Robbins, Ed (2017) *Solvers for Type Recovery and Decompilation of Binaries.* Doctor of Philosophy (PhD) thesis, University of Kent,.

SOLVERS FOR TYPE RECOVERY AND DECOMPILATION OF BINARIES

A THESIS SUBMITTED TO
THE UNIVERSITY OF KENT
IN THE SUBJECT OF COMPUTER SCIENCE
FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY.

By
Ed Robbins
January 2017

Abstract

Reconstructing the meaning of a program from its binary is known as reverse engineering. Since reverse engineering is ultimately a search for meaning, there is growing interest in inferring a type (a meaning) for the elements of a binary in a consistent way. Currently there is no consensus on how best to achieve this, with the few existing approaches utilising ad-hoc techniques which lack any formal basis. Moreover, previous work does not answer (or even ask) the fundamental question of what it means for recovered types to be correct.

This thesis demonstrates how solvers for Satisfiability Modulo Theories (SMT) and Constraint Handling Rules (CHR) can be leveraged to solve the type reconstruction problem. In particular, an approach based on a new SMT theory of rational tree constraints is developed and evaluated. The resulting solver, based on the reification mechanisms of Prolog, is shown to scale well, and leads to a reification driven SMT framework that supports rapid implementation of SMT solvers for different theories in just a few hundred lines of code.

The question of how to guarantee semantic relevance for reconstructed types is answered with a new and semantically-founded approach that provides strong guarantees for the reconstructed types. Key to this approach is the derivation of a witness program in a type safe high-level language alongside the reconstructed types. This witness has the same semantics as the binary, is type correct by construction, and it induces a (justifiable) type assignment on the binary. Moreover, the approach, implemented using CHR, yields a type-directed decompiler.

Finally, to evaluate the flexibility of reificiation-based SMT solving, the SMT framework is instantiated with theories of general linear inequalities, integer difference problems and octagons. The integer difference solver is shown to perform

competitively with state-of-the-art SMT solvers. Two new algorithms for incremental closure of the octagonal domain are presented and proven correct. These are shown to be both conceptually simple, and offer improved performance over existing algorithms. Although not directly related to reverse engineering, these results follow from the work on SMT solver construction.

Acknowledgements

Copyright

- The work presented in Chapters 2 and 3 is derived from work by Ed Robbins, Jacob Howe and Andy King, published in Elsevier [108] and ACM [109].
- The work presented in Chapter 4 is derived from work by Ed Robbins, Tom Schrijvers and Andy King, published in ACM [110].
- The work presented in Chapter 5 is derived from work by Aziem Chawdhary, Ed Robbins and Andy King, published in Springer [23].
- Ed Robbins was the first/lead author of [108, 109] and [110]. Aziem Chawdhary and Ed Robbins were joint first authors of [23].
- All other text copyright Ed Robbins.

Personal Acknowledgements

I'd like to thank my supervisor Andy King, for his good sense of humour, encouragement, technical brilliance, patience, for taking me on as his student in the first place, and for giving me a kick when necessary. I don't think there could be a better supervisor, and there's no one I'd rather have stood next to me when staring at a hard problem on a white board.

I couldn't have done this without the support of my wife, Sam, who has been brilliant, supportive, light-hearted and laid back throughout. Our son, Bertie, who is probably the most smiley person in the world, was born during my studies, and should be thanked for giving me the best reason in the world to complete them.

My family provided me with the encouragement to get here, and I'd particularly like to thank my parents for their emotional support and belief that I could do it (particularly my Mum, Susie). My Dad, Steve, provided the original inspiration and encouragement for me to become an engineer from a young age, and that has led me to getting this far.

Bill May, a brilliant teacher who taught me electronics A-level, was also a great source of inspiration, and kindly gave up many of his lunch hours to help me debug my circuits and programs, and to help me in designing whatever thing had caught my attention at the time. It didn't really occur to me that this might have been an inconvenience until after I completed my A-levels, but I don't actually think he thought of it as one. Thanks Bill!

Alan Mycroft provided the original inspiration for my thesis topic, and was helpful on several occasions in discussing ideas and inviting me to visit. In many ways this work is a continuation of his seminal work on the topic.

I owe a great deal to my ~~co-conspirators~~ collaborators, with whom I am priviledged to have worked: Jacob Howe, who formed the constraint solving core of the team on the SMT work, Tom Schrijvers, whose brilliance allowed the decompilation work to flower, and Aziem Chawdhary whose insight enabled the development of the octagon work.

I'd like to thank the other occupants of my office space, "the unix zoo", for their open attitude, helpfulness, and for the awesome atmosphere and culture that they engendered in the room. Particularly Martin Ellis (Mex), with whom I worked very closely for a time, and Edd Barrett, who are the greatest guys to hack with, and also Tom Schilling, who always knew something useful about whatever problem someone might be stuck on.

Finally I'd like to thank all my other friends and colleagues, who have been helpful and/or supportive throughout.

Contents

Abstract **ii**

Acknowledgements **iv**

Contents **vi**

List of Tables **ix**

List of Figures **x**

1 Introduction **1**

1.1 Type reconstruction 4

1.1.1 SMT solving with lazy-basic 6

1.1.2 SMT solving with theory propagation 8

1.2 Type reconstruction as decompilation 10

1.3 Decision procedures for the octagon abstract domain 13

2 Theory Propagation and Reification **17**

2.1 Illustrative example 17

2.2 SMT and Theory Propagation 21

2.2.1 SAT solving and unit propagation 21

2.2.2 SMT solving, the lazy-basic approach 23

2.2.3 SMT, the DPLL(T) approach 24

2.3 Theory Propagation and Reification 26

2.3.1 Theory propagation 26

2.3.2 Labelling strategies 28

2.3.3 Calculating an unsatisfiable core 29
2.4 Summary 33

3 Implementing SMT with Reification 34
3.1 Instantiation for Rational-Trees 35
3.2 Instantiation for Linear Real Arithmetic 36
3.3 Instantiation for Difference Logic 38
3.3.1 The Floyd-Warshall algorithm 39
3.3.2 Theory propagation in difference logic 41
3.4 Experimental Results 45
3.4.1 Rational-tree solver 45
3.4.2 Integer difference logic solver 48
3.5 Discussion 51
3.5.1 Rational-tree solver 51
3.5.2 Integer difference logic solver 53
3.6 Summary 54

4 Semantics-Driven Decompilation of Recursive Datatypes 55
4.1 System overview 56
4.2 The MinX Language 57
4.2.1 Memory 57
4.2.2 Syntax 58
4.2.3 Structured Operational Semantics 60
4.3 The MinC Language 62
4.3.1 Syntax 62
4.3.2 Type System 64
4.3.3 Semantics 65
4.3.4 Type Safety 68
4.4 Decompilation Relation 70
4.4.1 Meta-Theoretical Properties 71
4.5 Implementation 79
4.6 Evaluation 81
4.7 Summary 83

5 Simple and Efficient Algorithms for Octagons **85**
5.1 Preliminaries 86
5.1.1 The Octagon Domain and its Representation 86
5.1.2 Definitions of Closure 88
5.1.3 High-level Overview 90
5.2 Incremental Closure 91
5.2.1 Classical Incremental Closure 92
5.2.2 Improved Incremental Closure 93
5.3 Simpler Proofs for Classical Strong Closure 99
5.3.1 Classical Strong Closure 99
5.4 Experiments 100
5.5 Discussion 101
5.6 Summary 104

6 Related work **105**
6.1 Type recovery 105
6.2 Satisfiability Modulo Theories 108
6.3 The Octagon domain 109

7 Conclusions **111**
7.1 Reflection upon Chapters 2 and 3 111
7.2 Reflection upon Chapter 4 113
7.3 Reflection upon Chapter 5 113
7.4 Closing remark 114

A Proof Appendix **115**
A.1 Semantics-Driven Decompilation 115
A.1.1 Type Safety 115
A.1.2 Well-Typed Decompilation 125
A.1.3 Semantics Preservation 129
A.2 Simple and Efficient Algorithms for Octagons 144

Bibliography **150**

List of Tables

1 Generated constraints 20
2 Benchmarking for a selection of type recovery problems 47
3 Benchmarking for a selection of QF_IDL problems 49
4 Erroneous constraint generation 52
5 Solutions and Recovered Structures 82

List of Figures

1 Prolog SAT solver using watched literals [63] . . . 22
2 Recursive formulation of the DPLL(T) algorithm . . . 25
3 Interface to the DPLL(T) solver . . . 27
4 Finding an unsatisfiable core . . . 30
5 Theory propagation for rational-tree constraints . . . 36
6 Theory propagation for linear real arithmetic . . . 37
7 Illustrating Floyd-Warshall . . . 39
8 Floyd-Warshall algorithm: non-incremental and incremental versions 40
9 Setting up the Floyd-Warshall matrix and the watch matrix . . . 42
10 Propagating from the SAT solver to the theory solver . . . 43
11 Propagating from the theory solver to the SAT solver . . . 44
12 System overview . . . 56
13 Structured Operational Semantics of MinX blocks (b) . . . 59
14 Structured Operational Semantics of MinX instructions (ι) . . . 61
15 Well-formed type declarations of MinC programs . . . 63
16 Type-correct MinC programs . . . 64
17 Subtyping relations of MinC programs . . . 65
18 Structured Operational Semantics of MinC programs . . . 67
19 Well-typed addresses and values . . . 69
20 Decompilation of mov instructions . . . 72
21 Decompilation of $\mathsf{op}^{\oplus}$, $\mathsf{op}^{\otimes}$, alloc and call instructions . . . 73
22 Decompilation of basic blocks and function definitions . . . 74
23 Value correspondence . . . 75
24 The MinX to MinC toolchain . . . 80
25 Iterative summation of a linked list in MinX (left) and MinC (right) 81

26 Example of an octagonal system and its DBM representation 87
27 Intuition behind strong closure: Two closed graphs representing the same octagon: $x \leqslant 2 \wedge y \leqslant 4$ 90
28 Non-incremental closure and strengthening 91
29 Minè's Incremental Closure Algorithm 92
30 Four ways to reduce the distance between x'_i and x'_j 93
31 Incremental Closure 94
32 Incremental Closure with code hoisting 95
33 Before and after adding $x_0 - x_1 \leqslant 0$ 96
34 Experiments with strong closure algorithms on 40, 50 and 60 variables 102
35 Experiments with strong closure algorithms on 100 and 120 variables 103
36 Experiments with strong closure algorithms on 120 constraints . . . 104

Chapter 1

Introduction

The author of a high-level language program may utilise a wide range of abstractions for describing their algorithm: variables, compound arithmetic expressions, loops, functions, high-level types (arrays, structures, lists, maps etc), object hierarchies, threads and so on. But during compilation programs are converted into low-level operations on registers and memory, and all these carefully crafted abstractions are lost. This is unfortunate. Consider taking a poem and summarising it in shorthand: Semantically the actions and descriptions of the poem can be entirely reproduced from the summary. Yet all rhyme and rhythm, the nuances of meaning, emotion and word plays, sentence structure and even visual cues like the form of the text on the page, are lost. Similarly, in compilation programs are transformed from elegantly engineered operational or declarative descriptions into a series of opaque instructions. Reverse engineering (reversing) attempts to make good this loss and uncover the operation and intent of a compiled program, either by abstracting over the machine code to recover the original program features, or by identifying entirely new traits that expose program functionality.

The history of software reversing has roots in the history of programming language development itself: The earliest reverse engineering tools were disassemblers developed to debug the first assembly language programs. A recurring application is the reuse of legacy software, initially during the transition from second generation (transistor based) to third generation (integrated circuit based) computers in the mid 1960's to early 1970's [53]. Though the earliest high level languages (such as ALGOL, COBOL and Fortran) were developed on second generation machines, most of the programs of the era were written in assembler, and thus not portable. As second generation machines were rapidly replaced by third generation installations, there was a widespread desire to reuse existing programs, which resulted in the first research into decompilation [44, 52, 53, 58, 60]. In the 1980's a prolific use of reversing was for breaking software protection to distribute software (games, mostly) illegally [107]. Compaq famously reverse engineered the IBM PC BIOS (Basic Input/Output System) to produce the first IBM PC clones in 1982 [20]. The theme of reversing proprietary systems has continued with the reverse engineering of hardware drivers and software protocols to produce equivalents for open source operating systems [25, 32, 48, 117].

However, reverse engineering remained mostly a niche activity until relatively recently, when its most important applications became apparent; those relating to software security. The last two decades have seen numerous exposures of serious security vulnerabilities in widely used software (see, for example [40, 95, 97]), and the advent of a global computer fraud and malware industry that is estimated to cost the global economy hundreds of billions of dollars a year [67]. Anti-virus companies use reversing to uncover the operation of malware, while government, defense and security agencies reverse Commercial-off-the-Shelf (COTS) software in search of security vulnerabilities. Governments are also thought to utilise reverse engineering to engage in more disreputable activities, such as breaking into the nuclear facilities of other sovereign nations [88].

Due to the niche it occupies, advances in software reversing have trailed those in programming languages, compilers, and the complexity of programs themselves. As a result modern binaries present many challenges to reversing. Unfortunately, there is a lack of well developed tools to meet these challenges [116], and the issue is exacerbated by the secrecy within which the groups most actively engaged in binary reversing operate: Relatively few studies [6, 116] have examined the workflow and processes used by reverse engineers themselves, which has made it difficult for industry to develop commercial software targeted at reversing.

The industry standard tool is Interactive Disassembler Pro (IDA Pro, or simply IDA), a disassembler and debugger targeting a wide range of binary formats and processor architectures [57, 116]. IDA has gained popularity as the best available tool for the job, but has significant limitations, mostly due to its reliance on pattern matching and heuristics to recognise the code constructs emitted by specific compilers. When confronted with a change to the way a compiler emits, for example, C switch statements, IDA does not recognise the new pattern and cannot continue [76]. When faced with intentionally obfuscated code (from malware, and with increasing regularity, standard commercial software), IDA becomes almost as much of a hindrance as a grace. It is no surprise that reversing binaries with IDA generally takes weeks or months of effort [116].

Given the limited commercial attention to software reversing, it is somewhat astonishing that the topic receives so much attention from academia. There are several well established conferences (e.g. [15, 39, 47, 80, 89], though WCRE, the Working Conference on Reverse Engineering, recently folded) and the field, while still playing catch up, is reasonably mature. Some disparity between academia and industry is not unusual, but it is particularly marked in this case, in part because academics are not typically able to communicate directly with reverse engineers, and because it is not clear to industry which academic advances, that are broadly speaking only ever prototyped, might be developed into fully fledged tools. Studies into the work practices of reverse engineers have stated scalability and reliability of automated tools as problems [116]. Tools that do not scale to binaries beyond a few megabytes in size are of limited use, as are those that cannot deal with irregularities introduced by unusual compiler constructs, or obfuscated malware code. This precludes the vast majority of academic implementations, which naturally seek to demonstrate and evaluate science first, and focus on engineering second.

Yet, academia has developed techniques to meet challenges such as Control Flow Graph (CFG) recovery [26] and type reconstruction [100], and tailored abstract interpreters to compute useful approximations of possible machine code values [35] (topics that will be covered here in due course). In addition, the last decade has seen the rise of powerful boolean satisfiability (SAT), Satisfiability Modulo Theory (SMT) [34] and Constraint Handling Rules (CHR) [45] solvers, that are able to solve a range of computationally hard problems. This thesis makes the case that these solvers are perfectly suited to building the next generation of principled, accurate and scalable reverse engineering tools, and that, in particular, the open problems of automated type reconstruction and decompilation are fully achievable.

1.1 Type reconstruction

Reversing amounts to searching for a high-level meaning that is consistent across a binary. Typing, likewise, checks the elements of a high-level program combine in a consistent, meaningful way. Types themselves can expose the semantics of a program, yielding a powerful abstraction for the reverse engineer [81]. Types can also guide test-generation in fuzzing [114], help locate information in a core dump in memory-based forensics [22], and support program reconstruction [37, 100].

Machine code is untyped, as all program data is stored in all-encompassing registers and memory that must be able to hold data of any type. Registers are of limited size (generally 64 bits or less) and are therefore normally only used to store scalar values. Composite types such as arrays, structs, and arbitrary data structures (objects, lists, trees etc) must therefore be stored in memory (stack or heap) and accessed via pointers. Individual array entries, struct fields etc are stored contiguously in memory, though they may be separated by redundant padding, to word-align each element. The object is then referenced by a pointer to its first element. Since low-level machine code instructions can only operate directly on scalar values, high-level language operations on composite types must be broken down to access individual elements of the composite object. This is done either by modifying the pointer to the object directly (for example by iterating it to point at subsequent array entries) or by using indexed addressing modes. Consider for example, the following C linked list type, and code for iteratively summing the elements in such a list:

```c
struct list {
  int value;
  struct list *next_node;
};

int
iterative_sum(struct list *list_node) {
  int sum = 0;
  while (list_node != NULL) {
    sum += list_node->value;
    list_node = list_node->next_node;
```

```
  }
  return sum;
}
```

The equivalent x86 code, given below, accesses the `value` element of the list node by dereferencing the pointer stored in `edx`, and the `next_node` element using indexed address arithmetic that adds four bytes to the same pointer and then dereferences the result, i.e. `[edx+0x4]`. The pointer to `list_node` itself is passed to the function on the stack at `[esp+0x4]`.

```
      mov edx, [esp+0x4]
      mov eax, 0x0
loop: test edx, edx
      jz end
      add eax, [edx]
      mov edx, [edx+0x4]
      jmp loop
end:  ret
```

The seminal work on type reconstruction was by Mycroft [100], who observed that although machine instructions are essentially untyped, a given instruction has a finite number of possible typings with respect to a specific high-level type system. Mycroft formulated each possible typing as a constraint over type variables, and gave the typing for an instruction as a disjunction of those constraints. To illustrate consider the instruction `mov eax, 0x0`, that moves the literal zero into a register, and can be typed in two possible ways (with respect to a C-like type system). The first possibility is that `eax` is a four byte integer being set to zero, the second that it is a pointer of some sort being initialised with `NULL`. In this case, the constraint $T_{\texttt{eax}} = \texttt{int32} \vee T_{\texttt{eax}} = \mathit{ptr}(\alpha)$ might be generated to express this, where `int32` indicates a 32 bit integer, and $\mathit{ptr}(\alpha)$ a pointer to some unknown type α.

Mycroft showed that a solution to a set of such constraints would type a program fragment through type unification [91], and that, by using circular unification [85], recursive types (such as the linked list given in the above example) could also be recovered. It was also shown that the disjunctive nature of the constraints can in general lead to multiple solutions. However, beyond a description of his modified unification algorithm, Mycroft did not provide a method to actually solve such constraint systems.

The first body of work in this thesis is motivated by the problem of solving type constraints similar to those demonstrated by Mycroft, in order to type machine code automatically. It is shown that this can be achieved by formulating an SMT instance over the theory of rational-trees [87] (which is the term commonly used for circular unification in the context of constraint solving). Satisfiability Modulo Theories (SMT) [103] describes a class of problems that consist of formulae in a given first-order theory, composed with Propositional connectives. The first-order theory of rational-trees is not currently supported in any off-the-shelf SMT solver, but efficient rational-tree unification [68] is integral to many Prolog systems. Prolog is therefore a natural choice for implementing such a solver as the theory part of the solver is provided essentially for free.

1.1.1 SMT solving with lazy-basic

One straightforward approach to SMT solving is to apply the so-called lazy-basic technique which decouples Boolean satisfiability (SAT) solving [34] from theory solving. To illustrate, consider the SMT formula $f = (x \leqslant -1 \vee -x \leqslant -1) \wedge (y \leqslant -1 \vee -y \leqslant -1)$ and the SAT formula $g = (p \vee q) \wedge (r \vee s)$ that corresponds to its Propositional skeleton. In the skeleton, the Propositional variables p, q, r and s, respectively, indicate whether the theory constraints $(x \leqslant -1)$, $(-x \leqslant -1)$, $(y \leqslant -1)$ and $(-y \leqslant -1)$ hold. In this approach, a model is found for $(p \vee q) \wedge (r \vee s)$, for instance, $\{p \mapsto \mathit{true}, q \mapsto \mathit{true}, r \mapsto \mathit{true}, s \mapsto \mathit{false}\}$. Then, from the model, a conjunction of theory constraints $(x \leqslant -1) \wedge (-x \leqslant -1) \wedge (y \leqslant -1) \wedge \neg(-y \leqslant -1)$ is constructed, with the polarity of the constraints reflecting the truth assignment. This conjunction is then tested for satisfiability in the theory component. In this case it is unsatisfiable, which triggers a diagnostic stage. This amounts to finding a conjunct, in this case $(x \leqslant -1) \wedge (-x \leqslant -1)$, which is also unsatisfiable, that identifies a source of the inconsistency. From this conjunct, a blocking clause $(\neg p \vee \neg q)$ is added to g to give g' which ensures that conflict between the theory constraints is never encountered again. Then, solving the augmented Propositional formula g' might, for example, yield the model $\{p \mapsto \mathit{false}, q \mapsto \mathit{true}, r \mapsto \mathit{true}, s \mapsto \mathit{true}\}$, from which a second clause $(\neg r \vee \neg s)$ is added to g'. Any model subsequently found, for instance, $\{p \mapsto \mathit{false}, q \mapsto \mathit{true}, r \mapsto \mathit{true}, s \mapsto \mathit{false}\}$, will give a conjunction that is satisfiable in the theory component, thereby solving the SMT problem.

The lazy-basic approach is particularly attractive when combining an existing SAT solver with an existing decision procedure, for instance, a solver provided by a constraint library. By using a foreign language interface a SAT solver can be invoked from Prolog [27] and a constraint library can be used to check satisfiability of the conjunction of theory constraints. A layer of code can then be added to diagnose the source of any inconsistency. This provides a simple way to construct an SMT solver that compares very favourably with the coding effort required to integrate a new theory into an existing open source SMT solver. The latter is normally a major undertaking and often can only be achieved in conjunction with the expert who is responsible for maintaining the solver. Thus, although one might expect implementing a new theory to be merely an engineering task, it is actually far from straightforward.

Prolog has rich support for implementing decision procedures for theories, for instance, attributed variables [56, 59]. (Attributed variables provide an interface between Prolog and a constraint solver by permitting logical variables to be associated with state, for instance, the range of values that a variable can possibly assume.) Several theories come prepackaged with many Prolog systems. This raises the questions of how to best integrate a theory solver with a SAT solver, and how powerful an SMT solver written in a declarative language can actually be. This motivates further study of the coupling between the theory and the Propositional component of the SAT solver which goes beyond the lazy-basic approach, to the roots of logic programming itself.

The equation Algorithm = Logic + Control [78] expresses the idea that in logic programming algorithm design can be decoupled into two separate steps: specifying the logic of the problem, classically as Horn clauses, and orchestrating control of the sub-goals. The problem of satisfying a SAT formula is conceptually one of synchronising activity between a collection of processes where each process checks the satisfiability of a single clause. Therefore it is perhaps no surprise that control primitives such as delay declarations [102] can be used to succinctly specify the watched literal technique [98]. In this technique, a process is set up to monitor two variables of each clause. To illustrate, consider the clause $(x \vee y \vee \neg z)$. The process for this clause will suspend on two of its variables, say x and y, until one of them is bound to a truth-value. Suppose x is bound. If x is bound to *true* then the clause is satisfied, and the process terminates; if x is bound to *false*, then the process suspends until either y or z is bound. Suppose z is subsequently bound, either by another process or by labelling. If z is *true* then y is bound to *true* since otherwise the clause is not satisfied; if z is *false* then the clause is satisfied and the process closes down without inferring any value for y. Note that in these steps the process only waits on two variables at any one time. Unit propagation is at the heart of SAT solving and when implemented by watched literals combined with backtracking, the resulting solver is efficient enough to solve some non-trivial Propositional formulae [61, 62, 64]. In addition to issues of performance the correctness of this approach has been examined [38]. To summarise, Prolog not only provides constraint libraries, but also the facility to implement a succinct SAT solver [64]. The resulting solver can be regarded as a glass box, as opposed to a black one, which allows a solver to be extended to support, among other things, new theories and theory propagation.

1.1.2 SMT solving with theory propagation

The lazy-basic approach to SMT alternates between SAT solving and checking whether a conjunction of theory constraints is satisfiable which, though having conceptual and implementation advantages, is potentially inefficient. With a glass box solver it is possible to refine this interaction by applying theory propagation. In theory propagation, the SAT solving and theory checking are interleaved. The solver not only checks the satisfiability of a conjunction of theory constraints, but decides whether a conjunction of some constraints entails or disentails others. Returning to the earlier example, observe that $(x \leqslant -1) \wedge (-x \leqslant -1)$ is unsatisfiable, hence for the partial assignment $\{p \mapsto \mathit{true}\}$ it follows that $(x \leqslant -1)$ holds in the theory component, therefore $(-x \leqslant -1)$ is disentailed and the assignment can be extended to $\{p \mapsto \mathit{true}, q \mapsto \mathit{false}\}$. Theory propagation is essentially the coordination problem of scheduling unit propagation with the simultaneous checking of whether theory constraints are entailed or disentailed.

Chapter 2 presents an SMT solving framework that utilises reification in Prolog to synchronise theory and unit propagation, and in so doing enables development of SMT solvers that are both efficient and elegant. Reification is a constraint handling mechanism in which a constraint is augmented with a Boolean variable that indicates whether the constraint is entailed (implied by the store) or disentailed (is inconsistent with the store). This enables close coupling of unit and theory propagation, such that the SAT and theory solvers drive each other towards a solution instead of being led purely by SAT search heuristics as in the lazy-basic approach [46, 103]. Together Chapters 2 and 3 build on this mechanism to offer an expanded and revised version of the work presented in [108] and [109], and make the following contributions:

- A framework for using reification as a mechanism to realise theory propagation is presented. The idea is simple in hindsight and can be realised straightforwardly in Prolog. The clarity and brevity of the code contrasts with the investment required to integrate a theory into an existing open source SMT solver.
- This framework is illustrated for three theories:
  - The first theory is that of rational-trees, where the control is provided by block and when-declarations to realise reification.
  - The second theory is that of quantifier-free linear real arithmetic, where CLP($\mathcal{R}$) provides a decision procedure for the theory part of the solver; reification is achieved using a combination of delay declarations and entailment checking.
  - The third theory is that of quantifier-free integer difference logic, with the decision procedure for the theory coded in Prolog and theory propagation realised again using entailment checking and delay declarations.
- A new algorithm for finding the unsatisfiable core of an SMT instance is presented, alongside a correctness argument. The algorithm is faster than the well known QuickXplain [74] algorithm.
- Quantifier-free integer difference logic is a theory that is widely supported by SMT solvers. As a strength test, the Prolog-based solver is benchmarked

against a popular open-source SMT solver, CVC, using standard SMT-LIB benchmarks.

- It is demonstrated that an elegant Prolog-based solver is capable of recovering types for a range of binaries. The solver is benchmarked on these type recovery problems and also compared against an SMT solver constructed from interfacing PicoSAT as a black-box solver using the lazy-basic approach (i.e. without theory propagation). It is also shown how the failed literal technique [83] is simply realised in Prolog to optimise the search. This is a SAT heuristic that discards those Propositional variables (or literals) that falsify the Propositional formula in a single propagation step, so that the search can be focused on other literals.

- Cutting through all of these contributions, it is argued that SMT has a role in type recovery, indeed an SMT formula is a natural medium for expressing the disjunctive nature of the types that arise in reverse engineering.

1.2 Type reconstruction as decompilation

The work of Chapters 2 and 3 succeeded in typing some hand-crafted binaries, and critically, in achieving its intended objective of building a capable solver for the rational tree constraint system proposed by Mycroft. However, when the type reconstruction was evaluated against larger real-world binaries (the results of which are presented in Chapter 3), an inherent shortcoming of the approach was revealed. Subtle errors in the constraints produced for a single instruction were found to lead to conflicts during constraint resolution that were hard to diagnose. Similar issues could be caused by errors in the intermediate machine code representation. Though in time these (seemingly minor) problems could be in some sense "fixed", it was impossible to have confidence that all constraints were correct. In fact, even if solving yielded apparently meaningful types, and even if they corresponded empirically to the original program types, it was impossible to have confidence that the system as a whole was fundamentally correct. This situation is naturally quite unsatisfactory. On reflection, the problem (which has been recognised independently [19]) stems from the absence of a formal link between the x86 instruction semantics (which are themselves not formally defined), the constraints, and the target high-level language type system itself.

Unfortunately, other work in the field suffers from the same problem: Type recovery has been more make-shift and make-do than a discipline shaped by formal principles. IDA Pro applies heuristics to assign simple types to locals [49]. REWARDS [84] recovers types from a single execution trace, which sheds no light on other traces. TIE [81], SecondWrite [41] and Retypd [104] badge themselves as being principled, but do not relate their type judgements to the semantics of the binary (which remain unspecified). Further, a recent survey [19] identified 38[1] works on binary type inference, none of which report anything more than an empirical correctness evaluation. Yet a firm semantic footing for these type systems is essential; a type recovery system which derives an incorrect type can easily mislead a reverse engineer undertaking a security audit, or misdirect a fuzzer into the wrong search space. The consensus is that types assigned to the binary should correspond to the original types of the source. But, needless to say, in reverse engineering this is almost always unavailable. This begs the question: what does it mean for the types to be correct, if there is no source to check correctness against?

Chapter 4 (an expanded and revised version of [110]), answers this question from a semantic perspective by constructing a witness program in a type-safe high-level language. The witness is not an arbitrary program, but carefully constructed to semantically coincide with the binary. Then, by proving that the witness is type-correct, it is established unequivocally that the binary inhabits the recovered types. Structural operational semantics (SOS) define the exemplar low-level and high-level languages, inspired by x86 and C respectively. The centrepiece of the formalisation is a decompilation relation that defines how the witness faithfully mimics the executable, and under what conditions. Together these components add up to a semantically-justified type-based decompiler. In summary, Chapter 4 makes the following contributions:

1. A novel semantics-driven approach to type recovery is presented that validates the types inferred for a low-level (MinX) program with a high-level

[1] Note that this large number is attributable to the inclusion of works on shape analysis, that identify pointer based structures in memory dynamically (using concrete values), and are thus not directly related to static type reconstruction.

witness (MinC) program. Unique to the work is a rigorous connection from the MinX binary to the MinC witness, founded on three key semantic components:

(a) an SOS for MinX, designed as an abstraction of x86 to elucidate crucial control-flow details, such as the argument passing convention, needed to show that the MinX and MinC memories remain truly in sync;

(b) an SOS and static type system for MinC, a type-safe dialect of C designed to illustrate decompilation of pointer arithmetic and the recovery of recursive structures;

(c) a decompilation relation, that conservatively specifies when a MinX program corresponds to a MinC program.

2. In two steps it is formally proven that the MinX program inhabits the recovered types:

   (a) First it is shown that the witness program is type-correct for the derived types.

   (b) Then the operational equivalence of the MinC witness program and the original MinX program is established in the form of memory consistency.

3. A type-based decompiler is distilled from the decompiler relation and demonstrates the potential of the approach.

   (a) It is shown how non-deterministic choices in the relation can be replaced with constraint propagators to give a solver that incrementally infers the witness and the types; the resulting solver is thus solidly based on the decompilation relation.

   (b) The solver is applied to MinX binaries generated from over 21 textbook [122] C programs that manipulate splay trees, treaps, pairing heaps, etc. All (recursive) datatypes are successfully recovered.

1.3 Decision procedures for the octagon abstract domain

As detailed previously, Chapters 2, 3 and 4 focus on different aspects of reconstructing the *types* applicable to the registers and memory of machine code programs. The next body of work in this thesis targets a different aspect of binary reversing, namely the problem of statically inferring the possible *values* that registers and memory may take. When considering this problem, the first point to consider is that it is not tractable to enumerate all the possible values of each register and memory location at each program point through every path in a program. Even for small programs, the number of possible paths is large enough that it might take a lifetime to enumerate them all [8, Chapter 1].

Instead, the accepted wisdom is to deal with a program abstraction, using the abstract interpretation (AI) framework first established in [28]. The force of AI is that it provides a tractable means to compute an over-approximation of the possible values of a program's variables (or in this case, registers and memory). This is achieved by defining a concrete domain that captures the property of interest, and an abstract domain that abstracts that property. Perhaps the simplest example is the interval domain, where ranges are constructed that describe the maximum and minimum values that program variables may take, as an abstraction of the actual concrete values. To complete the abstract interpretation, the semantics of both domains must be defined and related. The concrete domain is described by the collecting semantics, which details how concrete values are constructed by program execution. Meanwhile, the abstract semantics perform the same function for the abstract domain, instead describing how each program statement transforms the abstract state.

The relationship between the domains is defined by two mappings; the abstraction mapping α, and the concretisation mapping γ. The former, α, maps concrete states to their abstract counterparts, while γ performs the dual. Taking the interval domain as an example, a given set of possible concrete values for a variable x, say $\{1, 2, 5\}$ are abstracted under α to yield $\alpha(\{1, 2, 5\}) = [1, 5]$. At the same time, it is easy to see that the abstraction creates an over-approximation, because the concretisation of the interval is $\gamma([1, 5]) = \{1, 2, 3, 4, 5\}$, a strict superset of

$\{1, 2, 5\}$. Note that though information has been lost, all the concrete states have been captured, and there is still sufficient information to reason that x cannot be strictly less than 1 and strictly greater than 5. AI is applied by using the abstract semantics to compute the abstract state until a fixpoint is reached (i.e. the abstract state no longer changes). Various techniques, most notably widening [28], may be employed to guarantee convergence to a fixpoint. However, at this point we must refer the reader to the very thorough explanation of AI given in [1]. Instead, we will now focus on one particular abstract domain; the octagon domain.

The octagon domain [94] has become the de facto standard domain for large-scale program analysis. Each invariant in the domain is a system (conjunction) of inequalities over the variables in the program; each inequality takes the restricted form of $\pm x_i \pm x_j \leqslant c$, where x_i and x_j are variables and c is a numerical constant. When $x_i = x_j$ the inequality is unary otherwise it is dyadic (binary). A unary inequality can express a lower or an upper bound on a variable; whereas a dyadic inequality places a lower or an upper bound on either the difference between two variables or their sum. A solid planar octagon is expressed as the system

$$\begin{array}{rcrcrcrc} x_2 \leqslant 1 & \wedge & x_1 + x_2 \leqslant 1 & \wedge & x_1 \leqslant 1 & \wedge & x_1 - x_2 \leqslant 1 & \wedge \\ -x_2 \leqslant 1 & \wedge & -x_1 - x_2 \leqslant 1 & \wedge & -x_1 \leqslant 1 & \wedge & -x_1 + x_2 \leqslant 1 & \end{array}$$

hence the name of the domain. The domain of octagons is more expressive than the domain of intervals [54] because intervals cannot express differences [92]. Moreover, octagons are more expressive than differences [92] since differences cannot bound sums. Yet the domain of octagons is not as rich as the two-variable-per-inequality (TVPI) abstract domain [112] which relaxes the requirement that coefficients are ± 1 and the TVPI domain is, in turn, less expressive than general polyhedra [30] that permit arbitrary n-ary inequalities to be represented.

Domain construction is a balancing act since increasing expressiveness normally degrades performance. The octagon domain has proved to be popular because it is rich enough to support many client applications, yet all its operations can be reduced to shortest path problems [43]. Octagons have been applied in model checking [71], shape analysis [86], interpolation [50], proving program termination [13] and deployed in commercial static analysis tools [29]. Any computational improvement for this domain thus promises to have wide impact.

Minè , who first proposed this domain [94], used difference bound matrices (DBMs) to represent a system of octagonal inequalities. His insight was to introduce auxiliary variables $x'_{2i+1} = -x_i$ and put $x'_{2i} = x_i$ so that inequalities such as $x_i + x_j \leqslant c$ and $-x_i - x_j \leqslant c$ can be translated into differences, namely $x'_{2i} - x'_{2j+1} \leqslant c$ and $x'_{2i+1} - x'_{2j} \leqslant c$, and thereby represented with DBMs. Moreover the unary inequalities $x_i \leqslant c$ and $-x_i \leqslant c$ can also be represented as differences by $x'_{2i} - x'_{2i+1} \leqslant 2c$ and $x'_{2i+1} - x'_{2i} \leqslant 2c$ respectively. Minè derived a canonical form for octagons by applying a Floyd-Warshall style algorithm [43] on the DBMs; he also showed how all the domain operations can be reduced to computing this canonical form, which is derived by an operation called closure. The intuition behind closure is that it makes explicit all entailed unary and binary constraints and thereby provides a canonical representation. As well as combining two differences such as $x'_i - x'_j \leqslant c_1$ and $x'_j - x'_k \leqslant c_2$ to derive the entailed inequality $x'_i - x'_k \leqslant c_1 + c_2$, closure amalgamates unary constraints into a binary constraint. This requires special logic since $x_i \leqslant c_1$ and $x_j \leqslant c_2$ are encoded as $x'_{2i} - x'_{2i+1} \leqslant 2c_1$ and $x'_{2j} - x'_{2j+1} \leqslant 2c_1$ which need to be combined to give $x'_{2i} - x'_{2j+1} \leqslant 2c_1 + 2c_2$ that encodes $x_i + x_j \leqslant c_1 + c_2$. Likewise $-x_i \leqslant c_1$ and $-x_j \leqslant c_2$ need to be combined to give $-x_i - x_j \leqslant c_1 + c_2$, etc.

Minè adapted the Floyd-Warshall algorithm, which repeatedly combines differences, to handle unary constraints. Later it was independently shown, through an ingenious correctness argument [3], that unary constraints can be handled outside the main loop of the Floyd-Warshall algorithm, in a post-processing step called strengthening. This result led to a performance improvement of approximately 20% [3] which is truly worthwhile. Another worthwhile refinement, which was advocated by Minè himself [93], is to exploit the frequent use-case in which a single inequality is added to a closed system. Minè reordered the columns and rows of the DBM (at least conceptually) so that only entries in the last two columns and rows were recomputed, which led to a quadratic algorithm. Chapter 5 makes the observation that an incremental algorithm for closure can be derived by considering only the paths that might change when adding a new constraint. As with the strengthening refinement, which is essentially a very clever form of code motion, this observation likewise leads to simpler and more efficient code.

Chapter 5 proposes a new incremental algorithm for closing an octagon represented as a DBM. The chapter is a revised and expanded version of [23], and additionally corrects an error in that paper (see example 6 in Section 5.2.2). The chapter makes the following contributions:

- The new algorithm is simple and offers a significant performance benefit over prior approaches.
- The correctness of the algorithm is proven and experimental results are presented which quantify their relative speed.
- Substantially simpler and more concise (than [3]) correctness proofs are provided for non-incremental versions of the algorithm.

Chapter 2

Theory Propagation and Reification

DPLL-based SAT solvers have advanced to the point where they can rapidly decide the satisfiability of structured problems that involve tens of thousands of variables. SAT Modulo Theories (SMT) seeks to extend these ideas beyond Propositional formulae to formulae that are constructed from logical connectives that combine constraints drawn from a given underlying theory. This chapter introduces the motivating problem of type reconstruction from binaries, and explains why it leads to work on theory propagation in a Prolog SMT solver. An SMT solving framework based on reification in Prolog is introduced, including a new algorithm for calculating the unsatisfiable core of an insoluble SMT instance, whose correctness is argued.

2.1 Illustrative example

Returning to the earlier example of Section 1.1, consider the problem of inferring types for the registers of the x86 assembly function, which is reprinted below:

```
         mov edx, [esp+0x4]
         mov eax, 0x0
loop:    test edx, edx
         jz end
         add eax, [edx]
         mov edx, [edx+0x4]
         jmp loop
end:     ret
```

Recall that the function iteratively sums the elements in a C linked list of type:

```
struct list { int value; struct list *next_node; }
```

The function is simple: first `edx` is set to point at the first list item (from the argument carried at `[esp+0x4]`) and `eax`, the accumulator, is initialised to 0 (lines 1 and 2). In the loop body the `value` of the item is added to `eax` (line 5) and `edx` is set to point to the next item by dereferencing the `next_node` field from `[edx+0x4]` (line 6). This repeats until a NULL pointer is found by the test on line 3, whereupon execution jumps to `end` and the function returns.

Before typing the function, indirect addressing is simplified by introducing new operations on fresh intermediate variables (A, B and C below). This reduction ensures that indirect addressing only ever occurs on `mov` instructions, thus simplifying the constraints on all other instructions. Registers are then broken into live ranges by transforming into Single Static Assignment (SSA) form [33]. This gives each variable a new index whenever it is written to, and joins variables at control flow merge points with ϕ functions. The listing below shows the result of applying these transformations:

```
1           mov A_1, esp_0
2           add A_2, 0x4
3           mov edx_1, [A_2]
4           mov eax_1, 0x0
5   loop:   mov (eax_2,edx_2),
                 φ((eax_1,edx_1),(eax_3,edx_3))
6           test edx_2, edx_2
7           jz end
8           mov B_1, [edx_2]
9           add eax_3, B_1
10          mov C_1, edx_2
11          add C_2, 0x4
12          mov edx_3, [C_2]
13          jmp loop
14  end:    ret
```

Rational-tree [65] constraints, describing unification of terms and type variables, are now derived for each instruction. These are similar to the disjunctive constraints described by Mycroft for RTL [100], but include a memory model that tracks pointer manipulation by representing memory in 'pointed to' locations as a 3-tuple. These work in much the same way as the zipper data structure of functional languages [66]. The type of the specific location being pointed to is the middle element, the first element is a list of types for the bytes preceding the location, and the last the types for the bytes succeeding. The lists are open, as indicated by the ellipsis (...), since the areas of memory extending to either side are unknown. For example, consider `add` on line 11. This gives rise to two constraints, one for each possible meaning of the code:

$$(T_{C_2} = \text{basic}(\_, \text{int}, 4) \wedge T_{C_1} = T_{C_2})$$
$$\vee \left(\begin{array}{l} T_{C_1} = \text{ptr}(\langle [\ldots], \beta_0, [\beta_1, \beta_2, \beta_3, \beta_4, \ldots] \rangle) \wedge \\ T_{C_2} = \text{ptr}(\langle [\ldots, \beta_0, \beta_1, \beta_2, \beta_3], \beta_4, [\ldots] \rangle) \end{array} \right)$$

The first clause of the disjunction states that $\mathtt{C}_2$ is of basic type, specifically a four byte integer (derived from the register size) with unknown signedness (as indicated by a sign parameter that is an uninstantiated variable), the result of adding 4 to $\mathtt{C}_1$, which has the same type. This is disjoint from the second clause, that asserts that $\mathtt{C}_1$ is a pointer to an unknown type β_0, whose address is incremented by 4 by the `add` operation so that its new instance, $\mathtt{C}_2$, points to another location of type β_4. Observe how T_{C_1} prescribes types of objects that follow the object of type β_0 in memory whereas T_{C_2} details types of objects that precede the object of type β_4. If further information is later added to T_{C_2} due to unification it will propagate into T_{C_1}, and vice-versa, thus aggregate types analogous to C structs are derived.

Table 1 shows all constraints generated for the program. Note that some type variables have been relaxed to _, indicating an uninstantiated variable, so as to simplify the presentation of the types. The complete problem is described by the conjunction of these constraints. Type recovery then amounts to solving the constraints such that the type equations remain consistent, whilst also ensuring that the Propositional skeleton of the problem is satisfied.

In the case of the example, for the register used to store the argument of type `struct list *`, constraint solving will derive a recursive type:

$$T_{edx_1} = \text{ptr}(\langle [\ldots], \text{basic}(\_, \text{int}, 4), [\_, \_, \_, T_{edx_1}, \_, \_, \_, \ldots] \rangle)$$

Line	Generated Constraints
1	$T_{A_1} = T_{esp_0}$
2	$(T_{A_2} = \text{basic}(\_,\ \text{int},\ 4) \wedge T_{A_1} = T_{A_2}) \vee \left(\begin{array}{c} T_{A_1} = \text{ptr}(\langle[\ldots], \alpha_0, [\_, \_, \_, \alpha_1, \ldots]\rangle) \wedge \\ T_{A_2} = \text{ptr}(\langle[\ldots, \alpha_0, \_, \_, \_], \alpha_1, [\ldots]\rangle) \end{array}\right)$
3	$T_{A_2} = \text{ptr}(\langle[\ldots], T_{edx_1}, [\_, \_, \_, \ldots]\rangle)$
4	$T_{eax_1} = \text{basic}(\_,\ \text{int},\ 4) \vee T_{eax_1} = \text{ptr}(\langle[\ldots], \alpha_2, [\ldots]\rangle)$
5	$(T_{eax_2} = T_{eax_1} \wedge T_{edx_2} = T_{edx_1}) \wedge (T_{eax_2} = T_{eax_3} \wedge T_{edx_2} = T_{edx_3})$
8	$T_{edx_2} = \text{ptr}(\langle[\ldots], T_{B_1}, [\ldots]\rangle)$
9	$\left(\begin{array}{c} T_{eax_3} = \text{basic}(\_,\text{int},4) \wedge \\ T_{eax_2} = T_{eax_3} \wedge T_{B_1} = T_{eax_3} \end{array}\right) \vee \left(\begin{array}{c} T_{eax_3} = \text{ptr}(\langle[\ldots], \alpha_3, [\ldots]\rangle) \wedge \\ T_{eax_2} = \text{ptr}(\langle[\ldots], \alpha_4, [\ldots]\rangle) \wedge \\ T_{B_1} = \text{basic}(\_,\text{int},4)) \end{array}\right) \vee \left(\begin{array}{c} T_{eax_3} = \text{ptr}(\langle[\ldots], \alpha_5, [\ldots]\rangle) \wedge \\ T_{eax_2} = \text{basic}(\_,\text{int},4) \wedge \\ T_{B_1} = \text{ptr}(\langle[\ldots], \alpha_6, [\ldots]\rangle) \end{array}\right)$
10	$T_{edx_2} = T_{C_1}$
11	$(T_{C_2} = \text{basic}(\_,\ \text{int},\ 4) \wedge T_{C_1} = T_{C_2}) \vee \left(\begin{array}{c} T_{C_1} = \text{ptr}(\langle[\ldots], \alpha_7, [\_, \_, \_, \alpha_8, \ldots]\rangle) \wedge \\ T_{C_2} = \text{ptr}(\langle[\ldots, \alpha_7, \_, \_, \_], \alpha_8, [\ldots]\rangle) \end{array}\right)$
12	$T_{C_2} = \text{ptr}(\langle[\ldots], T_{edx_3}, [\_, \_, \_, \ldots]\rangle)$

Table 1: Generated constraints

which requires rational-tree unification. Note that as T_{edx_1} is a pointer, it has a size of four bytes, hence the 3 underscores that follow T_{edx_1} which correspond to 3 bytes of padding in this representation.

Observe that there may be multiple solutions; in fact the problem outlined above has two solutions, which differ in typing $\texttt{eax}_1$, $\texttt{eax}_2$ and $\texttt{eax}_3$. The first correctly infers that they are (like $\texttt{B}_1$) integers of size 4 bytes, while the second defines them as pointers to an unknown type, $\text{ptr}(\langle[\ldots], \alpha_5, [\ldots]\rangle)$. Both solutions have the following typings in common:

$$
\begin{gathered}
T_{B_1} = \text{basic}(\_,\text{int},4) \\
T_{edx_1} = T_{edx_2} = T_{edx_3} = T_{C_1} = \text{ptr}(\langle[...], \text{basic}(\_,\ \text{int},\ 4), [\_, \_, \_, T_{edx_1}, \_, \_, \_, ...]\rangle) \\
T_{C_2} = \text{ptr}(\langle[..., \text{basic}(\_,\text{int},4), \_, \_, \_], T_{edx_1}, [\_, \_, \_, ...]\rangle) \\
T_{A_2} = \text{ptr}(\langle[..., \alpha_0, \_, \_, \_], T_{edx_1}, [\_, \_, \_, ...]\rangle) \\
T_{A_1} = T_{esp_0} = \text{ptr}(\langle[...], \alpha_0, [\_, \_, \_, T_{edx_1}, \_, \_, \_, ...]\rangle)
\end{gathered}
$$

The second solution is equivalent to typing `eax` as `void*` and performing addition using pointer arithmetic. In the wider context of a program, this solution is removed by constraints derived from the `main()` function.

2.2 SMT and Theory Propagation

The Boolean satisfiability problem (SAT) is the problem of determining whether for a given Boolean formula, there is a truth assignment to the variables of the formula under which the formula evaluates to *true*. Most recent SAT solvers are based on the Davis, Putnam, Logemann, Loveland (DPLL) algorithm [34] with watched literals [98].

2.2.1 SAT solving and unit propagation

At the heart of the DPLL approach is unit propagation. Let f be a Propositional formula in Conjunctive Normal Form (CNF) over a set of Propositional variables X. Let $\theta : X \to \{\mathit{true}, \mathit{false}\}$ be a partial (truth) function. Unit propagation examines each clause in f to deduce a truth assignment θ' that extends θ and necessarily holds for f to be satisfiable. For example, suppose $f = (\neg x \vee z) \wedge (u \vee \neg v \vee w) \wedge (\neg w \vee y \vee \neg z)$ so that $X = \{u, v, w, x, y, z\}$ and θ is the partial function $\theta = \{x \mapsto \mathit{true}, y \mapsto \mathit{false}\}$. In this instance for the clause $(\neg x \vee z)$ to be satisfiable, hence f as a whole, it is necessary that $z \mapsto \mathit{true}$. Moreover, for $(\neg w \vee y \vee \neg z)$ to be satisfiable, it follows that $w \mapsto \mathit{false}$. The satisfiability of $(u \vee \neg v \vee w)$ depends on two unknowns, u and v, hence no further information can be deduced from this clause. Therefore $\theta' = \theta \cup \{w \mapsto \mathit{false}, z \mapsto \mathit{true}\}$.

```
sat(Clauses, Vars) :-
    problem_setup(Clauses), elim_var(Vars).

elim_var([]).
elim_var([Var | Vars]) :-
    elim_var(Vars), (Var = true; Var = false).

problem_setup([]).
problem_setup([Clause | Clauses]) :-
    clause_setup(Clause),
    problem_setup(Clauses).

clause_setup([Pol-Var | Pairs]) :- set_watch(Pairs, Var, Pol).

set_watch([], Var, Pol) :- Var = Pol.
set_watch([Pol2-Var2 | Pairs], Var1, Pol1):-
    watch(Var1, Pol1, Var2, Pol2, Pairs).

:- block watch(-, ?, -, ?, ?).
watch(Var1, Pol1, Var2, Pol2, Pairs) :-
    nonvar(Var1) ->
        update_watch(Var1, Pol1, Var2, Pol2, Pairs);
        update_watch(Var2, Pol2, Var1, Pol1, Pairs).

update_watch(Var1, Pol1, Var2, Pol2, Pairs) :-
    Var1 == Pol1 -> true; set_watch(Pairs, Var2, Pol2).
```

Figure 1: Prolog SAT solver using watched literals [63]

Searching for a satisfying assignment proceeds as follows: starting from an empty truth function θ, an unassigned variable occurring in f, x, is selected and $x \mapsto \mathit{true}$ is added to θ. Unit propagation extends θ until either no further propagation is possible or a contradiction is established. In the first case, if all clauses are satisfied then f is satisfied, else another unassigned variable is selected. In the second case, $x \mapsto \mathit{false}$ is added to θ; if this fails the search backtracks to a previous assignment.

Figure 1 provides the code listing for a Prolog SAT solver using watched literals, taken from [63]. The watched literals technique is founded on the observation that a particular clause can provide further information only if it does not contain two unassigned variables [98]. Therefore, for each clause of a problem, two unassigned variables are watched, with propagation occurring once either is assigned. The SAT solver in Figure 1 takes a problem in CNF, specified as a list of clauses, and a list of variables. Each clause is itself a list of pairs `Pol-Var`, where `Var` is a variable, and `Pol` indicates whether the variable has positive or negative polarity by being either `true` or `false` respectively. For each clause, the `set_watch` predicate is called, which sets up the first two variables in the clause (`Var1` and `Var2`) to be watched. This is achieved by using a `block` declaration to suspend the `watch` predicate as a coroutine until `Var1` or `Var2` is instantiated (as specified by the corresponding '`-`' in the arguments of the declaration).

When a variable is instantiated `watch` resumes and executes `update_watch`. If the instantiated variable matches its polarity, the clause is satisfied, and the coroutine exits successfully. Otherwise another variable is selected for watching. If there are no variables left then the clause is unsatisfiable and the goal will fail. The search is realised as previously explained by using the `elim_var` predicate to assign variables to `true` or `false`, with unit propagation occurring via `watch` whenever watched variables are assigned. The search for a satisfying assignment is then completed simply by Prolog backtracking.

2.2.2 SMT solving, the lazy-basic approach

SAT modulo theories (SMT) gives a general scheme for determining the satisfiability of problems consisting of a formula over atomic constraints in some theory T, whose set of literals is denoted Σ [103, 115]. The scheme separates the Propositional skeleton – the logical structure of combinations of theory literals – and the meaning of the literals. A bijective encoder mapping $e : \Sigma \rightarrow X$ associates each literal with a unique Propositional variable. Then the encoder mapping e is lifted to theory formulae, using $e(\phi)$ to denote the Propositional skeleton of a theory formula ϕ.

Consider the theory of quantifier-free linear real arithmetic where the constants are numbers, the functors are interpreted as addition and subtraction, and the predicates include equality, disequality and both strict and non-strict inequalities. The problem of checking the entailment $(a < b) \wedge (a = 0 \vee a = 1) \wedge (b = 0 \vee b = 1) \models (a + b = 1)$ amounts to determining that the theory formula $\phi = (a < b) \wedge (a = 0 \vee a = 1) \wedge (b = 0 \vee b = 1) \wedge \neg(a + b = 1)$ is not satisfiable. For this problem, the set of literals is $\Sigma = \{a < b, \ldots, a + b = 1\}$. Suppose, in addition, that the encoder mapping is defined:

$$\begin{array}{lll} e(a < b) = x, & e(a = 0) = y, & e(a = 1) = z, \\ e(b = 0) = u, & e(b = 1) = v, & e(a + b = 1) = w \end{array}$$

Then the Propositional skeleton of ϕ, given e, is $e(\phi) = x \wedge (y \vee z) \wedge (u \vee v) \wedge \neg w$. A SAT solver gives a truth assignment θ satisfying the Propositional skeleton. From this, a conjunction of theory literals, $\hat{Th}_{\Sigma}(\theta, e)$ is constructed. The conjunction contains the literal ℓ if $\theta(e(\ell)) = \mathit{true}$ and $\neg\ell$ if $\theta(e(\ell)) = \mathit{false}$. The subscript will be omitted when Σ refers to all literals in a problem. This problem is passed to a solver for the theory that can determine satisfiability of conjunctions of constraints. Either satisfiability or unsatisfiability is determined, in the latter case the SAT solver is asked for further satisfying truth assignments. Details of a Prolog implementation of this approach can be found in [64].

2.2.3 SMT, the DPLL(T) approach

The approach detailed in the previous section finds complete satisfying assignments to the SAT problem given by the Propositional skeleton before computing the satisfiability of the theory problem $\hat{Th}(\theta, e)$. Another approach is to couple the SAT problem and the theory problem more tightly by determining constraints entailed by the theory and propagating the bindings back into the SAT problem. This is known as theory propagation and is encapsulated in the DPLL(T) approach. Figure 2 gives a recursive formulation of DPLL(T) derived from Algorithm 11.2.3 of [79]. A more general formulation of DPLL(T) might replace lines (11)-(15) with a conflict analysis step that would encapsulate not just the approach presented, but also backjumping and clause learning heuristics. However, the key component of DPLL(T) is the interleaving of unit and theory propagation and the choice of conflict analysis is an orthogonal issue. The instantiation to chronological backtracking presented in Figure 2 was chosen to match the implementation work.

The first argument to the function DPLL(T) is a Boolean formula f, its second a partial truth assignment, θ, and its third an encoder mapping, e. In the initial call, f is the Propositional skeleton of the input problem, $e(\phi)$, and θ is empty. DPLL(T) returns a truth assignment if the problem is satisfiable and the constant

```
function DPLL(T)(f: CNF formula, θ : truth assignment, e : Σ → X)
begin
    (θ₃, res) := propagate(f, θ, e, ∅);
    if (is-satisfied(f, θ₃)) then
        return θ₃;
    else if (res = ⊥) then
        return ⊥;
    else
        x := choose-free-variable(f, θ₃);
        (θ₄, res) := DPLL(T)(f, θ₃ ∪ {x ↦ true}, e);
        if (res = ⊤) then
            return θ₄;
        else
            return DPLL(T)(f, θ₃ ∪ {x ↦ false}, e);
        endif
    endif
end
```

```
function propagate(f: CNF formula, θ : truth assignment,
                    e : Σ → X, D : set of theory literals)
begin
    θ₁ := θ ∪ {e(ℓ) ↦ true | ℓ ∈ D ∩ Σ} ∪ {e(ℓ) ↦ false | ¬ℓ ∈ D ∧ ℓ ∈ Σ};
    θ₂ := θ₁∪ unit-propagation(f,θ₁);
    (D, res) := deduction(T̂h(θ₂, e));
    if (D = ∅ ∨ res = ⊥)
        return (θ₂, res);
    else
        return propagate(f, θ₂, e, D);
    endif
end
```

Figure 2: Recursive formulation of the DPLL(T) algorithm

$\perp$ otherwise.

The call to propagate is the key operation. The function returns a pair consisting of a truth assignment and res taking value $\top$ or $\perp$ indicating the satisfiability of f and $\hat{Th}(\theta, e)$. The fourth argument to propagate is a set of theory literals, D, and the function begins by extending the truth assignment by assigning Propositional variables identified by the encoder mapping. Next, unit propagation as described in Section 2.2.1 is applied. The deduction function then infers those literals that hold as a consequence of the extended truth assignment. The function returns a pair consisting of a set of theory literals entailed by $\hat{Th}(\theta_2, e)$ and a flag res whose value is $\perp$ if $\hat{Th}(\theta_2, e)$ or θ_2 is inconsistent and $\top$ otherwise. The function propagate calls itself recursively until no further propagation is possible. After deduction returns, if f is not yet satisfied then a further truth assignment is made and DPLL(T) calls itself recursively.

The key difference between the lazy-basic approach and the DPLL(T) approach is that where the lazy-basic approach computes a complete satisfying assignment to the variables of the Propositional skeleton before investigating the satisfiability of the corresponding theory formula, the DPLL(T) approach incrementally investigates the consistency of the posted constraints as Propositional variables are assigned. Further, it identifies literals, ℓ, such that $\hat{Th}(\theta, e) \models \ell$, allowing $e(\ell)$ to be assigned during propagation. It is the interplay between Propositional satisfiability, posting constraints and the consistency of the store $\hat{Th}(\theta, e)$ that is at the heart of this investigation.

2.3 Theory Propagation and Reification

This section provides a framework for incorporating theory propagation into the propagation framework of the SAT solver introduced in section 2.2.1. The approach is based on reifying theory literals with logical variables. As will be illustrated in subsequent sections, this allows the use of the control provided by delay declarations to realise theory propagation. The integration is almost seamless since the base SAT solver is also realised using logical variables and by exploiting the control provided by delay declarations.

2.3.1 Theory propagation

There are three major steps in setting up a DPLL(T) solver for some problem ϕ:

```
dpll_t(Prob):-
    setup(Prob, TheoryLiterals, Skeleton, Vars),
    post_theory(TheoryLiterals),
    post_boolean(Skeleton),
    elim_vars(Vars).
```

Figure 3: Interface to the DPLL(T) solver

1. Setting up the encoder map e, linking each theory literal in a problem with a logical variable.
2. Posting theory propagators (adding constraints) that reify the theory literals with the logical variables provided by e.
3. Posting the SAT problem defined by the Propositional skeleton $e(\phi)$.

The code in Figure 3 describes the high level call to the solver.

Set up Where `Prob` is an SMT formula over some theory, let lit(`Prob`) be the set of literals occurring in `Prob`. `TheoryLiteral` is a list of pairs $(\ell, e(\ell))$, where $\ell \in lit$(`Prob`), that defines the encoder mapping e. `Skeleton` represents the Propositional skeleton of the problem, e(`Prob`). `Vars` represents the set of variables $e(\ell)$, where $\ell \in lit$(`Prob`). The role of the predicate `setup(+,-,-,-)` is, given `Prob`, to instantiate the remaining variables.

Theory propagators The role of `post_theory` is to set up predicates to reify each theory literal. The control on these predicates is key; the predicates need to be blocked until either $e(\ell)$ is assigned, or the literal (or its negation) is entailed by the constraint store $\hat{Th}(\theta, e)$. That is, the predicate for $(\ell, e(\ell))$ will propagate in one of four ways:

- If $\hat{Th}(\theta, e) \models \ell$ then $e(\ell) \mapsto \mathit{true}$
- If $\hat{Th}(\theta, e) \models \neg\ell$ then $e(\ell) \mapsto \mathit{false}$
- If $e(\ell) = \mathit{true}$ then the store is updated to $\hat{Th}(\theta \cup \{e(\ell) \mapsto \mathit{true}\}, e)$
- If $e(\ell) = \mathit{false}$ then the store is updated to $\hat{Th}(\theta \cup \{e(\ell) \mapsto \mathit{false}\}, e)$

Boolean propagators The role of `post_boolean` is to set up propagators for the SAT part of the problem $e($`Prob`$)$. This uses Prolog `block` declarations to wait for Propositional variables to be sufficiently instantiated to propagate bindings to other Propositional variables following the watched literals technique as explained in section 2.2.1.

Implementing the interface provided by predicates `setup` and `post_theory`, together with the SAT solver from section 2.2.1 results in a DPLL(T) SMT solver. Note that the propagators posted for the theory and Boolean components are intended to capture the spirit of the function propagate from Figure 2. Indeed, the integration between theory and Boolean propagation is even tighter than the algorithm indicates. Rather than performing unit propagation to completion, then performing theory propagation, then repeating, here the assignment of a Boolean variable is immediately communicated to the theory. This tactic is known as immediate propagation [79, Chapter 11] and is a natural consequence of using Prolog's control to implement propagators. Immediate propagation does away with the need to analyse failure to determine an unsatisfiable core when a set of theory constraints is unsatisfiable, but attracts a cost in monitoring the entailment status of the theory literals.

2.3.2 Labelling strategies

The solvers presented in [64] maintain Boolean variables in a list and `elim_vars` assigns them values in the order in which they occur; the list has typically been ordered by the number of occurrences of the variables in the SAT instance before the search begins, the most frequently occurring assigned first. This tactic is straightforward to accommodate into a solver coded in Prolog. However, in the case of type recovery problems constraints are derived over entire programs, leading to SMT instances that can have hundreds of thousands of clauses even for binaries of less than 100 kilobytes. This desire to make type recovery problems tractable motivates the adoption of more sophisticated heuristics for variable assignment.

One classic strategy for labelling that is also straightforward to incorporate into a solver written in a declarative language is to rank variables by their number of occurrences in clauses of minimal size [70]. This associates a weight to each unbound variable according to its number of occurrences in the unsatisfied clauses of the (Boolean) problem. The ranking weights variables with fewer unbound literals less heavily than those in clauses with a greater number of unbound literals. A variable with greatest weight is selected for labelling, the aim being to assign one that is more likely to lead to propagation.

However, this tactic is still insufficient to solve type recovery problems within a reasonable time frame (less than several hours) for the benchmarked binaries. A refinement is to apply lookahead [83] in conjunction with this labelling tactic. Each variable with greatest weight, and therefore each candidate for labelling, is speculatively assigned a truth value. For example, if `X` is assigned *true* and this results in failure, then in order to satisfy the Propositional formula (skeleton) then `X` must be assigned *false*. Likewise, if failure occurs when `X` is assigned *false* then `X` must be *true*. Moreover, if one variable can be assigned using lookahead, then often so can others, hence this tactic is repeatedly applied until no further variables can be bound. Thus lookahead is tried before any variable is assigned by search. Scoping this activity over the variables of greatest weight limits the overhead of lookahead. The net effect is to direct the search away from variable assignments that will ultimately fail. In practice lookahead is crucial for reducing the search time required for real-world type recovery problems.

2.3.3 Calculating an unsatisfiable core

Given an unsatisfiable SMT problem, it can be useful to find an unsatisfiable core of this problem, that is, a subset of the theory literals, $\Sigma' \subseteq \Sigma$, such that $\hat{Th}_{\Sigma'}(\theta, e)$ is not satisfiable for any assignment θ, and for all $\Sigma'' \subset \Sigma'$ there exists a θ such that $\hat{Th}_{\Sigma''}(\theta, e)$ is satisfiable.

The unsatisfiable core needs to be calculated in the lazy-basic approach in order to generate a blocking clause. Further, in the application to type recovery problems, it is useful to be able to diagnose the cause of unsatisfiability to resolve errors in the type system. An unsatisfiable core for the type recovery problems is typically small and due to the size of the problems can take several hours to find using existing algorithms (such as QuickXplain [74]). This motivates a more aggressive algorithm for pruning out literals that are not in a core. Such an

function findcore ($e = [t_1 \mapsto x_1, \ldots, t_n \mapsto x_n] : \Sigma \to X$,
 f : CNF formula, c : int, $core : \Sigma \to X$)
begin
  if ($e = [\,]$)
    return $core$;
  else if ($c = 0$)
    $core' := [t_1 \mapsto x_1, t_n \mapsto x_n] \cup core$;
    return findcore($[t_2 \mapsto x_2, \ldots, t_{n-1} \mapsto x_{n-1}]$, f, $\lfloor \frac{n-1}{2} \rfloor$, $core'$);
  else
    $i := 1$; $j := n$;
    if ($\neg$DPLL(T)(f, $\emptyset$, $[t_{c+1} \mapsto x_{c+1}, \ldots, t_n \mapsto x_n] \cup core$))
      $i := c + 1$;
    endif
    if ($\neg$DPLL(T)(f, $\emptyset$, $[t_i \mapsto x_i, \ldots, t_{n-c} \mapsto x_{n-c}] \cup core$))
      $j := n - c$;
    endif
    if ($c = 1$)
      $c' := 0$;
    else
      $c' := \lfloor \frac{c+1}{2} \rfloor$;
    endif
    return findcore($[t_i \mapsto x_i, \ldots, t_j \mapsto x_j]$, f, c', $core$);
  endif
end

Figure 4: Finding an unsatisfiable core

algorithm is presented in Figure 4.

The first argument to findcore is (an ordered representation of) a partial encoder mapping from theory literals to Propositional variables; the second argument is a Propositional formula, namely $e(\phi)$, the Propositional skeleton of the initial problem; the third argument is an integer, giving the number of elements of the mapping on literals that will be pruned from one end (and then the other end) in order to investigate satisfiability; the fourth argument is a partial mapping from theory literals to Propositional variables, where the theory literals are part of the unsatisfiable core. The initial call to the function is findcore(e,$e(\phi)$,$\lceil \frac{m}{2} \rceil$,$\emptyset$), where e is the complete encoder map for Σ, $[t_1 \mapsto e(t_1), \ldots, t_m \mapsto e(t_m)]$.

The algorithm removes c elements from the beginning of the mapping (represented as a list) and tests the resulting problem for satisfiability. If the problem remains unsatisfiable, the c elements removed are not part of the unsatisfiable core and can be pruned all at once. This is repeated for the end of the mapping. The c value begins large and is logarithmically reduced until it has value 0, at which point the first and last elements of the list representing the mapping must be in the core. The function findcore is then again recursively called with these end points removed and the process continues until a core has been found.

The findcore algorithm is related to the previously mentioned QuickXplain algorithm, which likewise computes an unsatisfiable core by removing blocks of consistent constraints. The QuickXplain algorithm recursively divides a set of constaints into two subsets. If the first subset is inconsistent, then the second can be discarded immediately in the search for a core. Otherwise, some constraints from the second are merged with constraints from the first to derive a core. This divide-and-conquer algorithm resembles a precedessor of findcore though, crucially, the above incarnation of the algorithm attempts to remove the first $c_1 = \lceil \frac{m}{2} \rceil$ constraints, then the last c_1, then the first $c_2 = \lceil \frac{c_1}{2} \rceil$, then the last c_2, etc, as it converges onto a core. This appears to be a more aggressive pruning strategy than that applied in QuickXplain which maintains the first subset intact (see [74, Figure 1, Line 9]) whilst pruning the second, though a precise comparison has not been made.

The following Proposition asserts that the algorithm correctly computes a core.

Proposition 1. If $\hat{Th}_\Sigma(\theta, e)$ is unsatisfiable for any assignment θ and e is an encoder mapping for Σ, then function findcore returns an encoder mapping that represents an unsatisfiable core, $\Sigma' \subseteq \Sigma$.

Proof. First it is argued that findcore terminates. Where a call to findcore is findcore$(e, f, c, core)$ consider the ordered pair $\langle |e|, c \rangle$. For each recursive call to findcore the value of this pair lexicographically decreases, therefore by induction findcore terminates.

Next it is demonstrated that findcore returns an encoder mapping representing an unsatisfiable set of variables. A precondition of a call to findcore is that

$\hat{Th}_\Sigma(\theta, e)$ is unsatisfiable. Since *core* is initially empty, the encoder $e \cup core$ represents an unsatisfiable set of variables. Lines (12) and (15) are the two places where findcore changes $e \cup core$. Observe that the conditions on line (11) and (14) state that the change of e to e' (where e' is the updated encoder mapping) occurs only when $e' \cup core$ represents an unsatisfiable set of variables. Hence that $e \cup core$ represents an unsatisfiable set of variables is invariant through the algorithm. When $|e| = 0$ findcore returns, therefore returning $core = e \cup core$ which represents an unsatisfiable set of variables.

Lastly it is demonstrated that the set of variables represented by the encoder mapping returned represents an unsatisfiable core. Note that if some encoder mapping d represents an unsatisfiable set, then $d' \supseteq d$ also represents an unsatisfiable set. Variable mappings are added to *core* on line (7). Suppose $t \mapsto x$ is added to *core* at line (7). In the preceding call with $c = 1$, either $(e \backslash [t_1 \mapsto x_1]) \cup core$ represents a satisfiable set or it does not. (The argument for $t_n \mapsto x_n$ is symmetric.) If satisfiable, then $t_1 \mapsto x_1$ is required for unsatisfiability, and this is the $t \mapsto x$ added in line (7). If unsatisfiable, then the $t \mapsto x$ added in line (7) is $t_2 \mapsto x_2$.

Note that in the initial call to findcore with mapping e (where $|e| = m$) all elements of e to the left of $t_2 \mapsto x_2$ are not part of the representation of the unsatisfiable core since they are removed by lines (11)-(13). The number of elements removed is described by Lemma 1, that is, $m - 1$. Therefore $t_2 \mapsto x_2$ is the only element remaining and this must be part of the representation of the unsatisfiable core, since otherwise it would have been removed by lines (14)-(16). Therefore no redundant elements are added and findcore returns an unsatisfiable core. □

Lemma 1. Where $m \in \mathbb{N}$, let $c_1(m) = \lceil \frac{m}{2} \rceil$ and $c_i(m) = \lceil \frac{c_{i-1}(m)}{2} \rceil$, if $c_{i-1}(m) > 1$ and $c_i(m) = 0$ otherwise. Then $\sum_i c_i(m) \geqslant m - 1$.

Proof. The proof proceeds by induction. Suppose $m = 1$, then $c_1(m) = 1 \geqslant m-1$. Suppose $m = 2$, then $c_1(m) = 1$ and $c_2(m) = 0$, hence $\sum_i c_i(m) = 1 \geqslant m-1$. Now suppose m is odd, that is $m = 2k - 1$ (where $k \in \mathbb{N}$, $k \geqslant 1$), then $c_1(2k-1) = \lceil \frac{2k-1}{2} \rceil = k$ and $c_2(2k-1) = c_1(k)$. Hence $\sum_i c_i(2k-1) = k + \sum_i c_i(k) \geqslant 2k - 1 \geqslant 2k - 2$. Suppose m is even, that is $m = 2k$ (where $k \geqslant 1$) then $c_1(2k) = \lceil \frac{2k}{2} \rceil = k$ and $c_2(2k) = c_1(k)$. Hence $\sum_i c_i(2k) = k + \sum_i c_i(k) \geqslant 2k - 1$. □

Note that when this lemma is applied, $c_i(m)$ corresponds to the third argument of findcore (c) at iteration i. This value is a function of m, the size of the initial encoder map.

2.4 Summary

This chapter showed that the type reconstruction problem can be expressed as an SMT instance over the theory of rational-tree unification, and introduced an SMT solving framework for Prolog based on reification. Prolog is a natural choice for type reconstruction because rational-tree unification is a Prolog language feature. Reification in Prolog is an equally natural fit for SMT solving, because it enables completely automatic synchronisation of unit and theory propagation: The proposed framework eliminates the need to algorithmically interleave or schedule unit and theory propagation (as in classic DPLL(T) algorithms) at all.

A complementary algorithm for finding an unsatisfiable core of an SMT problem was also presented, and its correctness argued. Notably, the new algorithm is more aggressive at pruning the search space than the related QuickXplain algorithm [74].

Chapter 3

Implementing SMT with Reification

Chapter 2 introduced an SMT framework for Prolog based on reification. This chapter takes the logical next step of instantiating the framework with solvers for three theories; rational-tree unification, Linear Real Arithmetic (LRA) and Integer Difference Logic (IDL). In each case it is demonstrated that very close coupling of unit and theory propagation may be achieved by reifying theory literals (constraints) with the logical variables corresponding to Booleans in the Propositional skeleton of the SMT formula. This chapter highlights that the framework enables fast development of succinct (in terms of lines of code) and efficient SMT solvers in Prolog, regardless of whether the underlying theory is an intrinsic Prolog language feature (rational-tree unification), available in the language via a library (LRA), or has to be implemented in Prolog from scratch (IDL).

The rational tree solver is evaluated in the context of type reconstruction against a range of binaries. To test the efficacy of the approach, it is also compared to a lazy-basic (i.e. without theory propagation) solver that uses the PicoSAT open source SAT solver as a black box. Since Quantifier Free Integer Difference Logic (QF_IDL) is a theory commonly found in contemporary SMT solvers, the IDL solver is evaluated against the CVC3 and CVC4 solvers using SMTLib competition benchmarks.

3.1 Instantiation for Rational-Trees

The theory component of an SMT solver requires a decision procedure for determining the satisfiability of a conjunction of theory literals. Unification is at the heart of Prolog and many Prolog systems are based on rational-tree unification, hence a decision procedure for conjunctions of rational-tree constraints comes essentially for free. This can be coupled with the control provided by delay declarations to reify rational-tree constraints, hence implementing the interface described in the previous chapter (Section 2.3). The code in Figure 5 demonstrates the use of delay to realise theory propagation over rational-tree constraints via reification.

An SMT problem over rational-trees consists of Boolean combinations of theory literals ℓ. The call to `setup/4` will instantiate `TheoryLiterals` to a list of pairs of the form $(\ell, e(\ell))$; the Propositional skeleton and a list of the $e(\ell)$ variables are also produced. In the following, a labelled literal `(eqn(Term1, Term2), X)` is discussed. The `post_theory` predicate sets up propagators for each theory literal in two steps, while `theory_wait` propagates from the theory constraints into the Boolean variables.

The predicate `theory_wait` uses the builtin control predicate `when/2`, which blocks the goal in its second argument until the first argument evaluates to *true*. In this instance the condition `?=(Term1, Term2)` is *true* either if `Term1` and `Term2` are identical, or if the terms cannot be unified. That is, if `Term1=Term2` is entailed by the store then `theory_prop` is called and assigns `X=true`. Similarly, if the constraint is not consistent with the store, then `Term1` and `Term2` cannot be unified and again `theory_prop` reflects this by assigning `X=false`. In the opposite direction, `bool_wait` communicates assignments made to Boolean variables to the theory literals. The predicate is blocked on the instantiation of the logical variables, waking when they become `true` or `false`. When `true` the constraint must hold so `Term1` and `Term2` are unified. When `false`, it is not possible for the two terms to be unified, hence the constraint is discarded and the call to `bool_wait` succeeds. Note that it is not possible to post a constraint that asserts that two terms cannot be unified, since the control predicate `dif/2` is defined as:

```
dif(X, Y) :- when(?=(X, Y), X \== Y).
```

That is, it blocks until either `X` and `Y` are identical or they cannot be unified, then

```
post_theory([]).
post_theory([(eqn(Term1,Term2), X)|Rest]) :-
    setup_reify(X, Term1, Term2),
    post_theory(Rest).

setup_reify(X, Term1, Term2) :-
    bool_wait(X, Term1, Term2),
    theory_wait(X, Term1, Term2).

:- block bool_wait(-, ?, ?).
bool_wait(true, Term1, Term2) :-
    Term1 = Term2, !.
bool_wait(false, _Term1, _Term2).

theory_wait(X, Term1, Term2) :-
    when(?=(Term1, Term2), theory_prop(X, Term1, Term2)).

theory_prop(X, Term1, Term2) :-
    Term1 == Term2 ->
        X = true
    ;
        X = false
    .
```

Figure 5: Theory propagation for rational-tree constraints

tests whether or not they are identical. Hence `dif/2` acts as a test, rather than a propagating constraint. Consistency of the store is maintained by `theory_wait`; if `X=false` and the constraint is discarded, then later it is determined that `Term1=Term2`, `theory_wait` will attempt to unify `X` with `true`, which will fail.

3.2 Instantiation for Linear Real Arithmetic

Many Prolog systems come with the CLP($\mathcal{R}$) constraints package, which can determine consistency of conjunctions of linear arithmetic constraints. This makes quantifier-free linear real arithmetic a sensible theory for the solver. The challenge is to implement reification for the constraints, an operation not directly supported.

```
post_theory(TheoryLiterals):-
    setup_reify(TheoryLiterals, _).

setup_reify([], _).
setup_reify([(C, V)|Cs], Y) :-
    negate(C, NegC),
    theory_wait(V, Y, C, NegC),
    setup_reify(Cs, Y).

negate(X =< Y, X > Y).
negate(X < Y, X >= Y).
negate(X = Y, X =\= Y).

next_var(Y, Z) :-
    var(Y), !, Y = Z.
next_var(prop(Y), Z) :-
    next_var(Y, Z).

:- block theory_wait(-, -, ?, ?).
theory_wait(V, Y, C, _NegC) :-
    V == true, !,
    {C}, Y = prop(_).
theory_wait(V, Y, _C, NegC) :-
    V == false, !,
    {NegC}, Y = prop(_).
theory_wait(V, Y, C, _NegC) :-
    nonvar(Y), entailed(C), !,
    V = true.
theory_wait(V, Y, _C, NegC) :-
    nonvar(Y), entailed(NegC), !,
    V = false.
theory_wait(V, Y, C, NegC) :-
    next_var(Y, U),
    theory_wait(V, U, C, NegC).
```

Figure 6: Theory propagation for linear real arithmetic

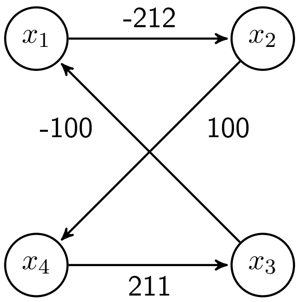
The code in Figure 6 demonstrates the integration of a linear real arithmetic decision procedure as realised by CLP($\mathcal{R}$) into the DPLL(T) scheme. It assumes that the input problem has been normalised so that all the constraint predicates are drawn from =, =< and <. The propagators, `theory_wait`, are blocked on two variables. The first of these is the labelling variable e(`C`) – if this is instantiated, the appropriate constraint is posted. To complete the reification, the propagators need to detect the entailment of the linear constraint (or its negation). This can be achieved using the builtin `entailed/1`, however the control for ensuring that this is called at an appopriate time is less obvious.

Once a new constraint has been posted (or once the constraint store has changed) other constraints or their negations might be entailed and this needs to be detected and propagated. The communication between the propagators to capture this is achieved with the second argument to `theory_wait`. Each propagator is set with its second argument the same logical variable (`Y` in the code) and the propagators are blocked on this second argument. When a constraint is posted, `Y` is instantiated, `Y = prop(_)`. This wakes all active propagators which either propagate or block again on the new variable. An alternative approach, which would invoke the propagators less frequently, would be to only wake up the activate propagators for those constraints that share a variable with the posted constraint.

3.3 Instantiation for Difference Logic

Thus far it has been demonstrated how theory propagation can be realised for SMT solvers over rational-tree unification and linear constraints, both of which are constraint systems that are built-in to Prolog. This begs the question of whether logical variables can be used to orchestrate theory propagation for a solver over a theory, such as difference constraints, that is not readily available in Prolog. The challenge is to find a way for efficiently deciding which theory constraints are entailed or disentailed, and then communicating this information to the SAT solver.

Difference constraints are a strict subclass of linear constraints in which each constraint has unary coefficients and is over at most two variables. To be precise, each constraint must take the form $x_i - x_j \leqslant c$, where x_i and x_j are variables, and c is a constant. A designated variable x_0 is interpreted as zero to encode



$$\begin{array}{cc} & \begin{array}{cccc} x_1 & x_2 & x_3 & x_4 \end{array} \\ \begin{array}{c} x_1 \\ x_2 \\ x_3 \\ x_4 \end{array} & \begin{bmatrix} 0 & \infty & -100 & \infty \\ -212 & 0 & \infty & \infty \\ \infty & \infty & 0 & 211 \\ \infty & 100 & \infty & 0 \end{bmatrix} \end{array} \qquad \begin{array}{cc} & \begin{array}{cccc} x_1 & x_2 & x_3 & x_4 \end{array} \\ \begin{array}{c} x_1 \\ x_2 \\ x_3 \\ x_4 \end{array} & \begin{bmatrix} -1 & 211 & -101 & 110 \\ -213 & -1 & -313 & -102 \\ 99 & 311 & -1 & 210 \\ -113 & 99 & -213 & -2 \end{bmatrix} \end{array}$$

Figure 7: Illustrating Floyd-Warshall

unary constraints, and the constants themselves are either integers, rationals or reals. Difference constraints are emblematic of other two variable systems and can be readily modified to richer domains such as octagons [94] (see Chapter 5). For this study we focus on integer difference logic. Integer difference logic (QF_IDL) is an SMT problem where integer difference constraints are composed with logical connectives.

3.3.1 The Floyd-Warshall algorithm

Decision procedures for systems of difference constraints are often obtained by viewing a system as a weighted directed graph. To illustrate, consider the constraints $(x_1 - x_2 \leqslant -212) \wedge (x_3 - x_1 \leqslant -100) \wedge (x_4 - x_3 \leqslant 211) \wedge (x_2 - x_4 \leqslant 100)$, which can be interpreted as the graph given in the left column of Figure 7. The graph in turn can be represented by the adjacency matrix given in the middle column. Solving the all pairs shortest path problem [43, 121] then populates the matrix with entries that describe the shortest paths between any two variables, as shown in the right column of Figure 6. Observe that the diagonals of this final matrix are negative, indicating that the graph contains negative cycles, showing that the system of constraints is inconsistent.

The Floyd-Warshall algorithm [43, 121], which is in $\mathcal{O}(n^3)$, solves the all pairs shortest path problem where n is the number of variables. Moreover, an incremental version can be formulated that is only $\mathcal{O}(n^2)$ for each new added constraint [10]. Both versions are shown in Figure 8. The non-incremental version compares all possible paths through the graph between each pair of variables x_i and x_j by

```
function
floyd-warshall(m, n)
for k = 1 to n
    for i = 1 to n
        for j = 1 to n
            m_{i,j} := min(m_{i,j}, m_{i,k} + m_{k,j})
        endfor
        if (m_{i,i} < 0)
            return UNSAT
        endif
    endfor
endfor
return SAT
```

```
function
floyd-warshall(m, n, x_p - x_q ⩽ c)
for i = 1 to n
    for j = 1 to n
        m_{i,j} := min(m_{i,j}, m_{i,p} + c + m_{q,j})
    endfor
    if (m_{i,i} < 0)
        return UNSAT
    endif
endfor
return SAT
```

Figure 8: Floyd-Warshall algorithm: non-incremental and incremental versions

checking whether the path between them can be shortened by passing through another variable x_k.

The incremental version takes a new constraint $x_p - x_q \leqslant c$, and checks whether the path between x_i and x_j could be reduced by travelling via the new edge that represents the constraint. That is, if the cost of travelling from x_i to x_p, then from x_p to x_q (a distance of c) and then from x_q to x_j, is less than that of moving from x_i to x_j directly. Observe that $m_{p,q}$ will be updated to c if $c < m_{p,q}$ when $i = p$ and $j = q$ since $m_{i,i} = 0$ and $m_{j,j} = 0$, assuming consistency. In both cases the consistency check is placed inside the main loop so that negative cycles cause rapid failure (though failure could be made faster at the expense of decomposing the innermost loop).

3.3.2 Theory propagation in difference logic

For rational-trees, a built-in test can be used within a wait declaration to block a goal until a rational-tree constraint is entailed or disentailed, which binds the Propositional variable that reifies the constraint when the goal resumes. It is less obvious how to program an analogous control structure for difference logic in order to realise theory propagation. The rest of this section explains how this control can be achieved.

3.3.2.1 Data-structures

The proposal is to shadow the Floyd-Warshall matrix with a square matrix of the same dimension that records which constraints in the SMT formula are possibly entailed or disentailed. This matrix, dubbed the watch matrix, is so named because it is inspired by watched literals [98] that are key to efficient SAT solving. The matrix is a control structure for efficiently identifying which Propositional variables should be bound, and to what truth values, when an entry in the Floyd-Warshall matrix is updated.

The watch matrix is constructed once. This matrix, in conjunction with the Floyd-Warshall matrix, constitutes the store. The Floyd-Warshall matrix maps each variable pair x-y (its indices) to a value $v \in \mathbb{Z} \cup \{\infty\}$ which is the length of the shortest known path from x to y (∞ used to indicate the absence of any such path). An entry of v can be interpreted as asserting the inequality $x - y \leqslant v$, which vacuously holds if $v = \infty$.

The watch matrix maps the indices x-y to a pair `EntL-DisL` where `EntL` and `DisL` are themselves lists of pairs, referred to as the entailed pairs and the disentailed pairs. Each of the entailed pairs takes the form `(c, Prop)` where `Prop` is a Propositional variable that reifies a constraint $x-y \leqslant c$ which occurs somewhere in the SMT formula. When the (x, y) entry is updated with v in the Floyd-Warshall matrix, the corresponding list `EntL` is traversed to find those pairs `(c, Prop)` for which $v \leqslant c$. Each pair corresponds to a constraint $x-y \leqslant c$ in the formula that is entailed and moreover reified with `Prop`. Thus `Prop` can be bound to `true` thereby achieving partial theory propagation.

Complete theory propagation is realised by additionally recording in `DisL` those

```
post_theory(Encoding, Store) :-
    build_store(Encoding, Store),
    setup_reify(Encoding, Queue),
    process_queue(Queue, Store).

build_store([], Store) :-
    empty_avl(Matrix), empty_avl(Watch),
    Store = store(Matrix, 0, Watch).
build_store([(X-Y =< C)-Prop | Rest], Store) :-
    build_store(Rest, StoreRest),
    StoreRest = store(Matrix1, N1, Watch1),
    Store = store(Matrix3, N3, Watch3),
    add_var(X, N1, Matrix1, N2, Matrix2),
    add_var(Y, N2, Matrix2, N3, Matrix3),
    (avl_fetch((X, Y), Watch1, EntL-DisL) ->
        avl_store((X, Y), Watch1, [(C, Prop) | EntL]-DisL, Watch2)
    ;
        avl_store((X, Y), Watch1, [(C, Prop)]-[], Watch2)
    ),
    NegC is -(C + 1),
    (avl_fetch((Y, X), Watch2, EntL2-DisL2) ->
        avl_store((Y, X), Watch2, EntL2-[(NegC, Prop) | DisL2], Watch3)
    ;
        avl_store((Y, X), Watch2, []-[(NegC, Prop)], Watch3)
    ).
```

Figure 9: Setting up the Floyd-Warshall matrix and the watch matrix

disentailed pairs `((-c - 1), Prop)` for which there exists a constraint $y - x \leqslant c$ in the SMT formula. When $v \leqslant -c - 1$ it follows that $x - y \leqslant -c - 1$, thus $y - x \leqslant c$ is disentailed since $y - x \leqslant c < c + 1 \leqslant y - x$. Disentailment is communicated to the SAT solver by setting `Prop` to `false`. Note that `Prop` may already be bound, though not necessarily to the same truth value, in which case inconsistency is detected. Detecting all disentailed constraints again only involves a list traversal and low-cost inequality checks.

```
setup_reify([], _).
setup_reify([((X-Y =< C), Prop) | Rest], Queue) :-
    bool_wait(Prop, X, Y, C, Queue),
    setup_reify(Rest, Queue).

:- block bool_wait(-, ?, ?, ?, ?).
bool_wait(Prop, X, Y, C, Queue) :-
    Prop == true, !,
    insert_queue(Queue, X, Y, C).
bool_wait(Prop, X, Y, C, Queue) :-
    Prop == false,
    NegC is -(C + 1),
    insert_queue(Queue, Y, X, NegC).

insert_queue(Queue, X, Y, C) :-
    var(Queue), !,
    Queue = [(X-Y =< C) | _Cons].
insert_queue([_Con | Cons], X, Y, C) :-
    insert_queue(Cons, X, Y, C).

:- block process_queue(-, ?).
process_queue(Queue, Store1) :-
    nonvar(Queue),
    Queue = [(X-Y =< C) | Cons],
    process_constraint(X-Y =< C, Store1, Store2),
    process_queue(Cons, Store2).
```

Figure 10: Propagating from the SAT solver to the theory solver

3.3.2.2 Setup

The watch matrix is set up in tandem with the Floyd-Warshall matrix by the `build_store` predicate (Figure 9). This predicate traverses the encoder map, considering each reified constraint `(X-Y =< C)-Prop` in turn. Both matrices are represented by AVL trees so that elements of these matrices are accessed and updated by `avl_fetch` and `avl_store` respectively. The body of `build_store` adds `(C, Prop)` to the list of entailed pairs in the watch matrix location `(X, Y)` and `(NegC, Prop)` (where `NegC` $= -$`C` $- 1$) to the list of disentailed pairs at `(Y,`

```
process_constraint(X-Y =< C, StoreIn, StoreOut) :-
    StoreIn = store(Matrix1, N, Watch),
    StoreOut = store(Matrix3, N, Watch),
    avl_fetch((X, Y), Matrix1, C_XY),
    min(C_XY, C, Min),
    (C == Min ->
        matrix_update((X, Y), C, Matrix1, Matrix2, Watch),
        floyd_warshall(N, Matrix2, Matrix3, Watch)
    ;
        Matrix3 = Matrix1
    ).

matrix_update(Key, Value, Matrix1, Matrix2, Watch) :-
    (avl_fetch(Key, Watch, EntL-DisL) ->
        true
    ;
        EntL = [], DisL = []
    ),
    entailed(EntL, Value),
    disentailed(DisL, Value),
    avl_store(Key, Matrix1, Value, Matrix2).

entailed([], _).
entailed([(C, Prop) | Rest], Min) :-
    (Min =< C -> Prop = true ; true),
    entailed(Rest, Min).
```

Figure 11: Propagating from the theory solver to the SAT solver

`X)`. The calls to the `add_var` predicate, when necessary, add a new row and column to the Floyd-Warshall matrix and populates the new diagonal element with zero and any other new entries with ∞.

3.3.2.3 Posting difference constraints

Reification provides a channel for passing information from the theory solver to the SAT solver. Conversely, when the SAT solver binds a Propositional variable that reifies a difference constraint, either the constraint, or its negation, must be posted (added) to the store. As a consequence of the update, incremental Floyd-Warshall should be applied together with any ensuing theory propagation.

Logical variables also provide a mechanism for coordinating these events, which must be fully backtrackable. This can be elegantly achieved with an open list that queues up the difference constraints that are to be posted to the store, as shown in Figure 10. The predicate `insert_queue` inserts a difference constraint at the end of `Queue`. The predicate `process_queue` blocks until a constraint is in the queue at which point the constraint is passed onto `process_constraint` that activates Floyd-Warshall, before inspecting, and if necessary blocking, until another element appears in the `Queue`. Observe how the store is updated as the constraints are processed. The predicate `process_constraint` (see Figure 11) invokes Floyd-Warshall, though only if the new constraint $x - y \leqslant c$ has a constant c that is strictly smaller than the value stored in the matrix at index (x, y). The update is performed by `matrix_update` which extracts two lists from the watch matrix: the entailed pairs and the disentailed pairs. The predicate `entailed` serves to illustrate how lightweight this form of theory propagation actually is once the watch matrix has been constructed. The predicate `disentailed` is defined analogously.

The predicate `bool_wait` which, recall, waits until a reification variable is set to a truth-value, in this setting merely pushes the constraint, or its negation, into the queue.

3.4 Experimental Results

3.4.1 Rational-tree solver

The DPLL(T) solver for rational-trees has been coded in SICStus Prolog 4.2.1, as described in Section 3.1. Henceforth this will be called the Prolog solver. To assess this solver it has been applied to a benchmark suite of 84 type recovery problems, its target application. The first eight benchmarks are drawn from compilations at different optimisation levels of three small programs manufactured to check their types against those derived by the solver. These benchmarks are designed to check that the inferred types match against those prescribed in the source file, and also assess the robustness of the type recovery in the face of various compilation modes. The remaining benchmarks are taken from version 8.9 of the coreutils suite of programs, standard UNIX command line utilities such as *wc*, *uniq*, *echo* etc. With an eye to the future, the DynInst toolkit [90] was used to parse the binaries and reconstruct the CFGs. This toolkit can recover the full CFG for many obfuscated, packed and stripped binaries, and even succeeds at determining most indirect jump targets. CFG recovery is followed by SSA conversion which, in turn, is followed by the generation of the type constraints, and the corresponding SMT formula complete with its Propositional skeleton. The latter rewriting steps are naturally realised as a set of Prolog rules.

At the time of publication [108], this work represented the first time that recursive types have been automatically derived, hence it was not possible to compare to previous approaches. Furthermore, no comparison is made with an open source SMT solver equipped with rational-trees since no such system exists. Nevertheless, to provide a comparative evaluation a lazy-basic SMT solver based on an off-the-shelf SAT solver, PicoSAT [14], has been constructed. This solver is also implemented in SICStus Prolog 4.2.1 but uses bindings to PicoSAT to solve the SAT formulae. PicoSAT, though small by comparison with some solvers at approximately 6000 lines of C, applies learning, random restarts, etc, a range of tactics not employed in the Prolog SAT solver. This SMT solver will henceforth be called the hybrid solver. However, crucially, the hybrid solver does not apply theory propagation; it simply alternates SAT solving with satisfiability testing following the lazy-basic approach, which is all one can do when the SAT solver is used as a black box.

The experiments were run on a single core of a MacBook Pro with a 2.4GHz Intel Core 2 Duo processor and 4GB of memory. A representative selection of the results are given in Table 2. The first column gives the binary from which the constraints were generated, the second column the number of instructions in the binary, the third the number of clauses in the problem, the fourth the number of Propositional variables, and the fifth the number of theory variables. In terms of timings, the sixth column records the runtime in seconds to find a model or a core for the Prolog solver, the seventh gives the number of times the Prolog SMT solver was called, the eighth gives the runtime in seconds to find a model or a core for the hybrid solver, and the final column gives the number of times the PicoSAT solver was called. To clarify, consider benchmark 1. The

Table 2: Benchmarking for a selection of type recovery problems

benchmark	insns	clauses	vars prop	vars theory	SMT time (s)	SMT calls	SAT time (s)	SAT calls
1 iter-sum.O1	296	2047	564	779	14.57	SAT	413.36	796
2 iter-sum.O2	312	2132	586	812	52.34	SAT	seg	
3 recu-sum.O1	302	2129	588	809	15.37	SAT	6382.50	998
4 mergesort.O0	480	3216	888	1220	585.89	70	seg	
5 mergesort.O1	387	2636	718	1011	20.05	SAT	1176.58	1720
6 mergesort.O2	395	2628	713	1017	20.30	SAT	805.93	860
7 mergesort.Os	444	3275	907	1244	>14400		seg	
8 mergesort.O3	2586	15696	3741	6670	1551.23	31	>14400	
9 false	3747	27645	5357	12957	19.46	51	3250.05	536
10 true	3747	27645	5357	12955	19.27	51	3247.02	536
11 tty	3825	28255	5417	13373	20.02	51	3509.06	552
12 sync	3901	28706	5571	13466	70.76	52	3607.01	553
15 hostid	3912	28973	5576	13634	62.70	52	3651.77	550
19 basename	4114	30125	5829	14212	69.48	53	3939.21	544
20 env	4016	29670	5589	13956	22.69	53	3914.54	544
22 uname	4074	31048	5653	15034	32.28	52	3676.94	534
23 cksum	4259	31973	5975	15370	101.85	52	4516.21	554
24 sleep	4442	32993	6343	15637	84.89	51	4876.85	566
29 echo	4310	33087	6064	15571	41.41	51	4723.52	564
30 nice	4397	33057	6000	15719	11.23	51	4907.31	581
33 nl	5719	43834	7692	21240	17.20	56	seg	
34 comm	5563	45401	7790	22797	108.09	53	10667.25	650
42 wc	6377	52105	8818	26713	93.91	52	12681.63	575
43 uniq	6595	52779	9013	27190	35.46	53	13281.49	581
51 join	7946	67168	10844	34688	85.93	60	>14400	
53 sha384sum	11612	78776	16419	36153	191.87	53	>14400	
54 cut	8173	68332	11248	36736	185.84	60	>14400	
58 ln	9369	83877	12668	44935	292.21	54	>14400	
61 getlimits	10797	92504	14856	47845	396.81	54	>14400	
66 timeout	12063	98544	16306	50019	126.79	53	>14400	
78 ptx	15919	141197	21850	76881	702.67	55	>14400	
89 mbslen	25895	257132	35148	148102	1935.12	56	>14400	

SMT formula is satisfiable, hence a core is not derived, and the problem is solved with just one call to the Prolog SMT solver. The hybrid solver also requires just one call but this, in turn, requires PicoSAT to be invoked 796 times, on all but the last occasion adding a single blocking clause to the Propositional skeleton. By way of contrast, benchmark 9 is unsatisfiable hence a core is computed that pinpoints a type conflict. The Prolog SMT solver is invoked 51 times to identify this core; the hybrid SMT solver requires exactly the same number of calls, hence the number is not repeated in the table. However, these 51 calls to the hybrid solver cumulatively require 536 invocations of PicoSAT. On occasions the hybrid solver terminated with a memory error[1], indicated by seg, invariably after several hours of computation. The fault is repeatable.

In addition to these timing results, the recursive types inferred for mergesort, as well as those for iterative-sum and recursive-sum, have been checked against the types prescribed in the source. The sum programs both build lists of integers but then traverse them in different ways. Another point not revealed from the table is that the largest benchmarks can take over 20 minutes to parse, reconstruct the CFG, perform SSA conversion and then generate the SMT formula. Thus the time required to solve the SMT formulae does not exceed the time required to generate them, at least for the Prolog solver.

3.4.2 Integer difference logic solver

One of the attractions of the current work is that new theories can be coded in Prolog and be integrated straightforwardly into the SMT solver via reification. For rational-trees a comparison was made against a hybrid solver using PicoSAT as a SAT engine. Since the theory of quantifier-free integer difference logic (QF_IDL) is a standard part of SMT packages, for this theory a more direct comparison between the solver presented and an off-the-shelf solver can be made. That said, off-the-shelf solvers deploy learning and random restarts, among other things, so as to not get lost in the search space, whereas Prolog difference logic solver does not even apply lookahead.

For this more demanding strength test, the Prolog-based SMT solver is benchmarked against the open-source CVC3 (version 2.4.1) and CVC4 (version 1.3) solvers, which both consist of many hundreds of thousands of lines of C++ code

[1] This bug has been fixed in the forthcoming SICStus 4.3, though at the time of writing the latest version available is 4.2.3

Table 3: Benchmarking for a selection of QF_IDL problems

benchmark	sat	p-vars	t-vars	cvc3	cvc4	Prolog
queen8-1.smt2	✓	352	9	288	702	**30**
toroidal_queen7-1.smt2	✓	434	8	132	99	**20**
queen9-1.smt2	✓	450	10	908	462	**120**
jobshop6-2-3-3-2-4-12.smt2	✓	252	25	36378	295	**160**
super_queen11-1.smt2	✓	834	12	356	296	**90**
queen12-1.smt2	✓	816	13	752	521	**140**
SortingNetwork4_live_bgmc002.smt2	✓	503	21	28	**17**	110
inf-bakery-invalid-4.smt2	✓	536	23	**20**	46	48050
super_queen12-1.smt2	✓	984	13	988	260	**130**
queen14-1.smt2	✓	1120	15	11445	**1810**	1850
LinearSearch_live_bgmc003.smt2	✓	673	36	28	**0**	430
queen15-1.smt2	✓	1290	16	14397	3272	**2150**
toroidal_queen13-1.smt2	✓	1586	14	3548	678	**630**
super_queen15-1.smt2	✓	1506	16	12109	**659**	5280
queen16-1.smt2	✓	1472	17	1728	767	**360**
super_queen16-1.smt2	✓	1704	17	50195	**3770**	12270
queen17-1.smt2	✓	1666	18	7876	1213	**850**
super_queen6-1.smt2	✗	264	7	64	77	**10**
jobshop4-2-2-2-4-4-11.smt2	✗	112	17	64	**40**	310
toroidal_queen6-1.smt2	✗	288	7	344	310	**30**
super_queen7-1.smt2	✗	354	8	216	368	**40**
super_queen8-1.smt2	✗	456	9	444	770	**110**
toroidal_queen8-1.smt2	✗	544	9	4680	**670**	740
jobshop6-2-3-3-2-4-9.smt2	✗	252	25	7288	**321**	>60000
super_queen9-1.smt2	✗	570	10	896	**128**	210
diamonds.11.3.i.a.u.smt2	✗	188	78	25526	**817**	>60000
toroidal_queen9-1.smt2	✗	738	10	14005	**2267**	3030
diamonds.16.2.i.a.u.smt2	✗	209	81	>60000	**49910**	>60000
diamonds.12.3.i.a.u.smt2	✗	205	85	57284	**1584**	>60000
diamonds.17.2.i.a.u.smt2	✗	222	86	>60000	>60000	>60000
toroidal_queen10-1.smt2	✗	880	11	>60000	**7346**	16490
jobshop8-2-4-4-4-4-12.smt2	✗	448	33	>60000	**184**	>60000
diamonds.10.5.i.a.u.smt2	✗	251	111	26782	**525**	>60000
DTP_k2_n35_c175_s4.smt2	✗	699	35	10317	**1076**	>60000
inf-bakery-mutex-7.smt2	✗	914	38	**132**	288	>60000

and have performed well in the SMT competitions [17]. Problems from the latest (2013 at time of publication) version of the SMT-LIB library of SMT benchmarks [7] were used for testing and evaluation. SMT-LIB provides benchmarks for many SMT theories, notably QF_IDL problems. Moreover, benchmarks from this suite are conveniently labelled according to whether they are satisfiable or not, making it straightforward to test the Prolog solver for correctness, even for unsatisfiable instances.

To read the instances, the Prolog solver was extended with a parser for the smtlib2 input language, written using flex and bison, that outputs an abstract syntax tree for an SMT instance represented as a single Prolog term. To resolve overloading on the equality operator, which can be interpreted either as logical bi-implication or as a relational arithmetical operator, the abstract syntax tree was traversed to infer types and thereby disambiguate the usage of equality terms.

The benchmarks include both industrial problems, and difficult crafted problems designed specifically to test the performance of a solver. Instances in the QF_IDL class were ranked according to size, and directed at the Prolog solver and at CVC3 and CVC4. CVC3 was timed using the unix *time* command, measuring overall runtime, while the runtime of CVC4 was measured using its stats command line option. Note that CVC4 sometimes reports a runtime of zero; this is not an error. One might suppose that the problem was solved using some heuristic before the search even began, but the true cause is unclear. The runtime of the Prolog solver was found using the statistics predicate, though it was only able to resolve the run time with a granularity of ten milliseconds. Table 3 gives a selection of the results taken from the first two hundred benchmarks, which have been pruned to remove any whose run time was less than fifty milliseconds for all three solvers (which removed 35 benchmarks before the table even started). Benchmarks with similar names and performance were also removed to make space for a wider range of instances.

3.5 Discussion

3.5.1 Rational-tree solver

The results in Table 2 demonstrate that an SMT solver equipped with an appropriate theory can be used to successful automate the recovery of recursive types, a problem not previously solved.

On no occasion is the hybrid solver faster than the Prolog solver, which suggests that a succinct implementation of theory propagation is more powerful than deploying an off-the-shelf SAT solver as a black box in combination with a hand-crafted theory solver using the lazy-basic approach.

It can be observed in Table 2 that many of the problems are unsatisfiable. For these problems an explanation for a type conflict is returned rather than a satisfying type assignment. As a strength test of the solver these problems are good since the exhaustive search required to demonstrate unsatisfiability is more demanding than the search for a first satisfying assignment. There are two results that require discussion. Benchmark 4 has an unsatisfiable core of 26 constraints, whereas most cores have less than 10 constraints. This explains why it is relatively slow. Benchmark 7 has timed out, a reminder that large SMT problems can be hard to solve. Efficiency could be improved by passing constraints to the SMT solver one x86 function at a time, gradually working over the call graph in a top-down manner, thus making the SMT problem more tractable (though increasing the number of calls to the SMT solver).

The time required for type recovery is sensitive to optimisation level. It is obvious that optimisations that increase code size and consequently the size of the SMT instance will take longer to solve, but the precise effect of individual optimisations is not clear. One might postulate that some optimisations could actually reduce the complexity of the problem, but this requires further investigation.

For the unsatisfiable problems, a core of unsatisfiable constraints is calculated using multiple calls to the DPLL(T) solver as indicated. This core can be used to diagnose unsatisfiability, in turn allowing the analysis to be refined to return

Instruction	Constraint
`mov A₁, rax₁`	$T_{A_1} = T_{rax_1}$
`add A₂, rax₁`	$\begin{pmatrix} T_{A_2} = \text{basic(\_,int,4)} \wedge T_{A_1} = T_{A_2} \wedge \\ T_{rax_1} = T_{A_2} \end{pmatrix} \vee$ $\begin{pmatrix} T_{A_2} = \text{ptr}(\langle [...], \alpha_1, [...] \rangle) \wedge \\ T_{A_1} = \text{ptr}(\langle [...], \alpha_2, [...] \rangle) \wedge \\ T_{rax_1} = \text{basic(\_,int,4)} \end{pmatrix} \vee$ $\begin{pmatrix} T_{A_2} = \text{ptr}(\langle [...], \alpha_3, [...] \rangle) \wedge \\ T_{A_1} = \text{basic(\_,int,4)} \wedge \\ T_{rax_1} = \text{ptr}(\langle [...], \alpha_4, [...] \rangle) \end{pmatrix}$
`mov A₃, [A₂]`	$T_{A_2} = \text{ptr}(\langle [...], T_{A_3}, [\_, \_, \_, ...] \rangle)$
`nop A₃`	

Table 4: Erroneous constraint generation

meaningful information despite the initial result. In the benchmarks unsatisfiability is often caused by incorrect intermediate code generation, or unusual/unforeseen cases in constraint generation. For example, `nop` instructions such as `nop [rax+rax+0x0]` appear in some binaries (this example is taken from the false binary). This instruction does nothing, but has been generated by the compiler with an encoded operand in order to make it a specific size for optimal performance/code size. The indirect addressing is broken down and constraints generated as shown in Table 4. The final constraint states that A_2 must have pointer type, hence those for the `add` dictate that one of A_1 and rax_1 must be of basic type, and the other a pointer; however, the first constraint says they have the same type, so the system is inconsistent.

Another unexpected source of inconsistency is the hard-coded pointer addresses sometimes found in `mov` instructions. These are often addresses of strings included in the binary, but also include constructor and destructor lists, added by the linker for construction and destruction of objects. For example, the instruction `mov ebx₁, 0x605e38` appears in the cksum binary, and moves the address of a string into ebx_1 resulting in the constraint $T_{ebx_1} = \text{basic(\_,int,4)}$. Later however, ebx_1 is dereferenced, which implies that it is a pointer, and conflicts with the earlier inference.

Quite apart from the disjunctive nature of the constraints, the sheer number of x86 instructions pose an engineering challenge when writing a type recovery tool; indeed the constraint generator module has taken longer to develop than both SMT solvers together. Moreover, as the above two examples illustrate, type conflicts stem from type interactions between different instructions which makes the conflicts difficult to anticipate. The result produced from the solver is either a successful recovery of types, or a core of inconsistent types, both of which can be achieved sufficiently quickly. Since the core is typically small, it is of great utility in pinpointing omissions in the type generation phase. It seems attractive to augment the solver with a domain specific language for expressing and editing the type constraints so that they can be refined, if necessary, by a user.

3.5.2 Integer difference logic solver

From the results in Table 3, together with the 35 benchmarks solved in less than 50ms by all solvers, it can be observed that the Prolog QF_IDL solver performs surprisingly well, solving many benchmark instances in times comparable to a well established SMT solver. We suspect that the performance stems partly from our incremental algorithm and partly from the watch matrices which enable lightweight theory propagation, though it is not straightforward to determine the relative importance of these techniques. The performance is particularly pleasing considering that the solver employs neither learning nor lookahead. Morever, this performance has to be balanced against the engineering effort required to implement and integrate the theory in the Prolog solver, which totalled less than six hundred lines of code. The brevity of the code also facilitates easy modification and experimentation, an advantage of any declarative language.

Timeouts occur with several of the benchmarks. The diamonds benchmarks, handcrafted by Ofer Strichman [113], are particularly hard for the Prolog solver, and indeed CVC3 and CVC4 both also timeout on several of these problems. These benchmarks are recognised as challenging problems [2], for which special tactics have been suggested [106], so it is no great surprise that they cause difficulty, particularly as this Prolog solver does not employ any labelling heuristics. The jobshop and DTP benchmarks are also hand-crafted problems designed to stress a solver.

3.6 Summary

This chapter has presented a DPLL(T) SMT solver coded in Prolog for three theories – rational-tree unification, linear arithmetic and integer difference constraints. The motivation for this work was the need for an SMT solver over rational-tree unification to recover types from x86 binaries; with Prolog providing a decision procedure for rational-tree unification the integration with the SAT solver in [64] is a natural development. The effectiveness of the approach has been demonstrated by the successful application of the solver to a suite of type recovery problems.

The integer difference solver performs surprisingly well, and it would certainly be worthwhile investigating what further improvements could be made in order to tackle harder problems. The search and labelling heuristics described in Section 2.3.2 provide a good starting point, and additional domain specific optimisations could also be made. The solver could also be extended to other similar domains, such as Octagons [94] (see Chapter 5).

Chapter 4

Semantics-Driven Decompilation of Recursive Datatypes

Leading on from the type reconstruction work of Chapters 2 and 3, this chapter poses, and attempts to answer, a new question: In the absence of source code to compare against, what does it mean for the recovered types to be correct? To confront this challenge a new and semantically-founded approach is presented that provides strong guarantees for the reconstructed types. Key to the approach is the derivation of a witness program in a high-level language alongside the reconstructed types. Since this witness has the same semantics as the binary, and is type correct by construction, it induces a (justifiable) type assignment on the binary. Moreover, this approach effectively yields a type-directed decompiler.

Specifically, this chapter offers a formalisation for the reversing of MinX, an abstraction of x86, to MinC, a type-safe dialect of C with recursive datatypes. The approach is implemented using Constraint Handling Rules (CHR) [45], and evaluated by compiling a range of textbook C programs illustrating various data-structures to MinX, and then recovering the original structures.

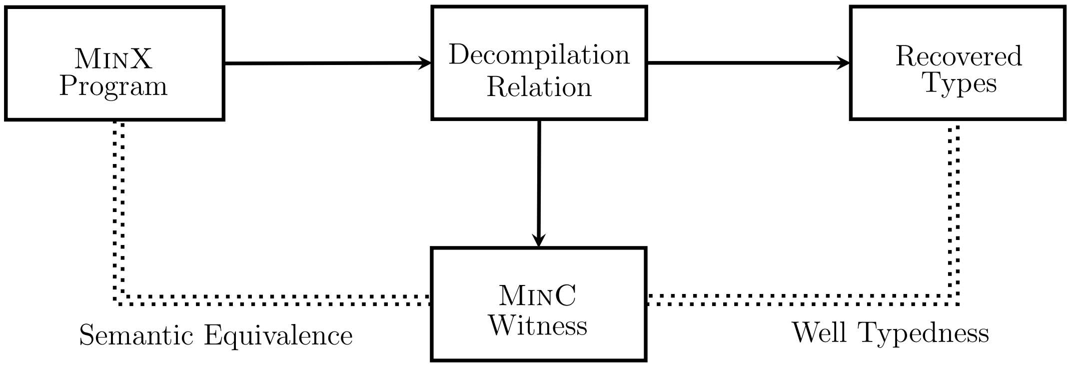


Figure 12: System overview

4.1 System overview

Figure 12 illustrates how the components of the new type recovery system fit together. Solid arrows indicate flow whereas dotted lines indicate bi-directional semantic connections.

The decompilation relation sits at the heart of the diagram. It relates an input MinX program to two outputs: the recovered types, and the MinC witness program. The latter is subservient to the former, since its role (in type recovery) is to justify the recovered types. The decompilation relation is exactly that, a mathematical relation, that specifies what it means for a MinX program to be in correspondence with a MinC program. Nevertheless, a solver can be constructed, in conformance with the relation, which, given the input MinX program, computes the two outputs. Hence the annotated direction of flow.

The semantic connection on the right indicates that the MinC witness program inhabits the recovered types; the connection on the left expresses that the MinX program is semantically equivalent to the witness, in the sense that their memories remain in sync. The whole construction semantically relates the MinX program to the recovered types; a connection that follows by transitive closure.

The structure of this chapter reflects the diagram; components are covered as the diagram is swept left-to-right. Starting on the left, Structured Operational

Semantics (SOS) for MinX programs are detailed in Section 4.2; followed by SOS for MinC programs in Section 4.3. With these semantic foundations in place, the decompilation (specification) relation is introduced in Section 4.4. Section 4.5 concerns the automation of the relation and Section 4.6 assesses the quality of the recovered types, on the right.

4.2 The MinX Language

A lean instruction set, called MinX (Minimal x86), is used to illustrate the semantic construction. MinX, by design, abstracts away from control-flow details that distract from the main goal of recovering recursive datatypes. MinX fixes a single calling convention, which avoids the need for argument detection techniques and control-flow recovery [42, 4]. The instruction set also supports an unbounded number of registers, which means that register spilling does not need to be considered and SSA conversion [119] can be avoided. These restrictions can be relaxed in actual binaries by preprocessing steps.

4.2.1 Memory

An important challenge is that presented by differently-sized values. At the binary level, the field of a structure is accessed by a byte offset, which depends on the sizes of the data objects that precede it. This layout problem is intrinsic to type recovery and therefore it must be assumed that memory is organised into bytes, and that data objects are permitted to straddle contiguous bytes. Thus we define $\mathit{Bit} = \{0, 1\}$ and let $\mathit{Byte} = \mathit{Bit}^8 \cup \{\bot\}$ where $\bot$ denotes a single byte of uninitialised (random) memory. A word is then a vector of bytes, that is, $\mathit{Word} = \mathit{Byte}^4$. The operator : concatenates vectors of bytes (and single bytes).

Register Bank The set of registers, each of which is of word size, is denoted $\mathit{Regs} = \{r_0, r_1, ...\}$. A function $R : \mathit{Regs} \to \mathit{Word}$ maps registers to the words they contain. To manipulate 1 byte and 2 byte objects within a register, we introduce an accessor function $R_{i:j} : \mathit{Regs} \to \mathit{Byte}^*$ which slices from byte i (counting from zero) up to but not including byte j of a given register r. This is defined by

$R_{i:j}(r) = \vec{b}'$ where $R(r) = \vec{b} : \vec{b}' : \vec{b}''$ and

$$\vec{b} \in \mathit{Byte}^{i} \qquad \vec{b}' \in \mathit{Byte}^{j-i} \qquad \vec{b}'' \in \mathit{Byte}^{4-j}$$

For the register map, R, we define a notion of partial update in which only the least significant $w = |\vec{b}|$ bytes of register r are updated with the bytes $\vec{b}$ as follows:

$$R \circ_w \{r \mapsto \vec{b}\} = R \circ \{r \mapsto \vec{b} : R_{w:4}(r)\}$$

Heap Memory The heap is modelled as a (partial) function $H : \mathit{Word} \rightharpoonup \mathit{Byte}$ and therefore is byte addressable. To read stored objects that straddle w consecutive bytes we define a function $H^w : \mathit{Word} \to \mathit{Byte}^w$ that reads and amalgamates w bytes of the heap into a single vector as follows:

$$H^w(a) = \begin{cases} \{\bot\}^w & \text{if } \bot \in a \\ H(a +_4 w -_4 1) : \cdots : H(a) & \text{otherwise} \end{cases}$$

where the operations $+_4$ and $-_4$ denote addition and subtraction in 4 byte bit-vector arithmetic, and 1 denotes a 4 byte bit-vector. Note that if the address supplied to H^w is uninitialised (contains $\bot$) then an unknown value ($\{\bot\}^w$) is returned. This is one way in which uninitialised values can cause $\bot$ to propagate. It is also worth noting that accesses to the heap can wrap, for example by reading 4 bytes from address $a = 2^{32} - 1$. This does not match the semantics of x86, where a bus error would occur due to the unaligned read, but exceptions are orthogonal to the type reconstruction problem and therefore not (currently) pertinent to MinX.

4.2.2 Syntax

The syntax of MinX programs consists of four syntactic categories. The first, $w ::= 2 \mid 4$, denotes the width, in bytes, of the primitive data objects that are supported

$$\boxed{\lambda_x; \vec{R} \vdash \langle H, R, b\rangle \xrightarrow{b} \langle H', R', b'\rangle}$$

$$\text{x-seq}\frac{\vec{R} \vdash \langle H, R, \iota\rangle \xrightarrow{\iota} \langle H', R'\rangle}{\lambda_x; \vec{R} \vdash \langle H, R, \iota; b\rangle \xrightarrow{b} \langle H', R', b\rangle} \qquad \text{x-goto}\frac{}{\lambda_x; \vec{R} \vdash \langle H, R, \mathsf{goto}\ a\rangle \xrightarrow{b} \langle H, R, \lambda_x(a)\rangle}$$

$$\text{x-if-true}\frac{\perp \notin R_{0:w}(r_i) \qquad R_{0:w}(r_i) \neq 0}{\lambda_x; \vec{R} \vdash \langle H, R, (\mathsf{if}_w\ r_i\ \mathsf{goto}\ a); b\rangle \xrightarrow{b} \langle H, R, \lambda_x(a)\rangle}$$

$$\text{x-if-false}\frac{\perp \notin R_{0:w}(r_i) \qquad R_{0:w}(r_i) = 0}{\lambda_x; \vec{R} \vdash \langle H, R, (\mathsf{if}_w\ r_i\ \mathsf{goto}\ a); b\rangle \xrightarrow{b} \langle H, R, b\rangle}$$

Figure 13: Structured Operational Semantics of MinX blocks (b)

by the instruction set. The second, ι, defines the instructions themselves:

$$\begin{array}{rlll|ll}
\iota ::= & \mathsf{mov}_w & r, c & & \mathsf{mov}_w & r_i, r_j \\
| & \mathsf{mov}_w & r_i, [r_j] & | & \mathsf{mov}_w & [r_i], r_j \\
| & \mathsf{mov}_w & r_i, [r_j + c] & | & \mathsf{mov}_w & [r_i + c], r_j \\
| & \mathsf{eq}_w & r_i, r_j, r_k & | & \mathsf{op}_w^{\oplus} & r_i, r_j * c \\
| & \mathsf{op}_w^{\otimes} & r_i, r_j & | & \mathsf{op}_w^{\otimes} & r_i, c \\
| & \mathsf{alloc} & r_i, r_j & | & \mathsf{alloc} & r_i, r_j * c \\
| & \mathsf{call} & r_u, a, \vec{r_v} & & &
\end{array}$$

where c denotes a numeric constant and $a \in \mathit{Word}$ is the location of the function that is to be invoked. (A Harvard architecture is assumed throughout). Square brackets indicate indirection. The instructions $\mathsf{op}_w^{\oplus}$ and $\mathsf{op}_w^{\otimes}$ are themselves parameterised by the categories $\oplus \in \{+, -\}$ and $\otimes \in \{+, -, *, /, \&, |, ...\}$, and the width w of their operands. The third category, b, defines blocks:

$$b ::= \iota; b \mid (\mathsf{if}_w\ r_i\ \mathsf{goto}\ a); b \mid \mathsf{goto}\ a \mid \mathsf{ret}$$

Observe that blocks are terminated by control instructions, but conditional jumps only arise within a block. The final category, d_x, defines how functions are declared:

$$d_x ::= \langle \overrightarrow{r_{arg}}, r_{ret}, \overrightarrow{r_{loc}}, \lambda_x, a_0 \rangle$$

where $\lambda_x : \mathit{Word} \rightharpoonup b$ is a partial mapping from addresses to blocks. Moreover, if

$$\mathsf{labels}_x(b) = \begin{cases} \{a\} & \text{if } b = \mathsf{goto}\ a \\ \{a\} \cup \mathsf{labels}_x(b') & \text{else if } b = (\mathsf{if}_w\ r_i\ \mathsf{goto}\ a); b' \\ \mathsf{labels}_x(b') & \text{else if } b = \iota; b' \\ \varnothing & \text{otherwise} \end{cases}$$

it is required that $\bigcup\{\mathsf{labels}_x(b) \mid a \mapsto b \in \lambda_x\} \subseteq dom(\lambda'_x)$ so that jump targets are contained within λ_x. Finally, $\overrightarrow{r_{arg}}$ and $\overrightarrow{r_{loc}}$ denote vectors of (distinct) registers and $a_0 \in dom(\lambda_x)$.

4.2.3 Structured Operational Semantics

The SOS for MinX are manifested in two related judgements: $\lambda_x; \vec{R} \vdash \langle H, R, b\rangle \xrightarrow{b} \langle H', R', b'\rangle$ and $\vec{R} \vdash \langle H, R, \iota\rangle \xrightarrow{\iota} \langle H', R'\rangle$. The former, presented in Figure 13, details the behaviour of blocks, and the latter, presented in Figure 14, of single instructions. Both are parameterised by a vector $\vec{R}$ of register assignments (needed solely in the proofs) to state that data accessible from these (shadowed) registers is not mutated by a call. The rules for blocks are straightforward, but note that x-if-true and x-if-false will get stuck if the decision register contains an uninitialised byte.

As noted previously, the arithmetic/logical and move instructions operate on a variable number of bytes w, and this manifests in a few different ways in the semantics. In the rule x-mov-rr the least significant w bytes of the register r_i are mutated while the remaining $4-w$ bytes are left intact. x-mov-ir copies the lowest w bytes from r_j into the w consecutive bytes at the heap address contained in r_i. Note that all four bytes of r_i must be initialised with an address, otherwise the computation will get stuck. Other rules must also account for uninitialised data; for example x-mov-ri dereferences r_j and copies the result into r_i, therefore, if r_j contains a single unintialised value, r_i must be set to four unitialised values. The instruction x-$\oplus$-r* is included to support pointer arithmetic, and takes an additional operand c, holding the size of the data objects. In this rule and others utilising constants, c denotes a vector of bytes, but the arrow is omitted for brevity.

Observe how x-alloc sets allocated memory to $\perp$, to indicate unitialised memory. If the size operand, register r_j, itself contains an uninitialised byte, then rule

$$\boxed{\vec{R} \vdash \langle H, R, \iota \rangle \xrightarrow{\iota} \langle H', R' \rangle}$$

$$\text{x-mov-rc}\ \frac{R' = R \circ_w \{r_i \mapsto c\}}{\vec{R} \vdash \langle H, R, \mathsf{mov}_w\ r_i, c \rangle \xrightarrow{\iota} \langle H, R' \rangle} \qquad \text{x-mov-rr}\ \frac{R' = R \circ_w \{r_i \mapsto R_{0:w}(r_j)\}}{\vec{R} \vdash \langle H, R, \mathsf{mov}_w\ r_i, r_j \rangle \xrightarrow{\iota} \langle H, R' \rangle}$$

$$\text{x-mov-ir}\ \frac{\begin{array}{c}\bot \notin H^4(R(r_i)) \\ H' = H \circ \{H^4(R(r_i)) +_4 n \mapsto R_{n:n+1}(r_j)\}_{n=0}^{w-1}\end{array}}{\vec{R} \vdash \langle H, R, \mathsf{mov}_w\ [r_i], r_j \rangle \xrightarrow{\iota} \langle H', R \rangle}$$

$$\text{x-mov-ri}\ \frac{R' = R \circ_w \{r_i \mapsto H^w(R(r_j))\}}{\vec{R} \vdash \langle H, R, \mathsf{mov}_w\ r_i, [r_j] \rangle \xrightarrow{\iota} \langle H, R' \rangle}$$

$$\text{x-mov-r+}\ \frac{R' = R \circ_w \{r_i \mapsto H^w(R(r_j) +_4 c)\}}{\vec{R} \vdash \langle H, R, \mathsf{mov}_w\ r_i, [r_j + c] \rangle \xrightarrow{\iota} \langle H, R' \rangle}$$

$$\text{x-mov-+r}\ \frac{\begin{array}{c}\bot \notin H^4(R(r_i)) \\ H' = H \circ \{H^4(R(r_i)) +_4 c +_4 n \mapsto R_{n:n+1}(r_j)\}_{n=0}^{w-1}\end{array}}{\vec{R} \vdash \langle H, R, \mathsf{mov}_w\ [r_i + c], r_j \rangle \xrightarrow{\iota} \langle H', R \rangle}$$

$$\text{x-eq-true}\ \frac{\begin{array}{c}R_{0:w}(r_j) = R_{0:w}(r_k) \\ R' = R \circ_w \{r_i \mapsto 1\}\end{array}}{\vec{R} \vdash \langle H, R, \mathsf{eq}_w\ r_i, r_j, r_k \rangle \xrightarrow{\iota} \langle H, R' \rangle} \qquad \text{x-eq-false}\ \frac{\begin{array}{c}R_{0:w}(r_j) \neq R_{0:w}(r_k) \\ R' = R \circ_w \{r_i \mapsto 0\}\end{array}}{\vec{R} \vdash \langle H, R, \mathsf{eq}_w\ r_i, r_j, r_k \rangle \xrightarrow{\iota} \langle H, R' \rangle}$$

$$\text{x-}\oplus\text{-r*}\ \frac{R' = R \circ_w \{r_i \mapsto R_{0:w}(r_i) \oplus_w (R_{0:w}(r_j) *_w c)\}}{\vec{R} \vdash \langle H, R, \mathsf{op}_w^{\oplus}\ r_i, r_j * c \rangle \xrightarrow{\iota} \langle H, R' \rangle}$$

$$\text{x-}\otimes\text{-rc}\ \frac{R' = R \circ_w \{r_i \mapsto R_{0:w}(r_i) \otimes_w c\}}{\vec{R} \vdash \langle H, R, \mathsf{op}_w^{\otimes}\ r_i, c \rangle \xrightarrow{\iota} \langle H, R' \rangle} \qquad \text{x-}\otimes\text{-rr}\ \frac{R' = R \circ_w \{r_i \mapsto R_{0:w}(r_i) \otimes_w R_{0:w}(r_j)\}}{\vec{R} \vdash \langle H, R, \mathsf{op}_w^{\otimes}\ r_i, r_j \rangle \xrightarrow{\iota} \langle H, R' \rangle}$$

$$\text{x-alloc}\ \frac{\begin{array}{c}\bot \notin R(r_j) \quad \{l, l +_4 1, \dots, l +_4 R(r_j) -_4 1\} \cap dom(H) = \varnothing \quad R' = R \circ \{r_i \mapsto l\} \\ H' = H \circ \{l \mapsto \{\bot\}^4, l +_4 1 \mapsto \{\bot\}^4, \dots, l +_4 R(r_j) -_4 1 \mapsto \{\bot\}^4\}\end{array}}{\vec{R} \vdash \langle H, R, \mathsf{alloc}\ r_i, r_j \rangle \xrightarrow{\iota} \langle H', R' \rangle}$$

$$\text{x-alloc-bot}\ \frac{\bot \in R(r_j) \quad R' = R \circ \{r_i \mapsto \{\bot\}^4\}}{\vec{R} \vdash \langle H, R, \mathsf{alloc}\ r_i, r_j \rangle \xrightarrow{\iota} \langle H, R' \rangle}$$

$$\text{x-alloc-*}\ \frac{\begin{array}{c}\bot \notin R(r_j) \quad \{l, l +_4 1, \dots, l +_4 R(r_j) *_4 c -_4 1\} \cap dom(H) = \varnothing \quad R' = R \circ \{r_i \mapsto l\} \\ H' = H \circ \{l \mapsto \{\bot\}^4, \dots, l +_4 (R(r_j) *_4 c) -_4 1 \mapsto \{\bot\}^4\}\end{array}}{\vec{R} \vdash \langle H, R, \mathsf{alloc}\ r_i, r_j * c \rangle \xrightarrow{\iota} \langle H', R' \rangle}$$

$$\text{x-alloc-*-bot}\ \frac{\bot \in R(r_j) \quad R' = R \circ \{r_i \mapsto \{\bot\}^4\}}{\vec{R} \vdash \langle H, R, \mathsf{alloc}\ r_i, r_j * c \rangle \xrightarrow{\iota} \langle H, R' \rangle}$$

$$\text{x-call}\ \frac{\begin{array}{c}\phi_x(a) = \langle \overrightarrow{r_{arg}}, r_{ret}, \overrightarrow{r_{loc}}, \lambda_x, a_0 \rangle \quad R' = \{\overrightarrow{r_{arg}} \mapsto \overrightarrow{r_j}, r_{ret} \mapsto \{\bot\}^4, \overrightarrow{r_{loc}} \mapsto \{\bot\}^4\} \\ \lambda_x; \vec{R}, R \vdash \langle H, R', \lambda_x(a_0) \rangle \xrightarrow{b}{}^{*} \langle H', R'', \mathsf{ret} \rangle\end{array}}{\vec{R} \vdash \langle H, R, \mathsf{call}\ r_i, a, \overrightarrow{r_j} \rangle \xrightarrow{\iota} \langle H', R \circ \{r_i \mapsto R''(r_{ret})\} \rangle}$$

Figure 14: Structured Operational Semantics of MinX instructions (ι)

x-alloc-bot must also set the destination register (r_i) to be uninitialised. The rule x-alloc-* is employed to allocate an array of objects, each of size c.

The x-call rule is worthy of note, as it is particularly unusual. To copy arguments to the callee function's register set, four bytes of each of the registers $\overrightarrow{r_j}$ are copied to $\overrightarrow{r_{arg}}$, which is abbreviated to $\overrightarrow{r_{arg}} \mapsto \overrightarrow{r_j}$ in the judgement. The callee's local registers $\overrightarrow{r_{loc}}$ and return register, r_{ret}, are set to uninitialised values. The entry block $\lambda_x(a_0)$ is executed, and any subsequent block that leads on from it, until `ret` is encountered, whereupon the register values are restored and register r_j updated with the return value.

4.3 The MinC Language

MinC (Minimal C) is the language of witness programs and the target of the decompilation. Even though C is expressive enough to capture the semantics of machine instructions, it lacks one crucial ingredient: type safety. In contrast, MinC is designed to be type-safe, while remaining close enough to C to recover MinX programs.

4.3.1 Syntax

MinC features three different kinds of types:

$$t ::= \mathsf{short} \mid \mathsf{long} \qquad \theta ::= t \mid \tau* \qquad \tau ::= \theta \mid \theta[] \mid N$$

A primitive type t is either a short or a long integer. A compact type θ is a type that can be assigned to a program variable, it is either a primitive type t or a pointer type $\tau*$. Finally, a general type τ is either a compact type θ, an array type $\theta[]$ or a struct type identified by its name N.

Structs are declared using their name and a list of types for the fields, $\vec{\theta}$, and may also be forward declared provided that they are fully defined elsewhere. The syntax for these declarations is given by decl:

$$\mathsf{decl} ::= \mathsf{struct}\ N \mid \mathsf{struct}\ N(\vec{\theta})$$

Rules for defining structs, including resolving forward references in mutually recursive structures using a placeholder mapping in decl, are detailed in Figure 15. Note that by construction arrays and structs may only exist on the heap, and may only encapsulate other structs and arrays by holding pointers to them, since their elements and fields must be in θ.

MinC programs themselves are defined in terms of the categories of statements s, expressions e and lvalues ℓ as follows:

$$\begin{array}{lll}
s ::= (\ell := e); s & \ell ::= x & e ::= \ell \mid c \mid f(\vec{e}) \\
\quad \mid (\textsf{if } e \textsf{ goto } l); s & \quad \mid *x & \quad \mid \textsf{new } \theta \mid \textsf{new } \theta[e] \\
\quad \mid \textsf{goto } l & \quad \mid x \to c & \quad \mid \textsf{new struct } N \\
\quad \mid \textsf{return} & \quad \mid x[e] & \quad \mid (e_1 \oplus e_2) \mid (e_1 \otimes e_2)
\end{array}$$

where $\oplus$ and $\otimes$ are defined as in MinX. Function declarations,

$$d_c ::= f(\overrightarrow{x : \theta})\langle \overrightarrow{y : \theta'}, l, \lambda_c, j \rangle$$

include a vector of arguments and their types $\overrightarrow{x : \theta}$, a vector of locals and their types $\overrightarrow{y : \theta'}$, an entry label $l \in \mathit{Labels}$, a partial mapping of labels to statements, $\lambda_c : \mathit{Labels} \rightharpoonup s$, and an index j into y indicating which local variable holds the return value. Labels must be contained within $\mathit{dom}(\lambda_c)$, analogously to the definition given for λ_x in MinX.

$$\boxed{\Sigma \vdash \theta} \quad \frac{}{\Sigma \vdash \textsf{short}} \quad \frac{}{\Sigma \vdash \textsf{long}} \quad \frac{\Sigma \vdash \tau}{\Sigma \vdash \tau *} \quad \frac{N \in \Sigma}{\Sigma \vdash N}$$

$$\boxed{\Sigma \vdash \textsf{decls} \xrightarrow{d} \Sigma'} \quad \frac{}{\Sigma \vdash \epsilon \xrightarrow{d} \Sigma} \quad \frac{N \notin \mathit{dom}(\Sigma) \quad \Sigma' = \Sigma \circ \{N \mapsto \bot\} \quad \Sigma' \vdash \textsf{decls} \xrightarrow{d} \Sigma''}{\Sigma \vdash \textsf{struct } N; \textsf{decls} \xrightarrow{d} \Sigma''}$$

$$\frac{\Sigma(N) = \bot \vee N \notin \mathit{dom}(\Sigma) \quad \Sigma' = \Sigma \circ \{N \mapsto \vec{\theta}\} \quad \forall \theta_i \in \vec{\theta}.(\Sigma' \vdash \theta_i) \quad \Sigma' \vdash \textsf{decls} \xrightarrow{d} \Sigma''}{\Sigma \vdash \textsf{struct } N(\vec{\theta}); \textsf{decls} \xrightarrow{d} \Sigma''}$$

Figure 15: Well-formed type declarations of MinC programs

$$\boxed{\Sigma \vdash d} \qquad \text{t-def}\,\frac{\Gamma = \{\overrightarrow{x:\theta}, \overrightarrow{y:\theta'}\} \quad l \in dom(\lambda_c) \quad y_j \in \vec{y} \quad \forall l' \in dom(\lambda_c):\ \Gamma;\Sigma \vdash \lambda_c(l')}{\Sigma \vdash f(\overrightarrow{x:\theta})\langle \overrightarrow{y:\theta'}, l, \lambda_c, j\rangle}$$

$$\boxed{\Gamma;\Sigma \vdash s} \qquad \text{t-assn}\,\frac{\Gamma;\Sigma \vdash \ell:\theta_1 \quad \Gamma;\Sigma \vdash e:\theta_2 \qquad \Sigma \vdash \theta_2 <: \theta_1 \quad \Gamma;\Sigma \vdash s}{\Gamma;\Sigma \vdash (\ell := e); s} \qquad \text{t-if}\,\frac{\Gamma;\Sigma \vdash e:\theta \qquad \Gamma;\Sigma \vdash s}{\Gamma;\Sigma \vdash (\mathsf{if}\ e\ \mathsf{goto}\ l); s}$$

$$\text{t-goto}\,\frac{}{\Gamma;\Sigma \vdash \mathsf{goto}\ l} \qquad \text{t-ret}\,\frac{}{\Gamma;\Sigma \vdash \mathsf{return}}$$

$$\boxed{\Gamma;\Sigma \vdash \ell:\theta} \qquad \text{t-var}\,\frac{x:\theta \in \Gamma}{\Gamma;\Sigma \vdash x:\theta} \qquad \text{t-ptr}\,\frac{\Gamma;\Sigma \vdash x:\tau*}{\Gamma;\Sigma \vdash *x:\tau}$$

$$\text{t-fld}\,\frac{\Gamma;\Sigma \vdash x:N* \qquad \Sigma(N) = \langle \theta_0, \dots, \theta_{n-1}\rangle}{\Gamma;\Sigma \vdash x \to i:\theta_i} \qquad \text{t-ar}\,\frac{\Gamma;\Sigma \vdash x:\theta[]* \qquad \Gamma;\Sigma \vdash e:t}{\Gamma;\Sigma \vdash x[e]:\theta}$$

$$\boxed{\Gamma;\Sigma \vdash e:\theta} \qquad \text{t-l}\,\frac{}{\Gamma;\Sigma \vdash c_l : \mathsf{long}} \qquad \text{t-s}\,\frac{}{\Gamma;\Sigma \vdash c_s : \mathsf{short}} \qquad \text{t-null}\,\frac{}{\Gamma;\Sigma \vdash 0_* : \tau*}$$

$$\text{t-new}\,\frac{}{\Gamma;\Sigma \vdash \mathsf{new}\ \theta : \theta*} \qquad \text{t-new-str}\,\frac{}{\Gamma;\Sigma \vdash \mathsf{new\ struct}\ N : N*}$$

$$\text{t-new-ar}\,\frac{\Gamma;\Sigma \vdash e:t}{\Gamma;\Sigma \vdash \mathsf{new}\ \theta[e] : \theta[]*} \qquad \text{t-}\otimes\,\frac{\Gamma;\Sigma \vdash e_1:t \quad \Gamma;\Sigma \vdash e_2:t}{\Gamma;\Sigma \vdash (e_1 \otimes e_2):t}$$

$$\text{t-ptr-}\oplus\,\frac{\Gamma;\Sigma \vdash e_1:\theta[]* \quad \Gamma;\Sigma \vdash e_2:t}{\Gamma;\Sigma \vdash (e_1 + e_2):\theta[]*} \qquad \text{t-+-ptr}\,\frac{\Gamma;\Sigma \vdash e_1:t \quad \Gamma;\Sigma \vdash e_2:\theta[]*}{\Gamma;\Sigma \vdash (e_1 + e_2):\theta[]*}$$

$$\text{t-call}\,\frac{\phi_c(f) = f(\overrightarrow{x:\theta})\langle \overrightarrow{y:\theta''}, l, \lambda_c, j\rangle \qquad \forall e_i \in \vec{e}, \theta'_i \in \vec{\theta'}:\ \Gamma;\Sigma \vdash e_i:\theta'_i \quad \Sigma \vdash \vec{\theta'} <: \vec{\theta} \quad \Sigma \vdash \theta''_j <: \theta_j}{\Gamma;\Sigma \vdash f(\vec{e}):\theta_j}$$

Figure 16: Type-correct MinC programs

4.3.2 Type System

Figure 16 defines the MinC type system as well-typing judgements $\Sigma \vdash d$, $\Gamma;\Sigma \vdash s$, $\Gamma;\Sigma \vdash \ell:\theta$ and $\Gamma;\Sigma \vdash e:\theta$ for the four syntactic sorts. As well as the variable type environment Γ, the judgements make use of Σ, which maps struct names N to vectors $\vec{\theta}$ of their field types. A global environment ϕ_c that contains all function definitions is also assumed.

$$\boxed{\Sigma \vdash \theta_1 <: \theta_2} \qquad \text{sub-refl}\frac{}{\Sigma \vdash \theta <: \theta} \qquad \text{sub-trans}\frac{\Sigma \vdash \theta_1 <: \theta_2 \quad \Sigma \vdash \theta_2 <: \theta_3}{\Sigma \vdash \theta_1 <: \theta_3}$$

$$\text{sub-ptr}\frac{\Sigma \vdash \theta_1 <: \theta_2}{\Sigma \vdash \theta_1 * <: \theta_2 *} \qquad \text{sub-arr}\frac{\Sigma \vdash \theta_1 <: \theta_2}{\Sigma \vdash \theta_1[] * <: \theta_2[] *}$$

$$\text{sub-elm}\frac{}{\Sigma \vdash \theta[] * <: \theta *} \qquad \text{sub-fld}\frac{\Sigma(N) = \langle \theta_0, \dots, \theta_{n-1} \rangle}{\Sigma \vdash N * <: \theta_0 *}$$

Figure 17: Subtyping relations of MinC programs

Because it admits type safety, the type system is stricter than that of C. For example, C allows any integer to be added to the address of a primitive type, which is not permitted in MinC. Unlike C, arrays are first-class types, which means that it is possible to pass them as arguments to functions and return them, without demoting them to simple pointers. The upshot of this is that in MinC it is possible to distinguish between $\theta*$ and $\theta[]*$.

To regain some of C's flexibility, without compromising on type safety, MinC's type system is equipped with subtyping (see Figure 17). For instance, an array $\theta[]*$ is a subtype of $\theta*$, which allows the assignment of an array to a compatible pointer.

As evident in the rules t-s and t-l, MinC literals are tagged: short values are denoted c_s and long values c_l. Although in general pointer literals do not exist in MinC, an exception is the special value 0_*, equivalent to `NULL` in C, and therefore 0_* has its own typing rule t-null.

4.3.3 Semantics

Figure 18 presents the semantics of MinC. The environment and store maps have signatures $\rho : Var \mapsto \mathbb{N}$ and $\sigma : \mathbb{N}_\perp \rightharpoonup Val_\perp$ where Var is a set of program variables and $\mathbb{N}$ is assumed to exclude 0. In addition, a set of non-empty ranges $\pi \subseteq \{[l, u] \mid 0 \leqslant l \leqslant u \leqslant 2^{32} - 1\}$ is included to represent a set of (disjoint) memory regions to which arrays or structs have been allocated.

The set π enables memory safe pointer arithmetic on arrays, by use of some auxiliary functions: The functions $\mathsf{untag}_t(n_t) = n$ and $\mathsf{tag}_t(n) = n_t$ remove and add a tag t from a value n. Addition of an address v and a short u is then defined using $+_\pi$ as follows:

$$v +_\pi u = \begin{cases} \bot & \text{if } v = \bot \vee u = \bot \\ \mathsf{tag}_*(s) & \text{if } \exists r \in \pi.[\mathsf{untag}_*(v), s] \subseteq r \\ & \quad \text{where } s = \mathsf{untag}_*(v) +_4 \mathsf{untag}_s(u) \\ \mathsf{err} & \text{otherwise} \end{cases}$$

This checks that the interval $[\mathsf{untag}_*(v), s]$, which starts at the address of the original pointer and ends at the result of the addition, is a subset of some existing interval (allocated memory range) r in π. This ensures that the result must fall within the same range of π as v, otherwise an error state occurs. Note that if v or u is unitialised ($\bot$) this is propagated. Addition of short/address, address/long, and long/address are analogously defined. The sum of two longs is simply defined:

$$u +_\pi v = \begin{cases} \bot & \text{if } u = \bot \vee v = \bot \\ \mathsf{tag}_l(\mathsf{untag}_l(u) +_4 \mathsf{untag}_l(v)) & \text{otherwise} \end{cases}$$

The sum of two shorts is defined likewise, to cumulatively give the partial map $+_\pi : \mathit{Val}^2 \rightharpoonup \mathit{Val}$, where Val is the set of all tagged values. Subtraction is defined likewise in a piecewise manner.

The SOS is structured according to the three syntactic categories: ℓ resolves lvalues to addresses, e resolves expressions to values, and statements s.

The rules l-ptr, l-fld and l-ar check $\rho(x) \neq 0$ since location 0 is reserved as a sentinel. If these checks fail then auxiliary rules, l-ptr-err, l-fld-err and l-ar-err, trigger an err state. For instance:

$$\text{l-ptr-err}\frac{\rho(x) = 0}{\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, *x \rangle \xrightarrow{\ell} \mathsf{err}}$$

Rules l-fld-err and l-ar-err are defined analogously. Moreover, the rules l-fld and l-ar check $a \in \cup\pi'$ to ensure the computed address a stays within an allocated memory region. Further complementary rules trigger err if this does not hold, indeed many rules have many error modes. The e-call rule is an extreme case: it has one error mode for the evaluation of each expression e_i passed as a function parameter, and execution of the block labelled $\lambda_c(l)$ can itself trigger an err.

$$\boxed{\Sigma;\vec{\rho};\rho \vdash \langle \sigma,\pi,\ell\rangle \xrightarrow{\ell} \langle \sigma',\pi',a\rangle + \mathsf{err}}$$

$$\text{l-var}\ \frac{}{\Sigma;\vec{\rho};\rho \vdash \langle \sigma,\pi,x\rangle \xrightarrow{\ell} \langle \sigma,\pi,\rho(x)\rangle} \qquad \text{l-ptr}\ \frac{\rho(x) \neq 0}{\Sigma;\vec{\rho};\rho \vdash \langle \sigma,\pi,*x\rangle \xrightarrow{\ell} \langle \sigma,\pi,\sigma(\rho(x))\rangle}$$

$$\text{l-fld}\ \frac{\rho(x) \neq 0 \quad a = \sigma(\rho(x)) +_{\bot} c \quad a \in \cup\pi}{\Sigma;\vec{\rho};\rho \vdash \langle \sigma,\pi,x \to c\rangle \xrightarrow{\ell} \langle \sigma,\pi,a\rangle} \qquad \text{l-ar}\ \frac{\begin{array}{c}\Sigma;\vec{\rho};\rho \vdash \langle \sigma,\pi,e\rangle \xrightarrow{e} \langle \sigma',\pi',v\rangle \\ \rho(x) \neq 0 \quad a = \sigma(\rho(x)) +_{\bot} v \quad a \in \cup\pi'\end{array}}{\Sigma;\vec{\rho};\rho \vdash \langle \sigma,\pi,x[e]\rangle \xrightarrow{\ell} \langle \sigma',\pi',a\rangle}$$

$$\boxed{\Sigma;\vec{\rho};\rho \vdash \langle \sigma,\pi,e\rangle \xrightarrow{e} \langle \sigma',\pi',v\rangle + \mathsf{err}} \qquad \text{e-const}\ \frac{}{\Sigma;\vec{\rho};\rho \vdash \langle \sigma,\pi,c\rangle \xrightarrow{e} \langle \sigma,\pi,c\rangle}$$

$$\text{e-lval}\ \frac{\Sigma;\vec{\rho};\rho \vdash \langle \sigma,\pi,\ell\rangle \xrightarrow{\ell} \langle \sigma',\pi',a\rangle}{\Sigma;\vec{\rho};\rho \vdash \langle \sigma,\pi,\ell\rangle \xrightarrow{e} \langle \sigma',\pi',\sigma'(a)\rangle} \qquad \text{e-op}\ \frac{\begin{array}{c}\Sigma;\vec{\rho};\rho \vdash \langle \sigma,\pi,e_1\rangle \xrightarrow{e} \langle \sigma',\pi',v_1\rangle \\ \Sigma;\vec{\rho};\rho \vdash \langle \sigma',\pi',e_2\rangle \xrightarrow{e} \langle \sigma'',\pi'',v_2\rangle \\ v_1 \otimes_{\pi''} v_2 = v\end{array}}{\Sigma;\vec{\rho};\rho \vdash \langle \sigma,\pi,(e_1 \otimes e_2)\rangle \xrightarrow{e} \langle \sigma'',\pi'',v\rangle}$$

$$\text{e-new}\ \frac{\begin{array}{c} a \notin dom(\sigma) \cup \{0\} \\ \sigma' = \sigma \circ \{a \mapsto \bot\}\end{array}}{\Sigma;\vec{\rho};\rho \vdash \langle \sigma,\pi,\mathsf{new}\ \theta\rangle \xrightarrow{e} \langle \sigma',\pi,a\rangle} \qquad \text{e-str}\ \frac{\begin{array}{c}\{a,\dots,a+|\Sigma(N)|-1\} \cap (dom(\sigma) \cup \{0\}) = \varnothing \\ \pi' = \pi \cup \{[a, a+|\Sigma(N)|-1]\} \\ \sigma' = \sigma \circ \{a \mapsto \bot, \dots, a+|\Sigma(N)|-1 \mapsto \bot\}\end{array}}{\Sigma;\vec{\rho};\rho \vdash \langle \sigma,\pi,\mathsf{new\ struct}\ N\rangle \xrightarrow{e} \langle \sigma',\pi',a\rangle}$$

$$\text{e-ar}\ \frac{\begin{array}{c}\Sigma;\vec{\rho};\rho \vdash \langle \sigma,\pi,e\rangle \xrightarrow{e} \langle \sigma',\pi',v\rangle \quad \{a,\dots,a+v-1\} \cap (dom(\sigma') \cup \{0\}) = \varnothing \\ \sigma'' = \sigma' \circ \{a \mapsto \bot, \dots, a+v-1 \mapsto \bot\} \quad \pi'' = \pi' \cup \{[a, a+v-1]\}\end{array}}{\Sigma;\vec{\rho};\rho \vdash \langle \sigma,\pi,\mathsf{new}\ \theta[e]\rangle \xrightarrow{e} \langle \sigma'',\pi'',a\rangle}$$

$$\text{e-call}\ \frac{\begin{array}{c}\phi_c(f) = f(\overrightarrow{x:\theta})\langle \overrightarrow{y:\theta'}, l, \lambda_c, j\rangle \quad \forall e_i \in \vec{e}:\ \Sigma;\vec{\rho};\rho \vdash \langle \sigma_{i-1},\pi_{i-1},e_i\rangle \xrightarrow{e} \langle \sigma_i,\pi_i,v_i\rangle \quad |\,\vec{x}\,| = n \\ \{\vec{a},\vec{a'}\} \cap (dom(\sigma_n) \cup \{0\}) = \varnothing \quad \rho' = \{\overrightarrow{x \mapsto a}, \overrightarrow{y \mapsto a'}\} \quad \sigma' = \sigma_n \circ \{\overrightarrow{a \mapsto v}, \overrightarrow{a' \mapsto \bot}\} \\ \Sigma;\lambda_c;\vec{\rho},\rho;\rho' \vdash \langle \sigma',\pi_n,\lambda_c(l)\rangle \xrightarrow{s}{}^{*} \langle \sigma'',\pi',\mathsf{return}\rangle\end{array}}{\Sigma;\vec{\rho};\rho \vdash \langle \sigma_0,\pi_0,f(\vec{e})\rangle \xrightarrow{e} \langle \sigma'',\pi',\sigma''(\rho'(y_j))\rangle}$$

$$\boxed{\Sigma;\lambda_c;\vec{\rho};\rho \vdash \langle \sigma,\pi,s\rangle \xrightarrow{s} \langle \sigma',\pi',s'\rangle + \mathsf{err}} \qquad \text{s-assn}\ \frac{\begin{array}{c}\Sigma;\vec{\rho};\rho \vdash \langle \sigma,\pi,\ell\rangle \xrightarrow{\ell} \langle \sigma',\pi',a\rangle \\ \Sigma;\vec{\rho};\rho \vdash \langle \sigma',\pi',e\rangle \xrightarrow{e} \langle \sigma'',\pi'',v\rangle \\ \sigma''' = \sigma'' \circ \{a \mapsto v\}\end{array}}{\Sigma;\lambda_c;\vec{\rho};\rho \vdash \langle \sigma,\pi,(\ell := e);s\rangle \xrightarrow{s} \langle \sigma''',\pi'',s\rangle}$$

$$\text{s-goto}\ \frac{l \in dom(\lambda_c)}{\Sigma;\lambda_c;\vec{\rho};\rho \vdash \langle \sigma,\pi,\mathsf{goto}\ l\rangle \xrightarrow{s} \langle \sigma,\pi,\lambda_c(l)\rangle}$$

$$\text{s-if-true}\ \frac{l \in dom(\lambda_c) \quad v \neq \bot \quad v \neq 0 \quad \Sigma;\vec{\rho};\rho \vdash \langle \sigma,\pi,e\rangle \xrightarrow{e} \langle \sigma',\pi',v\rangle}{\Sigma;\lambda_c;\vec{\rho};\rho \vdash \langle \sigma,\pi,(\mathsf{if}\ e\ \mathsf{goto}\ l);s\rangle \xrightarrow{s} \langle \sigma',\pi',\lambda_c(l)\rangle}$$

$$\text{s-if-false}\ \frac{l \in dom(\lambda_c) \quad v \neq \top \quad v = 0 \quad \Sigma;\vec{\rho};\rho \vdash \langle \sigma,\pi,e\rangle \xrightarrow{e} \langle \sigma',\pi',v\rangle}{\Sigma;\lambda_c;\vec{\rho};\rho \vdash \langle \sigma,\pi,(\mathsf{if}\ e\ \mathsf{goto}\ l);s\rangle \xrightarrow{s} \langle \sigma',\pi',s\rangle}$$

Figure 18: Structured Operational Semantics of MinC programs

Of particular note is e-op: if v_1 is tagged as a pointer and v_2 as an integer, the binary operation $v_1 \oplus_{\pi''} v_2$ itself will err if the resulting pointer falls outside the region enclosing v_1. This is trapped by a rule that complements e-op so as to propagate err in the expected way.

Finally note how freshly allocated memory is marked as uninitialised in rules e-new, e-ar and e-str.

4.3.4 Type Safety

MinC is a type-safe variant of C. This means that well-typed MinC programs do not get stuck. We can be formally precise about this property in the usual way, stating preservation and progress properties for the syntactic sorts of MinC.

Type safety depends on the notion of a store typing Ψ that associates a type θ with every address a in the store σ. A store σ is well-typed, denoted $\Sigma; \Psi \vdash \sigma; \pi$, iff

$$\forall (a : \theta) \in \Psi \;.\; \Sigma; \Psi; \sigma; \pi \vdash a : \theta$$

Figure 19 defines the auxiliary judgements for well-typed addresses and values. Similar to a store, an environment ρ is well-typed, denoted $\Gamma; \Sigma; \Psi \vdash \rho$, iff

$$\forall (x : \theta) \in \Gamma \;.\; \Sigma; \Psi \vdash \rho(x) : \theta * \wedge \rho(x) \neq 0$$

Proposition 2 below asserts preservation of type safety for MinC expressions. To be precise, it states that, given a consistent local environment ρ, store $\sigma; \pi$, expression e, and a store typing Ψ, the evaluation of e to yield v will preserve type safety. That is, under a new store typing Ψ', the environment ρ and new store $\sigma'; \pi'$ are well typed, as is the result of the evaluation, v.

Proposition 2 (Preservation of MinC Expressions)**.** If $\Gamma; \Sigma; \Psi \vdash \rho$, $\Sigma; \Psi \vdash \sigma; \pi$, $\Gamma; \Sigma \vdash e : \theta$ and $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, e \rangle \xrightarrow{e} \langle \sigma', \pi', v \rangle$ then for some $\Psi' \supseteq \Psi$:

$$\Gamma; \Sigma; \Psi' \vdash \rho \quad \wedge \quad \Sigma; \Psi' \vdash \sigma'; \pi' \quad \wedge \quad \Sigma; \Psi' \vdash v : \theta$$

Proposition 3 asserts that well-typed MinC expressions make progress. If the local environment ρ, store $\sigma; \pi$, and expression e are well typed, then evaluating e will

$$\boxed{\Sigma; \Psi; \sigma; \pi \vdash a : \tau} \quad \text{st-comp}\frac{\Sigma; \Psi \vdash \sigma(a) : \theta}{\Sigma; \Psi; \sigma; \pi \vdash a : \theta}$$

$$\text{st-fld}\frac{\Sigma(N) = \langle \theta_0, \ldots, \theta_{n-1} \rangle \quad [a, a+n-1] \in \pi \quad \forall i \in [0, n-1]. \Sigma; \Psi \vdash \sigma(a+i) : \theta_i}{\Sigma; \Psi; \sigma; \pi \vdash a : N} \quad \text{st-ar}\frac{[a-n, a+m] \in \pi \quad \forall i \in [-n, m]. \Sigma; \Psi \vdash \sigma(a+i) : \theta}{\Sigma; \Psi; \sigma; \pi \vdash a : \theta[]}$$

$$\boxed{\Sigma; \Psi \vdash v : \theta} \quad \text{vt-bot}\frac{}{\Sigma; \Psi \vdash \bot : \theta} \quad \text{vt-s}\frac{}{\Sigma; \Psi \vdash c_s : \mathsf{short}} \quad \text{vt-l}\frac{}{\Sigma; \Psi \vdash c_l : \mathsf{long}}$$

$$\text{vt-null}\frac{}{\Sigma; \Psi \vdash 0_l : \tau*} \quad \text{vt-addr}\frac{(a : \tau) \in \Psi}{\Sigma; \Psi \vdash a : \tau*} \quad \text{vt-subt}\frac{\Sigma; \Psi \vdash v : \theta_1 \quad \Sigma \vdash \theta_1 <: \theta_2}{\Sigma; \Psi \vdash v : \theta_2}$$

Figure 19: Well-typed addresses and values

either produce a value v and new state $\sigma'; \pi'$ or an error err:

Proposition 3 (Progress of MinC Expressions)**.** If $\Gamma; \Sigma; \Psi \vdash \rho$, $\Sigma; \Psi \vdash \sigma; \pi$ and $\Gamma; \Sigma \vdash e : \theta$ then $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, e \rangle \xrightarrow{e} \langle \sigma', \pi', v \rangle$ or $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, e \rangle \xrightarrow{e} \mathsf{err}$.

Due to the big-step nature of the MinC semantics, it is not possible to guarantee that non-terminating programs do not get stuck. Non-termination is subsumed by the err case of Proposition 9, for progress of functions:

Proposition 9 (Progress for MinC functions)**.**
If

- $\Sigma \vdash f(\overrightarrow{x : \theta}) \langle \overrightarrow{y : \theta'}, l, \lambda_c, j \rangle$
- $\Sigma; \lambda_c; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \lambda_c(l) \rangle \xrightarrow{s}{}^{*} \langle \sigma', \pi', \mathsf{return} \rangle$
- $\Gamma = \{\overrightarrow{x : \theta}, \overrightarrow{y : \theta'}\}$
- $\Gamma; \Sigma; \Psi \vdash \rho$
- $\Sigma; \Psi \vdash \sigma; \pi$

then

1. $\Sigma; \lambda_c; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \lambda_c(l) \rangle \xrightarrow{s}{}^{*} \langle \sigma', \pi', \mathsf{return} \rangle$ or

2. $\Sigma; \lambda_c; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \lambda_c(l) \rangle \xrightarrow{s}{}^{*} \mathsf{err}$ (assuming this subsumes divergence).

Similar Propositions for preservation and progress of lvalues, statements and functions appear alongside all proofs in appendix A.1.1.

4.4 Decompilation Relation

This section relates MinX and MinC programs through a decompilation relation. The relation is spread out over three judgements, one for each of the three main syntactic categories of MinX.

Instruction Decompilation Figures 20 and 21 define the decompilation judgement $\mu_\Gamma; \Gamma; \Sigma \vdash \iota \overset{\iota}{\rightsquigarrow} \ell := e$ which explains how to decompile a MinX instruction ι into a MinC assignment $\ell := e$. Figure 20 details mov instructions, and Figure 21 other instructions. Additional parameters to the judgement are a variable mapping μ_Γ that relates MinX registers to MinC local variables, a MinC typing environment Γ for those local variables, and a set of MinC struct definitions Σ.

The judgement is defined by syntax-directed rules, as is usually the case for compilation relations. The main difference is that, as the latter usually define (partial) functions; they are deterministic with typically one rule per syntactic construct. In contrast the decompilation judgement is non-deterministic and features multiple rules per syntactic construct, one for each distinct typing that can be assigned to the instruction. For instance, the three rules tr-mov-ri$_1$, tr-mov-ri$_2$ and tr-mov-ri$_3$ decompile instruction $\mathsf{mov}_w\ r_i, [r_j]$ into either $x := *y$, $x := y[0]$ or $x := y \to 0$ depending on whether y has type $\theta*$, $\theta[]*$ or $N*$.

The instruction widths w play an important role in restricting the possibilities for the recovered MinC types. For instance, rule tr-$\oplus$-rc ensures that the width w of the arithmetic operation $\mathsf{op}_w^{\otimes}$ is identical to the size of the recovered primitive type t. Hence, a width of 4 gives rise to long and a width of 2 to short. An auxilliary function $\mathit{sizeof}(\theta)$ gives the size of an object of type θ in bytes:

$$\mathit{sizeof}(\mathsf{short}) = 2 \qquad \mathit{sizeof}(\mathsf{long}) = 4 \qquad \mathit{sizeof}(\tau*) = 4$$

On several occasions the rules must make up for the difference in memory granularity between MinX and MinC. In particular, in MinC the stride between array elements is always 1. However, in MinX, the stride depends on the size of the elements. Hence, rules such as tr-$\oplus$-rc for array pointer arithmetic convert between a MinX stride of $c = m * \mathit{sizeof}(\theta)$ and a corresponding MinC stride of m.

When allocating a statically known amount of memory the rules tr-alloc-rc$_1$, tr-alloc-rc$_2$ and tr-alloc-rc$_3$ also exploit the sizes of types to determine whether a primitive type, a particular struct or an array is allocated. The decompilation of dynamic memory allocation is only supported for arrays (rule tr-alloc-r$*$), where it can be statically verified that the amount of allocated memory is a multiple of the size of the array elements. This rule, amongst others, make the decompilation relation conservative and incomplete. It is a price gladly paid in order to provide strong guarantees about the validity of the recovered types.

Basic Block Decompilation The top half of Figure 22 defines the judgement $\mu_\lambda; \mu_\Gamma; \Gamma; \Sigma \vdash b \overset{b}{\rightsquigarrow} s$ for decompiling MinX basic blocks b into MinC statements s. This judgement has one additional parameter compared to the judgement for instructions: the label map μ_λ relates MinX labels to the corresponding MinC labels. This map is used for decompiling `goto` and `if`. As the rules preserve the basic control flow from MinX to MinC and do not affect the types directly, they are deterministic and syntax-directed.

Function Definition Decompilation The bottom half of Figure 22 defines the judgement $\Sigma \vdash d_x \rightsquigarrow d_c$ that decompiles a MinX function definition d_x into a MinC definition d_c. The single rule sets up the variable and label maps, and decompiles the basic blocks with respect to an appropriate typing environment.

4.4.1 Meta-Theoretical Properties

The decompilation relation satisfies two strong properties that justify its relevance: 1) the produced MinC witness program is well-typed, and 2) the witness has the same operational semantics as the original MinX program. Taken together these two properties give meaning to the statement that the original MinX program inhabits the recovered types.

$$\boxed{\mu_\Gamma; \Gamma; \Sigma \vdash \iota \overset{\iota}{\rightsquigarrow} \ell := e}$$

$$\text{tr-mov-rc}\ \frac{\begin{array}{cc}(r_i : x)_w \in \mu_\Gamma & (x : t) \in \Gamma \\ \Gamma; \Sigma \vdash c : t & \textit{sizeof}(t) = w\end{array}}{\mu_\Gamma; \Gamma; \Sigma \vdash \mathsf{mov}_w\ r_i, c \overset{\iota}{\rightsquigarrow} x := c}$$

$$\text{tr-mov-r0}\ \frac{(r_i : x)_4 \in \mu_\Gamma \qquad (x : \tau*) \in \Gamma}{\mu_\Gamma; \Gamma; \Sigma \vdash \mathsf{mov}_4\ r_i, 0 \overset{\iota}{\rightsquigarrow} x := 0}$$

$$\text{tr-mov-rr}\ \frac{\begin{array}{c}(r_i : x)_w \in \mu_\Gamma \qquad (r_j : y)_w \in \mu_\Gamma \qquad (x : \theta_1), (y : \theta_2) \in \Gamma \\ \textit{sizeof}(\theta_1) = \textit{sizeof}(\theta_2) = w \qquad \Sigma \vdash \theta_2 <: \theta_1\end{array}}{\mu_\Gamma; \Gamma; \Sigma \vdash \mathsf{mov}_w\ r_i, r_j \overset{\iota}{\rightsquigarrow} x := y}$$

$$\text{tr-mov-ri}_1\ \frac{\begin{array}{c}(r_i : x)_w, (r_j : y)_4 \in \mu_\Gamma \qquad (x : \theta_1), (y : \theta_2*) \in \Gamma \\ \Sigma \vdash \theta_2 <: \theta_1 \qquad \textit{sizeof}(\theta_1) = \textit{sizeof}(\theta_2) = w\end{array}}{\mu_\Gamma; \Gamma; \Sigma \vdash \mathsf{mov}_w\ r_i, [r_j] \overset{\iota}{\rightsquigarrow} x := *y}$$

$$\text{tr-mov-ri}_2\ \frac{\begin{array}{c}(r_i : x)_w, (r_j : y)_4 \in \mu_\Gamma \qquad (x : \theta_1), (y : \theta_2[]*) \in \Gamma \\ \Sigma \vdash \theta_2 <: \theta_1 \qquad \textit{sizeof}(\theta_1) = \textit{sizeof}(\theta_2) = w\end{array}}{\mu_\Gamma; \Gamma; \Sigma \vdash \mathsf{mov}_w\ r_i, [r_j] \overset{\iota}{\rightsquigarrow} x := y[0]}$$

$$\text{tr-mov-ri}_3\ \frac{\begin{array}{c}(r_i : x)_w, (r_j : y)_4 \in \mu_\Gamma \qquad (x : \theta), (y : N*) \in \Gamma \\ \Sigma(N) = \langle \theta_0, \ldots, \theta_{n-1} \rangle \qquad \Sigma \vdash \theta_0 <: \theta \qquad \textit{sizeof}(\theta_0) = \textit{sizeof}(\theta) = w\end{array}}{\mu_\Gamma; \Gamma; \Sigma \vdash \mathsf{mov}_w\ r_i, [r_j] \overset{\iota}{\rightsquigarrow} x := y \rightarrow 0}$$

$$\text{tr-mov-ir}_1\ \frac{\begin{array}{c}(r_i : x)_4, (r_j : y)_w \in \mu_\Gamma \qquad (x : \theta_1*), (y : \theta_2) \in \Gamma \\ \Sigma \vdash \theta_2 <: \theta_1 \qquad \textit{sizeof}(\theta_1) = \textit{sizeof}(\theta_2) = w\end{array}}{\mu_\Gamma; \Gamma; \Sigma \vdash \mathsf{mov}_w\ [r_i], r_j \overset{\iota}{\rightsquigarrow} *x := y}$$

$$\text{tr-mov-ir}_2\ \frac{\begin{array}{c}(r_i : x)_4, (r_j : y)_w \in \mu_\Gamma \qquad (x : \theta_1[]*), (y : \theta_2) \in \Gamma \\ \Sigma \vdash \theta_2 <: \theta_1 \qquad \textit{sizeof}(\theta_1) = \textit{sizeof}(\theta_2) = w\end{array}}{\mu_\Gamma; \Gamma; \Sigma \vdash \mathsf{mov}_w\ [r_i], r_j \overset{\iota}{\rightsquigarrow} x[0] := y}$$

$$\text{tr-mov-ir}_3\ \frac{\begin{array}{c}(r_i : x)_4, (r_j : y)_w \in \mu_\Gamma \qquad (x : N*), (y : \theta) \in \Gamma \\ \Sigma(N) = \langle \theta_0, \ldots, \theta_{n-1} \rangle \qquad \Sigma \vdash \theta <: \theta_0 \qquad \textit{sizeof}(\theta_0) = \textit{sizeof}(\theta) = w\end{array}}{\mu_\Gamma; \Gamma; \Sigma \vdash \mathsf{mov}_w\ [r_i], r_j \overset{\iota}{\rightsquigarrow} x \rightarrow 0 := y}$$

$$\text{tr-mov-ri+}_1\ \frac{\begin{array}{c}(r_i : x)_w, (r_j : y)_4 \in \mu_\Gamma \qquad (x : \theta_1), (y : \theta_2[]*) \in \Gamma \qquad \Sigma \vdash \theta_2 <: \theta_1 \\ \textit{sizeof}(\theta_1) = \textit{sizeof}(\theta_2) = w \qquad c = m * w \qquad \Gamma; \Sigma \vdash m : t\end{array}}{\mu_\Gamma; \Gamma; \Sigma \vdash \mathsf{mov}_w\ r_i, [r_j + c] \overset{\iota}{\rightsquigarrow} x := y[m]}$$

$$\text{tr-mov-ri+}_2\ \frac{\begin{array}{c}(r_i : x)_w, (r_j : y)_4 \in \mu_\Gamma \qquad (x : \theta), (y : N*) \in \Gamma \qquad \Sigma \vdash \theta_m <: \theta \\ \textit{sizeof}(\theta_m) = \textit{sizeof}(\theta) = w \qquad \Sigma(N) = \langle \theta_0, \ldots, \theta_{n-1} \rangle \qquad c = \sum_{k=0}^{m-1} \textit{sizeof}(\theta_k)\end{array}}{\mu_\Gamma; \Gamma; \Sigma \vdash \mathsf{mov}_w\ r_i, [r_j + c] \overset{\iota}{\rightsquigarrow} x := y \rightarrow m}$$

$$\text{tr-mov-i+r}_1\ \frac{\begin{array}{c}(r_i : x)_4, (r_j : y)_w \in \mu_\Gamma \qquad (x : \theta_1[]*), (y : \theta_2) \in \Gamma \qquad \Sigma \vdash \theta_2 <: \theta_1 \\ \textit{sizeof}(\theta_1) = \textit{sizeof}(\theta_2) = w \qquad c = m * w \qquad \Gamma; \Sigma \vdash m : t\end{array}}{\mu_\Gamma; \Gamma; \Sigma \vdash \mathsf{mov}_w\ [r_i + c], r_j \overset{\iota}{\rightsquigarrow} x[m] := y}$$

$$\text{tr-mov-i+r}_2\ \frac{\begin{array}{c}(r_i : x))_4, (r_j : y)_w \in \mu_\Gamma \qquad (x : N*), (y : \theta) \in \Gamma \qquad \Sigma \vdash \theta <: \theta_m \\ \textit{sizeof}(\theta_m) = \textit{sizeof}(\theta) = w \qquad \Sigma(N) = \langle \theta_0, \ldots, \theta_{n-1} \rangle \qquad c = \sum_{k=0}^{m-1} \textit{sizeof}(\theta_k)\end{array}}{\mu_\Gamma; \Gamma; \Sigma \vdash \mathsf{mov}_w\ [r_i + c], r_j \overset{\iota}{\rightsquigarrow} x \rightarrow m := y}$$

Figure 20: Decompilation of mov instructions

$$\boxed{\mu_\Gamma; \Gamma; \Sigma \vdash \iota \overset{\iota}{\rightsquigarrow} \ell := e}$$

$$\text{tr-}\oplus\text{-r*}_1 \frac{\begin{array}{cc}(r_i : x)_4, (r_j : y)_4 \in \mu_\Gamma & (x : \theta[]*), (y : \mathsf{long}) \in \Gamma \\ c = m * \mathit{sizeof}(\theta) & \Gamma; \Sigma \vdash m : \mathsf{long}\end{array}}{\mu_\Gamma; \Gamma; \Sigma \vdash \mathsf{op}_4^{\oplus}\ r_i, r_j * c \overset{\iota}{\rightsquigarrow} x := x \oplus (y * m)}$$

$$\text{tr-}\oplus\text{-r*}_2 \frac{\begin{array}{ccc}(r_i : x)_w \in \mu_\Gamma & & (r_j : y)_w \in \mu_\Gamma \\ (x : t) \in \Gamma & (y : t) \in \Gamma & \Gamma; \Sigma \vdash c : t\end{array}}{\mu_\Gamma; \Gamma; \Sigma \vdash \mathsf{op}_w^{\oplus}\ r_i, r_j * c \overset{\iota}{\rightsquigarrow} x := x \oplus (y * c)}$$

$$\text{tr-}\oplus\text{-rc} \frac{\begin{array}{cc}(r_i : x)_4 \in \mu_\Gamma & (x : \theta[]*) \in \Gamma \\ c = m * \mathit{sizeof}(\theta) & \Gamma; \Sigma \vdash m : t\end{array}}{\mu_\Gamma; \Gamma; \Sigma \vdash \mathsf{op}_4^{\oplus}\ r_i, c \overset{\iota}{\rightsquigarrow} x := x \oplus m}$$

$$\text{tr-}\otimes\text{-rc} \frac{\begin{array}{cc}(r_i : x)_w \in \mu_\Gamma & (x : t) \in \Gamma \\ \mathit{sizeof}(t) = w & \Gamma; \Sigma \vdash c : t\end{array}}{\mu_\Gamma; \Gamma; \Sigma \vdash \mathsf{op}_w^{\otimes}\ r_i, c \overset{\iota}{\rightsquigarrow} x := x \otimes c}$$

$$\text{tr-}\otimes\text{-rr} \frac{\begin{array}{ccc}(r_i : x)_w \in \mu_\Gamma & & (r_j : y)_w \in \mu_\Gamma \\ (x : t) \in \Gamma & (y : t) \in \Gamma & \mathit{sizeof}(t) = w\end{array}}{\mu_\Gamma; \Gamma; \Sigma \vdash \mathsf{op}_w^{\otimes}\ r_i, r_j \overset{\iota}{\rightsquigarrow} x := x \otimes y}$$

$$\text{tr-alloc-r*} \frac{\begin{array}{ccc}(r_i, x)_4 \in \mu_\Gamma & (r_j, y)_{\mathit{sizeof}(t)} \in \mu_\Gamma & \Gamma; \Sigma \vdash m : t \\ (x : \theta[]*) \in \Gamma & (y : t) \in \Gamma & c = \mathit{sizeof}(\theta) * m\end{array}}{\mu_\Gamma; \Gamma; \Sigma \vdash \mathsf{alloc}\ r_i, r_j * c \overset{\iota}{\rightsquigarrow} x := \mathsf{new}\ \theta[y * m]}$$

$$\text{tr-alloc-rc}_3 \frac{\begin{array}{cc}(r_i, x)_4 \in \mu_\Gamma & (x : \theta[]*) \in \Gamma \\ c = m * \mathit{sizeof}(\theta) & \Gamma; \Sigma \vdash m : t\end{array}}{\mu_\Gamma; \Gamma; \Sigma \vdash \mathsf{alloc}\ r_i, c \overset{\iota}{\rightsquigarrow} x := \mathsf{new}\ \theta[m]}$$

$$\text{tr-alloc-rc}_1 \frac{(r_i, x)_4 \in \mu_\Gamma \quad \mathit{sizeof}(\theta) = c \quad (x : \theta*) \in \Gamma}{\mu_\Gamma; \Gamma; \Sigma \vdash \mathsf{alloc}\ r_i, c \overset{\iota}{\rightsquigarrow} x := \mathsf{new}\ \theta}$$

$$\text{tr-alloc-rc}_2 \frac{(r_i, x)_4 \in \mu_\Gamma \quad \mathit{sizeof}(N) = c \quad (x : N*) \in \Gamma}{\mu_\Gamma; \Gamma; \Sigma \vdash \mathsf{alloc}\ r_i, c \overset{\iota}{\rightsquigarrow} x := \mathsf{new\ struct}\ N}$$

$$\text{tr-call} \frac{\begin{array}{cccc}\phi_c(f) = f(\overrightarrow{x : \theta})\langle \overrightarrow{y : \theta'}, l, \lambda_c, j\rangle & (r_u : u)_{\mathit{sizeof}(\theta_u)} \in \mu_\Gamma & & \overrightarrow{(r_v : v)_{\mathit{sizeof}(\theta_v)}} \in \mu_\Gamma \\ (u : \theta_u) \in \Gamma & \overrightarrow{(v : \theta_v)} \in \Gamma & \Sigma \vdash \theta'_j <: \theta_u & \Sigma \vdash \vec{\theta}_v <: \vec{\theta}\end{array}}{\mu_\Gamma; \Gamma; \Sigma \vdash \mathsf{call}\ r_u, f, \overrightarrow{r_v} \overset{\iota}{\rightsquigarrow} u := f(\vec{v})}$$

Figure 21: Decompilation of $\mathsf{op}^{\oplus}$, $\mathsf{op}^{\otimes}$, alloc and call instructions

$$\boxed{\mu_\lambda; \mu_\Gamma; \Gamma; \Sigma \vdash b \overset{b}{\rightsquigarrow} s}$$

$$\text{tr-ret}\frac{}{\mu_\lambda; \mu_\Gamma; \Gamma; \Sigma \vdash \mathsf{ret} \overset{b}{\rightsquigarrow} \mathsf{return}} \qquad \text{tr-goto}\frac{\mu_\lambda(a) = l}{\mu_\lambda; \mu_\Gamma; \Gamma; \Sigma \vdash \mathsf{goto}\ a \overset{b}{\rightsquigarrow} \mathsf{goto}\ l}$$

$$\text{tr-if}\frac{\mu_\lambda(a) = l \quad \mu_\Gamma(r_i) = x \quad (x : \theta) \in \Gamma \quad sizeof(\theta) = w \quad \mu_\lambda; \mu_\Gamma; \Gamma; \Sigma \vdash b \overset{b}{\rightsquigarrow} s}{\mu_\lambda; \mu_\Gamma; \Gamma; \Sigma \vdash (\mathsf{if}_w\ r_i\ \mathsf{goto}\ a); b \overset{b}{\rightsquigarrow} (\mathsf{if}\ x\ \mathsf{goto}\ l); s} \qquad \text{tr-instr}\frac{\mu_\Gamma; \Gamma; \Sigma \vdash \iota \overset{\iota}{\rightsquigarrow} \ell := e \quad \mu_\lambda; \mu_\Gamma; \Gamma; \Sigma \vdash b \overset{b}{\rightsquigarrow} s}{\mu_\lambda; \mu_\Gamma; \Gamma; \Sigma \vdash \iota; b \overset{b}{\rightsquigarrow} \ell := e; s}$$

$$\boxed{\Sigma \vdash d_x \rightsquigarrow d_c}$$

$$\text{tr-def}\frac{\begin{array}{c}\mu_\Gamma = \{\overrightarrow{r_x \mapsto x}, \overrightarrow{r_y \mapsto y}\} \qquad \Gamma = \{\overrightarrow{x : \theta}, \overrightarrow{y : \theta'}\} \qquad r_{y_j} \in \overrightarrow{r_y} \qquad a \in dom(\lambda_x) \\ \mu_\lambda = \{dom(\lambda_x) \mapsto dom(\lambda_c)\} \quad \mu_\lambda(a) = l \quad \forall (a \mapsto l) \in \mu_\lambda :\ \mu_\lambda; \mu_\Gamma; \Gamma; \Sigma \vdash \lambda_x(a) \overset{b}{\rightsquigarrow} \lambda_c(l)\end{array}}{\Sigma \vdash \langle f, \overrightarrow{r_x}, \overrightarrow{r_y}, a, \lambda_x, j\rangle \rightsquigarrow f(\overrightarrow{x : \theta})\langle \overrightarrow{y : \theta'}, l, \lambda_c, j\rangle}$$

Figure 22: Decompilation of basic blocks and function definitions

4.4.1.1 Well-Typing

The first claim states that the recovered witness program is well-typed. This is asserted as three Propositions, one for each of the judgements.

Proposition 10 (Well-Typed Instruction Decompilation)**.**
If $\mu_\Gamma; \Gamma; \Sigma \vdash \iota \overset{\iota}{\rightsquigarrow} \ell := e$, then for some θ_1 and θ_2:

$$\Gamma; \Sigma \vdash \ell : \theta_1 \quad \wedge \quad \Gamma; \Sigma \vdash e : \theta_2 \quad \wedge \quad \Sigma \vdash \theta_2 <: \theta_1$$

Proposition 11 (Well-Typed Block Decompilation)**.**
If $\mu_\lambda; \mu_\Gamma; \Gamma; \Sigma \vdash b \overset{b}{\rightsquigarrow} s$ then $\Gamma; \Sigma \vdash s$.

Proposition 12 (Well-Typed Definition Decompilation)**.**
If $\Sigma \vdash d_x \rightsquigarrow d_c$ then $\Sigma \vdash d_c$.

For the proofs of these propositions, refer to appendix A.1.2. Due to the type safety of MinC, it follows that the witness program is operationally well-behaved.

$$\boxed{\mu_a \vdash \vec{b} \leftrightsquigarrow v} \qquad \frac{}{\mu_a \vdash 0^4 \leftrightsquigarrow 0_*} \qquad \frac{(a : n_*) \in \mu_a}{\mu_a \vdash a \leftrightsquigarrow n_*}$$

$$\frac{}{\mu_a \vdash \bot^4 \leftrightsquigarrow \bot_a} \qquad \frac{}{\mu_a \vdash n^2 \leftrightsquigarrow n_s} \qquad \frac{}{\mu_a \vdash \bot^2 \leftrightsquigarrow \bot_s}$$

$$\frac{}{\mu_a \vdash n^4 \leftrightsquigarrow n_l} \qquad \frac{}{\mu_a \vdash \bot^4 \leftrightsquigarrow \bot_l}$$

Figure 23: Value correspondence

4.4.1.2 Memory Correspondence

The main semantic effect of both MinX and MinC programs is a transformation of program memory. However, because MinX and MinC programs act on very different memory structures, the semantic correspondence is not readily expressed. First it must be defined how low-level and high-level memory structures correspond. Then semantics preservation can be expressed as the preservation of this correspondence. There are three types of memory correspondence to consider: MinX versus MinC values, MinX heaps versus MinC stores, and MinX registers versus MinC local variables.

Value Correspondence Figure 23 defines the judgement $\mu_a \vdash \vec{b} \leftrightsquigarrow v$ that states the basic correspondence between a MinX byte sequence $\vec{b}$ and a MinC value v. This judgement is parameterised by an address map μ_a that relates MinX and MinC addresses. The rules are obvious, relating 0 pointers, addresses, bottoms, and numeric values of the appropriate byte sizes.

Registers versus Local Variables The following relation denotes that MinX register banks $\vec{R}$ and MinC local variables $\vec{\rho}$ are pair-wise related:

$$\mu_a; \vec{\mu_\Gamma}; \sigma \vdash \vec{R} \leftrightsquigarrow \vec{\rho}$$

The relation is parameterized by an address map μ_a, register-variable maps $\vec{\mu_\Gamma}$ and a store σ. The relation stands for:

$$\forall (r : x)_w \in \mu_{\Gamma,i} : \exists n_* : (x : n_*) \in \rho_i : \\ \exists v : (n_* : v) \in \sigma \wedge \exists \vec{b} : \vec{b} = R_{i,0:w}(r) \wedge \mu_a \vdash \vec{b} \leftrightsquigarrow v$$

This expresses that any related register r and local variable x have associated values $\vec{b}$ and v that are related. The main complication is that the local variables are store-mapped whereas the registers are not.

Heaps versus Stores The MinX heap H and MinC store σ are related with:

$$\mu_a; \nu_a; \pi; \vec{\rho} \vdash H \leftrightsquigarrow \sigma$$

This relation summarises six different properties addressing four concerns. Firstly, as in the previous cases, this relation is parameterised by an address map μ_a that relates addresses in H with addresses in σ. Obviously, these related addresses point to related values.

$$\forall (a, n_*)_w \in \mu_a : \exists \vec{b}, v : \vec{b} = H_{0:w}(a) \wedge v = \sigma(a) \wedge \mu_a \vdash \vec{b} \leftrightsquigarrow v$$

Secondly, we have to contend with the difference in granularity between MinX and MinC: While values can only be addressed as a whole in MinC, MinX addresses individual bytes and can point into the middle of a value. To bridge this gap, the address map μ_a only covers the addresses in H that point at the first byte of a value. The complementary header map ν_a relates each address in H (especially those pointing into the middle of a value) to the address of the first byte of the value and its width.

$$\forall a \in dom(H) : \exists a', w : \nu_a(a) = \langle a', w \rangle \wedge (a' + w) \geqslant a$$

These header addresses are fixpoints of ν_a:

$$\forall \langle a, w \rangle \in range(\nu_a) : \nu_a(a) = \langle a, w \rangle$$

Moreover, they are covered by μ_a:

$$\forall \langle a, w \rangle \in \mathit{range}(\nu_a) : \exists n_* : (a : n_*)_w \in \mu_a$$

Thirdly, not all addresses in σ are related to an address in H. This is a consequence of the discrepancy between registers and local variables: the store-mapped local variables (i.e., those tracked in $\vec{\rho}$ or ρ) have no counterpart in H. Hence, they need not have a counterpart in the relation.

$$\forall n_* \in (\mathit{dom}(\sigma) - \mathit{range}(\vec{\rho}, \rho)) : \exists a, w : (a : n_*)_w \in \mu_a$$

Finally, adjacent MinC addresses in a range tracked by π must be related to adjacent addresses in MinX (taking into account the width w of the value).

$$\forall [n, n + c] \in \pi : \forall i \in [0, c - 1] : \exists a, a', w, w' : \\ a + w = a' \wedge (a, n + i)_w \in \mu_a \wedge (a', n + i + 1)_{w'} \in \mu_a$$

4.4.1.3 Semantics Preservation

With the memory relations in place one important aspect of semantics preservation can be stated as: the original MinX program and the corresponding decompiled MinC program take related memories to related memories.

Proposition 13 (Preservation of Related Memory for Instructions). If

- $\mu_\Gamma; \Gamma; \Sigma \vdash \iota \overset{\iota}{\rightsquigarrow} \ell := e$
- $\Gamma; \Sigma; \Psi \vdash \rho$
- $\Sigma; \Psi \vdash \sigma; \pi$
- $\mu_a; \nu_a; \pi; \vec{\rho}, \rho \vdash H \leftrightsquigarrow \sigma$
- $\mu_a; \vec{\mu_\Gamma}, \mu_\Gamma; \sigma \vdash \vec{R}, R \leftrightsquigarrow \vec{\rho}, \rho$
- $\vec{R} \vdash \langle H, R, \iota \rangle \xrightarrow{\iota} \langle H', R' \rangle$,
- $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \ell \rangle \xrightarrow{\ell} \langle \sigma', \pi', a \rangle$, and

- $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma', \pi', e \rangle \xrightarrow{e} \langle \sigma'', \pi'', v \rangle$

then for some $\mu'_a \supseteq \mu_a$ and $\nu'_a \supseteq \nu_a$:

- $\mu'_a; \vec{\mu_\Gamma}, \mu_\Gamma; \sigma' \circ \{a \mapsto v\} \vdash \vec{R}, R \leftrightsquigarrow \vec{\rho}, \rho$
- $\mu'_a; \nu'_a; \pi'; \vec{\rho}, \rho \vdash H' \leftrightsquigarrow \sigma' \circ \{a \mapsto v\}$

A second aspect of the semantics preservation is that, if the MinX program does not get stuck, the MinC program may get stuck only through a violation of memory that is guarded by π, or by non-termination.

Proposition 14 (Preservation of Progress for Instructions)**.** If

- $\mu_\Gamma; \Gamma; \Sigma \vdash \iota \overset{\iota}{\rightsquigarrow} \ell := e$
- $\Gamma; \Sigma; \Psi \vdash \rho$
- $\Sigma; \Psi \vdash \sigma; \pi$
- $\mu_a; \nu_a; \pi; \vec{\rho}, \rho \vdash H \leftrightsquigarrow \sigma$
- $\mu_a; \vec{\mu_\Gamma}, \mu_\Gamma; \sigma \vdash \vec{R}, R \leftrightsquigarrow \vec{\rho}, \rho$, and
- $\vec{R} \vdash \langle H, R, \iota \rangle \xrightarrow{\iota} \langle H', R' \rangle$

then

- $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \ell \rangle \xrightarrow{\ell} \mathsf{err}$ or
- $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \ell \rangle \xrightarrow{\ell} \langle \sigma', \pi', a \rangle$ and $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma', \pi', e \rangle \xrightarrow{e} \mathsf{err}$, or
- $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \ell \rangle \xrightarrow{\ell} \langle \sigma', \pi', a \rangle$ and $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma', \pi', e \rangle \xrightarrow{e} \langle \sigma'', \pi'', v \rangle$.

There are similar such propositions for basic blocks and for function definitions. Again, proofs can be found in the appendix (A.1.3).

4.5 Implementation

The decompilation relation is a conceptual device, literally a relation, which details what it means for a MinX program to be in correspondence with a MinC program. An algorithm, however, can be derived for solving a problem by adding control to Horn clauses that specify the problem [78]. Following this methodology, the decompilation relation was translated rule-for-rule (almost verbatim) into Horn clauses, programming the control using Constraint Handling Rules (CHR) [45], which is an extension to Prolog. Control defaults to leftmost goal selection, with the exception of predicates annotated as CHR. These are interpreted as constraints, which reside in a constraint store, and interact with one another to realise propagation, delay non-deterministic choice, and thereby avoid needless backtracking. As an illustration, consider the `struct`(N,c,m,θ) constraint which holds iff $\Sigma(N) = \langle\theta_0, \ldots, \theta_{n-1}\rangle$, $c = \sum_{i=0}^{m-1} \mathsf{sizeof}(\theta_i)$ and $\theta = \theta_m$. Two such constraints in the store that share the same N can be combined into one provided they share the same byte offset c, an action that both simplifies the store and performs propagation. This can be specified in CHR as:

```
struct(N,C,M1,Ty1) \ struct(N,C,M2,Ty2) <=>
    M1 = M2, Ty1 = Ty2.
```

Furthermore, given a CHR constraint `sizeOf`(θ,w) that holds iff $w = \mathsf{sizeof}(\theta)$, and two constraints `struct`(N,c,m,θ) and `struct`(N,$c+w$,m',θ') it follows $m' = m + 1$. This form of propagation can be realised in CHR using:

```
struct(N,C1,M1,Ty1), struct(N,C2,M2,_Ty2) ==>
    nonvar(C1), nonvar(C2), sizeOf(Ty1,W),
    nonvar(W), C2 =:= C1 + W
    |
    M2 #= M1 + 1.
```

CHR rules are likewise used to express the subtyping relation. In all, this gives a solver for computing a witness and its type, in less than 900 LOC, but more importantly, derives one that is faithful to the rules of the decompilation relation.

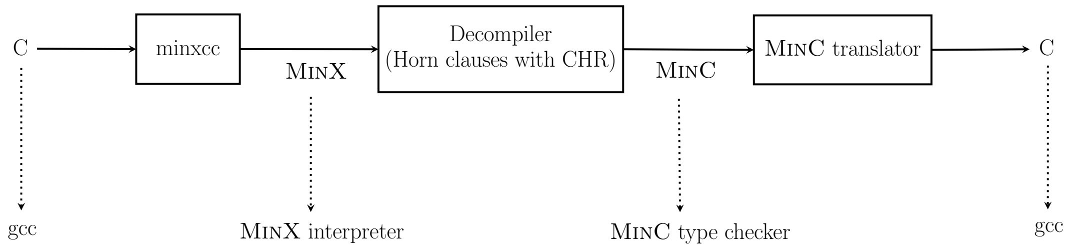


Figure 24: The MinX to MinC toolchain

To generate input for the solver, the self-hosting ANSI C89 compiler `ucc` [120] was retargeted to generate MinX. The resulting derivative, dubbed `minxcc`, supports the core features of C89. To stay within MinX, `minxcc` applies some rather unusual transformations. Each constant string, which is normally encoded as a global pointer literal, is converted into a function that returns a pointer to newly allocated heap memory, that contains the string. `malloc` (and its friends) are replaced by the `alloc` instruction, while calls to `free` are removed completely. Memory thus grows as execution proceeds, exactly as specified in Fig 18. Mathematical operations such as `=<` and logical operations such as `xor` are reduced to MinX operations using equivalences taken from Hacker's Delight [72].

As a sanity check, a Haskell interpreter was written for MinX, following the SOS semantics of Figures 13 and 14. For each benchmark, the results of interpreting the MinX code, on various inputs, were then checked against those obtained by executing the benchmark after compilation using `gcc` (which was taken as ground truth). For the satisfaction of going full circle, a translator was written in Haskell to convert MinC witness programs into C, and each witness was then compiled and checked again with `gcc`. The complete toolchain for compiling a C program to MinX, decompiling to MinC, and finally converting back to C is shown in Figure 24. The dotted lines point to checks that were performed on various inputs and outputs of the toolchain components.

The solver requires input to be pre-processed and presented in the MinX language. The left pane of Figure 25 lists (pretty-printed) MinX code for summing the elements of a linked list (the same example used in Sections 1.1 and 2.1). The arguments, return register and locals, denoted $\overrightarrow{r_{arg}}, r_{ret}, \overrightarrow{r_{loc}}$ in Section 4.2, correspond to `(r1)`, `r0` and `(r2)` respectively in the code listing. The mapping λ'_x

```
iterative_sum {
   mov4 r0, 0 ;
   goto .BB1
.BB0:
   mov4 r2, [r1] ;
   add4 r0, r2 ;
   mov4 r1, [r1 + 4] ;
.BB1:
   if4 r1 goto .BB0 ;
   ret
} <(r1), r0, (r2)>
```

```
struct struct1 {
  long;
  struct1*;
};

iterative_sum(struct1* x) {
   long y1, y2
0: y1 = 0;
   goto 2
1: y2 = x->0;
   y1 = y1 + y2;
   x = x->1;
2: if x goto 1;
   return y1
}
```

Figure 25: Iterative summation of a linked list in MinX (left) and MinC (right)

is represented using the `.BB0` and `.BB1` labels to directly tag their corresponding block. Note that blocks can overlap. The label of the entry block, a_0, is implicit, adopting the convention that the first block is always the entry block.

The right pane of Figure 25 presents the MinC witness program generated by the solver, again pretty-printed for human comprehensibility since the solver represents the witness as an abstract syntax tree. The local variables, denoted $\overrightarrow{y:\theta'}$ in Section 4.3, are given immediately before the entry block. The mapping λ_c is represented, again by using labels to tag the blocks. The index j, used to identify which local variable is returned in Section 4.3, is identified by printing each `return` statement with the variable y_j.

4.6 Evaluation

The solver was deployed on a suite of textbook [122] programs, chosen because of their use of data-structures. Figure 5 lists the *benchmarks*, complete with *LoC* for the C and MinX assembly files. The *solns* column records the number of type assignment and witness program pairs generated by the solver. The *original structs* columns indicates the number of struct types defined in the benchmark, whereas

benchmark	LoC C	LoC MinX	original structs	solns	recovered structs	time (s)	structure type
aatree	315	2,734	1	1	1	6.543	binary tree
avltree	269	2,188	1	1	2	2.041	binary tree
binheap	184	2,558	2	2	2/1	0.109	binary tree
binomial	303	3,732	2	2	5/4	0.249	binary tree
hashsep	256	1,017	2	4	6/5/6/5	0.077	linked list
hashquad	260	977	2	4	2/3/2/2	0.049	array
kdtree	112	891	1	1	1	1.527	binary tree
leftheap	182	762	1	1	1	0.825	binary tree
list	262	829	1	2	2/1	1.054	linked list
mergesort	135	664	1	1	1	0.628	linked list
pairheap	298	3,216	1	1	2	3.316	tree/doubly linked list
queue	188	1,960	1	1	1	0.886	array
redblack	317	2,918	1	2	3/2	0.193	binary tree
sets	120	355	0	1	0	0.276	array
skip	239	2,616	1	1	1	2.089	linked list
sort	364	2,339	0	1	0	2.904	array
splay	332	2,648	1	1	2	4.975	binary tree
stackar	161	1,680	1	1	1	0.640	array
stackli	140	1,270	1	2	2/2	0.464	linked list
treap	288	2,580	1	1	2	3.834	binary tree
tree	208	1,104	1	1	2	2.158	binary tree

Table 5: Solutions and Recovered Structures

recovered structs records the number of struct definitions in each of the solutions. Thus 6/5/6/5, for example, indicates that the first solution has 6 recovered structs, the second has 5, the third 6, and the fourth 5; backtracking enabling all solutions to be enumerated. The *time* column indicates the time required to recover all solutions for a given benchmark. Finally, the *structure type* column indicates the general type of the datastructures used in the benchmark; array, binary tree, tree, linked list, or doubly linked list. Note that there are no benchmarks that use mutually recursive structures, however since there is no theoretical reason why the system could not recover such types further testing would be worthwhile. The benchmarks were run on a single core Intel Atom Z540 at 1.86GHz with

2GB of RAM. The benchmarks, assembly files, and witnesses (which embed the recovered types) are all available in an on-line appendix, that is available at `http://kar.kent.ac.uk/51448/`.

Observe that some benchmarks have more than one solution and some solutions have a different number of struct definitions than the original program. This is due to several factors: Homogeneous structs, where every (accessed) field has the same type, cannot be distinguished from arrays, and therefore can be typed either as an array or a struct. This issue is exhibited by the `binheap` benchmark.

When the same struct type is used in separate parts of a program (e.g., in functions that are never called) the decompiler generates distinct copies of the struct. In most cases the definitions are identical (as in the `tree` benchmark), however a function may not access every field of a struct, leading to an under-constrained type assignment problem and a struct definition that omits the unaccessed fields, as in the `treap` and `avltree` benchmarks. Combining these issues can result in multiple solutions that differ in their types and number of structs, which arises in the `binomial` benchmark.

The key point, however, is not visible from the table: in every case there exists one witness program whose regenerated types are identical to those of the original benchmark. Moreover, for benchmarks with multiple solutions, every witness has types compatible with those of the original program (in the sense that arrays are compatible with homogeneous structs). In addition, all *recursive* types in every witness are present in the original, and every *recursive* type present in the original appears in every witness. Furthermore, when translated back into C, each witness behaves as the original benchmark.

4.7 Summary

This chapter has answered the fundamental question of how to derive types from a binary executable that truly have semantic meaning. The solution is both principled and unique in that it derives a high-level witness program in concert with the types (provided one exists). By proving that the witness inhabits the inferred types, and that the witness and binary are memory consistent, it has been demonstrated unequivocally that the binary conforms to the inferred types. Apart from establishing type correctness, the construction also yields a type-based decompiler.

The decompiler was evaluated on more than 20 textbook programs and, for all, recovered a witness program in a type-safe dialect of C, complete with the original recursive datatypes.

Chapter 5

Simple and Efficient Algorithms for Octagons

While the previous chapters were most directly concerned with type reconstruction from binaries, this chapter focuses on improving the principal operations of the octagon domain, improvements that were discovered during the development of the SMT solver framework (originally designed for type recovery) detailed in Chapters 2 and 3. As detailed previously (in the introductory chapter, Section 1.3), the octagon domain has many applications in static program analysis. Moreover, because octagons can express relationships between program variables, the domain can ultimately be used to determine possible values of variables (or machine code registers/memory) when employed within an abstract interpretation framework, such as that required to perform control-flow recovery from binaries [77].

The chapter begins with a primer on octagons. The primer first explains how octagonal constraints can be translated into paths in a weighted graph, which permits octagonal systems to be represented using difference bound matrices (DBMs) which are widely used for integer differences (as detailed in Section 3.3.1). Focus is then on the principal operation of the domain: closure. Closure makes explicit all entailed unary and binary constraints in a system. For example, from the constraints $x_i - x_j \leqslant c_1$ and $x_j - x_k \leqslant c_2$ closure derives the entailed inequality $x_i - x_k \leqslant c_1 + c_2$. Once all entailed constraints have been derived, the octagonal system is closed. Closure is central to octagons because it reveals the satisfiability of an octagonal system and a closed system is a canonical form. Moreover, all other key operations of the domain (join and projection) are easily reduced to it.

The core of the chapter introduces a new algorithm for incremental closure (i.e. adding a single new constraint to an already closed system), and proves its correctness. Classically, closure is computed using shortest path algorithms [43]. Minè demonstrated [93] that incremental closure can be computed more efficiently by virtually rearranging the columns and rows of a matrix so that only the last two columns and rows need to be recomputed. The new algorithm presented in this chapter stems from the observation that introducing a new constraint can only reduce the distance between existing variables in a limited number of ways. This leads to an algorithm that is conceptually simple, and provides a greater performance benefit over existing approaches. Subsequently, the chapter will present straightforward and concise correctness proofs for non-incremental closure. Finally, the new algorithm is evaluated against the existing approaches, and the performance demonstrated and discussed.

5.1 Preliminaries

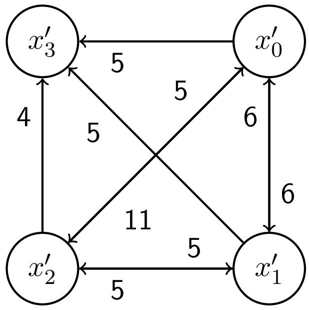
This section serves as a self-contained introduction to the definitions and concepts required in subsequent sections. For more details, please consult the seminal [93, 94] and subsequent [3] works on octagons.

5.1.1 The Octagon Domain and its Representation

An octagonal constraint is a two variable inequality of the form $\pm x_i \pm x_j \leqslant d$ where x_i and x_j are variables and d is a constant. An octagon is a set of points satisfying a system of octagonal constraints. The octagon domain is the set of all octagons that can be defined over the variables $x_0, \ldots, x_{n-1}$.

Implementations of the octagon domain reuse the machinery developed for solving difference constraints of the form $x_i - x_j \leqslant d$. Minè [94] showed how to translate octagonal constraints to difference constraints over an extended set of variables $x'_0, \ldots, x'_{2n-1}$. A single octagonal constraint translates into a conjunction

$$\begin{array}{r} x_0 \leqslant 3 \\ x_1 \leqslant 2 \\ x_0 + x_1 \leqslant 6 \\ -x_0 - x_1 \leqslant 5 \\ -x_0 \leqslant 3 \end{array} \qquad \begin{array}{r} x'_0 - x'_1 \leqslant 6 \\ x'_2 - x'_3 \leqslant 4 \\ x'_0 - x'_3 \leqslant 6 \\ x'_2 - x'_1 \leqslant 6 \\ x'_1 - x'_2 \leqslant 5 \\ x'_3 - x'_0 \leqslant 5 \\ x'_1 - x'_0 \leqslant 6 \end{array}$$



$$\begin{array}{c c} & \begin{array}{cccc} x'_0 & x'_1 & x'_2 & x'_3 \end{array} \\ \begin{array}{c} x'_0 \\ x'_1 \\ x'_2 \\ x'_3 \end{array} & \begin{bmatrix} \infty & 6 & \infty & 6 \\ 6 & \infty & 5 & \infty \\ \infty & 6 & \infty & 4 \\ 5 & \infty & \infty & \infty \end{bmatrix} \end{array}$$

$$\begin{array}{c c} & \begin{array}{cccc} x'_0 & x'_1 & x'_2 & x'_3 \end{array} \\ \begin{array}{c} x'_0 \\ x'_1 \\ x'_2 \\ x'_3 \end{array} & \begin{bmatrix} 0 & 6 & 11 & 5 \\ 6 & 0 & 5 & 5 \\ 5 & 5 & 0 & 4 \\ 5 & 11 & 16 & 0 \end{bmatrix} \end{array}$$

Figure 26: Example of an octagonal system and its DBM representation

of one or more difference constraints as follows:

$$\begin{array}{rclcl} x_i - x_j \leqslant d & \rightsquigarrow & x'_{2i} - x'_{2j} \leqslant d & \wedge & x'_{2j+1} - x'_{2i+1} \leqslant d \\ x_i + x_j \leqslant d & \rightsquigarrow & x'_{2i} - x'_{2j+1} \leqslant d & \wedge & x'_{2j} - x'_{2i+1} \leqslant d \\ -x_i - x_j \leqslant d & \rightsquigarrow & x'_{2i+1} - x'_{2j} \leqslant d & \wedge & x'_{2j+1} - x'_{2i} \leqslant d \\ x_i \leqslant d & \rightsquigarrow & x'_{2i} - x'_{2i+1} \leqslant 2d & & \\ -x_i \leqslant d & \rightsquigarrow & x'_{2i+1} - x'_{2i} \leqslant 2d & & \end{array}$$

A common representation for difference constraints is a difference bound matrix (DBM) which is a square matrix of dimension $n \times n$, where n is the number of variables in the difference system. The value of the entry $d = \mathbf{m}_{i,j}$ represents the constant d of the inequality $x_i - x_j \leqslant d$ where the indices $i, j \in \{0, \dots, n-1\}$. An octagonal constraint over n variables translates to a difference constraint over $2n$ variables, hence a DBM representing an octagon has dimension $2n \times 2n$.

Example 1. *Figure 26 serves as an example of how an octagon translates to a system of differences. The entries of the upper DBM correspond to the constants in the difference constraints. Note how differences which are (syntactically) absent from the system lead to entries which take a symbolic value of ∞. Observe too how the DBM represents an adjacency matrix for the illustrated graph where the weight of a directed edge abuts its arrow.*

The interpretation of a DBM representing an octagon is different to a DBM representing difference constraints. Consequently there are two concretisations for

DBMs: one for interpreting differences and another for interpreting octagons, although the latter is defined in terms of the former:

Definition 1. *Concretisation for rational* ($\mathbb{Q}^n$) *solutions:*

$$\gamma_{diff}(\mathbf{m}) = \{\langle v_0, \dots, v_{n-1}\rangle \in \mathbb{Q}^n \mid \forall i, j. v_i - v_j \leqslant \mathbf{m}_{i,j}\}$$
$$\gamma_{oct}(\mathbf{m}) = \{\langle v_0, \dots, v_{n-1}\rangle \in \mathbb{Q}^n \mid \langle v_0, -v_0, \dots, v_{n-1}, -v_{n-1}\rangle \in \gamma_{diff}(\mathbf{m})\}$$

where the concretisation for integer ($\mathbb{Z}^n$) *solutions can be defined analogously.*

Example 2. *Since octagonal inequalities are modelled as two related differences, the upper DBM of Figure 26 contains duplicated entries, for instance,* $\mathbf{m}_{0,2} = \mathbf{m}_{3,1}$.

Operations on a DBM representing an octagon must maintain equality between the two entries that share the same constant of an octagonal inequality. This requirement leads to the definition of coherence:

Definition 2 (Coherence)**.** *A DBM* $\mathbf{m}$ *is coherent iff* $\forall i.j.\mathbf{m}_{i,j} = \mathbf{m}_{\bar{j},\bar{\imath}}$ *where* $\bar{\imath} = i + 1$ *if* i *is even and* $i - 1$ *otherwise.*

Example 3. *For the upper DBM of Figure 26 observe* $\mathbf{m}_{0,3} = 5 = \mathbf{m}_{2,1} = \mathbf{m}_{\bar{3},\bar{0}}$. *Coherence holds in a degenerate way for unary inequalities, note* $\mathbf{m}_{3,2} = 4 = \mathbf{m}_{3,2} = \mathbf{m}_{\bar{2},\bar{3}}$.

Care should be taken to preserve coherence when manipulating DBMs, by either using carefully designed algorithms or a data structure that maintains coherence automatically [93, Section 4.5]. One final property is necessary for satisfiability:

Definition 3 (Consistency)**.** *A DBM* $\mathbf{m}$ *is consistent iff* $\forall i.\mathbf{m}_{i,i} \geqslant 0$.

5.1.2 Definitions of Closure

Closure properties define canonical representations of DBMs, and can decide satisfiability and support operations such as join and projection. Bellman [11] showed that the satisfiability of a difference system can be decided using shortest path algorithms on a graph representing the differences. If the graph contains a negative cycle then the difference system is unsatisfiable. The same applies for DBMs representing octagons. Closure propagates all the implicit (entailed) constraints in a system, leaving each entry in the DBM with the sharpest possible constraint entailed between the variables. Closure is formally defined below:

Definition 4 (Closure)**.** *A DBM* $\mathbf{m}$ *is closed iff*

- $\forall i.\mathbf{m}_{i,i} = 0$
- $\forall i, j, k.\mathbf{m}_{i,j} \leqslant \mathbf{m}_{i,k} + \mathbf{m}_{k,j}$

Example 4. *The top right DBM in Figure 26 is not closed. By running an all-pairs shortest path algorithm we get the following DBM:*

$$\begin{array}{c c} & \begin{array}{cccc} x'_0 & x'_1 & x'_2 & x'_3 \end{array} \\ \begin{array}{c} x'_0 \\ x'_1 \\ x'_2 \\ x'_3 \end{array} & \begin{bmatrix} 11 & 6 & 11 & 6 \\ 6 & 11 & 5 & 9 \\ 9 & 6 & 11 & 4 \\ 5 & 11 & 16 & 11 \end{bmatrix} \end{array}$$

Notice that the diagonal values have non-negative elements implying that the constraint system is satisfiable. Running shortest path closure algorithms propagates all constraints and makes explicit all constraints implied by the original system. Once satisfiability has been established, we can set the diagonal values to zero to satisfy the definition of closure.

Closure is not enough to provide a canonical form for DBMs representing octagons. Minè defined the notion of strong closure in [93, 94] to do so:

Definition 5 (Strong closure)**.** *A DBM* $\mathbf{m}$ *is strongly closed iff*

- $\mathbf{m}$ *is closed*
- $\forall i, j.\mathbf{m}_{i,j} \leqslant \mathbf{m}_{i,\bar{\imath}}/2 + \mathbf{m}_{\bar{\jmath},j}/2$

The strong closure of DBM m can be computed by $\textsc{Str}(\mathbf{m})$, the code for which is given in Figure 28. The algorithm propagates the property that if $x'_j - x'_{\bar{\jmath}} \leqslant c_1$ and $x'_{\bar{\imath}} - x'_i \leqslant c_2$ both hold then $x'_j - x'_i \leqslant (c_1 + c_2)/2$ also holds. This sharpens the bound

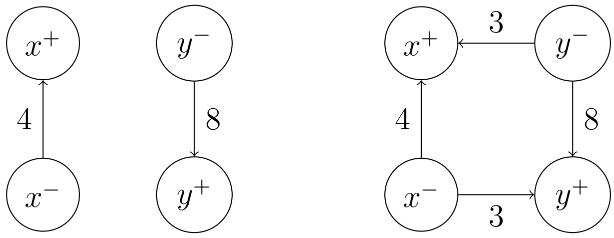


Figure 27: Intuition behind strong closure: Two closed graphs representing the same octagon: $x \leqslant 2 \wedge y \leqslant 4$

on the difference $x'_j - x'_i$ using the two unary constraints encoded by $x'_j - x'_{\bar{\jmath}} \leqslant c_1$ and $x'_{\bar{\imath}} - x'_i \leqslant c_1$, namely, $2x'_j \leqslant c_1$ and $-2x'_i \leqslant c_2$. Note that this constraint propagation is not guaranteed to occur with a shortest path algorithm since there is not necessarily a path from $\mathbf{m}_{i,\bar{\imath}}$ to $\mathbf{m}_{\bar{\jmath},j}$. An example in Figure 27 shows such a situation: the two graphs represent the same octagonal system, but a shortest path algorithm will not propagate constraints on the left graph; hence we need strengthening to bring the two graphs to the same normal form. Strong closure yields a canonical representation: there is a unique strongly closed DBM for any (non-empty) octagon [94]. Thus any semantically equivalent octagonal constraint systems are represented by the same strongly closed DBM. Strengthening is the act of computing strong closure.

Example 5. *The lower right DBM in Figure 26 represents the strong closure of the example octagon system. The strengthen step is run after the shortest path algorithm.*

5.1.3 High-level Overview

Strong closure computation begins with application of a closure algorithm to a DBM. Next, consistency is checked by observing the diagonal has non-negative entries indicating the octagon is satisfiable. If satisfiable, then the DBM is strengthened, resulting in a strongly closed DBM. Note that consistency does not need to be checked again after strengthening.

Figure 28 lists the algorithms required to perform these steps for non-incremental strong closure. A Floyd-Warshall all-pairs shortest path algorithm [43, 121] can

```
function Close(m)
    for k ∈ {0, ..., 2n − 1} do
        for i ∈ {0, ..., 2n − 1} do
            for j ∈ {0, ..., 2n − 1} do
                m'_{i,j} ← min(m_{i,j}, m_{i,k} + m_{k,j})
            end for
        end for
    end for
    return m'
end function
```

```
function CheckConsistent(m)
    for i ∈ {0, ..., 2n − 1} do
        if m_{i,i} < 0 then
            return false
        else
            m_{i,i} = 0
        end if
    end for
    return true
end function
```

```
function Str(m)
    for i ∈ {0, ..., 2n − 1} do
        for j ∈ {0, ..., 2n − 1} do
            m'_{i,j} ← min(m_{i,j}, (m_{i,ī} + m_{j̄,j})/2)
        end for
    end for
    return m'
end function
```

Figure 28: Non-incremental closure and strengthening

be applied to a DBM to compute closure, which is cubic in n. The check for consistency involves a pass over the matrix diagonal to check for a strictly negative entry, as illustrated in the figure. (Note that the consistency check does not reset a positive diagonal entry to zero as in [3, 94], since the incremental algorithms presented in this chapter never relax a zero diagonal entry to a strictly positive value.) This is linear in n. Strong closure can be additionally obtained by following closure with a single call to strengthen, the code for which is also listed in the Figure. This is quadratic in n.

5.2 Incremental Closure

This chapter is concerned with the specific use case of adding a new octagonal constraint to an existing octagon. Minè designed an incremental algorithm for this very task, which can be refactored into computing closure and separately strengthening. His incremental algorithm, and a refinement, are discussed in Section 5.2.1. Section 5.2.2 presents a new incremental algorithm with better performance.

```
1: function MinéIncClose(m, x'_a − x'_b ⩽ d)
2:     m_{a,b} ← min(m_{a,b}, d);
3:     m_{b̄,ā} ← min(m_{b̄,ā}, d);
4:     for k ∈ {0, … , 2n − 1} do
5:         for i ∈ {0, … , 2n − 1} do
6:             for j ∈ {0, … , 2n − 1} do
7:                 if {i, j, k} ∩ {a, ā, b, b̄} ≠ ∅ then
8:                     m_{i,j} ← min(m_{i,j}, m_{i,k} + m_{k,j})
9:                 end if
10:            end for
11:        end for
12:    end for
13:    return m
14: end function
```

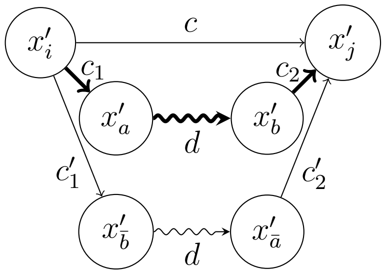
Figure 29: Minè's Incremental Closure Algorithm

5.2.1 Classical Incremental Closure

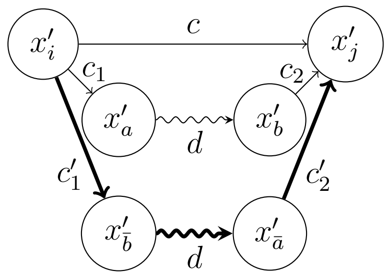
Minè designed an incremental algorithm based on the observation that a new constraint will not affect all the variables of the octagon [93, Section 4.3.4]. Without loss of generality, suppose the inequality $x'_a - x'_b \leqslant d$ is added to the DBM (unary constraints are supported by putting $b = \bar{a}$). Adding $x'_a - x'_b \leqslant d$ implies that the equivalent constraint $x'_{\bar{b}} - x'_{\bar{a}} \leqslant d$ is added too, and the entries $\mathbf{m}_{a,b}$ and $\mathbf{m}_{\bar{b},\bar{a}}$ are strengthened to d to reflect this.

Figure 29 presents a version of Minè's incremental algorithm, specialised for adding $x'_a - x'_b \leqslant d$ to a closed DBM. The algorithm relies on the observation that updating $\mathbf{m}_{a,b}$ and $\mathbf{m}_{\bar{b},\bar{a}}$ will only (initially) mutate the rows and columns for the $x'_a, x'_b, x'_{\bar{a}}, x'_{\bar{b}}$ variables. Since $\mathbf{m}$ was closed, despite the updates, it still follows that $\mathbf{m}_{i,j} \leqslant \mathbf{m}_{i,k} + \mathbf{m}_{k,j}$ if $\{i, j, k\} \cap \{a, \bar{a}, b, \bar{b}\} = \emptyset$. To restore closure it only remains to enforce $\mathbf{m}_{i,j} \leqslant \mathbf{m}_{i,k} + \mathbf{m}_{k,j}$ for $\{i, j, k\} \cap \{a, \bar{a}, b, \bar{b}\} \neq \emptyset$. The incremental algorithm thus applies Floyd-Warshall closure but only updates an entry $\mathbf{m}_{i,j}$ when $\{i, j, k\} \cap \{a, \bar{a}, b, \bar{b}\} \neq \emptyset$ (lines 7 and 8).

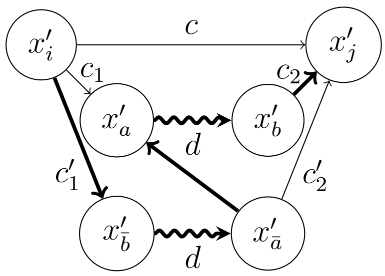
Note that the check $\{i, j, k\} \cap \{a, \bar{a}, b, \bar{b}\} \neq \emptyset$ can be decomposed into three separate checks $i \in \{a, \bar{a}, b, \bar{b}\}$, $j \in \{a, \bar{a}, b, \bar{b}\}$ or $k \in \{a, \bar{a}, b, \bar{b}\}$. Then $k \in \{a, \bar{a}, b, \bar{b}\}$ can be hoisted outside the two inner loops, and likewise $i \in \{a, \bar{a}, b, \bar{b}\}$ can be hoisted outside the inner loop. Furthermore, $i \in \{a, \bar{a}, b, \bar{b}\}$ can be reduced to four



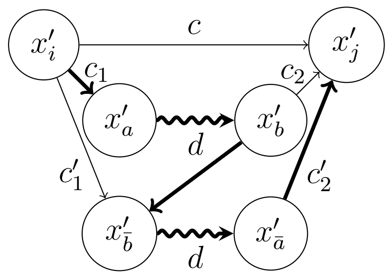
(i, a) and (b, j) are not affected by new constraints



$(i, \bar{b})$ and $(\bar{a}, j)$ are not affected by new constraints



(i, a) shortened by $(i, \bar{b}) + d + (\bar{a}, a)$ or $(\bar{a}, j)$ shortened by $(\bar{a}, a) + d + (b, j)$



$(i, \bar{b})$ shortened by $(i, a) + d + (b, \bar{b})$ (b, j) shortened by $(b, \bar{b}) + d + (\bar{a}, j)$

Figure 30: Four ways to reduce the distance between x'_i and x'_j

constant-time equality checks $(i = a) \vee (i = \bar{a}) \vee (i = b) \vee (i = \bar{b})$. These strength reductions mitigate against overhead of the check $\{i, j, k\} \cap \{a, \bar{a}, b, \bar{b}\} \neq \emptyset$ itself.

This guard reduces the number of min operations from $8n^3$ to $8n^3 - (2n-4)^3 = 48n^2 - 96n + 64$ (notwithstanding those in Str), but at the overhead of $8n^3$ checks. Thus the incremental algorithm is quadratic in the number of min operations but cubic in the number of checks (even with code hoisting).

5.2.2 Improved Incremental Closure

To give the intuition behind the new incremental closure algorithm, consider adding the constraint $x'_a - x'_b \leqslant d$, and thus $x'_{\bar{b}} - x'_{\bar{a}} \leqslant d$, to the closed DBM $\mathbf{m}$. The four diagrams given in Figure 30 illustrate how the path between variables x'_i and x'_j can be shortened. The distance between x'_i and x'_j is c ($\mathbf{m}_{i,j} = c$),

function INCCLOSE($\mathbf{m}$, $x'_a - x'_b \leqslant d$)
  for $i \in \{0, \ldots, 2n-1\}$ **do**
    for $j \in \{0, \ldots, 2n-1\}$ **do**

$$\mathbf{m'}_{i,j} \leftarrow \min \begin{pmatrix} \mathbf{m}_{i,j}, \\ \mathbf{m}_{i,a} + d + \mathbf{m}_{b,j}, \\ \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},j}, \\ \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,j}, \\ \mathbf{m}_{i,a} + d + \mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},j} \end{pmatrix}$$

    end for
  end for
  if CHECKCONSISTENT($\mathbf{m}'$) **then**
    return $\mathbf{m}'$
  else
    return $false$
  end if
end function

Figure 31: Incremental Closure

the distance between x'_i and x'_a is c_1 ($\mathbf{m}_{i,a} = c_1$), etc. The wavy lines denote the new constraints $x'_a - x'_b \leqslant d$ and $x'_{\bar{b}} - x'_{\bar{a}} \leqslant d$ and the heavy lines indicate short-circuiting paths between x'_i and x'_j.

The bottom left path of the figure illustrates how the distance between x'_i and x'_a can be reduced from c_1 by the $x'_{\bar{b}} - x'_{\bar{a}} \leqslant d$ constraint. The same path illustrates how to shorten the distance between $x'_{\bar{a}}$ and x'_j from c'_2 using the $x'_a - x'_b \leqslant d$ constraint. The bottom right path of the figure gives two symmetric cases in which c'_1 and c_2 are sharpened by the addition of $x'_a - x'_b \leqslant d$ and $x'_{\bar{b}} - x'_{\bar{a}} \leqslant d$ respectively. Note that we cannot have the two paths from x'_i to x'_a and from x'_b to x'_j both shortened: at most one of them can change. The same holds for the two paths from x'_i to $x'_{\bar{b}}$ and $x'_{\bar{a}}$ to x'_j. These extra paths lead to the following strategy for updating $\mathbf{m'}_{i,j}$:

$$\mathbf{m'}_{i,j} \leftarrow \min \begin{pmatrix} \mathbf{m}_{i,j}, \\ \mathbf{m}_{i,a} + d + \mathbf{m}_{b,j}, \\ \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},j}, \\ \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,j} \\ \mathbf{m}_{i,a} + d + \mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},j} \end{pmatrix}$$

function $\textsc{IncCloseHoist}(\mathbf{m},\, x'_a - x'_b \leqslant d)$
  $t_1 \leftarrow d + \mathbf{m}_{\bar{a},a} + d;$
  $t_2 \leftarrow d + \mathbf{m}_{b,\bar{b}} + d;$
  for $i \in \{0, \ldots, 2n-1\}$ **do**
    $t_3 \leftarrow \min(\mathbf{m}_{i,a} + d, \mathbf{m}_{i,\bar{b}} + t_1);$
    $t_4 \leftarrow \min(\mathbf{m}_{i,\bar{b}} + d, \mathbf{m}_{i,a} + t_2);$
    for $j \in \{0, \ldots, 2n-1\}$ **do**
      $\mathbf{m}'_{i,j} \leftarrow \min(\mathbf{m}_{i,j}, t_3 + \mathbf{m}_{b,j}, t_4 + \mathbf{m}_{\bar{a},j})$
    end for
    if $\mathbf{m}'_{i,i} < 0$ **then**
      return $false$
    end if
  end for
  return $\mathbf{m}'$
end function

Figure 32: Incremental Closure with code hoisting

This leads to the incremental closure algorithm listed in Figure 31. Quintic min can be realised as four binary min operations, hence the total number of binary min operations required for IncClose is $16n^2$, which is quadratic in n. The listing in Figure 32 shows how commonality can be factored out so that each iteration of the inner loop requires a single ternary min to be computed. Factorisation reduces the number of binary min operations to $2n(2 + 4n) = 8n^2 + 4n$ in IncCloseHoist. Furthermore, like IncClose, IncCloseHoist is not sensitive to the specific traversal order of the DBM, hence has potential for parallelisation. In addition, both IncClose and IncCloseHoist do not incur any checks.

What is intriguing is that the incremental closure algorithms proposed in [23] (upon which this chapter is based) omitted the last two cases of the quinary min operation, and yet this was not exposed by fuzz testing against non-incremental closure. The problem was only exposed in an attempt to justify [23] with simple proofs. The following counter-example demonstrates the oversight:

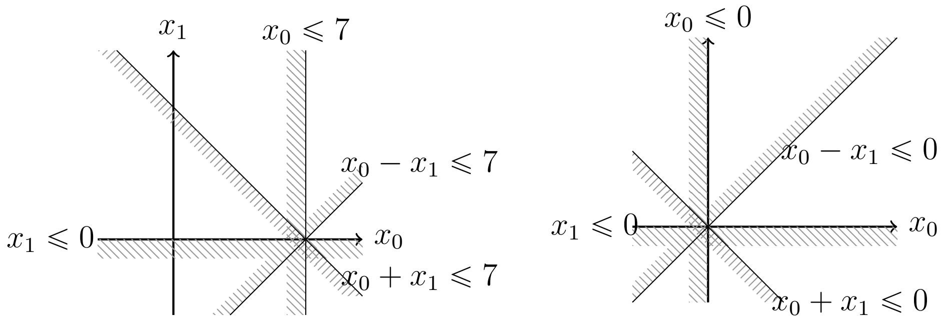


Figure 33: Before and after adding $x_0 - x_1 \leqslant 0$

Example 6. *To illustrate how the incremental closure algorithm of [23] omits a form of propagation, consider adding $x_0 - x_1 \leqslant 0$ to the system on the left*

$$\begin{array}{ll} x_0 \leqslant 7 & \wedge \\ x_1 \leqslant 0 & \wedge \\ x_0 - x_1 \leqslant 7 & \wedge \\ x_0 + x_1 \leqslant 0 & \end{array} \qquad \begin{array}{c c} & \begin{array}{cccc} x'_0 & x'_1 & x'_2 & x'_3 \end{array} \\ \begin{array}{c} x'_0 \\ x'_1 \\ x'_2 \\ x'_3 \end{array} & \left[\begin{array}{cccc} 0 & 14 & 7 & 7 \\ \infty & 0 & \infty & \infty \\ \infty & 7 & 0 & 0 \\ \infty & 7 & \infty & 0 \end{array}\right] \end{array}$$

whose DBM $\mathbf{m}$ *is given on right. The system is illustrated spatially on the left hand side of Figure 33; the right hand side of the same figure shows the effect of adding the constraint $x_0 - x_1 \leqslant 0$. Adding $x_0 - x_1 \leqslant 0$ using the incremental closure algorithm from [23] gives the DBM* $\mathbf{m}'$*;* IncClose *gives the DBM* $\mathbf{m}''$*:*

$$\mathbf{m}' = \begin{array}{c c} & \begin{array}{cccc} x'_0 & x'_1 & x'_2 & x'_3 \end{array} \\ \begin{array}{c} x'_0 \\ x'_1 \\ x'_2 \\ x'_3 \end{array} & \left[\begin{array}{cccc} 0 & 7 & 0 & 0 \\ \infty & 0 & \infty & \infty \\ \infty & 0 & 0 & 0 \\ \infty & 0 & \infty & 0 \end{array}\right] \end{array} \qquad \mathbf{m}'' = \begin{array}{c c} & \begin{array}{cccc} x'_0 & x'_1 & x'_2 & x'_3 \end{array} \\ \begin{array}{c} x'_0 \\ x'_1 \\ x'_2 \\ x'_3 \end{array} & \left[\begin{array}{cccc} 0 & 0 & 0 & 0 \\ \infty & 0 & \infty & \infty \\ \infty & 0 & 0 & 0 \\ \infty & 0 & \infty & 0 \end{array}\right] \end{array}$$

The DBM $\mathbf{m}'$ *represents the constraint $x \leqslant \frac{7}{2}$ but* $\mathbf{m}''$ *encodes the tighter constraint $x \leqslant 0$. The reason for the discrepancy between entries* $\mathbf{m}'_{0,1}$ *and* $\mathbf{m}''_{0,1}$ *is shown* *by the following calculations:*

$$\mathbf{m}'_{0,1} = \min \begin{pmatrix} \mathbf{m}_{0,1} \\ \mathbf{m}_{0,0} + 0 + \mathbf{m}_{2,1} \\ \mathbf{m}_{0,\bar{2}} + 0 + \mathbf{m}_{0,1} \end{pmatrix} = \min \begin{pmatrix} 14, \\ 0+0+7 \\ 7+0+0 \end{pmatrix} = 7$$

$$\mathbf{m}''_{0,1} = \min \begin{pmatrix} \mathbf{m}_{0,1} \\ \mathbf{m}_{0,0} + 0 + \mathbf{m}_{2,1} \\ \mathbf{m}_{0,\bar{2}} + 0 + \mathbf{m}_{0,1} \\ \mathbf{m}_{0,0} + 0 + \mathbf{m}_{2,\bar{2}} + 0 + \mathbf{m}_{0,1} \\ \mathbf{m}_{0,\bar{2}} + 0 + \mathbf{m}_{0,0} + 0 + \mathbf{m}_{2,1} \end{pmatrix} = \min \begin{pmatrix} 14 \\ 0+0+7 \\ 7+0+0 \\ 0+0+0+0+0 \\ 7+0+\infty+0+7 \end{pmatrix} = 0$$

The entry at $\mathbf{m}'_{0,1}$ is calculated using $\mathbf{m}_{2,1}$, but this entry will itself reduce to 0; $\mathbf{m}'_{0,1}$ must take into account the change that occurs to $\mathbf{m}_{2,1}$. More generally, when calculating $\mathbf{m}'_{i,j}$, the min *expression of [23] overlooks how the added constraint can tighten $\mathbf{m}_{i,a}$, $\mathbf{m}_{i,b}$, $\mathbf{m}_{i,\bar{b}}$ or $\mathbf{m}_{\bar{a},j}$. This is remedied in* IncClose *whose correctness is asserted in Theorem 1.* □

The new incremental algorithm is justified by Theorem 1 which, in turn, is supported by Lemma 1. Lemma 1 justifies a quick way to check if a given octagonal constraint o will result in a DBM $\mathbf{m}$ being inconsistent, by checking if any of four paths are non-zero. This is used by the function CheckConsistent, which is used in Figures 31 and 32.

Lemma 1 (Correctness of CheckConsistent)**.** *Suppose* $\mathbf{m}$ *is a closed DBM,* $\mathbf{m}'$ $=$ IncClose$(\mathbf{m}, o)$ *and* $o = (x'_a - x'_b \leqslant d)$*. If* $\mathbf{m}'$ *is consistent then*

- $\mathbf{m}_{b,a} + d \geqslant 0$
- $\mathbf{m}_{\bar{a},\bar{b}} + d \geqslant 0$
- $\mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,\bar{b}} + d \geqslant 0$
- $\mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d \geqslant 0$

The corresponding proof of Lemma 1 is deferred to appendix A.2.

Theorem 1 (Correctness of IncClose). *Suppose* $\mathbf{m}$ *is a closed DBM,* $\mathbf{m}' =$ IncClose$(\mathbf{m}, o)$ *and* $o = (x'_a - x'_b \leqslant d)$*. Then* $\mathbf{m}'$ *is either closed or it is not consistent.*

Proof. Here a proof is outlined. For a complete proof refer to appendix A.2. Suppose $\mathbf{m}'$ is consistent. Because $\mathbf{m}$ is closed $0 = \mathbf{m}_{i,i} \geqslant \mathbf{m}'_{i,i} \geqslant 0$ hence $\mathbf{m}'_{i,i} = 0$. It therefore remains to show $\forall i, j, k . \mathbf{m}'_{i,k} + \mathbf{m}'_{k,j} \geqslant A$ where:

$$A = \min \begin{pmatrix} \mathbf{m}_{i,j}, \\ \mathbf{m}_{i,a} + d + \mathbf{m}_{b,j}, \\ \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},j}, \\ \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,j}, \\ \mathbf{m}_{i,a} + d + \mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},j} \end{pmatrix}$$

There are 5 cases for $\mathbf{m}'_{i,k}$ and 5 for $\mathbf{m}'_{k,j}$ giving 25 in total. The following cases are illustrative:

1-1. Suppose $\mathbf{m}'_{i,k} = \mathbf{m}_{i,k}$ and $\mathbf{m}'_{k,j} = \mathbf{m}_{k,j}$. Because $\mathbf{m}$ is closed:

$$\mathbf{m}'_{i,k} + \mathbf{m}'_{k,j} = \mathbf{m}_{i,k} + \mathbf{m}_{k,j} \geqslant \mathbf{m}_{i,j} \geqslant A$$

1-4. Suppose $\mathbf{m}'_{i,k} = \mathbf{m}_{i,k}$ and $\mathbf{m}'_{k,j} = \mathbf{m}_{k,\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,j}$. Because $\mathbf{m}$ is closed:

$$\begin{aligned} \mathbf{m}'_{i,k} + \mathbf{m}'_{k,j} &= \mathbf{m}_{i,k} + \mathbf{m}_{k,\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,j} \\ &\geqslant \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,j} \geqslant A \end{aligned}$$

3-5. Suppose $\mathbf{m}'_{i,k} = \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},k}$ and $\mathbf{m}'_{k,j} = \mathbf{m}_{k,a} + d + \mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},j}$. Because $\mathbf{m}$ is closed and by Lemma 1:

$$\begin{aligned} \mathbf{m}'_{i,k} + \mathbf{m}'_{k,j} &= \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},k} + \mathbf{m}_{k,a} + d + \mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},j} \\ &= \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},j} \\ &= \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},j} \geqslant A \end{aligned}$$

□

As a final remark on the new algorithm, it is interesting to see that in some circumstances unsatisfiability can be detected without applying any min operations at all. This is justified by the following corollary of Lemma 1:

Corollary 1. *Suppose* $\mathbf{m}$ *is a closed DBM,* $\mathbf{m}' = \textsc{IncClose}(\mathbf{m}, o)$ *and* $o = (x'_a - x'_b \leqslant d)$*. If*

- $\mathbf{m}_{b,a} + d < 0$ *or*
- $\mathbf{m}_{\bar{a},\bar{b}} + d < 0$ *or*
- $\mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,\bar{b}} + d < 0$ *or*
- $\mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d < 0$

then $\mathbf{m}'$ *is not consistent.*

5.3 Simpler Proofs for Classical Strong Closure

The proof of Theorem 1 is compelling since it follows from basic definitions of DBMs and closure. A similar approach can be applied to obtain more direct proofs for the classical strong closure defined in [3].

5.3.1 Classical Strong Closure

The classical strong closure by Minè strengthens repeatedly within the main Floyd-Warshall loop, but it was later shown [3] that this is equivalent to applying strengthening just once after the main loop. The following Theorem [3] justifies this tactic, alongside a more direct proof.

Theorem 2 (Correctness of Strong Closure)**.** *Suppose* $\mathbf{m}$ *is a closed, coherent DBM and* $\mathbf{m}' = \textsc{Str}(\mathbf{m})$*. Then* $\mathbf{m}'$ *is a strongly closed DBM.*

Proof. Observe that $\mathbf{m}'_{i,\bar{\imath}} = \min(\mathbf{m}_{i,\bar{\imath}}, (\mathbf{m}_{i,\bar{\imath}} + \mathbf{m}_{i,\bar{\imath}})/2) = \mathbf{m}_{i,\bar{\imath}}$ and likewise $\mathbf{m}'_{j,\bar{\jmath}} = \mathbf{m}_{j,\bar{\jmath}}$. Therefore $\mathbf{m}'_{i,j} \leqslant (\mathbf{m}_{i,\bar{\imath}} + \mathbf{m}_{\bar{\jmath},j})/2 = (\mathbf{m}'_{i,\bar{\imath}} + \mathbf{m}'_{\bar{\jmath},j})/2$.

Because $\mathbf{m}$ is closed $0 = \mathbf{m}_{i,i} \leqslant \mathbf{m}_{i,\bar{\imath}} + \mathbf{m}_{\bar{\imath},i}$ and thus

$$\mathbf{m}'_{i,i} = \min(\mathbf{m}_{i,i}, (\mathbf{m}_{i,\bar{\imath}} + \mathbf{m}_{\bar{\imath},i})/2) = \min(0, (\mathbf{m}_{i,\bar{\imath}} + \mathbf{m}_{\bar{\imath},i})/2) = 0$$

To show $\mathbf{m'}_{i,j} \leqslant \mathbf{m'}_{i,k} + \mathbf{m'}_{k,j}$ we proceed by case analysis:

- Suppose $\mathbf{m'}_{i,k} = \mathbf{m}_{i,k}$ and $\mathbf{m'}_{k,j} = \mathbf{m}_{k,j}$. Because $\mathbf{m}$ is closed:

$$\mathbf{m'}_{i,j} \leqslant \mathbf{m}_{i,j} \leqslant \mathbf{m}_{i,k} + \mathbf{m}_{k,j} = \mathbf{m'}_{i,k} + \mathbf{m'}_{k,j}$$

- Suppose $\mathbf{m'}_{i,k} \neq \mathbf{m}_{i,k}$ and $\mathbf{m'}_{k,j} = \mathbf{m}_{k,j}$. Because $\mathbf{m}$ is closed and coherent:

$$\begin{aligned} 2\mathbf{m'}_{i,k} + 2\mathbf{m'}_{k,j} &= \mathbf{m}_{i,\bar{\imath}} + \mathbf{m}_{\bar{k},k} + 2\mathbf{m}_{k,j} \geqslant \mathbf{m}_{i,\bar{\imath}} + \mathbf{m}_{\bar{k},j} + \mathbf{m}_{k,j} \\ &= \mathbf{m}_{i,\bar{\imath}} + \mathbf{m}_{\bar{\jmath},k} + \mathbf{m}_{k,j} \geqslant \mathbf{m}_{i,\bar{\imath}} + \mathbf{m}_{\bar{\jmath},j} \geqslant 2\mathbf{m'}_{i,j} \end{aligned}$$

- Suppose $\mathbf{m'}_{i,k} = \mathbf{m}_{i,k}$ and $\mathbf{m'}_{k,j} \neq \mathbf{m}_{k,j}$. Symmetric to the previous case.
- Suppose $\mathbf{m'}_{i,k} \neq \mathbf{m}_{i,k}$ and $\mathbf{m'}_{k,j} \neq \mathbf{m}_{k,j}$. Because $\mathbf{m}$ is closed:

$$\begin{aligned} 2\mathbf{m'}_{i,k} + 2\mathbf{m'}_{k,j} &= \mathbf{m}_{i,\bar{\imath}} + \mathbf{m}_{\bar{k},k} + \mathbf{m}_{k,\bar{k}} + \mathbf{m}_{\bar{\jmath},j} \\ &\geqslant \mathbf{m}_{i,\bar{\imath}} + \mathbf{m}_{\bar{k},\bar{k}} + \mathbf{m}_{\bar{\jmath},j} = \mathbf{m}_{i,\bar{\imath}} + 0 + \mathbf{m}_{\bar{\jmath},j} \geqslant 2\mathbf{m'}_{i,j} \end{aligned}$$

□

5.4 Experiments

OCaml implementations have been developed to test the closure algorithms presented in this chapter. Specifically, the prototype randomly generates a satisfiable octagonal constraint system with a specific number of variables and constraints, closes it, then adds a single constraint and performs strong closure. The DBM entries were IEEE 754 standard precision floats. The randomly generated DBM is always satisfiable because constraints are generated around a point using a random pair of variables from the set, if a constraint is unsatisfiable it is "flipped" by negating the left hand side. Note however that when adding the final constraint the resulting DBM may be infeasible, short-circuiting a call to Str. The following algorithms were implemented and tested:

- NIC: Close followed by CheckConsistent and Str;

- MIC: MinéIncClose followed by CheckConsistent and Str;
- MICH: MinéIncClose followed by CheckConsistent and Str, but with loop-invariant code motion [99, Section 13.2] applied to hoist constant DBM expressions to minimise reads and writes to the DBM;
- ICH: IncCloseHoist (with its own consistency check) followed by Str, and an additional check for rapidly detecting unsatisfiability using Corollary 1;

The resulting DBMs were then all checked for equality against Close (using a tolerance threshold to handle the floats). For each problem instance (number of variables and number of constraints), the experiments were repeated 5000 times and the timings averaged. The experiments were run on a 32-core Intel Xeon workstation with 128GB of memory, using the OCaml forkwork library to run multiple experiments together.

The results of the experiments are summarised in Figures 34, 35 and 36. The labels on the horizontal axis give the number of variables n and the number of constraints m for each experiment, abbreviated to $n - m$. The vertical axis gives average time, in seconds, taken for each experiment. A log scale is used on the vertical so that the timings for the new incremental algorithms are discernible.

5.5 Discussion

Minè's incremental closure algorithm (MIC) is faster than non-incremental closure (NIC) and thus the additional overhead of checking the guard at line 7 of Figure 29 does not negate the saving gained in the min operations. However the key difference between MIC and MICH is that the guard in MICH is decomposed into three separate checks to permit loop-invariant code motion. This suggests that the incremental algorithm of Minè is sensitive to how the check at line 7 of Figure 29 is realised, no doubt because it is applied $O(n^3)$ times. The new ICH algorithm is $O(n^2)$ and is uniformly faster than MICH, approaching or achieving an order of magnitude faster in most cases. Interestingly, none of the algorithms appear to be very sensitive to the number of constraints, m. The running time increases with m for small m, as the likelihood of writing a DBM entry increases as the DBM

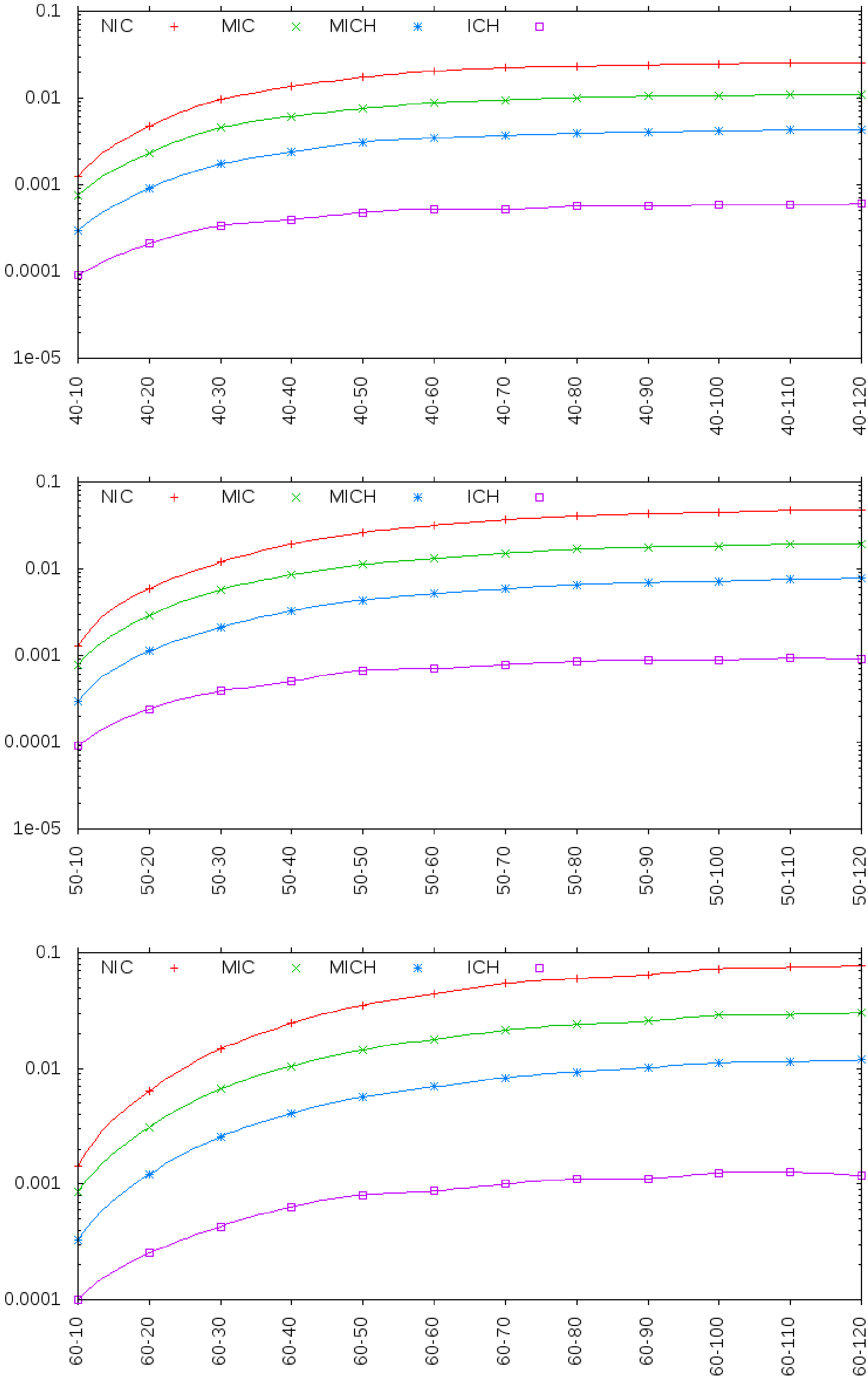


Figure 34: Experiments with strong closure algorithms on 40, 50 and 60 variables

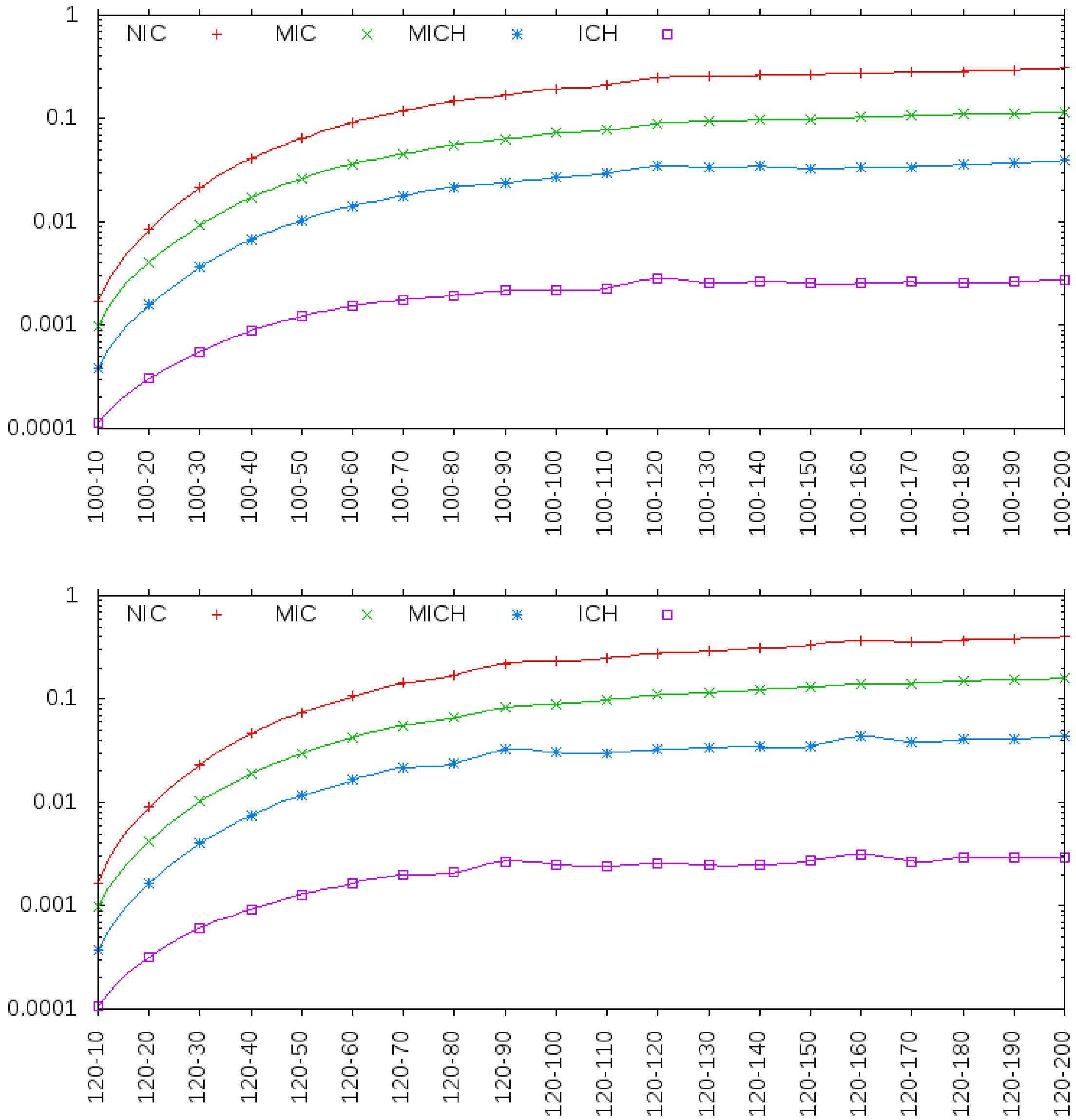


Figure 35: Experiments with strong closure algorithms on 100 and 120 variables

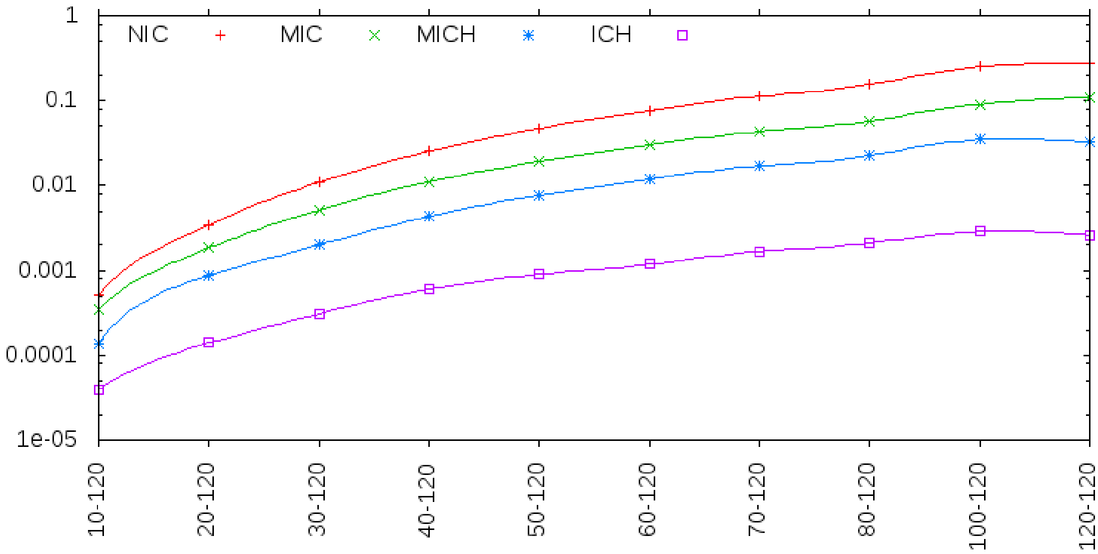


Figure 36: Experiments with strong closure algorithms on 120 constraints

becomes more populated. However, once the DBM is densely populated, which happens when m is large, the running times stabilise, demonstrating that the key parameter is the number of variables n, rather than m.

5.6 Summary

The widespread use of the octagon domain means that any computational improvement could impact on many areas of analysis and verification. A new incremental closure algorithm has been presented, geared towards the popular DBM representation of octagons [94]. The simplicity of the algorithm paired with its computational efficiency makes it an attractive choice for any implementor. Further improvements to incremental closure have been proposed, pending review [24]. Future work will focus on exploiting incrementality to accurately model machine arithmetic [111].

Chapter 6

Related work

6.1 Type recovery

A self-contained introduction to type recovery is given in [119, Chapter 5], which summarises the problem as "The ... problem for a decompiler is to associate each piece of data with a high-level type". The author, like others [37, 118], introduces a dataflow analysis over a type lattice of primitive types, but accepted wisdom is to formulate type inference as constraint solving because dataflow analysis classically deals with unidirectional flows.

Dynamic type recovery Dynamic techniques have been suggested for type recovery [84], in which types are reconstructed from an execution trace. Each memory location accessed by the program is tagged with a timestamp because the same location can store values of different types over its lifetime. Each location and timestamp pair is then assigned a type, in either the on-line or off-line phases of the type recovery algorithm. In the on-line phase known types are propagated to the pairs, as a value which inhabits a type, is stored in a location. However, the type may remain unknown until the control encounters a system call or a library call, or some machine instruction, whose arguments or operands expose the type. The on-line phase is thus augmented with an off-line phase which propagates a type assignment against the control to resolve any unassigned pair. The method requires an oracle to supervise the selection of the trace, and an examination of one trace will fail to infer types that hold universally across the whole program.

Somewhat surprisingly, Bayesian unsupervised learning has been applied to recognise structure in memory images [31]. The memory image is scanned, looking for all pointers, which are then used to locate the positions of objects and their size, which are bounded by the distance to the next object. Unsupervised learning is then used to classify malware according to its memory layout, a technique that could be taken further with static type recovery.

Recursive datatypes Mycroft [100] recognised that type reconstruction could rule out inconsistent decompilation steps and thereby aid program reconstruction. This link is formalised in the decompilation relation that is the centrepiece of the formalisation of Chapter 4. His work was inspired by the desire to synthesise datatypes from register transfer language (RTL) code generated from BCPL, which itself is untyped, not distinguishing between arrays and structures. He discussed the issue of padding, which arises when some of the fields of a structure, but not all, can be inferred, as well as proposing a type unification algorithm for synthesising a recursive datatype when a type variable is unified with a term containing it.

SecondWrite [42] extends work on variable recovery [4, 5] with so-called best effort pointer analysis [42, Section 5.2] to infer some datatypes: they “dig into the points-to set to discover if it is pointing to an address which is declared as the starting point of a structure”. Generic types are used for symbols for which they cannot infer types, and type casts are introduced to convert the generic type to the actual types used in an operation.

Following the idea that “well-typed programs cannot go wrong” [91], type recovery has been muted as a check for the validity of low-level code [105]. Recursive types are recovered using a rational-tree solver from the low-level typeless template code of a graph reduction machine. If the solver fails, the code is judged unsafe.

Shape analysis Shape analysis [123] aims to determine the shape of objects in the heap by creating a map of how heap memory cells relate to one another. This is achieved by building a summary of the heap structures that can arise at each program point. Shape analysis can detect cycles within heap objects, categorise objects as pertaining to a given shape (trees, lists etc), identify data sharing/aliasing between objects, and track object lifetime and reachability. The technique is normally applied statically for verification of source code, however there are a number of works on shape analysis of execution traces ([18, 51, 73]) which bear similarities to binary type reconstruction (as noted elsewhere [19]).

Shape analysis can provide richer results than type reconstruction: for example the work presented in this thesis is able to derive recursive linked list types, while shape analysis may also be able to determine whether a program creates cycles in the heap from linked list nodes. Berdine et al. have even shown [12] that static shape analysis can recover mutually recursive data structures from source code. However, recursive object recovery thus far eludes *static* binary shape analysis, with all existing approaches utilising execution traces [19]. Static binary shape analysis faces a scalability issue, as it is expensive even on source code.

Verified decompilation and disassembly Decompilation is not always to C: Java bytecode has been decompiled into recursive functions, based on type theory [75], which is amenable to formal reasoning. Worthy of particular note, is the decompilation of machine code into the language of HOL4 [101]. With a view to proving full functional correctness, machine code is decompiled into tail-recursive functions. These functions describe the effect of the machine code, yet offer a layer of abstraction above it. Properties proved for the function are, by an automatically derived theorem, related to the original machine code, so the decompiler does not need to be proved correct. Recursive predicates could be defined in HOL4 to assert that memory conforms to a recursive datatype, but for the purposes of engaging with the reverse engineer, it seems more natural to decompile to a type-safe dialect of C.

Disassembly, the act of decoding the bit patterns of machine instructions into a textual representation, is itself non-trivial for self-modifying code. Self-modification is used to disguise malware but also arises in JIT compilation. In general, disassembly requires indirect jump targets to be computed, which can be approximated by abstract interpretation [77]. In the case of self modifying code, each memory write needs to be checked to determine how it modifies the code base. Modifications to the code base themselves entail a form of abstract decoding in which the analyser does not recover the exact instruction, but a collection of applicable instructions. Nevertheless disassembly of self-modifiying code has been formalised [16], though the topic is beyond the scope of this thesis.

Trusted compilation Further afield, is the wide body of work on trusted compilation, most notably represented by the CompCert project that produced a fully certified optimising C compiler [82]. The CompCert compiler transforms source code into machine code incrementally through a large number of intermediate languages, each of which is designed to handle a specific compilation stage such as common subexpression elimination, register allocation or control flow linearisation. Correctness is verified at each successive level of transformation with proofs created using a proof assistant. Where CompCert aims at proving semantic correctness of these optimisations, the interest of Chapter 4 is in type preservation and any witness, no matter how closely it mirrors the binary, is sufficient for the type correctness argument.

Typed Assembly Language (TAL) [96] represents another approach to trusted compilation, where the aim is to prove that type consistency is maintained through compilation to an assembly language that can be type checked to prove safety properties of the executable code. The limitation is that no machine exists that can execute TAL, so as a compromise the code is type checked either when assembled to machine code or by a runtime loader that recognises its special typed object format. In contrast Chapter 4 provides a type inference algorithm for assembler that allows type checking without the presence of any explicit type information in the binary.

6.2 Satisfiability Modulo Theories

To briefly recap, Satisfiability Modulo Theories (SMT) [103] describes a class of problems that consist of formulae in a given first-order theory, composed with Propositional connectives. Each theory formula is thus an atom in the Boolean satisfiability (SAT) [34] problem formed by the Propositional connectives. And so a solution to an SMT instance must satisfy both the skeleton of the SAT problem and the underlying theory problem, given by the entailed theory formulae.

SAT is perhaps the best exemplar of an NP-complete problem, and is encompassed by SMT, therefore it is of no surprise that SMT problems may have even greater worst-case complexity [9]. Yet DPLL(T) [46, 103], which provides a general scheme for SMT solving by augmenting the DPLL algorithm for SAT solving [34] with a solver for a theory T, has shown that it is indeed possible to solve many practical SMT problems. Further, since the conception of DPLL(T) there have been many refinements.

Conflict Driven Clause Learning (CDCL) [103] is one of the primary heuristics used by most DPLL SAT and DPLL(T) SMT solvers. However, an alternative that is more suited to implementation in a logic programming setting is lookahead [83]. Lookahead can be considered the dual of clause learning since the former seeks to avoid inconsistency by considering assignments that are still to be made, whereas the latter diagnoses an inconsistency from an assignment that has previously been made. The case for lookahead versus learning has been studied [83], but in a declarative context, particularly one where backtracking is supported, lookahead is very simple to implement.

Rational trees are commonly employed to type-check recursive types, as well as being a primary component of logic programming languages. Disjunctive rational tree constraints were employed by Mycroft's seminal work on type recovery [100] (though it is not explained how to solve such systems), and Demoen's work [36] shows how type inference for ad-hoc polymorphism is reduced to the same problem, and demonstrates a CLP solver that is similar (in principle at least) to that of Chapter 4. However, solving disjunctive systems of rational trees as part of an SMT solving framework (as in Chapter 3) is entirely novel.

6.3 The Octagon domain

There is arguably no better exemplar of a weakly relational abstract domain than octagons. Minè defined the octagon domain in his thesis [93] and subsequent journal paper [94] and developed an open source implementation [69]. By using DBMs, Minè was able to exploit existing algorithms for solving difference constraints. However the encoding of octagonal constraints into differences requires some conditions, encapsulated in the notion of strong closure, to define a canonical representation of octagons using DBMs. Minè [93, 94] showed that strong closure was cubic and that, by swapping rows and columns in the DBM, an incremental version of the algorithm could be derived. Independently a faster algorithm for strong closure was discovered [3], based on the observation that strong closure could be decomposed into two separate algorithmic phases. The incremental algorithm presented in Chapter 5 was inspired by a refinement to the Floyd-Warshall algorithm that was suggested for disjunctive spatial reasoning for solving constraint satisfaction problems [10], and the research question of whether, when adapted to DBMs and octagons, the refinement could improve on Minè 's incremental algorithm. The solver proposed in [10] resembles another independently proposed [55] for incrementally solving integer unit two-variable constraints. Though incremental, the integer solver [55] does not decompose constraint solving into the layers of closure, strengthening and consistency checking, which appears to be important for overall efficiency [3]. The work presented in Chapter 5 can be viewed as bringing incrementality to a decomposed solver architecture for DBMs.

Chapter 7

Conclusions

In summary, this thesis has demonstrated how solvers for Satisfiability Modulo Theories (SMT) and Constraint Handling Rules (CHR) can be applied to solve the problem of machine code type reconstruction, and answered the question of how types can be recovered that truly have semantic meaning (by the incidental construction of the first semantics preserving decompiler). Along the way, it has been shown that the logic programming setting garners many benefits for an SMT framework, enabling rapid implementation of solvers that are both concise and efficient. Furthermore, the implementation of a solver for the theory of Quantifier Free Integer Difference Logic (QF_IDL) led to an interesting contribution to the octagon abstract domain in the form of an improved incremental closure algorithm, which has further applications in machine code analysis.

7.1 Reflection upon Chapters 2 and 3

Chapter 2 introduced an SMT solving framework for Prolog based on reification. The motivation was the problem of type reconstruction, and it was demonstrated that the type reconstruction problem can be expressed as an SMT instance over the theory of rational-tree unification, a theory not available in existing SMT solvers. Rational-tree unification is a core Prolog language feature, so Prolog was a natural fit for type reconstruction. Reification in Prolog is equally suited to SMT solving, because it affords entirely automatic synchronisation of unit and theory propagation, alleviating the programmer of the need to algorithmically orchestrate the interleaving of the two processes (as in classic DPLL(T) algorithms).

Chapter 3 demonstrated the instantiation of the framework with solvers for three theories – rational-tree unification, linear arithmetic and integer difference constraints. The effectiveness of the approach was demonstrated by the successful application of the solver to a suite of type recovery and integer difference logic problems. The performance of the QF_IDL solver was particularly pleasing.

A complementary algorithm for finding an unsatisfiable core of an SMT problem was also presented in Chapter 2, and its correctness argued. Notably, the new algorithm is more aggressive at pruning the search space than the related QuickXplain algorithm [74].

The framework could be extended by providing decision procedures for further theories. Finite domain solvers, such as SICStus CLP(FD), often allow reified constraints [21], hence finite domain constraints might appear a good candidate to incorporate into the DPLL(T) framework. Unfortunately, finite domain constraint solvers typically maintain stores that are potentially inconsistent, hence without labelling (an unattractive step) a decision procedure for conjunctions of theory constraints is not readily available. That said, finite domain techniques have been applied to infer typings for the predicates of logic programs that are disjunctive [36]. Like the work of Chapters 2 and 3, this approach avoids the need for fixpoint computation, and its use of propagator constraints echoes theory propagation. However, this work assumes type definitions are prescribed up-front and recasting type recovery as SMT, with use of the core algorithm, can in principle resolve inconsistencies by applying MaxSMT.

The performance of the solver on the type recovery problems might be further improved by tailoring the search heuristics to the structure of these problems. One approach to this might be to incorporate learning. However, although [64] demonstrated how this can be realised in a Prolog based solver, learning is not a natural fit with the current approach.

7.2 Reflection upon Chapter 4

Chapter 4 demonstrated that strong guarantees of the validity of types recovered from machine code programs can be provided through a semantically founded approach. The solution also derives a high-level witness program alongside the types, and thus provides a type-based decompiler. By providing the witness in a type-safe language, and proving it preserves the semantics of the binary (in terms of memory consistency), it has been demonstrated irrefutably that the binary is faithful to the inferred types. The decompiler was evaluated on two dozen textbook datastructure manipulation programs and, for all, recovered a witness program in a type-safe dialect of C, complete with the original recursive datatypes.

Leading on from this work, the next logical step is to machine verify the proofs. Further than that, the type system, languages and formalisation could be expanded to support more machine code constructs exhibited by real binaries, for example those produced by C unions or casts, implicit sign and zero extension, and by C++ objects. MinX could also be dropped in favour of, for example, LLVM bitcode, which shares many similarities with MinX and would enable evaluation directly against real world binaries, without the need for a compiler from C to MinX. Interestingly, it should also be possible to derive higher level types than those present in the original source code, for example dependent types might be inferred from programs originally written in C.

7.3 Reflection upon Chapter 5

Chapter 5 introduced a new algorithm for computing incremental closure, that offers a significant performance improvement over prior algorithms. The chapter also shows that the clarity of the new algorithm lends itself to a simpler proof strategy, and how that strategy can be applied to the classic non-incremental closure algorithm. The improved performance of the new algorithm is backed up by extensive testing.

Further work has already been undertaken, which demonstrates further speed-ups for incremental closure (but is pending review) [24]. The octagon domain is used for many applications due to its expressiveness and ease of implementation, relative to other relational abstract domains. Future work will be to demonstrate the use of the domain to model modulo arithmetic. The octagon domain is ideally suited to this application, since wrapping can be handled by repeatedly adding single constraints to linear systems [111].

7.4 Closing remark

The question of whether real world reverse engineering problems can be solved using principled tools and advanced constraint solvers has not been fully answered by this thesis. However, it can be hoped that through the novelties found herein, a worthwhile contribution has been made.

Appendix A

Proof Appendix

A.1 Semantics-Driven Decompilation

A.1.1 Type Safety

We write $\Sigma; \Psi \vdash \sigma; \pi$ to signify that the store is type consistent:

$$\forall (a : \theta) \in \Psi \; . \; \Sigma; \Psi; \sigma; \pi \vdash a : \theta$$

We write $\Gamma; \Sigma; \Psi \vdash \rho$ to signify that the local environment is type consistent:

$$\forall (x : \theta) \in \Gamma \; . \; \Sigma; \Psi \vdash \rho(x) : \theta * \wedge \rho(x) \neq 0$$

Moreover, we write $\Gamma; \Sigma \vdash \lambda_c$ to signify that statements in λ_c are type consistent:

$$\forall s \in \mathit{range}(\lambda_c). \; \Gamma; \Sigma \vdash s$$

Proposition 2 (Preservation of MinC Expressions)**.**
If

- $\Gamma; \Sigma; \Psi \vdash \rho$
- $\Sigma; \Psi \vdash \sigma; \pi$
- $\Gamma; \Sigma \vdash e : \theta$

- $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, e \rangle \xrightarrow{e} \langle \sigma', \pi', v \rangle$

then, for some $\Psi' \supseteq \Psi$

1. $\Gamma; \Sigma; \Psi' \vdash \rho$
2. $\Sigma; \Psi' \vdash \sigma'; \pi'$
3. $\Sigma; \Psi' \vdash v : \theta$

Proposition 3 (Progress of MinC Expressions)**.**
If

- $\Gamma; \Sigma; \Psi \vdash \rho$
- $\Sigma; \Psi \vdash \sigma; \pi$
- $\Gamma; \Sigma \vdash e : \theta$

then

1. $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, e \rangle \xrightarrow{e} \langle \sigma', \pi', v \rangle$ or
2. $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, e \rangle \xrightarrow{e} \mathsf{err}$.

Proposition 4 (Preservation of MinC lvalues)**.**
If

- $\Gamma; \Sigma; \Psi \vdash \rho$
- $\Sigma; \Psi \vdash \sigma; \pi$
- $\Gamma; \Sigma \vdash \ell : \theta$
- $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \ell \rangle \xrightarrow{\ell} \langle \sigma', \pi', a \rangle$

then, for some $\Psi' \supseteq \Psi$

1. $\Gamma; \Sigma; \Psi' \vdash \rho$
2. $\Sigma; \Psi' \vdash \sigma'; \pi'$

3. $\Sigma; \Psi' \vdash a : \theta*$

Proposition 5 (Progress of MinC lvalues)**.**
If

- $\Gamma; \Sigma; \Psi \vdash \rho$
- $\Sigma; \Psi \vdash \sigma; \pi$
- $\Gamma; \Sigma \vdash \ell : \theta$

then

1. $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \ell \rangle \xrightarrow{\ell} \langle \sigma', \pi', a \rangle$ or
2. $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \ell \rangle \xrightarrow{\ell} \mathsf{err}$.

Proposition 6 (Preservation for MinC statements)**.**
If

- $\Gamma; \Sigma \vdash s$
- $\Sigma; \lambda_c; \vec{\rho}; \rho \vdash \langle \sigma, \pi, s \rangle \xrightarrow{s} \langle \sigma', \pi', s' \rangle$
- $\Gamma; \Sigma; \Psi \vdash \rho$
- $\Sigma; \Psi \vdash \sigma; \pi$

then for some $\Psi' \supseteq \Psi$

1. $\Gamma; \Sigma; \Psi' \vdash \rho$
2. $\Sigma; \Psi' \vdash \sigma'; \pi'$
3. $\Gamma; \Sigma \vdash s'$

Proposition 7 (Progress for MinC statements)**.**
If

- $\Gamma; \Sigma; \Psi \vdash \rho$
- $\Sigma; \Psi \vdash \sigma; \pi$

- $\Gamma; \Sigma \vdash s$
- $\Gamma; \Sigma \vdash \lambda_c$

then

1. $\Sigma; \lambda_c; \vec{\rho}; \rho \vdash \langle \sigma, \pi, s \rangle \xrightarrow{s} \langle \sigma', \pi', s' \rangle$ or
2. $\Sigma; \lambda_c; \vec{\rho}; \rho \vdash \langle \sigma, \pi, s \rangle \xrightarrow{s} \mathsf{err}$ or
3. $s = \mathsf{return}$.

Proposition 8 (Preservation for MinC functions)**.**
If

- $\Sigma \vdash f(\overrightarrow{x : \theta})\langle \overrightarrow{y : \theta'}, l, \lambda_c, j \rangle$
- $\Sigma; \lambda_c; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \lambda_c(l) \rangle \xrightarrow{s}{}^{*} \langle \sigma', \pi', \mathsf{return} \rangle$
- $\Gamma = \{\overrightarrow{x : \theta}, \overrightarrow{y : \theta'}\}$
- $\Gamma; \Sigma; \Psi \vdash \rho$
- $\Sigma; \Psi \vdash \sigma; \pi$

then, for some $\Psi' \supseteq \Psi$

1. $\Gamma; \Sigma; \Psi' \vdash \rho$
2. $\Sigma; \Psi' \vdash \sigma'; \pi'$

Proposition 9 (Progress for MinC functions)**.**
If

- $\Sigma \vdash f(\overrightarrow{x : \theta})\langle \overrightarrow{y : \theta'}, l, \lambda_c, j \rangle$
- $\Sigma; \lambda_c; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \lambda_c(l) \rangle \xrightarrow{s}{}^{*} \langle \sigma', \pi', \mathsf{return} \rangle$
- $\Gamma = \{\overrightarrow{x : \theta}, \overrightarrow{y : \theta'}\}$
- $\Gamma; \Sigma; \Psi \vdash \rho$

- $\Sigma; \Psi \vdash \sigma; \pi$

then

1. $\Sigma; \lambda_c; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \lambda_c(l) \rangle \xrightarrow{s}{}^* \langle \sigma', \pi', \mathsf{return} \rangle$ or
2. $\Sigma; \lambda_c; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \lambda_c(l) \rangle \xrightarrow{s}{}^* \mathsf{err}$ (we assume this subsumes divergence).

Proof. Propositions 2 to 9 are proved together by mutual structural induction on the typing judgements for ℓ, e, s and d_c.

- By case analysis on $\Gamma; \Sigma \vdash e : \theta$ in Figure 16. To show that either 3.2 or conversely 3.1, 2.1, 2.2 and 2.3 hold. Observe that 2.1 holds if $\Psi' \supseteq \Psi$.

  1. Let $e : \theta = c_l : \mathsf{long}$. By rule e-const $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, c_l \rangle \xrightarrow{e} \langle \sigma, \pi, c_l \rangle$. Hence 3.1.

     Let $\Psi' = \Psi$. By rule vt-l $\Sigma; \Psi \vdash c_l : \mathsf{long}$. Hence 2.3. Also 2.2.
  2. Let $e : \theta = c_s : \mathsf{short}$. By rule e-const $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, c_s \rangle \xrightarrow{e} \langle \sigma, \pi, c_s \rangle$. Hence 3.1.

     Let $\Psi' = \Psi$. By rule vt-s $\Sigma; \Psi \vdash c_s : \mathsf{short}$. Hence 2.3. Also 2.2.
  3. Let $e : \theta = 0_l : \tau*$. By rule e-const $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, 0_l \rangle \xrightarrow{e} \langle \sigma, \pi, 0_l \rangle$. Hence 3.1.

     Let $\Psi' = \Psi$. By rule vt-null $\Sigma; \Psi \vdash c_s : \tau*$. Hence 2.3. Also 2.2.
  4. Let $e : \theta = \mathsf{new}\ \tau : \tau*$. By rule e-new $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \mathsf{new}\ \tau \rangle \xrightarrow{e} \langle \sigma', \pi, a \rangle$ where $\sigma' = \sigma \circ \{a \mapsto \bot\}$. Hence 3.1.

     Let $\Psi' = \Psi \circ \{a \mapsto \tau\}$. By rule vt-addr $\Sigma; \Psi \vdash a : \tau*$ hence 2.3. Also by rule vt-bot $\Sigma; \Psi' \vdash \bot : \tau$ by and rule st-comp $\Sigma; \Psi'; \sigma'; \pi \vdash a : \tau$ hence $\Sigma; \Psi' \vdash \sigma'; \pi$ and 3.2 holds.
  5. Let $e : \theta = \mathsf{new\ struct}\ N : N*$ and $n = |\Sigma(N)|$. By rule e-str $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \mathsf{new\ struct}\ N \rangle \xrightarrow{e} \langle \sigma', \pi', a \rangle$ where $\sigma' = \sigma \circ \{a \mapsto \bot, \ldots, a+n-1 \mapsto \bot\}$ and $\pi' = \pi \cup \{[a, a+n-1]\}$. Put $\Psi' = \Psi \cup \{a : N, a+1 : \theta_1, \ldots, a+n-1 : \theta_{n-1}\}$. By rule vt-addr $\Sigma; \Psi' \vdash a : N*$ hence 2.3 holds.

     Let $i \in [0, n-1]$. Then $\sigma'(a+i) = \bot$ hence $\Sigma; \Psi' \vdash \sigma'(a+i) : \theta_i$ by rule vt-bot therefore $\Sigma; \Psi'; \sigma'; \pi' \vdash a+i : \theta_i$. By rule st-fld $\Sigma; \Psi'; \sigma'; \pi' \vdash a : N$ hence 2.2 holds.

6. Let $e : \theta = \mathsf{new}\ \theta[e] : \theta[]*$. By rule t-new-ar $\Gamma; \Sigma \vdash e : t$ hence by induction:
   - Either $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, e \rangle \xrightarrow{e} \mathsf{err}$. By rule e-ar-err $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \mathsf{new}\ \theta[e] \rangle \xrightarrow{e} \mathsf{err}$. Hence 3.2.
   - Or $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, e \rangle \xrightarrow{e} \langle \sigma', \pi', v \rangle$. By rule e-ar $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \mathsf{new}\ \theta[e] \rangle \xrightarrow{e} \langle \sigma'', \pi'', a \rangle$ where $\sigma'' = \sigma' \circ \{a \mapsto \bot, \dots, a + v - 1 \mapsto \bot\}$. Hence 3.1. By induction there exists $\Phi' \supseteq \Phi$ such that $\Sigma; \Psi' \vdash \sigma'; \pi'$. Put $\Psi'' = \Psi' \circ \{a \mapsto \theta[], \dots, a + v - 1 \mapsto \theta[]\}$. By rule vt-addr it follows $\Sigma; \Psi'' \vdash a : \theta[]*$ hence 2.3. By rule vt-bot it follows $\Sigma; \Psi'' \vdash \bot : \theta[]$ and by st-comp it follows $\Sigma; \Psi''; \sigma''; \pi'' \vdash a{+}i : \theta[]$ for all $i \in [0, v{-}1]$ hence 2.2.
7. Let $e : \theta = (e_1 \oplus e_2) : t$. By rule t-$\otimes$ $\Gamma; \Sigma \vdash e_1 : t$ and $\Gamma; \Sigma \vdash e_2 : t$. Hence by induction:
   - Either $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, e_1 \rangle \xrightarrow{e} \mathsf{err}$. By rule e-op-err$_1$ $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, (e_1 \oplus e_2) \rangle \xrightarrow{e} \mathsf{err}$. Hence 3.2.
   - Or $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma', \pi', e_2 \rangle \xrightarrow{e} \mathsf{err}$. Like previous case.
   - Or $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, e_1 \rangle \xrightarrow{e} \langle \sigma', \pi', v_1 \rangle$ and $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma', \pi', e_2 \rangle \xrightarrow{e} \langle \sigma'', \pi'', v_2 \rangle$.
     * Either $v_1 \oplus_\pi v_2 = \mathsf{err}$. By rule e-op-err$_3$ $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, (e_1 \oplus e_2) \rangle \xrightarrow{e} \mathsf{err}$. Hence 3.2.
     * Or $v_1 \oplus_\pi v_2 = v$. By rule e-op $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, (e_1 \oplus e_2) \rangle \xrightarrow{e} \langle \sigma', \pi, v \rangle$. Hence 3.1.
       By induction $\Sigma; \Psi'' \vdash v_1 : t$ and $\Sigma; \Psi'' \vdash v_2 : t$. If $t = \mathsf{short}$ then $v = \bot$ or $v = n_s$ where $n \in [-2^{15}, 2^{15} - 1]$. If $v = \bot$ then $\Sigma; \Psi'' \vdash v : \mathsf{short}$. by rule vt-bot. Otherwise if $v = n_s$ then $\Sigma; \Psi'' \vdash v : \mathsf{short}$ by rule vt-s. An analgous argument holds if $t = \mathsf{long}$ hence 2.3. Also 2.2 trivially by induction.
8. Let $e : \theta = (e_1 \oplus e_2) : \tau[]*$. Similar to previous case.
9. Let $e : \theta = f(\vec{e}) : \theta_j$. By rule t-call $\Gamma; \Sigma \vdash e_i : \theta'_i$ where $\phi_c(f) = f(\overrightarrow{x : \theta})\langle \overrightarrow{y : \theta''}, l, \lambda_c, j \rangle$ and $\Sigma \vdash \vec{\theta'} <: \vec{\theta}$. With respect to e_i there are two possibilities:

- Either for some i: $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma_{i-1}, \pi_{i-1}, e_i \rangle \xrightarrow{e} \mathsf{err}$. Then by rule e-call-err it follows that 3.2 holds.
- Or for all i: $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma_{i-1}, \pi_{i-1}, e_i \rangle \xrightarrow{e} \langle \sigma_i, \pi_i, v_i \rangle$ and by the inductive hypothesis $\Sigma; \Psi_i \vdash \theta_i : v_i$ and $\Sigma; \Psi_i \vdash \sigma_i; \pi_i$. Let $\Psi' = \Psi_n \cup \{\overrightarrow{a : \theta}, \overrightarrow{a' : \theta'}\}$. Then it is easy to verify $\Sigma; \Psi' \vdash \sigma'; \pi_n$ and $\Gamma; \Sigma; \Psi' \vdash \rho'$. By the progress induction hypothesis we then have for s:
  - Either $\Sigma; \lambda_c; \vec{\rho}, \rho; \rho' \vdash \langle \sigma', \pi_n, \lambda_c(l) \rangle \xrightarrow{s} {}^*\langle \sigma'', \pi', \mathsf{return} \rangle$. Hence 3.1.
  - Otherwise 3.2.

  Preservation follows from the induction hyptheses for all e_i and s.

- By case analysis on $\Gamma; \Sigma \vdash \ell : \theta$ in Figure 16. To show 5.2 or conversely 5.1, 4.1, 4.2 and 4.3 hold. Observe that 4.1 holds if $\Psi' \supseteq \Psi$.

  1. Let $\ell = x$. By rule l-var $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x \rangle \xrightarrow{\ell} \langle \sigma, \pi, a \rangle$ where $a = \rho(x)$ hence 5.1 holds. Put $\Psi' = \Psi$. Since $\Gamma; \Sigma; \Psi \vdash \rho$ it follows $\Sigma; \Psi' \vdash \rho(x) : \theta*$ and 4.3 holds. Moreover $\Sigma; \Psi' \vdash \sigma; \pi$ and 4.2 holds.
  2. Let $\ell : \theta = *x : \tau$. Since $\Gamma; \Sigma; \Psi \vdash \rho$ it follows $a = \rho(x) \neq 0$. By rule l-ptr $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, *x \rangle \xrightarrow{\ell} \langle \sigma, \pi, \sigma(a) \rangle$ thus 5.1 holds. Put $\Psi' = \Psi$. By rule t-ptr $\Gamma; \Sigma \vdash x : \tau*$ and by $\Gamma; \Sigma; \Psi \vdash \rho$ it follows $\Sigma; \Psi \vdash a : \tau**$. By rule vt-addr $(a : \tau*) \in \Psi$ and by $\Sigma; \Psi \vdash \sigma; \pi$ it follows $\Sigma; \Psi; \sigma; \pi \vdash a : \tau*$. By rule st-comp $\Sigma; \Psi \vdash \sigma(a) : \tau*$ thus $\Sigma; \Psi' \vdash \sigma(a) : \tau*$ and 4.3 holds. Moreover $\Sigma; \Psi' \vdash \sigma; \pi$ and 4.2 holds.
  3. Let $\ell : \theta = x \to c : \theta_c$. Since $\Gamma; \Sigma; \Psi \vdash \rho$ let $a = \rho(x) \neq 0$ and let $v = \sigma(a) +_\perp c$. If $\rho(x) = 0$ or $v \notin \cup\pi$ then 5.2 holds. Otherwise $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x \to c \rangle \xrightarrow{\ell} \langle \sigma, \pi, v \rangle$ and 5.1 holds. Put $\Psi' = \Psi$. By rule t-fld $\Gamma; \Sigma \vdash x : N*$ and by rule t-var $(x : N*) \in \Gamma$ and by $\Gamma; \Sigma; \Psi \vdash \rho$ it follows $\Sigma; \Psi \vdash \rho(x) : N**$. By rule vt-addr $(\rho(x) : N*) \in \Psi$ and by $\Sigma; \Psi \vdash \sigma; \pi$ it follows $\Sigma; \Psi; \sigma; \pi \vdash \rho(x) : N*$ and by rule st-comp $\Sigma; \Psi \vdash \sigma(\rho(x)) : N*$. By rule vt-addr $(\sigma(\rho(x)) : N) \in \Psi$ and by $\Gamma; \Sigma; \Psi \vdash \rho$ it follows $\Sigma; \Psi; \sigma; \pi \vdash \sigma(\rho(x)) : N$ and by rule st-fld $\Sigma; \Psi \vdash \sigma(\sigma(\rho(x)) + c) : \theta_c$. By rule st-comp $\Sigma; \Psi; \sigma; \pi \vdash \sigma(\rho(x)) + c : \theta_c$ and by $\Gamma; \Sigma; \Psi \vdash \rho$ it follows

$(\sigma(\rho(x)) + c : \theta_c) \in \Psi$ and by rule vt-addr $\Sigma; \Psi \vdash \sigma(\rho(x)) + c : \theta_c*$ and 4.3 holds since $\Psi' = \Psi$. Moreover $\Sigma; \Psi' \vdash \sigma; \pi$ and 4.2 holds.

4. Let $\ell = x[e']$. By rule t-ar $\Gamma; \Sigma \vdash e' : t$ hence by mutual induction:
   - Either $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, e' \rangle \xrightarrow{e} \mathsf{err}$. By rule e-lval-err $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x[e'] \rangle \xrightarrow{e} \mathsf{err}$. Hence 5.2.
   - Or $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, e' \rangle \xrightarrow{e} \langle \sigma', \pi', v \rangle$. If $\rho(x) = 0$ then 5.1 holds by rule e-lval-err. Otherwise let $a = \sigma'(\rho(x)) +_\perp v$. If $a \notin \cup\pi'$ then 5.1 holds. Otherwise by rule l-ar $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x[e'] \rangle \xrightarrow{\ell} \langle \sigma', \pi', a \rangle$. Hence 5.1 holds.

     By induction there exists $\Psi' \supseteq \Psi$ such that $\Sigma; \Psi' \vdash \sigma'; \pi'$. By rule t-ar $\Gamma; \Sigma \vdash x : \theta[]*$ and by rule t-var $(x : \theta[]*) \in \Gamma$ and by $\Gamma; \Sigma; \Psi' \vdash \rho$ it follows $\Sigma; \Psi' \vdash \rho(x) : \theta[]**$. By rule vt-addr $(\rho(x) : \theta[]*) \in \Psi'$ and by $\Sigma; \Psi' \vdash \sigma'; \pi'$ it follows $\Sigma; \Psi'; \sigma'; \pi' \vdash \rho(x) : \theta[]*$ and by rule st-comp $\Sigma; \Psi' \vdash \sigma'(\rho(x)) : \theta[]*$. By rule vt-addr $(\sigma'(\rho(x)) : \theta[]) \in \Psi'$ and by $\Gamma; \Sigma; \Psi' \vdash \rho$ it follows $\Sigma; \Psi'; \sigma'; \pi' \vdash \sigma'(\rho(x)) : \theta[]$ and by rule st-ar $\Sigma; \Psi' \vdash \sigma'(\sigma'(\rho(x)) + v) : \theta$. By rule st-comp $\Sigma; \Psi'; \sigma'; \pi' \vdash \sigma'(\rho(x)) + v : \theta$ and by $\Gamma; \Sigma; \Psi' \vdash \rho$ it follows $(\sigma'(\rho(x)) + v : \theta) \in \Psi'$ and by rule vt-addr $\Sigma; \Psi' \vdash \sigma'(\rho(x)) + v : \theta*$ and 4.3 holds. Moreover $\Sigma; \Psi' \vdash \sigma; \pi$ and 4.2 holds.

- By case analysis on $\Gamma; \Sigma \vdash s$ in Figure 16. To show that either 7.2 or conversely 7.1 or 7.3, and 6.1, 6.2 and 6.3 hold. Observe that 6.1 holds if $\Psi' \supseteq \Psi$.

  1. Let $\Gamma; \Sigma \vdash (\ell := e); s$. From the induction hypothesis for ℓ, either $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \ell \rangle \xrightarrow{\ell} \mathsf{err}$, and hence 7.2, or $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \ell \rangle \xrightarrow{\ell} \langle \sigma', \pi', a \rangle$. In the latter case, we have either $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma', \pi', e \rangle \xrightarrow{e} \mathsf{err}$, and hence 7.2, or $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma', \pi', e \rangle \xrightarrow{e} \langle \sigma'', \pi'', v \rangle$. By s-assn we then have $\Sigma; \lambda_c; \vec{\rho}; \rho \vdash \langle \sigma, \pi, (\ell := e); s \rangle \xrightarrow{s} \langle \sigma''', \pi'', s \rangle$ where $\sigma''' = \sigma'' \circ \{a \mapsto v\}$ and hence 7.1.

     We get $\Gamma; \Sigma \vdash s$ from t-assn. Hence 6.3. From the induction hypotheses for ℓ and e we get type preservations $\Sigma; \Psi'' \vdash a : \theta_1*$ and $\Sigma; \Psi'' \vdash v : \theta_2$

and type consistency $\Sigma; \Psi'' \vdash \sigma''; \pi''$. Hence, through rule vt-addr we know that $(a : \theta_1) \in \Psi''$. From rule t-assn we know $\Sigma \vdash \theta_2 <: \theta_1$. Hence, through rule vt-subt we have $\Sigma; \Psi'' \vdash v : \theta_1$. Since $\sigma'''(a) = v$ we have hence by rule st-comp $\Sigma; \Psi''; \sigma'''; \pi'' \vdash a : \theta_1$. Hence $\Sigma; \Psi'' \vdash \sigma'''; \pi''$. Thus 6.2.

2. Let $\Gamma; \Sigma \vdash (\mathsf{if}\ e\ \mathsf{goto}\ l); s$. Then
   - Either $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, e \rangle \xrightarrow{e} \mathsf{err}$. Hence 7.2.
   - Or $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, e \rangle \xrightarrow{e} \langle \sigma', \pi', v \rangle$. Then
     * Either $v = \bot$. Hence 7.2.
     * Or $v = 0$. Then by rule s-if-false $\Sigma; \lambda_c; \vec{\rho}; \rho \vdash \langle \sigma, \pi, (\mathsf{if}\ e\ \mathsf{goto}\ l); s \rangle \xrightarrow{s} \langle \sigma', \pi s,' \rangle$. Hence 7.1. We call this scenario 1.
     * Or $v \neq 0 \wedge v \neq \bot$. Then
       · Either $l \notin dom(\lambda_c)$. Then 7.2.
       · Or $s' = \lambda_c(l)$. Then by rule s-if-true $\Sigma; \lambda_c; \vec{\rho}; \rho \vdash \langle \sigma, \pi, (\mathsf{if}\ e\ \mathsf{goto}\ l); s \rangle \xrightarrow{s} \langle \sigma', \pi s',' \rangle$. Hence 7.1. We call this scenario 2.

   In scenario 1 we have from t-if $\Gamma; \Sigma \vdash s$. Hence 6.3. In scenario 2 we have that $s' \in range(\lambda_c)$. Hence $\Gamma; \Sigma \vdash s'$. Hence 6.3. In both scenarios we have from the induction hypthesis for e that $\Sigma; \Psi' \vdash \sigma'; \pi'$. Hence 6.2.

3. Let $\Gamma; \Sigma \vdash \mathsf{goto}\ l$. Then either $l \notin dom(\lambda_c)$ and thus $\Sigma; \lambda_c; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \mathsf{goto}\ l \rangle \xrightarrow{s} \mathsf{err}$. Hence 6.2. Alternatively $\lambda_c(l) = s$. Then by rule s-goto $\Sigma; \lambda_c; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \mathsf{goto}\ l \rangle \xrightarrow{s} \langle \sigma, \pi, s \rangle$. Hence 7.1.

   From $\Gamma; \Sigma \vdash \lambda_c$ it follows that $\Gamma; \Sigma \vdash s$. Hence 6.3. Let $\Psi' = \Psi$. Then 6.2.

4. Let $\Gamma; \Sigma \vdash \mathsf{return}$. Hence 7.3. Also vacuously 6.3 and 6.2.

- Propositions 8 and 9 follow by the repeated application of Propositions 6 and 7, combining progress and preservation at every step.

  Besides the givens of Proposition 8, Proposition 6 also requires $\Gamma; \Sigma \vdash \lambda_c$. This is given by rule t-def which is the only possible way that the well-typing of the function definition could have been constructed.

∎

A.1.2 Well-Typed Decompilation

Proposition 10 (well-typed instruction decompilation)**.** If $\mu_\Gamma; \Gamma; \Sigma \vdash \iota \overset{\iota}{\leadsto} \ell := e$ then for some θ_1 and θ_2

1. $\Gamma; \Sigma \vdash \ell : \theta_1$
2. $\Gamma; \Sigma \vdash e : \theta_2$
3. $\Sigma \vdash \theta_2 <: \theta_1$

Proposition 11 (well-typed block decompilation)**.** If $\mu_\lambda; \mu_\Gamma; \Gamma; \Sigma \vdash b \overset{b}{\leadsto} s$ then $\Gamma; \Sigma \vdash s$.

Proposition 12 (well-typed definition decompilation)**.** If $\Sigma \vdash d_x \leadsto d_c$ then $\Sigma \vdash d_c$.

Proof. Propositions 10 through 12 are proved together through mutual structural induction over the judgements of the translation relation (Figures 20 through 22). This necessity is induced by the circularity of call instructions, which require a valid function definition for the target of the call.

- By case analysis on the inference rules of the instruction translation relation (Figures 20 and 21).

    1. Case tr-$\oplus$-r$^*_1$. Let $\theta_1 = \theta_2 = \theta[]*$. From tr-$\oplus$-r$^*_1$ we have $(x : \theta[]*) \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash x : \theta[]*$. Hence 10.1. From tr-$\oplus$-r$^*_1$ we have $\Gamma; \Sigma \vdash m : \mathsf{long}$. From tr-$\oplus$-r$^*_1$ we have $(y : \mathsf{long}) \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash y : \mathsf{long}$. From both of these we get by rule t-$\otimes$ $\Gamma; \Sigma \vdash y * m : \mathsf{long}$. From that and the type of x we get through rule t-ptr-$\oplus$ $\Gamma; \Sigma \vdash x \oplus (y * m) : \theta[]*$. Hence 10.2. From rule sub-refl 10.3.
    2. Case tr-$\oplus$-r$^*_2$. Let $\theta_1 = \theta_2 = t$. From tr-$\oplus$-r$^*_2$ we have $(x : t) \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash x : t$. Hence 10.1. From tr-$\oplus$-r$^*_2$ we have $\Gamma; \Sigma \vdash c : t$. From tr-$\oplus$-r$^*_1$ we have $(y : t) \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash y : t$. From both of these we get by rule t-$\otimes$ $\Gamma; \Sigma \vdash y{*}c : t$. From that and the type of x we get through rule t-$\otimes$ $\Gamma; \Sigma \vdash x \oplus (y * c) : t$. Hence 10.2. From rule sub-refl 10.3.

3. Case tr-$\otimes$-rc. Let $\theta_1 = \theta_2 = t$. From tr-$\otimes$-rc we have $(x : t) \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash x : t$. Hence 10.1. From tr-$\otimes$-rc we have $\Gamma; \Sigma \vdash c : t$. From that and the previous $\Gamma; \Sigma \vdash x : t$ we have by rule t-$\otimes$ $\Gamma; \Sigma \vdash x \otimes c : t$. Hence 10.2. From rule sub-refl 10.3.

4. Case tr-$\otimes$-rr. Let $\theta_1 = \theta_2 = t$. From tr-$\otimes$-rr we have $(x : t) \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash x : t$. Hence 10.1. From tr-$\otimes$-rr we have $(y : t) \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash y : t$. From that and the previous $\Gamma; \Sigma \vdash x : t$ we have by rule t-$\otimes$ $\Gamma; \Sigma \vdash x \otimes y : t$. Hence 10.2. From rule sub-refl 10.3.

5. Case tr-$\oplus$-rc. Let $\theta_1 = \theta_2 = \theta[]*$. From tr-$\oplus$-rc we have $(x : \theta[]*) \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash x : \theta[]*$. Hence 10.1. From tr-$\oplus$-rc we have $\Gamma; \Sigma \vdash m : t$. From that and the previous $\Gamma; \Sigma \vdash x : \theta[]*$ we have by rule t-ptr-$\oplus$ $\Gamma; \Sigma \vdash x \oplus m : \theta[]*$. Hence 10.2. From rule sub-refl 10.3.

6. Case tr-mov-rc. Let $\theta_1 = \theta_2 = t$. From tr-mov-rc we have $(x : t) \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash x : t$. Hence 10.1. From tr-mov-rc we have $\Gamma; \Sigma \vdash c : t$. Hence 10.2. From rule sub-refl 10.3.

7. Case tr-mov-r0. Let $\theta_1 = \theta_2 = \tau*$. From tr-mov-r0 we have $(x : \tau*) \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash x : \tau*$. Hence 10.1. From t-null we have $\Gamma; \Sigma \vdash 0 : \tau*$. Hence 10.2. From rule sub-refl 10.3.

8. Case tr-mov-rr. From tr-mov-rr we have $(x : \theta_1) \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash x : \theta_1$. Hence 10.1. From tr-mov-rr we have $(y : \theta_2) \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash y : \theta_2$. Hence 10.2. From tr-mov-rr we have $\Sigma \vdash \theta_2 <: \theta_1$. Hence 10.3.

9. Case tr-mov-ri$_1$. From tr-mov-ri$_1$ we have $(x : \theta_1) \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash x : \theta_1$. Hence 10.1. From tr-mov-ri$_1$ we have $(y : \theta_2*) \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash y : \theta_2*$. Then by rule t-ptr $\Gamma; \Sigma \vdash *y : \theta_2$. Hence 10.2. From tr-mov-ri$_1$ we have $\Sigma \vdash \theta_2 <: \theta_1$. Hence 10.3.

10. Case tr-mov-ir$_1$. From tr-mov-ir$_1$ we have $(x : \theta_1*) \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash x : \theta_1*$. Then by rule t-ptr $\Gamma; \Sigma \vdash *x : \theta_1$. 10.Hence 1. From tr-mov-ir$_1$ we have $(y : \theta_2) \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash y : \theta_2$. Hence 10.1. From tr-mov-ir$_1$ we have $\Sigma \vdash \theta_2 <: \theta_1$. Hence 10.3.

11. Case tr-mov-ri$_2$. From tr-mov-ri$_2$ we have $(x : \theta_1) \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash x : \theta_1$. Hence 10.1. From tr-mov-ri$_2$ we have $(y : \theta_2[]*) \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash y : \theta_2[]*$. Also by rule t-l $\Gamma; \Sigma \vdash 0 : \mathsf{long}$. Then by rule t-ar $\Gamma; \Sigma \vdash y[0] : \theta_2$. Hence 10.2. From tr-mov-ri$_2$ we have $\Sigma \vdash \theta_2 <: \theta_1$. Hence 10.3.
12. Case tr-mov-ir$_2$. From tr-mov-ir$_2$ we have $(x : \theta_1[]*) \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash x : \theta_1[]*$. Also by rule t-l $\Gamma; \Sigma \vdash 0 : \mathsf{long}$. Then by rule t-ar $\Gamma; \Sigma \vdash x[0] : \theta_1$. Hence 10.1. From tr-mov-ir$_2$ we have $(y : \theta_2) \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash y : \theta_2$. Hence 10.2. From tr-mov-ir$_2$ we have $\Sigma \vdash \theta_2 <: \theta_1$. Hence 10.3.
13. Case tr-mov-ri$_3$. From tr-mov-ri$_3$ we have $(x : \theta) \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash x : \theta$. Hence 10.1. From tr-mov-ri$_3$ we have $(y : N*) \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash y : N*$. Then by rule t-fld $\Gamma; \Sigma \vdash y \to 0 : \theta_0$. Hence 10.2. From tr-mov-ri$_3$ we have $\Sigma \vdash \theta_0 <: \theta$. Hence 10.3.
14. Case tr-mov-ir$_3$. From tr-mov-ir$_3$ we have $(x : N*) \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash x : N*$. Then by rule t-fld $\Gamma; \Sigma \vdash x \to 0 : \theta_0$. Hence 10.1. From tr-mov-ir$_3$ we have $(y : \theta) \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash y : \theta$. Hence 10.2. From tr-mov-ir$_3$ we have $\Sigma \vdash \theta <: \theta_0$. Hence 10.3.
15. Case tr-mov-ri+$_1$. From tr-mov-ri+$_1$ we have $(x : \theta_1) \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash x : \theta_1$. Hence 10.1. From tr-mov-ri+$_1$ we have $(y : \theta_2[]*) \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash y : \theta_2[]*$. Also from tr-mov-ri+$_1$ we have $\Gamma; \Sigma \vdash m : t$. Then by rule t-ar $\Gamma; \Sigma \vdash y[m] : \theta_2$. Hence 10.2. From tr-mov-ri+$_1$ we have $\Sigma \vdash \theta_2 <: \theta_1$. Hence 10.3.
16. Case tr-mov-i+r$_1$. From tr-mov-i+r$_1$ we have $(x : \theta_1[]*) \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash x : \theta_1[]*$. Also from tr-mov-i+r$_1$ we have $\Gamma; \Sigma \vdash m : t$. Then by rule t-ar $\Gamma; \Sigma \vdash x[m] : \theta_1$. Hence 10.1. From tr-mov-i+r$_1$ we have $(y : \theta_2) \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash y : \theta_2$. Hence 10.2. From tr-mov-i+r$_1$ we have $\Sigma \vdash \theta_2 <: \theta_1$. Hence 10.3.
17. Case tr-mov-ri+$_2$. From tr-mov-ri+$_2$ we have $(x : \theta) \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash x : \theta$. Hence 10.1. From tr-mov-ri+$_2$ we have $(y : N*) \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash y : N*$. Then by rule t-fld $\Gamma; \Sigma \vdash y \to m : \theta_m$. Hence 10.2. From tr-mov-ri+$_2$ we have $\Sigma \vdash \theta_m <: \theta$. Hence 10.3.

18. Case tr-mov-i+r$_2$. From tr-mov-i+r$_2$ we have $(x : N*) \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash x : N*$. Then by rule t-fld $\Gamma; \Sigma \vdash x \to m : \theta_m$. Hence 10.1. From tr-mov-i+r$_2$ we have $(y : \theta_2) \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash y : \theta_2$. Hence 10.2. From tr-mov-i+r$_2$ we have $\Sigma \vdash \theta <: \theta_m$. Hence 10.3.

19. Case tr-alloc-r$*$. From tr-alloc-r$*$ we have $(x : \theta[]*) \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash x : \theta[]*$. Hence 10.1. From tr-alloc-r$*$ we have $\Gamma; \Sigma \vdash m : t$. From tr-alloc-r$*$ we have $(y : t) \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash y : t$. From both of these we get by rule t-$\otimes$ $\Gamma; \Sigma \vdash y * m : t$. Then from t-new-ar we get $\Gamma; \Sigma \vdash \mathsf{new}\ \theta[y * m] : \theta[]*$. Hence 10.2. From rule sub-refl 10.3.

20. Case tr-alloc-rc$_1$. From tr-alloc-rc$_1$ we have $(x : \theta*) \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash x : \theta*$. Hence 10.1. From t-new we get $\Gamma; \Sigma \vdash \mathsf{new}\ \theta : \theta*$. Hence 10.2. From rule sub-refl 10.3.

21. Case tr-alloc-rc$_2$. From tr-alloc-rc$_2$ we have $(x : N*) \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash x : N*$. Hence 10.1. From t-new-str we get $\Gamma; \Sigma \vdash \mathsf{new}\ N : N*$. Hence 10.2. From rule sub-refl 10.3.

22. Case tr-alloc-rc$_3$. From tr-alloc-rc$_3$ we have $(x : \theta[]*) \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash x : \theta[]*$. Hence 10.1. From tr-alloc-rc$_3$ we have $\Gamma; \Sigma \vdash m : t$. Then from rule t-new-ar we have $\Gamma; \Sigma \vdash \mathsf{new}\ \theta[m] : \theta[]*$. Hence 10.2. From rule sub-refl 10.3.

23. Case tr-call. From tr-call we have $(u : \theta_u) \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash u : \theta_u$. Hence 10.1. We have:
    - From tr-call and by Proposition 12 we have $\phi_c(f) = f(\overrightarrow{x : \theta})\langle \overrightarrow{y : \theta'}, l, \lambda_c, j\rangle$.
    - From tr-call we have $\overrightarrow{(v : \theta_v)} \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash \vec{v} : \vec{\theta}_v$.
    - From tr-call we have $\Sigma \vdash \vec{\theta}_v <: \vec{\theta}$.
    - By rule sub-refl we have $\Sigma \vdash \theta'_j <: \theta'_j$.

    Hence by rule t-call we have $\Gamma; \Sigma \vdash: \theta'_j$. Hence 10.2. From tr-call we have $\Sigma \vdash \theta'_j <: \theta_u$. Hence 10.3.

- By structural induction on the block translation relation (Figure 22).

1. Case tr-instr. From tr-instr we have $\mu_\Gamma; \Gamma; \Sigma \vdash \iota \overset{\iota}{\rightsquigarrow} \ell := e$. Hence, by Proposition 10 we have $\Gamma; \Sigma \vdash \ell : \theta_1$, $\Gamma; \Sigma \vdash e : \theta_2$ and $\Sigma \vdash \theta_2 <: \theta_1$. Also by rule tr-instr we have $\mu_\lambda; \mu_\Gamma; \Gamma; \Sigma \vdash b \overset{b}{\rightsquigarrow} s$. Hence by the induction hypothesis we have $\Gamma; \Sigma \vdash s$. Then by rule t-assn we have $\Gamma; \Sigma \vdash \ell := e; s$

2. Case tr-if. From tr-if we have $(x : \theta) \in \Gamma$. Then by rule t-var $\Gamma; \Sigma \vdash x : \theta_u$. Also from tr-if we have $\mu_\lambda; \mu_\Gamma; \Gamma; \Sigma \vdash b \overset{b}{\rightsquigarrow} s$. Hence, from the induction hypothesis we have $\Gamma; \Sigma \vdash s$ Then the Proposition follows from rule t-if.

3. Case tr-goto. This follows from rule t-goto.

4. Case tr-ret. This follows from rule t-ret.

- By showing that the four preconditions to rule t-def (Figure 22) are satisfied:

  1. From rule tr-def we know that $\Gamma = \{\overrightarrow{x : \theta}, \overrightarrow{y : \theta'}\}$.

  2. From rule tr-def we know that $a \in dom(\lambda_x)$ and $l = \mu_\lambda(a)$. Hence $l \in range(\mu_\lambda)$. From the rule we also know that $range(\mu_\lambda) = dom(\lambda_c)$. Hence $l \in dom(\lambda_c)$.

  3. From rule tr-def we know that $r_{y_j} \in \overrightarrow{r_y}$. We also know that $y_j = \mu_\Gamma(r_{y_j})$ and that $\overrightarrow{y} = \mu_\Gamma(\overrightarrow{r_y})$. Hence $y_j \in \overrightarrow{y}$.

  4. From rule tr-def we know that $\forall (a \mapsto l) \in \mu_\lambda : \ \mu_\lambda; \mu_\Gamma; \Gamma; \Sigma \vdash \lambda_x(a) \overset{b}{\rightsquigarrow} \lambda_c(l)$. From Proposition 11 we then know that $\forall l \in range(\mu_\lambda) : \ \Gamma; \Sigma \vdash \lambda_c(l)$. From rule tr-def we know that $range(\mu_\lambda) = dom(\lambda_c)$. Hence $\forall l \in dom(\lambda_c) : \ \Gamma; \Sigma \vdash \lambda_c(l)$.

  Hence by rule t-def we conclude $\Sigma \vdash f(\overrightarrow{x : \theta})\langle \overrightarrow{y : \theta'}, l, \lambda_c, j \rangle$.

$\square$

A.1.3 Semantics Preservation

A.1.3.1 Instructions

Proposition 13 (Preservation of Related Memory for Instructions)**.** If

- $\mu_\Gamma; \Gamma; \Sigma \vdash \iota \overset{\iota}{\rightsquigarrow} \ell := e$
- $\Gamma; \Sigma; \Psi \vdash \rho$
- $\Sigma; \Psi \vdash \sigma; \pi$
- $\mu_a; \nu_a; \pi; \vec{\rho}, \rho \vdash H \leftrightsquigarrow \sigma$
- $\mu_a; \vec{\mu_\Gamma}, \mu_\Gamma; \sigma \vdash \vec{R}, R \leftrightsquigarrow \vec{\rho}, \rho$
- $\vec{R} \vdash \langle H, R, \iota \rangle \xrightarrow{\iota} \langle H', R' \rangle$,
- $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \ell \rangle \xrightarrow{\ell} \langle \sigma', \pi', a \rangle$, and
- $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma', \pi', e \rangle \xrightarrow{e} \langle \sigma'', \pi'', v \rangle$

then for some $\mu'_a \supseteq \mu_a$ and $\nu'_a \supseteq \nu_a$:

- $\mu'_a; \vec{\mu_\Gamma}, \mu_\Gamma; \sigma' \circ \{a \mapsto v\} \vdash \vec{R}, R \leftrightsquigarrow \vec{\rho}, \rho$
- $\mu'_a; \nu'_a; \pi'; \vec{\rho}, \rho \vdash H' \leftrightsquigarrow \sigma' \circ \{a \mapsto v\}$

Proposition 14 (Preservation of Progress for Instructions)**.** If

- $\mu_\Gamma; \Gamma; \Sigma \vdash \iota \overset{\iota}{\rightsquigarrow} \ell := e$
- $\Gamma; \Sigma; \Psi \vdash \rho$
- $\Sigma; \Psi \vdash \sigma; \pi$
- $\mu_a; \nu_a; \pi; \vec{\rho}, \rho \vdash H \leftrightsquigarrow \sigma$
- $\mu_a; \vec{\mu_\Gamma}, \mu_\Gamma; \sigma \vdash \vec{R}, R \leftrightsquigarrow \vec{\rho}, \rho$, and
- $\vec{R} \vdash \langle H, R, \iota \rangle \xrightarrow{\iota} \langle H', R' \rangle$

then

- $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \ell \rangle \xrightarrow{\ell} \mathsf{err}$ or
- $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \ell \rangle \xrightarrow{\ell} \langle \sigma', \pi', a \rangle$ and $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma', \pi', e \rangle \xrightarrow{e} \mathsf{err}$, or
- $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \ell \rangle \xrightarrow{\ell} \langle \sigma', \pi', a \rangle$ and $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma', \pi', e \rangle \xrightarrow{e} \langle \sigma'', \pi'', v \rangle$.

We prove Propositions 13 and 14 together.

Proof. The proof proceeds by case analysis on the derivation of the judgement $\mu_\Gamma; \Gamma; \Sigma \vdash \iota \overset{\iota}{\rightsquigarrow} \ell := e$.

1. Case tr-$\oplus$-r$*_1$. Then $\iota = (\mathsf{op}_4^{\oplus}\ r_i, r_j * c)$, $\ell = x$ and $e = x \oplus (y * m)$.

   (a) This case is not possible. Rule x-$\oplus$-r$*$ always applies.

   (b) In this case rules x-$\oplus$-r$*$ is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{op}_4^{\oplus}\ r_i, r_j * c\rangle \xrightarrow{\iota} \langle H, R'\rangle$. Here $R' = R \circ_4 \{r_i \mapsto \vec{b}_i \oplus_4 (\vec{b}_j *_4 c)\}$ where $\vec{b}_i = R_{0:4}(r_i)$ and $\vec{b}_j = R_{0:4}(r_j)$.

   Similarly, through rule l-var $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x\rangle \xrightarrow{\ell} \langle \sigma, \pi, a\rangle$ with $a = \rho(x)$. Also through rules e-op, e-lval, l-var and e-const we obtain $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, (x \oplus (y * m))\rangle \xrightarrow{e} \langle \sigma, \pi, v\rangle$ where $v = v_x \oplus_\pi (v_y *_\pi m)$, $v_x = \sigma(a)$, $a' = \rho(y)$ and $v_y = \sigma(a')$.

   From rule tr-$\oplus$-r$*_1$ we know $(r_i : x)_4 \in \mu_\Gamma$. Hence from the related registers we know $\mu_a \vdash \vec{b}_i \leftrightsquigarrow v_x$. Similarly, we know $\mu_a \vdash \vec{b}_j \leftrightsquigarrow v_y$. Then from $(x : \theta[]*) \in \Gamma$ and the store typing of σ it follows that $v_x = n_*$ and from the success of the addition, it also follows that $[n_*, n_* \oplus (v_y * m)] \subseteq\in \pi$. Hence, also from the store typing all m values at the addresses in this range have type θ. From the related heaps it then follows with $c/m = \mathit{sizeof}(\theta)$ that $\mu_a \vdash (\vec{b}_i \oplus_4 (\vec{b}_j *_4 c)) \leftrightsquigarrow (v \oplus_\pi (v_y * m))$. Hence, the update registers are still related.

2. Case tr-$\oplus$-r$*_2$. Then $\iota = (\mathsf{op}_w^{\oplus}\ r_i, r_j * c)$, $\ell = x$ and $e = x \oplus (y * c)$.

   (a) This case is not possible. Rule x-$\oplus$-r$*$ always applies.

   (b) In this case rules x-$\oplus$-r$*$ is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{op}_w^{\oplus}\ r_i, r_j * c\rangle \xrightarrow{\iota} \langle H, R'\rangle$. Here $R' = R \circ_w \{r_i \mapsto \vec{b}_i \oplus_w (\vec{b}_j *_w c)\}$ where $\vec{b}_i = R_{0:w}(r_i)$ and $\vec{b}_j = R_{0:w}(r_j)$.

   Similarly, through rule l-var $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x\rangle \xrightarrow{\ell} \langle \sigma, \pi, a\rangle$ with $a = \rho(x)$. Also through rules e-op, e-lval, l-var and e-const we obtain $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, (x \oplus (y * m))\rangle \xrightarrow{e} \langle \sigma, \pi, v\rangle$ where $v = v_x \oplus_\pi (v_y *_\pi m)$, $v_x = \sigma(a)$, $a' = \rho(y)$ and $v_y = \sigma(a')$.

From rule tr-$\oplus$-r$*_2$ we know $(r_i : x)_w \in \mu_\Gamma$. Hence from the related registers we know $\mu_a \vdash \vec{b}_i \leftrightsquigarrow v_x$. Similarly, we know $\mu_a \vdash \vec{b}_j \leftrightsquigarrow v_y$. It then follows that $\mu_a \vdash (\vec{b}_i \oplus_w (\vec{b}_j *_w c)) \leftrightsquigarrow (v \oplus_\pi (v_y * c))$. Hence, the update registers are still related.

3. Case tr-$\otimes$-rc. Then $\iota = (\mathsf{op}^{\otimes}_w\ r_i, c)$, $\ell = x$ and $e = x \otimes c$.

   (a) This case is not possible. Rule x-$\otimes$-rc always applies.

   (b) In this case rules x-$\otimes$-rc is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{op}^{\otimes}_w\ r_i, c\rangle \xrightarrow{\iota} \langle H, R'\rangle$. Here $R' = R \circ_w \{r_i \mapsto \vec{b} \otimes_w c\}$ where $\vec{b} = R_{0:w}(r_i)$.

   Similarly, through rule l-var $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x\rangle \xrightarrow{\ell} \langle \sigma, \pi, a\rangle$ with $a = \rho(x)$. Also through rules e-op, e-lval, l-var and e-const we obtain $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, (x \otimes c)\rangle \xrightarrow{e} \langle \sigma, \pi, v'\rangle$ where $v' = v \otimes_\pi c$ and $v = \sigma(a)$.

   From rule tr-$\otimes$-rc we know $(r_i : x)_w \in \mu_\Gamma$. Hence from the related registers we know $\mu_a \vdash \vec{b} \leftrightsquigarrow v$. Then from $(x : t) \in \Gamma$ and $w = \mathit{sizeof}(t)$ it follows that $\mu_a \vdash (\vec{b} \otimes_w c) \leftrightsquigarrow (v \otimes_\pi c)$. Hence, the update registers are still related.

4. Case tr-$\oplus$-rc. Then $\iota = (\mathsf{op}^{\oplus}_4\ r_i, c)$, $\ell = x$ and $e = x \oplus m$.

   (a) This case is not possible. Rule x-$\otimes$-rc always applies.

   (b) In this case rules x-$\otimes$-rc is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{op}^{\oplus}_4\ r_i, c\rangle \xrightarrow{\iota} \langle H, R'\rangle$. Here $R' = R \circ_4 \{r_i \mapsto \vec{b} \oplus_4 c\}$ where $\vec{b} = R_{0:4}(r_i)$.

   Similarly, through rule l-var $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x\rangle \xrightarrow{\ell} \langle \sigma, \pi, a\rangle$ with $a = \rho(x)$. Also through rules e-op, e-lval, l-var and e-const we obtain $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, (x \oplus m)\rangle \xrightarrow{e} \langle \sigma, \pi, v'\rangle$ where $v' = v \oplus_\pi m$ and $v = \sigma(a)$.

   From rule tr-$\oplus$-rc we know $(r_i : x)_4 \in \mu_\Gamma$. Hence from the related registers we know $\mu_a \vdash \vec{b} \leftrightsquigarrow v$. Then from $(x : \theta[]*) \in \Gamma$ and the store typing of σ it follows that $v = n_*$ and from the success of the addition, it also follows that $[n_*, n_* \oplus m] \subseteq\in \pi$. Hence, also from the store typing all m values at the addresses in this range have type θ. From the related heaps it then follows with $c/m = \mathit{sizeof}(\theta)$ that $\mu_a \vdash (\vec{b} \oplus_4 c) \leftrightsquigarrow (v \oplus_\pi m)$. Hence, the update registers are still related.

5. Case tr-$\otimes$-rr. Then $\iota = (\mathsf{op}_w^{\otimes}\ r_i, r_j)$, $\ell = x$ and $e = x \otimes y$.

   (a) This case is not possible. Rule x-$\otimes$-rr always applies.

   (b) In this case rules x-$\otimes$-rc is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{op}_w^{\otimes}\ r_i, r_j\rangle \xrightarrow{\iota} \langle H, R'\rangle$. Here $R' = R \circ_w \{r_i \mapsto \vec{b}_i \oplus_w \vec{b}_j\}$ where $\vec{b}_i = R_{0:w}(r_i)$ and $\vec{b}_j = R_{0:w}(r_j)$.

   Similarly, through rule l-var $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x\rangle \xrightarrow{\ell} \langle \sigma, \pi, a\rangle$ with $a = \rho(x)$. Also through rules e-op, e-lval and l-var we obtain $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, (x \otimes y)\rangle \xrightarrow{e} \langle \sigma, \pi, v\rangle$ where $v = v_x \otimes_\pi v_y$, $v_x = \sigma(a)$, $a' = \rho(y)$ and $v_y = \sigma(a')$.

   From rule tr-$\otimes$-rr we know $(r_i : x)_w \in \mu_\Gamma$. Hence from the related registers we know $\mu_a \vdash \vec{b}_i \leftrightsquigarrow v_x$. By similar reasoning we know $\mu_a \vdash \vec{b}_j \leftrightsquigarrow v_y$. Then from $(x : t) \in \Gamma$, $(y : t) \in \Gamma$ and $w = \mathit{sizeof}(t)$ it follows that $\mu_a \vdash (\vec{b}_i \otimes_w \vec{b}_j) \leftrightsquigarrow (v_x \otimes_\pi v_y)$. Hence, the update registers are still related.

6. Case tr-mov-rc. Then $\iota = (\mathsf{mov}_w\ r_i, c)$, $\ell = x$ and $e = c$.

   (a) This case is not possible. Rule x-mov-rc always applies.

   (b) In this case rules x-mov-rc is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{mov}_w\ r_i, c\rangle \xrightarrow{\iota} \langle H, R'\rangle$. Here $R' = R \circ_w \{r_i \mapsto c\}$.

   Similarly, through rule l-var $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x\rangle \xrightarrow{\ell} \langle \sigma, \pi, a\rangle$ with $a = \rho(x)$. Also through rule e-const we obtain $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, c\rangle \xrightarrow{e} \langle \sigma, \pi, c\rangle$.

   We know that $\mu_a \vdash c \leftrightsquigarrow c$. Hence, the update registers are still related.

7. Case tr-mov-r0. Then $\iota = (\mathsf{mov}_4\ r_i, 0)$, $\ell = x$ and $e = 0$.

   (a) This case is not possible. Rule x-mov-rc always applies.

   (b) In this case rules x-mov-rc is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{mov}_4\ r_i, 0\rangle \xrightarrow{\iota} \langle H, R'\rangle$. Here $R' = R \circ_4 \{r_i \mapsto 0\}$.

   Similarly, through rule l-var $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x\rangle \xrightarrow{\ell} \langle \sigma, \pi, a\rangle$ with $a = \rho(x)$. Also through rule e-const we obtain $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, 0\rangle \xrightarrow{e} \langle \sigma, \pi, 0\rangle$.

We know that $\mu_a \vdash 0 \leftrightsquigarrow 0$. Hence, the update registers are still related.

8. Case tr-mov-rr. Then $\iota = (\mathsf{mov}_w\ r_i, r_j)$, $\ell = x$ and $e = y$.

   (a) This case is not possible. Rule x-mov-rr always applies.

   (b) In this case rules x-mov-rr is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{mov}_w\ r_i, r_j\rangle \xrightarrow{\iota} \langle H, R'\rangle$. Here $R' = R \circ_w \{r_i \mapsto \vec{b}\}$ where $\vec{b} = R_{0:w}(r_j)$.

   Similarly, through rule l-var $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x\rangle \xrightarrow{\ell} \langle \sigma, \pi, a\rangle$ with $a = \rho(x)$. Also through rules e-lval and l-var we obtain $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, y\rangle \xrightarrow{e} \langle \sigma, \pi, v\rangle$ where $v = \sigma(a')$ and $a' = \rho(y)$.

   From rule tr-mov-rr we know $(r_j : y)_w \in \mu_\Gamma$. Hence from the related registers we know $\mu_a \vdash \vec{b} \leftrightsquigarrow v$. Also from rule tr-mov-rr we know $(r_i : x)_w \in \mu_\Gamma$. Hence, the registers are related. After the update we can see that they are still related.

9. Case tr-mov-ri$_1$. Then $\iota = (\mathsf{mov}_w\ r_i, [r_j])$, $\ell = x$ and $e = *y$.

   (a) This case is possible iff $R(r_j) = 0$ or $R(r_j) = \bot$. Because of the related registers and, from rule tr-mov-ri$_1$, $(r_j : y)_4 \in \mu_\Gamma$, we have $\mu_a \vdash R(r_j) \leftrightsquigarrow \sigma(\rho(y))$. In either of the cases for $R(r_j)$ we also have $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, y\rangle \xrightarrow{e} \mathsf{err}$.

   (b) In this case rules x-mov-ri is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{mov}_w\ r_i, [r_j]\rangle \xrightarrow{\iota} \langle H, R'\rangle$. Here $R' = R \circ_w \{r_i \mapsto \vec{b}_2\}$ where $\vec{b}_2 = H^w(\vec{b}_1)$ and $\vec{b}_1 = R_(r_j)$.

   Similarly, through rule l-var $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x\rangle \xrightarrow{\ell} \langle \sigma, \pi, a\rangle$ with $a = \rho(x)$. Also through rules e-lval , l-ptr and l-var we obtain $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, *y\rangle \xrightarrow{e} \langle \sigma, \pi, v_2\rangle$ where $v_2 = \sigma(v_1)$, $v_1 = \sigma(a')$ and $a' = \rho(y)$.

   From rule tr-mov-ri$_1$ we know $(r_j : y)_4 \in \mu_\Gamma$. Hence from the related registers we know $\mu_a \vdash \vec{b}_1 \leftrightsquigarrow v_1$. From related stores, we also know $\mu_a \vdash \vec{b}_2 \leftrightsquigarrow v_2$. Also from rule tr-mov-ri$_1$ we know $(r_i : x)_w \in \mu_\Gamma$. Hence, the registers are related. After the update we can see that they are still related.

10. Case tr-mov-ri$_2$. Then $\iota = (\mathsf{mov}_w\ r_i, [r_j])$, $\ell = x$ and $e = y[0]$.

(a) This case is possible iff $R(r_j) = 0$ or $R(r_j) = \bot$. Because of the related registers and, from rule tr-mov-ri$_2$, $(r_j : y)_4 \in \mu_\Gamma$, we have $\mu_a \vdash R(r_j) \leftrightsquigarrow \sigma(\rho(y))$. In either of the cases for $R(r_j)$ we also have $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, y \rangle \xrightarrow{e} \mathsf{err}$.

(b) In this case rules x-mov-ri is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{mov}_w\ r_i, [r_j] \rangle \xrightarrow{\iota} \langle H, R' \rangle$. Here $R' = R \circ_w \{r_i \mapsto \vec{b}_2\}$ where $\vec{b}_2 = H^w(\vec{b}_1)$ and $\vec{b}_1 = R_(r_j)$. Similarly, through rule l-var $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x \rangle \xrightarrow{\ell} \langle \sigma, \pi, a \rangle$ with $a = \rho(x)$. Also through rules e-lval , l-ar and e-const we obtain $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, y[0] \rangle \xrightarrow{e} \langle \sigma, \pi, v_2 \rangle$ where $v_2 = \sigma(v_1)$, $v_1 = \sigma(a')$ and $a' = \rho(y)$.

From rule tr-mov-ri$_2$ we know $(r_j : y)_4 \in \mu_\Gamma$. Hence from the related registers we know $\mu_a \vdash \vec{b}_1 \leftrightsquigarrow v_1$. From related stores, we also know $\mu_a \vdash \vec{b}_2 \leftrightsquigarrow v_2$. Also from rule tr-mov-ri$_2$ we know $(r_i : x)_w \in \mu_\Gamma$. Hence, the registers are related. After the update we can see that they are still related.

11. Case tr-mov-ri$_3$. Then $\iota = (\mathsf{mov}_w\ r_i, [r_j])$, $\ell = x$ and $e = y \to 0$.

(a) This case is possible iff $R(r_j) = 0$ or $R(r_j) = \bot$. Because of the related registers and, from rule tr-mov-ri$_3$, $(r_j : y)_4 \in \mu_\Gamma$, we have $\mu_a \vdash R(r_j) \leftrightsquigarrow \sigma(\rho(y))$. In either of the cases for $R(r_j)$ we also have $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, y \rangle \xrightarrow{e} \mathsf{err}$.

(b) In this case rules x-mov-ri is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{mov}_w\ r_i, [r_j] \rangle \xrightarrow{\iota} \langle H, R' \rangle$. Here $R' = R \circ_w \{r_i \mapsto \vec{b}_2\}$ where $\vec{b}_2 = H^w(\vec{b}_1)$ and $\vec{b}_1 = R_(r_j)$. Similarly, through rule l-var $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x \rangle \xrightarrow{\ell} \langle \sigma, \pi, a \rangle$ with $a = \rho(x)$. Also through rules e-lval and l-fldwe obtain $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, y \to 0 \rangle \xrightarrow{e} \langle \sigma, \pi, v_2 \rangle$ where $v_2 = \sigma(v_1)$, $v_1 = \sigma(a')$ and $a' = \rho(y)$.

From rule tr-mov-ri$_3$ we know $(r_j : y)_4 \in \mu_\Gamma$. Hence from the related registers we know $\mu_a \vdash \vec{b}_1 \leftrightsquigarrow v_1$. From related stores, we also know $\mu_a \vdash \vec{b}_2 \leftrightsquigarrow v_2$. Also from rule tr-mov-ri$_3$ we know $(r_i : x)_w \in \mu_\Gamma$. Hence, the registers are related. After the update we can see that they are still related.

12. Case tr-mov-ir$_1$. Then $\iota = (\mathsf{mov}_w\ [r_i], r_j)$, $\ell = *x$ and $e = y$.

(a) This case is possible iff $R(r_i) = 0$ or $R(r_i) = \bot$. Because of the related registers and, from rule tr-mov-ir$_1$, $(r_i : x)_4 \in \mu_\Gamma$, we have $\mu_a \vdash R(r_i) \leftrightsquigarrow \sigma(\rho(x))$. In either of the cases for $R(r_i)$ we also have $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x\rangle \xrightarrow{\ell} \mathsf{err}$.

(b) In this case rules x-mov-ir is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{mov}_w\ [r_i], r_j\rangle \xrightarrow{\iota} \langle H', R\rangle$. Here $H' = H \circ \{\vec{b}_1, \dots, \vec{b}_1 + (w-1) \mapsto \vec{b}_2\}$ where $\vec{b}_1 = R(r_i)$ and $\vec{b} = R_{0:w}(r_j)$.

Similarly, through rule l-ptr $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, *x\rangle \xrightarrow{\ell} \langle \sigma, \pi, v_1\rangle$ with $v_1 = \sigma(a)$ and $a = \rho(x)$. Also through rules e-lval and l-varwe obtain $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, y\rangle \xrightarrow{e} \langle \sigma, \pi, v_2\rangle$ where $v_2 = \sigma(a')$ and $a' = \rho(y)$.

From rule tr-mov-ir$_1$ we know $(r_j : y)_w \in \mu_\Gamma$. Hence from the related registers we know $\mu_a \vdash \vec{b}_2 \leftrightsquigarrow v_2$. From related stores, we also know $\mu_a \vdash \vec{b}_2 \leftrightsquigarrow v_2$. Also from rule tr-mov-ir$_1$ we know $(r_i : x)_w \in \mu_\Gamma$. Hence, $\mu_a \vdash \vec{b}_1 \leftrightsquigarrow v_1$. Since $(x : \theta_1 *) \in \Gamma$, we know that v_1 is an address. Because of related heaps, we then know that $(\vec{b}_1, v_1) in \mu_a$. After the update we can see that they are still related.

13. Case tr-mov-ir$_2$. Then $\iota = (\mathsf{mov}_w\ [r_i], r_j)$, $\ell = x[0]$ and $e = y$.

(a) This case is possible iff $R(r_i) = 0$ or $R(r_i) = \bot$. Because of the related registers and, from rule tr-mov-ir$_2$, $(r_i : x)_4 \in \mu_\Gamma$, we have $\mu_a \vdash R(r_i) \leftrightsquigarrow \sigma(\rho(x))$. In either of the cases for $R(r_i)$ we also have $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x\rangle \xrightarrow{\ell} \mathsf{err}$.

(b) In this case rules x-mov-ir is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{mov}_w\ [r_i], r_j\rangle \xrightarrow{\iota} \langle H', R\rangle$. Here $H' = H \circ \{\vec{b}_1, \dots, \vec{b}_1 + (w-1) \mapsto \vec{b}_2\}$ where $\vec{b}_1 = R(r_i)$ and $\vec{b} = R_{0:w}(r_j)$.

Similarly, through rule l-ar and e-const $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x[0]\rangle \xrightarrow{\ell} \langle \sigma, \pi, v_1\rangle$ with $v_1 = \sigma(a)$ and $a = \rho(x)$. Also through rules e-lval and l-varwe obtain $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, y\rangle \xrightarrow{e} \langle \sigma, \pi, v_2\rangle$ where $v_2 = \sigma(a')$ and $a' = \rho(y)$.

From rule tr-mov-ir$_2$ we know $(r_j : y)_w \in \mu_\Gamma$. Hence from the related registers we know $\mu_a \vdash \vec{b}_2 \leftrightsquigarrow v_2$. From related stores, we also know $\mu_a \vdash \vec{b}_2 \leftrightsquigarrow v_2$. Also from rule tr-mov-ir$_2$ we know $(r_i : x)_w \in \mu_\Gamma$. Hence, $\mu_a \vdash \vec{b}_1 \leftrightsquigarrow v_1$. Since $(x : \theta_1[]*) \in \Gamma$, we know that v_1 is an address. Because of related heaps, we then know that $(\vec{b}_1, v_1) in \mu_a$. After the update we can see that they are still related.

14. Case tr-mov-ir$_3$. Then $\iota = (\mathsf{mov}_w\ [r_i], r_j)$, $\ell = x \to 0$ and $e = y$.

    (a) This case is possible iff $R(r_i) = 0$ or $R(r_i) = \bot$. Because of the related registers and, from rule tr-mov-ir$_3$, $(r_i : x)_4 \in \mu_\Gamma$, we have $\mu_a \vdash R(r_i) \leftrightsquigarrow \sigma(\rho(x))$. In either of the cases for $R(r_i)$ we also have $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x \rangle \xrightarrow{\ell} \mathsf{err}$.

    (b) In this case rules x-mov-ir is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{mov}_w\ [r_i], r_j \rangle \xrightarrow{\iota} \langle H', R \rangle$. Here $H' = H \circ \{\vec{b}_1, \ldots, \vec{b}_1 + (w-1) \mapsto \vec{b}_2\}$ where $\vec{b}_1 = R(r_i)$ and $\vec{b} = R_{0:w}(r_j)$.

    Similarly, through rule l-fld $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x \to 0 \rangle \xrightarrow{\ell} \langle \sigma, \pi, v_1 \rangle$ with $v_1 = \sigma(a)$ and $a = \rho(x)$. Also through rules e-lval and l-varwe obtain $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, y \rangle \xrightarrow{e} \langle \sigma, \pi, v_2 \rangle$ where $v_2 = \sigma(a')$ and $a' = \rho(y)$.

    From rule tr-mov-ir$_3$ we know $(r_j : y)_w \in \mu_\Gamma$. Hence from the related registers we know $\mu_a \vdash \vec{b}_2 \leftrightsquigarrow v_2$. From related stores, we also know $\mu_a \vdash \vec{b}_2 \leftrightsquigarrow v_2$. Also from rule tr-mov-ir$_3$ we know $(r_i : x)_w \in \mu_\Gamma$. Hence, $\mu_a \vdash \vec{b}_1 \leftrightsquigarrow v_1$. Since $(x : N*) \in \Gamma$, we know that v_1 is an address. Because of related heaps, we then know that $(\vec{b}_1, v_1) in \mu_a$. After the update we can see that they are still related.

15. Case tr-mov-ri+$_1$. Then $\iota = (\mathsf{mov}_w\ r_i, [r_j + c]$, $\ell = x$ and $e = y[m]$.

    (a) This case is possible iff $R(r_j) = 0$, $R(r_j) = \bot$ or $(R(r_j) + c) \notin dom(H)$. Because of the related registers and heaps, and from rule tr-mov-ri+$_1$$(r_j : y)_4 \in \mu_\Gamma$, we have $\mu_a \vdash R(r_j) \leftrightsquigarrow \sigma(\rho(y))$. In either of the first two cases for $R(r_j)$ we also have $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, y[m] \rangle \xrightarrow{\ell} \mathsf{err}$. In the last case, because of related heaps, it also has to be that $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, y[m] \rangle \xrightarrow{\ell} \mathsf{err}$.

    (b) In this case rules x-mov-r+ is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{mov}_w\ r_i, [r_j + c] \rangle \xrightarrow{\iota} \langle H, R' \rangle$. Here $R' = R \circ_w \{r_i \mapsto \vec{b}\}$ where $\vec{b} = H^w(\vec{b}')$ and $\vec{b} = R(r_j) +_4 c$.

Similarly, through rule l-var $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x \rangle \xrightarrow{\ell} \langle \sigma, \pi, a \rangle$ with $a = \rho(x)$. Also through rules e-lval , l-arand e-const we obtain $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, y[m] \rangle \xrightarrow{e} \langle \sigma, \pi, v \rangle$ where $v = \sigma(a'' + m)$, $a'' = \sigma(a')$ and $a' = \rho(y)$.

From rule tr-mov-ri+$_1$ we know $(r_j : y)_4 \in \mu_\Gamma$. Hence from the related registers we know $\mu_a \vdash \vec{b'} \leftrightsquigarrow a''$. From the translation rule we also have $(y : \theta[]*) \in \Gamma$. Because of the progress, it means that $[a'', a'' + m] \subseteq\in \pi$. Because of the related heaps and well-typed store it follows that $\mu_a \vdash \vec{b} \leftrightsquigarrow v$. Also from rule tr-mov-ri+$_1$ we know $(r_i : x)_w \in \mu_\Gamma$. After the update we can see that they are still related.

16. Case tr-mov-ri+$_2$. Then $\iota = (\mathsf{mov}_w\ r_i, [r_j + c]$, $\ell = x$ and $e = y \to m$.

    (a) This case is possible iff $R(r_j) = 0$, $R(r_j) = \bot$ or $(R(r_j) + c) \notin dom(H)$. Because of the related registers and heaps, and from rule tr-mov-ri+$_2$$(r_j : y)_4 \in \mu_\Gamma$, we have $\mu_a \vdash R(r_j) \leftrightsquigarrow \sigma(\rho(y))$. In either of the first two cases for $R(r_j)$ we also have $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, y[m] \rangle \xrightarrow{\ell} \mathsf{err}$. In the last case, because of related heaps, it also has to be that $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, y \to m \rangle \xrightarrow{\ell} \mathsf{err}$.

    (b) In this case rules x-mov-r+ is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{mov}_w\ r_i, [r_j + c] \rangle \xrightarrow{\iota} \langle H, R' \rangle$. Here $R' = R \circ_w \{r_i \mapsto \vec{b}\}$ where $\vec{b} = H^w(\vec{b'})$ and $\vec{b} = R(r_j) +_4 c$.

    Similarly, through rule l-var $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x \rangle \xrightarrow{\ell} \langle \sigma, \pi, a \rangle$ with $a = \rho(x)$. Also through rules e-lval and l-fld we obtain $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, y \to m \rangle \xrightarrow{e} \langle \sigma, \pi, v \rangle$ where $v = \sigma(a'' + m)$, $a'' = \sigma(a')$ and $a' = \rho(y)$.

    From rule tr-mov-ri+$_2$ we know $(r_j : y)_4 \in \mu_\Gamma$. Hence from the related registers we know $\mu_a \vdash \vec{b'} \leftrightsquigarrow a''$. From the translation rule we also have $(y : N*) \in \Gamma$ and $\Sigma(N) = \langle \theta_0, \dots, \theta_n \rangle$. Because of the progress, it means that $[a'', a'' + m] \subseteq\in \pi$. Because of the related heaps and well-typed store it follows that $\mu_a \vdash \vec{b} \leftrightsquigarrow v$. Also from rule tr-mov-ri+$_1$ we know $(r_i : x)_w \in \mu_\Gamma$. After the update we can see that they are still related.

17. Case tr-mov-i+r$_1$. Then $\iota = (\mathsf{mov}_w\ [r_i + c], r_j$, $\ell = x[m]$ and $e = y$.

(a) This case is possible iff $R(r_i) = 0$, $R(r_i) = \bot$ or $(R(r_i) + c) \notin dom(H)$. Because of the related registers and heaps, and from rule tr-mov-i+r$_1$ $(r_i : x)_4 \in \mu_\Gamma$, we have $\mu_a \vdash R(r_i) \leftrightsquigarrow \sigma(\rho(x))$. In either of the first two cases for $R(r_i)$ we also have $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x[m] \rangle \xrightarrow{\ell} \mathsf{err}$. In the last case, because of related heaps, it also has to be that $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x[m] \rangle \xrightarrow{\ell} \mathsf{err}$.

(b) In this case rules x-mov-+r is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{mov}_w\ [r_i + c], r_j \rangle \xrightarrow{\iota} \langle H', R \rangle$. Here $H' = H \circ \{H(R(r_i)) +_4 c +_4 n \mapsto R_{n:n+1}(r_j)\}_{n=0}^{w-1}$. Similarly, through rule l-ar $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x[m] \rangle \xrightarrow{\ell} \langle \sigma, \pi, a \rangle$ with $a = a' + m$ and $a' = \rho(x)$. Also through rules e-lval and l-var we obtain $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, y \rangle \xrightarrow{e} \langle \sigma, \pi, v \rangle$ where $v = \sigma(a'')$ and $a'' = \rho(y)$.

From rule tr-mov-i+r$_1$ we know $(r_i : x)_4 \in \mu_\Gamma$. Hence from the related registers we know $\mu_a \vdash R(r_i) \leftrightsquigarrow a'$. From the translation rule we also have $(x : \theta[]*) \in \Gamma$. Because of the progress, it means that $[a', a' + m] \subseteq\in \pi$. Because of the related heaps and well-typed store it follows that $(R(r_i) + c, a' + m) \in \mu_a$. Also from rule tr-mov-ri+$_1$ we know $(r_j : y)_w \in \mu_\Gamma$. Hence, $\mu_a \vdash R_{0:w}(r_j) \leftrightsquigarrow v$. After the update we can see that $(R(r_i) + c)$ and $a' + m$ are still related.

18. Case tr-mov-i+r$_2$. Then $\iota = (\mathsf{mov}_w\ [r_i + c], r_j$, $\ell = x \to m$ and $e = y$.

(a) This case is possible iff $R(r_i) = 0$, $R(r_i) = \bot$ or $(R(r_i) + c) \notin dom(H)$. Because of the related registers and heaps, and from rule tr-mov-i+r$_2$ $(r_i : x)_4 \in \mu_\Gamma$, we have $\mu_a \vdash R(r_i) \leftrightsquigarrow \sigma(\rho(x))$. In either of the first two cases for $R(r_i)$ we also have $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x \to m \rangle \xrightarrow{\ell} \mathsf{err}$. In the last case, because of related heaps, it also has to be that $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x \to m \rangle \xrightarrow{\ell} \mathsf{err}$.

(b) In this case rules x-mov-+r is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{mov}_w\ [r_i + c], r_j \rangle \xrightarrow{\iota} \langle H', R \rangle$. Here $H' = H \circ \{H(R(r_i)) +_4 c +_4 n \mapsto R_{n:n+1}(r_j)\}_{n=0}^{w-1}$. Similarly, through rule l-ar $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x \to m \rangle \xrightarrow{\ell} \langle \sigma, \pi, a \rangle$ with $a = a' + m$ and $a' = \rho(x)$. Also through rules e-lval and l-var we obtain $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, y \rangle \xrightarrow{e} \langle \sigma, \pi, v \rangle$ where $v = \sigma(a'')$ and $a'' = \rho(y)$.

From rule tr-mov-i+r$_2$ we know $(r_i : x)_4 \in \mu_\Gamma$. Hence from the related registers we know $\mu_a \vdash R(r_i) \leftrightsquigarrow a'$. From the translation rule we also have $(x : N*) \in \Gamma$. Because of the progress, it means that $[a', a' + m] \subseteq\in \pi$. Because of the related heaps and well-typed store it follows that $(R(r_i) + c, a' + m) \in \mu_a$. Also from rule tr-mov-ri$+_1$ we know $(r_j : y)_w \in \mu_\Gamma$. Hence, $\mu_a \vdash R_{0:w}(r_j) \leftrightsquigarrow v$. After the update we can see that $(R(r_i) + c)$ and $a' + m$ are still related.

19. Case tr-alloc-r$*$. Then $\iota = (\mathsf{alloc}\ r_i, r_j * c$, $\ell = x$ and $e = \mathsf{new}\ \theta[y * m]$.

    (a) Rule x-alloc-* only fails iff $R(r_j) = \bot$. Similarly, while rules l-var, e-const and e-op do not fail, rule e-ar fails iff $\sigma(\rho(y)) = \bot$. Since $(r_j : y) \in \mu_\Gamma$, both failures coincide.

    (b) This case is similar to that of tr-alloc-rc$_2$.

20. Case tr-alloc-rc$_1$. Then $\iota = (\mathsf{alloc}\ r_i, c$, $\ell = x$ and $e = \mathsf{new}\ \theta$.

    (a) Rule x-alloc cannot fail. Similarly, rules l-var and e-new do not fail.

    (b) In this case rules x-alloc is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{alloc}\ r_i, c\rangle \xrightarrow{\iota} \langle H', R'\rangle$. Here $R' = R \circ_4 r_i \mapsto a$. Also $H' = H \circ \{a + i \mapsto \bot\}_{i=0}^{c-1}$. Similarly, through rule l-var $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x\rangle \xrightarrow{\ell} \langle \sigma, \pi, a'\rangle$ where $a' = \rho(x)$. Also through rule e-new we obtain $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \mathsf{new}\ \theta\rangle \xrightarrow{e} \langle \sigma', \pi, a''\rangle$ where $\sigma' = \sigma \circ \{a'' \mapsto \bot\}$.

       Then choose $\mu'_a = \mu_a \circ \{(a : a'')_c\}$. Since $\mu_a \vdash \bot \leftrightsquigarrow \bot$ these fresh addresses are related. Also pick $\nu'_a = \nu_a \circ \{a + i \mapsto (a, c)\}_{i=0}^{c-1}$.

21. Case tr-alloc-rc$_2$. Then $\iota = (\mathsf{alloc}\ r_i, c$, $\ell = x$ and $e = \mathsf{new\ struct}\ N$.

    (a) Rule x-alloc cannot fail. Similarly, rules l-var and e-str do not fail.

    (b) In this case rules x-alloc is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{alloc}\ r_i, c\rangle \xrightarrow{\iota} \langle H', R'\rangle$. Here $R' = R \circ_4 r_i \mapsto a$. Also $H' = H \circ \{a + i \mapsto \bot\}_{i=0}^{c-1}$. Similarly, through rule l-var $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x\rangle \xrightarrow{\ell} \langle \sigma, \pi, a'\rangle$ where $a' = \rho(x)$. Also through rule e-str we obtain $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \mathsf{new\ struct}\ \theta\rangle \xrightarrow{e} \langle \sigma', \pi, a''\rangle$ where $\sigma' = \sigma \circ \{a'' + i \mapsto \bot\}_{i=0}^{n-1}$ with n is the number of fields in the struct.

       The new memory relations are straightforward.

22. Case tr-alloc-rc$_3$. Then $\iota = (\mathsf{alloc}\ r_i, c$, $\ell = x$ and $e = \mathsf{new}\ \theta[m]$.

    (a) Rule x-alloc cannot fail. Similarly, rules l-var,e-str and e-const do not fail.

    (b) In this case rules x-alloc is used for progress on ι: $\vec{R} \vdash \langle H, R, \mathsf{alloc}\ r_i, c\rangle \xrightarrow{\iota} \langle H', R'\rangle$. Here $R' = R \circ_4 r_i \mapsto a$. Also $H' = H \circ \{a + i \mapsto \bot\}_{i=0}^{c-1}$.
    Similarly, through rule l-var $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, x\rangle \xrightarrow{\ell} \langle \sigma, \pi, a'\rangle$ where $a' = \rho(x)$. Also through rule e-ar we obtain $\Sigma; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \mathsf{new}\ \theta[m]\rangle \xrightarrow{e} \langle \sigma', \pi, a''\rangle$ where $\sigma' = \sigma \circ \{a'' + i \mapsto \bot\}_{i=0}^{m-1}$.
    The new memory relations are straightforward.

23. Case tr-call. This case follows coinductively.

□

A.1.3.2 Basic Blocks

The two Propositions for basic blocks are the following.

Proposition 15 (Preservation of Progress for Basic Blocks)**.** If

- $\mu_\lambda; \mu_\Gamma; \Gamma; \Sigma \vdash b \overset{b}{\rightsquigarrow} s$
- $\forall (a : l) \in \mu_\lambda : \mu_\lambda; \mu_\Gamma; \Gamma; \Sigma \vdash \lambda_x(a) \overset{b}{\rightsquigarrow} \lambda_c(l)$
- $\Gamma; \Sigma; \Psi \vdash \rho$
- $\Sigma; \Psi \vdash \sigma; \pi$
- $\mu_a; \nu_a; \pi; \vec{\rho}, \rho \vdash H \leftrightsquigarrow \sigma$
- $\mu_a; \vec{\mu_\Gamma}, \mu_\Gamma; \sigma \vdash \vec{R}, R \leftrightsquigarrow \vec{\rho}, \rho$
- $\lambda_x; \vec{R} \vdash \langle H, R, b\rangle \xrightarrow{b} \langle H', R', b'\rangle$

then

- $\Sigma; \lambda_c; \vec{\rho}; \rho \vdash \langle \sigma, \pi, s\rangle \xrightarrow{s} \mathsf{err}$ or
- $\Sigma; \lambda_c; \vec{\rho}; \rho \vdash \langle \sigma, \pi, s\rangle \xrightarrow{s} \langle \sigma', \pi', s'\rangle$.

Proposition 16 (Preservation of Related Memory for Basic Blocks)**.** If

- $\mu_\lambda; \mu_\Gamma; \Gamma; \Sigma \vdash b \overset{b}{\rightsquigarrow} s$
- $\forall (a : l) \in \mu_\lambda : \mu_\lambda; \mu_\Gamma; \Gamma; \Sigma \vdash \lambda_x(a) \overset{b}{\rightsquigarrow} \lambda_c(l)$
- $\Gamma; \Sigma; \Psi \vdash \rho$
- $\Sigma; \Psi \vdash \sigma; \pi$
- $\mu_a; \nu_a; \pi; \vec{\rho}, \rho \vdash H \leftrightsquigarrow \sigma$
- $\mu_a; \vec{\mu_\Gamma}, \mu_\Gamma; \sigma \vdash \vec{R}, R \leftrightsquigarrow \vec{\rho}, \rho$
- $\lambda_x; \vec{R} \vdash \langle H, R, b \rangle \xrightarrow{b} \langle H', R', b' \rangle$
- $\Sigma; \lambda_c; \vec{\rho}; \rho \vdash \langle \sigma, \pi, s \rangle \xrightarrow{s} \langle \sigma', \pi', s' \rangle$

then for some $\mu'_a \supseteq \mu_a$ and $\nu'_a \supseteq \nu_a$:

- $\mu'_a; \vec{\mu_\Gamma}, \mu_\Gamma; \sigma' \vdash \vec{R}, R \leftrightsquigarrow \vec{\rho}, \rho$
- $\mu'_a; \nu'_a; \pi'; \vec{\rho}, \rho \vdash H' \leftrightsquigarrow \sigma'$

Proof. The proof is straightforward. □

A.1.3.3 Function Definitions

The two Propositions for function definitions are the following.

Proposition 17 (Preservation of Progress for Function Definitions)**.** If

- $\Sigma \vdash \langle f, \overrightarrow{r_x}, \overrightarrow{r_y}, a, \lambda_x, j \rangle \rightsquigarrow f(\overrightarrow{x : \theta})\langle \overrightarrow{y : \theta'}, l, \lambda_c, j \rangle$
- $\mu_\Gamma = \{\overrightarrow{r_x \mapsto x}, \overrightarrow{r_y \mapsto y}\}$
- $\Gamma = \{\overrightarrow{x : \theta}, \overrightarrow{y : \theta'}\}$
- $\Gamma; \Sigma; \Psi \vdash \rho$
- $\Sigma; \Psi \vdash \sigma; \pi$

- $\mu_a; \nu_a; \pi; \vec{\rho}, \rho \vdash H \leftrightsquigarrow \sigma$
- $\mu_a; \vec{\mu_\Gamma}, \mu_\Gamma; \sigma \vdash \vec{R}, R \leftrightsquigarrow \vec{\rho}, \rho$
- $\lambda_x; \vec{R} \vdash \langle H, R, \lambda_x(a)\rangle \xrightarrow{b} \langle H', R', b'\rangle$

then

- $\Sigma; \lambda_c; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \lambda_c(l)\rangle \xrightarrow{s} \mathsf{err}$ or
- $\Sigma; \lambda_c; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \lambda(l)\rangle \xrightarrow{s} \langle \sigma', \pi', s'\rangle$.

Proposition 18 (Preservation of Related Memory for Function Definitions)**.** If

- $\mu_\lambda; \mu_\Gamma; \Gamma; \Sigma \vdash b \overset{b}{\rightsquigarrow} s$
- $\mu_\Gamma = \{\overrightarrow{r_x \mapsto x}, \overrightarrow{r_y \mapsto y}\}$
- $\Gamma = \{\overrightarrow{x : \theta}, \overrightarrow{y : \theta'}\}$
- $\Gamma; \Sigma; \Psi \vdash \rho$
- $\Sigma; \Psi \vdash \sigma; \pi$
- $\mu_a; \nu_a; \pi; \vec{\rho}, \rho \vdash H \leftrightsquigarrow \sigma$
- $\mu_a; \vec{\mu_\Gamma}, \mu_\Gamma; \sigma \vdash \vec{R}, R \leftrightsquigarrow \vec{\rho}, \rho$
- $\lambda_x; \vec{R} \vdash \langle H, R, \lambda_x(a)\rangle \xrightarrow{b} \langle H', R', b'\rangle$
- $\Sigma; \lambda_c; \vec{\rho}; \rho \vdash \langle \sigma, \pi, \lambda(l)\rangle \xrightarrow{s} \langle \sigma', \pi', s'\rangle$.

then for some $\mu'_a \supseteq \mu_a$ and $\nu'_a \supseteq \nu_a$:

- $\mu'_a; \vec{\mu_\Gamma}, \mu_\Gamma; \sigma' \vdash \vec{R}, R \leftrightsquigarrow \vec{\rho}, \rho$
- $\mu'_a; \nu'_a; \pi'; \vec{\rho}, \rho \vdash H' \leftrightsquigarrow \sigma'$

Proof. The proof is straightforward. □

A.2 Simple and Efficient Algorithms for Octagons

Lemma 1 (Correctness of CHECKCONSISTENT). *Suppose* $\mathbf{m}$ *is a closed DBM,* $\mathbf{m}' = \text{INCCLOSE}(\mathbf{m}, o)$ *and* $o = (x'_a - x'_b \leqslant d)$*. If* $\mathbf{m}'$ *is consistent then*

- $\mathbf{m}_{b,a} + d \geqslant 0$
- $\mathbf{m}_{\bar{a},\bar{b}} + d \geqslant 0$
- $\mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,\bar{b}} + d \geqslant 0$
- $\mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d \geqslant 0$

Proof. Since $\mathbf{m}'$ is consistent $\mathbf{m}'_{\bar{a},\bar{a}} \geqslant 0$ hence

$$\min \begin{pmatrix} \mathbf{m}_{\bar{a},\bar{a}}, \\ \mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,\bar{a}}, \\ \mathbf{m}_{\bar{a},\bar{b}} + d + \mathbf{m}_{\bar{a},\bar{a}}, \\ \mathbf{m}_{\bar{a},\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,\bar{a}}, \\ \mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},\bar{a}} \end{pmatrix} = \mathbf{m}'_{\bar{a},\bar{a}} \geqslant 0$$

Therefore $\mathbf{m}_{\bar{a},\bar{b}} + d + \mathbf{m}_{\bar{a},\bar{a}} \geqslant 0$ and $\mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},\bar{a}} \geqslant 0$. Since $\mathbf{m}$ is closed $\mathbf{m}_{\bar{a},\bar{a}} = 0$ hence $\mathbf{m}_{\bar{a},\bar{b}} + d \geqslant 0$ and $\mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,\bar{b}} + d \geqslant 0$.

Repeating the argument $\mathbf{m}'_{b,b} \geqslant 0$ hence

$$\min \begin{pmatrix} \mathbf{m}_{b,b}, \\ \mathbf{m}_{b,a} + d + \mathbf{m}_{b,b}, \\ \mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},b}, \\ \mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,b}, \\ \mathbf{m}_{b,a} + d + \mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},b} \end{pmatrix} = \mathbf{m}'_{b,b} \geqslant 0$$

Therefore $\mathbf{m}_{b,a} + d + \mathbf{m}_{b,b} \geqslant 0$ and $\mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,b} \geqslant 0$. Since $\mathbf{m}_{b,b} = 0$ it follows that $\mathbf{m}_{b,a} + d \geqslant 0$ and $\mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d \geqslant 0$. □ □

Theorem 1 (Correctness of INCCLOSE). *Suppose* $\mathbf{m}$ *is a closed DBM,* $\mathbf{m}' = \text{INCCLOSE}(\mathbf{m}, o)$ *and* $o = (x'_a - x'_b \leqslant d)$*. Then* $\mathbf{m}'$ *is either closed or it is not consistent.*

Proof. Suppose $\mathbf{m}'$ is consistent. Because $\mathbf{m}$ is closed $0 = \mathbf{m}_{i,i} \geqslant \mathbf{m}'_{i,i} \geqslant 0$ hence $\mathbf{m}'_{i,i} = 0$. It therefore remains to show $\forall i,j,k.\mathbf{m}'_{i,k} + \mathbf{m}'_{k,j} \geqslant A$ where

$$A = \min \begin{pmatrix} \mathbf{m}_{i,j}, \\ \mathbf{m}_{i,a} + d + \mathbf{m}_{b,j}, \\ \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},j}, \\ \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,j}, \\ \mathbf{m}_{i,a} + d + \mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},j} \end{pmatrix}$$

There are 5 cases for $\mathbf{m}'_{i,k}$ and 5 for $\mathbf{m}'_{k,j}$ giving 25 in total:

1-1. Suppose $\mathbf{m}'_{i,k} = \mathbf{m}_{i,k}$ and $\mathbf{m}'_{k,j} = \mathbf{m}_{k,j}$. Because $\mathbf{m}$ is closed:

$$\mathbf{m}'_{i,k} + \mathbf{m}'_{k,j} = \mathbf{m}_{i,k} + \mathbf{m}_{k,j} \geqslant \mathbf{m}_{i,j} \geqslant A$$

1-2. Suppose $\mathbf{m}'_{i,k} = \mathbf{m}_{i,k}$ and $\mathbf{m}'_{k,j} = \mathbf{m}_{k,a} + d + \mathbf{m}_{b,j}$. Because $\mathbf{m}$ is closed:

$$\mathbf{m}'_{i,k} + \mathbf{m}'_{k,j} = \mathbf{m}_{i,k} + \mathbf{m}_{k,a} + d + \mathbf{m}_{b,j} \geqslant \mathbf{m}_{i,a} + d + \mathbf{m}_{b,j} \geqslant A$$

1-3. Suppose $\mathbf{m}'_{i,k} = \mathbf{m}_{i,k}$ and $\mathbf{m}'_{k,j} = \mathbf{m}_{k,\bar{b}} + d + \mathbf{m}_{\bar{a},j}$. Because $\mathbf{m}$ is closed:

$$\mathbf{m}'_{i,k} + \mathbf{m}'_{k,j} = \mathbf{m}_{i,k} + \mathbf{m}_{k,\bar{b}} + d + \mathbf{m}_{\bar{a},j} \geqslant \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},j} \geqslant A$$

1-4. Suppose $\mathbf{m}'_{i,k} = \mathbf{m}_{i,k}$ and $\mathbf{m}'_{k,j} = \mathbf{m}_{k,\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,j}$. Because $\mathbf{m}$ is closed:

$$\begin{aligned} \mathbf{m}'_{i,k} + \mathbf{m}'_{k,j} &= \mathbf{m}_{i,k} + \mathbf{m}_{k,\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,j} \\ &\geqslant \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,j} \geqslant A \end{aligned}$$

1-5. Suppose $\mathbf{m}'_{i,k} = \mathbf{m}_{i,k}$ and $\mathbf{m}'_{k,j} = \mathbf{m}_{k,a} + d + \mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},j}$. Because $\mathbf{m}$ is closed:

$$\begin{aligned} \mathbf{m}'_{i,k} + \mathbf{m}'_{k,j} &= \mathbf{m}_{i,k} + \mathbf{m}_{k,a} + d + \mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},j} \\ &\geqslant \mathbf{m}_{i,a} + d + \mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},j} \geqslant A \end{aligned}$$

2-1. Suppose $\mathbf{m'}_{i,k} = \mathbf{m}_{i,a} + d + \mathbf{m}_{b,k}$ and $\mathbf{m'}_{k,j} = \mathbf{m}_{k,j}$. Symmetric to case 1-2.

2-2. Suppose $\mathbf{m'}_{i,k} = \mathbf{m}_{i,a} + d + \mathbf{m}_{b,k}$ and $\mathbf{m'}_{k,j} = \mathbf{m}_{k,a} + d + \mathbf{m}_{b,j}$. Because $\mathbf{m}$ is closed and by Lemma 1:

$$\begin{aligned}\mathbf{m'}_{i,k} + \mathbf{m'}_{k,j} &= \mathbf{m}_{i,a} + d + \mathbf{m}_{b,k} + \mathbf{m}_{k,a} + d + \mathbf{m}_{b,j} \\ &\geqslant \mathbf{m}_{i,a} + d + \mathbf{m}_{b,a} + d + \mathbf{m}_{b,j} \geqslant \mathbf{m}_{i,a} + d + \mathbf{m}_{b,j} \geqslant A\end{aligned}$$

2-3. Suppose $\mathbf{m'}_{i,k} = \mathbf{m}_{i,a} + d + \mathbf{m}_{b,k}$ and $\mathbf{m'}_{k,j} = \mathbf{m}_{k,\bar{b}} + d + \mathbf{m}_{\bar{a},j}$. Because $\mathbf{m}$ is closed:

$$\begin{aligned}\mathbf{m'}_{i,k} + \mathbf{m'}_{k,j} &= \mathbf{m}_{i,a} + d + \mathbf{m}_{b,k} + \mathbf{m}_{k,\bar{b}} + d + \mathbf{m}_{\bar{a},j} \\ &\geqslant \mathbf{m}_{i,a} + d + \mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},j} \geqslant A\end{aligned}$$

2-4. Suppose $\mathbf{m'}_{i,k} = \mathbf{m}_{i,a} + d + \mathbf{m}_{b,k}$ and $\mathbf{m'}_{k,j} = \mathbf{m}_{k,\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,j}$. Because $\mathbf{m}$ is closed and by Lemma 1:

$$\begin{aligned}\mathbf{m'}_{i,k} + \mathbf{m'}_{k,j} &= \mathbf{m}_{i,a} + d + \mathbf{m}_{b,k} + \mathbf{m}_{k,\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,j} \\ &\geqslant \mathbf{m}_{i,a} + d + \mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,j} \\ &\geqslant \mathbf{m}_{i,a} + d + \mathbf{m}_{b,j} \geqslant A\end{aligned}$$

2-5. Suppose $\mathbf{m'}_{i,k} = \mathbf{m}_{i,a} + d + \mathbf{m}_{b,k}$ and $\mathbf{m'}_{k,j} = \mathbf{m}_{k,a} + d + \mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},j}$. Because $\mathbf{m}$ is closed and by Lemma 1:

$$\begin{aligned}\mathbf{m'}_{i,k} + \mathbf{m'}_{k,j} &= \mathbf{m}_{i,a} + d + \mathbf{m}_{b,k} + \mathbf{m}_{k,a} + d + \mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},j} \\ &\geqslant \mathbf{m}_{i,a} + d + \mathbf{m}_{b,a} + d + \mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},j} \\ &\geqslant \mathbf{m}_{i,a} + d + \mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},j} \geqslant A\end{aligned}$$

3-1. Suppose $\mathbf{m'}_{i,k} = \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},k}$ and $\mathbf{m'}_{k,j} = \mathbf{m}_{k,j}$. Symmetric to case 1-3.

3-2. Suppose $\mathbf{m'}_{i,k} = \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},k}$ and $\mathbf{m'}_{k,j} = \mathbf{m}_{k,a} + d + \mathbf{m}_{b,j}$. Symmetric to case 2-3.

3-3. Suppose $\mathbf{m'}_{i,k} = \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},k}$ and $\mathbf{m'}_{k,j} = \mathbf{m}_{k,\bar{b}} + d + \mathbf{m}_{\bar{a},j}$. Because $\mathbf{m}$ is closed and by Lemma 1:

$$\begin{aligned}
\mathbf{m'}_{i,k} + \mathbf{m'}_{k,j} &= \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},k} + \mathbf{m}_{k,\bar{b}} + d + \mathbf{m}_{\bar{a},j} \\
&\geqslant \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},\bar{b}} + d + \mathbf{m}_{\bar{a},j} \\
&\geqslant \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},j} \geqslant A
\end{aligned}$$

3-4. Suppose $\mathbf{m'}_{i,k} = \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},k}$ and $\mathbf{m'}_{k,j} = \mathbf{m}_{k,\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,j}$. Because $\mathbf{m}$ is closed and by Lemma 1:

$$\begin{aligned}
\mathbf{m'}_{i,k} + \mathbf{m'}_{k,j} &= \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},k} + \mathbf{m}_{k,\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,j} \\
&\geqslant \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,j} \\
&\geqslant \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,j} \geqslant A
\end{aligned}$$

3-5. Suppose $\mathbf{m'}_{i,k} = \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},k}$ and $\mathbf{m'}_{k,j} = \mathbf{m}_{k,a} + d + \mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},j}$. Because $\mathbf{m}$ is closed and by Lemma 1:

$$\begin{aligned}
\mathbf{m'}_{i,k} + \mathbf{m'}_{k,j} &= \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},k} + \mathbf{m}_{k,a} + d + \mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},j} \\
&= \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},j} \\
&= \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},j} \geqslant A
\end{aligned}$$

4-1. Suppose $\mathbf{m'}_{i,k} = \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,k}$ and $\mathbf{m'}_{k,j} = \mathbf{m}_{k,j}$. Symmetric to case 1-4.

4-2. Suppose $\mathbf{m'}_{i,k} = \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,k}$ and $\mathbf{m'}_{k,j} = \mathbf{m}_{k,a} + d + \mathbf{m}_{b,j}$. Symmetric to case 2-4.

4-3. Suppose $\mathbf{m'}_{i,k} = \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,k}$ and $\mathbf{m'}_{k,j} = \mathbf{m}_{k,\bar{b}} + d + \mathbf{m}_{\bar{a},j}$. Symmetric to case 3-4.

4-4. Suppose $\mathbf{m'}_{i,k} = \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,k}$ and $\mathbf{m'}_{k,j} = \mathbf{m}_{k,\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,j}$.

Because $\mathbf{m}$ is closed and by Lemma 1:

$$\begin{aligned}\mathbf{m'}_{i,k} + \mathbf{m'}_{k,j} &= \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,k} + \mathbf{m}_{k,\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,j} \\ &\geqslant \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,j} \\ &\geqslant \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,j} \geqslant A\end{aligned}$$

4-5. Suppose $\mathbf{m'}_{i,k} = \mathbf{m}_{i,\bar{b}}+d+\mathbf{m}_{\bar{a},a}+d+\mathbf{m}_{b,k}$ and $\mathbf{m'}_{k,j} = \mathbf{m}_{k,a}+d+\mathbf{m}_{b,\bar{b}}+d+\mathbf{m}_{\bar{a},j}$. Because $\mathbf{m}$ is closed and by Lemma 1:

$$\begin{aligned}\mathbf{m'}_{i,k} + \mathbf{m'}_{k,j} &= \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,k} + \mathbf{m}_{k,a} + d + \mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},j} \\ &\geqslant \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,a} + d + \mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},j} \\ &\geqslant \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},j} \\ &\geqslant \mathbf{m}_{i,\bar{b}} + d + \mathbf{m}_{\bar{a},j} \geqslant A\end{aligned}$$

5-1. Suppose $\mathbf{m'}_{i,k} = \mathbf{m}_{i,a} + d + \mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},k}$ and $\mathbf{m'}_{k,j} = \mathbf{m}_{k,j}$. Symmetric to case 1-5.

5-2. Suppose $\mathbf{m'}_{i,k} = \mathbf{m}_{i,a} + d + \mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},k}$ and $\mathbf{m'}_{k,j} = \mathbf{m}_{k,a} + d + \mathbf{m}_{b,j}$. Symmetric to case 2-5.

5-3. Suppose $\mathbf{m'}_{i,k} = \mathbf{m}_{i,a} + d + \mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},k}$ and $\mathbf{m'}_{k,j} = \mathbf{m}_{k,\bar{b}} + d + \mathbf{m}_{\bar{a},j}$. Symmetric to case 3-5.

5-4. Suppose $\mathbf{m'}_{i,k} = \mathbf{m}_{i,a}+d+\mathbf{m}_{b,\bar{b}}+d+\mathbf{m}_{\bar{a},k}$ and $\mathbf{m'}_{k,j} = \mathbf{m}_{k,\bar{b}}+d+\mathbf{m}_{\bar{a},a}+d+\mathbf{m}_{b,j}$. Symmetric to case 4-5.

5-5. Suppose $\mathbf{m'}_{i,k} = \mathbf{m}_{i,a}+d+\mathbf{m}_{b,\bar{b}}+d+\mathbf{m}_{\bar{a},k}$ and $\mathbf{m'}_{k,j} = \mathbf{m}_{k,a}+d+\mathbf{m}_{b,\bar{b}}+d+\mathbf{m}_{\bar{a},j}$. Because $\mathbf{m}$ is closed and by Lemma 1:

$$\begin{aligned}\mathbf{m'}_{i,k} + \mathbf{m'}_{k,j} &= \mathbf{m}_{i,a} + d + \mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},k} + \mathbf{m}_{k,a} + d + \mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},j} \\ &\geqslant \mathbf{m}_{i,a} + d + \mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},a} + d + \mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},j} \\ &\geqslant \mathbf{m}_{i,a} + d + \mathbf{m}_{b,\bar{b}} + d + \mathbf{m}_{\bar{a},j} \geqslant A\end{aligned}$$

□

□

Theorem 2 (Correctness of Strong Closure). *Suppose* $\mathbf{m}$ *is a closed, coherent DBM and* $\mathbf{m}' = \textsc{Str}(\mathbf{m})$. *Then* $\mathbf{m}'$ *is a strongly closed DBM.*

Proof. Observe that $\mathbf{m}'_{i,\bar{\imath}} = \min(\mathbf{m}_{i,\bar{\imath}}, (\mathbf{m}_{i,\bar{\imath}} + \mathbf{m}_{i,\bar{\imath}})/2) = \mathbf{m}_{i,\bar{\imath}}$ and likewise $\mathbf{m}'_{j,\bar{\jmath}} = \mathbf{m}_{j,\bar{\jmath}}$. Therefore $\mathbf{m}'_{i,j} \leqslant (\mathbf{m}_{i,\bar{\imath}} + \mathbf{m}_{\bar{\jmath},j})/2 = (\mathbf{m}'_{i,\bar{\imath}} + \mathbf{m}'_{\bar{\jmath},j})/2$.

Because $\mathbf{m}$ is closed $0 = \mathbf{m}_{i,i} \leqslant \mathbf{m}_{i,\bar{\imath}} + \mathbf{m}_{\bar{\imath},i}$ and thus

$$\mathbf{m}'_{i,i} = \min(\mathbf{m}_{i,i}, (\mathbf{m}_{i,\bar{\imath}} + \mathbf{m}_{\bar{\imath},i})/2) = \min(0, (\mathbf{m}_{i,\bar{\imath}} + \mathbf{m}_{\bar{\imath},i})/2) = 0$$

To show $\mathbf{m}'_{i,j} \leqslant \mathbf{m}'_{i,k} + \mathbf{m}'_{k,j}$ we proceed by case analysis:

- Suppose $\mathbf{m}'_{i,k} = \mathbf{m}_{i,k}$ and $\mathbf{m}'_{k,j} = \mathbf{m}_{k,j}$. Because $\mathbf{m}$ is closed:

$$\mathbf{m}'_{i,j} \leqslant \mathbf{m}_{i,j} \leqslant \mathbf{m}_{i,k} + \mathbf{m}_{k,j} = \mathbf{m}'_{i,k} + \mathbf{m}'_{k,j}$$

- Suppose $\mathbf{m}'_{i,k} \neq \mathbf{m}_{i,k}$ and $\mathbf{m}'_{k,j} = \mathbf{m}_{k,j}$. Because $\mathbf{m}$ is closed and coherent:

$$\begin{aligned} 2\mathbf{m}'_{i,k} + 2\mathbf{m}'_{k,j} &= \mathbf{m}_{i,\bar{\imath}} + \mathbf{m}_{\bar{k},k} + 2\mathbf{m}_{k,j} \geqslant \mathbf{m}_{i,\bar{\imath}} + \mathbf{m}_{\bar{k},j} + \mathbf{m}_{k,j} \\ &= \mathbf{m}_{i,\bar{\imath}} + \mathbf{m}_{\bar{\jmath},k} + \mathbf{m}_{k,j} \geqslant \mathbf{m}_{i,\bar{\imath}} + \mathbf{m}_{\bar{\jmath},j} \geqslant 2\mathbf{m}'_{i,j} \end{aligned}$$

- Suppose $\mathbf{m}'_{i,k} = \mathbf{m}_{i,k}$ and $\mathbf{m}'_{k,j} \neq \mathbf{m}_{k,j}$. Symmetric to the previous case.

- Suppose $\mathbf{m}'_{i,k} \neq \mathbf{m}_{i,k}$ and $\mathbf{m}'_{k,j} \neq \mathbf{m}_{k,j}$. Because $\mathbf{m}$ is closed:

$$\begin{aligned} 2\mathbf{m}'_{i,k} + 2\mathbf{m}'_{k,j} &= \mathbf{m}_{i,\bar{\imath}} + \mathbf{m}_{\bar{k},k} + \mathbf{m}_{k,\bar{k}} + \mathbf{m}_{\bar{\jmath},j} \\ &\geqslant \mathbf{m}_{i,\bar{\imath}} + \mathbf{m}_{\bar{k},\bar{k}} + \mathbf{m}_{\bar{\jmath},j} = \mathbf{m}_{i,\bar{\imath}} + 0 + \mathbf{m}_{\bar{\jmath},j} \geqslant 2\mathbf{m}'_{i,j} \end{aligned}$$

□

Bibliography

[1] S. Abramsky and C. Hankin. An introduction to abstract interpretation. In *Abstract Interpretation of Declarative Languages*. Ellis Horwood, 1987.

[2] A. Armando, C. Castellini, E. Giunchiglia, M. Idini, and M. Maratea. TSAT++: an Open Platform for Satisfiability Modulo Theories. *Electronic Notes in Theoretical Computer Science*, 125(3):25–36, 2005.

[3] R. Bagnara, P. M. Hill, and E. Zaffanella. Weakly-relational Shapes for Numeric Abstractions: Improved Algorithms and Proofs of Correctness. *Formal Methods in System Design*, 35(3):279–323, 2009.

[4] G. Balakrishnan and T. Reps. Analyzing Memory Accesses in x86 Executables. In *CC*, volume 2985 of *LNCS*, pages 5–23. Springer, 2004.

[5] G. Balakrishnan and T. Reps. Divine: Discovering Variables in Executables. In *VMCAI*, volume 4349 of *LNCS*, pages 1–28. Springer, 2007.

[6] J. Baldwin, A. Teh, E. Baniassad, D. van Rooy, and Y. Coady. Requirements for tools for comprehending highly specialized assembly language code and how to elicit these requirements. *Requirements Engineering*, 21(1):131–159, 2016.

[7] C. Barrett, A. Stump, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

[8] E. Barrett. *Range Analysis of Binaries with Decision Procedures* . PhD thesis, University of Kent, 2013.

[9] A. Bauer, M. Leucker, C. Schallhart, and M. Tautschnig. Don't care in SMT: building flexible yet efficient abstraction/refinement solvers. *International Journal on Software Tools for Technology Transfer*, 12(1):23–37, 2010.

[10] C. Baykan and M. Fox. Spatial Synthesis by Disjunctive Constraint Satisfaction. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 11(4):245–262, 1997.

[11] R. Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[12] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *Computer Aided Verification*, pages 178–192. Springer, 2007.

[13] J. Berdine, A. Chawdhary, B. Cook, D. Distefano, and P. O'Hearn. Variance Analyses from Invariance Analyses. In *POPL*, pages 211–224. ACM, 2007.

[14] A. Biere. PicoSAT Essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:75–97, 2008.

[15] S. Blazy and T. Jensen, editors. *Static Analysis = 22nd International Symposium, SAS, Saint-Malo, France, September 9-11*, volume 9291 of *LNCS*. Springer, 2015.

[16] S. Blazy, V. Laporte, and D. Pichardie. Verified Abstract Interpretation Techniques for Disassembling Low-level Self-modifying Code. *Journal of Automated Reasoning*, 56(3):283–308, 2016.

[17] R. Bruttomesso, D. Cok, and A. Griggio. SMT-COMP 2012. `http://smtcomp.sourceforge.net/2012/`, 2012.

[18] J. Caballero, G. Grieco, M. Marron, Z. Lin, and D. Urbina. ARTISTE: Automatic Generation of Hybrid Data Structure Signatures from Binary Code Executions. Technical report, TR-IMDEA-SW-2012-001, IMDEA Software Institute, Madrid, Spain, 2012.

[19] J. Caballero and Z. Lin. Type inference on executables. *ACM Computing Surveys*, 48(4):65:1–65:35, 2016.

[20] R. Canion. *Open: How Compaq Ended IBM's PC Domination and Helped Invent Modern Computing*. BenBella Books, 2013.

[21] M. Carlsson, G. Ottosson, and B. Carlson. An Open-Ended Finite Domain Constraint Solver. In *Programming Languages: Implementations, Logics, and Programs*, volume 1292 of *LNCS*, pages 191–206. Springer, 1997.

[22] E. Chan, S. Venkataraman, N. Tkach, K. Larson, A. Gutierrez, and R. H. Campbell. Characterizing Data Structures for Volatile Forensics. In *Systematic Approaches to Digital Forensic Engineering*, pages 1–9, 2011.

[23] A. Chawdhary, E. Robbins, and A. King. Simple and Efficient Algorithms for Octagons. In *APLAS*, volume 8858 of *LNCS*, pages 296–313. Springer, 2014.

[24] A. Chawdhary, E. Robbins, and A. King. Incrementally closing octagons. *Submitted to Formal Methods in System Design*, 2016. Manuscript available at `https://arxiv.org/pdf/1610.02952.pdf`.

[25] V. Chipounov and G. Candea. Reverse Engineering of Binary Device Drivers with RevNIC. In *European Conference on Computer Systems*, pages 167–180. ACM Press, 2010.

[26] C. Cifuentes. An environment for the reverse engineering of executable programs. In *Asia-Pacific Software Engineering Conference (APSEC)*, pages 410–419. IEEE Computer Society, 1995.

[27] M. Codish, V. Lagoon, and P. J. Stuckey. Logic Programming with Satisfiability. *Theory and Practice of Logic Programming*, 8(1):121–128, 2008.

[28] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Program by Construction or Approximation of Fixpoints. In *POPL*, pages 238–252. ACM Press, 1977.

[29] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE Analyser. In *European Symposium on Programming*, volume 3444 of *LNCS*, pages 21–30. Springer, 2005.

[30] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *POPL*, pages 84–97. ACM Press, 1978.

[31] A. Cozzie, F. Stratton, H. Xue, and S. T. King. Digging For Data Structures. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 231–244. USENIX, 2008.

[32] W. Cui, J. Kannan, and H. Wang. Discoverer: Automatic Protocol Reverse Engineering from Network Traces. In *USENIX Security Symposium*, pages 14:1–14:14. USENIX Association, 2007.

[33] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[34] M. Davis, G. Logemann, and D. Loveland. A Machine Program for Theorem Proving. *Communications of the ACM*, 5(7):394–397, 1962.

[35] S. Debray. Abstract interpretation and low-level code optimization. In *Partial Evaluation and Semantics-based Program Manipulation (PEPM)*, pages 111–121. ACM, 1995.

[36] B. Demoen, M. García de la Banda, and P. J. Stuckey. Type Constraint Solving for Parametric and Ad-hoc Polymorphism. In *Australian Computer Science Conference*, pages 217–228. Springer, 1999.

[37] E. Dolgova and A. Chernov. Automatic Reconstruction of Data types in the Decompilation Problem. *Programming and Computer Software*, 35(2):105–119, 2009.

[38] W. Drabent. Logic + Control: An Example. In *ICLP (Technical Communications)*, volume 17 of *LIPIcs*, pages 301–311. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012.

[39] D. D'Souza, A. Lal, and K. Larsen, editors. *Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI, Mumbai, India, January 12-14*, volume 8931 of *LNCS*. Springer, 2015.

[40] Z. Durumeric, J. Kasten, D. Adrian, J. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer, and V. Paxson. The matter of heartbleed. In *Internet Measurement Conference*, pages 475–488. ACM, 2014.

[41] K. ElWazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua. Scalable variable and data type detection in a binary rewriter. In *PLDI*, pages 51–60. ACM, 2013.

[42] K. Elwazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua. Scalable Variable and Data Type Detection in a Binary Rewriter. In *PLDI*, pages 51–60, 2013.

[43] R. Floyd. Algorithm 97: Shortest Path. *Communications of the ACM*, 5(6):345, 1962.

[44] F. Freidman. Inverse compilation feasibility. In *ACM Annual Conference*, pages 750–750. ACM, 1974.

[45] T. Frühwirth. *Constraint Handling Rules*. CUP, 2009.

[46] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast Decision Procedures. In *Computer Aided Verification*, pages 175–188. Springer, 2004.

[47] Y. Guéhéneuc, B. Adams, and A. Serebrenik, editors. *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER, Montreal, QC, Canada, March 2-6*. IEEE Computer Society, 2015.

[48] B. Guha, C. Davis, and B. Mukherjee. Network Security via Reverse Engineering of TCP code: Vulnerability Analysis and Proposed Solutions. In *Conference on Computer Communications*, pages 603–610. IEEE Computer Society, 1996.

[49] I. Guilfanov. A Simple Type System for Program Reengineering. In *WCRE*, pages 357–. IEEE Computer Society, 2001.

[50] B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani. Automatically Refining Abstract Interpretations. In *TACAS*, volume 4963 of *LNCS*, pages 443–458. Springer, 2008.

[51] I. Haller, A. Slowinska, and H. Bos. MemPick: High-level data structure detection in C/C++ binaries. In *Working Conference on Reverse Engineering*, pages 32–41. IEEE Computer Society, 2013.

[52] M. Halstead. *Machine Independent Computer Programming*. Spartan Books, 1962.

[53] M. Halstead. Machine-independence and third-generation computers. In *Fall Joint Computer Conference*, pages 587–592. ACM, 1967.

[54] W. H. Harrison. Compiler Analysis for the Value Ranges of Variables. *IEEE Transactions on Software Engineering*, SE-3(3):243–250, 1977.

[55] W. Harvey and P. J. Stuckey. A Unit Two Variable Per Inequality Integer Constraint Solver for Constraint Logic Programming. In M. Patel and R. Kotagiri, editors, *Twentieth Australasian Computer Science Conference*, pages 102–111. Macquarie University, 1997. Also available as TR 95/30 from University of Melbourne.

[56] M. Hermenegildo, D. Cabeza, and M. Carro. Using Attributed Variables in the Implementation of Concurrent and Parallel Logic Programming Systems. In *International Conference on Logic Programming*, pages 631–645. MIT Press, 1995.

[57] Hex-Rays SA. IDA: About, 2016. `http://https://www.hex-rays.com/products/ida/`.

[58] C. Hollander. A syntax-directed approach to inverse compilation. In *ACM Annual Conference*, pages 750–750. ACM, 1974.

[59] C. Holzbaur. Metastructures vs. Attributed Variables in the Context of Extensible Unification. In *International Symposium on Programming Language Implementation and Logic Programming*, volume 631 of *LNCS*, pages 260–268. Springer, 1992.

[60] B. Housel and M. Halstead. A methodology for machine language decompilation. In *ACM Annual Conference*, pages 254–260. ACM, 1974.

[61] J. M. Howe and A. King. Positive Boolean Functions as Multiheaded Clauses. In *International Conference on Logic Programming*, volume 2237 of *LNCS*, pages 120–134. Springer, 2001.

[62] J. M. Howe and A. King. Efficient Groundness Analysis in Prolog. *Theory and Practice of Logic Programming*, 3(1):95–124, 2003.

[63] J. M. Howe and A. King. A Pearl on SAT Solving in Prolog. In *Functional and Logic Programming*, volume 6009 of *LNCS*, pages 165–174. Springer, 2010.

[64] J. M. Howe and A. King. A Pearl on SAT and SMT Solving in Prolog. *Theoretical Computer Science*, 435:43–55, 2012.

[65] G. Huet. *Résolution d'équations dans les langages d'ordre 1, 2, . . . , ω.* PhD thesis, Université Paris VII, 1976.

[66] Gérard Huet. The Zipper. *Journal of Functional Programming*, 7(5):549–554, 1997.

[67] Intel Security-McAfee. Net Losses: Estimating the Global Cost of Cybercrime, 2014. `http://www.mcafee.com/uk/resources/reports/rp-economic-impact-cybercrime2.pdf`.

[68] J. Jaffar. Efficient Unification over Infinite Terms. *New Generation Computing*, 2(3):207–219, 1984.

[69] B. Jeannet and A. Miné. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *CAV*, volume 5643 of *LNCS*, pages 661–667. Springer, 2009.

[70] R. G. Jeroslow and J. Wang. Solving Propositional Satisfiability Problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990.

[71] R. Jhala and R. Majumdar. Software Model Checking. *ACM Computing Surveys*, 41(4):21:1–21:54, 2009.

[72] H. S. Warren Jr. *Hacker's Delight*. Addison-Wesley, 2002.

[73] C. Jung and N. Clark. DDT: Design and Evaluation of a Dynamic Program Analysis for Optimizing Data Structure Usage. In *IEEE/ACM International Symposium on Microarchitecture*, pages 56–66. ACM, 2009.

[74] U. Junker. QuickXPlain: Preferred Explanations and Relaxations for Over-Constrained Problems. In *AAAI*, pages 167–172. AAAI Press, 2004.

[75] S. Katsumata and A. Ohori. Proof-Directed De-compilation of Low-Level Code. In *ESOP*, volume 2028 of *LNCS*, pages 352–366. Springer, 2001.

[76] Neil Kettle. Digit security. Private communication, 2016.

[77] J. Kinder, H. Veith, and F. Zuleger. An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries. In *VMCAI*, volume 5403 of *LNCS*, pages 214–228. Springer, 2009.

[78] R. A. Kowalski. Algorithm = Logic + Control. *Communication of the ACM*, 22(7):424–436, 1979.

[79] D. Kroening and O. Strichman. *Decision Procedures*. Springer, 2008.

[80] R. Lämmel, R. Oliveto, and R. Robbes, editors. *20th Working Conference on Reverse Engineering, WCRE, Koblenz, Germany, October 14-17*. IEEE Computer Society, 2013.

[81] J. Lee, T. Avgerinos, and D. Brumley. TIE: Principled Reverse Engineering of Types in Binary Programs. In *Network and Distributed System Security Symposium*. The Internet Society, 2011.

[82] X. Leroy. Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assistant. In *POPL*, pages 42–54, 2006.

[83] C. M. Li and Anbulagan. Look-Ahead Versus Look-Back for Satisfiability Problems. In *Constraint Programming*, volume 1330 of *LNCS*, pages 341–355. Springer, 1997.

[84] Z. Lin, X. Zhang, and D. Xu. Automatic Reverse Engineering of Data Structures from Binary Execution. In *Network and Distributed System Security Symposium*. The Internet Society, 2010.

[85] D. MacQueen, G. Plotkin, and R. Sethi. An Ideal Model for Recursive Polymorphic Types. In *Principles of Programming Languages*, pages 165–174. ACM Press, 1983.

[86] S. Magill, J. Berdine, E. Clarke, and B. Cook. Arithmetic Strengthening for Shape Analysis. In *SAS*, volume 4634 of *LNCS*, pages 419–436. Springer, 2007.

[87] M. J. Maher. Complete Axiomatizations of the Algebras of Finite, Rational and Infinite Trees. In *Logic in Computer Science*, pages 348–357. IEEE Computer Society, 1988.

[88] A. Matrosov, E. Rodionov, D. Harley, and J. Malcho. Stuxnet Under the Microscope. Technical report, ESET, 06 2011.

[89] J. McDonald, M. Preda, and N. Stakhanova, editors. *Proceedings of the 5th Program Protection and Reverse Engineering Workshop, PPREW, Los Angeles, CA, USA, December 8*. ACM, 2015.

[90] B. P. Miller and A. R. Bernat. Anywhere, Any Time Binary Instrumentation. In *Workshop on Program Analysis for Software Tools and Engineering*, September 2011. See also `http://www.dyninst.org/dyninst`.

[91] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Science*, 17:348–375, 1978.

[92] A. Miné. A New Numerical Abstract Domain Based on Difference-Bound Matrices. In *Programs as Data Objects*, volume 2053 of *LNCS*, pages 155–172. Springer, 2001.

[93] A. Miné. *Weakly Relational Numerical Abstract Domains*. PhD thesis, École Polytechnique, 2004. `http://www.di.ens.fr/~mine/these/these-color.pdf`.

[94] A. Miné. The Octagon Abstract Domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[95] D. Moore, C. Shannon, and K. Claffy. Code-red: A case study on the spread and victims of an internet worm. In *Workshop on Internet Measurment*, pages 273–284. ACM, 2002.

[96] G. Morrisett and D. Walker. From System F to Typed Assembly Language. *TOPLAS*, 21(3):527–568, 1999.

[97] J. Morrison. Blaster revisited. *Queue*, 2(4):34–43, June 2004.

[98] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Design Automation Conference*, pages 530–535. ACM Press, 2001.

[99] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[100] A. Mycroft. Type-Based Decompilation (or Program Reconstruction via Type Reconstruction). In *European Symposium on Programming*, volume 1576 of *LNCS*, pages 208–223. Springer, 1999.

[101] M. O. Myreen, M. J. C. Gordon, and K. Slind. Machine-Code Verification for Multiple Architectures - An Application of Decompilation into Logic. In *FMCAD*, pages 1–8, 2008.

[102] L. Naish. *Negation and Control in Logic Programs*, volume 238 of *LNCS*. Springer, 1986.

[103] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.

[104] M. Noonan, A. Loginov, and D. Cok. Polymorphic type inference for machine code. In *PLDI*, pages 27–41. ACM, 2016.

[105] M. P. Peres Cervantes. *Static Methods to Check Low-Level Code for a Graph Reduction Machine*. PhD thesis, University of York, 2014. `http://etheses.whiterose.ac.uk/id/eprint/6248`.

[106] D. Pham, J. Thornton, and A. Sattar. Modelling and Solving Temporal Reasoning as Propositional Satisfiability. *Artificial Intelligence*, 172(15):1752–1782, 2008.

[107] M. Reunanen, P. Wasiak, and D. Botz. Piracy & Social Change | Crack Intros: Piracy, Creativity and Communication. *International Journal of Communication*, 9(0), 2015.

[108] E. Robbins, J. Howe, and A. King. Theory Propagation and Reification. *Science of Computer Programming*, 111(1):3–22, 2015.

[109] E. Robbins, J. M. Howe, and A. King. Theory Propagation and Rational-Trees. In *Principles and Practice of Declarative Programming*, pages 193–204. ACM, 2013.

[110] E. Robbins, A. King, and T. Schrijvers. From MinX to MinC: Semantics-Driven Decompilation of Recursive Datatypes. In *POPL*, pages 191–203. ACM, 2016.

[111] A. Simon and A. King. Taming the Wrapping of Integer Arithmetic. In *SAS*, volume 4634 of *LNCS*, pages 121–136. Springer, 2007.

[112] A. Simon, A. King, and J. M. Howe. The Two Variable Per Inequality Abstract Domain. *HOSC*, 31(1):182–196, 2010. `http://kar.kent.ac.uk/30678`.

[113] O. Strichman, S. Seshia, and R. Bryant. Deciding Separation Formulas with SAT. In *CAV*, volume 2404 of *LNCS*, pages 209–222. Springer, 2002.

[114] M. Sutton, A. Greene, and P. Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley, 2007.

[115] C. Tinelli. A DPLL-based Calculus for Ground Satisfiability Modulo Theories. In *European Conference on Logics in Artificial Intelligence*, volume 2424 of *Lecture Notes in Artificial Intelligence*, pages 308–319. Springer, 2002.

[116] C. Treude and M. Salois. An exploratory study of software reverse engineering in a security context. In *Working Conference on Reverse Engineering*, pages 184–188. IEEE, 2011.

[117] A. Tridgell. How Samba Was Written, 2003. `http://samba.org/ftp/tridge/misc/french_cafe.txt`.

[118] K. Troshina, Y. Derevenets, and A. Chernov. Reconstruction of composite types for Decompilation. In *Working Conference on Source Code Analysis and Manipulation*, pages 179–188. IEEE Computer Society, 2010.

[119] M. J. Van Emmerik. *Static Single Assignment for Decompilation*. PhD thesis, University of Queensland, 2007. `http://espace.library.uq.edu.au/view/UQ:158682`.

[120] W. Wang. Ucc, 2014. `http://ucc.sourceforge.net/`.

[121] S. Warshall. A Theorem on Boolean Matrices. *Journal of the ACM*, 9(1):11–12, 1962.

[122] M. A. Weiss. *Data Structures and Algorithm Analysis in C*. Addison-Wesley, 1996.

[123] R. Wilhelm, M. Sagiv, and T. Reps. Shape analysis. In *Compiler Construction*, volume 1781 of *LNCS*, pages 1–17. Springer, 2000.