s}_e\ \underline{exec}\ e'\,\underline{;}\ \vec{s},\ \bar{e} \rangle$ *where* $\langle \vec{s}_e,\ e' \rangle = [\![e]\!]^E \wedge \langle \vec{d},\ \vec{s},\ \bar{e} \rangle = [\![seq]\!]^S$

translation of expression at the end of a monadic sequence: evaluate e, apply it to the global state, use the result of the application as result for the sequence

$[\![e]\!]^S = \langle \emptyset,\ \vec{s},\ \underline{exec}\ e' \rangle$ *where* $\langle \vec{s},\ e' \rangle = [\![e]\!]^E$

Figure 5.14.: Translation scheme from Core to Imp (part 2).

translation of string-, integer- and bit vector-constants: return the respective boxed value

$[\![\underline{"} string \underline{"}]\!]^E = \langle \emptyset, \underline{"} string \underline{"} \rangle$ $\qquad [\![n]\!]^E = \langle \emptyset, \underline{box} \underline{(} \underline{int} \; n \underline{)} \rangle$

$[\![\underline{'} b_n \ldots b_1 \underline{'}]\!]^E = \langle \emptyset, \underline{box} \underline{(} \underline{bits} \underline{[} n \underline{]} \; 2^{n-1} * b_n + \ldots + 2^0 * b_1 \underline{)} \rangle$

translation of constructors without argument: place the tag into a heap cell

$[\![x]\!]^E = \langle \emptyset, x \rangle \qquad [\![con]\!]^E = \langle \emptyset, \underline{box} \underline{(} \underline{int} \; tag \underline{)} \rangle$ $\qquad$ if *con* has no argument

constructors with argument: generate a function that places the tag and the playload onto the heap

$[\![con]\!]^E = \langle \emptyset, \underline{gen\text{-}closure} \; con_{cl} \underline{[} \; \underline{]} \rangle$ $\qquad$ if *con* has an argument

declare $\underline{closure} \; con_{cl} \underline{[} \; \underline{]} \underline{(} \underline{obj} \; payload \underline{)} \underline{:} \underline{obj} \; \underline{\rightarrow} \; con$

declare $\underline{constructor} \; con \underline{(} \underline{obj} \; payload \underline{)}$

Figure 5.15.: Translation scheme from Core to Imp (part 3).

5.2.1. Translating Monadic Sequences

The Core language provides a state monad, that is, a monad that threads a state through a sequence of monadic actions. An action may query or update this state using the primitives `query` and `update`. The `do`-notation is borrowed from Haskell and is syntactic sugar for combining the actions in a sequence:

```
1 val f x y = do
2   a <- actA;
3   b <- actB;
4   actC
5 end
```

```
val f x y =
  actA >>= (λa →
    actB >>= (λb →
      actC))
```

Here, the type of `actA : M r` is a monad where `r` is the type of the result. In our language, `M r` is syntactic sugar for $s \rightarrow \langle r, t \rangle$, that is, our monad has an internal state (of type s) that each action may modify to type t. Two actions are combined using the infix “bind” operator `>>=` as follows:

```
1 val (act >>= cont) = λs →
2     let val <a,t> = act s in cont a t end
```

Compiling monadic code into efficient machine code that does not generate any closures requires that `>>=` and the λ-term in its right argument `cont` are suitably inlined.

Not only is the desugared `do`-notation hard to read, avoiding generating closures for the λ-expressions also requires inlining of `>>=` and β-reduction (evaluation of λ-terms). These optimizations destroy the structure of the input program and may lead to excessive code duplication if not applied carefully. Thus, we propose a less literal translation by observing that the state `s` in the definition of `>>=` is not used once `act` is evaluated and the new state `t` is produced. Hence, we use a global variable s to store the state and use monadic functions that, instead of returning a tuple like `act`, destructively update s and only return the result `a`. This translation preserves the pure functional semantics of the input program because of some deliberate restrictions of our language. In particular, the evaluation of the sequence of monadic actions is started at exactly one place, i.e. in the `main` function, and the actions of a `do`-sequence are emitted in the order in which they appear in the source program. Monadic actions can call non-monadic functions, but these then cannot call any monadic actions. Also, there is no parallelism. Imagine the language would offer a primitive `fork` that expects two actions as parameters which are then run in different threads. Here, each of these two actions would need to work on a copy of the state. However, with our translation, both would write to the same shared global state and, thus, interfere with each other.

The translation scheme in Fig. 5.13 and 5.14 makes the idea of using a global variable explicit. The `do` statement is translated literally to a special lambda abstraction $\lambda \mathrm{s} \to b\ e$ where b is the **Block** representing the body of the `do` and e is the last expression in the sequence. The $[\![\cdot]\!]^S$ scheme translates each action e by wrapping it with *exec* e which applies e to the global state s. Indeed, *exec* e could be written e s but we chose the former notation for the sake of readability. The result of e s is the result of the monadic action, that is, no new state is returned. Thus, all built-in monadic functions have to use destructive assignments to s to update the state. The only function that modifies the state is *update* e, which is translated into the destructive assignment $\mathrm{s} = *e'(\mathrm{s})$; (here, e' is a closure containing e as its body) and otherwise behaves as an action $\lambda \mathrm{s} \to box(\underline{int}, 0)$ that returns the dummy value zero. The *query* e action merely invokes e on s and returns the result. As an example, consider the following code:

```
val f = do
  a <- actA;
  opndsz <- query $opndsz;
  actC
end
```

Applying the translation scheme to this code yields the Imp code shown below.

```
f(): obj = fRes where
```

```
2    fRes = λs →
3      obj a;
4      obj opndsz;
5      a = exec actA;
6      opndsz = exec (λ s → *(gen-closure select_opndsz_cl)( s));
7      exec actC
```

This code is optimized further. In particular, the λ-expressions are removed as detailed in Sect. 5.3.2. In the next section, we introduce the optimizations we apply to the Imp code.

5.3. Optimizing the Intermediate Representation Imp

This section details various optimizations that bring the code closer to natural C code, that is, C code that might have been written by hand. The challenge lies in removing the functional language artifacts as far as possible. These artifacts mostly relate to the boxing of values. The key transformations to avoid the boxing of functions and values are local simplifications, as detailed next. The transformation of monadic actions and the type-based refinement in Sect. 5.3.2 and 5.3.3, respectively, merely transform the program so that further simplifications are possible. The simplifier is run once after each transformation.

As a running example, we apply the translation scheme to the code in Fig. 5.3 which is based on lines 19–22 of `decode` in Fig. 5.2 (see page 45), yielding the code in Fig. 5.16 (note that `r/m16`, etc. are constants that are turned into calls with no arguments).

The code contains the construction of many unnecessary closures such as the invocation `*(gen-closure binop[])` that is equivalent to a direct call to `binop`.

5.3.1. Simplifying Imp

The simplifier traverses the AST looking for sequences of computations that can be replaced with cheaper or no operations. A dead code elimination pass is run as final pass in order to remove any declarations that are no longer required due to the optimizations. The rules of the simplifier are presented in Table 5.1 and motivated as follows:

- Rules 1 and 2 remove superfluous boxing and unboxing pairs on base types.
- Rule 3 identifies monadic closures that are immediately executed. However, a monadic closure $\lambda s \to \vec{d}\ ;\ \vec{s}\ e$ declares additional variables $\vec{d}$ and executes the

```
1  decode(): obj = decodeRes where
2    decodeRes = λ s →
3      obj opndsz; obj ite;
4      opndsz = exec (λ s → *(gen-closure select_opndsz_cl)( s));
5      if unbox(int, opndsz) then
6        ite =
7          *(gen-closure binop[]) (gen-closure ADD_cl[], r/m16(), r16());
8      else
9        ite =
10         *(gen-closure binop[]) (gen-closure ADD_cl[], r/m32(), r32());
11     ;
12     exec ite;
```

Figure 5.16.: Unoptimized translation of Fig. 5.3.

statements $\vec{s}$ before returning the value e. Hence, these declarations must be moved to the enclosing block to ensure that they are visible when evaluating $\vec{s}$ and e. The statements $\vec{s}$ must be executed before evaluating e.

- Rule 4 identifies closures that are constructed and then immediately invoked. The pattern corresponds to the Core call `f` $\vec{e}$ with $\vec{e} \in \mathbf{Expr}^n$ where `f` is defined as a function that expects n parameters.

- Rules 5 and 6 inline functions that have been generated during the conversion from Core to Imp, namely functions *select-foo* for a record selector $*foo* and *update-f1*$\cdots f_n$ for record update functions. The rules apply when the functions are applied to a record in which case the select and update expressions from Imp can be used directly. Note that this is not always the case: A selector or update function is a first-class object in Core.

The rules of the simplifier are rather standard when compiling functional programs [37]. Applying them to the example code of the last section yields:

no	rule			remark
1	$box(t, unbox(t, e))$	$\rightsquigarrow$	e	
2	$unbox(t, box(t, e))$	$\rightsquigarrow$	e	
3	$x = exec(\lambda s \rightarrow \vec{d};\ \vec{s};\ e)$	$\rightsquigarrow$	$\vec{s};\ x = e$	insert declarations $\vec{d}$ in surrounding block
4	$*(gen\text{-}closure\ f_{cl}[\vec{a}])(\vec{a}')$	$\rightsquigarrow$	$f(\vec{a}\ \vec{a}')$	f_{cl} is declared as *closure* $f_{cl}[\vec{d}](\vec{d}') : t_r \rightarrow f$, $\vec{a} \in \mathbf{Expr}^n$, $\vec{d} \in \underline{id}^n$, $\vec{a}' \in \mathbf{Expr}^m$, $\vec{d}' \in \underline{id}^m$
5	$select\text{-}f(r)$	$\rightsquigarrow$	$\$f(r)$	*select-f* was generated for the selector $\$f$
6	$update\text{-}f_1 \cdots f_n(e_1 \ldots e_n\ r)$	$\rightsquigarrow$	$@\{f_1 = e_1 \ldots f_n = e_n\}\ (r)$	$update\text{-}f_1 \cdots f_n$ was generated for the record update function $@\{f_1 = e_1 \ldots f_n = e_n\}$

Table 5.1.: Rules of the Simplifier

```
1  decode() : obj = decodeRes where
2    decodeRes = λ s →
3      obj opndsz; obj ite;
4      opndsz = $opndsz s;
5      if unbox(int,opndsz) then
6         ite = binop(gen-closure ADD_cl[], r/m16(), r16());
7      else
8         ite = binop(gen-closure ADD_cl[], r/m32(), r32());
9      ;
10     exec ite;
```

In particular, rules 3, 4, and 5 have been applied to line 4 of Fig. 5.16. Rule 4 has been applied to lines 7 and 10.

As another example again consider the code in Fig. 5.12. Applying the rules yields the optimized code in Fig. 5.9. Note that the closure for `f` is then unused and can be omitted.

5.3.2. Removing Monadic Actions

A major source of inefficiency and illegibility of the code generated so far relates to monadic actions: every Core function whose body consists of a `do`-block returns a

closure. One example is the `binop` function that commences as follows:

```
1  binop(obj cons, obj giveOp1, obj giveOp2) : obj
2    = binopRes where binopRes = λ s → body
```

Note that, in order to obtain a result from `binop`, it must first be called with its three arguments, then the resulting closure must be run by applying *exec* to it. Since the state of the monad is updated destructively, the translation retains the source code semantics if the sequence in which computations are performed remains the same. In particular, we can rewrite the function `binop` : $(\underline{obj}, \underline{obj}) \to S \to \underline{obj}$ and all its call-sites so that its type becomes $(\underline{obj}, \underline{obj}) \to \underline{obj}$ without altering when a result is computed. Hence, we replace λs $\to$ *body* with *body* and we convert each call site of `binop(...)` to λs $\to$`binop(...)`. In general, we traverse the program and gather all functions whose top-level expression is a monadic closure λs $\to \ldots$. If this set contains a function f whose closure is computed anywhere in the program, it has to be removed from the set since a correct transformation would have to transform all invoke-expressions to which the closure of f can flow and, in turn, all other functions from closure computations that flow to this invoke-expression. In Fig. 5.16, all monadic functions can be converted:

```
1  decode() : obj = decodeRes where
2    obj opndsz; obj ite;
3    opndsz = $opndsz s;
4    if unbox(int, opndsz) then
5       ite =
6         λs → binop(gen-closure ADD_cl[], λs → r/m16(), λs → r16());
7    else
8       ite =
9         λs → binop(gen-closure ADD_cl[], λs → r/m32(), λs → r32());
10   ;
11   decodeRes = exec ite;
```

The converted program cannot be simplified since the monadic abstractions λs $\to$ are not surrounded by *exec*. Note, however, that the monadic closure that is assigned to *ite* is executed in the last line. We thus apply a transformation by performing a backward-substitution on variables that were generated by the Core to Imp translation. In particular, the assignment `decodeRes = exec ite` is removed and any assignment of the form `ite =` *exp* is replaced by `decodeRes = exec` *exp*. In general, we also propagate the simpler pattern `var1 = var2` (where `var1` was generated by the translation to Imp)

```
1  decode() : obj = decodeRes where
2    obj opndsz; obj ite;
3    opndsz = $opndsz s;
4    if unbox(int, opndsz) then
5       decodeRes =
6         binop(gen-closure ADD_cl[], λs → r/m16(), λs → r16());
7    else
8       decodeRes =
9         binop(gen-closure ADD_cl[], λs → r/m32(), λs → r32());
10   ;
```

Figure 5.17.: Partially optimized code after applying backwards substitution.

backwards and replace `var2` = *exp* with `var1` = *exp*. Applying this transformation on the code above and running the simplifier yields the code in Fig. 5.17.

Finally, we perform a pass that generates for each expression λs $\rightarrow$ *body* a top-level function containing *body* that takes the free variables $x_1, \ldots x_n$ of *body* as arguments. The λ-expression is then replaced by `gen-closure f_cl[`$x_1, \ldots x_n$`]`. As a consequence, *exec e* has to be turned into a function invocation $*e()$. Applying this transformation to `decode()` from above yields the following code:

```
1  closure r16_new_cl[](): obj -> r16_new
2  r16_new() : obj = Res where
3    Res = r16();
4
5  closure r/m16_new_cl[](): obj -> r/m16_new
6  r/m16_new() : obj = Res where
7    Res = r/m16();
8
9  closure r32_new_cl[](): obj -> r32_new
10 r32_new() : obj = Res where
11   Res = r32();
12
13 closure r/m32_new_cl[](): obj -> r/m32_new
14 r/m32_new() : obj = Res where
15   Res = r/m32();
16
17 decode() : obj = decodeRes where
```

```
18    obj opndsz; obj ite;
19    opndsz = $opndsz s;
20    if unbox(int, opndsz) then
21       decodeRes = binop(gen-closure ADD_cl[],
22         gen-closure r/m16_new_cl[], gen-closure r16_new_cl[]);
23    else
24       decodeRes = binop(gen-closure ADD_cl[],
25         gen-closure r/m32_new_cl[], gen-closure r32_new_cl[]);
26    ;
```

A better translation of the code in Fig. 5.17 is possible for patterns of the form $\lambda s \rightarrow f()$ since replacing it by the closure of a top-level function that calls $f()$ is equivalent to replacing $\lambda s \rightarrow f()$ by `gen-closure f_cl[]`. Thus, in the example, the monadic closure $\lambda s \rightarrow$ `r/m16()` is translated to `gen-closure r/m16_cl[]`:

```
1  closure r16_cl[](): obj -> r16
2  closure r/m16_cl[](): obj -> r/m16
3  closure r32_cl[](): obj -> r32
4  closure r/m32_cl[](): obj -> r/m32
5
6  decode() : obj = decodeRes where
7    obj opndsz; obj ite;
8    opndsz = $opndsz s;
9    if unbox(int, opndsz) then
10      decodeRes = binop(gen-closure ADD_cl[],
11        gen-closure r/m16_cl[], gen-closure r16_cl[]);
12   else
13      decodeRes = binop(gen-closure ADD_cl[],
14        gen-closure r/m32_cl[], gen-closure r32_cl[]);
15   ;
```

This concludes the optimization of monadic actions. In the following, we explain our unboxing optimization which makes of use a special kind of type inference.

5.3.3. Unboxing by Type Inference

By default, all variables are pointers to the heap, thus requiring heap-allocating the result of each computation which is slow and produces hard-to-read C code. Our

a)

$$\begin{array}{lcl} \textbf{Type} & ::= & \underline{void} \mid \underline{obj} \mid \underline{int} \\ & \mid & \underline{bits}\ \underline{[}\ \underline{num}\ \underline{]} \mid \underline{vec} \mid \underline{str} \\ & \mid & (\,\textbf{Type}^*\,) \overset{\textbf{Flag}}{\rightarrow} \textbf{Type} \\ & \mid & \underline{box}\ \underline{[}\ \textbf{Type}\ \underline{]} \mid \underline{M}\ \ \textbf{Type} \\ & \mid & \underline{\{}\ (\underline{field}\ \underline{:}\ \textbf{Type}\ \underline{,}\ )^*\ \textbf{Flag}\ \underline{\}} \\ \textbf{Flag} & ::= & \underline{true} \mid \underline{false} \end{array}$$

b)

$$\begin{array}{lcl} \underline{void} \sqcup t = t \sqcup \underline{void} & = & t \\ t \sqcup t & = & t \\ \underline{bits}\ \underline{[}\ c\ \underline{]} \sqcup \underline{vec}\ = \underline{vec} \sqcup \underline{bits}\ \underline{[}\ c\ \underline{]} & = & \underline{vec} \\ \underline{bits}\ \underline{[}\ c_1\ \underline{]} \sqcup \underline{bits}\ \underline{[}\ c_2\ \underline{]} & = & \underline{vec} \\ \underline{box}\ \underline{[}\ t_1\ \underline{]} \sqcup \underline{box}\ \underline{[}\ t_2\ \underline{]} & = & \underline{box}\ \underline{[}\ t_1 \sqcup t_2\ \underline{]} \\ \underline{M}\ \ t_1 \sqcup \underline{M}\ \ t_2 & = & \underline{M}\ \ (t_1 \sqcup t_2) \\ (t_1, \ldots t_n) \overset{c}{\rightarrow} r \sqcup (t'_1, \ldots t'_n) \overset{c'}{\rightarrow} r' & = & (t_1 \sqcup t'_1, \ldots t_n \sqcup t'_n) \overset{c \vee c'}{\rightarrow} r \sqcup r' \\ \{f_l, e_l\} \sqcup \{f_r, e_r\} & = & mergeRec(f_l, e_l, f_r, e_r) \\ t_1 \sqcup t_2 & = & \underline{obj} \end{array}$$

c)

$$mergeRec\left(\begin{array}{l} f^c_1 : t^l_1, \ldots f^c_k : t^l_k,\ f^l_{k+1} : t^l_{k+1}, \ldots f^l_n : t^l_n,\ e_l, \\ f^c_1 : t^r_1, \ldots f^c_k : t^r_k,\ f^r_{k+1} : t^r_{k+1}, \ldots f^r_m : t^r_m,\ e_r \end{array} \right) =$$

$$\left\{ \begin{array}{ll} \underline{obj} & \text{if } e_r \wedge n \neq k \vee e_l \wedge m \neq k \\ \underline{\{} f^c_1 : t^l_1 \sqcup t^r_1,\ \ldots f^c_k : t^l_k \sqcup t^r_k,\ f^l_{k+1} : t^l_{k+1},\ \ldots f^l_n & \\ \quad : t^l_n, f^r_{k+1} : t^r_{k+1},\ \ldots f^r_m : t^r_m, e_l \vee e_r \} & \text{otherwise} \end{array} \right.$$

Figure 5.18.: Definition of Types, their union and merging of record types.

central and – as far as we know – novel optimization is a type inference that determines which data can be stored in variables.

$$\frac{\Gamma(f) = (t_1 \dots t_n) \overset{\mathit{false}}{\to} t_r \qquad \Gamma(a_i) = t_i \qquad \Gamma(r) = t_r \qquad \Gamma \vdash b : t_r}{\Gamma \vdash f(t_1 a_1 \dots t_n a_n) : t_r = r \; \mathit{where} \; b} \quad (\textsc{fun})$$

$$\frac{\Gamma(id) = (t) \overset{\mathit{false}}{\to} \underline{obj}}{\Gamma \vdash \mathit{constructor} \; id(t \; a)} \quad (\textsc{con})$$

$$\frac{\Gamma(cid) = (t_1 \dots t_k) \overset{\mathit{false}}{\to} (t_{k+1}, \dots t_n) \overset{k>0}{\to} t_r \qquad \Gamma(f) = (t_1 \dots t_n) \overset{\mathit{false}}{\to} t_r}{\Gamma \vdash \mathit{closure} \; cid[t_1 a_1 \dots t_k a_k](t_{k+1} a_{k+1} \dots t_n a_n) : t_r \to f} \quad (\textsc{clo})$$

$$\frac{\Gamma(x) = t \qquad \Gamma \vdash e : t}{\Gamma \vdash x = e;} \quad (\textsc{ass}) \qquad \frac{\Gamma \vdash e : \underline{int} \qquad \Gamma \vdash b_t \qquad \Gamma \vdash b_e}{\Gamma \vdash \mathit{if} \; e \; \mathit{then} \; b_t \; \mathit{else} \; b_e;} \quad (\textsc{if})$$

$$\frac{\Gamma \vdash e : \underline{int} \qquad \Gamma \vdash b_i}{\Gamma \vdash \mathit{case} \; e \; \mathit{of} \; p_1 : b_1; \dots p_n : b_n;} \quad (\textsc{case}) \qquad \frac{\Gamma[x_1 \mapsto t_1, \dots x_n \mapsto t_n] \vdash s_i}{\Gamma \vdash t_1 \; x_1 \dots t_n \; x_n; s_1 \dots s_m} \quad (\textsc{block})$$

$$\Gamma \vdash id : \Gamma(id) \quad (\textsc{var}) \qquad \Gamma \vdash 42 : \underline{int} \quad (\textsc{int}) \qquad \Gamma \vdash \text{"}\mathit{string}\text{"} : \underline{str} \quad (\textsc{str})$$

$$\Gamma \vdash {'}b_1 \dots b_n{'} : \underline{bits} \; \underline{[} \, n \, \underline{]} \quad (\textsc{vec}) \qquad \frac{\Gamma \vdash e : (t_1 \dots t_n) \overset{\mathit{false}}{\to} t_r \qquad \Gamma \vdash e_i : t_i}{\Gamma \vdash e(e_1 \dots e_n) : t_r} \quad (\textsc{call})$$

$$\frac{\Gamma \vdash e_i : t_i}{\Gamma \vdash \{f_1 = e_1 \dots f_n = e_n\} : \{f_1 : t_1 \dots f_n : t_n, \mathit{true}\}} \quad (\textsc{rec})$$

$$\frac{\Gamma \vdash e : \underline{box} \; \underline{[} \, (t_1 \dots t_n) \overset{\mathit{non\text{-}triv}}{\to} t_r \, \underline{]} \qquad \Gamma \vdash e_i : t_i}{\Gamma \vdash *e(e_1 \dots e_n) : t_r} \quad (\textsc{inv}) \text{ where } \mathit{non\text{-}triv} \in \{\mathit{true}, \mathit{false}\}$$

$$\frac{\Gamma \vdash r : \{f : t, \; e\}}{\Gamma \vdash \$f \; r : t} \quad (\textsc{sel}) \qquad \frac{\Gamma \vdash e_i : t_i \qquad \Phi(f_i) = t_i}{\Gamma \vdash \{f_1 = e_1 \dots f_n = e_n\} : \underline{obj}} \quad (\textsc{rec-glob})$$

$$\frac{\Gamma \vdash e_i : t_i \qquad \Gamma(cid) = (t_1 \dots t_k) \overset{\mathit{false}}{\to} (t_{k+1} \dots t_n) \overset{\mathit{non\text{-}triv}}{\to} t_r}{\Gamma \vdash \mathit{gen\text{-}closure} \; cid[e_1 \dots e_k] : \underline{box} \; \underline{[} \, (t_{k+1} \dots t_n) \overset{\mathit{non\text{-}triv}}{\to} t_r \, \underline{]}} \quad (\textsc{gen-clo})$$

Figure 5.19.: Typing rules that characterize programs on which unboxing can be applied (part 1).

$$\frac{\Gamma \vdash r : \underline{obj} \qquad \Phi(f) = t}{\Gamma \vdash \$f\ r : t} \text{(SEL-GLOB)} \qquad \frac{\Gamma \vdash e_i : t_i \qquad \Gamma \vdash r : \underline{obj} \qquad \Phi(f_i) = t_i}{\Gamma \vdash @\{f_1 = e_1 \ldots f_n = e_n\}\ r : \underline{obj}} \text{(UPD)}$$

$$\frac{\Gamma \vdash e : \underline{box}\,\underline{[}\,t\,\underline{]}}{\Gamma \vdash unbox(t, e) : t} \text{(UNBOX)} \qquad \frac{\Gamma \vdash e : t}{\Gamma \vdash box(t, e) : \underline{box}\,\underline{[}\,t\,\underline{]}} \text{(BOX)}$$

$$\frac{\Gamma \vdash e : \underline{M}\ \ t}{\Gamma \vdash exec\ e : t} \text{(EXEC)} \qquad \frac{\Gamma \vdash b : t_b \qquad \Gamma \vdash e : t_r}{\Gamma \vdash \lambda s \to b\ e : \underline{M}\ \ t_r} \text{(DO)}$$

Figure 5.20.: Typing rules that characterize programs on which unboxing can be applied (part 2).

The type universe of Imp is shown in Fig. 5.18 a) (Fig. 5.18 b) and Fig. 5.18 c) are discussed on page 68). Here, the special type $\underline{void}$ represents the empty set of program values whereas $\underline{obj}$ represents all possible program values. Other types are $\underline{bits}\,\underline{[}\,n\,\underline{]}$ for vectors of n bits, $\underline{vec}$ for bit vectors whose size is not statically known and whose size is therefore tracked at runtime, monadic actions $\underline{M}\ \ r$ with result type r, boxed types $\underline{box}\,\underline{[}\,t\,\underline{]}$ where t is not a monadic action, and record types. Figures 5.19 and 5.20 present typing rules that characterize Imp programs that have monomorphic typings which are exactly those programs whose variables have a fixed type in all executions and which therefore can be unboxed. Most Imp programs are not well-typed under these rules, in these cases, our inference will over-approximate this typing.

The top three rules in Fig. 5.19 specify how the top-level declarations are represented in the environment Γ. Rules for statements and for expressions follow. A specialty of our types is the flag f in a function type $(t_1 \ldots t_n) \xrightarrow{f} t$ that is *true* if $k > 0$ in rule (CLO), that is, if the closure contains environment variables. Such non-trivial closures always have boxed function types and are later represented by a heap-allocated structure that contains a function pointer together with the required environment variables.

Another non-standard aspect are the types of records that may be either flexible (later represented by a linked list of field/value pairs) or fixed (later represented by a C `struct`). Any record r to which *update* r is applied is flexible, its type is $\underline{obj}$ and the type of the fields is given in a global map Φ (rules (UPD), (REC-GLOB) and (SEL-GLOB)). Fixed records have the type $\{f_1 : t_1,\ \ldots f_n : t_n,\ all\}$ where the flag *all* indicates whether all fields of the record are known. For instance, the set of fields in a record $\{f_1 = 1,\ f_2 = 2\} = \{f_1 : \underline{int}\,,\ f_2 : \underline{int}\,,\ true\}$ is always known whereas the set in a selector type $\$f_1 : (\{f_1 : t,\ false\}) \to t$ is not known which is reflected by the *all* flag. The remaining rules are standard.

In order to infer a typing that is sound with respect to these rules, we define the union of two types $t_1 \sqcup t_2$ in Fig. 5.18b). The shown rules are to be read from top to

rule	inferred	new type	original	new
writing x	$\underline{box}$ [$\underline{int}$]	$\underline{int}$	$x = e$	$x = unbox(\underline{int}, e)$
reading x	$\underline{box}$ [$\underline{int}$]	$\underline{int}$	x	$box(\underline{int}, x)$
writing x	$\underline{box}$ [$\underline{vec}$]	$\underline{vec}$	$x = e$	$x = unbox(\underline{vec}, e)$
reading x	$\underline{box}$ [$\underline{vec}$]	$\underline{vec}$	x	$box(\underline{vec}, x)$
writing x	$\underline{box}$ [$\underline{bits}$ [c]]	$\underline{int}$	$x = e$	$x = unbox(\underline{bits}$ [c], $unbox(\underline{vec}, e))$
reading x	$\underline{box}$ [$\underline{bits}$ [c]]	$\underline{int}$	x	$box(\underline{vec}, box(\underline{bits}$ [c], $x))$
closure $cid \to f$	$\underline{box}$ [$(\vec{t}) \overset{false}{\to} t_r$]	$(\vec{t}) \overset{false}{\to} t_r$	*gen-closure cid*[]	f
invoke e	$\underline{box}$ [$(\vec{t}) \overset{false}{\to} t_r$]	$(\vec{t}) \overset{false}{\to} t_r$	$*e(a_1, \ldots a_n)$	$e(a_1, \ldots a_n)$

Table 5.2.: Unboxing rules.

bottom, thus, $\underline{bits}$ [5] $\sqcup$ $\underline{bits}$ [5] $=$ $\underline{bits}$ [5] due to the third rule. The union of two fixed records is defined in Fig. 5.18c). The first rule applies if all fields are known in one of the records (say, the right one: $e_r = true$) and the other record contains extra fields ($n \neq k$). Returning $\underline{obj}$ implies that the record can only be represented as a flexible record. For example, consider the `insn` type in Fig. 5.1 and the expression `case inst of ADD arg2 -> eval arg2 | INC arg1 -> eval arg1`. Since `arg1` and `arg2` are both passed to `eval`, a common type must be computed. But $\{opnd1 : \underline{obj}, opnd2 : \underline{obj}, true\} \sqcup \{opnd1 : \underline{obj}, true\} = \underline{obj}$ since $e_r = true \wedge 2 = n \neq k = 1$. Hence, the argument to `eval` and to `ADD` and `INC` must be a flexible record.

Algorithmically, we replace the $\underline{void}$ type in the universe of types **Type** with a set of type variables and associate a different variable with each record operation and all positions in the program where the grammar in Fig. 5.7 specifies **Type**. Based on the typing rules, we equate these type variables with the stipulated types and apply the union operation $\sqcup$ if two types differ. This inference can be implemented very efficiently using a union-find data structure where an equivalence class representative holds the type of the type variables in that class.

Note that our type inference is non-standard: a fresh type variable corresponds to the $\underline{void}$ type and equating it with other types applies $\sqcup$ which encodes anti-unification. In the worst case, the most general type $\underline{obj}$ is inferred. In contrast, standard type inference performs unification and a fresh type variable corresponds to $\underline{obj}$, the most general type. In the worst case, the least general type $\underline{void}$ is inferred, indicating a type error since no program values exist in this type. Thus, while standard type inference

Figure 5.21.: GDSL program assembly.

infers the most general type, ours infers the most specific type, thereby solving the problem of inferring a type that is sufficiently specific to unbox the arguments of the `cmp` function in Sect. 5.0.2 on page 41.

Given the inferred types, Table 5.2 shows the rules used to re-write various expressions in the program. The rewriting of function return values and arguments are handled analogously to writing/reading rules. The *box*/*unbox* constructors also show when a variable contains a bit-vector <u>*bits*</u> <u>*[c]*</u> of fixed size *c* in which case the bit-string is stored as <u>*int*</u>.

Finally, we infer a fixed record for the monadic state if all update actions are of the form *update* @$\{\ldots\}$ and all query actions are of the form *query* $\$f$. In this case the record does not escape and can be updated in-situ. An update rule analogous to (SEL) infers the set of fields and we translate *update* @$\{f = 42\}$ to `state.f = 42` where `state` contains the C `struct` of the monadic state. This is exemplified by lines 26–28 of Fig. 5.5.

5.4. Implementation

We designed the GDSL toolkit with the goal to write concise x86 instruction decoders, semantic translators to RReil and RReil optimizers. The optimizations are crucial to reduce the size of the resulting code. Figure 5.21 illustrates how a GDSL library is synthesized. In particular, a generic runtime template is specialized for the input GDSL program by adding declarations for the monadic state, constructors, and heap allocation functions. This bespoke runtime is combined with the generated code and compiled to a library. A host C program can then link against this library.

Our C back-end adds a pointer to the runtime environment `s` as the first argument to every emitted function. This runtime environment also contains the monadic state in

`s->state`, thereby alleviating the need for any global state and, thus, making the library thread-safe - this is important since GDSL does not provide means for concurrency itself and, thus, concurrency has to be implemented by using multiple GDSL instances. As hinted at in Fig. 5.21, the host program obtains an initial runtime environment by calling `init()`. It is then at liberty to call and use the result of any GDSL function that returns basic C types and `structs`. In order to transfer algebraic data types, such as ASTs, to C, a structural induction can be programmed in the DSL that calls a different function for each constructor. Passing these functions as a record generates a traversal function that takes a C `struct` containing well-typed pointers to C functions. Thus, marshaling data between the DSL and C is very simple and no extra tools are needed.

Once the results are extracted, the host program may reset the GDSL heap through the `reset_heap()` function. The function frees all heap space except for the first page which yields a slight performance advantage when running many small DSL functions. GDSL uses fast bumper pointer allocation [30] within each page and increases the heap size by one four kilobyte page when running out of memory.

The GDSL language requires the programmer to specify which functions to export. All non-exported functions are declared static and are aggressively optimized by the C compiler. Indeed, by annotating the runtime function that allocates a new page so that it is not inlined, the size of the executable decreases by nearly one third. Modern C compilers turn tail-recursive function calls into jumps, thereby allowing recursive loops written in the DSL to run without any overhead over loops written in C.

5.5. Experimental Evaluation

We evaluated our implementation regarding the following three aspects: effectiveness of the optimizations, comparison of the performance to hand-written code, and heap consumption for various task sizes. We assess the optimizations by benchmarking the decoding and pretty printing of the 11Mb `clang` binary. Table 5.3 shows the performance results for different GDSL compiler optimization configurations. Table 5.4 contains measurements for the memory usage. We use the term *fixed records* when emitting C `structs` whenever possible. We call records unboxed if the C `structs` are passed by value rather than allocated on the heap. In contrast, compilation without fixed records exclusively relies on lists of field/value pairs which are always heap-allocated. The table shows the difference to the optimal case where type-based refinement has been applied, records are fixed if possible and all fixed records are unboxed. The second column shows the size of the generated C code. Thereafter, the binary code size of the decoder and the program runtime are displayed both with and without C compiler optimizations (-O0 and -O2). Finally, the last columns contain the average

optimization options	lines of code	with -O0 exe size	with -O0 time	with -O2 exe size	with -O2 time
all optimizations	36k	511kb	3.6s	295kb	1.1s
all w/o unboxed records	36k	499kb	4.0s	284kb	1.2s
all w/o type refinement	41k	855kb	5.4s	719kb	1.5s
all w/o fixed records	37k	605kb	9.9s	388kb	3.7s
all w/o fixed records w/o type refinement	42k	945kb	11.4s	764kb	3.8s

Table 5.3.: Decoding performance depending on the GDSL compiler optimization level.

	avg. heap residency	max. heap residency
all optimizations	1.0kb	2.5kb
all w/o unboxed records	1.1kb	3.14kb
all w/o type refinement	1.9kb	6.2kb
all w/o fixed records	4.6kb	10.9kb
all w/o fixed records w/o type refinement	5.3kb	14.4kb

Table 5.4.: Decoding memory footprint depending on the GDSL compiler optimization level.

per-instruction and the maximum heap residency of the decoder.
Table 5.5 compares our decoder to Intel's XED decoder [12] using the same binary input as above. In Sect. 2.3, we have observed XED to be the fastest freely available instruction decoder that is also correct. Since XED is distributed in compiled form, we cannot assess what changes were made between versions 2.11 and 2.12 besides them being compiled by different gcc compiler versions. Still an increase in speed by nearly 30% suggests that the library has been manually tuned for speed. Given that all optimizations that our GDSL program has encountered lie in the application-agnostic optimization passes described in this work, we believe that this highlights the merit of using pure functional programs based on a state monad as core for a DSL.

Finally, Table 5.6 presents measurements for various GDSL programs. The test cases are ordered by task complexity; we used the same test input as above. The table starts by recapitulating the performance of the decoder. The decoder processes the binary instructions independently, that is, the heap is reset after the decoding of each

decoder	time	dec. insn.	exe size
XED 2.12	1.2s	2667248	1344kb
XED 2.11	1.7s	2667248	1024kb
GDSL opt.	1.1s	2667248	295kb

Table 5.5.: Decoding performance of XED from the Intel Pin toolkit.

GDSL program	time	heap residency avg.	heap residency max.	alloc. rate
x86 decoder + printing	1.2s	1.0kb	2.5kb	2090Mb/s
x86 decoder + translator + printing	8.2s	7.7kb	66kb	2459Mb/s
x86 decoder + translator + liveness	92s	206kb	67Mb	1305Mb/s
x86 decoder + translator + lookahead live.	241s	497kb	67Mb	1203Mb/s

Table 5.6.: GDSL program performance using all optimizations

instruction. Next, measurements for the semantic translation of single instructions are shown. Again, instructions are handled independently, but the heap can only be reset after the semantic translation of a decoded instruction completes. The third line of the tables provides measurements for our liveness analysis which processes the binary input basic-block-wise. In order to perform the analysis, the translation of the whole basic block needs to be kept in memory and the heap can only be reset after finishing that block. Finally, the last line presents measurements for an enhanced liveness analysis that does not only consider one basic block, but also its successors (if they can be determined). Here, the data for up to three basic blocks needs to be stored in memory. The high maximal memory usage of 67Mb indicate that there is a single basic block that is rather large. In production quality applications, the task size run as DSL program should be artificially limited in size in order to prevent an out-of-heap situation. In our case, it would be enough to split up basic blocks once they have reached a certain size.

5.6. Related Work

A common perception of domain-specific languages is that they should be “small” and not Turing-complete [22] since they otherwise encompass more than the domain-specific aspect of the problem. This work proposes that a general functional language as carrier is an effective approach to symbolic computations. Since it is seamlessly embedded

into C, as soon as logic has to be encoded for which a pure functional language is unsuitable, it can be easily implemented in C.

Our translation scheme can furthermore be characterized as a “shallow embedding”, that is, an implementation that translates the DSL program into the native operations of the target language. In contrast, a “deep embedding” is an implementation in which the DSL program is evaluated using an interpreter written in the target language [21]. In practice, there is a continuum between these extremes and, indeed, our type-based optimization transforms the DSL program to be more shallow: as an example, a non-optimized program represents every record as a linked list of field/value pairs which is not a common way to represent data structures in C. The optimization of turning most of the records into C `structs` makes the DSL program more C-like and, hence, more shallowly embedded.

While many DSLs focus on making programming simpler and safer, the FFTW library for computing discrete Fourier transforms uses a DSL specifically to obtain programs that are more efficient than hand-written C programs [24]. The underlying principle is to express the transform as a directed acyclic graph of codelets that implement building blocks and to optimize this graph by rewriting. Another example is the Pan library for image manipulation [20] which embeds a DSL into the general purpose language Haskell. The DSL constructs are Haskell functions that generate an abstract syntax tree which is then optimized using inlining and common-subexpression elimination. The resulting code is emitted as C code and compiled by a C compiler. In both approaches, the goal of obtaining good performance sacrifices the structure of the input program during translation, so that finding bugs in the emitted program is difficult for the user of the DSL. An even more ambitious way of optimizing a functional DSL is to use standard compiler techniques such as a translation to continuation passing style (CPS). Our initial back-end was based on the CPS transformation and optimization by Kennedy [31]. We found that removing the bind function that concatenates two monadic actions was crucial to obtain code with few closure allocations. Unfortunately, this optimization required a somewhat aggressive inlining (using β-reduction). Controlling the inlining turned out to be a major difficulty and even in the best setting, a translation of the x86 decoder alone resulted in 13MB of C code which becomes difficult to compile and very hard to debug. Moreover, the resulting code had about 1/3 of the performance of our current back-end. The presented translation scheme therefore lies at a sweet-spot in that it creates readable C code that has a small executable footprint while achieving the performance of hand-written C/C++ code.

The idea of performing a task on a fresh heap and discarding the heap upon completion is a form of region-based memory management [30]. It finds widespread use in the implementation of plug-in modules for the Apache web-server. In some applications, the DSL program might allocate a lot of memory and a garbage collector

might be useful. However, a generic garbage collector for our DSL requires that data on the heap is tagged, which may reduce the performance of the generated code. Instead, a built-in garbage collection primitive could be added that allows the programmer to specify which data to keep and ensures that all other identifiers can no longer be accessed. In this case, the compiler would generate a bespoke copying function for the data to keep based on the types. Note that these types must be monomorphic since copying polymorphic types requires the data on the heap to be tagged in order to determine the size of the heap object. An interesting case is to apply the garbage collection primitive only to the monadic state. Such a setup would enable the use of high-level, functional DSLs in small embedded control systems that repeatedly execute an infinite loop. The DSL would require that the state of the system is stored inside the monad which is then the only data that remains alive between loop iterations.

Type-based unboxing of heap values has been investigated by Leroy in the context of the OCaml compiler [37]. His work was generalized to also cope with the module system of ML and, thus, separate compilation [49]. The latter work observes that many functions are used with a type that is more monomorphic than their inferred type as illustrated by the `cmp` function in Sect. 5.0.2. Since a polymorphic argument to a function requires that it is boxed, Bjørner proposes to specialize a function type as much as possible in order to perform more unboxing [8]. Our approach achieves the same effect by performing a monomorphic type inference and to use the special type $\underline{obj}$ to represent a type that is not monomorphic. Moreover, any constructor that is only used to store specific types will have its payload stored unboxed. Thus, a polymorphic list that is only used with integers will store the integer directly in each list cell. On the downside, compiling DSL modules separately requires that the monomorphic type information can be propagated between the different modules in order to propagate the type requirements between all call sites and functions. The infrastructure for communicating types between separate modules is always built into the type checker/inference of the compiler front-end but is unlikely to be available to the back-end. Thus, our monomorphic type inference is easiest to apply as a whole-program analysis that requires all DSL modules at once. A bespoke type inference algorithm that can deal with separate compilation has been presented by Thiemann [59]. While his inference uses a simpler lattice and allows for polymorphic function types, our context-insensitive inference seems to be much simpler.

5.7. Conclusion

We have presented a translation scheme for a purely functional language that provides a built-in state monad. The goal is to perform a structure-preserving translation that

enables the user to easily relate the emitted code with the DSL program and thereby allows for simple debugging and profiling using the emitted code. To this end, several simple transformations were presented that replace concepts from the input DSL (such as boxed values, closures, and curried functions) into concepts used in C programs (such as `structs` and function pointers). The key insight is that even larger DSL programs are often simple enough to optimize most closure and boxing operations away, thereby yielding an imperative program that resembles hand-written C code. Due to this resemblance, many of the optimizations found in off-the-shelf C compilers are applicable to the generated code, thus yielding a highly efficient executable.

Part III.

Scalability Through Modular Analysis

6. Modular Analysis of Executables using On-Demand Heyting Completion

One challenge in static analysis is the sheer size of the input program. This is particularly true for the analysis of executables that have easily an order of magnitude more statements than the corresponding source program. One key to scalability is the treatment of functions: On the one hand, the highest precision needed to prove the absence of run-time errors [4] can be obtained by inlining functions at each call site with the cost of increasing the code to be analyzed dramatically. On the other hand, the duplicate evaluation of code can be avoided by performing a context-insensitive analysis in which all calling contexts of a function are merged and the return state is propagated to all call sites. A context-sensitive analysis without duplicate evaluation of functions can be obtained by inferring an input/output relation for each function. These function summaries are then combined to obtain a solution to the whole program using a global fixpoint computation. This approach is known as modular analysis [14].

We illustrate the challenges of a modular analysis using the code in Fig. 6.1. Here, the tests `CheckEven` and `CheckOdd` rely on the helper function `Check` to test an invariant of the two sub-classes `Even` and `Odd`. In a modular analysis, the methods `Even::IsEven` and `Even::IsOdd` are summarized by their effect of returning a constant value. The `Odd::IsEven` method modifies the `even_call` field pointed-to by `this`. A summary for this method must therefore assume the existence of a memory region at `*this` containing an `int` field. A precise summary of this function can be expressed by $x' = x + 1$ where x, x' is the value of the field before, resp. after, the call[1]. A more challenging task is the summary of `Check`. Invoking the virtual methods accessed through the `parity` pointer amounts to an indirect function call. Without knowing which functions can be dispatched to, a summary of this function would have to make worst case assumptions: the invoked function may modify any memory reachable from global variables or the `this` pointer. Without any additional information, a summary of a function f_i containing indirect calls provides little or no information.

One way to ensure that no precision loss occurs, even in the presence of higher-order functions, is to use only abstract domains that are able to condense the effect of a

[1]Note that using field names requires a well-typed program. As discussed later, our implementation therefore uses offsets instead of field names.

```
struct Parity {
  virtual bool IsEven() = 0;
  virtual bool IsOdd() = 0;
};

struct Even : public Parity {
  bool IsEven() { return true; }
  bool IsOdd() { return false; }
};

struct Odd : public Parity {
  bool IsEven() {
    even_call++;
    return false;
  }
  bool IsOdd() { return true; }
  int even_call = 0;
};
```

```
void CheckEven() {
  Even even;
  Check(&even);
}

void CheckOdd() {
  Odd odd;
  Check(&odd);
  assert(odd.even_call > 0);
}

void Check(Parity* parity) {
  assert(parity->IsEven()
      != parity->IsOdd());
}
```

Figure 6.1.: The running example C++ program.

function without loss of precision. By using these so-called condensing domains [25], it is possible to compute a summary of a function even if it takes other functions as parameters. Examples are type inference for functional programs [54], groundness analysis in Prolog [40] and instances of the IFDS framework [46]. These well-known domains are too imprecise to distinguish function behaviors based on pointer aliasing and numeric properties.

One particular kind of condensing domains are those whose meet distributes over the join of the lattice, i.e. $s \sqcap (t \sqcup u) = (s \sqcap t) \sqcup (s \sqcap u)$. Giacobazzi and Scozzari propose Heyting completion to make an existing domain meet-distributive [26]. This process adds new elements to a domain and may thereby refine an abstract domain until it is isomorphic to the concrete domain (which is a set of states and thus forms a distributive lattice). Heyting completion is therefore not generally practical. In this work, we use Heyting completion on-demand, namely when the analysis of a function requires it to avoid a severe loss of precision. In particular, once a particular property p is identified for which we want to avoid the lossy approximation $\{p\} \sqcap (s \sqcup t) \sqsupset (\{p\} \sqcap s) \sqcup (\{p\} \sqcap t)$, we track a new abstract state $p \to (\{p\} \sqcap s) \sqcup (\{p\} \sqcap t)$ and postpone the computation of a state in which p does not hold until a call site is

encountered that requires it. Ultimately, a function is summarized by a table $[p_1 \mapsto \{p_1\} \sqcap s_1, \ \ldots \ p_n \mapsto \{p_n\} \sqcap s_n]$ and a call site c applies this summary by computing $\bigsqcup_i c \sqcap \{p_i\} \sqcap s_i$. We present an analysis whose predicates p state that an input to a function is equal to a function address. For instance, analyzing `CheckEven` creates a summary s_E of `Check` and stores the mapping (`parity->vtable[0]` $= a_E) \mapsto s_E$ where a_E is the address of `Even::IsEven`. A second summary of `Check` is created for the call site in `CheckOdd`. A call site such as `Check(rand() ? new Odd() : new Even())` can thereafter be evaluated by instantiating the two summaries and without re-analyzing `Check`.

Given a function with the predicated summary $[p_1 \mapsto s_1, \ \ldots \ p_n \mapsto s_n]$ and a call site with state c, the question arises if the predicates cover the state described by c, i.e. if $\gamma(c) \subseteq \gamma(p_1) \cup \ldots \cup \gamma(p_n)$. If not, new predicates $p_{n+1}, \ \ldots \ p_{n+k}$ must be identified and a new summary must be computed for each predicate. For instance, calling `Check` with a new sub-class `Mark` whose method `Mark::IsEven` has address a_M, the computation of a new summary s_M of `Check` is needed, giving the table entry (`parity->vtable[0]` $= a_M) \mapsto s_M$. The challenge here is how to observe when a new predicate is needed and how to obtain it. Our contribution to this end is to represent predicates as a Herbrand abstraction (uninterpreted terms with variables as placeholder for other terms) which gives the analyzer the flexibility to express cross-cutting properties from several abstract domains. By evaluating these predicates wrt. a call site state, the variables in the predicates will be instantiated with values that make the predicate true. Each variable assignment of a predicate gives a ground (i.e. fully instantiated) Herbrand term. A summary of the function is computed for each ground Herbrand term.

In summary, we make the following contributions towards modular analysis:

- We apply Heyting completion [26] on-demand in order to make the summary of a function complete for some predicate. Predicates are created on-demand, namely when incompleteness would lead to an unusably imprecise result.
- We propose Herbrand abstractions to express symbolic predicates that functions postulate and that call-sites instantiate, thereby providing an abstract interface between the base analysis and the completion mechanism.
- We present an implementation of this framework using an inter-procedural control-flow-graph analysis that is able to resolve function calls in an x86 executable compiled from our higher-order functional language GDSL.

The remainder of this chapter is organized as follows: The next section defines a collecting and abstract semantics for an imperative language with indirect function calls. Section 6.2 generalizes these semantics to one that relates function inputs to outputs.

$$
\begin{array}{rcl@{\qquad}rcl}
Prog & ::= & FDecl^* & Stmt & ::= & Loc_S : \ br \ \underline{(}Expr : Loc_S;\underline{)?} \ Loc_S \\
FDecl & ::= & \texttt{ident}()\{Stmt^*\} & & | & Loc_S : \ Lhs = Expr \\
Lhs & ::= & \texttt{ident.field}\underline{(}{\rightarrow}\texttt{field}\underline{)?} & & | & Loc_S : \ call \ Expr \\
Expr & ::= & Lhs \mid Loc & & | & Loc_S : \ return
\end{array}
$$

Figure 6.2.: The abstract grammar of the analyzed program. $\underline{(}E\underline{)?}$ denotes zero or one E.

Section 6.3 enhances this abstract interpretation with the generation of Herbrand terms and presents how a fixpoint is obtained in a modular way. Section 6.4 discusses our implementation before Sect. 6.5 presents related work.

6.1. Preliminary Definitions

In this section we define a language with functions and define a collecting semantics for it. Let $[]$ denote an empty map, $m := [k_1 \mapsto v_1, \ \dots \ k_n \mapsto v_n]$ a map where n values can be looked up with $m[k_i] = v_i$, let $m \setminus k$ denote a map without a mapping for k and let $m[k \mapsto v]$ denote an update at k. Let $\text{dom}(m)$ denote the keys in m. Let $Loc = Loc_S \uplus Loc_M$ be the set of memory locations of a program P that is partitioned into statement labels Loc_S and statically and dynamically allocated memory regions Loc_M. Define $Loc_F \subseteq Loc_S$ to be the set of function entry points which coincide with the first statement in each function. We assume a C-like language where a variable v is stored at address $\&v \in Loc_M$. Let $\sigma \in \Sigma : Loc_M \to (\mathcal{F} \to \mathbb{V})$ define the program state with $\sigma(m)$ being a field map of the memory at address $m \in Loc_M$. A field map takes field names $\mathcal{F}$ to their content $\mathbb{V}$ where $\mathbb{V} := Loc \cup \mathbb{Z}$ denotes numeric values and addresses. The ability to partition a memory region into fields allows our analysis to express that a function call only accesses some but not all fields of a memory region.

Figure 6.2 defines the grammar of $P \in \mathcal{L}(Prog)$. A function is a sequence of statements consisting of conditional jumps, assignments, function calls, and returns. Note that every statement is preceded by its unique address $l \in Loc_S$. The statement $Lhs = Expr$ updates the specified field of a memory region or, via the optional C arrow notation, a field in the pointed-to memory region. For brevity, we write `even_call` for `Even::IsEven:this.this->even_call` $\in \mathcal{L}(Lhs)$ (where `Even::IsEven` is the method in Fig. 6.1). The concrete semantics of a statement takes an input program state $\sigma \in \Sigma$ and returns a tuple consisting of the output state and the location where execution continues. The individual rules are explained below.

$$
\begin{aligned}
[\![\cdot]\!]^\natural \quad &: \quad \mathcal{L}(Stmt) \times \Sigma \to (Loc_S \times \Sigma) \\
[\![l_s\text{: } br\ e : l_t; l_f]\!]^\natural \sigma \quad &= \quad \begin{cases} \langle l_t,\ \sigma\rangle & \text{if } [\![e]\!]^\natural_{Expr}\sigma = 0 \\ \langle l_f,\ \sigma\rangle & \text{otherwise} \end{cases} && (6.1) \\
[\![l_s\text{: } \texttt{m.f} = e]\!]^\natural \sigma \quad &= \quad \langle next(l_s),\ \sigma[m \mapsto \sigma(m)[f \mapsto v]]\rangle \text{ where } v = [\![e]\!]^\natural_{Expr}\sigma && (6.2) \\
[\![l_s\text{: } \texttt{m.f} \to \texttt{f}' = e]\!]^\natural \sigma \quad &= \quad [\![l_s\text{: } \texttt{m}'.\texttt{f}' = e]\!]^\natural \sigma \text{ where } \&m' = [\![\texttt{m.f}]\!]^\natural_{Expr}\sigma && (6.3) \\
[\![l_s\text{: } call\ e]\!]^\natural \sigma \quad &= \quad \langle \&\text{f},\ \sigma[\texttt{f} \mapsto [\texttt{ret} \mapsto next(l_s)]]\rangle \text{ where } \&\texttt{f} = [\![e]\!]^\natural_{Expr}\sigma && (6.4) \\
[\![l_s\text{: } return]\!]^\natural \sigma \quad &= \quad \langle l_r,\ \sigma \setminus \texttt{f}\rangle \text{ where } l_r = [\![\texttt{f.ret}]\!]^\natural_{Expr}\sigma && (6.5)
\end{aligned}
$$

The evaluation of an expression $e \in \mathcal{L}(Expr)$ is defined as follows:

$$
\begin{aligned}
[\![\cdot]\!]^\natural_{Expr} \quad &: \quad \mathcal{L}(Expr) \times \Sigma \to \mathbb{V} \\
[\![\texttt{m.f}]\!]^\natural_{Expr}\sigma \quad &= \quad \sigma(m)(f) && (6.6) \\
[\![\texttt{m.f} \to \texttt{f}']\!]^\natural_{Expr}\sigma \quad &= \quad [\![\texttt{m}'.\texttt{f}']\!]^\natural_{Expr}\sigma \text{ where } \&m' = [\![\texttt{m.f}]\!]^\natural_{Expr}\sigma && (6.7) \\
[\![l]\!]^\natural_{Expr}\sigma \quad &= \quad l \qquad\qquad (\text{rule } Expr ::= Loc) && (6.8)
\end{aligned}
$$

Jumps, defined by Eqn. 6.1, are unconditional if $e\ : l_t$ is omitted. Equation 6.2 updates the field f in $\sigma(m)$. It returns the program location following this statement using a function $next : Loc_S \to Loc_S$ that we assume to be suitably defined for all non-branching statements. A write through a pointer in Eqn. 6.3 assumes that the pointer value `m.f` matches the beginning of a memory region m' and is undefined otherwise. Thus, we do not model general pointer arithmetic and array accesses but assume that `parity->vtable[0]` is interpreted such that `vtable[0]` is a field name. Our implementation supports general pointer arithmetic.

The call instruction in Eqn. 6.4 continues execution at the called function. For a called function `f`, it creates a memory region `f` and a region `f:var` for each local variable `var` of `f` which altogether serve as the stack frame. We denote the region `f:var` of a local variable by `var` if the function is clear from the context. A variable of base type (e.g. an integer or a pointer) corresponds to a memory region with a single field that has the same name as the region itself. The return instruction in Eqn. 6.5 jumps to the location in the field `f.ret`, where `f` is the current function. (Note that supporting recursion requires the use of unique names for stack frames.) The semantics of expressions in Eqns. 6.6 to 6.8 is straight forward.

A suitable collecting semantics is the classic merge-over-all-paths solution. Let $\Sigma_s \subseteq \Sigma$ be the initial state at the program entry point l_{main}. We define:

Definition 1. *The collecting semantics of P is a map $col_P : Loc_S \to \wp(\Sigma)$ satisfying $\Sigma_s \subseteq col_P(l_{main})$ and for all $l : stmt \in P$, $\sigma \in col_P(l)$, and $\langle l', \sigma' \rangle = [\![l\colon stmt]\!]^\natural(\sigma)$ it holds that $\sigma' \in col_P(l')$.*

The structure $\langle Loc_S \to \wp(\Sigma), \dot{\subseteq}, \dot{\cup} \rangle$ is the complete partial order of the concrete domain where $\dot{\subseteq}$ and $\dot{\cup}$ are the point-wise liftings of the corresponding operations on the images of the map. The next section details how it is approximated by an abstract domain.

6.1.1. Abstract Interpretation of the Collecting Semantics

The segregation of memory into distinct regions lies at the heart of a modular analysis where a function summary leaves all but a small set of memory regions untouched. We therefore lift the concept of a memory region to the abstract.

Specifically, an abstract interpretation of the collecting semantics abstracts the unbounded set of memory regions in the concrete environments Σ by a bounded set of abstract memory regions $\mathcal{R}$. The memory regions define a set of non-overlapping areas of memory. The structure of a memory region $r \in \mathcal{R}$ is defined by a map $MS = \mathcal{R} \to (\mathcal{F} \to \mathcal{X})$ whose mappings are written $[r_1 \mapsto \phi_1^\sharp, \ldots, r_n \mapsto \phi_n^\sharp]$ where each $\phi_i^\sharp$ maps fields of a memory region r_i to a value domain variable $x \in \mathcal{X}$ that takes on values in $\mathbb{V} = \mathbb{Z} \cup Loc$.

The values of $X \subseteq \mathcal{X}$ are given by a domain $\mathcal{D}_X = \langle D_X, \sqsubseteq_{D_X}, \sqcup_{D_X}, \sqcap_{D_X}, \bot_D \rangle$. Here, X is the support set of $\mathcal{D}_X$, that is, the variables that D_X restricts. In our implementation, $\mathcal{D}_X$ is a reduced product [13] of several abstract domains. Since the inference of summaries requires the ability to express relations between input and output variables, a domain $d \in D_X$ must be concretized in a way that retains these relations. Thus, the concretization $\gamma_{\mathcal{D}_\mathcal{X}} : D_X \to \wp(\mathbb{V}^*)$ maps $d \in D_X$ to $\gamma_{\mathcal{D}_X}(d) = \{\vec{v}_1, \ldots\}$ where each vector $\vec{v}_i$ has one dimension for each abstract variable $x \in \mathcal{X}$. For instance, let $d \in D_{\{x, y\}}$ have its variables restricted by the interval constraint $x \in [3, 5]$ and the equality $x + 1 = y$ then $\langle x, y \rangle \in \gamma_\mathcal{D}(d) = \{\langle 3, 4 \rangle, \langle 4, 5 \rangle, \langle 5, 6 \rangle\}$. We write $\vec{v}(x)$ to extract the value from the vector corresponding to the dimension $x \in \mathcal{X}$. Changes to the support set X of a domain $\mathcal{D}_X$ are implemented by two functions $addVar_x : \mathcal{D}_X \to \mathcal{D}_{X \uplus \{x\}}$ (leaving x unrestricted) and $delVar_x : \mathcal{D}_{X \uplus \{x\}} \to \mathcal{D}_X$ that are defined iff $x \notin X$.

6.1.1.1. Combining Memory Structure and Value Domain

We now describe how MS and $\mathcal{D}_X$ are combined. For the sake of this section, let $vars(ms) \subseteq \mathcal{X}$ denote the variables occurring in $ms \in MS$. The lattice of our analysis contains elements $\langle m, d \rangle \in MS \times \{\mathcal{D}_X \mid X \subseteq \mathcal{X}\}$ such that $d \in \mathcal{D}_{vars(m)}$. We denote

this universe as $MS \times D$. The concretization of $MS \times D$ to environments Σ proceeds in three steps: First, we define a function *embed* that updates an environment $\sigma \in \Sigma$ with the values in a vector $\vec{v} \in \mathbb{V}^*$ based on the fields of a memory region. The function recursively processes each mapping by pattern matching against the empty map and a map $\{r \mapsto \phi^\sharp\} \uplus m$ containing a mapping for region r and other mappings m:

$$
\begin{aligned}
&embed : MS \times (Loc \cup \mathbb{Z})^* \times \Sigma \to \Sigma\\
&embed([], \vec{v}, \sigma) = \sigma\\
&embed([r \mapsto \phi^\sharp] \uplus m, \vec{v}, \sigma) = embed(m, \vec{v}, \sigma[r \mapsto embed_\phi(\sigma(r), \phi^\sharp, \vec{v})])\\
&\text{where } embed_\phi(m, \phi^\sharp, \vec{v}) = m[f \mapsto \vec{v}(\phi^\sharp(f)) \mid f \in \mathrm{dom}(\phi^\sharp)]
\end{aligned}
$$

Second, we apply the function *embed* to the set of all concrete stores Σ, thereby obtaining $\{embed(m, \vec{v}, \sigma) \mid \sigma \in \Sigma\}$, the set of concrete stores in which the fields tracked by the abstract domain are restricted to values in $\vec{v}$. The final step is to compute this set for each value vector, giving the concretization function:

$$
\begin{aligned}
\gamma_{MS \times D} &: MS \times D \to \wp(\Sigma)\\
\gamma_{MS \times D}(\langle m, d\rangle) &= \bigcup_{\vec{v} \in \gamma_{\mathcal{D}}(d)} \{embed(m, \vec{v}, \sigma) \mid \sigma \in \Sigma\}
\end{aligned}
$$

We now address the task of defining the lattice operations on $MS \times D$. The problem to address is that two structures $m_1, m_2 \in MS$, that are propagated to the same program point, are associated with domains $d_i \in \mathcal{D}_{vars(m_i)}$, $i = 1, 2$, so that d_1 and d_2 range over different variables and cannot be compared or joined.

We address this problem using cofibered abstract domains [61] and define three sound morphisms[2] $addRegion_r$, $addField_{r,f}$, $renameField_f : MS \times D \to MS \times D$ that are

[2]In categorical terms, $MS \times D$ is a Grothendieck construction $F \rtimes C$ using functor $F : C \to \mathbf{Cat}$ where C is a small category with $obj(C) = MS$ and $\mathbf{Cat}$ is a category of small categories with $obj(\mathbf{Cat}) = \{\langle \mathcal{D}_X, \rho\rangle \mid X \subseteq \mathcal{X}, \rho : X \to (Loc_M \times \mathcal{F})\}$. Here, the translation ρ provides information on how X relates to the field names of memory regions. F maps a category of memory structures to a category of domains over variables in that memory structure. Thus, the category $F \rtimes C$ contains tuples $\langle m, \langle d, \rho\rangle\rangle \in obj(F \rtimes C)$ where $m \in MS$ and $d \in \mathcal{D}_{vars(m)}$. The morphisms $\langle m_1, \langle d_1, \rho_1\rangle\rangle \xrightarrow{(f,g)} \langle m_2, \langle d_2, \rho_2\rangle\rangle \in hom_{F \rtimes C}$ are pairs (f, g) where $m_1 \xrightarrow{f} m_2$ is a functor in C and g is a morphism $F(f)(\langle d_1, \rho_1\rangle) \xrightarrow{g} \langle d_2, \rho_2\rangle$ in $\mathbf{Cat}$. A morphism is sound if g defines an inclusion relation between its arguments [61] which is given if the values of d_1 are a subset of those in d_2 modulo the translation of variables: $g(\langle d_1, \rho_1\rangle, \langle d_2, \rho_2\rangle)$ iff $\forall \vec{v}_1 \in \gamma_{\mathcal{D}_\mathcal{X}}(d_1). \exists \vec{v}_2 \in \gamma_{\mathcal{D}_\mathcal{X}}(d_2). \forall x \in \mathrm{dom}(\rho_1) \wedge \rho_1(x) \in \mathrm{dom}(\rho_2^{-1}) . \vec{v}_1(x) = \vec{v}_2(\rho_2^{-1}(\rho_1(x)))$. We omit ρ when defining morphisms as it is not needed.

applied if the memory structures ms_1, ms_2 differ:

$$\langle m,\ d\rangle \xrightarrow{addRegion_r} \langle m[r \mapsto []],\ d\rangle \tag{6.9}$$

$$\langle [r \mapsto \phi^\sharp] \uplus m,\ d\rangle \xrightarrow{addField_{r,f}} \langle [r \mapsto \phi^\sharp[f \mapsto x]] \uplus m,\ addVar_x(d)\rangle \tag{6.10}$$

$$\langle [r \mapsto \phi^\sharp[f \mapsto x]] \uplus m,\ d\rangle \xrightarrow{renameField_{f,x,y}} \langle\ r \mapsto \phi^\sharp[f \mapsto y]] \uplus m,\ delVar_x([\![y := x]\!]^\sharp addVar_y(d))\rangle \tag{6.11}$$

Here, $[\![y := x]\!]^\sharp$ in Eqn. 6.11 is the update transformer on $\mathcal{D}_X$. By applying a composition of the three morphisms on the domain tuples $\langle m_i,\ d_i\rangle$, $i = 1,\ 2$, one can obtain tuples $\langle m'_i,\ d'_i\rangle$ with $m'_1 = m'_2$ so that the lattice operations $\sqsubseteq_{D_X}$, $\sqcup_{D_X}$ can be applied to d'_i. The morphisms can be shown as sound wrt. $\gamma_{MS\times D}$ and we obtain the abstract lattice $\langle MS \times D,\ \sqsubseteq_{MS\times D},\ \sqcup_{MS\times D},\ \perp_{MS\times D}\rangle$.

Example 1. *We give an intuition on where the above morphisms are applied using an alias domain with universe $D_X = X \to \wp(Loc \cup \{a_{bad}\})$. It implements $addVar_x$ adding the mapping $x \mapsto \{a_{bad}\}$ where a_{bad} is a symbolic constant that represents all illegal addresses. Consider the following two functions:*

```
1  void foo() {
2    struct { void* a; } s;
3    if (rand())
4      s.a = &f;
5  }
6
7  void bar() {
8    struct{ void* a; } s;
9    if(rand())
10     s.a = &f;
11   else
12     s.a = &g;
13 }
```

Assume that `s` *is initially associated with a region without fields, i.e. $s \mapsto []$. Assume further that, in* `foo` *and* `bar`*, the then-branch updates* `s` *such that $s \mapsto [a \mapsto x_1]$. For* `foo`*, we have to apply the $addField_{s,a}$ morphism on the else-branch state before the join; the join, consequently, results in the alias set $x_1 \mapsto \{a_{bad}, \&f\}$. In the else-branch of* `bar`*, the update creates, e.g., $s \mapsto [a \mapsto x_2]$. In this case, we have to apply $renameField_{a,x_2,x_1}$ so that the states to be joined have the same support set. The join results in $x_1 \mapsto \{\&f, \&g\}$ for the field* `a`.

The presented memory structures MS do not allow for summarized memory regions as every abstract memory region $r \in \mathcal{R}$ corresponds to exactly one concrete memory region in σ, albeit at varying addresses. Although this suffices to illustrate our modular analysis, our implementation requires a simple form of summarized memory regions as described in Sect. 6.4. A concretization that caters for summarized memory regions [52] would complicate the presentation unnecessarily.

6.2. Modular Program Semantics

In this section we generalize the collecting semantics and its abstract interpretation to function summaries. Specifically, we summarize the behavior of a function by a set of tuples $\langle \sigma, \bar{\sigma} \rangle$ that state how an input environment σ is mapped to an output environment $\bar{\sigma}$ and lift this relation to an abstract input/output relation.

We first define the input/output function semantics for a single input state. Recall that the semantics of calling f and returning from f in Eqns. 6.4 and 6.5 use the field `f.ret` to store the return address. In order to define the semantics of f independently of a caller, we evaluate it in an environment $\sigma = [\texttt{f.ret} \mapsto l_f^{res}]$ where $l_f^{res} \in Loc$ is a location that is not used in P.

Definition 2. *The semantics of f at $l_f \in Loc_S$ and executing in state σ is a map $col_f^\sigma : Loc_S \to \wp(\Sigma)$ satisfying $\sigma[\texttt{f.ret} \mapsto l_f^{res}] \subseteq col_f^\sigma(l_f)$ and for all $l : stmt \in P$, $\sigma' \in col_f^\sigma(l)$, and $\langle \sigma'', l' \rangle = [\![l \colon stmt]\!]^\natural(\sigma')$ it holds that $\sigma'' \in col_f^\sigma(l')$.*

We use the previous definition to define the relational semantics of f, that is, how each input state relates to the states at each statement of f:

Definition 3. *The relational semantics $rel_f : Loc_S \to \wp(\Sigma \times \Sigma)$ of a function f is given by $rel_f(l) = \{\langle \sigma, \bar{\sigma} \rangle \mid \sigma \in \Sigma \wedge \bar{\sigma} \in col_f^\sigma(l)\}$.*

Observe that rel_f is defined in terms of Eqn. 6.4 which defines the semantics of a call to evaluate the called function rather than using the summary rel_f. We therefore use the following definition from now on:

$$[\![l_s \colon call\ e]\!]^\natural \sigma \quad = \quad \langle next(l_s),\ \bar{\sigma} \rangle \text{ where } \&\text{f} = [\![e]\!]^\natural_{Expr} \sigma \wedge \langle \sigma,\ \bar{\sigma} \rangle \in rel_f(l_f^{res}) \qquad (6.12)$$

6.2.1. Abstract Interpretation of the Relational Semantics

The relational semantics of a function is approximated by an abstract domain $MS^2 \times D$ that is used to abstract $rel_f(l)$ for all locations $l \in Loc_S$ within function f. Here, $MS^2 = MS \times MS$ are two memory structures, the first describing the memory at the

entry point of f, the second describing the memory at l. The relation between the abstract and the concrete domain is given by $\gamma_{MS^2 \times D}$:

$$\begin{aligned} \gamma_{MS^2 \times D} &: MS^2 \times D \to \wp(\Sigma \times \Sigma) \\ \gamma_{MS^2 \times D}(\langle m_{in}, m_{out}, d \rangle) &= \bigcup_{\vec{v} \in \gamma_{\mathcal{D}}(d)} \{ \langle embed(m_{in}, \vec{v}, \sigma), embed(m_{out}, \vec{v}, \sigma) \rangle \mid \sigma \in \Sigma \} \end{aligned}$$

The concretization retains the relational character of rel_l in two ways: first, the *embed* functions are applied on the same numeric vector $\vec{v} \in \mathbb{Z}^*$ so that relational information between numeric variables are manifest in the concrete states. Second, the information of the abstract domain is embedded into the same $\sigma \in \Sigma$. As a consequence, a field in any concrete memory region in σ that is not present in either m_{in} nor m_{out} is not altered. These relational properties are illustrated in the following example:

Example 2. *Let $l_e \in Loc_F$ be the entry point of the method* `Odd::IsEven()` *in Fig. 6.1. The relational semantics at l_e is the identity, that is, $rel_{l_e}(l_e) = \{\langle \sigma, \sigma \rangle \mid \sigma \in \Sigma\} = \gamma_{MS^2 \times D}(\langle m_{in}, m_{out}, d \rangle)$ where $m_{in} = m_{out} = []$ and $d \in \mathcal{D}_\emptyset$. Let $l_i \in Loc_S$ denote the location after the* `even_call++` *statement, then $\langle \sigma_{in}, \sigma_{out} \rangle \in rel_{l_e}(l_i)$ contains a memory region o at $l_o \in Loc_M$ that contains the object instance. An abstract state $s = \langle m_1, m_2, d \rangle \in MS^2 \times D$ with $rel_{l_e}(l_i) \in \gamma_{MS^2 \times D}(s)$ is $m_i = [$*`Odd::IsEven:this` *$\mapsto [$*`this` *$\mapsto y^i_{val}], o \mapsto [$*`even_call` *$\mapsto x^i_{val}]], i = 1, 2$ and a value domain $d \in D$ containing the constraints $y^1_{val} = y^2_{val} = l_o$ and $x^1_{val} + 1 = x^2_{val}$.*

The algebra $\langle MS^2 \times D, \sqsubseteq_{MS^2 \times D}, \sqcup_{MS^2 \times D}, \perp_{MS^2 \times D}, \bowtie_{MS^2 \times D} \rangle$ defines the abstract domain. Here, $\bowtie_{MS^2 \times D}$ is a special meet operator that combines the current state in a caller with a function summary. It is explained below. Other operations can be reduced to $\mathcal{D}$ using the following morphisms:

$$\langle m_1, m_2, d \rangle \xrightarrow{addRegion_r} \langle m_1[r \mapsto []], m_2[r \mapsto []], d \rangle \quad (6.13)$$

$$\langle [r \mapsto \phi^\sharp_1] \uplus m_1, m_2, d \rangle \xrightarrow{addField^1_{r,f}} \langle [r \mapsto \phi^\sharp_1[f \mapsto x_1]] \uplus m_1, m_2, addVar_{x_1}(d) \rangle \quad (6.14)$$

$$\langle [r \mapsto \phi^\sharp_1 \uplus [f \mapsto x_1]] \uplus m_1, [r \mapsto \phi^\sharp_2] \uplus m_2, d \rangle \xrightarrow{addField^2_{r,f}} \langle [r \mapsto \phi^\sharp_1 \uplus [f \mapsto x_1]] \uplus m_1, [r \mapsto \phi^\sharp_2[f \mapsto x_2]] \uplus m_2, [\![x_2 := x_1]\!]^\sharp addVar_{x_2}(d) \rangle \quad (6.15)$$

$$\langle [r \mapsto \phi^\sharp[f \mapsto x]] \uplus m_1, m_2, d \rangle \xrightarrow{renameField^1_{f,x,y}} \langle [r \mapsto \phi^\sharp[f \mapsto y]] \uplus m_1, m_2, delVar_x([\![y := x]\!]^\sharp addVar_y(d)) \rangle \quad (6.16)$$

$$\begin{array}{lcl} \langle\ [r_{from} \mapsto \phi_1^\sharp] \uplus m_1, & \xrightarrow{renameRegion_{r_{from},\, r_{to}}} & \langle\ [r_{to} \mapsto \phi_1^\sharp] \uplus m_1, \\ \quad [r_{from} \mapsto \phi_2^\sharp] \uplus m_2,\ d\rangle & & \quad [r_{to} \mapsto \phi_2^\sharp] \uplus m_2,\ d\rangle \end{array} \tag{6.17}$$

One obvious difference between these morphisms and those in Eqns. 6.9 - 6.11 is that they operate on two memory structures, namely the input ms_1 and the current state m_2 that eventually becomes the output state. In Eqn. 6.13, $addRegion_r$ is defined such that a region is always added in both the input and output memory structure, thereby ensuring that $\text{dom}(m_1) = \text{dom}(m_2)$ at all times. Note that this is motivated by the fact that each memory region that is present in the output structure of a function *fun* must have had some state in the input of *fun*, even if it is the result of a memory allocation within *fun*. In Eqn. 6.14 resp. 6.15 we define morphisms that add a variable and field to a region of the input resp. output memory structure. The morphism $addField^2_{r,\, f}$ is only used if the respective variable is already contained in m_2. It additionally makes sure that the output variable is made equal to the input variable in the numeric domain, so that the domain maps each value of the field in the input to the same value in the output. When accessing an unknown field in m_2, these two morphisms need to used together. For this, we define the helper function $addField_{r,\, f}$ as follows:

$$addField_{r,\, f}(m_1,\ m_2,\ d) \quad = \quad \begin{cases} addField^2_{r,\, f}(m_1,\ m_2,\ d) & f \in \text{dom}(m_1(r)) \\ addField^2_{r,\, f} \circ & \\ \quad addField^1_{r,\, f}(m_1,\ m_2,\ d) & otherwise \end{cases} \tag{6.18}$$

Analogous to Eqn. 6.11, Eqn. 6.16 renames the value domain variable of a field f in m_1 from x to y. We omit the symmetric definition $renameField^2_f$ that renames a variable of a field in m_2 for brevity. Eqn. 6.17 contains the definition of $renameRegion_{r_{from},\, r_{to}}$ which renames region r_{from} to r_{to} in both the input and the output memory structure.

6.2.2. Abstract Semantics of Memory Accesses

This section details the abstract semantics of memory accesses and illustrates how to deal with accesses to unknown locations. Figure 6.3 presents the abstract semantics for expressions (abstracting Eqns. 6.6 and 6.7 by Eqns. 6.19 and 6.20, respectively) and assignments (abstracting Eqns. 6.2 and 6.3 by Eqns. 6.21 and 6.22, respectively).

The expression semantics returns a set of variables or locations so that Eqn. 6.20 can return one variable for each dereferenced pointer. Note here that $\gamma_{\mathcal{D}}$ returns vectors of possible values and that $ms_2(m)(f)$ returns the domain variable that is used to index into the vector. As shown in Eqn. 6.21, each element returned by the expression semantics is assigned by Eqn. 6.2 and the various results are joined. Equation 6.22

$$\begin{aligned}
[\![\cdot]\!]^{\sharp}_{Expr} &: \mathcal{L}(Expr) \times (MS^2 \times D) \to \wp(\mathcal{X} \cup Loc) \\
[\![\mathtt{m.f}]\!]^{\sharp}_{Expr}\langle ms_1,\ ms_2,\ d\rangle &= \{ms_2(m)(f)\} && (6.19) \\
[\![\mathtt{m.f} \to \mathtt{f}']\!]^{\sharp}_{Expr}\langle ms_1,\ ms_2,\ d\rangle &= \bigcup_{\&m' \in \gamma_{\mathcal{D}}(d)(ms_2(m)(f))} [\![\mathtt{m}'.\mathtt{f}']\!]^{\sharp}_{Expr}\langle ms_1,\ ms_2,\ d\rangle && (6.20)
\end{aligned}$$

$$\begin{aligned}
[\![\cdot]\!]^{\sharp} &: \mathcal{L}(Stmt) \times (MS^2 \times D) \to \wp(Loc_S \times MS^2 \times D) \\
[\![l_s\colon \mathtt{m.f} = e]\!]^{\sharp}\langle ms_1,\ ms_2,\ d\rangle &= \{\langle next(l_s), && (6.21) \\
&\qquad \bigsqcup_{e' \in [\![e]\!]^{\sharp}_{Expr}\langle ms_1,\ ms_2,\ d\rangle} \langle ms_1,\ ms_2,\ [\![ms_2(m)(f) = e']\!]^{\sharp} d\rangle\rangle\} \\
[\![l_s\colon \mathtt{m.f} \to \mathtt{f}' = e]\!]^{\sharp}(s = \langle ms_1,\ ms_2,\ d\rangle) &= \{\langle next(l_s), && (6.22) \\
&\qquad \bigsqcup_{\&m' \in \gamma_{\mathcal{D}}(d)(ms_2(m)(f))} s' \text{ with } \{\langle l,\ s'\rangle\} = [\![l_s\colon \mathtt{m}'.\mathtt{f}' = e]\!]^{\sharp} s\rangle\} \\
[\![l_s\colon return]\!]^{\sharp}\langle ms_1,\ ms_2,\ d\rangle &= \{\langle l^{res}_f,\ \langle ms_1,\ ms_2,\ d\rangle\rangle\} && (6.23) \\
[\![l_s\colon br\ e : l_t; l_f]\!]^{\sharp} s &= \{\langle l_t,\ [\![test\ [\![e]\!]^{\sharp}_{Expr}\ s \neq \{0\}]\!]^{\sharp} s\rangle, \\
&\quad \langle l_f,\ [\![test\ [\![e]\!]^{\sharp}_{Expr}\ s = \{0\}]\!]^{\sharp} s\rangle\} && (6.24)
\end{aligned}$$

Figure 6.3.: Abstract Semantics (without Call).

computes the assignment via a pointer as the join of writing to all possible locations $\&m'$.

Note that the expression $ms_2(m)(f)$ is undefined when either the memory region m does not exist in ms_2 or it does not contain a field f. Rather than handling this case in the semantic definition, we assume that the morphisms in Eqn. 6.13 and 6.18 are applied to prevent undefinedness. In case the transformer would access an unknown location through a pointer (i.e. `m.f` in Eqn. 6.20 or 6.22), a new region r is added using Eqn. 6.13 and `m.f` is restricted to point to it. Note that this behavior is not sound as it assumes that `m.f` does not alias with any other function inputs which may be wrong. We discuss this design choice in Sect. 6.4.

Example 3. *We analyze* `even_count++` *in* `Odd::IsEven` *of Fig 6.1. Let* `f.this == &i` *be a test that forces* `this` *to point to the object instance i. For brevity, we use* `f` *for* `Odd::IsEven`, `ev` *for* `even_count`*, and write $d \in \mathcal{D}$ as set of constraints:*

$$\langle [], [], \emptyset \rangle \xrightarrow{addRegion_{\texttt{f:this}}} \langle [\texttt{f:this} \mapsto []], [\texttt{f:this} \mapsto []], \emptyset \rangle \xrightarrow{addField_{\texttt{f:this}, this}}$$
$$\langle [\texttt{f:this} \mapsto [this \mapsto x_1]], [\texttt{f:this} \mapsto [this \mapsto x_2]], \{x_1 = x_2\} \rangle \xrightarrow{addRegion_{\texttt{i}}} \xrightarrow{\texttt{f:this.this}==\&i}$$
$$\langle [\ldots, \texttt{i} \mapsto []], [\ldots, \texttt{i} \mapsto []], \{x_1 = x_2 = l_i\} \rangle \xrightarrow{addField_{\texttt{i}, ev}} \langle [\ldots,$$
$$\texttt{i} \mapsto [ev \mapsto x_3]], [\ldots, \texttt{i} \mapsto [ev \mapsto x_4]], \{x_1 = x_2 = l_i, x_3 = x_4\} \rangle \xrightarrow{[\![\texttt{f:this.this} \rightarrow ev{+}{+}]\!]^\sharp}$$
$$\langle [\ldots, \texttt{i} \mapsto [ev \mapsto x_3]], [\ldots, \texttt{i} \mapsto [ev \mapsto x_4]], \{x_1 = x_2 = l_i, x_3 + 1 = x_4\} \rangle$$

6.2.3. Application of Function Summaries

The idea of applying morphisms as a precursor to a domain operation is also the underlying idea for defining the $\bowtie_{MS^2 \times D}$ operation that combines a call site state $\langle m^1_{in}, m^1_{out}, d_1 \rangle$ with the summary of a function $\langle m^2_{in}, m^2_{out}, d_2 \rangle$. Assuming that the value domain states d_1 and d_2 share no variables, we define $\bowtie_{MS^2 \times D}$ in terms of a function $applySummary_{params, globals}$ which we discuss below:

$$\langle m^1_{in}, m^1_{out}, d_1 \rangle \bowtie_{MS^2 \times D} \langle m^2_{in}, m^2_{out}, d_2 \rangle = \\ applySummary_{params, globals}(m^1_{in}, m^1_{out}, d_1, m^2_{in}, m^2_{out}, d_2) \tag{6.46}$$

Note that the analyzer has to know globally defined variables $globals \subseteq \mathcal{R}$ and the function parameters $params \subseteq \mathcal{R} \times \mathcal{R}$ of the called function from the context of the call. Here, a function parameter is a tuple consisting of the actual parameter (caller memory region) and formal parameter (callee memory region). The definition of $applySummary_{params, globals}$ can be found in Fig. 6.4. The function commences by building a mapping $regm \subseteq \mathcal{R} \times \mathcal{R}$ that states which caller memory region corresponds to which callee memory region. The relation $regm$ is initialized in Eqn. 6.25 such that

$$
\begin{aligned}
&\mathit{applySummary}_{\mathit{params},\,\mathit{globals}} : MS^2 \times D \times MS^2 \times D \to MS^2 \times D \\
&\mathit{applySummary}_{\mathit{params},\,\mathit{globals}}(m^1_{in},\, m^1_{out},\, d_1,\, m^2_{in},\, m^2_{out},\, d_2) = \\
&\quad \text{let } \mathit{regm} = \mathit{params} \cup \{\langle g,\, g\rangle \mid g \in \mathit{globals} \wedge g \in \mathrm{dom}(m^2_{in})\} && (6.25)\\
&\quad \text{let } \langle \mathit{regm},\, m^1_{in},\, m^1_{out},\, d_1\rangle := && (6.26)\\
&\qquad \mathit{buildRegionMap}(\mathit{regm},\, m^1_{in},\, m^1_{out},\, d_1,\, m^2_{in},\, m^2_{out},\, d_2) && (6.27)\\
&\quad \text{for } r_{caller} \in \{r_{caller} \mid \langle r_{caller},\, _\rangle \in \mathit{regm}\} && (6.28)\\
&\qquad \text{let } rs_{callee} = \{r_{callee} \mid \langle r_{caller},\, r_{callee}\rangle \in \mathit{regm}\} && (6.29)\\
&\qquad \text{if } |rs_{callee}| \leq 1 \text{ then continue} && (6.30)\\
&\qquad \langle r_{combined},\, m^2_{in},\, m^2_{out},\, d_2\rangle := \mathit{handleCalleeAliasing}(rs_{callee},\, m^2_{in},\, m^2_{out},\, d_2) && (6.31)\\
&\qquad \mathit{regm} := (\mathit{regm} \setminus \{\langle r_{caller},\, r_{callee}\rangle \mid r_{callee} \in rs_{callee}\}) \uplus \{\langle r_{caller},\, r_{combined}\rangle\} && (6.32)\\
&\quad \text{let } \mathit{foldMap} = \emptyset && (6.33)\\
&\quad \text{for } r_{callee} \in \{r_{callee} \mid \langle _,\, r_{callee}\rangle \in \mathit{regm}\} && (6.34)\\
&\qquad \text{let } rs_{caller} = \{r_{caller} \mid \langle r_{caller},\, r_{callee}\rangle \in \mathit{regm}\} && (6.35)\\
&\qquad \text{if } |rs_{caller}| \leq 1 \text{ then continue} && (6.36)\\
&\qquad \langle r_{combined},\, m^1_{in},\, m^1_{out},\, d_1\rangle := \mathit{handleCallerAliasing}(rs_{caller},\, m^1_{in},\, m^1_{out},\, d_1) && (6.37)\\
&\qquad \mathit{regm} := (\mathit{regm} \setminus \{\langle r_{caller},\, r_{callee}\rangle \mid r_{caller} \in rs_{caller}\}) \uplus \{\langle r_{combined},\, r_{callee}\rangle\} && (6.38)\\
&\qquad \mathit{foldMap} := \mathit{foldMap} \uplus \{\langle r_{combined},\, rs_{caller}\rangle\} && (6.39)\\
&\quad \text{for } \langle r,\, r'\rangle \in \mathit{regm} && (6.40)\\
&\qquad \langle m^2_{in},\, m^2_{out},\, d_2\rangle := \mathit{renameRegion}_{r',\,r}(m^2_{in},\, m^2_{out},\, d_2) && (6.41)\\
&\quad \langle m^2_{in},\, m^2_{out},\, d_2\rangle := \mathit{addOrRenameFields}(m^1_{in},\, m^1_{out},\, d_1,\, m^2_{in},\, m^2_{out},\, d_2) && (6.42)\\
&\quad \text{let } d' = \mathit{delVar}_{\mathit{vars}(m^1_{out}) \cup \mathit{vars}(m^2_{in})}(\mathit{addVar}_{\mathit{vars}(m^2_{out})}(d_1) \sqcap_D \mathit{addVar}_{\mathit{vars}(m^1_{in})}(d_2)) && (6.43)\\
&\quad \langle m^1_{in},\, m^2_{out},\, d'\rangle := \mathit{expandCallerRegions}(\mathit{foldMap},\, m^1_{in},\, m^2_{out},\, d') && (6.44)\\
&\quad \text{return } \langle m^1_{in},\, m^2_{out},\, d'\rangle && (6.45)
\end{aligned}
$$

Figure 6.4.: Definition of the $\mathit{applySummary}_{\mathit{params},\,\mathit{globals}}$ function which applies a summary (last three arguments) to a call site state (first three arguments).

it associates each actual parameter with a formal parameter and each callee global with the same global within the caller. The remaining callee regions are associated by following pointers in both the callee and the caller. This is implemented by the function *buildRegionMap* which updates *regm* and is discussed in Sect. 6.2.3.1. It is our goal to use *regm* as basis for renaming the callee regions such that they match the caller regions. However, it is possible for the same caller (resp. callee) region to appear multiple times in the first (resp. second) position of tuples in *regm* as a result of pointer aliasing (see below for two example scenarios). In order to deal with this, we first look for caller regions that are associated with multiple callee regions (Eqn. 6.28 through Eqn. 6.32). For each such region r, we call *handleCalleeAliasing* in Eqn. 6.31 which is discussed in Sect. 6.2.3.3. The function returns an updated summary and a combined callee region $r_{combined}$. We replace all mappings in *regm* to regions in rs_{callee} by a single mapping to the combined region in Eqn. 6.32.

Next, we look for callee regions that are associated with multiple caller regions (Eqn. 6.33 through Eqn. 6.39). Our basic idea to handle this case is to derive different caller states for every possible aliasing configuration, apply the summary to each of these states, and finally join all of them into a new state for the caller after the call. However, given a function that accepts k parameters that each have an alias set of size n, this results in up to n^k many summary applications. Thus, the running time is exponential in the number of parameters in the worst case. As a consequence, this approach is not practical. Instead, we combine all caller regions associated with a single callee region. For this, we call *handleCallerAliasing* (see Sect. 6.2.3.4) for each callee region r_{callee} that is associated with multiple caller regions rs_{caller} in Eqn. 6.37. The function returns the modified caller state and – as was the case for *handleCalleeAliasing* –, the combined region $r_{combined}$. In Eqn. 6.38, we replace all occurrences of r_{callee} in *regm* with the one single association $\langle r_{combined}, r_{callee} \rangle$. In addition, we collect the combined regions in Eqn. 6.39 because the summary effects on them need to be propagated back to the original caller regions at the end of the summary application (see Eqn. 6.44 and Fig. 6.10 which are explained below).

After considering both caller and callee aliasing, we can be sure that *regm* is indeed a $1:1$ mapping between caller and callee regions. As a result, we are now able rename the callee regions based on *regm* as shown in Eqn. 6.40f so that they match the call site regions. In addition, we call *addOrRenameFields* in Eqn. 6.42 which makes sure that each region and field of the caller output memory structure exists in the callee input memory structure and that the respective value domain variables have the same name. The function is explained in detail in Sect. 6.2.3.5.

Finally, we compute the meet of the value domain states in Eqn. 6.43. Note that due to the fact the input value domain variables of the summary have been renamed to match the output value domain variables at the call site, this applies the effect of the

function to the call site state. Afterwards, we remove variables contained in m^1_{out} and m^2_{in} from the combined value domain state so that the resulting state d' only contains the input variables of the caller and the output variables that reflect the state after the call.

At the end of the function summary application, the contents of the combined caller regions need to be propagated back to the respective original caller regions. In order to do this, we call *expandCallerRegions* explained below in Eqn. 6.44 and pass *foldMap* as parameter.

The next subsections detail the individual steps in applying the function summary, namely computing the caller/callee relation *regm* in Sect. 6.2.3.1, the concept of merging memory regions in Sect. 6.2.3.2 which is used to handle callee (Sect. 6.2.3.3) and caller (Sect. 6.2.3.4) aliasing, and adding missing caller regions and fields to the function summary (Sect. 6.2.3.5).

6.2.3.1. Construction of the Caller/Callee Region Relation

This section details how *regm* is constructed which relates caller and callee memory regions. Recall that the callee memory regions were created on-demand when computing the summary of the callee and therefore have arbitrary names. We compute the relation $regm \subseteq \mathcal{R} \times \mathcal{R}$ between the caller and the callee memory regions by iteratively following pointers, starting with the actual and formal function arguments (as shown by the initialization in Eqn. 6.25). The matching is implemented through the function *buildRegionMap* shown in Fig. 6.5 which is called by *applySummary* in Eqn. 6.27. Note that a single callee region is associated with a set of caller regions. This is because we match the callee input memory structure to the caller output memory structure. Because the input memory structure is created through read accesses by assuming that input memory regions do not alias, the input regions form a tree. However, the points-to relationships in the output structure are arbitrary. Thus, dereferencing a pointer in the caller output memory structure can lead to multiple regions and, thus, associate a set of regions with a single callee region.

The matching proceeds using a queue *curr* of region associations from which we dequeue one element in Eqn. 6.49. For each callee field and caller region (Eqn. 6.50), Eqns. 6.51 through 6.54 ensure that a field and its region exist in the caller memory structure. Note that a region name can be introduced without an actual region through $addField_{r,f}$. Equation 6.55 queries the points-to set of the field in the callee which contains exactly one element p^2 because it is a field in the callee input. However, the region p^2 may not be tracked by the callee memory structure, in which case the traversal of the callee tree stops here (Eqn. 6.56). Otherwise, we query the caller aliases in Eqn. 6.57. Note that a freshly inserted caller field aliases its respective input field; if

$$\mathit{buildRegionMap} : \mathcal{R}^2 \times MS^2 \times D \times MS^2 \times D \to \mathcal{R}^2 \times MS^2 \times D$$

$$\mathit{buildRegionMap}(\mathit{regm},\ m^1_{in},\ m^1_{out},\ d_1,\ m^2_{in},\ m^2_{out},\ d_2) =$$

$$\text{let } \mathit{curr} = \{\langle\{r_1\},\ r_2\rangle \mid \langle r_1,\ r_2\rangle \in \mathit{regm}\} \tag{6.47}$$

$$\text{while } \exists\langle R_1,\ r_2\rangle \in \mathit{curr} \text{ do} \tag{6.48}$$

$$\mathit{curr} := \mathit{curr} \setminus \{\langle R_1,\ r_2\rangle\} \tag{6.49}$$

$$\text{for } f \in \{f \mid \langle f,\ _\rangle \in m^2_{in}(r_2)\},\ r_1 \in R_1 \text{ do} \tag{6.50}$$

$$\text{if } r_1 \notin \mathrm{dom}(m^1_{out}) \text{ then} \tag{6.51}$$

$$\langle m^1_{in},\ m^1_{out},\ d_1\rangle := \mathit{addRegion}_{r_1}(\langle m^1_{in},\ m^1_{out},\ d_1\rangle) \tag{6.52}$$

$$\text{if } f \notin \mathrm{dom}(m^1_{out}(r_1)) \text{ then} \tag{6.53}$$

$$\langle m^1_{in},\ m^1_{out},\ d_1\rangle := \mathit{addField}_{r_1,\ f}(\langle m^1_{in},\ m^1_{out},\ d_1\rangle) \tag{6.54}$$

$$\text{let } \{p^2\} = \mathit{queryPointsTo}(d_2,\ m^2_{in}(r_2)(f)) \tag{6.55}$$

$$\text{if } p^2 \notin \mathrm{dom}(m^2_{in}) = \emptyset \text{ then continue} \tag{6.56}$$

$$\text{let } P^1 = \mathit{queryPointsTo}(d_1,\ m^1_{out}(r_1)(f)) \tag{6.57}$$

$$\text{if } P^1 = \emptyset \text{ then issue invalid dereference warning and continue} \tag{6.58}$$

$$\mathit{regm} := \mathit{regm} \cup \{\langle p^1,\ p^2\rangle \mid p^1 \in P^1\} \tag{6.59}$$

$$\mathit{curr} := \{c \in \mathit{curr} = \langle R_1,\ r_2\rangle \mid r_2 \neq p^2\} \cup \tag{6.60}$$

$$\{\langle R_1 \cup P^1,\ p^2\rangle \mid \langle R_1,\ p^2\rangle \in \mathit{curr}\} \tag{6.61}$$

$$\text{return } \langle \mathit{regm},\ m^1_{in},\ m^1_{out},\ d_1\rangle \tag{6.62}$$

Figure 6.5.: Definition of *buildRegionMap* which constructs an association between caller and callee memory regions.

```
1  struct S {
2    int i;
3  };
4
5  S *s;
6
7  void g() {
8    s->i = 42;
9  }
10
11 void f() {
12   g();
13 }
```

Figure 6.6.: Example code that requires a caller region to be added during *buildRegionMap*.

the caller alias set is empty, this means the field has been overwritten by a non-pointer value and, thus, we output an invalid dereference warning in Eqn. 6.58. Otherwise, we update *regm* in Eqn. 6.59 and insert a new association into our queue in Eqns. 6.60f.

Note that because we follow pointers in the memory structure of the callee input and this callee input forms a tree, the algorithm is guaranteed to terminate.

Example 4. *Consider the code in Fig. 6.6. Note that the caller* `f` *of* `g` *does not initialize the variable* `s` *in line 5. As a result, the summary at the call site of* `g` *is empty. The summary of* `g`*, in contrast, contains the memory region resulting from the assignment to the memory pointed to by* `s`*, that is* $\langle[\texttt{s} \mapsto [s \mapsto x_1^{in}], \texttt{*s} \mapsto [i \mapsto x_2^{in}], \ldots], \langle\{\ldots\}, [x_1^{in} \mapsto \{\texttt{\&(*s)}\}, x_2^{in} \mapsto \{\texttt{\&(**s)}\}]\rangle, [\texttt{s} \mapsto [s \mapsto x_1^{out}], \texttt{*s} \mapsto [i \mapsto x_2^{out}], \ldots], \langle\{\ldots, x_2^{out} = 42\}, [x_1^{out} \mapsto \{\texttt{\&(*s)}\}, x_2^{out} \mapsto \{a_{bad}\}]\rangle\rangle$*. In buildRegionMap, we first encounter the callee region* `s` *which does not exist in the caller. As a result, we add* `s` *and its only field s in Eqns. 6.51f and Eqns. 6.53f, respectively, to the caller state. Dereferencing the one alias* `&(*s)` *in the alias set of* x_1^{in} *leads us to the region* `*s` *which is contained in the summary input memory structure. As a result,* $\langle\{\texttt{*s}\}, \texttt{*s}\rangle$ *is added to curr and the loop in Eqn. 6.48 is entered again. This time, Eqns. 6.51f and Eqns. 6.53f add the region* `*s` *and its field i to the caller state. In contrast to the last iteration, however, the one alias of* `&(**s)` *in the alias set of* x_2^{in} *is not tracked by the summary memory structure. As a result, curr is not updated and the main loop in Eqn. 6.48 exits.*

$$merge_{r_{dst},\, r_{src}} : MS^2 \times D \to MS^2 \times D$$
$$merge_{r_{dst},\, r_{src}}(m_{in},\, m_{out},\, d) =$$
$$\text{let } \langle \phi^\sharp_{dst},\, \phi^\sharp_{src} \rangle = \langle m_{out}(r_{dst}),\, m_{out}(r_{src}) \rangle \tag{6.63}$$
$$\text{for } f \in \mathrm{dom}(\phi^\sharp_{dst}) \cap \mathrm{dom}(\phi^\sharp_{src}) \text{ do} \tag{6.64}$$
$$d := d \sqcup_D ([\![\phi^\sharp_{dst}(f) := \phi^\sharp_{src}(f)]\!]^\sharp\, d) \tag{6.65}$$
$$\text{for } f \in \mathrm{dom}(\phi^\sharp_{src}) \setminus \mathrm{dom}(\phi^\sharp_{dst}) \text{ do} \tag{6.66}$$
$$\langle m_{in},\, m_{out},\, d \rangle := addField_{r_{dst},\, f}(\langle m_{in},\, m_{out},\, d \rangle) \tag{6.67}$$
$$d := [\![m_{out}(r_{dst})(f) := \phi^\sharp_{src}(f)]\!]^\sharp\, d \tag{6.68}$$
$$\text{return } \langle m_{in},\, m_{out},\, d \rangle \tag{6.69}$$

Figure 6.7.: Definition of the *merge* function that merges r_{src} into r_{dst}.

6.2.3.2. Merging of Regions

In the following, we use the function $merge_{r_{dst},\, r_{src}}$ defined in Fig. 6.7 that merges two regions. For each field f contained in r_{src}, the function distinguishes two cases – if f is also contained in r_{dst}, the value of f in r_{dst} is updated such that it is an upper bound of its current value and the value of f in r_{src} (shown in Eqn. 6.64f). If, on the other hand, f is not contained in r_{dst}, a new field is added to r_{dst} and its value is copied from r_{src} (shown in Eqns. 6.66ff). This way, when merging several regions $r^1_{src}, r^2_{src}, \ldots$ into a region r_{dst}, each field in r_{dst} contains an upper bound of all existing fields in all source regions.

6.2.3.3. Handling of Callee Aliasing

A caller of a function may pass the same pointer to a callee multiple times, i.e. the summary might have been computed using the incorrect assumption that no input pointers alias. In order to deal with this case, we call *handleCalleeAliasing* shown in Fig. 6.8 for each non-singleton set of callee regions that is associated with a single caller region. If two aliasing callee regions contain at least one common field, the summary has been built using a wrong aliasing assumption. As a consequence, it is unsound to apply the summary to the current call site: it has to be rebuild with an additional aliasing constraint. Currently, our analyzer issues a warning and ignores the summary, i.e. continues as if no targets for the call site were known. The analysis is sound if no warnings are emitted. This behavior is implemented in *handleCalleeAliasing* in

$$handleCalleeAliasing : \wp(\mathcal{R}) \times MS^2 \times D \to \mathcal{R} \times MS^2 \times D$$

$$handleCalleeAliasing(rs_{callee},\ m^2_{in},\ m^2_{out},\ d_2) =$$

$$\text{for } \{r_2,\ r'_2\} \subseteq rs_{callee} \text{ do} \tag{6.70}$$

$$\text{let } \langle \phi^\sharp_2,\ \phi^\sharp_{2'} \rangle = \langle m^2_{in}(r_2),\ m^2_{in}(r'_2) \rangle \tag{6.71}$$

$$\text{if } \exists f \in \text{dom}(\phi^\sharp_2).f \in \text{dom}(\phi^\sharp_{2'}) \text{ then} \tag{6.72}$$

$$\text{issue warning and abort summary application} \tag{6.73}$$

$$\langle m^2_{in},\ m^2_{out},\ d_2 \rangle := addRegion_{r_{combined}}(\langle m^2_{in},\ m^2_{out},\ d_2 \rangle) \tag{6.74}$$

$$\text{for } r_{callee} \in rs_{callee} \text{ do} \tag{6.75}$$

$$\langle m^2_{in},\ m^2_{out},\ d_2 \rangle := merge_{r_{combined},\ r_{callee}}(\langle m^2_{in},\ m^2_{out},\ d_2 \rangle) \tag{6.76}$$

$$\text{return } \langle r_{combined},\ m^2_{in},\ m^2_{out},\ d_2 \rangle \tag{6.77}$$

Figure 6.8.: Definition of the *handleCalleeAliasing* function which combines aliasing regions in a summary if possible and aborts the summary application otherwise.

Eqns. 6.70 through 6.73. The following example demonstrates that a summary becomes invalid in the presence of callee aliasing.

Example 5. *Consider the code in Fig. 6.13. Here, the caller passes a reference to a single* `S` *object to the callee as first and as second parameter. As a result, these two parameters alias. The summary is built on the assumption that no input pointers alias. However, the function* `g` *also accesses the same field* `i` *through the two pointers. On first sight, it might seem reasonable to merge the two aliasing regions, joining the respective variables of the fields. However, this does not result in a correct over-approximation and it is, thus, unsound to use the summary for the given call site as shown in the following. In the summary of function* `g`*, say* $\langle [\ldots],\ [\texttt{*s1} \mapsto [i \mapsto x_1^{out}],\ \texttt{*s2} \mapsto [i \mapsto x_2^{out}],\ \ldots],\ \{x_1^{out} = \{4\},\ x_2^{out} = \{4\},\ \ldots\}\rangle$*, the two different fields representing the same callee field* `i` *both contain the value* 4 *as can be seen by following the assignments carried out in* `g`*. Thus, joining the respective domain variables* x_1^{out} *and* x_2^{out} *yields the singleton set* $\{4\}$*. In fact, however, the field* `i` *contains the value* 2 *after the call to* `g`*. It is thus obvious that it is unsound to apply this summary to the caller state.*

Yet, if no regions contain common fields, we implement one special case where we allow callee aliasing by using the function $merge_{r_{dst},\ r_{src}}$ to merge aliasing regions; here, this amounts to copying fields which only exist in either r_{src} or r_{dst} (see Eqns. 6.74ff). One might consider the extra effort for handling this case overly complex; however, region aliasing with no common fields is not as much of a special case as it might

$$handleCallerAliasing : \wp(\mathcal{R}) \times MS^2 \times D \rightarrow \mathcal{R} \times MS^2 \times D$$
$$handleCallerAliasing(rs_{caller},\ m^1_{in},\ m^1_{out},\ d_1) =$$
$$\langle m^1_{in},\ m^1_{out},\ d_1\rangle := addRegion_{r_{combined}}(\langle m^1_{in},\ m^1_{out},\ d_1\rangle) \tag{6.78}$$
$$\text{for } r_{caller} \in rs_{caller} \text{ do} \tag{6.79}$$
$$\langle m^1_{in},\ m^1_{out},\ d_1\rangle := merge_{r_{combined},\ r_{caller}}(\langle m^1_{in},\ m^1_{out},\ d_1\rangle) \tag{6.80}$$
$$\text{return } \langle r_{combined},\ m^1_{in},\ m^1_{out},\ d_1\rangle \tag{6.81}$$

Figure 6.9.: Definition of *handleCallerAliasing* which combines multiple caller regions.

seem. This is because it is common to pass references to stack variables to functions. In our implementation that works with field offsets and sizes instead of names, all stack variables reside in the same region, i.e. the region the stack pointer dereferences to. As a result, it is indeed important to precisely handle aliasing callee regions with no overlapping fields.

6.2.3.4. Handling of Caller Aliasing

As pointed out above, there is another kind of aliasing which we call *caller aliasing*. Caller aliasing results from passing a pointer parameter to a callee that has a non-singleton alias set in the caller. In order to deal with caller aliasing, we call the function *handleCallerAliasing* defined in Fig. 6.9 for each non-singleton set of caller regions that is associated with a single callee region. Again, we use the function $merge_{r_{dst},\ r_{src}}$ in Eqn. 6.80. As explained above, $merge_{r_{dst},\ r_{src}}$ copies fields that only exist in either r_{dst} or r_{src}. Note, however, that this time it is possible to deal with the case of finding a field in multiple regions; see Sect. 6.2.3.2 for details.

The *expandCallerRegions* function shown in Fig. 6.10 is used to propagate the result of applying a function summary to combined regions back to their original regions. For this, the function iterates over the associations in *foldMap* (Eqn. 6.82). For each association, we expand the callee region (Eqn. 6.83) yielding a new region r_{new}. In general, expanding a region into another region consists of a field-wise assignment while stripping relational information between the newly created fields. (This is a simplification of a generic *expand* function [53].) We then weakly assign each field in r_{new} to the corresponding field in the original region r_{orig} by computing the least upper bound of the value domain state after the assignment and the current value domain state d as shown in Eqn. 6.86. The weak assignment is necessary because we update multiple original abstract memory regions while a concrete execution of the

$$expandCallerRegions : \mathcal{R} \times \wp(\mathcal{R}) \times MS^2 \times D \rightarrow MS^2 \times D$$
$$expandCallerRegions(foldMap,\ m^1_{in},\ m^2_{out},\ d) =$$
$$\text{for } r_{orig} \in rs_{orig} \text{ where } \langle r_{combined},\ rs_{orig}\rangle \in foldMap \text{ do} \tag{6.82}$$
$$\langle m^1_{in},\ m^2_{out},\ d\rangle := expand_{r_{combined},\, r_{new}}(\langle m^1_{in},\ m^2_{out},\ d\rangle) \tag{6.83}$$
$$\text{let } \langle \phi^\sharp_{orig},\ \phi^\sharp_{new}\rangle = \langle m^2_{out}(r_{orig}),\ m^2_{out}(r_{new})\rangle \tag{6.84}$$
$$\text{for } f \in \text{dom}(\phi^\sharp_{orig}) \text{ do} \tag{6.85}$$
$$d := d \sqcup_D ([\![\phi^\sharp_{orig}(f) := \phi^\sharp_{new}(f)]\!]^\sharp\ d) \tag{6.86}$$
$$\text{for } f \in \text{dom}(m^2_{out}(r_{combined})) \setminus \text{dom}(\phi^\sharp_{orig}) \text{ do} \tag{6.87}$$
$$\langle m^1_{in},\ m^2_{out},\ d\rangle := addField_{r_{orig},\, f}(\langle m^1_{in},\ m^2_{out},\ d\rangle) \tag{6.88}$$
$$d := [\![m^2_{out}(r_{orig})(f) := \top_D]\!]\ d \tag{6.89}$$
$$\text{return } \langle m^1_{in},\ m^2_{out},\ d\rangle \tag{6.90}$$

Figure 6.10.: Definition of *expandCallerRegions* which expands folded caller regions and propagates their contents back to the original regions.

function only updates one specific region, depending on the actual pointer value. In Eqns. 6.87ff, we set all fields to $\top_D$ which only exist in a combined region, but not in the corresponding original region. Finally, *expandCallerRegions* returns the updated state.

It is not trivial to understand why our combination of using the functions $merge_{r_{dst},\, r_{src}}$ and *expandCallerRegions* is sound. For this, we have to consider two cases:

1. **Missing fields**: Consider a region r_{orig} and a field f which exists in the combined region $r_{combined}$ but not in r_{orig} to which the contents of $r_{combined}$ are propagated. Such a field is set to $\top_D$. Thus, the value of f after the summary application is an over-approximation of the actual contents of f.

2. **Existing field**: Consider a region r_{orig} and a field f which exists in the combined region $r_{combined}$ and also in r_{orig} to which the contents of $r_{combined}$ are propagated. In this case, the existing value of f has been taken into account by $merge_{r_{dst},\, r_{src}}$ when computing the upper bound for the combined region to which the effect of the function is applied. Because of this and the fact that we do a weak update, the resulting value of f is again a sound over-approximation of the concrete value of f after the function call.

$$
\begin{aligned}
&\mathit{addOrRenameFields} : MS^2 \times D \times MS^2 \times D \to MS^2 \times D \\
&\mathit{addOrRenameFields}(m^1_{in},\ m^1_{out},\ d_1, m^2_{in},\ m^2_{out},\ d_2) = \\
&\quad \text{for } r \in \mathrm{dom}(m^1_{in}) \text{ do} && (6.91) \\
&\quad\quad \text{if } r \notin \mathrm{dom}(m^2_{in}) && (6.92) \\
&\quad\quad\quad \langle m^2_{in},\ m^2_{out},\ d_2\rangle := \mathit{addRegion}_r(\langle m^2_{in},\ m^2_{out},\ d_2\rangle) && (6.93) \\
&\quad\quad \text{for } f \in \mathrm{dom}(m^1_{in}(r)) \wedge f \notin \mathrm{dom}(m^2_{in}(r)) \text{ do} && (6.94) \\
&\quad\quad\quad \langle m^2_{in},\ m^2_{out},\ d_2\rangle := \mathit{addField}_{r,\,f}(\langle m^2_{in},\ m^2_{out},\ d_2\rangle) && (6.95) \\
&\quad\quad \text{for } f \mapsto x^1_{out} \in m^1_{out}(r),\ f \mapsto x^2_{in} \in m^2_{in}(r) \text{ do} && (6.96) \\
&\quad\quad\quad \langle m^2_{in},\ m^2_{out},\ d_2\rangle := \mathit{renameField}^1_{r,\,f,\,x^2_{in},\,x^1_{out}}(\langle m^2_{in},\ m^2_{out},\ d_2\rangle) && (6.97) \\
&\quad \text{return } \langle m^2_{in},\ m^2_{out},\ d_2\rangle && (6.98)
\end{aligned}
$$

Figure 6.11.: Definition of the *addOrRenameFields* function which makes sure that each region and field of the caller output memory structure (first three arguments) exists in the summary input memory structure (second three arguments) and that the respective value domain variables have the same name.

Example 6. *Consider the code in Fig. 6.12. Here, the caller* `f` *first initializes two* `S` *objects before it randomly passes one to the callee* `g`*. Thus, dereferencing the passed-in pointer leads to two possible memory regions in the caller. In the callee, only the memory pointed to by a single variable, the parameter, is changed. Thus, there is one callee region related to multiple caller regions in regm. Assume the caller state at the call site to be* $\langle[...],\ [\texttt{s1} \mapsto [i \mapsto x^{out}_1],\ \texttt{s2} \mapsto [i \mapsto x^{out}_2],\ \ldots],\ \{x^{out}_1 = 0,\ x^{out}_2 = 1,\ \ldots\}\rangle$*. Because the two memory regions* `s1` *and* `s2` *are associated with a single callee region, handleCallerAliasing merges them into a new summary region* $r_{new} = [i \mapsto x]$ *with* $d_1(x) = \{0,\ 1\}$*. Given the summary* $\langle[\texttt{*s} \mapsto [i \mapsto x^{in}],\ \ldots],\ [\texttt{*s} \mapsto [i \mapsto x^{out}],\ \ldots],\ \{x^{out} = x^{in} + 1,\ \ldots\}\rangle$*, Eqn. 6.43 computes* d' *with* $d'(x^{out}) = \{1,\ 2\}$*. The call to expandCallerRegions in Eqn. 6.44 propagates* x^{out} *back to* x^{out}_1 *and* x^{out}_2 *by setting* x^{out}_1 *to* $d'(x^{out}_1) \sqcup d'(x^{out}) = \{0\} \sqcup \{1,\ 2\} = \{0,\ 1,\ 2\}$ *and* x^{out}_2 *to* $d'(x^{out}_2) \sqcup d'(x^{out}) = \{1\} \sqcup \{1,\ 2\} = \{1,\ 2\}$*.*

6.2.3.5. Aligning the Summary Input to the Caller Output

The *addOrRenameFields* function defined in Fig. 6.11 makes sure that each region and field of the output part of the caller memory structure exists in the callee input memory structure and that the respective value domain variables have the same name. Note

```
struct S {
  int i;
};

void f() {
  S s1 = { 0 };
  S s2 = { 1 };
  g(rnd() ? &s1 : &s2);
}

void g(S *s) {
  s->i += 1;
}
```

Figure 6.12.: Example code that results in one callee region that maps to multiple caller regions. The summary effect has to be applied to both caller regions.

that at the point at which the function is called, summary regions have already been renamed to match caller regions according the caller/callee relation *regm*. Thus, when iterating over the caller regions as shown in Eqn. 6.91, the regions can be used to index into the memory structure of the function summary. If the caller region is not found in the summary, a fresh region is added to the summary (Eqns. 6.92f). Next, we add missing caller fields to the function summary (Eqns. 6.94f). Finally, we rename the value domain variables in the summary fields such that they have the same name as in the caller output (Eqns. 6.96f).

6.2.4. Computing a Fixpoint of the Abstract Relational Semantics

This section details how the modular abstract semantics is used to compute a fixpoint of the whole program. A whole-program analysis populates a table $T \in \mathbb{T} = Loc_F \to MS^2 \times D$ that takes function addresses to their summaries. Since a function f may call other functions, a call statement in f will access T to obtain the most up-to-date summary for the called function. The semantics of the `call` statement is therefore parameterized by T:

$$[\![l_s\colon call\ e]\!]^\sharp_T \langle ms_1, ms_2, d\rangle = \{\langle next(l), \bigsqcup_{l_f \in \gamma_D(d)([\![e]\!]^\sharp)} \langle ms_1, ms_2, d\rangle \bowtie_{MS^2 \times D} T(l_f)\rangle\} \tag{6.99}$$

```
1  struct S {
2    int i;
3  };
4
5  void f() {
6    S s = { 0 };
7    g(&s, &s);
8  }
9
10 void g(S *s1, S *s2) {
11   s1->i = 4;
12   s2->i = 2;
13   s2->i = s1->i;
14 }
```

Figure 6.13.: Example code that results in two callee regions that relate to one caller region.

The resulting summary for f must therefore be re-computed if any summaries taken from T change. In the presence of recursive calls, widening [13] must be applied on the summaries to ensure termination.

The summary of f, given the table T and initial state s (which may or may not be specialized as described in the following sections), is defined as follows:

Definition 4. *The abstract state of f is a map $abs^T_{f,s} : Locs_S \to MS^2 \times D$ with $s \sqsubseteq_{MS^2 \times D} abs^T_{f,s}(l_f)$ and for all $l : stmt \in P$ and $\langle l', s' \rangle \in [\![l : stmt]\!]^\sharp_T(abs^T_{f,s}(l))$ it holds that $s' \sqsubseteq_{MS^2 \times D} abs^T_{f,s}(l')$. (Note: The abstract semantics in Fig. 6.3 has been written as $[\![l : stmt]\!]^\sharp$ instead of $[\![l : stmt]\!]^\sharp_T$ for simplicity; the table of function summaries T has to be added as parameter.)*

Let $init = \langle m_1, m_2, \{x_1 = x_2 = l^{res}_f\}\rangle$ with $m_i = [f \mapsto [ret \mapsto x_i]],\ i = 1, 2$ be the initial summary state. The summary semantics of f relates the first statement of the function at l_f with the location l^{res}_f that the return statement branches to:

Definition 5. *The abstract summary of f under T is $sum^T_f = abs^T_{f,init}(l^{res}_f)$.*

This concludes the presentation of the concrete relational semantics and the abstract summary domain and semantics. The next section tackles the challenge of computing precise summaries in the presence of indirect function calls.

6.3. On-Demand Heyting Completion

This section details how we use Herbrand terms to refine a function summary in cases where the most generic input would lead to an unacceptable precision loss. In particular, the next sections discuss the creation of Herbrand terms to express a need for refinement, the computation of a specialized function summary and the *call* semantics that combines specialized function summaries.

6.3.1. Extracting Refinement Information using Herbrand Terms

The challenge in specializing the summary of `Check` in Fig. 6.1 is that the variable over which to specialize is not known until the indirect call `parity->IsEven()` is analyzed. Our solution is that the analysis poses the question "What value can `parity->vtable[0]` take on?" to all callers of `Check` who may answer "The expression `parity->vtable[0]` may contain `&Odd::IsEven()`". (Recall that we use `vtable[0]` as a field name to fit our restricted grammar.) For each different answer, the summary of the analysis is specialized to the value in that answer. The analysis of `Check` can now proceed to the next indirect call `parity->IsOdd()` for which a new question is posed to the caller. Once the indirect function calls are resolved, `Check` can be summarized without posing further questions.

The "question" in the exposition above is represented by a Herbrand term that contains variables in places where the answer is expected. The answer to the question is given by a set of ground Herbrand terms, that is, Herbrand terms where the variables have been replaced by values.

Definition 6. *Herbrand terms Herb* $= \mathcal{L}(\mathit{Term})$ *are defined by the grammar*

$$\begin{array}{rcl} \mathit{Term} & ::= & \texttt{constructor}\ \mathit{Term}^* \\ & | & \texttt{variable} \end{array}$$

where `variable` *is drawn from* $\mathcal{X}_H$. *Note that* $\mathcal{X}_H$ *is distinct from* $\mathcal{X}$. *Let* $\mathit{vars}(h)$ *denote all variables in* $h \in \mathit{Herb}$. *Let* $\mathit{GHerb} = \{h \in \mathit{Herb} \mid \mathit{vars}(h) = \emptyset\}$ *denote ground Herbrand terms. A substitution* $\theta \in \Theta : \mathcal{X}_H \to \mathit{Herb}$ *is a total map with* $\theta(x) = x$ *except for a finite number of variables* $y \in \mathcal{X}_H$ *where* $\theta(y) \neq y$. *We write* $[x/y] \in \Theta$ *with* $[x/y](x) = y$ *and* $[x/y](v) = v$ *for all* $v \neq x$. *Given a term* $h \in \mathit{Herb}$, *we write* $\theta(h)$ *to denote the result of replacing all variables* x *in* h *by* $\theta(x)$. *Let* $\theta(H) = \{\theta(h) \mid h \in H\}$ *be the lifting to sets.*

The generic nature of Herbrand terms enables us to formulate questions that cut across several abstract domains in an abstract state $\langle m_1,\ m_2,\ d\rangle \in \mathit{MS}^2 \times D$.

Example 7. *Suppose that the constructors Deref and Field are used by the memory domain $m \in MS$ to denote a pointer or field access, respectively, while ConstPtr is used by the numeric domain $d \in D$ to denote a function pointer. Then the term ConstPtr (Field (Deref* `parity`*)* `vtable[0]`*) a_E is the request to access the field* `vtable[0]` *of the memory region pointed-to by* `parity` *and to extract the value as a constant pointer, denoting the result by $a_E \in \mathcal{X}_H$. This query accesses $m($*`f:parity`*$) = [$*`parity`* $\mapsto x, ...]$ where f is the currently analyzed function in order to obtain the numeric variable $x \in \mathcal{X}$ that contains the points-to set of* `parity`*. The numeric domain d is queried for the points-to set of x which resolves to, say, the address of memory region* `even` *$\in \mathcal{R}$. Finally, the memory domain is used to look up $m($*`even`*$) = [$*`vtable[0]`* $\mapsto vt_E, \ldots]$ and d is queried for the values of vt_E, the constant address* `vtable[0]` *of* `Even`*, which becomes the solution of a_E.*

For the sake of readability, we leave the exact definition of the term structure open and write `var->field` ... `->field` $= a_E$, that is, we use C-like access paths that generalize $\mathcal{L}(Expr)$ by allowing several indirections. Moreover, we also omit the memory region (i.e. we write `this->vtable[0]` instead of `f:this.this->vtable[0]`) since a Herbrand term is always relative to the stack frame of the current function.

Herbrand terms are used in the abstract semantics when a precise value is needed. For instance, the *call e* instruction requires a precise value for the function address e that determines which function is being invoked. An answer is computed using a function *herbEval* that evaluates a term set (e.g. $\{``e = x''\} \subseteq Herb$ for the call) given an abstract state. *herbEval* has the following signature:

$$herbEval : \wp(Herb) \times MS^2 \times D \rightarrow \wp(\Theta) \times \wp(Herb)$$

For variables $a_1, \ldots a_n$ in the input Herbrand terms, *herbEval* returns assignments in form of substitutions $\theta_1, \ldots \theta_k$ where each $\theta_j = [a_1/c_1^j, \ldots a_n/c_n^j]$ maps variables to constants $c_1^j, \ldots c_n^j \in \mathbb{V}$, $j = 1, \ldots k$, or it rewrites the Herbrand terms into terms over the function's input arguments and globals. In order to illustrate this, we say that a Herbrand term h_i matches a domain variable x_i if h_i represents a field access (possibly via one or more pointer indirections) whose value is given by the domain variable x_i. We give an intuitive overview of *herbEval* by describing the four cases it distinguishes:

A set of values for tabulation can be constructed. The term h_i with variable a_i matches a domain variable x_i, $i = 1, \ldots n$. In case x_i are finite in the value domain state d, *herbEval* returns a set of constant value vectors $\vec{c}^1, \ldots \vec{c}^k \in \{\langle \vec{v}(x_1), \ldots \vec{v}(x_n)\rangle \mid \vec{v} \in \gamma_{\mathcal{D}_X}(d)\}$ in the form of the k substitutions $\theta_j = [a_1/\vec{c}^j(x_1), \ldots a_n/\vec{c}^j(x_n)] \in \Theta$. For example, $herbEval(\{\mathtt{m}.f = a\}, \langle [\mathtt{m} \mapsto [f \mapsto x_1]], [\mathtt{m} \mapsto [f \mapsto x_2]], d\rangle)$ evaluates to $\langle \{[a/42]\}, \emptyset\rangle$ where $d = \{x_2 = 42\}$ represents the value domain.

```
1 bool Case1() {
2   Odd odd;
3   Even even;
4   Parity* parity =
5     rnd() ? &odd : &even;
6   return Check(parity);
7 }
```

```
1 void Case2(Parity *p) {
2   Check(p);
3 }
4
5 void Case3(Parity *p, Parity *q) {
6   Check(rnd() ? p : q);
7 }
```

Figure 6.14.: Creating Herbrand terms for calls to `Check` in Fig. 6.1.

An exact precondition can be synthesized. A term h_i matches a variable x_i. There exists $x'_i = x_i$ where x'_i is a domain variable of a field in the input memory region. For each x'_i, we return a Herbrand term h'_i that matches x'_i. For example, $herbEval(\{\texttt{m}.f = a\}, \langle[\texttt{m} \mapsto [f \mapsto x_1], \texttt{r} \mapsto [g \mapsto x_2]], [\texttt{m} \mapsto [f \mapsto x_3], \texttt{r} \mapsto [g \mapsto x_4]], d) = \langle\emptyset, \{\texttt{r}.g = a\}\rangle$ if $d = \{x_2 = x_3\}$ is the value domain.

A sufficient precondition can be synthesized. The term h_i matches a variable x_i. There exist several variables $\{x_i^1, \ldots x_i^{k_i}\}$ from which there is a flow of information to x_i. We translate the single term h_i to Herbrand terms $h_i^1, \ldots h_i^{k_i}$ that match $x_i^1, \ldots x_i^{k_i}$ and return the term $\texttt{Set}\ h_i^1 \ldots h_i^{k_i}$. For example, $herbEval(\{\texttt{t}.q = a\}, [\texttt{u} \mapsto [r \mapsto x_1], \texttt{v} \mapsto [s \mapsto x_2]], [\ldots, \texttt{t} \mapsto [q \mapsto x_3]], d) = \langle\emptyset, \{\texttt{Set u}.r = a_1\ \texttt{v}.s = a_2\}\rangle$ where $d = [x_1 \mapsto \{\&p_1\}, x_2 \mapsto \{\&p_2\}, x_3 \mapsto \{\&p_1, \&p_2\}]$ represents the information of our aliasing domain $D_X = X \to \wp(Loc \cup \{a_{bad}\})$ used in Ex. 1. We will disregard this case until our discussion in Sect. 6.4.

No values can be synthesized. The term h_i matches no variable x_i nor can a field variable be added using $addField_{r,f}$. Thus, the values of variables in h_i are neither finite nor traceable to the input. An empty set of substitutions and Herbrand terms is returned. A warning is generated so that the analysis is sound if no warnings are emitted.

Example 8. *We illustrate cases 1 to 3 using the functions in Fig. 6.14. We assume that* `Check` *has been analyzed with no specialization such that the first indirect call cannot be resolved. The resulting summary state is* $\langle\bot_{MS^2 \times D}, H\rangle$ *where* $H = \{\texttt{parity->vtable[0]} = a\}$. *As a consequence, H is evaluated at each call site using herbEval.*

Consider the code of `Case1` *in Fig. 6.14. When reaching the call to* `Check` *with summary state* $s \in MS^2 \times D$, *we evaluate* $herbEval(H, s)$ *which amounts to evaluating the value of* `parity->vtable[0]` *in s. In this case, the state at the call site contains a finite set of values for this field, namely* $\vec{v}_1 = \langle\texttt{\&Odd::IsEven}\rangle$ *and* $\vec{v}_2 = \langle\texttt{\&Even::IsEven}\rangle$. *Thus,*

two new table entries have to be generated for `Check`*, one for* $H_1 = \{$`parity->vtable[0]` $=$ `&Odd::IsEven`$\}$ *and* $H_2 = \{$`parity->vtable[0]` $=$ `&Even::IsEven`$\}$*. No further queries are raised. In* `Case2`*, the state at the call site of* `Check` *does not contain a finite set of values for the queried fields. However, there exists an equality relation with the parameter* `p`*. Thus, herbEval rewrites H to* $H' = \{$`p->vtable[0]` $= a\}$ *in terms of the parameter and propagates it to the callers of* `Case2`*. Finally, in* `Case3`*, herbEval is able to use the flow information computed by the points-to domain to determine that the l-values in* `parity->vtable[0]` *is a superset of the values in* `p->vtable[0]` *and* `q->vtable[0]`*. Thus, herbEval returns a single Herbrand term Set* h_p h_q *where* $h_i \equiv \{i$`->vtable[0]` $= a_i\}$.

We omit a formal definition of *herbEval* as it is parametric in the value domain it operates on: In this case, *herbEval* extracts finite value sets and equalities between variables from the value domain, but other information can be exploited as well. The next section discusses how *herbEval* is used to compute specialized summaries.

6.3.2. Specializing Summaries with Herbrand Terms

This section illustrates how a function summary is computed that is specialized wrt. a set of ground terms $H_g \in GHerb$. To this end, we first define the lattice of an abstract domain where transformers can generate Herbrand terms whenever the function context needs to be refined. The lattice of this analysis is a product of $MS^2 \times D$ and a set of Herbrand terms *Herb* that we write as $\langle MS^2 \times D \times \wp(Herb),\ \sqsubseteq_H,\ \sqcup_H,\ \bot_H \rangle$. All lattice operations are the point-wise liftings, i.e. $\langle s_1,\ H_1\rangle \sqsubseteq_H \langle s_2,\ H_2\rangle \equiv s_1 \sqsubseteq_{MS^2 \times D} s_2 \wedge H_1 \subseteq H_2$, etc. In particular, note that the product is not reduced [43], so that $\langle \bot_{MS^2 \times D},\ H\rangle \neq \bot_H$ unless $H = \emptyset$.

The analysis populates a table in $\mathbb{T}_{Herb} = Loc_F \times GHerb \rightarrow MS^2 \times D \times \wp(Herb)$. Each entry $\langle f,\ H_g\rangle \mapsto \langle s,\ H\rangle$ states that f, when specialized by H_g, has the summary s and requires further specializations by instantiating H in its callers. We define the following transformer to impose H_g on an abstract state:

$$[\![test\ H_g]\!]^\sharp \quad : \quad (MS^2 \times D) \rightarrow MS^2 \times D \tag{6.100}$$

For example, given the terms $H_g = \{$`var.field` $= 42\}$, the initial state *init* in Sect. 6.2.4 is refined to $[\![test\ H_g]\!]^\sharp init = \langle [$`f` $\mapsto [ret \mapsto x_1],$ `var` $\mapsto [field \mapsto x_3]],\ [$`f` $\mapsto [ret \mapsto x_2],$ `var` $\mapsto []],\ \{x_1 = x_2 = l_f^{res},\ x_3 = 42\}\rangle$. This refines the input of the summary state and may add empty regions to the output, however no output fields or variables are added. As soon as the output field `var.field` is accessed, the $addField^2_{\mathtt{var},field}$ morphism shown in Eqn. 6.15 is applied which results in the summary state $\langle m_1,\ m_2,\ \{x_1 = x_2 = l_f^{res},\ x_3 = x_4 = 42\}\rangle$ where $m_i = [$`f` $\mapsto [ret \mapsto x_i],$ `var` $\mapsto [field \mapsto x_{i+2}]]$ for

$i = 1, 2$. The semantics of a function f for a specialization H_g is defined by $sum_f^{T_H}$ that generalizes Def. 5. It uses $abs_{f,s}^{T_H}$ which generalizes $abs_{f,s}^{T}$ from Def. 4:

Definition 7. *The specialized abstract summary of f under $T_H \in \mathbb{T}_{Herb}$ is given by $sum_f^{T_H}$: $GHerb \to (MS^2 \times D) \times \wp(Herb)$ where $sum_f^{T_H}(H_g) = abs^{T_H}_{f, [\![test\ H_g]\!]^\sharp init}$.*

Here, $T_H \in \mathbb{T}_{Herb}$ is the table of specialized summaries. Its elements are defined in terms of $sum_f^{T_H}$:

Definition 8. *$T_H \in \mathbb{T}_{Herb}$ is a well-formed table if $T_H(\langle f,\ H_g\rangle) = sum_f^{T_H}(H_g)$ for all $\langle f,\ H_g\rangle \in \mathrm{dom}(T_H)$.*

The analysis bootstraps by computing a summary for each function f with no specialization, thus providing the table entries with key $\langle f,\ \emptyset\rangle$. For any specialization H_g, a result $\langle s,\ H\rangle \in T_H(\langle f,\ H_g\rangle)$ may contain a non-empty set $H \in Herb$ that states how the function input must be specialized further so that the summary is an over-approximation of the function's concrete semantics. We now define how a call site of f instantiates H to a set of ground Herbrand terms $H_g \in GHerb$ that can be used to compute a specialized function summary $\langle f,\ H_g\rangle$ in T_H.

6.3.3. Combining Specialized Function Summaries

We now explain the differences between the semantics of the *call*-statement in Eqn. 6.99 and the following definition over the $(MS^2 \times D) \times \wp(Herb)$ domain:

$$\begin{aligned}
[\![l_s\!:\ call\ e]\!]^\sharp_{T_H} &: (MS^2 \times D) \times \wp(Herb) \to \\
&\quad \wp(Loc[S] \times (MS^2 \times D) \times \wp(Herb)) \qquad (6.101)\\
[\![l_s\!:\ call\ e]\!]^\sharp_{T_H}\ \langle s,\ H\rangle &= \{\langle next(l_s), \\
&\quad \langle \bot_{MS^2\times D},\ H \cup H_f\rangle \sqcup_{MS^2\times D} \bigsqcup_{l_f \in \{l_f^1, \ldots\, l_f^n\}} applyEntries^{T_H}_{l_f}(s,\ \emptyset,\ \emptyset)\rangle\}\\
&\quad \langle\{[a/l_f^1],\ \ldots\ [a/l_f^n]\},\ H_f\rangle = herbEval(\{\text{"}e = a\text{"}\},\ s) \qquad (6.102)
\end{aligned}$$

Rather than using the concretization function $\gamma_{MS^2\times D}$ to obtain the callee addresses l_f, we evaluate a Herbrand term $e = a$ in the current state $s \in MS^2 \times D$ where e is the called expression. We obtain a set of function addresses l_f^i, $i \in [1,\ n]$ and/or Herbrand terms H_f. Recall that a non-empty H_f contains predicates over the inputs of this function that need to be restricted to a finite set of callers before this call has an effect. Thus, the predicates $H \cup H_f$ are returned with a bottom summary $\bot_{MS^2\times D}$. The

$$\begin{aligned}
&\mathit{applyEntries}_f^{T_H} : ((MS^2 \times D) \times \wp(\mathit{Herb}) \times \wp(\mathit{GHerb})) \to (MS^2 \times D) \times \wp(\mathit{Herb})\\
&\mathit{applyEntries}_f^{T_H}(s,\ H,\ H_g) =\\
&\quad \text{let } \langle s',\ H' \rangle \in T_H(\langle f,\ H_g \rangle) && (6.103)\\
&\quad \text{if } H' = \emptyset \text{ then return } \langle s \bowtie_{MS^2 \times D} s',\ \emptyset \rangle && (6.104)\\
&\quad \text{let } \langle \Theta,\ H^{new} \rangle = \mathit{herbEval}(H \cup H',\ s) && (6.105)\\
&\quad \text{let } \overline{H'_g} = \{ H'_g \mid H'_g = \theta(H \cup H') \cap \mathit{GHerb},\ \theta \in \Theta,\ H_g \subseteq H'_g \} && (6.106)\\
&\quad \text{return } \langle \bot_{MS^2 \times D},\ H^{new} \rangle \sqcup_{MS^2 \times D} \bigsqcup_{H'_g \in \overline{H'_g}} \mathit{applyEntries}_f^{T_H}(s,\ H \cup H',\ H'_g) && (6.107)
\end{aligned}$$

Figure 6.15.: Applying a specialized function summary in $T_H \in \mathbb{T}_{Herb}$

effect of each known callee at l_f^i is composed with the current state s using a helper function *applyEntries* that is defined in Fig. 6.15.

The idea of *applyEntries* is to find those specializations of callee f that match the caller state s and to combine those specializations with s. The arguments H and H_g always contain the same number of terms, where H_g is one specialization of H in s. In Eqn 6.103, we assume the table T_H contains an entry for the specialization $\langle f,\ H_g \rangle$. It is up to the fixpoint engine to compute a missing entry on-the-fly or to resume the evaluation of the caller once the entry is available. If the retrieved summary s' requires no new specializations, i.e. if $H' = \emptyset$, the summary s' is composed with the caller state in Eqn. 6.104 and returned. In case $H' \neq \emptyset$, the summary s' is an under-approximation and a more specialized summary must be consulted by instantiating $H \cup H'$ in the caller state as done in Eqn. 6.105. The evaluation has two outcomes (which are not necessarily mutually exclusive): if $H^{new} \neq \emptyset$ then *herbEval* was able to translate the terms H' of the callee to inputs of the caller. These terms are therefore returned with the bottom summary $\bot_{MS^2 \times D}$ so that the caller will be refined. The second case is that $H \cup H'$ could be instantiated to concrete values in form of a set of substitutions Θ. Equation 6.106 applies Θ to obtain sets of ground terms $\overline{H'_g} \in \wp(\wp(\mathit{GHerb}))$ of which only those are returned that match the current specialization H_g. Each set $H'_g \in \overline{H'_g}$ is used to look up a more specialized summary of f by calling *applyEntries* recursively. We illustrate these definitions with an example.

Example 9. *We illustrate the call semantics using the call to* `Check` *in* `Case1` *in Fig. 6.14. Assume that* T_H *has the following entries (*`vt` *is short for* `parity->vtable`*):*

1	$\langle$`&Check`, $\emptyset\rangle$	$\langle\bot_{MS^2\times D}$, {`vt[0]` $= a_0$}$\rangle$
2	$\langle$`&Check`, {`vt[0]=&Even::IsEven`}$\rangle$	$\langle s_1$, {`vt[1]` $= a_1$}$\rangle$
3	$\langle$`&Check`, {`vt[0]=&Odd::IsEven`}$\rangle$	$\langle s_2$, {`vt[1]` $= a_2$}$\rangle$
4	$\langle$`&Check`, {`vt[0] = &Even::IsEven`, `vt[1] = &Even::IsOdd`}$\rangle$	$\langle s_3, \emptyset\rangle$
5	$\langle$`&Check`, {`vt[0] = &Odd::IsEven`, `vt[1] = &Odd::IsOdd`}$\rangle$	$\langle s_4, \emptyset\rangle$

The abstract call semantics in Eqn. 6.102 invokes applyEntries$^{T_H}_{\texttt{\&Check}}(s, \emptyset, \emptyset)$ *where s is the caller state at the call site. The fact that Eqn. 6.103 returns a non-empty* $H' = \{\texttt{vt[0]} = a_0\}$ *means that a specialization needs to be computed, based on s which is done by Eqn. 6.105. Since s provides a finite set of values for* a_0, $\Theta = \{[a_0/\texttt{\&Even::IsEven}], [a_0/\texttt{\&Odd::IsEven}]\}$ *while* H^{new} *is empty. Applying these substitutions in Eqn. 6.106 gives two specializations in* $\overline{H'_g}$*, leading to two recursive calls in Eqn. 6.107, namely applyEntries*$(s, \{\texttt{vt[0]} = a_0\}, \{\texttt{vt[0]} = \texttt{\&Even::IsEven}\})$ *and applyEntries*$(s, \{\texttt{vt[0]} = a_0\}, \{\texttt{vt[0]} = \texttt{\&Odd::IsEven}\})$*. We only consider the first call as the second is analogous. Equation 6.103 extracts the 2nd table entry which, yet again, returns a non-empty* H'*. Equation 6.105 computes* $\Theta = \{\theta_1, \theta_2\}$ *where* $\theta_1 = [a_0/\texttt{\&Even::IsEven}, a_1/\texttt{\&Even::IsOdd}]$, $\theta_2 = [a_0/\texttt{\&Odd::IsEven}, a_1/\texttt{\&Odd::IsOdd}]$ *for the terms* $H \cup H' = \{\texttt{vt[0]} = a_0, \texttt{vt[1]} = a_1\}$*, thereby preserving the information at the call site that both,* `vt[0]` *and* `vt[1]`*, are taken from the same object instance. However,* $\theta_2(H \cup H')$ *is not a superset of* H_g *and is therefore discarded by Eqn. 6.106 as it is not a specialization of table entry 2. Thus, the only recursive call applyEntries*$(s, H \cup H', \{\texttt{vt[0]} = \texttt{\&Even::IsEven}, \texttt{vt[1]} = \texttt{\&Even::IsOdd}\})$ *consults table entry 4 and applies summary* s_3 *to the caller state using Eqn. 6.104.*

6.3.4. Heyting Completion

In this section we show that the iterative tabulation of specialized function summaries is a Heyting completion, a well-known domain refinement technique [26]. A domain refinement adds new elements to an abstract domain. Our contribution is that completion is done on-demand, that is, only those elements are added to the lattice that are required by the program that is being analyzed.

Let $\langle L, \sqsubseteq_L, \sqcup_L, \sqcap_L\rangle$ be a complete lattice and $\alpha_X : L \to X$ a closure operator, i.e., monotone $Y \sqsubseteq_L Z \Rightarrow \alpha_X(Y) \sqsubseteq_L \alpha_X(Z)$, idempotent $\alpha_X(\alpha_X(Y)) = \alpha_X(Y)$, extensive $Y \sqsubseteq_L \alpha_X(Y)$, $\forall Y, Z \subseteq L$. Then $\langle L, \alpha, X, id\rangle$ is a Galois insertion [43].

Let $\Rightarrow \in L^2 \to L$ be a binary operator with $a \Rightarrow b = \bigsqcup_L\{c \in L \mid a \sqcap_L c \sqsubseteq_L b\}$. If $a \sqcap_L (a \Rightarrow b) \sqsubseteq_L b$ then $a \Rightarrow b$ is called the pseudo-complement of a relative to b. A lattice in which all pairs of elements have a pseudo-complement is called a Heyting algebra. We lift $\cdot \Rightarrow \cdot$ to sets $A, B \subseteq L$ as $A \Rightarrow B = \{a \Rightarrow b \in L \mid a \in A, b \in B\}$.

For any $X \subseteq L$ let $\curlywedge(X) = \{\bigsqcap_L Y \mid Y \subseteq X\}$ define the Moore closure of X. Let $A, B \in L$ such that α_A, α_B exist. Then the Heyting completion of A with respect to B is $\curlywedge(A \Rightarrow B)$. Let $\mathbb{H} = \wp(\bigcup_{H_G \subseteq GHerb}\{[\![test\ H_G]\!]^\sharp s \mid s \in MS^2 \times D\})$.

Theorem 1. *$\mathbb{H}$ is a Heyting completion of GHerb with respect to $MS^2 \times D$.*

Proof. First, show $\curlywedge(\mathbb{H}) = \mathbb{H}$. Given $S_1, S_2 \in \mathbb{H}$, let $S = \{s_1 \sqcap_{MS^2 \times D} s_2 \mid s_1 \in S_1, s_2 \in S_2\}$. Let $[\![test\ H_i]\!]^\sharp s_i \in S_i$ for $i = 1, 2$. Then S contains an element $s = [\![test\ H_1]\!]^\sharp s_1 \sqcap_{MS^2 \times D} [\![test\ H_2]\!]^\sharp s_2$. Here, $s = [\![test\ H_1]\!]^\sharp [\![test\ H_2]\!]^\sharp (s_1 \sqcap_{MS^2 \times D} s_2)$ if there exists $H \in GHerb$ with $[\![test\ H]\!]^\sharp = [\![test\ H_1]\!]^\sharp \circ [\![test\ H_2]\!]^\sharp$. If "$e = c_i$" $\in H_i$ exists with $c_i \in \mathbb{Z}$ and $c_1 \neq c_2$ then $s = \bot_{MS^2 \times D}$. Otherwise, since $s_1 \sqcap_{MS^2 \times D} s_2$ has finitely many fields, there exists a finite $H \subseteq H_1 \cup H_2$. Thus, $H \in GHerb$. It follows that $S_1 \sqcap_{\mathbb{H}} S_2 = S \in \mathbb{H}$.

Now show $\mathbb{H} = GHerb \Rightarrow MS^2 \times D$. Let $S_1 \sqsubseteq_{\mathbb{H}} S_2$ if for all $s_1 \in S_1$ there exists $s_2 \in S_2$ with $s_1 \sqsubseteq_{MS^2 \times D} s_2$ and note that $\sqcap_{\mathbb{H}}$ exists due to $\curlywedge(\mathbb{H}) = \mathbb{H}$. Choose $H \subseteq GHerb$, $b \in MS^2 \times D$. Let $a = [\![test\ H]\!]^\sharp \langle [], [], \top_D \rangle \in MS^2 \times D$. For the sake of contradiction, assume there exist c_i with $\{a\} \sqcap_{\mathbb{H}} c_i \sqsubseteq_{\mathbb{H}} \{b\}$ and $\{a\} \sqcap_{\mathbb{H}} (c_1 \sqcup_{\mathbb{H}} c_2) \not\sqsubseteq_{\mathbb{H}} \{b\}$. Let $C = c_1 \sqcup_{\mathbb{H}} c_2 := c_1 \cup c_2$. From the definitions of $\sqcup_{\mathbb{H}}$ and $\sqsubseteq_{\mathbb{H}}$ it follows that there is an element $c \in C$ such that $\{a\} \sqcap_{\mathbb{H}} \{c\} \not\sqsubseteq_{\mathbb{H}} \{b\}$. However, since c must originate from either c_1 or c_2, this is contradictory to $\{a\} \sqcap_{\mathbb{H}} c_i \sqsubseteq_{\mathbb{H}} \{b\}$. □

Corollary 1. *The entries of the table $T_H \in \mathbb{T}_{Herb}$ defined in Def. 8 are a partial Heyting completion of GHerb with respect to $MS^2 \times D$.*

Proof. Given some table entry $\langle f, H_g \rangle \mapsto \langle s, \emptyset \rangle \in T_H$, let $S = \{[\![test\ H_G]\!]^\sharp s \mid s \in MS^2 \times D\} \in \mathbb{H}$ and observe that $s \in S$. □

Note that in the implementation, the result of an application of the $\sqcup_{\mathbb{H}}$ operator never contains two summaries that are constraint by the same preconditions, i.e. share a semantically equal input memory structure. This is because we only compute a new table entry if we encounter a call site that instantiates a set of Herbrand queries with new, thus far unknown constants.

6.4. Implementation

In this section, we discuss some implementation aspects of our analyzer. Further details and our evaluation results are presented in Chap. 7. Our analyzer reconstructs the control flow- and call graph of an x86 binary. The input binary is decoded and translated into the RReil language using the GDSL toolkit, starting at all function entry points defined in the ELF header.

Inter-procedurally, the analysis computes summaries for all functions starting from the initial state *init* defined in Sect. 6.2.4. The fixpoint computation proceeds by

computing the summary of a callee before continuing at a call site using a dynamically updated partial order on the caller/callee relation. Intra-procedurally, the basic blocks of a function are discovered on-the-fly and we identify loops by observing jumps from higher to lower machine addresses. Within each loop, we apply a combined widening and narrowing operator for faster convergence [2].

The value domain D of the analysis is implemented as a set of three domains. The equality domain tracks predicates of the form $x = y + c$ for $x,\ y \in \mathcal{X}$ and $c \in \mathbb{Z}$. The pointer domain $D_X = X \to \wp(Loc \cup \{a_{bad}\}) \times X$ tracks relationships of the form $x_p - x_o \in \{l_1, \ldots, l_n\}$ with $x_p,\ x_o \in \mathcal{X},\ l_i \in Loc$. Here, x_p is the pointer variable that is being tracked, x_o contains the offset relative to the beginning of l_i, the addresses of a memory region. Finally, the value set domain is used to track finite subsets of $\mathbb{Z}$ and intervals. We impose no fixed bound on the size of the subsets (i.e. no k-limiting) but widen a growing set to an interval. The three domains form a hierarchy where a parent domain forwards any domain operation to its child. For instance, the pointer domain transforms operations on pointer variables to operations on pointer offsets and passes them on to its child domain.

Section 6.3.1 raised the possibility that only necessary preconditions can be synthesized that are represented by a Herbrand term *Set* .... For instance, the call to `Case3` in Fig. 6.14 would generate the term `parity->vtable` $= a$ which is translated to a precondition *Set* $\{$`p->vtable` $= a_1$, `q->vtable` $= a_2\}$. Currently, we handle this case by generating a query for each set element, thus resulting in a different table entry for each possible variable instantiation. While this does not affect the precision of the analysis, a more considered handling would allow us to reduce the size of the resulting tables.

Our analysis uses a simple means to flag summary memory regions that are created when accessing memory regions within loops. This way, we are able to emit a warning when accessing a summary memory region so that the analysis is sound if no warnings are emitted. A more precise handling of summary memory regions is future work.

Currently, each time a pointer is accessed that can be traced to the input, we create a fresh memory region. As a result, we implicitly assume that none of the pointer parameters alias. As explained in Sect. 6.2.3, a conflicting aliasing assumption is recognized during summary application and the respective summary is ignored. Again, a warning can be generated such that the analysis is sound if no warnings are emitted. Future work will address how to incorporate the input aliasing configurations into the tabulation scheme such that a function is re-analyzed with specific aliasing assumptions as soon as this is required by a call site.

6.5. Related Work

One traditional approach of improving the precision of context-insensitive analysis is to only merge call sites whose last k parent call sites are the same (so-called k-CFA) [51]. While the k-CFA approach improves the precision (i.e. Fig. 6.1 verifies with $k = 1$), it does so without consideration for the semantics of the program.

Modular analyses are context-sensitive by combining summaries of components or functions to a solution of the whole program. There are four principles [14]: compute a global fixpoint over some simplified semantics of each component, compute summaries under worst case assumptions, compute summaries using (possibly user-supplied) interfaces, and symbolic relational separate analysis (input/output abstractions). Most analyses combine some of these four principles.

Analyses that rely on condensing domains [25, 40, 42, 46, 54] perform a pure symbolic relational analysis based on a restricted class of domains that comprise Herbrand terms with variables, Boolean functions and affine equalities.

The SeaHorn analyzer allows arguing over rich, numeric properties in a modular way [27]. It simplifies the input program into Horn clauses over predicates that are tailored to the analyzed program. These are then solved in a modular way. The downside is that no new invariants can be synthesized inter-procedurally. Our tabulation over Herbrand terms is, in theory, less efficient than SeaHorn's Horn clauses since we store a summary state for each set of predicates. Yet, our summaries allow the computation of new invariants even inter-procedurally.

Specializing the input of a summary falls into the category of summarizing with interfaces. One instance of this idea is the inference of preconditions that, when violated, lead to an error in the analyzed code [16]. An approach called "angelic verification" [17] goes further by restricting inputs to likely correct inputs.

Modular analyses that re-evaluate a component several times also adhere to the principle of computing summaries with interfaces, as each summary of a component is somehow specialized. The classic work on tabulation proposes to analyze a function for any possible input state and to combine table entries that match a call site [50]. Our approach is an on-demand tabulation that uses concrete values of function pointers as keys. Amato et al. perform tabulation based on the equality of the abstract input state [2]. Their tabulation approach may re-analyze a function unnecessarily, i.e. when a call site state has no match in the table but matches the join of several tabulated states. Moreover, matching tabulated states by equality may lead to non-monotone behavior [2, Example 1].

In the context of binary analysis, Xu et al. manually summarise functions using pre- and postconditions [64] that are similar to our Herbrand terms.

Finally, one "simplified semantics" idea is to break the program down so that it

consists of parts that can be summarized with little precision loss (with the extreme of synthesizing transfer functions for groups of instructions [9, 57]).

6.5.1. Conclusion

We presented a framework for modular analysis that judiciously computes multiple summaries. Each summary is specialized by Herbrand terms whose template is created by the function that is being analyzed and that is instantiated by its callers. We illustrated that this versatile approach corresponds to an on-demand Heyting completion of the domain and recovers indirect function calls.

Part IV.

Analysis Implementation and Evaluation

7. The *Summy* Analysis Tool

In part II we have presented the GDSL toolkit, a software framework for machine instruction disassemblers and semantics translators. GDSL translates machine code into our intermediate representation RReil, on top of which we have built our analysis tool *Summy*. The tool implements the analysis discussed in Chap. 6. In the following, we show how to use the tool and discuss the format of its output. Thereafter, we detail the design decisions we have made during the development and go into the most important details of the implementation.

7.1. Getting Started

Summy is distributed as open source software and is available as a Git repository at `https://github.com/gdslang/summy`. The analyzer runs on Linux only and can be built using the following shell commands:

```
1 git clone --recursive https://github.com/gdslang/summy.git
2 mkdir summy/build
3 cd summy/build/
4 cmake .. && make
```

Note that we use the `--recursive` flag for the `git clone` command which makes sure that the GDSL toolkit is cloned alongside with Summy. Building Summy requires CMake[1] and a C/C++ compiler (LLVM clang or GCC) recent enough to support C++17. GDSL is automatically built together with Summy, so all its dependencies are required as well – in particular, the MLton standard ML compiler[2]. After building Summy, a front-end for the machine architecture that we want to analyze needs to be chosen; in the following example, we select the front-end for the x86 architecture:

```
1 ln -s dependencies/gdsl-toolkit/libgdsl_x86_rreil.so libgdsl-current.so
```

[1] `https://cmake.org/`
[2] `http://mlton.org/`

The successful setup of Summy can be verified by running the tests using the `make test` command.

7.2. Running the Analyzer

The repository contains a simple driver tool (see the `tools/` folder) that can be used to run the analyzer on an ELF binary. It is crucial for our analysis to find a set of function start addresses within the executable. To this end, the driver tool performs the following preparatory steps:

1. **Extraction of function start addresses from ELF data**: The ELF data of a binary contains a number of function addresses, for example an entry point if the binary is an executable program (and not a library). It also contains the addresses and names of exported functions; associating function start addresses with names is helpful for debugging the analyzer.

2. **Call scanning**: A program usually consists of a large number of direct calls. The callee addresses can be collected and added to the set of known function start addresses. Functions that are only called indirectly cannot be found this way; indeed, resolving these indirect function calls is the task of the main analysis.

The main analysis uses the set of functions collected this far as a starting point. During the main analysis, more functions are discovered from indirect jumps. The main fixpoint computation only terminates after having analyzed each known function at least once (due to tabulation, a single function can be analyzed multiple times). In the following, we refer to program points using the term *node* as each node in a control flow graph represents a program point while an edge represents a statement. The driver tool accepts the command-line arguments described below.

- `--noopt`: Disable any RReil code optimizations that would result in the loss of a well-defined mapping between individual machine instructions and RReil code blocks. See Sect. 7.4 for details on the effects of turning optimizations on and off.

- `--noref`: Disable reference management for analysis states. With reference management turned on, the analysis state of a node is only kept if it may be involved in the computation of future states (e.g. during widening) or at function return sites. In particular, the states of intermediary nodes of straight-line code are freed, greatly reducing the memory footprint of the analyzer. Disabling reference management is helpful for debugging.

- `--node=n`: Output the whole state (instead of just the node id) for the node *n* in the output control flow graph. This argument can be passed multiple times.
- `--func=f`: Only analyze the function with the name `f`. Note that this only works if the ELF data of the binary contains a function with name `f` and a corresponding start address. This argument can be passed multiple times.

As a final argument, the analyzer expects a path to the ELF executable or library to analyze. As an example, the following command analyzes the `/bin/echo` program, outputting the whole state of nodes 42 and 247:

```
tools/driver/driver --node=42 --node=247 /bin/echo
```

7.3. Output of the Driver Tool

The analyzer outputs two files, `cfg.dot` and `cfg_machine.dot`. The former contains the inter-procedural control flow graph as discovered by the analysis. The latter contains the same graph, however reduced to the machine address level; the control flow within instructions or machine-level basic blocks[3], consisting of RReil code, is abstracted away. Note that depending on the level of optimization (see Sect. 7.4), the RReil translator either translates one machine instruction or one whole basic block at a time. A node in the machine control flow graph either represents an instruction or a basic block starting at a specific address. Thus, the machine control flow graph is much smaller, but does not offer enough information to debug the analyzer in detail.

As an example, consider the following simple C function:

```c
int f(int a, int b) {
  if(a > b)
    b = a;
  return a + b;
}
```

Fig. 7.1 shows a simplified and polished version of the resulting control flow graph with RReil optimizations turned on. For the last node of the graph, printing the state has been enabled. The conditional in line 2 can be found at node 249 in the graph. Due to our forward expression substitution, the branch condition has been substituted into

[3]We use the term *basic block* for any sequence of instructions that are executed consecutively; a basic block ends with a jump, call, or return instruction.

the actual test that determines which of the two branches is taken (shown as dotted blue graph edges). As a result, the analysis is able to constrain the values of `DI` and `SI` appropriately. We now detail the contents of the example node 260. In the example, the analysis tracks a state only for the default context. The label *context 0* refers to the line index of the table of the currently analyzed function. Next, memory input and output regions are shown. Note that instead of using field names, our implementation works with bit offsets into the different memory regions. Thus, a field has no name but a start address and a size; see Sect. 7.6 for details. The final section of the state contains the value domain state. In the example, there are three different child domains:

- The affine domain at the top tracks relationships of the form $x = y + c$ for two variables x and y and a constant offset c. For example, variable `#280` is equal to variable `#282` with an offset of zero; this is printed as `Eq(#280) -> {..., (#282 -> 0)}` in the figure.

- The alias domain tracks aliasing relationships. It uses the value set domain (see below) to model offsets. For example, the stack pointer output variable `#287` aliases the stack pointer input variable `#286` at offset 8 (printed as `P(#287) -> {<#286>}` in the alias domain and `#287 -> {8}` in the value set domain), i.e. it has been increased by 8 by the function. Integer values that do not represent a pointer and pointers to absolute addresses are stored as offsets in the value set domain. In this case, the alias domain tracks the `nullptr` as alias. For example, consider the instruction pointer (IP) – its value domain variable aliases the `nullptr` at an offset of 4195465. Thus, it contains the integer value 4195465. We use a special *bad* pointer to indicate that some variable may alias an invalid pointer.

- The value set domain tracks value sets and is used by the alias domain to store offsets. It can track finite sets of values and half-open intervals.

On first sight, the affine and alias domains seem to be storing similar properties. Note, however, that the alias domain tracks may-alias information while the affine domain tracks must-equal information. For example, `Eq(#0) -> {(#0 -> 0), (#1 -> 0), (#2 -> 0)}` in Fig. 7.1 means that variable `#0` is equal to itself and both variables `#1` and `#2` (the offset is 0 in all three cases) while `P(#0) -> {#1, #2}` means that `#0` either aliases `#1` or `#2`. As mentioned above, the analyzer also outputs a condensed *machine level* control flow graph that only contains a single node for each analyzed machine address; Sect. 7.4 discusses the machine level control flow graph for an example assembly program with and without RReil optimizations turned on.

In addition to the control flow graphs, the analyzer also outputs data for debugging and evaluation to the standard output. Fig. 7.3 shows an example of this output. The analysis commences by collecting function start addresses as can be seen in lines 1-4.

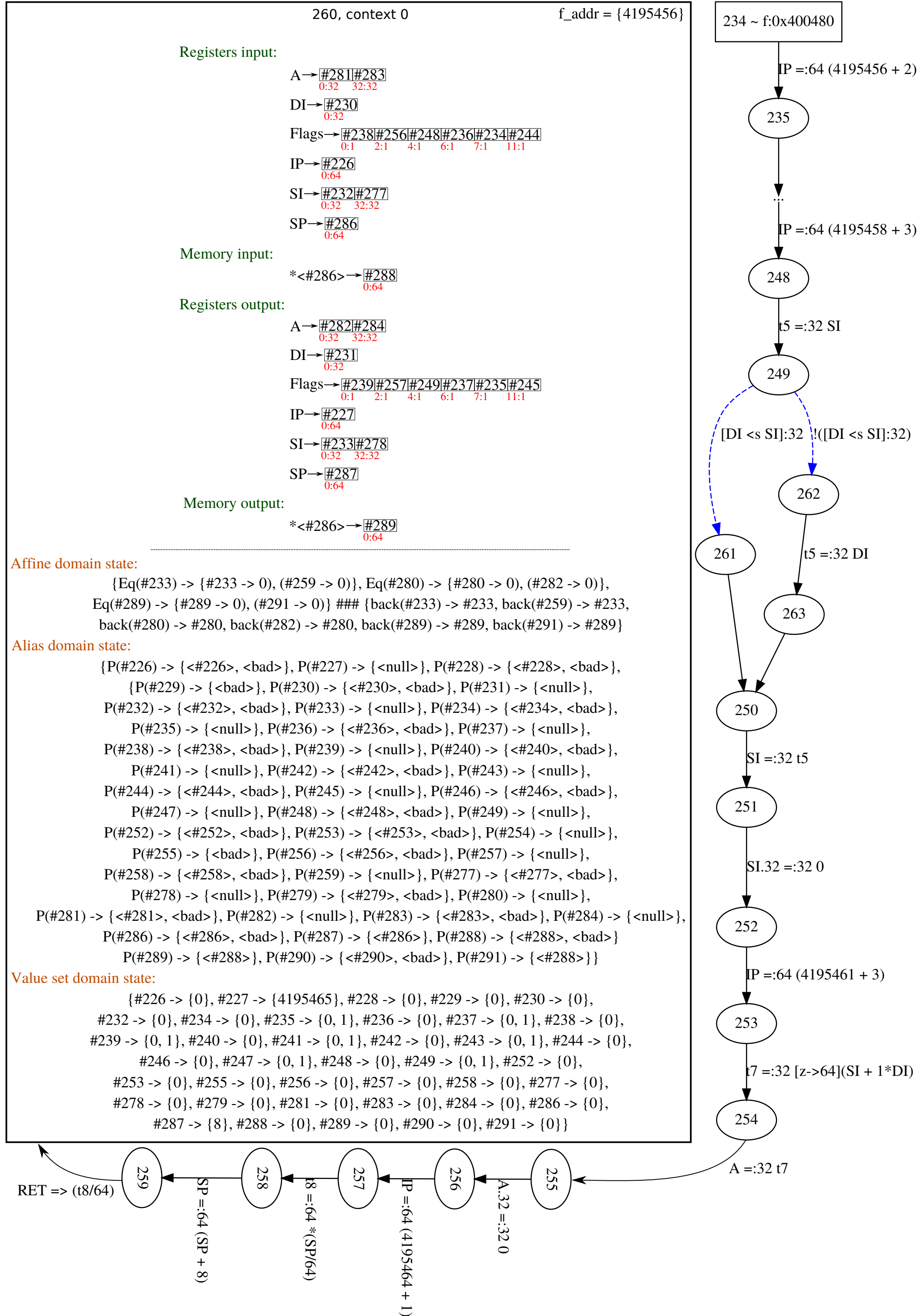


Figure 7.1.: Example control flow graph.

```
1 f:
2 ret
3
4 main:
5 mov %r11, f
6 xor %r11, 42
7 xor %r11, 42
8 call *%r11
9 ret
```

Figure 7.2.: Simple example for precision loss.

Next, the main analysis is started (lines 6-8). During the main analysis, the analyzer outputs progress data, e.g. the number of nodes visited (not shown here). After the analysis, statistical data is printed. The first block of statistics contains data about the binary (lines 10-13); here, the section size refers to the `.text` section of binary that contains all executable code. The number of decoded bytes are usually smaller than the section size. There are three possible reasons for bytes in the `.text` section not to be decoded by the analyzer:

- They belong to one or multiple functions that are not reachable from the functions discovered in the function collection phase of the analysis which has been described at the beginning of Sect. 7.2.
- They are part of padding; compilers sometimes insert padding bytes at the end of functions in order to make sure that function start addresses are aligned.
- The analyzer is unable to discover the target of a branch or call due to precision loss. A simple example for this case is shown in Fig. 7.2. Here, two `XOR` operations are used on the value of a pointer. While the value of the pointer does not change, the points-to relationship with the function `f` is lost. As a result, the function `f` is not decoded or analyzed (if its address is not found in the ELF header, for details see Sect. 7.2).

Next, the analyzer outputs data about its success in resolving branch targets (lines 15-20). Since the amount of actually possible branch targets per call site or jump instruction is not known, this is only meant as a rough indication of the performance of the analysis. Finally, lines 22 through 29 contain statistics regarding the tabulation of functions. In the example, there are 14 table entries for 12 functions; thus, at least one function has been

analyzed in multiple contexts. These two additional table entries correspond to two non-zero analysis contexts (context zero is the default context with no specializations), see line 27. Line 28 contains the number program points and Herbrand terms for which it was not possible to match the query to the given state. This happens, for example, if the query dereferences a field which does not point to a memory region in the caller state, e.g. because it has been modified using bitwise operations that result in a loss of aliasing information.

7.4. RReil Code Optimization

As described in Sect. 3.4, our intermediate representation RReil is designed towards allowing optimization. In order for optimization to be effective, the optimizer needs to transform the RReil statements of multiple machine instructions or even basic blocks in combination. As a consequence, transformations cut across the boundaries of individual machine instructions such that a well-defined mapping from machine instructions to a sequence of RReil statements is lost. Thus, with optimizations turned on, applying the RReil translator to a machine address does not return a RReil program for one instruction, but the optimized program for the basic block starting at the given address.

As an example, consider the following x86 code that implements a simple loop, using `r11` as loop counter:

```
1  main:
2  0x00: mov $0, %r11
3  head:
4  0x07: cmp $100, %r11
5  0x0b: je end
6  0x0d: inc %r11
7  0x10: jmp head
8  end:
9  0x12: ret
```

In order to ease the understanding of the control flow graphs, the listing contains the machine addresses of the instructions at the beginning of each line. Note that in line 7, the control branches back to evaluating the loop condition which lies in the middle of top basic block. Fig. 7.4 shows the control flow graph as recovered by the analysis without RReil optimizations on the left. Again, each node contains its id (first number) and the machine address of the respective instruction (second number). There is a 1:1 correspondence between the instructions in the assembly code shown

```
*** Starting the 'fcollect' analysis...
*** Collecting functions from ELF data...
Adding function 4004b0 (main)
Adding function 4004a0 (h)
(...)
Starting main analysis.
(...)
End of main analysis.
(...)
Section size: 466
Decoded bytes: 311
Analyzed addresses: 27
Decoded start addresses: 27
(...)
Total indirect branches: 7
Indirect branches with targets: 4 (57.142857%)
Total indirect jmps: 2
Indirect jmps with targets: 0 (0.000000%)
Total indirect calls: 5
Indirect calls with targets: 4 (80.000000%)
(...)
Maximum table entries: 3
Average table entries: 1.166667
Total number of functions: 12
Total table entries: 14
Total number of field requests: 1
Non-zero contexts at head nodes: 2 (zero: 5)
Path construction errors: 0
Zero HBs: 6, one HB: 1, multiple HBs: 0
```

Figure 7.3.: Example statistical output of the analyzer.

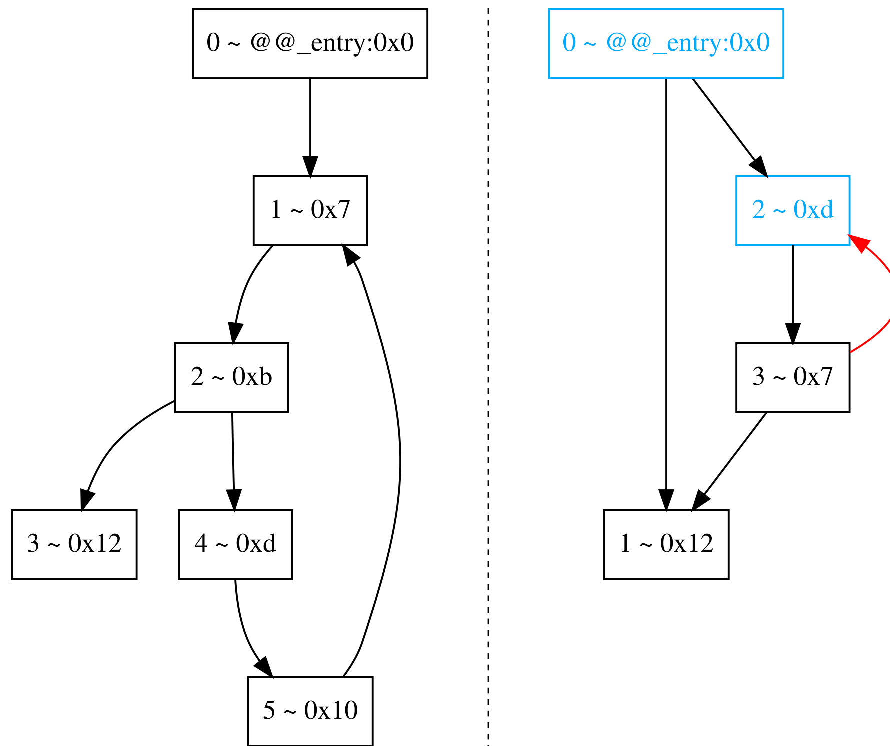


Figure 7.4.: Example control flow graph without (left) and with (right) RReil optimizations applied.

above and sub-graphs of the control flow graph; as a result, this representation lends itself for debugging. The control flow graph resulting from running the analyzer with optimizations turned on is depicted on the right. This time each node represents a basic block. However, since branches can jump to addresses inside existing basic blocks, the resulting blocks can overlap. In the example graph, the back branch from node 3 (shown in red) leads into the middle of the entry basic block. As a result, these two blocks (shown in light blue) overlap regarding the instructions from which their RReil code has been translated. Thus, the analysis may store multiple states for the same original program point, differentiating between different paths on which control reaches it. This can lead to a improved precision for these program points which, in turn, may have unexpected effects and, thus, makes debugging harder.

Our analyzer relies on our forward expression substitution pass as described in Sect. 3.4.2. As a consequence, turning block-wise optimizations off renders the analyzers unable to constrain variables that take part in the evaluation of branch conditions. This leads to a loss of precision.

$$locSmaller : Loc_S \times Loc_S \rightarrow Boolean$$
$$locSmaller(l_1,\ l_2) =$$
$$\text{if } stackDepth(l_1) \neq stackDepth(l_2) \tag{7.1}$$
$$\text{return } stackDepth(l_1) < stackDepth(l_2) \tag{7.2}$$
$$\text{if } machineAddress(l_1) \neq machineAddress(l_2) \tag{7.3}$$
$$\text{return } machineAddress(l_1) < machineAddress(l_2) \tag{7.4}$$
$$\text{return } nodeId(l_1) < nodeId(l_2) \tag{7.5}$$

Figure 7.5.: Program location comparer.

7.5. Fixpoint Computation

Our fixpoint implementation is based on worklist iteration. We use the box operator [3] for combined widening and narrowing. Using the operator requires that nodes are always visited in the same order so that termination is guaranteed. While any fixed order is sufficient for termination, choosing a meaningful order is crucial for the efficiency of the analyzer. However, finding such a node order is a non-trivial problem, in particular since we discover the control flow graph on the fly during the analysis. As a compromise between efficiency and simplicity, we have chosen a heuristics to impose a total ordering on node locations in our priority queue which we use as worklist. Our node comparator is shown in Fig. 7.5. We compare the nodes by location and locations based on the properties shown below. The first property regarding which two locations l_1 and l_2 are not equal defines order of the nodes at that l_1 and l_2.

1. First, we compare analysis specific properties; currently, the only property is the call stack depth of nodes. The idea behind this is to prioritize the computation of callee nodes over caller nodes. In Eqns. 7.1f, we use the function *stackDepth*() in order to retrieve the stack depth of a location. For this, we define the stack of a program location to be the stack of function calls the analyzer is analyzing when it first discovers a location. The stack depth is then defined as the number of functions in the stack. The code to track the current stack and store a stack depth together with each location is not shown.
2. Next, we compare nodes by machine address (see Eqns. 7.3f). Generally, control is more likely to flow in the direction of increasing than decreasing addresses. As a result, we prioritize nodes with a lower machine address (as we run a forward analysis).

3. Finally, within one machine instruction (or basic block if block-wise optimization is turned on), we resort to the numeric ids of nodes (see Eqn. 7.5). This way, we order the nodes according to the order in which they have been created during disassembly.

We identify widening / narrowing points using the *isBackEdge*(l_{from}, l_{to}) function that determines whether the edge from l_{from} to l_{to} is a backward edge. For example, an edge from a program point p that originates from a machine instruction at address a_p to a program q that originates from a machine instruction at a different address a_q with $a_q < a_p$ is a backward edge. If an edge is a backward edge, we apply the box operator on that edge. This approach guarantees at least one widening / narrowing point for each loop.

Because of using combined widening and narrowing through the box operator, our fixpoint algorithm iterates until states do neither grow nor shrink any longer. Newly discovered parts of the binary are disassembled on the fly and the resulting control flow graph nodes are added to the worklist in order to make sure that they take part in the fixpoint computation.

Fig. 7.6 contains the function *computeFixpoint* which implements our generic fixpoint algorithm. It expects a program *prog* as parameter which is used as a starting point for the analysis. The function computes a map $abs \in Loc_S \to State$ which assigns an abstract state to each program location. As explained in Sect. 7.2, we make use of a few simple heuristics to discover a base set of function start offsets within a given binary. The program is subsequently extended (see below) during fixpoint computation as new parts of the input binary program are discovered and finally returned together with the mapping *abs*. The function $next : Loc_S \to \wp(Loc_S)$ models the edges of the control flow graph. Note that given a call site l_c, $next(l_c)$ contains all program locations $l_f \in Loc_F$ that have been discovered as possible call targets of the call at l_c. In turn, given a return location l_f^{res} of a function f, the set $next(l_f^{res})$ contains all instructions following a call instruction which may call function f. As a result, the specialization of an initial state of a function can be implemented as a transformer on an edge from a call site to a called function, while the application of a summary can be implemented as a transformer on an edge from a return location to an instruction following a call[4]. The fixpoint computation proceeds using the loop in Eqn. 7.9 which updates the state of one program location in each iteration. We make use of a two-stage worklist initialized in Eqn. 7.6 and Eqn. 7.7, respectively. This way, we make sure that narrowing is performed after widening as explained in Sect. 7.5.1. The set *postproc* contains all locations for which the last application of the box operator has resulted in widening (see Eqns. 7.27 and 7.28). After finishing the main worklist iteration, such nodes are revisited by again

[4]The summary application additionally needs the state at the call instruction; this is not shown in Fig. 7.6.

adding them to the main worklist in Eqns. 7.10ff. The main worklist is initialized to the set of locations that do not appear as successor of any other location; this way, we approximate the set of function start addresses Loc_F within *prog* (see Eqn. 7.6). The next location to process is determined by a call to *orderedDequeue* in Eqn. 7.12 which dequeues a location while adhering to the fixed order of program points shown in Fig 7.5. Given this location, we iterate all incoming edges in Eqn. 7.16 and evaluate the abstract transformer associated with the statement on that edge (Eqn. 7.17). This transformer returns a set of abstract states and program locations where one element matches the control flow graph edge from l_{pred} to l.

Note that the transformer may also discover new instruction addresses. We model this by allowing the transformer to return an additional set L_{new}. The instructions at these new program addresses are decoded, translated, and added to the program in the following equation. Next, Eqn. 7.21 checks whether the current control flow graph edge is a backward edge. If so, we have to apply the box operator as shown in Eqn. 7.22. Note that the application of the box operator additionally returns whether it has applied widening; as explained above, this is important because such nodes have to be revisited. Finally, Eqns. 7.29 through 7.31 add successor program points of the current program point l to the worklist and update the state map if the state at l has changed. The function returns the updated program and the mapping that assigns an abstract state to each program location.

$$
\begin{array}{ll}
\mathit{computeFixpoint} : (\mathit{Loc}_S \to \mathcal{L}(\mathit{Stmt})) \to (\mathit{Loc}_S \to \mathcal{L}(\mathit{Stmt})) \times (\mathit{Loc}_S \to \mathit{State}) & \\
\mathit{computeFixpoint}(\mathit{prog}) = & \\
\quad \text{let } \mathit{worklist} = \{l \in \mathrm{dom}(\mathit{prog}) \mid \forall l' \in \mathrm{dom}(\mathit{prog}) \setminus \{l\}.\ l \notin \mathit{next}(l')\} & (7.6) \\
\quad \text{let } \mathit{postproc} = \emptyset & (7.7) \\
\quad \text{let } \mathit{abs} = \{l \mapsto \bot \mid l \in \mathrm{dom}(\mathit{prog})\} & (7.8) \\
\quad \text{while } \mathit{worklist} \cup \mathit{postproc} \neq \emptyset \text{ do} & (7.9) \\
\quad\quad \text{if } \mathit{worklist} = \emptyset \text{ then} & (7.10) \\
\quad\quad\quad \mathit{worklist} := \mathit{postproc};\ \mathit{postproc} := \emptyset & (7.11) \\
\quad\quad \text{let } \langle l,\ \mathit{worklist} \rangle = \mathit{orderedDequeue}(\mathit{worklist}) & (7.12) \\
\quad\quad \text{let } \mathit{preds} = \{l' \in \mathrm{dom}(\mathit{prog}) \mid l \in \mathit{next}(l')\} & (7.13) \\
\quad\quad \text{let } s = \text{ if } \mathit{preds} = \emptyset \text{ then } \mathit{init} \text{ else } \bot & (7.14) \\
\quad\quad \text{let } \mathit{widened} = \mathit{false} & (7.15) \\
\quad\quad \text{for } l_{pred} \in \mathit{preds} \text{ do} & (7.16) \\
\quad\quad\quad \text{let } \langle \{\langle l,\ s' \rangle,\ \ldots\},\ L_{new} \rangle = [\![l_{pred} : \mathit{prog}(l_{pred})]\!]^{\sharp}\ \mathit{abs}(l_{pred}) & (7.17) \\
\quad\quad\quad \mathit{abs} = \mathit{abs} \cup \{l \mapsto \bot \mid l \in L_{new}\} & (7.18) \\
\quad\quad\quad \mathit{prog} := \mathit{prog} \cup \mathit{decodeAndTranslate}(L_{new}) & (7.19) \\
\quad\quad\quad \mathit{worklist} := \mathit{worklist} \cup \{l \in L_{new} \mid \forall l' \in L_{new} \setminus \{l\}.\ l \notin \mathit{next}(l')\} & (7.20) \\
\quad\quad\quad \text{if } \mathit{isBackEdge}(l_{pred},\ l) \text{ then} & (7.21) \\
\quad\quad\quad\quad \text{let } \langle \mathit{widened}_{edge},\ s_{boxed} \rangle = \mathit{abs}(l) \mathbin{\Box} s' & (7.22) \\
\quad\quad\quad\quad s := s \sqcup_{MS^2 \times D} s_{boxed} & (7.23) \\
\quad\quad\quad\quad \mathit{widened} := \mathit{widened} \vee \mathit{widened}_{edge} & (7.24) \\
\quad\quad\quad \text{else} & (7.25) \\
\quad\quad\quad\quad s := s \sqcup_{MS^2 \times D} s' & (7.26) \\
\quad\quad \text{if } \mathit{widened} \text{ then} & (7.27) \\
\quad\quad\quad \mathit{postproc} := \mathit{postproc} \cup \{l\} & (7.28) \\
\quad\quad \text{if } s \neq \mathit{abs}(l) \text{ then} & (7.29) \\
\quad\quad\quad \mathit{worklist} := \mathit{worklist} \cup \mathit{next}(l) & (7.30) \\
\quad\quad\quad \mathit{abs}(l) := s & (7.31) \\
\quad \text{return } \langle \mathit{prog},\ \mathit{abs} \rangle & (7.32)
\end{array}
$$

Figure 7.6.: Fixpoint algorithm.

```
1  int f() {
2    int x = 4;
3    while(rnd())
4      x = 2;
5    return x;
6  }
```

Figure 7.7.: Example code for a loss of precision without an additional narrowing iteration.

7.5.1. Additional Narrowing Iteration

As explained above, our fixpoint algorithm uses a two-stage worklist that makes sure that we revisit a program point after performing widening during the application of the box operator. This is not required for a sound analysis result but potentially improves precision as demonstrated in the following. Consider the code in Fig. 7.7. The following table contains the steps performed by the fixpoint algorithm given a simple domain that tracks sets of values for x that are abstracted to half-open intervals during widening:

iteration	program point	state of x	worklist	postproc	remark
1	2	$\{4\}$	[3]	[]	
2	3	$\{4\}$	[4, 5]	[]	
3	4	$\{2\}$	[3, 5]	[]	
4	3	$[-\infty; 4]$	[4, 5]	[3]	widening
5	4	$\{2\}$	[5]	[3]	
6	5	$[-\infty; 4]$	[3]	[]	swapped worklist
7	3	$\{4, 2\}$	[4, 5]	[]	narrowing
8	4	$\{2\}$	[5]	[]	
9	5	$\{4, 2\}$	[]	[]	

Here, the *iteration* column contains the iteration count for the main loop in Eqn. 7.9. A *program point* corresponds to a line in the program; the state at a line is defined as the state tracked after applying the transformer of the statement on that line. The *worklist* and *postproc* columns show the contents of the worklist and the postproc list at the end of the respective iteration. Note that in iteration 5, the state of x at line 4 does not change although the box operator applied widening in iteration 4, yielding a less precise value for the loop head. As a result, the loop head in line 3 is not put into the

worklist again. Instead, iteration 6 evaluates the statement after the loop, resulting in the use of the widened value for x after the loop. After this iteration, the worklist is empty and the resulting state is a post-fixpoint. In order to refine this post-fixpoint and, thereby, the state at line 5, we re-evaluate the nodes in *postproc* which contains the loop head. This time, the box operator narrows the state at the loop head which then propagates to line 5, resulting in a more precise fixpoint for x.

7.6. Implementation of Memory Regions

According to our presentation in Chap. 6, a memory region maps named fields to value domain variables. However, our implementation uses a bit offset and size instead of a name to identify fields. As an invariant, our implementation makes sure that the fields of a single memory region never overlap. When using field names, it is sufficient to distinguish cases where fields exist or do not exist during memory accesses. However, when identifying fields by offset and size, more complex scenarios need to be considered due to overlappings between accessed parts of a region and the fields that are contained in the region.

7.6.1. Handling of Conflicting Accesses

Fig. 7.9 shows a few possible overlapping scenarios. Here, case (1) is a fully a aligned access which is similar to an access to an existing field in Chap. 6. Reading resp. writing such a field amounts to reading resp. writing the value domain variable of the field f_1. All other cases correspond to conflicting accesses into a memory region. It is possible to handle them by replacing all conflicting fields with fresh fields that contain unrestricted values. However, since our analyzer works on assembly code, this leads to precision loss. In order to see why, consider the Intel assembly code on left of Fig. 7.8. Here, we first load the address of a function into the register eax which corresponds to the lower 32 bis of register A. Compilers generate this instruction instead of a move to the full 64 bit register if they know the size of the function address fits into 32 bits. The 32 bit move instruction is preferable because the 64 bit version is two bytes longer. Note that the upper 32 bits of register A are automatically set to zero when writing the lower half of the register. This effect is made explicit in the translated RReil code shown on the right of Fig. 7.8. However, the call instruction accesses the full 64 bits of the register. As a result, when evaluating the call target during our analysis, a 64 bit field is queried within a memory region that contains two 32 bit fields. The corresponding access pattern is depicted in case (2) of Fig. 7.9.

Because the approach shown in the above example is very common in real-world binaries, our analyzer supports this access pattern without loss of precision. For this,

```
1 mov eax, func
2 call rax
```

```
1 IP =:64 (IP + 5)
2 A =:32 func
3 A.32 =:32 0
4 IP =:64 (IP + 2)
5 SP =:64 (SP - 8)
6 *[64]SP =:64 IP
7 goto [CALL] [64]A
```

Figure 7.8.: Example assembly (left) and RReil (right) code that results in an access that corresponds to case (2) in Fig. 7.9.

we first distinguish between read and write accesses. A conflicting read access does not change the fields tracked by the respective region. If the read access does not exactly cover a contiguous range of fields, the access returns an unconstrained value (see, for example, cases (3) and (4) of Fig. 7.9). If the read access covers a contiguous range of fields, we replace the access to a single field with a shift expression. For example, assume that the access in case (2) of Fig. 7.9 is a read access to region r and that field f_2 has a size of 14 bits. Then, the access returns the value domain expression $r(f_1) * 2^{14} + r(f_2)$[5]. Write accesses, on the other hand, always replace conflicting fields with a single new field that spans the accessed bit range.

7.6.2. Further Ideas for Improvement

The above ideas allow to preserve values when larger fields of a memory region (e.g. a 64 bit field) are created by incrementally writing smaller parts (e.g. higher and lower 32 bit fields). The approach could be easily extended to support the opposite direction – i.e. reading a smaller part of a field – by masking bits outside of the accessed range and shifting the resulting value(s) into place. Further note that merging and partitioning of fields can lead to a loss of precision – for example, think of an aliasing domain which looses information about aliasing relationships when fields are merged. A possible solution to this problem could be to allow fields to overlap.

7.7. Evaluation

We have evaluated our implementation on the set of example binaries shown in Table 7.1. In particular, the benchmarks starting with `libgdsl` are GDSL decoder and translator

[5]Note that this assumes that f_1 and f_2 contain nonnegative values only. A simple means to handle negative values would be to emit a warning or widen den result to $\top$.

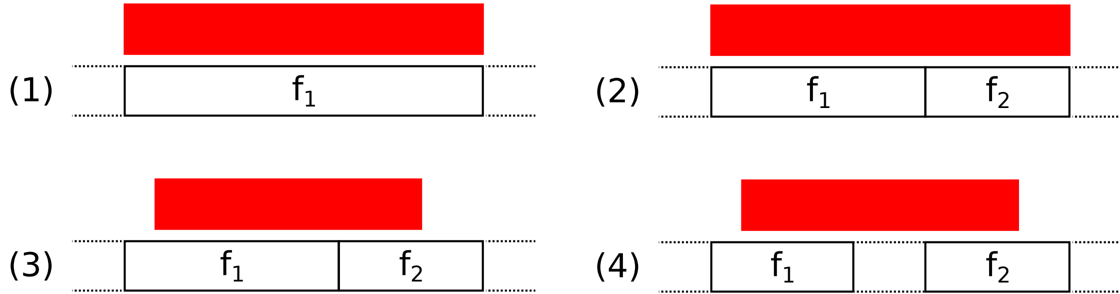


Figure 7.9.: Different access patterns; the accessed part of the region is marked in red.

Binary	**Exact** *H*	**Set** *H*	**None**	**Max. Tbl.**	**Avg. Tbl.**	**Ind.**	**Res.**	**Time**	**Size**
libgdsl_avr	147	0	3	23	1.054	223	156	9.63m	300kb
libgdsl_arm7	88	0	7	33	1.084	152	71	11.0m	406kb
echo	0	0	0	1	1.000	6	2	98s	7.9kb
cat	0	0	0	1	1.000	6	2	5.7m	11.9kb

Table 7.1.: Evaluation Results

libraries. As described in Chap. 5, the GDSL code is translated into idiomatic C code where higher-order functions are translated into C function pointers or heap-allocated closures containing function pointers.

Column **Exact** *H* contains the number of *call* / *br* statements for which we were able to synthesize an exact precondition (see page 105). The column **Set** *H* reports call sites that generate a term with a *Set* constructor, i.e. the cases where only necessary preconditions can be synthesized. The number of Herbrand terms that could not be translated to an input memory field is shown in **None**. Column **Max. Tbl.** contains the maximum size a single function table has got while column **Avg. Tbl.** contains the average table size. Columns **Ind.** and **Res.** show the total number of indirect *call*/*br* statements and the fraction of them that were resolved to at least one target. Not all call sites can be resolved. This can be caused by imprecision in our analysis, i.e. if the analyzed program uses complex expressions to compute a target address. In addition, it is also possible for targets to depend on running time data. For example, a library may have exported functions that expect function pointers as parameter. Finally, columns **Time** and **Size** contain the analysis time and the size of the `.text` section.

Note that the gathering of the experimental data has been done using a simplified implementation that does not build a minimal set of function summaries as detailed in Sect. 6.3.3. Instead, a summary is computed for every possible instantiation of the

Herbrand terms. This may lead to a greater table size in case one query depends on the answer to another query. In order to illustrate this, consider the following code:

```
int b() { return 2; }

int c() { return 3; }

int f(int (*fun1)(), int (*fun2)(int), int (*fun3)(int)) {
  int (*fp)();
  if(fun1()) fp = fun2;
  else fp = fun3;
  return fp() + fp();
}

int main() {
  auto pred = rnd() ? []() { return 1 } : []() { return 0 };
  return f(pred, rnd() ? &b : &c, rnd() ? &b : &c);
}
```

In the code, function `main` passes a predicate (i.e. a function pointer to a function returning a boolean value) to function `f` that assigns a function pointer conditionally depending on the return value of the predicate (lines 7 through 8). Note in particular that in this example the value of `fp` is fixed after picking one of the two possible predicates in `main`.

An analysis that is implemented as shown in Sect. 6.3.3 would build a table of size 5. The table would contain one entry for analyzing `f` without queries and two entries for each value of `pred`. This is because each assignment for `pred` results in one further query, i.e. for either the second or third function parameter. Our current analysis implementation, on the other hand, would produce 9 table entries - one entry for the default case with no query answers and one entry for each possible assignment of the three parameters of `f`.

Note that the summary application detailed in Sect. 6.2.3 and the fixpoint algorithm discussed in Sect. 7.5 do not support recursion. This is because the summary application expects the caller state and the summary to share no variables and the fixpoint algorithm uses a node ordering that always prioritizes callee nodes. In addition, widening would need to be applied to table entries in order to ensure termination in the presence of recursion. As a result, recursive calls are currently ignored in the implementation. Proper support for recursion is future work.

Part V.

Conclusion

Binary program analysis is an interesting subdomain of program analysis. It can be applied in cases where the source code of a program is not available, the semantics of a high level programming language is not fully defined, or possible compiler bugs have to be taken into account. However, binary program analysis also poses challenges, in particular due to the complexity of modern CPU architectures which are constantly being extended by the hardware manufacturers and due to scalability requirements in the absence of abstractions offered by modern programming languages.

In the first part of this work, we have shown how we achieve architecture independence by using our self-made DSL called GDSL which is geared towards the specification of instruction decoders by offering a special syntax for reading from a byte input stream and matching patterns on it. We have demonstrated the practicality of GDSL by implementing a decoder for one of the most complex CPU architectures on the market, namely Intel x86. An important design goal of the decoder syntax has been the ability to specify decoders in close resemblance to the documentation provided by the manufacturer in order to minimize the chances of bugs and allow for maintainability of the resulting code. Our x86 decoder demonstrates that the design of GDSL indeed allows such an intuitive specification. GDSL is a functional ML-like language that goes beyond its main purpose of offering syntax for instruction decoders. We have demonstrated this by implementing the translation of machine instructions into the RReil intermediate representation and a few simple optimizations for the resulting IR code directly in GDSL. Finally, we have shown how we translate functional GDSL code into naturally looking C code which allows us to debug GDSL code using an existing C debugger. It is an important observation that such a translation is indeed possible for real-world functional code as it drastically reduces the development overhead of a programming language if existing tools can be leveraged. The GDSL toolkit combines our DSL compiler with a few decoder and translator specifications, including the aforementioned x86 decoder and translator. Since the toolkit is open source software, we hope to involve the community in the future in order to provide a fully-fledged toolkit for disassembling and translating arbitrary machine code.

In the second part of this work, we have addressed the problem of achieving a scalable analysis by introducing an analysis algorithm that combines modularity and context-sensitivity in a novel way. A well-known technique to achieve scalability is to analyze each function of a program in isolation and apply a summary of a function at its call sites. However, this can lead to a severe precision loss as the analyzer has no knowledge about the context in which a function is called while analyzing it. Our idea to this end has been to find a reasonable compromise between modularity and scalability by tabulating for certain properties of the calling context which allows the analysis to achieve the required precision. As a result, the analyzer produces a table of function summaries for each function. However, it does not re-analyze a function

for every possible context it is called in, thereby retaining most of the benefits of a modular analysis. As a concrete application, we have implemented an analysis tool that recovers the inter-procedural control flow graph of an executable based on the analysis algorithm presented in this work. Recovering the control flow graph is an important first step towards further analyses that rely on the control flow graph to be known. For the control flow graph recovery, we tabulate for function pointers that are used as call targets. Future work needs to address a second property which is of great relevance here, namely different aliasing relationships between function parameters, and include them in the tabulation scheme. As discussed in our evaluation, however, our implementation already suggests that our approach is suitable to devise scalable and precise analyzers.

List of Figures

1.1. Example code that uses dynamic function binding. 4
1.2. Assembly code for the C++ program in Fig. 1.1 5
1.3. Translated RReil code for the main function in Fig. 1.2. 6
1.4. Software lifecycle from the viewpoint of binary analysis. 7

2.1. The GDSL language grammar without monadic and generic types; we use the well-known POSIX syntax for regular expressions [29]. 22
2.2. Specification for decoding the Intel ADD instruction. 23

3.1. The syntax of our RReil (Relational Reverse Engineering Language) IR. The construct "<u>:</u> <u>*int*</u>" denotes the size in bits whereas "<u>.</u> <u>*int*</u>" in the **var** rule denotes a bit offset. The statements are: assignment, read from address, write to address, conditional, loop (both only used to express the semantics within a native instruction), conditional branch, unconditional branch with a hint of its original purpose, and a primitive "*id*". 25
3.2. The translator function a) and a translation result b) 26
3.3. Translation of the native Intel instructions `cmp eax, ebx; jl tgt` into RReil and applying optimizations. Here, `CForZF`, `SFxorOF`, `SFxorOForZF` are *virtual flags*, that is, translation-specific variables whose value reflect what their names suggest [48]. Note that this example is idealized since the removed flags may not actually be dead. 29

4.1. Overview of the end-to-end GDSL test framework. 35

5.1. A minimal decoder for Intel x86 instructions. 44
5.2. Desugaring the decoders to Core. We omit code handling pattern match failures. 45
5.3. Example Core code which is based on lines 19–22 from Fig. 5.2. 45
5.4. The C code of the decoders (part 1). Some lines and variable declarations are rearranged for presentational purposes. 46
5.5. The C code of the decoders (part 2). Some lines and variable declarations are rearranged for presentational purposes. 47
5.6. The input language Core. 48

5.7. The intermediate language Imp. Note that the non-terminal **Type** is defined in Fig. 5.18. 49
5.8. A Core example program that uses a closure. 50
5.9. The optimized Imp program generated from the code in Fig. 5.8. 50
5.10. Simplified C code for a Core example that uses closures. 51
5.11. Application of substitutions on an Imp AST node. 53
5.12. The non-optimized Imp program generated from the code in Fig. 5.8 using the translation scheme in Fig. 5.13, 5.14, and 5.15. 54
5.13. Translation scheme from Core to Imp (part 1). 56
5.14. Translation scheme from Core to Imp (part 2). 57
5.15. Translation scheme from Core to Imp (part 3). 58
5.16. Unoptimized translation of Fig. 5.3. 61
5.17. Partially optimized code after applying backwards substitution. 64
5.18. Definition of Types, their union and merging of record types. 66
5.19. Typing rules that characterize programs on which unboxing can be applied (part 1). 67
5.20. Typing rules that characterize programs on which unboxing can be applied (part 2). 68
5.21. GDSL program assembly. 70

6.1. The running example C++ program. 79
6.2. The abstract grammar of the analyzed program. $\underline{(}E\underline{)?}$denotes zero or one E. 81
6.3. Abstract Semantics (without Call). 89
6.4. Definition of the $applySummary_{params,\, globals}$ function which applies a summary (last three arguments) to a call site state (first three arguments). . 91
6.5. Definition of *buildRegionMap* which constructs an association between caller and callee memory regions. 94
6.6. Example code that requires a caller region to be added during *buildRegionMap*. 95
6.7. Definition of the *merge* function that merges r_{src} into r_{dst}. 96
6.8. Definition of the *handleCalleeAliasing* function which combines aliasing regions in a summary if possible and aborts the summary application otherwise. 97
6.9. Definition of *handleCallerAliasing* which combines multiple caller regions. 98
6.10. Definition of *expandCallerRegions* which expands folded caller regions and propagates their contents back to the original regions. 99

6.11. Definition of the *addOrRenameFields* function which makes sure that each region and field of the caller output memory structure (first three arguments) exists in the summary input memory structure (second three arguments) and that the respective value domain variables have the same name. 100
6.12. Example code that results in one callee region that maps to multiple caller regions. The summary effect has to be applied to both caller regions. 101
6.13. Example code that results in two callee regions that relate to one caller region. 102
6.14. Creating Herbrand terms for calls to `Check` in Fig. 6.1. 105
6.15. Applying a specialized function summary in $T_H \in \mathbb{T}_{Herb}$ 108

7.1. Example control flow graph. 120
7.2. Simple example for precision loss. 121
7.3. Example statistical output of the analyzer. 123
7.4. Example control flow graph without (left) and with (right) RReil optimizations applied. 124
7.5. Program location comparer. 125
7.6. Fixpoint algorithm. 128
7.7. Example code for a loss of precision without an additional narrowing iteration. 129
7.8. Example assembly (left) and RReil (right) code that results in an access that corresponds to case (2) in Fig. 7.9. 131
7.9. Different access patterns; the accessed part of the region is marked in red. 132

List of Tables

2.1. Two typical instructions in the Intel x86 manual. 11
2.2. Evaluation of different disassembler frameworks. 19

3.1. Evaluating the reduction of the RReil code size due to dead-code optimization. The overall running time is the sum of the translation time plus the time for one of the optimizations. All measurements were obtained on an Intel Core i7 running at 3.40Ghz. 33

4.1. Handling of accesses during RReil code interpretation. 37

5.1. Rules of the Simplifier 62
5.2. Unboxing rules. 69
5.3. Decoding performance depending on the GDSL compiler optimization level. 72
5.4. Decoding memory footprint depending on the GDSL compiler optimization level. 72
5.5. Decoding performance of XED from the Intel Pin toolkit. 73
5.6. GDSL program performance using all optimizations 73

7.1. Evaluation Results 132

Bibliography

[1] A. Sepp, J. Kranz, and A. Simon. "GDSL: A Generic Decoder Specification Language for Interpreting Machine Language." In: *Tools for Automatic Program Analysis*. ENTCS. Deauville, France: Springer, Sept. 2012. URL: https://code.google.com/p/gdsl-toolkit/.

[2] G. Amato et al. "Efficiently intertwining widening and narrowing." In: *Sci. Comput. Program.* 120 (2016), pp. 1–24.

[3] K. Apinis, H. Seidl, and V. Vojdani. "How to Combine Widening and Narrowing for Non-monotonic Systems of Equations." In: *Programming Language Design and Implementation*. Seatle, Washington, USA: ACM, 2013, pp. 377–386.

[4] B. Blanchet et al. "A Static Analyzer for Large Safety-Critical Software." In: *Programming Language Design and Implementation*. File Cabinet.: ACM, June 2003.

[5] S. Bardin et al. "The BINCOA Framework for Binary Code Analysis." In: *Computer Aided Verification*. LNCS. Springer, 2011, pp. 165–170.

[6] *BeaEngine*. http://www.beaengine.org. Version 4.1 rev 172. 2012. URL: http://www.beaengine.org.

[7] L. Bettini. "Implementing Java-like languages in Xtext with Xsemantics." In: *Symposium on Applied Computing*. Ed. by S. Y. Shin and J. C. Maldonado. Coimbra, Portugal: ACM, Mar. 2013, pp. 1559–1564.

[8] N. S. Bjørner. "Minimal Typing Derivations." In: *Workshop on ML and its Applications*. ACM, 1994, pp. 120–126.

[9] J. Brauer and A. King. "Automatic Abstraction for Intervals using Boolean Formulae." In: *Static Analysis Symposium*. Ed. by R. Cousot and M. Martel. Vol. 6337. LNCS. Springer, Sept. 2010, pp. 182–196.

[10] L. Cardelli and P. Wegner. "On understanding types, data abstraction, and polymorphism." In: *ACM Computing Surveys* 17.4 (1985), pp. 471–522.

[11] C. Cifuentes and S. Sendall. "Specifying the Semantics of Machine Instructions." In: *International Workshop on Program Comprehension*. IWPC '98. Washington: IEEE Computer Society, 1998.

[12] Intel Corp. *Pin - A Dynamic Binary Instrumentation Tool*. http://www.pintool.org. 2012. URL: http://www.pintool.org.

[13] P. Cousot and R. Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints." In: *Principles of Programming Languages*. Los Angeles, California, USA: ACM, Jan. 1977, pp. 238–252.

[14] P. Cousot and R. Cousot. "Modular Static Program Analysis." In: *Compiler Construction*. Ed. by R. N. Horspool. invited paper. file cabinet: Springer, Apr. 2002, pp. 159–178.

[15] P. Cousot and R. Cousot. "Static Determination of Dynamic Properties of Programs." In: *International Symposium on Programming*. Ed. by B. Robinet. File Cabinet, Apr. 1976, pp. 106–130.

[16] P. Cousot et al. "Automatic Inference of Necessary Preconditions." In: *Verification, Model Checking, and Abstract Interpretation: 14th International Conference*. Rome, Italy: Springer Berlin Heidelberg, 2013, pp. 128–148. ISBN: 978-3-642-35873-9.

[17] A. Das et al. "Angelic Verification: Precise Verification Modulo Unknowns." In: *Computer Aided Verification: 27th International Conference, CAV 2015*. San Francisco, CA, USA: Springer International Publishing, 2015, pp. 324–342. ISBN: 978-3-319-21690-4.

[18] *distorm*. http://www.ragestorm.net/distorm/. Version 3.1. 2012. URL: http://www.ragestorm.net/distorm/.

[19] T. Dullien and S. Porst. *REIL: A platform-independent intermediate representation of disassembled code for static code analysis*. CanSecWest Vancouver, Canada. 2009. URL: http://www.zynamics.com/downloads/csw09.pdf.

[20] C. Elliott, S. Finne, and O. de Moor. "Compiling Embedded Languages." In: *Journal of Functional Programming* 13.2 (2003).

[21] M. Erwig and E. Walkingshaw. "Semantics-Driven DSL Design." In: *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*. Ed. by M. Mernik. IGI Global, Oct. 2013. Chap. 3, pp. 56–80. DOI: 10.4018/978-1-4666-2092-6.

[22] M. Fowler. *Domain Specific Languages*. 1st. Addison-Wesley Professional, 2010. ISBN: 978-0-321-71294-3.

[23] A. Fox and M. O. Myreen. "A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture." In: *Interactive Theorem Proving*. Vol. 6172. LNCS. Edinburgh, UK: Springer, 2010, pp. 243–258.

[24] M. Frigo. "A fast Fourier transform compiler." In: *Programming Language Design and Implementation*. Atlanta, Georgia, USA: ACM, 1999, pp. 169–180. ISBN: 1-58113-094-5. DOI: 10.1145/301618.301661.

[25] R. Giacobazzi, F. Ranzato, and F. Scozzari. "Making Abstract Domains Condensing." In: *Trans. Comput. Log.* 6.1 (2005), pp. 33–60.

[26] R. Giacobazzi and F. Scozzari. "A Logical Model for Relational Abstract Domains." In: *Transactions on Programming Languages and Systems* 20.5 (1998), pp. 1067–1109. ISSN: 0164-0925.

[27] A. Gurfinkel et al. "The SeaHorn Verification Framework." In: *Computer Aided Verification*. San Francisco, California, USA: Springer, July 2015, pp. 343–361.

[28] Hex-Rays. *IDA Pro Disassembler*. http://www.hex-rays.com/idapro. Version 6.0.101001. 2012. URL: http://www.hex-rays.com/idapro.

[29] IEEE and The Open Group. *The Open Group Base Specifications*. Issue 7. 2018. URL: http://pubs.opengroup.org/onlinepubs/9699919799/.

[30] R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. CRC Applied Algorithms and Data Structures. Chapman & Hall, Aug. 2012. ISBN: 978-1-420-08279-1.

[31] A. Kennedy. "Compiling with Continuations, Continued." In: *International Conference on Functional Programming*. Freiburg, Germany: ACM, 2007, pp. 177–190. DOI: 10.1145/1291151.1291179.

[32] J. Kranz. "Validating the Decoding and the Translation into Value Semantics of X86 Machine Code." Master's Thesis. Technical University of Munich, 2013. URL: https://mediatum.ub.tum.de/1470553.

[33] J. Kranz, A. Sepp, and A. Simon. "GDSL: A Universal Toolkit for Giving Semantics to Machine Language." In: *Asian Symposium on Programming Languages and Systems*. Ed. by C. Shan. Melbourne, Australia: Springer, Dec. 2013.

[34] J. Kranz and A. Simon. "Structure-Preserving Compilation: Efficient Integration of Functional DSLs into Legacy Systems." In: *Principles and Practice of Declarative Programming*. ACM, Sept. 2014.

[35] J. Kranz et al. *IR Preprocessing for Deep Binary Analysis*. Tech. rep. Lehrstuhl für Sprachen und Beschreibungsstrukturen in der Informatik (Prof. Seidl), 2016. URL: https://mediatum.ub.tum.de/1470488.

[36] Julian Kranz and Axel Simon. "Modular Analysis of Executables Using On-Demand Heyting Completion." In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Isil Dillig and Jens Palsberg. Cham: Springer International Publishing, 2018, pp. 291–312. ISBN: 978-3-319-73721-8.

[37] X. Leroy. "Unboxed objects and polymorphic typing." In: *Principles of Programming Languages*. ACM Press, 1992, pp. 177–188.

[38] *libopcodes*. http://packages.debian.org/testing. Package binutils-dev-2.22-6. 2012. URL: http://packages.debian.org/testing.

[39] J. Lim and T. Reps. "A System for Generating Static Analyzers for Machine Instructions." In: *Compiler Construction*. Ed. by L. Hendren. Vol. 4959. LNCS. Springer, 2008, pp. 36–52. DOI: 10.1007/978-3-540-78791-4_3. URL: http://dx.doi.org/10.1007/978-3-540-78791-4_3.

[40] K. Marriott and H. Søndergaard. "Precise and Efficient Groundness Analysis for Logic Programs." In: *ACM Lett. Program. Lang. Syst.* 2 (1-4 Mar. 1993), pp. 181–196.

[41] *metasm*. http://metasm.cr0.org/. Retrieved on 2012/05/25. 2012. URL: http://metasm.cr0.org/.

[42] M. Müller-Olm and H. Seidl. "Precise Interprocedural Analysis through Linear Algebra." In: *Principles of Programming Languages*. Venice, Italy: ACM, Jan. 2004, pp. 330–341.

[43] P. Cousot and R. Cousot. "Systematic Design of Program Analysis Frameworks." In: *Principles of Programming Languages*. San Antonio, Texas, USA: ACM, Jan. 1979, pp. 269–282.

[44] Simon Peyton Jones. *Haskell 98 Language and Libraries: the Revised Report*. Cambridge University Press, 2003.

[45] N. Ramsey and M. F. Fernández. "Specifying Representations of Machine Instructions." In: *Trans. of Programming Languages and Systems* 19.3 (May 1997), pp. 492–524.

[46] T. Reps, S. Horwitz, and M. Sagiv. "Precise Interprocedural Dataflow Analysis via Graph Reachability." In: *Principles of Programming Languages*. San Francisco, California, USA: ACM, 1995, pp. 49–61. DOI: 10.1145/199448.199462.

[47] Helmut Seidl, Reinhard Wilhelm, and Sebastian Hack. *Compiler Design - Analysis and Transformation*. Springer, 2012. ISBN: 978-3-642-17547-3. DOI: 10.1007/978-3-642-17548-0. URL: http://dx.doi.org/10.1007/978-3-642-17548-0.

[48] A. Sepp, B. Mihaila, and A. Simon. "Precise Static Analysis of Binaries by Extracting Relational Information." In: *Working Conference on Reverse Engineering*. Ed. by M.Pinzger and D. Poshyvanyk. Limerick, Ireland: IEEE, Oct. 2011.

[49] Z. Shao and A. W. Appel. "A Type-based Compiler for Standard ML." In: *Programming Language Design and Implementation*. La Jolla, California, USA, June 1995, pp. 116–129.

[50] M. Sharir and A. Pnueli. "Two Approaches to Interprocedural Data Flow Analysis." In: *Program Flow Analysis: Theory and Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1981. Chap. 7, pp. 189–234.

[51] O. Shivers. "Control-Flow Analysis of Higher-Order Languages." PhD thesis. Carnegie Mellon University: School of Computer Science, May 1991.

[52] H. Siegel and A. Simon. "FESA: Fold- and Expand-based Shape Analysis." In: *Compiler Construction*. Vol. 7791. LNCS. Rome, Italy: Springer, Mar. 2013, pp. 82–101.

[53] H. Siegel and A. Simon. "Summarized Dimensions Revisited." In: *Workshop on Numeric and Symbolic Abstract Domains*. Ed. by L. Mauborgne. ENTCS. Venice, Italy: Springer, Sept. 2011.

[54] A. Simon. "Deriving a Complete Type Inference for Hindley-Milner and Vector Sizes using Expansion." In: *Science of Computer Programming* 95, Part 2.0 (2014), pp. 254–271.

[55] A. Simon and J. Kranz. "The GDSL toolkit: Generating Frontends for the Analysis of Machine Code." In: *Program Protection and Reverse Engineering Workshop*. San Diego, California, USA: ACM, Jan. 2014.

[56] K. Slind and M. Norrish. "A Brief Overview of HOL4." In: *International Conference on Theorem Proving in Higher Order Logics*. LNCS. Springer, 2008, pp. 28–32.

[57] A. Thakur and T. Reps. "A Method for Symbolic Computation of Abstract Operations." In: *Computer Aided Verification*. LNCS. Berkeley, CA: Springer, 2012, pp. 174–192.

[58] *The Rust Programming Language: Validating References with Lifetimes*. 2018. URL: `https://doc.rust-lang.org/book/second-edition/ch10-03-lifetime-syntax.html`.

[59] P. J. Thiemann. "Unboxed Values and Polymorphic Typing Revisited." In: *International Conference on Functional Programming Languages and Computer Architecture*. FPCA. La Jolla, California, USA: ACM, 1995, pp. 24–35.

[60] *udis86*. `http://udis86.sourceforge.net`. Version 1.7. 2012. URL: `http://udis86.sourceforge.net`.

[61] A. Venet. "Abstract Cofibered Domains: Application to the Alias Analysis of Untyped Programs." In: *Static Analysis Symposium*. LNCS. London, UK: Springer, 1996, pp. 366–382.

[62] Xi Wang et al. "A Differential Approach to Undefined Behavior Detection." In: *Commun. ACM* 59.3 (Feb. 2016), pp. 99–106. ISSN: 0001-0782. DOI: `10.1145/2885256`. URL: `http://doi.acm.org/10.1145/2885256`.

[63] *xed2*. `http://www.pintool.org`. Version 2.13. 2013. URL: `http://www.pintool.org`.

[64] Z. Xu, T. Reps, and B. Miller. "Typestate Checking of Machine Code." In: *Programming Languages and Systems: 10th European Symposium on Programming, ESOP 2001*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 335–351. ISBN: 978-3-540-45309-3.