MODELING DYNAMIC ALLOCATIONS AND DEALLOCATIONS OF LOCAL MEMORY FOR TRANSLATION VALIDATION

ABHISHEK ROSE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY DELHI
FEBRUARY 2025

MODELING DYNAMIC ALLOCATIONS AND DEALLOCATIONS OF LOCAL MEMORY FOR TRANSLATION VALIDATION

by

Abhishek Rose

Department of Computer Science and Engineering

Submitted

in fulfillment of the requirements of the degree of

Doctor of Philosophy

to the

Indian Institute of Technology Delhi

FEBRUARY 2025

Certificate

This is to certify that the thesis titled **Modeling Dynamic Allocations and Deallocations of Local Memory for Translation Validation** being submitted by **Mr. Abhishek Rose** for the award of **Doctor of Philosophy** in **Computer Science and Engineering** is a record of bona fide work carried out by him under my guidance and supervision at the Department of Computer Science and Engineering, Indian Institute of Technology Delhi. The work presented in this thesis has not been submitted elsewhere, either in part or full, for the award of any other degree or diploma.

Prof. Sorav Bansal
Microsoft Chair Professor
Department of Computer Science and Engineering
Indian Institute of Technology Delhi
New Delhi 110016

Acknowledgements

I would like to thank all who contributed to this dissertation.

My sincere gratitude goes to my supervisor, Prof. Prof. Sorav Bansal, for expert guidance, insightful feedback, and unwavering support.

I also appreciate the valuable input of my committee members, Prof. Sanjiva Prasad, Prof. Rahul Purandare, and Prof. Subodh Sharma, whose critiques greatly improved this work.

I am grateful to my fellow students, Shubhani, Indrajit, Vaibhav, and Pratik, for their collaboration and stimulating discussions.

I also acknowledge the staff of the Department of Computer Science and Engineering for their administrative and technical support.

Finally, I thank the reviewers for their thorough evaluations and constructive comments, which helped refine the final dissertation.

Abhishek Rose

Abstract

End-to-End Translation Validation is the problem of verifying the executable code generated by a compiler against the corresponding input source code for a single compilation. This becomes particularly hard in the presence of dynamically-allocated local memory where addresses of local memory may be observed by the program. In the context of validating the translation of a C procedure to executable code, a validator needs to tackle constant-length local arrays, address-taken local variables, address-taken formal parameters, variable-length local arrays, procedure-call arguments (including variadic arguments), and the `alloca()` operator.

We make the following contributions in our work:

1. A formalization of the execution semantics for an unoptimized intermediate representation (IR) of a C program and its compiled 32-bit x86 assembly in the presence of dynamically (de)allocated local memory. This includes modeling of the various dynamic allocation constructs in C, such as address-taken local variables, constant- and variable-length local arrays, address-taken formal parameters, procedure-call arguments (including variadic arguments), and the `alloca()` operator.

2. A notion of correct translation from the IR to the assembly through a refinement definition. The definition incorporates the concept of undefined behavior (UB) within the IR program, originally translated from C, where refinement is permitted to hold

trivially.

3. An algorithm that converts the correct translation check to first-order logic queries over bitvectors, arrays, and uninterpreted functions that can be discharged using off-the-shelf SMT solvers. The algorithm is capable of operating in both blackbox and whitebox modes, with the blackbox mode enabling its usage with third-party compilers that may not employ a specific allocation strategy, such as preallocation. In particular, we are perhaps the first to enable support for dynamic stack allocation strategy for procedure-call arguments used by almost all production compilers (e.g., GCC, Clang/LLVM).

4. A prototype implementation of the algorithm and its comprehensive evaluation on a set of diverse benchmarks, including both micro-benchmarks and a real-world `bzip2` program. Our prototype performs blackbox translation validation of C procedures with up to 100+ SLOC against their corresponding assembly implementations with up to 140+ instructions generated by an optimizing production compilers (such as GCC, Clang/LLVM, ICC) with complex loop and vectorizing transformations.

Contents

List of Figures

Chapter 1

Introduction

Safety and mission-critical software systems such as those found in medical equipment and nuclear reactors require strong correctness guarantees. Techniques in formal methods can provide mathematically-sound guarantees on the behavior of a software artifact. Absence of null-pointer dereference, buffer and integer overflow are examples of behavior-bounding guarantees ensured by formal techniques. Such techniques are typically applied on the source code, e.g., C source code, which is different from the machine executable code, e.g., x86 assembly code, that is executed on physical hardware. The translation of the source code into machine executable code is performed by a compiler in an act of compilation. A compilation (is expected to) preserves the *semantics* of the input source code in the output executable code. This *semantics preservation* or *correctness* property of compilation enables transfer of a behavior bounding guarantee, assured by formal methods techniques applied on the source code, from the source code to the executable code. A bug in the compilation process, however, can impede this transfer thereby giving a false sense of assurance. For example, a miscompilation may introduce an error that was absent in the original source program. Thus, application of formal methods on the source code is ineffective until the compiler or, more specifically, a desired instance of compilation itself is validated to be correct.

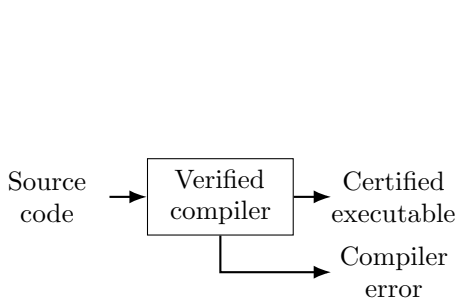
Besides upholding the semantic preservation or correctness property, compilers are also expected to perform optimizing transformations (optimizations) such that the compiled program exhibits certain performance characteristics, e.g., fast execution, power and memory efficiency, etc. An overwhelming majority of the complexity of modern optimizing compilers lies in this optimizing phase [16]. This additional responsibility of

optimization makes ensuring correctness harder. Indeed, correctness bugs in optimizing compilers are not uncommon [52, 46, 27, 45].

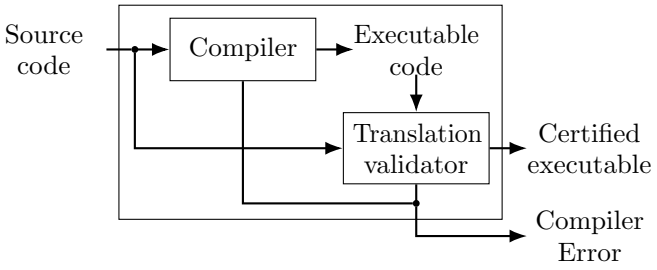
A way to validate a compilation, to ensure *semantics preservation*, is to formally verify the compiler itself. Such a *verified compiler* will only produce an output executable code, a *certified executable*, if it can ensure, through a formal argument, that the semantics of the input source code are preserved in the output executable code. However, writing a verified compiler is a formidable task. Modern non-verified optimizing compilers such as GCC [15] and Clang/LLVM[10] are multi-million lines of source code projects with thousands of changes being made with every release. Verifying the semantic preservation property on them or rewriting them to be easily verifiable while keeping with the pace of development seems impractical. Though there exist verified compilers such as CompCert [29] and CakeML [25], the optimization they perform and the source code language constructs they support are relatively limited in scope, prohibiting their large scale adoption. Another difficulty in development of verified compilers lies in the amount of expertise they require: writing formal proofs using proof assistants and/or using semi-automated verification tools arguably requires higher expertise than required of a typical compiler engineer.

Translation validation (TV for short) [37, 36] is an alternative approach where instead of verifying the entire compiler, an instance of compilation or translation is verified. A translation validation tool, i.e., a translation validator, takes two programs as input: (1) the *source* program passed as input to a translator; and (2) the *target* program emitted as output by the translator. The *target* program is compared against the input *source* program to verify the *semantics preservation* property. The validation, if successful, is accompanied by a proof of correctness that can be independently verified. The generation of an externally verifiable artifact allows the TV tool itself to potentially remain unverified while still being able to produce sound results. In the *end-to-end translation validation* setting, *source* represents the source code of a program written in a high-level language, such as C, and *target* represents the translated machine executable code, such as x86 machine code, obtained after compiling *source*. In contrast, translation validation can also be performed in a "pass-by-pass" manner, where the validator is invoked at the end of a transformation pass in the compiler. In this setting, *source* and *target* denote the programs before and after the transformation, respectively.

A verified compiler can also be obtained using a combination of a non-verified compiler and an *end-to-end* translation validator; fig. 1.1b shows a construction. In such a verified



(a) Block diagram of a verified compiler.



(b) Block diagram of a verified compiler realized using a combination of a non-verified compiler and an end-to-end translation validator.

Figure 1.1: Two ways of obtaining a certified executable from an input source code. The certified executable is guaranteed to have identical semantics as the input source code.

compiler, the end-to-end translation validator is invoked at the end of each compilation by the non-verified compiler to validate the output compilation. If the translation validator is able to successfully validate the compilation, the combined tool produces a certified executable. This separation of compilation and its validation afforded by translation validation makes the combined construction an appealing practical alternative to the verified compiler approach.

A major challenge in the construction of an end-to-end translation validator lies in the difference between the levels of abstractions in the two input programs' representations. In the context of translation from an *unoptimized intermediate representation (IR)* of a C program to an assembly code representation of executable code, which is the focus of this work, this manifests as difference in the abstractions available in the C programming language (equivalently, in its unoptimized IR) and the assembly language.

Prior work on translation validation relates the two programs through construction of a product program that correlates the program points of the constituent programs. While the control-flow constructs could potentially be rather directly mapped to or modeled through a combination of conditional and unconditional jump operations available in the assembly language[57, 6, 54], the translation or modeling of concepts such as objects (their allocation and deallocation), types (that may affect aliasing), and pointer provenance [19] (for determining aliasing) is not immediately clear. For example, instead of a potentially infinite set of pseudo-registers or local variables, assembly has a finite set of machine registers and a stack region delineated by a stackpointer

register. Furthermore, C allows taking (and observing) address of a variable. Translation validation of programs with such constructs requires identification of relations between stack addresses in assembly and addresses of the variables in the C program. This identification and modeling of the relation between an address-taken variable in C and its corresponding implementation somewhere in the assembly stack is not straightforward. The difficulty of this problem has been acknowledged in previous work on end-to-end translation validation with [40] calling it their "largest limitation" and other works assuming that the input programs do not exhibit such programming patterns [8, 17]. We motivate and describe the problem clearly through an example.

1.1 Problem Statement and Motivating Example

The goal of this work is end-to-end validation of a compilation from a C program to an assembly program where the C program may contain dynamic allocation and deallocation of local memory through local variable declaration including C99's variable-length arrays (VLAs), procedure-call arguments including variable-length arguments (variadic arguments), and the `alloca()` operator [2]. We transform the C program into an unoptimized intermediate representation (IR) where the allocation and deallocation of local memory is made explicit through IR instructions, and implementation-defined constructs such as types and their representation have been determinized. This unoptimized IR is obtained using a syntactic translation and does not involve any optimization. We use 32-bit x86 assembly with `cdecl` calling conventions, described in System V Application Binary Interface (ABI) for i386 systems [32], as a concrete assembly program representation. The translation from C to unoptimized IR uses the specifics of 32-bit x86 assembly ABI[1].

Thus, we are interested in showing that a 32-bit x86 assembly program $\mathbb{A}$ is a correct translation of the unoptimized IR $\mathbb{C}$ of a C program where $\mathbb{C}$ may dynamically allocate and deallocate local memory and may contain constructs such as constant-length and variable-length local arrays, address-taken local variables and formal parameters, procedure-call arguments including variable-length (variadic) arguments, and the `alloca()` operator.

[1]For example, the pointer type `void*` is determinized to a 32-bit bitvector representation matching the pointer representation in 32-bit x86.

```
C0:  int read_int() {
C1:    int x;
C2:    scanf("%d", &x);
C3:    return x;
C4:  }
```

(a) C program with an address-taken local variable `x`.

```
I0:  int read_int():
I1:    x = alloc int;
I2:    t1 = @STR; // address of "%d"
I3:    t2 = call scanf(t1, x);
I4:    r = *x;
I5:    dealloc I1;
I6:    ret r;
```

(b) (Abstracted) Unoptimized IR.

```
A0:  read_int:
       ; allocate 36 bytes on stack
A1:    esp -= 36;
A2:    eax = esp + 20;
       ; push procedure arguments on stack
A3:    push eax;
A4:    push @STR; ; address of "%d"
       ; (implicit) arguments to scanf:
       ; mem4[esp], mem4[esp+4]
A5:    call scanf;
A6:    eax = mem4[esp+28];
       ; deallocate 44 bytes from stack
A7:    esp += 44;
       ; return value in register eax
A8:    ret;
```

(c) (Abstracted) 32-bit x86 assembly of fig. 1.2a.

Figure 1.2: C program with an address-taken local and its IR and 32-bit x86 assembly lowerings. `@STR` denotes the address of the format string `"%d"`. $\texttt{mem}_4[a]$ denotes a 4-byte memory access to address a.

1.1.1 An address-taken local example

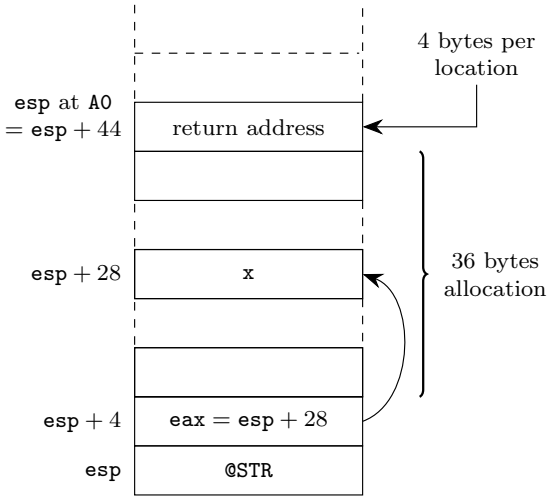
Consider the C program and its unoptimized IR and 32-bit x86 assembly shown in fig. 1.2. The `read_int()` procedure in fig. 1.2a defines a local integer `x`, passes the address of `x` to an external callee `scanf()`, and finally returns the value of `x`. Because the address of variable `x` is taken, through the *address-of* operator `&`, `x` is an *address-taken local*.

Unoptimized IR: The unoptimized IR of `read_int()`, shown in fig. 1.2b, is similar to LLVM IR [48] and has explicit instructions `alloc` and `dealloc` for local memory allocation and deallocation — the latter instruction is not available in LLVM IR. The `alloc` instruction in line `I1` performs allocation of an `int` object and returns the start address of the allocated region in `x`. Both the allocated region and its initial memory contents are non-deterministic but subject to two *well-formedness conditions*:

(1) the freshly allocated region is disjoint from other currently allocated regions;
(2) the start address of the allocated region is aligned by the alignment requirement for the allocated type (e.g., `x` of type `int` should be aligned by 4).

An allocation due to execution of `alloc` is expected to satisfy these well-formedness conditions. Unlike C, the identifier `x` in IR represents the address of variable `x` so that the call to `scanf` in line `I3` accepts `x` as argument (and not `&x` as in fig. 1.2a). The value of the variable `x` is obtained by dereferencing (`*x` in line `I4`). An allocation is uniquely identified by the location or PC of the `alloc` instruction. The `dealloc` instruction deallocates *all* allocated space due to execution(s) of an `alloc` instruction identified by its PC. In fig. 1.2b, `dealloc I1` deallocates the address region allocated by execution of instruction `I1`.

Assembly: Figure 1.2c shows the (abstracted) 32-bit x86 assembly of `read_int()` as generated by GCC. In the compiled assembly, the local `x` is allocated on the assembly stack which is a distinguished region in the machine address space. The lower, flexible end-point of the assembly stack region is identified by the stackpointer register `esp` so that decrementing `esp` *allocates* stack space and incrementing `esp` *deallocates* stack space. The stackpointer decrement in line `A1` allocates 36 bytes on the stack — 4 bytes of this allocated space are reserved for the local `x` (an analysis of the next few instructions reveal that `x` is located at an offset of 20 bytes in this newly allocated space). Unlike its IR counterpart, the x86 `call` instruction does not accept arguments other than the callee label (or address); instead, the arguments are passed implicitly through the stack. The stack addresses used for passing the arguments are specified in the ABI's calling conventions which enable identification of the passed arguments in the callee procedure. The `push` instructions in `A3` and `A4` set up the two arguments to `scanf` on stack — notice that the arguments are pushed in reverse order so that `A3` pushes the second argument, address of `x`, and `A4` pushes the first argument `@STR`, address of the format string. The figure on the right shows the layout of stack just before the call instruction: at the bottom of stack, labeled by `esp`, is `@STR`, above it is `eax` which contains the address `esp+28` of `x` (assigned as `esp+20` in `A2`) and so on. The stackpointer increment in line `A7` deallocates the stack space allocated due to the two `push` instructions and instruction `A1` in a single step (thereby deallocating both arguments to `scanf()` and the local `x`). The return



value of an assembly procedure is passed in register `eax` which is set to value of `x` in `A6` (just before deallocation).

1.1.2 Establishing Correct Translation

Let C denote the IR procedure `read_int()` in fig. 1.2b and A denote the assembly procedure `read_int()` in fig. 1.2c. In this section, we attempt to define a notion of correct translation and present an (loosely formal) argument that A is a correct translation of C. We will illustrate some key problems in validating a translation from C to A through this exercise.

According to the C standard [21], A will be considered a correct translation of C if for each execution of A, there exists an execution of C such that both executions produce identical *observables* — the observables produced by an execution of C constitute its *observable behavior* (para 6 in §5.1.2.3 of [21]). Notice that this definition implies a subset relationship for behaviors which makes sense because C is non-deterministic[2]. Thus, correct translation is defined in terms of *refinement* of observable behaviors — the assembly program refines (for e.g., by determinization) the behavior present in the C program.

The observables produced by an execution of C are (in order):

O1. the occurrence of procedure-call to `scanf()`;
O2. the program state passed to or accessible to the external callee `scanf()`, which includes the arguments `@STR` and `x` (line `I3` in fig. 1.2b);
O3. the return value `*x`.

Because the definition of the `scanf()` procedure is external to the translation unit, no assumptions can be made about its behavior. Therefore, we conservatively consider *both* the occurrence of the call to `scanf()` and the program state accessible to `scanf()` as observables.

To establish correct translation, each execution of A must produce observables such that an execution of C also produces identical observables (recall that C is non-deterministic). Consequently, an execution of A must produce the observables **O1**, **O2**, **O3** (in sequence). It can be observed that A produces **O1**: there is a call to `scanf()` at `A5` in fig. 1.2b

[2]Recall that the `alloc` instruction returns a region with non-deterministic start address and non-deterministic memory contents.

with no observable event[3] before it. Thus, every execution of A at least produces **O1**. Next, we must witness production of observable **O2** in (an execution of) A.

Recall that **O2** observes the program state passed to `scanf()` which includes both the address and the value of variable `x`, both of which are non-deterministic in C — the value of `x` is included because passing the address makes the variable reachable from `scanf()` (through a dereference). To produce an identical observable, (an execution of) A must also *pass* identical program state to `scanf()`. Recall that in A the arguments are passed implicitly through stack so that the number of passed arguments is not identifiable solely from the `call` instruction in A; the calling conventions specify the addresses of the arguments but not their count. This problem is exacerbated by the fact that `scanf()` is a variadic procedure so it may accept a variable number of arguments and even the type signature of `scanf()` in C cannot be used for determining the number of arguments. Even if we assume that only two arguments (as set up in `A3` and `A4` of fig. 1.2c) are passed to `scanf()`, we need to relate the (stack) address `esp+20` (set up in `A3`) with the non-deterministic address `x` in C. While `x` is subject to the well-formedness conditions, there is no guarantee that `esp+20` respects the same conditions so that it is possible for `esp+20` and `x` to not agree on the same value — recall that we want to establish that an execution of C may always produce a value `x` that is equal to the value `esp+20` produced by an execution of A. Moreover, as the stack is shared by both the allocated locals and spilled pseudo-registers, discriminating different stack writes becomes a problem. For example, it needs to be ascertained that the stack writes at lines `A3` and `A4` of fig. 1.2c do not mutate the stack region corresponding to the local `x`[4].

Further, because we are not able to ascertain the number of arguments passed to `scanf()` in A, we must conservatively assume that it may read an arbitrary number of arguments (by reading the stack addresses specified in the calling conventions). The external call to `scanf()` can be soundly (and over-approximately) modeled (in both C and A) as an arbitrary but deterministic mutation of the accessible program state subject to the input and accessible program state. With this modeling, the difference in number of arguments makes the observable **O2** in A different from the observable **O2** in C.

Lastly, the observable **O3** is also related to the modeling of procedure call to `scanf()`:

[3] Observable events include a call to an external procedure and a procedure return.

[4] While in this particular example it is easy to distinguish the writes due to use of the `push` instruction that writes to a freshly allocated region, doing so in presence of stack spills require tracking of addresses belonging to an allocated variable.

the return value of procedure A is obtained by reading machine memory at address `esp+28` immediately after the procedure call (line `A6` in fig. 1.2c); in C the return value is identified as the value `*x`. As both values are mutated arbitrarily by `scanf()`, due to difference in the number of arguments passed to `scanf()`, **O3** in A cannot be related to observable **O3** in C.

To summarize, we identified the following problems in relating the observables (**O2** and **O3**) in an execution of A with an execution C:

- In relating a stack allocation in A with an allocation in C so that a stack region in A can be distinctly identified and related to a non-deterministic but constrained allocated region in C.
- In identifying the behavior of a procedure call in A so that the program state accessible to an assembly callee can be precisely identified.

1.1.3 Subtleties

In this section, we take a closer look at the two programs in figs. 1.2a and 1.2c and consider some subtleties associated with a valid translation.

Assembly program may allocate more memory

Consider the stack allocation of 36 bytes at line `A1` in fig. 1.2c. It is possible for this stack allocation to overflow into other allocated space (e.g., heap) leading to abnormal termination of A's execution. Recall that correct translation requires every observable behavior of A, including abnormal termination, to be an observable behavior of C. We observe that C in fig. 1.2b may similarly terminate due to allocation failure while executing `alloc`. Recall that an `alloc` instruction allocates a region that does not overlap with currently allocated space; if it is not possible to meet this requirement, the execution terminates abnormally. Notice, however, that A allocates more memory than C: 36 bytes in A, compared to 4 bytes in C[5]. Thus, it is possible for a given execution of an assembly program starting at `read_int()` to run out of memory but a similar execution of the unoptimized IR program to terminate normally. A correct translation definition, therefore, must take into consideration abnormal termination of an assembly program due to running out of memory as an admissible observable behavior.

[5]The extra allocation, in this particular example, is to align the stackpointer to a 16 byte boundary for the procedure-call at `A5`

```
C0:  int foo(int n, int m)
C1:  {
       // VLA with n elements
C2:    char buf[n];
C3:    buf[m] = 0;
C4:    return bar(buf, n);
C5:  }
```

(a) C program with VLA

```
A0:  foo:
A1:    push ebp; ebp = esp;
A2:    esp -= 8;
A3:    edx = mem4[ebp+8];   ; argument 'n'
A4:    ecx = mem4[ebp+12]; ; argument 'm'
       ; variable-sized stack allocation
A5:    esp -= 0xFFFFFFF0 & (edx+15);
A6:    mem1[esp+ecx] = 0;
A7:    eax = esp;
A8:    esp -= 8;
A9:    push edx; push eax;
A10:   call bar;
A11:   esp = ebp; pop ebp;
A12:   ret;
```

(b) (Abstracted) 32-bit x86 assembly

Figure 1.3: C program with variable-length array (VLA) and its assembly lowerings. $\texttt{mem}_4[a]$ denotes a 4-byte memory access to address a. An execution where $\texttt{m} \geq \texttt{n}$ triggers undefined behavior in the C program and C semantics do not put restriction on behavior of the translated assembly program in such an execution.

We wish to point out this peculiarity arises due to the finite machine state available to an assembly program: while an unoptimized IR program may utilize an infinite set of pseudo-registers, the assembly has a finite set of registers and has to spill additional state into the stack. A notable consequence of having such a definition is that it is easy to construct a *sound* but vacuous translation for every C program that always terminates abnormally due to out-of-memory.

C program may trigger Undefined Behavior (UB)

C has undefined behavior semantics [51]. Consider the C program and its compiled assembly in fig. 1.3. The `foo()` procedure in fig. 1.3a accepts two integers `n` and `m` as parameters, allocates a variable-length array (VLA) `buf` with `n` characters and calls an external procedure `bar`, passing `buf` as an argument. Figure 1.3b shows the compiled assembly of the `foo` procedure. The assembly procedure receives the two parameters `n` and `m` on stack and reads them into registers `edx` and `ecx` in `A3` and `A4` respectively. The VLA is allocated on stack via a stackpointer decrement in `A5` — the extra arithmetic ensures that the resulting stackpointer value remains aligned by 16. The write to stack address `esp+ecx` in `A6` corresponds to the zero initialization of $\texttt{m}^{th}$ element of the VLA (`C3` in fig. 1.3a) in the assembly program.

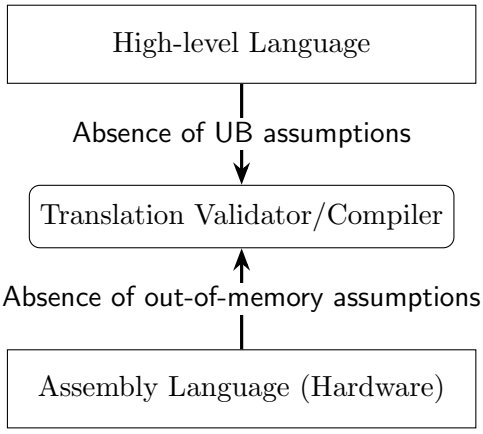


Figure 1.4: Conceptual representation of the flow of information from high-level source program and the low-level assembly program to the compiler/translation validator.

Let C and A denote the C and assembly `foo()` procedures in figs. 1.3a and 1.3b respectively. Consider an execution of A where $\mathtt{m} \geq \mathtt{n}$. In such an execution, the write at `A6` may potentially overstep stack bounds, accessing a memory location that is not mapped and thereby causing termination. Another possibility is that the write address remains within stack bounds, overwriting some address in the caller stack of the `foo()`, breaking some program invariant and eventually leading to some logic error. An execution of C under the same input ($\mathtt{m} \geq \mathtt{n}$) will trigger an out-of-bounds write at `C3`. An out-of-bounds access is undefined behavior (UB) in C and the C semantics do not put any restriction on valid translation for such executions. As a result, any behavior, including termination or a logic error, of the translated assembly program is considered a valid translation for a construct that causes undefined behavior in C. A correct translation definition must accommodate handling of undefined behavior in order to cater to the various possibilities available to the compiler.

Figure 1.4 summarizes the flow of information due to semantics of C and A. The undefined behavior semantics in C enable both the compiler and the translation validator to make certain "absence of UB" assumptions about the error-free execution of C — the information flowing downwards from "High-level Language" in fig. 1.4. Similarly, both the compiler and the translation validator may assume availability of adequate stack space; shown as "absence of out-of-memory assumptions" flowing from "Assembly Language (Hardware)" in fig. 1.4. A compiler performs a translation under these conditions and, therefore, the translation validator must assume them while validating the transformations produced by the compiler.

1.1.4 A sketch of proposed solution

Prior TV efforts identify a lockstep correlation between (potentially unrolled) iterations of loops in the two procedures to show equivalence [8]. These correlations can be represented through a *product program* that executes procedures C and A in lockstep, using a careful choice of program path correlations, to keep the machine states of both procedures related at the ends of correlated paths [54, 17].

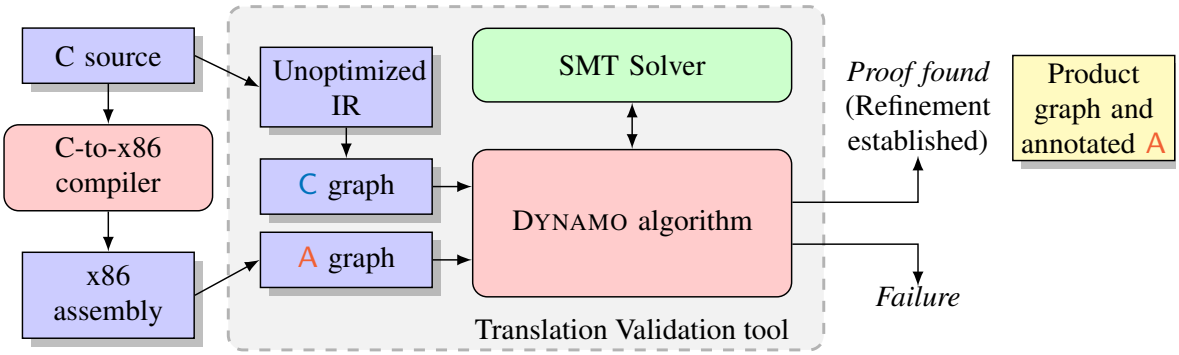


Figure 1.5: High-level components and flow of our translation validation approach.

Our TV algorithm, called DYNAMO, additionally attempts to identify a lockstep correlation between the *dynamic (de)allocation events* and *procedure-call events* performed in both procedures, i.e., we require the order and values of these execution events to be identical in both procedures. To identify a lockstep correlation, our algorithm annotates assembly procedure A with two kinds of annotations:

- (de)allocation instructions for identifying (de)allocation of a region in assembly;
- procedure-call annotations for identifying procedure-call arguments and memory regions accessible to an assembly callee.

Our key insight is to define a *refinement relation* between C and A through the existence of an annotated A. We carefully define the semantics for these annotations so that the refinement relation holds only if an annotation correctly captures the (de)allocation and procedure-call events in A. We also generalize the definition of a product program so it can be used to witness refinement in the presence of non-determinism due to addresses of dynamically-allocated local memory, undefined behavior (UB), and stack overflow in A. For example, our product program encodes the UB semantics that allow anything to happen in assembly (including out-of-bound stack access) if UB is triggered in C.

Our proof obligations are expressed in first-order SMT (Satisfiability Modulo Theories) logic over bitvectors, arrays, and uninterpreted functions that can be discharged by an off-the-shelf SMT solver. On successful execution, our algorithm produces two artifacts: (1) a product program encoding lockstep execution of unoptimized IR C and assembly procedure A in the form of a product graph and (2) an annotated version of A that satisfies the refinement relation with C. Figure 1.5 shows the high-level components of our translation validation approach.

1.2 Prior Work

Translation validation (TV) was proposed by Pnueli, Siegel, and Singerman [37] for validating translation of programs written in SIGNAL (a synchronous multi-clock data-flow language) to C code. This was followed shortly by Necula [36] who validated GCC optimization passes. Since then there has been much development in this area [54, 41, 49, 44, 56, 55, 35, 23]. TV has been successfully applied for finding bugs in real-world compilers [31, 34, 22], verifying complex loop unrolling and vectorizing transformations [8, 17], verification of individual passes inside a verified compiler [50], and, perhaps most ambitiously, in translation validation of a verified OS kernel [40]. However, no prior work tackled the problem of modeling dynamic local memory allocation and deallocation in the context of IR-to-assembly translation.

We give a brief survey of the prior TV approaches in rest of the section. We discuss their capability with respect to handling of dynamic local memory (de)allocations and highlight differences from an end-to-end IR-to-assembly TV such as ours.

1.2.1 IR-to-IR Translation Validation

An IR-to-IR translation validator (TV) validates transformation of a program represented in an intermediate language (IR)[6] into a program represented using (usually) same or (rarely) different IR [36, 31, 49, 44, 56, 55, 34, 35, 23]. Unlike IR-to-assembly, modeling dynamic local memory allocations is significantly simpler for IR-to-IR TV. For example, (pseudo)register-allocation of local variables can be tackled by identifying relational invariants that equate the value contained in a local variable's memory region

[6]In our discussion, we use intermediate language (IR) to denote a language suitable for use in target-independent phases of a compiler transformation pipeline. Typical characteristics of such a language include infinite pseudo-registers, types, and abstract mechanisms for representing allocation and deallocation of memory.

(in the original program) with the value in the corresponding pseudo-register (in the transformed program) [22, 23]. If the address of a local variable is observable by the C program (e.g., for an address-taken local variable), we need to additionally relate the variable addresses across both programs. These address correlations can be achieved by first correlating the corresponding allocation statements in both programs (e.g., through their names as identifiers) and then assuming that their return values are equal. Provenance-based syntactic pointer analyses, that show separation between distinct variables [3, 43], thus suffice for translation validation across IR-to-IR transformations.

Recent work on bounded TV[7] by Lee et al. [28] presents an efficient SMT encoding for modeling memory where memory is segmented into *separate blocks* and each allocation is identified distinctly using a block-ID. As the number of allocations is bounded, a pointer is represented as a combination of a block-ID and an offset into a block. While this suffices for the bounded TV setting, our problem setting with potentially unbounded allocations (e.g., due to use of `alloca()` in a loop) requires a more general representation of a dynamically-allocated variable and a more general SMT encoding.

1.2.2 IR-to-Assembly and Assembly-to-Assembly Translation Validation

Prior work on IR-to-assembly and assembly-to-assembly TV [40, 41, 39, 13, 8, 17, 42] assumes that local variables are either absent or their addresses are not observed in the program and so they are removed through (pseudo)register-allocation. This assumption simplifies the validation effort as the required relations are between (pseudo)registers and spilled pseudo-registers on stack — the stack addresses of these spilled pseudo-registers are not relevant and do not need to be related.

A problem that arises uniquely in case of IR-to-assembly TV is disambiguation of a stack memory access from a *non-stack* (or heap) memory access. Sewell [39] employs a trusted static analysis over the assembly program to label each memory access as either aliasing with stack or not aliasing with stack. This distinction enables identification of the stack as separate from rest of the memory array such that any write to stack does not affect the rest of the memory state and vice versa. A similar approach of using two

[7] In bounded TV, the behavior of the input programs is bounded. For example, the programs are made acyclic by unrolling all loops by a finite unroll factor. This limits both execution and memory consumption of the programs. As a result, the validation guarantee is limited to executions that fall within the specified bounds.

memory arrays, one for stack and another for non-stack, is used in [42].

Other cases of address-taken parameters including the variadic parameter, where the correlation of the stack address representing the currently-iterated parameter in the variable-argument list is required, are also assumed to be absent in these prior works. Our work targets both constructs.

1.2.3 Verified Compilation

Prior work on verified compilation, embodied in CompCert [29], validates its own transformation passes from IR to assembly, and supports both address-taken local variables and variadic parameters. However, CompCert sidesteps the task of having to model dynamic allocations by ensuring that the generated assembly code *preallocates* the space for all local variables and procedure-call arguments at the beginning of a procedure's body. Because preallocation is not possible if the size of an allocation is not known at compile time, CompCert does not support variable-sized local variables or `alloca()`. Moreover, preallocation is prone to stack space wastage. In contrast to a verified compiler, TV needs to validate the compilation of a third-party compiler, and thus needs to support an arbitrary allocation strategy. In particular, almost all production compilers use dynamic allocation strategy for procedure call arguments. An end-to-end TV must support dynamic (de)allocation to be a useful alternative.

1.3 Contributions

The central focus of this thesis is to investigate the applicability of translation validation as an alternative to verified compilation. Recent work [40, 17] has demonstrated that end-to-end translation validation is a feasible option for validating third-party compilations. However, as discussed in previous sections, the lack of support for handling (dynamic) local allocation restricts the applicability of the approach to the wide array of programs found in real-world. This work aims to rectify this limitation. We offer the following high-level contributions in this space.

- A formalization of the execution semantics for the unoptimized IR translation of a C program and its compiled 32-bit x86 assembly in the presence of dynamically (de)allocated local memory. This includes modeling of the dynamic local allocation constructs in C and assembly, such as constant and variable-length local arrays, and

`alloca()` operator in C and stack-allocated procedure-call arguments (including variadic arguments) in assembly. Our modeling is sound in the sense that we do not make any simplifying assumptions and model various constructs precisely. For example, we model C integers as bitvectors with wrap-around semantics (instead of mathematical integers) and the assembly stack is modeled as a part of the byte-addressable memory and not as a separate entity as done in prior work [40].

- A notion of correct translation from unoptimized IR to assembly through a refinement definition. The definition incorporates the concept of Undefined Behavior (UB) within the IR program (originally translated from C) where refinement is permitted to hold trivially when the IR program triggers UB. Similarly, the definition assumes the absence of stack overflow in assembly — the OS (or runtime environment) is expected to ensure the availability of sufficient stack space, and a translation validation is predicated on the availability of this space. A key limitation of this refinement definition is its failure to support interprocedural transformations (e.g., inlining, tail-call elimination, etc.).

- An algorithm that converts the correct translation check to first-order logic queries over bitvectors, arrays, and uninterpreted functions (AUFBV in SMT) through the automatic construction of a "product graph". These first-order logic queries are discharged using off-the-shelf SMT solvers. The algorithm is capable of operating in both *blackbox* and *whitebox* modes, with the blackbox mode enabling its usage with third-party compilers with arbitrary allocation strategies (with some limitations described later). This is in contrast to the verified compiler CompCert [29] that only employs the preallocation strategy for local allocations. In particular, we are perhaps the first effort to enable support for dynamic stack allocation strategy for procedure-call arguments used by almost all production compilers (e.g., GCC, Clang/LLVM, ICC).

- A prototype implementation of the algorithm and its comprehensive evaluation on a set of diverse benchmarks, including both micro-benchmarks and a real-world `bzip2` program. Our prototype is capable of performing blackbox translation validation of C procedures with up to 100+ SLOC against their corresponding assembly implementations with up to 140+ instructions generated by an optimizing production compilers (such as GCC, Clang/LLVM, ICC) with complex loop and vectorizing transformations.

1.4 Outline

The remainder of this thesis is divided into chapters as follows. We also provide a summary in each case.

- In chapter 2, we formalize the execution semantics of the unoptimized IR $\mathbb{C}$ obtained from translation of a C program and its compiled 32-bit x86 assembly $\mathbb{A}$, and present the definition of our refinement relation for encoding the notion of correct translation.

  The execution semantics of a procedure C of $\mathbb{C}$ and corresponding assembly procedure A of $\mathbb{A}$ are described in terms of translation of C and A to a labeled transition graph representation. The refinement relation is defined through existence of an annotation in A such that its observable behaviors matches with C.

- In chapter 3, we describe a cross-product or product program X between the annotated assembly A and unoptimized IR C. We impose a set of requirements on X so that the construction of X implies the existence of a refinement relation between C and A.

  We further describe *callers's virtual smallest* and *safety relaxed* semantics of A and C that help in realizing an efficient automatic construction algorithm for the product program between the modified A and C. We show that it is possible to construct a product program between the original A and C from a product program between A and C with modified semantics.

- In chapter 4, we describe our algorithm Dynamo for the simultaneous automatic construction of the product program X and annotations for A.

  Dynamo builds both the annotated A and product program X incrementally. This incremental construction enables use of counterexample-guided heuristics described in prior works.

- In chapter 5, we describe the SMT encoding for the proof obligations generated by Dynamo.

  We describe an allocation state array based encoding, which uses quantifiers over SMT arrays, and a faster interval encoding, which uses SMT bitvectors.

- In chapter 6, we present an implementation of Dynamo and its comprehensive evaluation over a set of varied benchmarks.

  Our benchmarks include C procedures sourced from various sources including prior work on translation validation adapted suitably for our setting and the `bzip2` program

from SPEC2000 [20] with up to 100+ SLOC. The compiled 32-bit x86 assemblies for these C procedures include vectorized code with up to 140+ instructions.

- In chapter 7, we give concluding remarks and some directions for future work.

Chapter 2

Execution Semantics and Notion of Correct Translation

Our objective is to show that an x86 executable $\mathbb{A}$ is a correct translation of a C program $\mathbb{C}$. Towards it, we first formalize the execution semantics of $\mathbb{C}$ and $\mathbb{A}$ in terms of translation to a transition graph representation. We then define correct translation in terms of refinement of behavior from $\mathbb{C}$ to $\mathbb{A}$. Our novel contribution here is showing refinement through existence of an annotation of $\mathbb{A}$ such that the annotated $\mathbb{A}$ refines $\mathbb{C}$.

This chapter is organized as follows: in section 2.1, we present our unoptimized IR and assembly representations; in section 2.2, we introduce our transition graph representation and in section 2.3, we formalize the execution semantics of IR and assembly through translation to transition graph. We present and discuss our refinement definition in sections 2.4 to 2.7.

2.1 Intermediate Source and Assembly Representations

As our very first step, we translate the C program $\mathbb{C}$ to an unoptimized intermediate representation (IR), which we also refer to as $\mathbb{C}$ — this translation is syntactic and does not involve any analysis or optimization over the C program $\mathbb{C}$. Similarly, we disassemble the x86 executable to obtain assembly program $\mathbb{A}$. In the section, we

```
     int printf(const char*, ...);

C0:  int fib(int n, int m) {
C1:    int v[n+2];
C2:    v[0]=0; v[1]=1;
C3:    for(int i=2; i<=m; ++i)
C4:      v[i]=v[i-1]+v[i-2];
C5:    printf("fib(%d) = %d", m, v[m]);
C6:    return v[m];
C7:  }
```

(a) C program with VLA.

```
I0:  int fib(int* n, int* m):
I1:    i=alloc 1, int, 4;
I2:    v=alloc *n+2, int, 4;
I3:    v[0]=0; v[1]=1;
I4:    *i=2;
I5:    if(*i >s *m) goto I9;
I6:      v[*i]=v[*i-1]+v[*i-2];
I7:      ++(*i);
I8:      goto I5;
I9:    pI9=alloc 1, char*, 4;
I10:   pI10=alloc 1, struct{int;int;}, 4;
I11:   *pI9=__S__; /* __S__ is the
                      address of the
                      format string */
I12:   *pI10=*m; *(pI10 + 4)=v[*m];
I13:   t=call int printf(pI9, pI10);
I14:   dealloc I10;
I15:   dealloc I9;
I16:   r=v[*m];
I17:   dealloc I2;
I18:   dealloc I1;
I19:   ret r;
```

(b) (Abstracted) Unoptimized IR.

```
A0:  fib:
A1:    push ebp; ebp = esp;
A2:    push {edi, esi, ebx};
A3:    esp -= 12;
A3¹:   vI1 = alloc_v 4, 4, I1;
A4:    eax = mem4[ebp+8]; ebx = mem4[ebp+12];
A5:    esp -= 0xFFFFFFF0 & (4*(eax+2)+15));
A5¹:   alloc_s esp, 4*(eax+2), 4, I2;
A6:    esi = ((esp+3)>>2)*4;
A7:    mem4[esi] = 0; mem4[esi+4] = 1;
A8:    if(ebx ≤s 1) jmp A15;
A9:    edi = 0; edx = 1; eax = 2;
A10:     ecx = edx+edi;
A11:     edi = edx; edx = ecx;
A12:     mem4[esi+4*eax] = ecx;
A13:     ++eax;
A14:     if(eax ≤s ebx) jmp A10;
A15:   edi = mem4[esi+4*ebx];
A16:   esp -= 4;
A17:   push {edi, ebx, __S__};
A17¹:  alloc_s esp,  4, 4, I9;
A17²:  alloc_s esp+4, 8, 4, I10;
A18:   call int printf
           (<char*> esp,
            <struct{int; int;}> esp+4)
           {hp, cl, I9, I10};
A18¹:  dealloc_s I10;
A18²:  dealloc_s I9;
A19:   eax = edi;
A19¹:  dealloc_s I2;
A19²:  dealloc_v I1;
A20:   esp = ebp-12;
A21:   pop {ebx, esi, edi, ebp};
A22:   ret;
```

(c) (Abstracted) 32-bit x86 assembly code.

Figure 2.1: C program with variable-length array (VLA) and its lowerings to unoptimized IR and assembly. Subscript '$_s$' denotes signed comparison. Red font (parts of) instructions in assembly are added by our algorithm.

describe the unoptimized intermediate source and assembly representations and the programming constructs we support through them. We discuss our logical model in the context of compilation to 32-bit x86 for the relative simplicity of the calling conventions in 32-bit mode.

We use the example shown in fig. 2.1 in our exposition. Figure 2.1 includes a C program, its unoptimized IR, and compiler generated assembly. The `fib` procedure in fig. 2.1a accepts two integers `n` and `m`, allocates a variable-length array (VLA) `v` of `n+2` elements, computes the first `m+1` Fibonacci numbers in `v`, calls `printf()`, and returns the m^{th} Fibonacci number. Notice that for a UB-free execution, both `n` and `m` must be non-negative and `m` must be less than `(n+2)`. The memory for local variables (`v` and `i`) and procedure-call arguments (for the call to `printf()`) is allocated dynamically through the `alloc` instruction in the IR program (fig. 2.1b). In the assembly program (fig. 2.1c), memory is allocated through instructions that manipulate the stackpointer register `esp`. We will continue to refer figs. 2.1b and 2.1c in remaining text.

2.1.1 Unoptimized IR

The unoptimized IR we use in our representation of the C program is mostly a subset of LLVM[48] — it supports all the primitive types (integer, float, code or PC labels) and the derived types (pointer, array, struct, procedure) of LLVM. Being unoptimized, our IR does not need to support LLVM's fine-grained undefined behavior semantics enabled by `undef` and `poison` values, it instead treats all error conditions identically as undefined behavior (UB). Syntactic conversion of C to LLVM IR entails the usual conversion of types and operators. A global variable name g or a parameter name y appearing in a C procedure body is translated to the variable's start address in IR, denoted $\texttt{lb.}g$ and $\texttt{lb.}y$ respectively[1]. A local variable declaration or an invocation of the `alloca()` operator [2] is converted to LLVM's *alloca*[1] instruction, and to distinguish the two, we henceforth refer to the latter as the "`alloc`" instruction. Unlike LLVM, our IR also supports a `dealloc` instruction that deallocates a variable at the end of its scope — we describe the semantics of both `alloc` and `dealloc` in section 2.1.3. Henceforth, we refer to our IR as $\texttt{LLVM}_d$, short for LLVM + `dealloc`.

We use a modified Clang[10], the de facto frontend for C in LLVM project, for our translation from C to $\texttt{LLVM}_d$. We use LLVM's `stacksave` and `stackrestore` intrinsics generated by Clang to introduce an explicit `dealloc` instruction for each `alloc` instruction that corresponds to a variable allocation. These intrinsics are inserted by Clang roughly at scope boundaries and we use them as proxy for scope while inserting `dealloc` instructions. The allocations due to `alloca()` operator are deallocated at the

[1]As we will also see later in section 2.2.3, $\texttt{lb.}v$ denotes the *lower bound* of the memory addresses occupied by variable with name v.

end of the procedure and do not require scope tracking.

In fig. 2.1b, explicit `alloc` and `dealloc` instructions are inserted for locals `v` and `i` in fig. 2.1a. The `dealloc` instructions for these `alloc` instructions are inserted just before the end of procedure at `I7` and `I8`.

Translation from C to $\texttt{LLVM}_d$ for Procedure Definitions and Calls

Like LLVM, a procedure definition in $\texttt{LLVM}_d$ can only return a scalar value — aggregate return value (of `struct` type) is passed in memory. Unlike LLVM, where a procedure takes parameters by value, $\texttt{LLVM}_d$ takes all parameters through pointers of corresponding types, e.g., both `n` and `m` are passed through pointers of type `int` in fig. 2.1b. This makes the translation of a procedure call from C to $\texttt{LLVM}_d$ slightly more verbose, as explicit instructions to (de)allocate memory for the arguments are required at the callsite. The $\texttt{LLVM}_d$ `call` instruction takes the pointers returned by these allocations as operands.

Figure 2.2 shows the pseudo-code for translation of a C procedure-call expression to $\texttt{LLVM}_d$. The procedure-call expression is represented by $\rho(e_1, e_2, \ldots, e_m)$ (shown at the top in fig. 2.2) where ρ is either the procedure name or a pointer to a procedure. The type of ρ is $(\tau_1, \ldots, \tau_n) \rightarrow \gamma$ where $\tau_1, \ldots, \tau_n$ represents the parameters' types and γ represents the return type. A well-formed expression should have $m \geq n$. A code fragment with a shaded background represents a *templatized* IR instruction with template slots (to be filled at runtime) marked by $\langle\!|\,|\!\rangle$, e.g., p_r `:= alloc 1,` $\langle\!|\gamma|\!\rangle$`,` $\langle\!|$`ALIGNOF(`γ`)`$|\!\rangle$`;` represents an IR instruction where $\langle\!|\gamma|\!\rangle$ and $\langle\!|$`ALIGNOF(`γ`)`$|\!\rangle$, are instantiated for a concrete type γ — the instruction corresponds to allocation of an element of type $\langle\!|\gamma|\!\rangle$ and alignment $\langle\!|$`ALIGNOF(`γ`)`$|\!\rangle$.

$\texttt{LLVM}_d$ instructions `alloc`, `store`, and `call` have similar syntactical structure as their LLVM counterparts: `alloc` takes the number of elements to be allocated, the $\texttt{LLVM}_d$ type, and required alignment respectively as parameters (see section 2.1.3 for semantics); `store` accepts the type of store target, its alignment, the target value, and the target pointer respectively as its parameters; and the return type-parametric `call` instruction takes the callee label or address and the call arguments as parameters and returns a value if the return type is non-`void`.

$\texttt{GEN}_\tau(e)$ returns the $\texttt{LLVM}_d$ variable holding value of expression e after casting it to type τ; `promoted_type(`e`)` returns the promoted type of C expression e obtained after application of *default argument promotion* rules (see §6.5.2.2 of [21]); `mk_struct_x86_cc(...)`

$\rho(e_1, e_2, \ldots, e_m)$ $\quad$ $(\tau_1, \ldots, \tau_n) \rightarrow \gamma$ is the type of $\rho, m \geq n$

```
argsP := [];      // list of argument pointers
if is_aggregate_type(γ) {
    EMIT(p_r := alloc 1, ⟪γ⟫, ⟪ALIGNOF(γ)⟫;)
    argsP := argsP · p_r;      // add pointer to allocated region as first argument
}
for i in 1...n {      // non-variadic arguments
    EMIT(p_i := alloc 1, ⟪τ_i⟫, ⟪ALIGNOF(τ_i)⟫;)
    EMIT(store ⟪τ_i⟫, ⟪ALIGNOF(τ_i)⟫, ⟪GEN_τi(e_i)⟫, p_i;)
    argsP := argsP · p_i;
}
if m > n {
    κ_1, ..., κ_i, ... κ_m−n := promoted_type(e_n+1), ..., promoted_type(e_m);
    η := mk_struct_x86_cc(κ_1, ..., κ_i, ..., κ_m−n);    // x86 calling conventions compatible type
    EMIT(pvar := alloc 1, ⟪η⟫, ⟪ALIGNOF(η)⟫;)
    argsP := argsP · pvar;
    EMIT(p_v := pvar;)
    for i in 1...(m − n) {
        EMIT(store ⟪κ_i⟫, ⟪ALIGNOF(κ_i)⟫, ⟪GEN_κi(e_i)⟫, p_v;)
        EMIT(p_v := p_v + ⟪OFFSETOF(η, i)⟫;)
    }
}
if γ = void {
    EMIT(call void ⟪ρ⟫(⟪argsP⟫);)
} else if is_aggregate_type(γ) {
    EMIT(call ⟪γ⟫ ⟪ρ⟫(⟪argsP⟫);)
    EMIT(result := ⟪AGG2REG(p_r)⟫;)    // distribute the populated aggregate into IR variables
} else {
    EMIT(result := call ⟪γ⟫ ⟪ρ⟫(⟪argsP⟫);)
}
// deallocate the allocated arguments
for a in reverse(argsP) {
    EMIT(dealloc ⟪a⟫;)
}
```

Figure 2.2: Pseudo-code for translation of a C procedure-call expression to $\texttt{LLVM}_d$ instructions. `alloc`, `dealloc`, `call`, `va_start_ptr`, etc. are $\texttt{LLVM}_d$ instructions.

returns a C '`struct`' type whose member fields' alignment matches the calling conventions' requirements of parameters for 32-bit x86; $\texttt{OFFSETOF}(\eta, i)$ returns the offset (in bytes) of i^{th} member field in `struct` type η. $\texttt{AGG2REG}(p)$ returns the list of values in aggregate pointed to by p.

For a procedure-call expression $\rho(e_1, e_2, \ldots, e_m)$ with parameter types $\tau_1, \ldots, \tau_n$, the translation proceeds as follows. For an aggregate return type ρ, an allocation for the return value is performed through `alloc` and the resulting pointer is saved as the first argument to the callee. For each non-variadic argument e_i $(1 \leq i \leq n)$, the algorithm performs allocation according to type τ_i (of i^{th} parameter) through `alloc` and store e_i into the allocated region (through `store`). A single `struct` with layout respecting the calling conventions requirement is allocated for the variable-argument list and each variadic argument e_j $(n+1 \leq j \leq m)$ is stored at appropriate offset inside the allocated region. The `call` instruction is passed the pointers to allocated regions as arguments. In the epilogue, each allocated region is deallocated through a `dealloc` instruction in reverse order of allocation.

An example of this translation is shown in fig. 2.1, where the call to `printf` at `C5` in fig. 2.1a translates to instructions `I9` to `I15` in fig. 2.1b. The $\texttt{LLVM}_d$ program performs two allocations in `I9` and `I10`, one for the format string (`"fib(%d) = %d"`) and another for the variable argument list (`m, v[m]`); the latter is represented as a single object of type "`struct {int;int;}`" containing two `int`s. The `call` instruction in `I13` takes the pointers returned by the two allocations as arguments and stores the (unused) return value in pseudo-register `t`.

The memory allocation for procedure-call arguments follows the 32-bit x86 calling conventions [32], where stack space is used for passing arguments, making it a suitable semantic choice for this TV setting.

Translation of C's variadic macros and `va_list` type

$$\frac{\texttt{va\_start}(ap, last)}{\begin{array}{l} a \coloneqq \texttt{va\_start\_ptr} \\ \texttt{store void*, 4, } a\texttt{, } \langle\!|ap|\!\rangle \end{array}} \qquad \frac{\texttt{va\_end}(ap)}{\texttt{store void*, 4, 0, } \langle\!|ap|\!\rangle}$$

$$\frac{\texttt{va\_arg}(ap, \tau)}{\begin{array}{l} a \coloneqq \texttt{load void*, 4, } \langle\!|ap|\!\rangle \\ result \coloneqq \texttt{load } \langle\!|\tau|\!\rangle\texttt{, } \langle\!|\texttt{alignof}(\tau)|\!\rangle\texttt{, } a \\ a' \coloneqq a + \langle\!|\texttt{roundup}_4(\texttt{sizeof}(\tau))|\!\rangle \\ \texttt{store void*, 4, } a'\texttt{, } \langle\!|ap|\!\rangle \end{array}} \qquad \frac{\texttt{va\_copy}(aq, ap)}{\begin{array}{l} a \coloneqq \texttt{load void*, 4, } \langle\!|ap|\!\rangle \\ \texttt{store void*, 4, } a\texttt{, } \langle\!|aq|\!\rangle \end{array}}$$

Figure 2.3: Translation of C's variadic macros to $\texttt{LLVM}_d$ instructions. $\texttt{roundup}_4(a)$ returns the closest multiple of 4 greater than or equal to a.

C's variadic macros `va_start`, `va_arg`, `va_end`, and `va_copy` are translated as shown in fig. 2.3. The translation is presented in the form of translation rules where templatized IR instructions (as used in fig. 2.2) are shown below C syntax. Like LLVM (in case of 32-bit x86 `cdecl` calling conventions), we translate `va_list` to a pointer type whose object (denoted by ap in fig. 2.2) is initialized to the first address of the variable argument list, obtained through $\texttt{LLVM}_d$'s `va_start_ptr` instruction, in `va_start`. `va_arg` increments the address in passed `va_list` ap according to the passed type τ and `va_end` resets the address in ap to `NULL` or 0 address. `va_copy` simply copies the value in source ap into destination aq.

2.1.2 Assembly

Broadly, an assembly program $\mathbb{A}$ consists of a code section with a sequence of assembly instructions, a data section with read-only and read-write global variables, and a symbol table that maps string symbols to memory addresses in code and data sections. Our translation validator checks that the address regions specified by the symbol table are well-aligned and non-overlapping, and uses it to relate a global variable (or procedure) in $\mathbb{C}$ to its address (or implementation) in $\mathbb{A}$. For read-only symbols common in both $\mathbb{C}$ and $\mathbb{A}$, the validator verifies that the memory contents are identical.

We assume that the OS guarantees the caller-side contract of the ABI calling conventions for the entry procedure, `main()`. For 32-bit x86, this means that at the start of program execution, the stackpointer is available in register `esp`, and the return address and input parameters `(argc,argv)` to `main()` are available in the stack region just above the stackpointer. For other procedure calls, the validator verifies the adherence to calling conventions at a callsite (in the caller) and assumes adherence at procedure entry (in the callee). Heap (de)allocation procedures like `malloc()` and `free()` are left uninterpreted, and so, the only compiler-visible way to allocate (and deallocate) memory in $\mathbb{A}$ is through the decrement (and increment) of the stackpointer stored in register `esp`.

2.1.3 Allocation and Deallocation

Allocation and deallocation instructions appear only in $\mathbb{C}$ and do not appear in $\mathbb{A}$. Let C represent a procedure in program $\mathbb{C}$.

An $\texttt{LLVM}_d$ instruction "p^a_{C}: `v` $\coloneqq$ `alloc n, τ, align`" at PC p^a_{C} allocates a contigu-

ous region of local memory with space for `n` elements of type τ aligned by `align` and returns its start address in `v`. The PC p^a_C of an `alloc` instruction is also called an *allocation site*. We denote an allocation site by z where $z = p^a_C$. Let the set of allocation sites in C be Z such that $z \in Z$. During translation of the C program to $\texttt{LLVM}_d$, we distinguish between allocation sites due to the declaration of a local variable (or a procedure-call argument) and allocation sites due to `alloca()` — we use Z_l for the former and Z_a for the latter, so that $Z = Z_l \cup Z_a$.

The address of an allocated region returned by `alloc` is non-deterministic, but is subject to two *Well-Formedness (WF) constraints*:

1. The newly allocated memory region should be separate from all currently allocated memory regions, i.e., there should be *no overlap*.
2. The address of the newly allocated memory region should be aligned by `align`.

An error-free execution of `alloc` will satisfy these two well-formedness constraints.

An $\texttt{LLVM}_d$ instruction "p^d_C: `dealloc` z" deallocates *all* local memory regions allocated due to the execution of (`alloc` instruction at) allocation site $z \in Z$. It is valid (i.e., not UB) to execute "`dealloc` z" when the `alloc` instruction at z was never executed — this can happen for an allocation site $z = za \in Z_a$ due to `alloca()`, where the `alloc` instruction is (conditionally) not executed but the `dealloc` instruction, inserted at the end of procedure C, is (unconditionally) executed.

In fig. 2.1b, the `alloc` instruction at `I2`, allocates space for `*n + 2` integers (of type `int`) with alignment 4 and stores the region's start address in `v`. The allocated region is identified by the allocation site `I2` and deallocated at `I17` using `dealloc I2`.

2.2 Transition Graph Representation

An $\texttt{LLVM}_d$ or assembly instruction may mutate the machine state, transfer control, perform I/O, or terminate the execution. We represent a C procedure, C in $\mathbb{C}$, as a transition graph, $C = (\mathcal{N}_C, \mathcal{E}_C)$, with a finite set of nodes $\mathcal{N}_C = \{n^s = n_1, n_2, \ldots, n_m\}$, and a finite set of labeled directed edges $\mathcal{E}_C$. A unique node n^s represents the start node or entry point of C, and every other node n_j $(2 \leq j \leq m)$ must be reachable from n^s. A node with no outgoing edges is a *terminating node*. A variable in C is identified by its scope-resolved unique name. The machine state σ_C of C consists of the set of

input parameters $\vec{y}$[2], set of temporary variables $\vec{t}$, and an explicit array variable M_C denoting the current state of memory. We use $\mathtt{i}_N$ to denote a bitvector type of size $N > 0$. The type $\mathtt{T}(M_C)$ of M_C is $\mathtt{i}_{32} \rightarrow \mathtt{i}_8$.

An assembly implementation of the C procedure C, identified through the symbol table in $\mathbb{A}$, is the assembly procedure A. Similarly to C, $A = (\mathcal{N}_A, \mathcal{E}_A)$ is also represented as a transition graph. The machine state σ_A of A consists of its hardware registers $\vec{regs}$ and memory M_A and is disjoint from the machine state σ_C of C.

Let $P \in \{C, A\}$. In addition to the memory (data) state M_P, we also need to track the allocation state, i.e., the set of intervals of addresses that have been allocated by the procedure. We use α (potentially with a subscript) to denote a memory address of bitvector type $\mathtt{i}_{32}$. Let $i = [\alpha_b, \alpha_e]$ represent an *address interval* starting at α_b and ending at α_e (both inclusive), such that $\alpha_b \leq_u \alpha_e$ (where $\leq_u$ is unsigned comparison operator for bitvectors). Let $[\alpha]_w$ be a shorthand for the address interval $[\alpha, \alpha+w-1_{\mathtt{i}_{32}}]$, where $1_{\mathtt{i}_{32}}$ ($n_{\mathtt{i}_{32}}$) is the two's complement representation of integer 1 (n) using 32 bits.

2.2.1 Address Set

Let Σ (potentially with a sub- or superscript) represent a set of addresses, or an *address set*. An empty address set is represented by $\emptyset$, and an address set of contiguous addresses is an address interval i. Two address sets overlap, written $\mathtt{ov}(\Sigma_1, \Sigma_2)$, iff $\Sigma_1 \cap \Sigma_2 \neq \emptyset$. Extended to $m > 2$ sets, $\mathtt{ov}(\Sigma_1, \Sigma_2, \ldots, \Sigma_m) \Leftrightarrow \exists_{1 \leq j_1 < j_2 \leq m} \mathtt{ov}(\Sigma_{j_1}, \Sigma_{j_2})$. $|\Sigma|$ represents the number of distinct addresses in Σ. For a non-empty address set, $\mathtt{lb}(\Sigma)$ and $\mathtt{ub}(\Sigma)$ represent the smallest and largest address respectively in Σ such that $\Sigma \subseteq [\mathtt{lb}(\Sigma), \mathtt{ub}(\Sigma)]$. $\mathtt{comp}(\Sigma)$ represents the *complement* of Σ, so that: $\forall_\alpha : (\alpha \in \Sigma) \Leftrightarrow (\alpha \notin \mathtt{comp}(\Sigma))$.

2.2.2 Memory Regions

To support dynamic (de)allocation and memory related transformations (e.g., re-ordering, elimination, etc.), an execution model in a validator needs to individually track regions of memory belonging to each variable, heap, stack, etc. We next describe the memory regions tracked by our model.

1. Let G be the set of names of all global variables in $\mathbb{C}$. For each global variable $g \in G$, we track the memory region belonging to that variable. We use the name of a global

[2]We use the notation $\vec{x}$ for representing a set.

variable $g \in G$ as its *region identifier* to identify the region belonging to g in both C and A.

2. For a procedure C, let Y be the set of names of formal parameters, including the variadic parameter, if present. We use the special name `vrdc` to identify the variadic parameter. The memory region belonging to a parameter $y \in Y$ is identified by y in both C and A.

3. The memory region allocated by allocation site $z \in Z$ is identified by z in C. In A, our algorithm potentially annotates allocation instructions corresponding to an allocation site z in C. Thus, the memory region allocated by these annotated instructions is also identified by z in A.

4. hp denotes the region belonging to the *program heap* (managed by the OS) in both C and A. Recall that we leave `malloc()` and `free()` uninterpreted so that hp does not grow or shrink as C (A) executes. This effectively models hp as a static region, even when `malloc()` and `free()` may implement dynamic growth and shrinking of its subregions that we do not track in our model.

5. Local variables and actual arguments may be allocated in the *call chain* of a procedure (caller, caller's caller, and so on). The accessible subset (accessible to procedure C[3]) is coalesced into a single region denoted by cl or *callers' locals* in both C and A.

6. In procedure A, stack memory can be allocated and deallocated through stackpointer decrement and increment. The addresses belonging to the stack frame of A (but not to a stack-allocated local variable $z \in Z$ or a parameter $y \in Y$) belong to the stk (stack) region in A. The stk region is absent in C.

7. Separate from stk, we use cs (*callers' stack*) to identify the region that belongs to the stack space (but not to cl) of the call chain of procedure A. cs is absent in C.

8. Program $\mathbb{A}$ may use more global memory than $\mathbb{C}$, e.g., to store precomputed constants to implement vectorizing transformations. Let F be the set of names of all non-empty *assembly-only global variables* in $\mathbb{A}$. For each $f \in F$, its memory region in A is identified by f.

[3] A local variable or actual argument v of procedure C′ in the call chain of procedure C is accessible in procedure C only if the address of v is accessible in C, i.e., v is address-taken in C′.

9. The region cv[4] denotes the inaccessible subset of local variables and actual arguments in the call chain of C. cv is present in both C and A, but inaccessible (i.e., cannot be read from or written to) in C and potentially accessible in A — we will elaborate on the accessibility aspect later when we discuss semantics of a memory access in A.

10. The region `free` denotes the free space, that does not belong to any of the aforementioned regions, in both C and A.

Let $R = G \cup F \cup Y \cup Z \cup \{hp, cl, cv, stk, cs, \texttt{free}\}$ represent all *region identifiers*; let $S = \{stk, cs\}$ denote the stack regions in A and $B = G \cup Y \cup Z \cup \{hp, cl\}$ ($B = R \setminus (F \cup S \cup \{cv, \texttt{free}\})$) denote the accessible regions in *b*oth C and A.

Let $G_r \subseteq G$ be the set of read-only global variables in $\mathbb{C}$; and, let $G_w = G \setminus G_r$ denote the set of read-write global variables. Let $F_r \subseteq F$ and $F_w = F \setminus F_r$ be defined analogously.

For each non-free region $r \in R \setminus \{\texttt{free}\}$, the machine state σ_P of a procedure P includes a unique variable Σ_P^r that tracks region r's address set as P executes. If Σ_P^r is a contiguous non-empty interval, we also refer to it as i_P^r. For $r \in G \cup F \cup Y \cup \{hp, cl, cv, cs\}$ ($r \in R \setminus (Z \cup \{stk, \texttt{free}\})$), Σ_P^r remains constant throughout P's execution. For $\vec{r} \subseteq R$, we define an expression $\Sigma_P^{\vec{r}} = \bigcup_{r \in \vec{r}} \Sigma_P^r$. Because C does not have a stack or an assembly-only global variable, $\Sigma_C^{F \cup S} = \emptyset$ holds throughout C's execution. At any point in P's execution, the free space can be computed as $\Sigma_P^{\texttt{free}} = \texttt{comp}(\Sigma_P^{B \cup F \cup S \cup \{cv\}})$. Notice that we do not use an explicit variable to track $\Sigma_P^{\texttt{free}}$.

For the IR and assembly procedures in fig. 2.1, $G = F = \emptyset$, $Y = \{\texttt{n}, \texttt{m}\}$, $Z = \{\texttt{I1}, \texttt{I2}, \texttt{I9}, \texttt{I10}\}$, and, therefore, $R = \{\texttt{n}, \texttt{m}, \texttt{I1}, \texttt{I2}, \texttt{I9}, \texttt{I10}, hp, cl, cv, stk, cs, \texttt{free}\}$.

2.2.3 Ghost Variables

Our validator introduces *ghost variables* in a procedure's execution semantics, i.e., variables that were not originally present in P. We use $\boxed{x}$ to indicate that x is a ghost variable. For each region $r \in G \cup Y \cup Z$ (resp. $r \in F$), we introduce $\boxed{\texttt{em.}r}$, $\boxed{\texttt{lb.}r}$, and $\boxed{\texttt{ub.}r}$ in C (resp. A) to track the *emptiness* (whether the region is empty), *lower bound* (smallest address), and *upper bound* (largest address) of Σ_C^r (resp. Σ_A^r) respectively; for $r \in G \cup Y$ (resp. $r \in F$), $\boxed{\texttt{sz.}r}$ tracks the size of Σ_C^r (resp. Σ_A^r), and for $z \in Z$, $\boxed{\texttt{lstSz.}z}$ tracks the *size of last allocation* due to execution of allocation site z. Two ghost variables

[4] cv stands for *callers' virtual*. The reason for tracking this region will become apparent when we discuss virtual allocation in section 2.6.

$\Sigma_P^{\texttt{rd}}$ and $\Sigma_P^{\texttt{wr}}$ track the set of addresses read and written by P respectively. Let $+$ be the set of all ghost variables.

2.2.4 Error Codes

Execution of C or A may terminate successfully, may never terminate, or may terminate with an error. We support two error codes to distinguish between two categories of errors: $\mathscr{U}$ and $\mathscr{W}$.

- In C: $\mathscr{U}$ represents an occurrence of UB, and $\mathscr{W}$ represents a violation of a well-formedness (WF) constraint that needs to be ensured either by the language or the OS (both external to the program itself).
- In A: $\mathscr{U}$ represents an occurrence of UB or a translation error, and $\mathscr{W}$ represents occurrence of a condition that can be assumed to never occur, e.g., if the OS ensures that it never occurs.

In summary, for a procedure P, $\mathscr{W}$ represents an error condition that P can *assume* to be absent (because the external environment ensures it), while $\mathscr{U}$ represents an error that P must *ensure* to be absent. For C, the programmer must ensure absence of UB; for A, the compiler must ensure absence of translation error.

2.2.5 Outside world and observable trace

Let Ω_P be a state of the outside world (OS/hardware) for P that supplies external inputs whenever P reads from it and consumes external outputs generated by P. Ω_P is assumed to mutate arbitrarily but deterministically based on the values consumed or produced due to the I/O operations performed by P during execution.

Let T_P be a potentially infinite sequence of observable trace events generated by an execution of P. A trace event t, produced during execution of P, is concatenated to T_P, written $T_P \coloneqq T_P \cdot t$. The trace events generated during an execution of P are as follows: (1) procedure-call invocation (`fcall`), (2) procedure return (`ret`), (3) procedure termination or halt (`exit`), (4) local allocation (`allocBegin` and `allocEnd`), (5) local deallocation (`dealloc`), and (6) a distinct silent trace event ($\perp$), indicating execution of an instruction that does not otherwise produce an observable event. A trace event may be associated with a set of values, e.g., a procedure-call invocation is associated with callee-observable values and a procedure return is associated with caller-observable

values — we will describe the exact values in each case later when we talk about graph instructions that produce these events (section 2.2.7). Procedure termination includes both error (indicated by an error code) and error-free termination (indicated by `exit`), each of which is identified distinctly.

2.2.6 Expressions

Let variable v and variables $\vec{v}$ or $\vec{x}$ be drawn from $\texttt{Vars} = \vec{t} \cup \vec{regs} \cup \boxed{+} \cup \{M_P, \Sigma_P^r\}$ for all $P \in \{\mathsf{C}, \mathsf{A}\}$ and for all $r \in R \setminus \texttt{free}$. Let $e(\vec{x})$ be an expression over $\vec{x}$, and $E(\vec{x})$ be a list of expressions over $\vec{x}$. An expression $e(\vec{x})$ is a well-formed combination of constants, variables $\vec{x}$, and arithmetic, logical, relational, memory access (read and write), and address set operators. For memory reads and writes, `select` (`sel` for short) and `store` (`st` for short) operations are used to access and modify M_P at a given address α. Further, the `sel` and `st` operators are associated with a `sz` parameter: $\texttt{sel}_{\texttt{sz}}(\texttt{arr}, \alpha)$ returns a little-endian concatenation of `sz` bytes starting at α in the array `arr`. Similarly, $\texttt{st}_{\texttt{sz}}(\texttt{arr}, \alpha, \texttt{data})$ returns a new array that has contents identical to `arr` except for the `sz` bytes starting at α that have been replaced by `data` in little-endian format. To encode reads/writes to a region of memory, we define projection and update operations.

Definition 2.2.1 ($\pi_\Sigma(M_P)$)**.** $\pi_\Sigma(M_P)$ *denotes the* ***projection*** *of* M_P *on addresses in* Σ*, i.e., if* $M'_P = \pi_\Sigma(M_P)$*, then* $\forall_{\alpha \in \Sigma} : \texttt{sel}_1(M'_P, \alpha) = \texttt{sel}_1(M_P, \alpha)$ *and* $\forall_{\alpha \notin \Sigma} : \texttt{sel}_1(M'_P, \alpha) = 0$*. The sentinel value* 0 *is used for the addresses outside* Σ*.*

We use $M_{P_1} =_\Sigma M_{P_2}$ as shorthand for $(\pi_\Sigma(M_{P_1}) = \pi_\Sigma(M_{P_2}))$, for $P_1, P_2 \in \{\mathsf{C}, \mathsf{A}\}$.

Definition 2.2.2 ($\texttt{upd}_\Sigma(M_P, M)$)**.** $\texttt{upd}_\Sigma(M_P, M)$ *denotes the* ***update*** *of* M_P *on addresses in* Σ *using the values in* M*. If* $M'_P = \texttt{upd}_\Sigma(M_P, M)$*, then* $M'_P =_\Sigma M$ *and* $M'_P =_{\texttt{comp}(\Sigma)} M_P$ *hold.*

2.2.7 Graph Instructions

Each labeled directed edge $e_P \in \mathcal{E}_P$ is labeled with one of the following *graph instructions*:

1. A *simultaneous assignment* of the form $\vec{v} := E(\vec{x})$: Because variables $\vec{v}$ and $\vec{x}$ may include M_P, an assignment suffices for encoding memory loads and stores. Similarly,

because the variables may be drawn from Σ_P^z (for an allocation site z), an assignment is also used to encode the allocation of an interval $i_{\texttt{new}}$ through $\Sigma_P^z := \Sigma_P^z \cup i_{\texttt{new}}$ and the deallocation of all addresses allocated due to z through $\Sigma_P^z := \emptyset$. Stack allocation and deallocation in A can be similarly represented as $\Sigma_{\mathsf{A}}^{stk} := \Sigma_{\mathsf{A}}^{stk} \cup i_{\texttt{new}}$ and $\Sigma_{\mathsf{A}}^{stk} := \Sigma_{\mathsf{A}}^{stk} \setminus i_{\texttt{new}}$ respectively.

2. A *guard* instruction of the form $e(\vec{x})?$: Instruction $e(\vec{x})?$ indicates that when execution reaches its head, the edge is taken iff its *edge condition* $e(\vec{x})$ evaluates to `true`. For every other instruction, the edge is always taken upon reaching its head, i.e., its edge condition is `true`.

   For a *non-terminating* node $n_P \in \mathcal{N}_P$ (i.e., n_P has at least one outgoing edge [5]), the guards of all edges departing from n_P must be mutually exclusive, and their disjunction must evaluate to *true*.

3. A type-parametric *choose* instruction $\theta(\vec{\tau})$: Instruction $\vec{v} := \theta(\vec{\tau})$ non-deterministically chooses values of types $\vec{\tau}$ and assigns them to variables $\vec{v}$, e.g., a memory with non-deterministic contents is obtained by using $\theta(\mathtt{i}_{32} \rightarrow \mathtt{i}_8)$ so that $M_P := \mathtt{upd}_\Sigma(M_P, \theta(\mathtt{i}_{32} \rightarrow \mathtt{i}_8))$ updates M_P at addresses in Σ with non-deterministically chosen data values.

4. A *read* (`rd`) or *write* (`wr`) I/O instruction: A read instruction $\vec{v} := \mathtt{rd}(\vec{\tau})$ reads values of types $\vec{\tau}$ from the outside world into variables $\vec{v}$, e.g., an address set is read using $\Sigma := \mathtt{rd}(2^{\mathtt{i}_{32}})$ where $2^{\mathtt{i}_{32}}$ represents the type of address set Σ.

   A write instruction $\mathtt{wr}(V(E(\vec{x})))$ writes the value constructed by value constructor V using $E(\vec{x})$ to the outside world. A value constructor $V(\ldots)$ is defined for each type of observable trace event.

   - For a procedure-call event, $\mathtt{fcall}(\rho, \vec{v}, \vec{r}, M)$ represents a value constructed for a procedure call to callee with name (or address) ρ, the actual arguments $\vec{v}$, callee-observable regions $\vec{r}$, and memory M.
   - For a procedure-return event, $\mathtt{ret}(E(\vec{x}))$ represents a value constructed during procedure return that captures observable values computed through $E(\vec{x})$.
   - For local allocation and deallocation events, $\mathtt{allocBegin}(z, w, a)$, $\mathtt{allocEnd}(z, i, M)$, and $\mathtt{dealloc}(z)$ represent the values constructed for allocation (`allocBegin` and `allocEnd`) and deallocation (`dealloc`) due to allocation site z with the associated observables, size of allocation w, alignment a, allocated interval i, and memory M.

[5] Recall that a terminating node has no outgoing edges.

A read or write instruction mutates outside world Ω_P arbitrarily based on the read and written values. Further, the data items read or written are appended to the observable trace T_P. Let $\texttt{read}_{\vec{\tau}}(\Omega_P)$ be an uninterpreted function that reads values of types $\vec{\tau}$ from Ω_P; and $\texttt{io}(\Omega_P, \texttt{rw}, E(\vec{x}))$ be an uninterpreted function that returns an updated state of Ω_P after an I/O operation of type $\texttt{rw} \in \{\texttt{r}, \texttt{w}\}$ (read or write) with values $E(\vec{x})$. Thus, in its explicit syntax, $\vec{v} \coloneqq \texttt{rd}(\vec{\tau})$ translates to a sequence of instructions: $\vec{v} \coloneqq \texttt{read}_{\vec{\tau}}(\Omega_P)$; $\Omega_P \coloneqq \texttt{io}(\Omega_P, \texttt{r}, \vec{v})$; $T_P \coloneqq T_P \cdot \vec{v}$, where $\cdot$ is the trace concatenation operator. Similarly, $\texttt{wr}(V(E(\vec{x})))$ translates to: $\Omega_P \coloneqq \texttt{io}(\Omega_P, \texttt{w}, V(E(\vec{x})))$; $T_P \coloneqq T_P \cdot V(E(\vec{x}))$. Henceforth, we only use the implicit syntax for brevity.

5. An error-free and error-indicating *halt* instruction that terminates execution. $\texttt{halt}(\emptyset)$ indicates termination without error and $\texttt{halt}(\kappa)$ indicates termination with error code $\kappa \in \{\mathscr{U}, \mathscr{W}\}$. Upon termination without error, a special `exit` event is appended to trace T_P; upon termination with error, the error code is appended to T_P.

   The destination of an edge with a `halt` instruction is a terminating node. We create a unique terminating node for an error-free exit. We also create a unique terminating node for each error code, also called an *error node*. An edge terminating at an error node is called an *error edge*. $\mathscr{U}_P$ and $\mathscr{W}_P$ represent error nodes in P for errors $\mathscr{U}$ and $\mathscr{W}$ respectively. Execution transfers to an error node upon encountering the corresponding error. Let $\mathcal{N}_P^{\mathscr{UW}} = \mathcal{N}_P \setminus \{\mathscr{U}_P, \mathscr{W}_P\}$ be the set of error-free nodes in P.

In addition to the observable trace events generated by `rd`, `wr`, and `halt` instructions, the execution of every instruction in P also appends an observable *silent trace event*, denoted $\perp$, to T_P. Silent trace events count the number of executed instructions as a proxy for observing the passage of time.

2.3 Translations of C and A to their Graph Representations

Figures 2.4 to 2.7 (and figs. 2.8, 2.10 and 2.11 later) present the key translation rules from $\texttt{LLVM}_d$ and (abstracted) assembly instructions to graph instructions. Each rule is composed of three parts separated by a horizontal line segment: on the left is the name of the rule, above the line segment is the $\texttt{LLVM}_d$/assembly instruction, and below the line segment is the graph instructions listing. For example, the top left corner of fig. 2.4 shows the parametric (Op) rule which gives the translation of an operation using

arithmetic/logical/relational operator `op` in $\texttt{LLVM}_d$ to corresponding graph instructions.

We describe the operators and predicates used in the rules in table 2.1. We use C-like constructs in graph instructions as syntactic sugar for brevity, e.g. ';' is used for sequencing, '?:' is used for conditional assignment, and `if`, `else`, and `for` are used for control flow transfer. We highlight the read and write I/O instructions with a shaded background and use **bold**, **colored** fonts for error-indicating `halt` instructions. We use "macros" `IF` and `ELSE` to choose translations based on a boolean condition on the input syntax.

2.3.1 Translation of C

Figures 2.4 and 2.5 shows the rules for translating $\texttt{LLVM}_d$ instructions to graph instructions. We discuss each in the following paragraphs.

The parametric rule (Op) gives the translation for application of an arithmetic/logical/relational operator `op` over arguments $\vec{x}$ (fig. 2.4). An application of `op` may trigger undefined behavior (UB) for certain inputs, as abstracted through the $\texttt{UB}_\mathsf{C}(\texttt{op}, \vec{x})$ operation. While there are many undefined behaviors in the C standard, we model only the following that we have seen getting exploited for optimization by the compiler:

1. *Logical or arithmetic shift operation*: The second operand should be bounded by a limit which is determined by the bit width of the first operand. This is required when a shift operation in C is translated to an x86 shift opcode in A.
2. *Address computation (*`getelementptr inbounds` *opcode in LLVM IR)*: No overflow and underflow in the intermediate and final computations. An optimizing compiler may assume this for conversion of inequality relations to disequality relations.
3. *Division operation*: The denominator operand should be non-zero. This is required for showing an exception-free execution of the corresponding division operation in A.

We will describe the $\beta(v) \coloneqq \ldots$ part of the translation shortly.

The ($\textsc{Load}_\mathsf{C}$) and ($\textsc{Store}_\mathsf{C}$) rules (fig. 2.4) show the translations for `load` and `store` instructions respectively. A UB-free execution of `load` and `store` requires the dereferenced pointer p to satisfy memory access safety constraints specified through `accessIsSafeC()` predicate (defined in table 2.1) — a *safe* memory access is recorded in ghost variable $\Sigma^{\texttt{rd}}_\mathsf{C}$ for `load` and $\Sigma^{\texttt{wr}}_\mathsf{C}$ for `store`. An access through pointer p is safe iff p is non-`NULL`

Table 2.1: Definitions of operators and predicates used in translations in figs. 2.4 to 2.8 and 2.10 to 2.12

Operator	Definition
$\texttt{sz}(\tau)$	Returns the size (in bytes) of type τ. For example, $\texttt{sz}(\texttt{i}_{32}) = 4$ and $\texttt{sz}(\texttt{i}_{8}*) = 4$.
$\texttt{T}(a)$	Returns the type τ of a where a may be a global variable, a parameter, or a register. For example, $\texttt{T}(\texttt{eax}) = \texttt{i}_{32}$.
$\triangle_\tau(\texttt{eax}, \texttt{edx})$	A type-parametric operator which derives the return value of an assembly procedure with return type τ from input registers eax and edx using the calling conventions, e.g., $\triangle_{\texttt{i}_8}(\texttt{eax}, \texttt{edx}) = \texttt{extract}_{7,0}(\texttt{eax})$, $\triangle_{\texttt{i}_{32}}(\texttt{eax}, \texttt{edx}) = \texttt{eax}$, $\triangle_{\texttt{i}_{64}}(\texttt{eax}, \texttt{edx}) = \texttt{concat}(\texttt{edx}, \texttt{eax})$, where $\texttt{extract}_{h,l}(a)$ extracts bits h down to l from a and $\texttt{concat}(a, b)$ returns the bitvector concatenation of a and b where b takes the less significant position.
$\nabla_\tau(v)$	Inverse of $\triangle_\tau(\texttt{eax}, \texttt{edx})$. Distributes the packed bitvector v of type τ into two bitvectors of 32 bit-width each, setting the bits not covered by v to some non-deterministic value.
$\texttt{ROM}^r_P(i)$	Returns a memory array containing the contents of read-only global variable named r in P. The contents are mapped at the addresses in the provided interval i.
$\texttt{addrSets}_F()$	Returns the address sets of the assembly-only global variables F using the symbol table in the executable $\mathbb{A}$.

Predicate	Definition
$\texttt{aligned}_n(a)$	Bitvector a is n bytes aligned. Equivalent to: $a\%n = 0$, where % is remainder operator.
$\texttt{isAlignedIntrvl}_a(p, w)$	A w-sized sequence of addresses starting at p is aligned by a and does not wraparound. Equivalent to: $\texttt{aligned}_a(p) \wedge (p \leq_u p + w - 1_{\texttt{i}_{32}})$.
$\texttt{accessIsSafeC}_{\tau,a}(p, \Sigma)$	Equivalent to: $\texttt{isAlignedIntrvl}_a(p, \texttt{sz}(\tau)) \wedge ([p]_{\texttt{sz}(\tau)} \subseteq \Sigma)$.
$\texttt{addrSetsAreWF}(\Sigma^{hp}_P, \Sigma^{cl}_P, \Sigma^{cv}_P, \ldots, i^g_P, \ldots, \Sigma^f_P, \ldots, i^y_P, \ldots, \Sigma^{\texttt{vrdc}}_P)$	The address sets passed as parameter are well-formed with respect to C semantics. Equivalent to: $(0_{\texttt{i}_{32}} \notin \Sigma^{G\cup F\cup Y\cup\{hp,cl,cv\}}_P) \wedge \neg\texttt{ov}(\Sigma^{hp}_P, \Sigma^{cl}_P, \ldots, i^g_P, \ldots, \Sigma^f_P, \ldots, i^y_P, \ldots, \Sigma^{\texttt{vrdc}}_P) \wedge \neg\texttt{ov}(\Sigma^{G\cup Y\cup\{hp,cl\}}_P, \Sigma^{cv}_P) \wedge (\Sigma^{\texttt{vrdc}}_P \neq \emptyset \Rightarrow \texttt{isInterval}(\Sigma^{\texttt{vrdc}}_P)) \wedge \forall_{r\in G\cup(Y\setminus\{\texttt{vrdc}\})\cup F} : (|i^r_P| = \texttt{sz}(\texttt{T}(r)) \wedge \texttt{aligned}_{\texttt{algnmnt}(r)}(\texttt{lb}(i^r_P)))$, where $\texttt{isInterval}(\Sigma^{\texttt{vrdc}}_P)$ holds iff the address set $\Sigma^{\texttt{vrdc}}_P$ is an interval, $\texttt{algnmnt}(r)$ returns the alignment of variable r.
$\texttt{intrvlInSet}(\alpha_b, \alpha_e, \Sigma)$	The pair (α_b, α_e) forms a valid interval inside the address set Σ. Equivalent to: $(\alpha_b \neq 0_{\texttt{i}_{32}}) \wedge (\alpha_b \leq_u \alpha_e) \wedge ([\alpha_b, \alpha_e] \subseteq \Sigma)$
$\texttt{intrvlInSet}_a(\alpha_b, \alpha_e, \Sigma)$	Equivalent to: $\texttt{aligned}_a(\alpha_b) \wedge \texttt{intrvlInSet}(\alpha_b, \alpha_e, \Sigma)$
$\texttt{obeyCC}(e_{\texttt{esp}}, \vec{\tau}, \vec{x})$	Pointers $\vec{x}$ match the expected addresses of arguments for a procedure call in assembly. Based on the calling conventions, obeyCC uses the value of the current stackpointer ($e_{\texttt{esp}}$) and parameter types ($\vec{\tau}$) to obtain the expected addresses of the arguments. For example, $\texttt{obeyCC}(\texttt{esp}, (\texttt{i}_8, \texttt{i}_{32}), (\texttt{esp}, \texttt{esp} + 4_{\texttt{i}_{32}}))$ holds.
$\texttt{overflow}_{mul}(a, b)$	Signed multiplication of bitvectors a, b overflows. E.g., $\texttt{overflow}_{mul}(2147483647_{\texttt{i}_{32}}, 2_{\texttt{i}_{32}})$ holds.
$\texttt{stkIsWF}(\texttt{esp}, \texttt{stk}_e, \texttt{cs}_e, \vec{\tau}, \Sigma^{hp}_A, \Sigma^{cl}_A, \Sigma^{G\cup F}_A, \ldots, i^y_A, \ldots, \Sigma^{\texttt{vrdc}}_A)$	The pairs $(\texttt{esp}, \texttt{stk}_e)$, $(\texttt{stk}_e, \texttt{cs}_e)$ represent well-formed intervals for initial *stk* region and initial *cs* region with respect to parameter types $\vec{\tau}$ and other (input) address sets in A. Equivalent to: $\texttt{aligned}_{16}(\texttt{esp} + 4_{\texttt{i}_{32}}) \wedge (\texttt{esp} \leq_u \texttt{esp} + 4_{\texttt{i}_{32}}) \wedge \neg\texttt{ov}([\texttt{esp}]_{4_{\texttt{i}_{32}}}, \Sigma^{G\cup F\cup Y\cup\{hp,cl\}}_A) \wedge \texttt{obeyCC}(\texttt{esp} + 4_{\texttt{i}_{32}}, \vec{\tau}, \ldots, \texttt{lb}(i^y_A), \ldots) \wedge (\texttt{stk}_e <_u \texttt{cs}_e) \wedge \neg\texttt{ov}([\texttt{stk}_e + 1_{\texttt{i}_{32}}, \texttt{cs}_e], \Sigma^{G\cup F\cup\{hp\}}_A) \wedge \Sigma^{cl}_A \subseteq [\texttt{stk}_e + 1_{\texttt{i}_{32}}, \texttt{cs}_e]$
$\texttt{UB}_P(\texttt{op}, \vec{x})$	Application of operation op of procedure P on arguments $\vec{x}$ triggers UB. E.g., $\texttt{UB}_C(\texttt{udiv}, (1_{\texttt{i}_{32}}, 0_{\texttt{i}_{32}}))$ holds.

$$\text{(Op)}\ \frac{p^j_{\mathsf{C}} :\ v \coloneqq \mathtt{op}(\vec{x})}{\begin{array}{l}\underline{\mathtt{if}}\ (\mathtt{UB}_{\mathsf{C}}(\mathtt{op}, \vec{x}))\ \mathtt{halt}(\mathscr{U});\\ v \coloneqq \mathtt{op}(\vec{x});\\ \ldots, x, \ldots \coloneqq \vec{x}; \qquad \beta(v) \coloneqq \beta^{\mathtt{op}}(\ldots, \beta(x), \ldots);\end{array}}$$

$$\text{(AssignConst)}\ \frac{p^j_{\mathsf{C}} :\ v \coloneqq c}{\begin{array}{l}v \coloneqq c;\\ \beta(v) \coloneqq \emptyset;\end{array}}$$

$$\text{(Load}_{\mathsf{C}}\text{)}\ \frac{p^j_{\mathsf{C}} :\ v \coloneqq \mathtt{load}\ \tau,\ a,\ p}{\begin{array}{l}\underline{\mathtt{if}}\ (\neg\mathtt{accessIsSafeC}_{\tau,a}(p, \Sigma^{\beta(p)}_{\mathsf{C}}))\\ \quad \mathtt{halt}(\mathscr{U});\\ v \coloneqq \mathtt{sel}_{\mathtt{sz}(\tau)}(M_{\mathsf{C}}, p);\\ \beta(v) \coloneqq \beta_M(\beta(p));\\ \boxed{\Sigma^{\mathtt{rd}}_{\mathsf{C}}} \coloneqq \boxed{\Sigma^{\mathtt{rd}}_{\mathsf{C}}} \cup [p]_{\mathtt{sz}(\tau)};\end{array}}$$

$$\text{(Store}_{\mathsf{C}}\text{)}\ \frac{p^j_{\mathsf{C}} :\ \mathtt{store}\ \tau,\ a,\ v,\ p}{\begin{array}{l}\underline{\mathtt{if}}\ (\neg\mathtt{accessIsSafeC}_{\tau,a}(p, \Sigma^{\beta(p)\setminus G_r}_{\mathsf{C}}))\\ \quad \mathtt{halt}(\mathscr{U});\\ M_{\mathsf{C}} \coloneqq \mathtt{st}_{\mathtt{sz}(\tau)}(M_{\mathsf{C}}, p, v);\\ \beta_M(\beta(p)) \coloneqq \beta_M(\beta(p)) \cup \beta(v);\\ \boxed{\Sigma^{\mathtt{wr}}_{\mathsf{C}}} \coloneqq \boxed{\Sigma^{\mathtt{wr}}_{\mathsf{C}}} \cup [p]_{\mathtt{sz}(\tau)};\end{array}}$$

$$\text{(VaStartPtr)}\ \frac{p^j_{\mathsf{C}} :\ p \coloneqq \mathtt{va\_start\_ptr}}{\begin{array}{l}\underline{\mathtt{if}}\ (\Sigma^{\mathtt{vrdc}}_{\mathsf{C}} = \emptyset)\ \{\\ \quad p \coloneqq 0_{\mathtt{i}_{32}}; \qquad \beta(p) \coloneqq \emptyset;\\ \}\ \underline{\mathtt{else}}\ \{\\ \quad p \coloneqq \boxed{\mathtt{lb.vrdc}}; \qquad \beta(p) \coloneqq \{\mathtt{vrdc}\};\\ \}\end{array}}$$

Figure 2.4: Translation rules for converting $\mathtt{LLVM}_d$ instructions to graph instructions. op represents an arithmetic, logical, or relational operator. c represents a constant.

($\neq 0_{\mathtt{i}_{32}}$ in our modeling[6]), aligned by the required alignment a, and have its access interval belong to the regions which p may *point to* or p may be *based on* (§6.5.6p8 of the C11 standard [21]).

To identify the regions a pointer p may be based on, we define two maps:

(1) $\beta : \mathtt{Vars} \to 2^R$ that tracks the set of regions a variable (e.g., a pointer) may be based on, so that for a variable $x \in \mathtt{Vars}$, $\beta(x)$ returns the set of regions x may point to.
(2) $\beta_M : R \to 2^R$ that tracks the set of regions that pointers in a memory region may be based on, so that for a region $r \in R$, $\beta_M(r)$ returns the set of regions that some (pointer) value stored in $\pi_{\Sigma^r_{\mathsf{C}}}(M_{\mathsf{C}})$ may point to.

For convenience, we extend β and β_M to also take as input a set of variables and a set of regions respectively so that $\beta(\vec{x})$ is equivalent to $\bigcup_{x\in\vec{x}} \beta(x)$, and $\beta_M(\vec{r})$ is equivalent to $\bigcup_{r\in\vec{r}} \beta_M(r)$. Similar extension is used in assignment to β_M so that $\beta_M(\vec{r_1}) \coloneqq \vec{r_2}$ is equivalent to '$\mathtt{for}\ r_1\ \mathtt{in}\ \vec{r_1}\ \{\ \beta_M(r_1) \coloneqq \vec{r_2};\ \}$'.

[6]The $\mathtt{accessIsSafeC}_{\tau,a}(p, \Sigma)$ definition in table 2.1 does not include the $\neq 0_{\mathtt{i}_{32}}$ clause because it assumes that $0_{\mathtt{i}_{32}} \notin \Sigma$ so that a $[p]_{\mathtt{sz}(\tau)} \subseteq \Sigma$ check implies $p \neq 0_{\mathtt{i}_{32}}$. $0_{\mathtt{i}_{32}} \notin \Sigma^r_{\mathsf{C}}$ for an allocated region r is an invariant in C.

The initialization and update of β and β_M due to each $\texttt{LLVM}_d$ instruction can be seen in figs. 2.4 and 2.5. In (Op), for an operation op, $\beta^{\texttt{op}} : (2^R \times 2^R \ldots \times 2^R) \rightarrow 2^R$ represents the over-approximate abstract transfer function for $v := \texttt{op}(\vec{x})$, that takes as input $\beta(x_1), \beta(x_2), \ldots, \beta(x_m)$ for $\vec{x} = x_1, x_2, \ldots, x_m$ and returns $\beta(v)$. We use:

- $\beta^{\texttt{op}}(\vec{r}) = \vec{r}$, if op is identity, bitwise complement and unary negation.
- $\beta^{\texttt{op}}(\vec{r_1}, \ldots, \vec{r_m}) = \bigcup_{1 \leq j \leq m} \vec{r_j}$, if op is bitvector addition, subtraction, shift, bitwise-{and,or}, extraction, or concatenation.
- $\beta^{\texttt{op}}(\vec{r_1}, \ldots, \vec{r_m}) = \emptyset$, if op is bitvector multiplication, division, logical, relational, or any other remaining operator.

The rule (AssignConst) for a constant assignment to variable v sets $\beta(v)$ to empty set making it impossible to fabricate pointers from integer literals.

(VaStartPtr) (fig. 2.4) gives the translation rule for the `va_start_ptr` instruction of $\texttt{LLVM}_d$ — recall that the `va_start_ptr` instruction is used during translation of the C variadic macro `va_start()` to $\texttt{LLVM}_d$ (fig. 2.3). The rule sets the assigned variable p to the first address of the variadic parameter region (obtained through `lb.vrdc`) if the address set $\Sigma_{\mathsf{C}}^{\texttt{vrdc}}$ of the variadic parameter region is non-empty, otherwise $0_{\texttt{i}_{32}}$ (`NULL` in our representation) is used. Because `lb.vrdc` is a pointer inside `vrdc` region, $\beta(p)$ is set to singleton {`vrdc`}.

The (Entry$_{\mathsf{C}}$) rule in fig. 2.5 presents the initialization performed at the entry of procedure C. The allocation state, address set Σ_{C}^r of each region $r \in R \setminus \{\texttt{free}\}$, and memory state, M_{C}, of C are initialized using reads from the outside world Ω_{C} — the contents of read-only global regions (G_r) are initialized separately using their predefined values (through $\texttt{ROM}_{\mathsf{C}}^g(i_{\mathsf{C}}^g)$ defined in table 2.1). The read address sets are checked for well-formedness with respect to C semantics through `addrSetsAreWF()` (defined in table 2.1), or else error $\mathscr{W}$ is triggered; well-formedness in this context requires that the address sets do not contain the `NULL` pointer ($0_{\texttt{i}_{32}}$), a global variable/argument address set is an interval, and the address sets do not overlap. The rule concludes with initialization of the ghost variables associated with the regions and the β, β_M maps used for tracking may based on information. Notice that the region cv is not included in the set of reachable regions through β and β_M making it unreachable throughout C's execution (this is ensured during a fresh allocation through an `alloc` instruction (Alloc) as well).

(RetV) and (Ret$_{\mathsf{C}}$) (fig. 2.5) present translations for procedure-return instructions

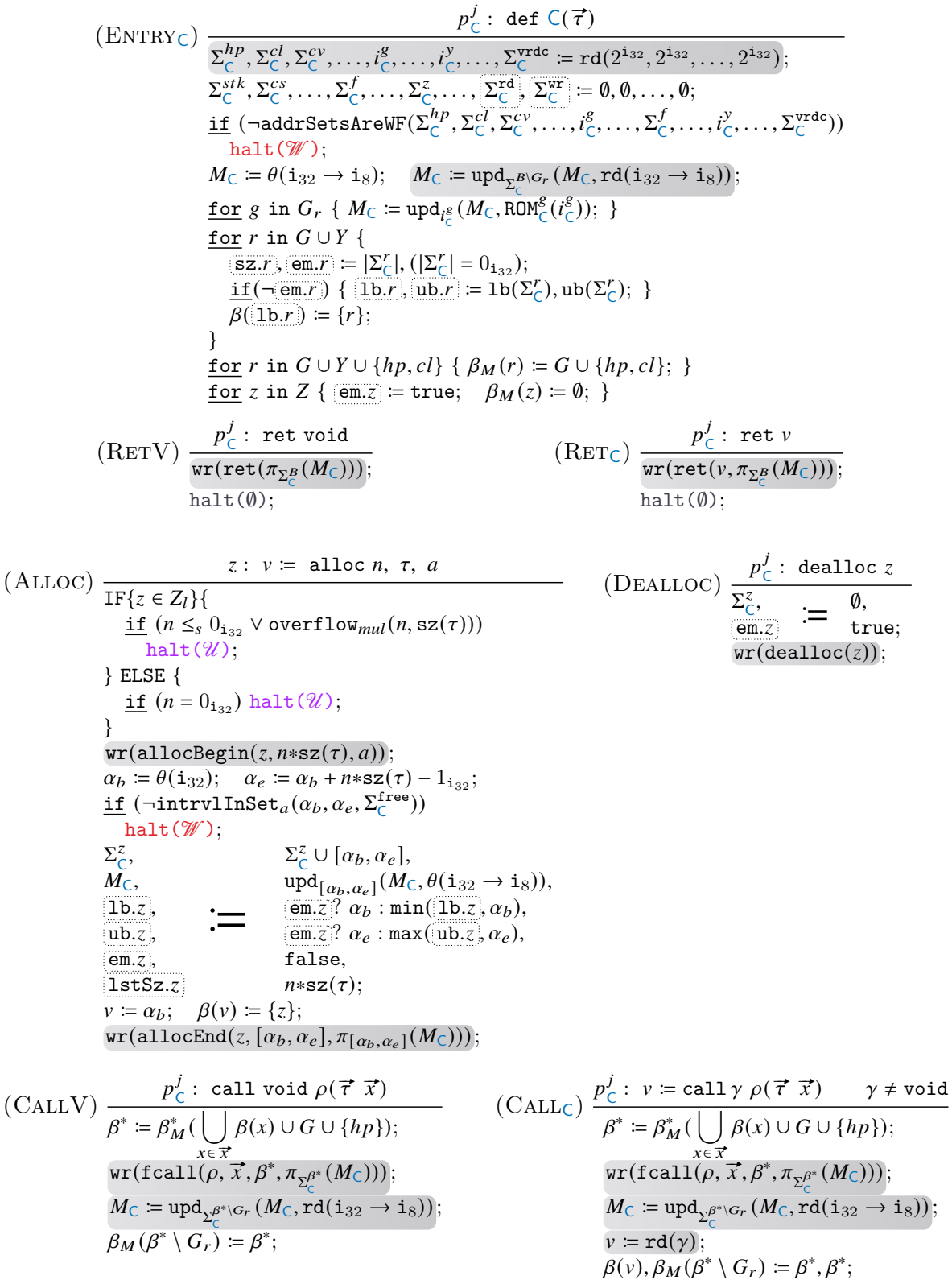


Figure 2.5: Translation rules for converting LLVM$_d$ instructions to graph instructions.

'`ret void`' and '`ret` v' respectively. Return from a procedure produces a non-silent observable event with the return value (in case of `ret` v) and the memory state of accessible regions in C written to outside world through `wr(ret(...))`.

The (Alloc) and (Dealloc) rules (fig. 2.5) give the semantics for the allocation and deallocation of local memory, identified by an allocation site z, through `alloc` and `dealloc` respectively. For an allocation, if $z \in Z_l$[7], the computation of allocation size, obtained by multiplying the number of elements allocated (n) with size of each element ($\mathtt{sz}(\tau)$), has a *no overflow* constraint for a UB-free execution (shown through translation-selecting $\mathtt{IF}\{z \in Z_l\}$). The translation uses the choose instruction ($\theta(\mathtt{i}_{32})$) for identifying the non-deterministic start address of the freshly allocated interval $[\alpha_b, \alpha_e]$. A freshly allocated interval must satisfy the two well-formedness (WF) constraints of *no overlap* with existing allocated regions and *alignment* of the start address, implemented through $\neg\mathtt{intrvlInSet}_a()$ check (defined in table 2.1), otherwise error $\mathscr{W}$ is triggered[8]. An allocation adds the allocated interval $[\alpha_b, \alpha_e]$ to the address set Σ^z_{C} of the local; a deallocation empties it. Similar to the start address, the memory contents of the allocated interval are non-deterministically initialized through $\mathtt{upd}_{[\alpha_b,\alpha_e]}(M_{\mathsf{C}}, \theta(\mathtt{i}_{32} \rightarrow \mathtt{i}_8))$. The various ghost variables associated with region z are updated in both cases: an allocation updates the lower bound $\boxed{\mathtt{lb}.z}$, upper bound $\boxed{\mathtt{ub}.z}$, and last allocation size $\boxed{\mathtt{lstSz}.z}$ and resets the boolean ghost variable $\boxed{\mathtt{em}.z}$ that tracks the emptiness of z; a deallocation simply sets $\boxed{\mathtt{em}.z}$ to `true`. We use the simultaneous assignment instruction for updating Σ^z_{C}, M_{C}, and the ghost variables in a single step.

A (de)allocation instruction generates observable traces using the `wr` instruction at the beginning and end of each execution of that instruction. We will later use these traces to identify a lockstep correlation of (de)allocation events between C and A, towards validating a translation.

Modeling procedure calls in C

The semantics of an $\mathtt{LLVM}_d$ procedure-call instruction is given by the rules (CallV) and (Call_{C}) (fig. 2.5). For an $\mathtt{LLVM}_d$ call instruction "`call` γ $\rho(\vec{\tau}\ \vec{x})$", we produce a non-silent observable trace event using the `wr` instruction with observables callee name/address ρ, arguments $\vec{x}$, and callee-accessible regions and memory state (β^*

[7]Recall that Z_l denotes the allocation sites due to the declaration of a local variable or a procedure-call argument

[8]Recall that $\mathscr{W}$ represents an error condition that is external to the procedure and can be assumed to never occur.

and $\pi_{\Sigma_C^{\beta^*}}(M_C)$ in fig. 2.5). A callee may access a memory region iff it is *transitively reachable* from a global variable $g \in G$, the heap hp, or one of the arguments $x \in \vec{x}$. The (transitively) reachable memory regions are over-approximately computed through a reflexive-transitive closure of β_M, denoted β_M^* in fig. 2.5.

To model return values and side-effects to the memory state due to a callee, `rd` instructions are used. A `rd` instruction is used to arbitrarily clobber each the callee-observable state element. Thus, if a callee procedure terminates normally (i.e., without error), `wr` and `rd` instructions over-approximately model the execution of a procedure call. Later, our definition of refinement (section 2.4) caters to the case when a callee procedure may not terminate or terminates with error (i.e., a termination with error is modeled identically to non-termination).

Lastly, a procedure call can potentially be recursive — our modeling does not differentiate between a recursive and a non-recursive call. As we will see later in section 2.5.4, a consequence of this over-approximate modeling is that the tail-call elimination optimization, where a tail recursive call is replaced by a loop, cannot be covered by our refinement definition (i.e., the transformed procedure will not be considered refinement of the original procedure).

2.3.2 Translation of A

The rules for translating assembly instructions to graph instructions are shown in figs. 2.6 and 2.7 (and later figs. 2.8, 2.10 and 2.11). We abstract the assembly opcodes to an IR-like syntax for ease of exposition. For example, in (Load$_A$), a memory read operation is represented by a `load` instruction which is annotated with address p, access size w (in bytes), and required alignment a[9]. Similarly, in (Store$_A$), a memory write operation is represented by a `store` instruction with similar operands. Both (Load$_A$) and (Store$_A$) translations update the ghost address sets $\Sigma_A^{\texttt{rd}}$ and $\Sigma_A^{\texttt{wr}}$, in the same manner as done in (Load$_C$) and (Store$_C$). A memory access error due to `NULL` address dereference or an unaligned access or an out-of-bounds access triggers a $\mathscr{U}$ error (indicating a translation error). An access through address p is deemed out-of-bounds if p lies in `free` region ($p \in \Sigma_A^{\texttt{free}}$) or p lies in that part of cv region which does not overlap with assembly-only regions $F \cup S$[10]. For a `store`, accessing read-only regions ($G_r \cup F_r$) is also considered as out-of-bounds. Other machine exceptions such

[9] In 32-bit x86 alignment is only mandatory for some instructions (e.g., vector instructions).
[10] Recall that the cv region represents the inaccessible subset of local memory in call chain of A.

$$\text{(Op-esp)}\ \frac{p_{\mathbb{A}}^{j}:\ \texttt{esp} \coloneqq \texttt{op}(\vec{x})}{\begin{array}{l}\underline{\texttt{if}}\ (\texttt{UB}_{\mathbb{A}}(\texttt{op},\vec{x}))\ \texttt{halt}(\mathscr{U});\\ t \coloneqq \texttt{op}(\vec{x});\\ \underline{\texttt{if}}\ (\texttt{isPush}(p_{\mathbb{A}}^{j},\texttt{esp},t))\ \{\\ \quad \underline{\texttt{if}}\ (\neg\texttt{intrvlInSet}(t,\texttt{esp}-1_{\mathtt{i}_{32}},\Sigma_{\mathbb{A}}^{\texttt{free}}\cup(\Sigma_{\mathbb{A}}^{cv}\setminus\Sigma_{\mathbb{A}}^{F})))\\ \quad\quad \texttt{halt}(\mathscr{W});\\ \quad \Sigma_{\mathbb{A}}^{stk} \coloneqq \Sigma_{\mathbb{A}}^{stk}\cup[t,\texttt{esp}-1_{\mathtt{i}_{32}}];\\ \quad M_{\mathbb{A}} \coloneqq \texttt{upd}_{[t,\texttt{esp}-1_{\mathtt{i}_{32}}]}(M_{\mathbb{A}},\theta(\mathtt{i}_{32}\rightarrow\mathtt{i}_{8}));\\ \}\ \underline{\texttt{else if}}\ (t\neq\texttt{esp})\ \{\\ \quad \underline{\texttt{if}}\ (\neg\texttt{intrvlInSet}(\texttt{esp},t-1_{\mathtt{i}_{32}},\Sigma_{\mathbb{A}}^{stk}))\\ \quad\quad \texttt{halt}(\mathscr{U});\\ \quad \Sigma_{\mathbb{A}}^{stk} \coloneqq \Sigma_{\mathbb{A}}^{stk}\setminus[\texttt{esp},t-1_{\mathtt{i}_{32}}];\\ \}\\ \texttt{esp} \coloneqq t;\quad \boxed{\texttt{sp}.p_{\mathbb{A}}^{j}} \coloneqq t;\end{array}}$$

$$\text{(Load}_{\mathbb{A}}\text{)}\ \frac{p_{\mathbb{A}}^{j}:\ v \coloneqq \texttt{load}\ w,\ a,\ p}{\begin{array}{l}\underline{\texttt{if}}\ (\quad \neg\texttt{isAlignedIntrvl}_{a}(p,w)\\ \qquad \vee\ \texttt{ov}([p]_{w},\Sigma_{\mathbb{A}}^{\texttt{free}}\cup(\Sigma_{\mathbb{A}}^{cv}\setminus\Sigma_{\mathbb{A}}^{F\cup S})))\\ \quad \texttt{halt}(\mathscr{U});\\ v \coloneqq \texttt{sel}_{w}(M_{\mathbb{A}},p);\\ \boxed{\Sigma_{\mathbb{A}}^{\texttt{rd}}} \coloneqq \boxed{\Sigma_{\mathbb{A}}^{\texttt{rd}}}\cup[p]_{w};\end{array}}$$

$$\text{(Op-Nesp)}\ \frac{p_{\mathbb{A}}^{j}:\ r \coloneqq \texttt{op}(\vec{x}) \qquad r\neq\texttt{esp}}{\begin{array}{l}\underline{\texttt{if}}\ (\texttt{UB}_{\mathbb{A}}(\texttt{op},\vec{x}))\ \texttt{halt}(\mathscr{U});\\ r \coloneqq \texttt{op}(\vec{x});\end{array}}$$

$$\text{(Store}_{\mathbb{A}}\text{)}\ \frac{p_{\mathbb{A}}^{j}:\ \texttt{store}\ w,\ a,\ v,\ p}{\begin{array}{l}\underline{\texttt{if}}\ (\quad \neg\texttt{isAlignedIntrvl}_{a}(p,w)\\ \qquad \vee\ \texttt{ov}([p]_{w},\Sigma_{\mathbb{A}}^{\{\texttt{free}\}\cup G_{r}\cup F_{r}}\cup(\Sigma_{\mathbb{A}}^{cv}\setminus\Sigma_{\mathbb{A}}^{F\cup S})))\\ \quad \texttt{halt}(\mathscr{U});\\ M_{\mathbb{A}} \coloneqq \texttt{st}_{w}(M_{\mathbb{A}},p,v);\\ \boxed{\Sigma_{\mathbb{A}}^{\texttt{wr}}} \coloneqq \boxed{\Sigma_{\mathbb{A}}^{\texttt{wr}}}\cup[p]_{w};\end{array}}$$

Figure 2.6: Translation rules for converting pseudo-assembly instructions to graph instructions. op represents an arithmetic, logical, or relational operator.

as division-by-zero are also modeled as $\mathscr{U}$ errors in $\mathbb{A}$ through the abstract $\texttt{UB}_{\mathbb{A}}(\ldots)$ operation (used in rules (Op-esp) and (Op-Nesp)).

(Op-esp) (fig. 2.6) shows the translation of an instruction that updates the stackpointer register esp. An assignment to stackpointer esp may indicate allocation (stack push) or deallocation (stack pop) of stack space. An assignment that corresponds to a stackpointer decrement (push) is identified through predicate $\texttt{isPush}(p_{\mathbb{A}}^{j},\iota_b,\iota_a)$ where ι_b and ι_a are the values of esp *before* and *after* the execution of the instruction. We use thresholding on the update distance $(\iota_b-\iota_a)$ in our definition of isPush:[11]

$$\texttt{isPush}(p_{\mathbb{A}}^{j},\iota_b,\iota_a) \Leftrightarrow (\iota_b\neq\iota_a)\wedge((\iota_b-\iota_a)\leq_u(2^{31}-1))$$

[11] $2^{31}-1$ = INT_MAX in our 32-bit setting.

While this choice of `isPush` suffices for most TV settings, we show in chapter A that if the translation is performed by an adversarial compiler, discriminating a stack push from a pop is trickier and may require external trusted guidance from the user.

For a stackpointer decrement (a push), a failure to allocate stack space, either due to wraparound or overlap with other allocated space, triggers $\mathscr{W}$, i.e., we expect the environment (e.g., OS) to ensure that the required stack space is available to A to prevent a wraparound or overlap; however, an overlap with region cv is permitted — we defer a discussion on this exception to section 2.6. For a stackpointer increment (a pop), it is a translation error if the stackpointer moves out of current stack frame bounds (captured by error code $\mathscr{U}$). The stackpointer value at the end of an assignment instruction at PC p^j_{A} is saved in a ghost variable named $\boxed{\texttt{sp}.p^j_{\mathsf{A}}}$. These ghost variables help with inference of invariants that relate a local variable's bounds with stack addresses (invariant inference in our algorithm is discussed in section 4.2). During push, the initial contents of the newly allocated stack region are chosen non-deterministically using θ — this admits the possibility of arbitrary clobbering of the unallocated stack region below the stackpointer due to asynchronous external interrupts, before it is allocated again.

The (Op-Nesp) rule in fig. 2.6 gives translation for an instruction '$r \coloneqq \texttt{op}(\vec{x})$' that does not update the `esp` register ($r \neq \texttt{esp}$). The $\texttt{UB}_{\mathsf{A}}(\texttt{op}, \vec{x})$ operation abstracts the condition for a machine exception during execution of `op` (e.g., a zero second operand for division).

($\textsc{Entry}_{\mathsf{A}}$) (in fig. 2.7) shows the initialization of state elements of procedure A at entry. For a region $r \in B$[12], the initialization of address set Σ^r_{A} and memory region $\pi_{\Sigma^r_{\mathsf{A}}}(M_{\ddot{\mathsf{A}}})$ is same as ($\textsc{Entry}_{\mathsf{C}}$). For an assembly-only region $f \in F$, the address set Σ^f_{A} is initialized using $\mathbb{A}$'s symbol table (abstracted through $\texttt{addrSets}_F()$). The memory contents of a read-only global variable $r \in G_r \cup F_r$ are initialized using $\texttt{ROM}^r_{\mathsf{A}}(i^r_{\mathsf{A}})$ (defined in table 2.1) — recall that the validator verifies the equality of memory contents of a common read-only global $g \in G$ so that $\texttt{ROM}^g_{\mathsf{A}}(i^g_{\mathsf{A}}) = \texttt{ROM}^g_{\mathsf{C}}(i^g_{\mathsf{C}})$ for $i^g_{\mathsf{A}} = i^g_{\mathsf{C}}$.

The machine registers are initialized with arbitrary contents using θ. The x86 stack of an assembly procedure includes the stack frame $\Sigma^{stk}_{\mathsf{A}}$ of the currently executing procedure A, the parameters Σ^Y_{A} of A, and the remaining space which includes caller-stack Σ^{cs}_{A} and, possibly, the locals Σ^{cl}_{A} defined in the call chain of A. Due to the calling conventions, we assume (through `stkIsWF()`) that:

[12]Recall that B is the set of regions common to both C and A.

$$(\textsc{Entry}_{\mathsf{A}})\ \frac{p^{j}_{\mathsf{A}} : \texttt{def}\ \mathsf{A}(\vec{\tau})}{\begin{array}{l}
\Sigma^{hp}_{\mathsf{A}}, \Sigma^{cl}_{\mathsf{A}}, \Sigma^{cv}_{\mathsf{A}}, \ldots, i^{g}_{\mathsf{A}}, \ldots, i^{y}_{\mathsf{A}}, \ldots, \Sigma^{\mathtt{vrdc}}_{\mathsf{A}} := \mathtt{rd}(2^{\mathtt{i}_{32}}, 2^{\mathtt{i}_{32}}, \ldots, 2^{\mathtt{i}_{32}});\\
\ldots, \Sigma^{f}_{\mathsf{A}}, \ldots := \mathtt{addrSets}_{F}();\\
\Sigma^{\mathtt{rd}}_{\mathsf{A}}, \Sigma^{\mathtt{wr}}_{\mathsf{A}}, \ldots, \Sigma^{z}_{\mathsf{A}}, \ldots := \emptyset, \emptyset, \ldots, \emptyset, \ldots;\\
\underline{\texttt{if}}\ (\neg\mathtt{addrSetsAreWF}(\Sigma^{hp}_{\mathsf{A}}, \Sigma^{cl}_{\mathsf{A}}, \Sigma^{cv}_{\mathsf{A}}, \ldots, i^{g}_{\mathsf{A}}, \ldots, \Sigma^{f}_{\mathsf{A}}, \ldots, i^{y}_{\mathsf{A}}, \ldots, \Sigma^{\mathtt{vrdc}}_{\mathsf{A}}))\\
\quad \mathtt{halt}(\mathscr{W});\\
M_{\mathsf{A}} := \theta(\mathtt{i}_{32} \rightarrow \mathtt{i}_{8});\quad M_{\mathsf{A}} := \mathtt{upd}_{\Sigma^{B}_{\mathsf{A}} \setminus G_r}(M_{\mathsf{A}}, \mathtt{rd}(\mathtt{i}_{32} \rightarrow \mathtt{i}_{8}));\\
\underline{\texttt{for}}\ r\ \texttt{in}\ G_r \cup F_r\ \{\ M_{\mathsf{A}} := \mathtt{upd}_{i^{r}_{\mathsf{A}}}(M_{\mathsf{A}}, \mathtt{ROM}^{r}_{\mathsf{A}}(i^{r}_{\mathsf{A}}));\ \}\\
\underline{\texttt{for}}\ x\ \texttt{in}\ \overrightarrow{regs}\ \{\ x := \theta(\mathtt{T}(x));\ \}\\
\mathtt{stk}_{e} := \Sigma^{Y}_{\mathsf{A}} \neq \emptyset\ ?\ \mathtt{ub}(\Sigma^{Y}_{\mathsf{A}}) : \mathtt{esp} + 3_{\mathtt{i}_{32}};\\
\mathtt{cs}_{e} := \theta(\mathtt{i}_{32});\\
\underline{\texttt{if}}\ (\neg\mathtt{stkIsWF}(\mathtt{esp}, \mathtt{stk}_{e}, \mathtt{cs}_{e}, \vec{\tau}, \Sigma^{hp}_{\mathsf{A}}, \Sigma^{cl}_{\mathsf{A}}, \Sigma^{G\cup F}_{\mathsf{A}}, \ldots, i^{y}_{\mathsf{A}}, \ldots, \Sigma^{\mathtt{vrdc}}_{\mathsf{A}}))\\
\quad \mathtt{halt}(\mathscr{W});\\
\Sigma^{stk}_{\mathsf{A}} := [\mathtt{esp}, \mathtt{stk}_{e}] \setminus \Sigma^{Y}_{\mathsf{A}};\\
\Sigma^{cs}_{\mathsf{A}} := [\mathtt{stk}_{e} + 1_{\mathtt{i}_{32}}, \mathtt{cs}_{e}] \setminus \Sigma^{cl}_{\mathsf{A}};\\
\mathtt{sp}.entry := \mathtt{esp};\quad M^{cs} := \pi_{\Sigma^{cs}_{\mathsf{A}}}(M_{\mathsf{A}});\\
ebp, esi, edi, ebx, eip := \mathtt{ebp}, \mathtt{esi}, \mathtt{edi}, \mathtt{ebx}, \mathtt{sel}_{4}(M_{\mathsf{A}}, \mathtt{esp});\\
\underline{\texttt{for}}\ f\ \texttt{in}\ F\ \{\\
\quad \mathtt{sz}.f, \mathtt{em}.f, \mathtt{lb}.f, \mathtt{ub}.f := |\Sigma^{f}_{\mathsf{A}}|, |\Sigma^{f}_{\mathsf{A}}| = 0_{\mathtt{i}_{32}}, \mathtt{lb}(\Sigma^{f}_{\mathsf{A}}), \mathtt{ub}(\Sigma^{f}_{\mathsf{A}});\\
\}
\end{array}}$$

$$(\textsc{Ret}_{\mathsf{A}})\ \frac{p^{j}_{\mathsf{A}} : \texttt{ret}\ \tau}{\begin{array}{l}
\underline{\texttt{if}}\ (\quad \mathtt{sp}.entry \neq \mathtt{esp} \vee ebp \neq \mathtt{ebp}\\
\qquad \vee\ esi \neq \mathtt{esi} \vee edi \neq \mathtt{edi} \vee ebx \neq \mathtt{ebx}\\
\qquad \vee\ eip \neq \mathtt{sel}_{4}(M_{\mathsf{A}}, \mathtt{esp})\\
\qquad \vee \neg(M^{cs} =_{\Sigma^{cs}_{\mathsf{A}}} M_{\mathsf{A}}))\\
\quad \mathtt{halt}(\mathscr{U});\\
\mathtt{IF}\{\tau = \mathtt{void}\}\{\\
\quad \mathtt{wr}(\pi_{\Sigma^{B}_{\mathsf{A}}}(M_{\mathsf{A}}));\\
\}\ \mathtt{ELSE}\ \{\\
\quad \mathtt{wr}(\mathtt{ret}(\triangle_{\tau}(\mathtt{eax}, \mathtt{edx}), \pi_{\Sigma^{B}_{\mathsf{A}}}(M_{\mathsf{A}})));\\
\}\\
\mathtt{halt}(\emptyset);
\end{array}}$$

Figure 2.7: Translation rules for converting pseudo-assembly instructions to graph instructions.

1. The parameters are laid out at addresses above the stackpointer (`esp`) as per calling conventions (abstracted through `obeyCC()` in `stkIsWF()`).
2. The value $\mathtt{esp} + 4_{\mathtt{i}_{32}}$ (stackpointer value at the callsite in caller) is 16-byte aligned.
3. The callers' stack (region cs) is above A's stack frame and laid out parameters (if any).
4. Stack stk and callers' stack cs do not overlap with other allocated space.

A violation of these conditions trigger $\mathscr{W}$, i.e., we expect the environment to ensure that these conditions are satisfied. The ghost variables $\texttt{stk}_e$ and $\texttt{cs}_e$ represent the largest addresses in $\Sigma_{\mathsf{A}}^{Y\cup\{stk\}}$ and $\Sigma_{\mathsf{A}}^{Y\cup\{stk,cs,cl\}}$ respectively, so that at entry, $\Sigma_{\mathsf{A}}^{stk} = [\texttt{esp}, \texttt{stk}_e] \setminus \Sigma_{\mathsf{A}}^{Y}$ and $\Sigma_{\mathsf{A}}^{cs} = [\texttt{stk}_\texttt{e} + 1_{\texttt{i32}}, \texttt{cs}_e] \setminus \Sigma_{\mathsf{A}}^{cl}$. If there are no parameters, $\texttt{stk}_e = \texttt{esp} + 3_{\texttt{i32}}$ represents the end of the region that holds the return address of procedure A. The rule concludes with initialization of ghost variables for: stack pointer at entry ($\texttt{sp}.entry$); memory contents of the caller's stack (M^{cs}); return address of A (eip); callee-saved registers (ebp, esi, edi, ebx); and the address set of an assembly-only global variable $f \in F$ ($\texttt{sz}.f$, $\texttt{em}.f$, $\texttt{lb}.f$, $\texttt{ub}.f$).

Upon procedure-return (rule ($\textsc{Ret}_{\mathsf{A}}$) in fig. 2.7), we require, as per the calling conventions, that the return address, callers' stack and the callee-save registers remain preserved — a violation of these conditions trigger $\mathscr{U}$. We use the ghost variables, $\texttt{sp}.entry$, ebp, esi, edi, eip, and M^{cs}, set at procedure entry for this check. Validating the calling conventions at procedure return enable us to assume them at a procedure call. For simplicity, we only tackle scalar return values, and ignore aggregate return values that need to be passed in memory.

Notice that unlike region $r \in B$, region cv may potentially overlap with assembly-only regions $F \cup S$. Thus, while an address $\alpha \in \Sigma_{\mathsf{C}}^{cv}$ is inaccessible in C, it is potentially accessible in A if $\alpha \in F \cup S$. We explain the rationale for this in section 2.6.1 when we discuss virtual allocation.

2.4 Observable traces and Refinement Definition

Recall that a procedure $P \in \{\mathsf{C}, \mathsf{A}\}$ execution yields an observable trace T_P containing silent and non-silent events.

Definition 2.4.1 ($e(T)$)**.** *The error code of a trace T, written $e(T)$, is either $\emptyset$ (indicating either non-termination or error-free termination), or one of $\mathscr{r} \in \{\mathscr{U}, \mathscr{W}\}$ (indicating termination with error code $\mathscr{r}$).*

For $\mathscr{r} \in \{\mathscr{U}, \mathscr{W}\}$, we call a trace T *$\mathscr{r}$-terminating* iff $e(T) = \mathscr{r}$.

Definition 2.4.2 ($\tilde{e}(T)$)**.** *The non-error part of a trace T, written $\tilde{e}(T)$, is T when $e(T) = \emptyset$ and T' such that $T = T' \cdot e(T)$ otherwise.*

Definition 2.4.3 ($(P \downarrow_\Omega T)$)**.** *$(P \downarrow_\Omega T)$ denotes the condition that for an initial outside*

world Ω*, the execution of a procedure* P *may produce an observable trace* T *for some sequence of non-deterministic choices.*

A compiler must ensure that if A exhibits UB (Undefined Behavior) then there must exist a sequence of non-deterministic choices such that C also exhibits UB, i.e., $(\mathsf{A} \downarrow_\Omega T \cdot \mathscr{U})$ implies $(\mathsf{C} \downarrow_\Omega T' \cdot \mathscr{U})$. Further, for an error-free execution of A, C should either be able to produce a trace containing identical sequence of non-silent events or trigger $\mathscr{U}$ — the latter case admitting the *anything is permissible* clause of UB in C. We define a refinement relation between C and A that ensures these properties.

Definition 2.4.4 $(T =_{st} T')$**.** *Traces* T *and* T' *are* ***stuttering equivalent****, written* $T =_{st} T'$*, iff they differ only by finite sequences of silent events* $\bot$*.*

For example, $T =_{st} T'$ holds for $T = (\texttt{rd}(\ldots), \bot, \bot, \texttt{rd}(\ldots), \bot, \bot, \bot, \texttt{wr}(\ldots), \texttt{exit})$ and $T' = (\texttt{rd}(\ldots), \bot, \texttt{rd}(\ldots), \bot, \texttt{wr}(\ldots), \texttt{exit})$.

Definition 2.4.5 $(T \leq_{st} T')$**.** *A trace* T *is a* ***stuttering prefix*** *of trace* T'*, written* $T \leq_{st} T'$*, iff* $(T =_{st} T') \vee (\exists T_r : (T \cdot T^r) =_{st} T')$*.*

For example, $T \leq_{st} T'$ holds for $T = (\texttt{rd}(\ldots), \bot, \bot, \texttt{rd}(\ldots), \bot, \bot, \bot)$ and $T' = (\texttt{rd}(\ldots), \bot, \texttt{rd}(\ldots), \bot, \texttt{wr}(\ldots), \texttt{exit})$.

Definition 2.4.6 $(W^{\Omega,T_\mathsf{A}}_{\texttt{pre}}(\mathsf{C}))$**.** $W^{\Omega,T_\mathsf{A}}_{\texttt{pre}}(\mathsf{C})$ *denotes the condition:*

$$(e(T_\mathsf{A}) = \mathscr{W}) \wedge (\exists T_\mathsf{C} : (\mathsf{C} \downarrow_\Omega T_\mathsf{C}) \wedge (\tilde{e}(T_\mathsf{A}) \leq_{st} T_\mathsf{C}))$$

Definition 2.4.7 $(U^{\Omega,T_\mathsf{A}}_{\texttt{pre}}(\mathsf{C}))$**.** $U^{\Omega,T_\mathsf{A}}_{\texttt{pre}}(\mathsf{C})$ *denotes the condition:*

$$\exists T_\mathsf{C} : (\mathsf{C} \downarrow_\Omega T_\mathsf{C} \cdot \mathscr{U}) \wedge (T_\mathsf{C} \leq_{st} T_\mathsf{A})$$

Definition 2.4.8 $(\mathsf{C} \sqsupseteq \mathsf{A})$**.** $\mathsf{C} \sqsupseteq \mathsf{A}$*, read* A *refines* C *(or* C *is refined by* A*), iff:*

$$\begin{aligned} \forall \Omega : (\mathsf{A} \downarrow_\Omega T_\mathsf{A}) \Rightarrow \quad & W^{\Omega,T_\mathsf{A}}_{\texttt{pre}}(\mathsf{C}) \\ \vee\ & U^{\Omega,T_\mathsf{A}}_{\texttt{pre}}(\mathsf{C}) \\ \vee\ & \exists T_\mathsf{C} : (\mathsf{C} \downarrow_\Omega T_\mathsf{C}) \wedge (T_\mathsf{A} =_{st} T_\mathsf{C}) \end{aligned}$$

The definition of $\mathsf{C} \sqsupseteq \mathsf{A}$ admits three possibilities for an execution of A: either (1) A triggers $\mathscr{W}$ $(W^{\Omega,T_\mathsf{A}}_{\texttt{pre}}(\mathsf{C}))$, or (2) C triggers $\mathscr{U}$ $(U^{\Omega,T_\mathsf{A}}_{\texttt{pre}}(\mathsf{C}))$, or (3) C may produce a trace with identical sequence of non-silent events to the one produced by A $(T_\mathsf{A} =_{st} T_\mathsf{C})$.

In the first case, $W^{\Omega,T_\mathsf{A}}_{\mathtt{pre}}(\mathsf{C})$ encodes the condition that A terminates with error $\mathscr{W}$ ($e(T_\mathsf{A}) = \mathscr{W}$) and the sequence of non-silent events in the trace T_A produced by A (before terminating) is equivalent to trace T_C produced by C up till that point ($\tilde{e}(T_\mathsf{A}) \leq_{st} T_\mathsf{C}$) — we call the latter a *trace prefix requirement*. Recall that we do not care about the case where A terminates with $\mathscr{W}$ because we assume that the external environment (e.g., OS) will ensure that this event never occurs. For example, we do not care for the case where a stack space allocation may fail in A (Op-esp). Thus, in the example in fig. 2.1c, we assume that the stackpointer decrement instruction at `A5` successfully allocates stack space. The trace prefix requirement $\tilde{e}(T_\mathsf{A}) \leq_{st} T_\mathsf{C}$ caters to the situation where a callee procedure may not terminate: if a callee in A[13] does not terminate (before A could halt with $\mathscr{W}$), then the trace prefix requirement ensures that the corresponding callee in C will also not terminate (because identical trace events have been recorded for each procedure call till that point). For example, in fig. 2.1c, consider the case when the stackpointer manipulation instructions at `A20` in the assembly procedure $\mathsf{A}_{\mathtt{fib}}$ trigger $\mathscr{W}$[14]: it is possible that before $\mathscr{W}$ may be triggered at `A20`, the preceding call to `printf` never terminates (or encounters error resulting in termination of program execution). In this situation, $\mathsf{A}_{\mathtt{fib}}$ would not actually trigger $\mathscr{W}$ at runtime (because `A20` will never execute). Our execution semantics do not explicitly model the possibility of non-termination (or termination with error) of `printf` and so they make it appear that `A20` will be executed and so $\mathscr{W}$ will be triggered. Thus, it is not enough to simply ignore the case where $\mathscr{W}$ is triggered; we also need to ensure that all prior procedure calls have identical (potentially non-terminating or termination with error) behavior. The trace prefix requirement, $\tilde{e}(T_\mathsf{A}) \leq_{st} T_\mathsf{C}$, ensures this as it requires the traces due to a procedure call to be identical up till the point of error in A. If the trace prefix requirement was absent, we may unsoundly admit a translation that passes differing arguments to a procedure call (`printf` in fig. 2.1) in C and A but triggers $\mathscr{W}$ in A thereafter.

In the second case, $U^{\Omega,T_\mathsf{A}}_{\mathtt{pre}}(\mathsf{C})$ encodes the condition that C may terminate with error $\mathscr{U}$ (i.e., it may exhibit UB). Due to UB semantics, we therefore do not care about A's behavior in this situation. For example, in fig. 2.1b, for `v[*i]` at `I6`, we do not care for the out-of-bounds case when `*i` > (`*n` + 2). The stuttering prefix condition, $T_\mathsf{C} \leq_{st} T_\mathsf{A}$,

[13] We discuss the exact execution semantics of a procedure call in A in next section. For now, let's assume that the semantics are similar to C with `wr` instruction for over-approximately observing and `rd` for arbitrarily but deterministically mutating the callee-observable state.

[14] Strictly speaking, $\mathscr{W}$ will not be triggered in fig. 2.1c at `A20`, because the execution would have halted earlier in this situation (for this example). However, for argument's sake, assume the error may get triggered at the mentioned PC.

ensures that all procedure calls (before C exhibits UB) produce identical traces in both C and A, thus ensuring identical termination behavior of prior procedure calls (similar to the first case).

In the third case, C may produce a stuttering equivalent trace as A, and all non-silent events (including non-termination) will be identical.

In each of the cases, a stuttering requirement, stuttering prefix in the first two cases and stuttering equivalent in the third case, ensures that both procedures execute at similar speeds up till the termination of either. It is to be noted that stuttering requirement is not related to preservation of termination behavior — the latter is due to observation of termination through the trace-producing `halt` instruction. A terminating execution has a finite trace ending with an error code (for erroneous termination) or the special `exit` event (for error-free termination). Due to the silent trace event $\perp$ generated by execution of every instruction, a non-terminating execution has an infinite trace.

In the absence of local variables and procedure calls in C, $\mathsf{C} \sqsupseteq \mathsf{A}$ implies a correct translation from C to A.

In our work, $\mathsf{C} \sqsupseteq \mathsf{A}$ is our definition of a *correct translation*. In subsequent sections, this formal definition will become more sophisticated, as it will involve annotating the assembly program with custom instructions introduced by us. These formal definintions, which include the custom instructions introduced by us, have been carefully constructed so that they capture the notion of observable equivalence as people usually understand it. We expect that these definitions are succinct and simple enough so that the reader can convince herself of their validity. We help the reader by introducing these formal definitions incrementally, with supporting explanations.

2.5 Refinement Definition in the presence of local variables and procedure calls when all local variables are allocated on the stack in A

For each local variable (de)allocation and for each procedure call, our execution semantics generate a `wr` trace event in C (fig. 2.5). Thus, to reason about refinement, we require correlated and equivalent trace events to be generated in A. For this, we annotate A with two types of annotations to obtain $\dot{\mathsf{A}}$:

1. *Special* $\texttt{alloc}_s$ *and* $\texttt{dealloc}_s$ *instructions:* $\texttt{alloc}_s$ and $\texttt{dealloc}_s$ instructions are added to explicitly indicate the (de)allocation of a local variable $z \in Z$ from stack. An address interval in the stack region may be marked as belonging to z through the $\texttt{alloc}_s$ instruction and subsequently marked as belonging to stack again through the $\texttt{dealloc}_s$ instruction.

2. *Annotations to assembly procedure-call instruction:* A procedure call is annotated with the types and addresses of the arguments and the set of memory regions observable by the callee. These annotations augment the assembly call instruction such that its semantics can be defined in a manner similar to the `call` instruction in C (section 2.3.1).

These annotations are intended to encode the correlations with the corresponding allocation, deallocation, and procedure-call events in the source procedure C. For now, we assume that the locations and values of these annotations in Ȧ are coming from an oracle — later in chapter 4, we present an algorithm to identify these annotations automatically in a best-effort manner.

Figure 2.8 presents the translations of the three new assembly instructions — $\texttt{alloc}_s$, $\texttt{dealloc}_s$, and `call` — to graph instructions.

2.5.1 (De)Allocation indicating $\texttt{alloc}_s$ and $\texttt{dealloc}_s$ instructions

An instruction '$p^j_{\dot{A}}$: $\texttt{alloc}_s$ e_v, e_w, a, z' represents the *stack allocation* of a local variable identified by allocation site z (fourth argument) at PC $p^j_{\dot{A}}$. e_v is the expression for the start address, e_w is the expression for the allocation size, and a is the required alignment of the start address. (AllocS) in fig. 2.8 presents the translation of $\texttt{alloc}_s$ to graph instructions. During the stack allocation of z, the allocated interval $i = [v, v + w - 1_{\mathtt{i32}}]$, identified by start address $v = e_v$ and allocation size $w = e_w$, must satisfy the required well-formedness (WF) constraints for separation and alignment, or else $\mathcal{U}$ is triggered. Recall that $\mathcal{U}$ is used for signaling translation errors in A, so a correctly translated assembly procedure must never falsify the separation and alignment constraints for i. A stack allocation removes i from the stack address set $\Sigma^{stk}_{\dot{A}}$ and adds it to the address set $\Sigma^{z}_{\dot{A}}$ of z. Thus, the separation WF constraint requires i to lie entirely within $\Sigma^{stk}_{\dot{A}}$ (encoded through $\texttt{intrvlInSet}_a(\ldots, \Sigma^{stk}_{\dot{A}})$ in fig. 2.8). Further,

$$(\textsc{AllocS})\ \frac{p^j_{\dot{A}} : \texttt{alloc}_s\ e_v, e_w, a, z}{\begin{array}{l}\texttt{wr}(\texttt{allocBegin}(z, e_w, a));\\ v, w \coloneqq e_v, e_w;\\ \underline{\texttt{if}}\ (\neg\texttt{intrvlInSet}_a(v, v+w-1_{\texttt{i}_{32}}, \Sigma^{stk}_{\dot{A}}))\\ \quad \texttt{halt}(\mathscr{U});\\ \underline{\texttt{if}}\ (\texttt{ov}([v]_w, \Sigma^{cv}_{\dot{A}}))\\ \quad \texttt{halt}(\mathscr{W});\\ \Sigma^z_{\dot{A}}, \Sigma^{stk}_{\dot{A}} \coloneqq \Sigma^z_{\dot{A}} \cup [v]_w, \Sigma^{stk}_{\dot{A}} \setminus [v]_w;\\ \texttt{wr}(\texttt{allocEnd}(z, [v]_w, \pi_{[v]_w}(M_{\dot{A}})));\end{array}}$$

$$(\textsc{DeallocS})\ \frac{p^j_{\dot{A}} : \texttt{dealloc}_s\ z}{\begin{array}{l}\Sigma^z_{\dot{A}}, \Sigma^{stk}_{\dot{A}} \coloneqq \emptyset, \Sigma^{stk}_{\dot{A}} \cup \Sigma^z_{\dot{A}};\\ \texttt{wr}(\texttt{dealloc}(z));\end{array}}$$

$$(\textsc{Call}_{\dot{A}})\ \frac{p^j_{\dot{A}} : \texttt{call}\ \gamma\ \rho(\vec{\tau}\ \vec{x})\ \beta^*}{\begin{array}{l}\underline{\texttt{if}}\ (\neg\texttt{aligned}_{16}(\texttt{esp}) \vee \neg\texttt{obeyCC}(\texttt{esp}, \vec{\tau}, \vec{x}))\\ \quad \texttt{halt}(\mathscr{U});\\ \texttt{wr}(\texttt{fcall}(\rho, \vec{x}, \beta^*, \pi_{\Sigma^{\beta^*}_{\dot{A}}}(M_{\dot{A}})));\\ M_{\dot{A}} \coloneqq \texttt{upd}_{\Sigma^{\beta^*}_{\dot{A}} \setminus G_r}(M_{\dot{A}}, \texttt{rd}(\texttt{i}_{32} \rightarrow \texttt{i}_8));\\ \texttt{ecx} \coloneqq \theta(\texttt{i}_{32});\\ \texttt{IF}\{\gamma = \texttt{void}\}\{\\ \quad \texttt{eax}, \texttt{edx} \coloneqq \theta(\texttt{i}_{32}, \texttt{i}_{32});\\ \}\ \texttt{ELSE}\ \{\\ \quad \texttt{eax}, \texttt{edx} \coloneqq \nabla_\gamma(\texttt{rd}(\gamma));\\ \}\end{array}}$$

Figure 2.8: Additional translation rules for converting pseudo-assembly instructions to graph instructions for procedures with only stack-allocated locals.

the allocated interval must be separate from region cv[15], otherwise $\mathscr{W}$ is triggered; we explain the rationale for triggering $\mathscr{W}$ in this case in the next section when we discuss virtual allocation.

An instruction '$p^j_{\dot{A}}$: $\texttt{dealloc}_s\ z$' represents the deallocation of z and empties the address set $\Sigma^z_{\dot{A}}$, adding the removed addresses to $\Sigma^{stk}_{\dot{A}}$. This action reverses the transfer from the stack to z performed by execution of an $\texttt{alloc}_s$ instruction. (DeallocS) in fig. 2.8 shows the translation of $\texttt{dealloc}_s$ to graph instructions.

The $\texttt{alloc}_s$ and $\texttt{dealloc}_s$ instructions do not change the accessible address set $\Sigma^{B \cup F \cup S}_{A}$ of $\dot{A}$: $\texttt{alloc}_s$ transfers addresses from stk to z and $\texttt{dealloc}_s$ transfers them back to stk. Thus, no "real allocation"[16] is performed, instead a subset of the stack frame of $\dot{A}$ is identified distinctly for the purpose of validation.

[15] Recall that cv may potentially overlap with stk unlike a region $r \in B$.

[16] In the sense of stack allocation through stackpointer decrement in A or local allocation through `alloc` in C.

Similar to `alloc` and `dealloc` in C, $\texttt{alloc}_s$ and $\texttt{dealloc}_s$ in Å produce non-silent trace events via `wr` instructions. Both use the same value constructors, so that identical observables (passed as arguments) produce identical trace events.

Figure 2.1c shows an assembly procedure annotated with $\texttt{alloc}_s$ instruction at line $\texttt{A5}^1$ and $\texttt{dealloc}_s$ instruction at line $\texttt{A19}^1$. The start address of allocation is `esp`, the allocated size is `4*(eax+2)`, and the required alignment on start address is 4. It can be observed that the stack allocated interval meets the WF constraints in this case: the preceding instruction `A5` allocates at least `4*(eax+2)` bytes on stack and aligns `esp` by 16. The $\texttt{alloc}_s$ instruction uses the allocation site `I2` from the IR procedure in fig. 2.1b for identifying the allocated interval. The $\texttt{dealloc}_s$ instruction at $\texttt{A19}^1$ refers to the same allocation site `I2` for transferring the allocated addresses due to execution of $\texttt{alloc}_s$ back to stack.

2.5.2 Annotated procedure-call instruction

We annotate an assembly procedure-call instruction '$p^j_{\text{Å}}$: `call` ρ' for a call to callee ρ as '$p^j_{\text{Å}}$: `call` γ $\rho(\vec{\tau}\ \vec{x})$ β^*' to explicitly specify:

- The start addresses $\vec{x}$ of the address regions belonging to the arguments.
- The types $\vec{\tau}$ of the arguments and the return type γ.
- The callee-observable regions β^*.

The address region of an argument should have previously been demarcated using an $\texttt{alloc}_s$ instruction to match the `alloc` instructions for allocating arguments to a procedure call in C. Additionally, these address regions should satisfy the constraints imposed by the calling conventions — represented through `obeyCC()` in rule ($\textsc{Call}_{\text{Å}}$) of fig. 2.8. The calling conventions also require the stackpointer `esp` to be 16-byte aligned. A failure to meet the calling conventions requirements trigger $\mathcal{U}$ to indicate a translation error. A procedure call is recorded as an observable event, along with the observation of the callee name (or address) ρ, the addresses of the arguments $\vec{x}$, the callee-observable regions and their memory contents (β^* and $\pi_{\Sigma^{\beta^*}_{\text{Å}}}(M_{\text{Å}})$), same as ($\textsc{Call}_{\text{C}}$). The returned values, modeled through $\texttt{rd}(\texttt{i}_{32} \rightarrow \texttt{i}_8)$ and $\texttt{rd}(\gamma)$, include the contents of the callee-observable memory regions and the scalar values returned by the callee (in registers `eax`, `edx`). The callee additionally clobbers the caller-save registers (`eax`, `ecx`, `edx`) using θ. We use the translation-selecting $\texttt{IF}\{\gamma = \texttt{void}\}\{\ldots\}\texttt{ELSE}\{\ldots\}$ construct in ($\textsc{Call}_{\text{Å}}$) for selecting translation, whether to clobber or read, based on

return type γ.

Notice that for an annotated call instruction in $\dot{\mathsf{A}}$ to produce identical observables as a call instruction in C, it must not only have identical callee but identical argument addresses, callee-observable regions, and memory contents of callee-observable regions — the latter three being determined by the annotations. While the address regions, identified from $\vec{\tau}$ $\vec{x}$, are validated against the calling conventions (through `obeyCC()`), we do not "validate" the callee-observable regions β^* and the return type γ, both of which are consequential for defining the semantics of `call`. Instead, we rely on correlation with corresponding `call` instruction in C: an instruction `call` ρ in $\dot{\mathsf{A}}$, say $\texttt{call}_{\dot{\mathsf{A}}}$, with stack-allocated arguments that match in address and memory contents with instruction `call` $\rho \ldots$ in C, say $\texttt{call}_{\mathsf{C}}$, is *assumed* to have identical mutation behavior, i.e., $\texttt{call}_{\dot{\mathsf{A}}}$ mutates $\dot{\mathsf{A}}$'s state identically as $\texttt{call}_{\mathsf{C}}$ mutates C's state. In other words, the callee-observable regions (and return value) of $\texttt{call}_{\mathsf{A}}$ in A are determined based on the behavior of a correlated $\texttt{call}_{\mathsf{C}}$ in C (with identical arguments).

Figure 2.1c shows an annotated `call` instruction at line `A18`; the annotated callee-observable regions $\{hp, cl, \texttt{I9}, \texttt{I10}\}$ indicate that `printf` may potentially access and mutate the heap (hp), accessible callers' locals (cl), and the stack subregion corresponding to the two arguments (`I9` and `I10`) but not the rest of the stack.

2.5.3 Refinement Definition with only stack-allocated locals and procedure calls

With the new instructions and annotations enabling presence of stack-allocated locals and procedure calls in A, we define refinement between C and A in terms of existence of an annotated $\dot{\mathsf{A}}$ such that $\dot{\mathsf{A}}$ refines C:

Definition 2.5.1 (Refinement with only stack-allocated locals and procedure calls)**.** $\mathsf{C} \dot{\sqsupseteq} \mathsf{A}$ *iff:* $\exists \dot{\mathsf{A}} : \mathsf{C} \sqsupseteq \dot{\mathsf{A}}$

$\mathsf{C} \dot{\sqsupseteq} \mathsf{A}$ encodes the property that it is possible to annotate A to obtain $\dot{\mathsf{A}}$ so that the local variable (de)allocation and procedure-call events of C and the annotated $\dot{\mathsf{A}}$ can be correlated in lockstep.

Soundness of an annotation

It must not be possible to annotate A to produce Ȧ such that C ⊒ Ȧ holds but the two procedures (C and Ȧ) have different observable behavior. Our annotated instructions are carefully constructed and generate observable events such that the refinement definition will never admit an incorrect translation. An informal argument in favor of soundness of Ȧ is as follows. An assembly procedure Ȧ, produced by annotating A with $\texttt{alloc}_s$, $\texttt{dealloc}_s$, and `call` instructions, may have executions that are not present in the unannotated procedure A such that these additional executions either terminate with error $\mathscr{U}$ (due to $\texttt{alloc}_s$ and `call`) or terminate with error $\mathscr{W}$ (due to $\texttt{alloc}_s$). We consider each case of error separately below.

- If Ȧ terminates with error $\mathscr{U}$, due to either $\texttt{alloc}_s$ or `call` instruction, producing a trace $T_{\dot{A}}$, then C ⊒ Ȧ requires C to have an execution with trace T_C such that either T_C and $T_{\dot{A}}$ are stuttering equivalent ($T_C =_{st} T_{\dot{A}}$ holds), i.e., C also terminates with $\mathscr{U}$, or T_C ends with $\mathscr{U}$ and the non-error part of T_C is stuttering prefix of $T_{\dot{A}}$ ($\tilde{e}(T_C) \leq_{st} T_{\dot{A}}$ holds). In both cases, C must exhibit UB before Ȧ does.
- If Ȧ terminates with error $\mathscr{W}$, producing a trace $T_{\dot{A}}$, then C ⊒ Ȧ holds if there exists an execution of C with trace T_C such that C produces identical sequence of observables as Ȧ before the latter's termination ($\tilde{e}(T_{\dot{A}}) \leq_{st} T_C$).
  Recall that Ȧ's execution terminates with error $\mathscr{W}$ due to $\texttt{ov}([v]_w, \Sigma^{cv}_{\dot{A}})$ condition in (AllocS) (fig. 2.8). Before this, the execution must have produced an observable event through `wr(allocBegin(...))` that must be present in T_C as well. This indicates that C also executed an `alloc` instruction ((Alloc) in fig. 2.5) with identical allocation size, alignment, and region identifier (all part of observables produced through `allocBegin`). Further, because the execution of Ȧ did not trigger $\mathscr{U}$, the region to be allocated, $[v]_w$, belonged to stack $\Sigma^{stk}_{\dot{A}}$ (`intrvlInSet()` check). The execution semantics of Ȧ prohibit overlap of region stk with B and because observable events in T_C and $T_{\dot{A}}$ match till occurrence of error, $[v]_w$ must belong to $\texttt{comp}(\Sigma^B_C) \cup \Sigma^{cv}_C$. Consequently, there exists a choice of interval $[\alpha_b, \alpha_e]$ such that the execution of C also triggers $\mathscr{W}$ and terminates. Thus, it is possible for C and Ȧ to produce identical observables.

Notice that the exchange of addresses between $\Sigma^{stk}_{\dot{A}}$ and $\Sigma^{z}_{\dot{A}}$ (for $z \in Z$) in $\texttt{alloc}_s$ and $\texttt{dealloc}_s$ instructions does not affect the "out-of-bounds" access checks in (Load_A) and (Store_A). For an error-free execution of $\texttt{alloc}_s$, $\texttt{dealloc}_s$, and `call` in Ȧ, C ⊒ Ȧ will require identical observable events, `allocBegin` and `allocEnd` for $\texttt{alloc}_s$, `dealloc`

for $\texttt{dealloc}_s$, and `fcall` for `call` to be produced in C as well. This concludes our soundness argument.

In the presence of stack-allocated local variables and procedure calls, $\mathsf{C} \dot{\sqsupseteq} \mathsf{A}$ implies a correct translation from C to A. In the absence of local variables and procedure calls, $\mathsf{C} \dot{\sqsupseteq} \mathsf{A}$ reduces to $\mathsf{C} \sqsupseteq \mathsf{A}$ with $\dot{\mathsf{A}} = \mathsf{A}$.

2.5.4 Capabilities and Limitations of $\mathsf{C} \dot{\sqsupseteq} \mathsf{A}$

Limitation on relative order of (de)allocation and procedure calls

Because our logical encoding observes each (de)allocation event (due to the `wr` instruction), a fundamental limitation of $\mathsf{C} \dot{\sqsupseteq} \mathsf{A}$ is that for allocations and procedure calls that *reuse the same stack space*, their relative order remains preserved. This requirement is sound but may be too strict for certain (arguably rare) compiler transformations that may reorder the (de)allocation instructions that reuse the same stack space.

Figure 2.9 shows an example of such a transformation where stack stack is reused between allocations and procedure calls. The hypothetical assembly procedure $\mathsf{A}_{\texttt{baz}}$ (shown in fig. 2.9b) is a correct translation[17] of the C procedure $\mathsf{C}_{\texttt{baz}}$ (in fig. 2.9a) but will not be admitted under $\mathsf{C} \dot{\sqsupseteq} \mathsf{A}$. This is because the relative order of allocation of the variable `x` and the procedure call `foo` is not preserved in the transformation from $\mathsf{C}_{\texttt{baz}}$ to $\mathsf{A}_{\texttt{baz}}$. In the latter, the stack allocation of the local `x` is performed after the procedure call to `foo`. The stack deallocation at `A3` prohibits an earlier placement of $\texttt{alloc}_s$, making an annotation $\dot{\mathsf{A}}_{\texttt{baz}}$ that meets $\mathsf{C}_{\texttt{baz}} \sqsupseteq \dot{\mathsf{A}}_{\texttt{baz}}$ impossible. Notice that the stack subregion used for allocating `x` was previously potentially used by `foo`. Figure 2.9c shows another possible compilation (generated by an optimizing compiler) of $\mathsf{C}_{\texttt{baz}}$ in which case an annotation is possible and refinement can be established.

Limited handling of interprocedural transformations

Recall that our translation rules for C and A associate production of a non-silent observable trace event with a procedure-call instruction (figs. 2.5 and 2.8). The name (or address) of the callee is also observed. Thus, an interprocedural transformation which eliminates a procedure call, inlines it (including partial inlining), specializes it (through procedure cloning), transforms the control-flow (e.g., tail-recursion elimination), uses

[17] As per the C standard and calling conventions for 32-bit

```
int baz()
{
  int x; // alloc
  foo();
  return bar(&x);
}
```

(a) C program with address-taken local

```
A0:  baz:
A1:    esp -= 12     ; for alignment
A2:    call foo
A3:    esp += 12     ; undo -12 above
A4:    esp -= 8      ; alloc 'x'
A5:    push (esp+4)  ; setup &x
A6:    call bar
A7:    esp += 12
A8:    ret
```

(b) Hypothetical (abstracted) 32-bit x86 assembly

```
baz:
  esp -= 12  ; alloc 'x'
  call foo
  mem4[esp] = esp+8
  call bar
  esp += 12
  ret
```

(c) Compiler generated (abstracted) 32-bit assembly

Figure 2.9: Example of transformation where relative order of (de)allocations and procedure calls is not preserved. The refinement definition will not admit the hypothetical assembly but will admit the compiler generated one.

a different calling convention (e.g., if the called procedure is not externally visible then compiler may pass some arguments through registers instead of stack), reuses the arguments of the caller procedure (so that $\texttt{alloc}_s$ annotation is not possible for the callee arguments), or performs some other transformation that makes use of information not encoded in the semantics of C and A will not be admitted.

Key transformations admitted under C $\dot{\sqsupseteq}$ A:

Merging of multiple allocations: C $\dot{\sqsupseteq}$ A supports *merging* of multiple allocations into a single stackpointer decrement instruction. Let p^s_{A} be the PC of a single stackpointer decrement instruction that implements multiple allocations. Merging can be encoded by adding multiple $\texttt{alloc}_s$ instructions to A, in the same order as they appear in C, to obtain $\dot{\mathsf{A}}$, so that these $\texttt{alloc}_s$ instructions execute only after p^s_{A} executes. The C and assembly code fragments below illustrate the construction (C on the left and abstracted assembly on the right):

```
int x, y, z
// alloc 'x'
// alloc 'y'
// alloc 'z'
```

```
esp -= 20
alloc_s 'x'
alloc_s 'y'
alloc_s 'z'
```

Similarly, the corresponding $\texttt{dealloc}_s$ instructions must execute before a stackpointer

increment instruction deallocates this stack space.

CompCert[29]'s preallocation is a special case of merging where stack space for all local variables and procedure call arguments is allocated in the assembly procedure's prologue and deallocated in the epilogue with no reuse of stack space. In this case, our approach annotates A with $\texttt{alloc}_s$ and $\texttt{dealloc}_s$ instructions, potentially in the middle of the procedure body, such that they execute in lockstep with the allocations and deallocations in C.

Reallocation of stack space: A compiler may *reallocate* stack space by reusing the same space for two or more local variables with non-overlapping lifetimes (potentially without an intervening stackpointer increment instruction). If the relative order of (de)allocations is preserved, reallocation can be encoded by annotating A with a $\texttt{dealloc}_s$ instruction (for deallocating the first variable) immediately followed by an $\texttt{alloc}_s$ instruction, such that the allocated region potentially overlaps with the previously deallocated region. The C fragment (on the left) and corresponding assembly code fragment (on the right) below illustrate the construction:

```
{                                  L0:  esp -= 4
  int x; // alloc                       alloc_s 'x'
  ...                              L1:  edi = esp
  // dealloc                             ...        ; esp preserved
}                                        dealloc_s 'x'
{                                        alloc_s 'y'
  int y; // alloc                  L2:  edi = esp ; same value as L1 above
  ...                                   ...
  // dealloc                       L3:  dealloc_s
}
```

The variables `x` and `y` have non-overlapping lifetimes. Common stack space for both `x` and `y` is allocated at location `L0` and then later used by `x` between locations `L1` and `L2` and used by `y` between locations `L2` and `L3`.

Our refinement definition may not be able to cater to a translation that changes the relative order of (de)allocation instructions during reallocation. The following C source and assembly pair demonstrate an incompleteness example[18]:

[18] See https://godbolt.org/z/6rMT7z5re for a GCC compilation.

```
int x, y; // alloc 'x'; alloc 'y'
if (...) {
  /* use only 'x' */
} else {
  /* use only 'y' */
}
// dealloc 'y'; dealloc 'x'
```

```
L0: esp -= 4  ; alloc stack space
    ...
    if (...) jmp L2
L1: edi = esp ; use as 'x'
    ...
L2: edi = esp ; use as 'y'
    ...
```

Evidently, the variables `x` and `y` do not have non-overlapping lifetimes. However, their uses are limited to disjoint scopes. In the generated assembly, both share the same stack space allocated at `L0`. In contrast to the previous example, a valid annotation is impossible in this scenario because the same stack space cannot be allocated twice (separation WF constraint).

Dynamic allocations: C ⪳ A notably supports *dynamic allocations*, a capability missing in CompCert[29] due to its preallocation strategy, which performs all allocation in the assembly procedure's prologue. Dynamic allocations, necessary for enabling variable-length arrays (VLA) and `alloca()`, and used for allocating procedure-call arguments by most production compilers, allow variable-sized local allocations (similar to `malloc()`) that are automatically deallocated at the end of scope (for variable declarations) or at the end of containing procedure (for `alloca()` allocations). Our modeling does not require any special handling of dynamic allocations. The C source and assembly pair below present an example of dynamic (de)allocation using VLA in a loop:

```
for (i = 1; i < n; ++i)
{
  int v[i];
  // alloc (4*i), int, ...
  ...
  // dealloc 'v'
}
```

```
    eax = 4          ; = 4*i (i = 1)
L0: edi = esp        ; save 'esp'
    esp = esp - eax  ; variable decrement
    alloc_s esp, (4 * eax), ...
    ...
    dealloc_s
    esp = edi        ; restore 'esp'
    eax = eax+4      ; corresponds to i++
```

In the C source, VLA `v` is declared to have `i` elements, where `i` is the for-loop index. Thus, `v` is allocated at the start of every loop body execution and deallocated at the end. In the assembly, the register `eax` holds value corresponding to `4*i`. At the start of

loop body (location `L0`), current stackpointer `esp` is saved in `edi` and stack allocation of size `eax` is performed. At the end of loop body, `esp` is restored using `edi` and `eax` is incremented by 4. The annotations are made just before (after) the allocation (deallocation).

Lastly, C ⊒ A admits intermittent register-allocation of (parts of) a local variable, but not complete register-allocation or elimination. We address this limitation in the next section.

2.6 Refinement in the presence of potentially register-allocated or eliminated local variables in A

If a local variable $z \in Z$ is either register-allocated or eliminated in A, there may not exist a region in stack of A that can be associated with z. However, recall that our execution model observes each allocation event in C through the `wr` instruction. Thus, for a successful refinement check, a correlated allocation event still needs to be annotated in A. It may not be possible to use an $\texttt{alloc}_s$ instruction for this annotation, as $\texttt{alloc}_s$ requires us to specify a region in stack and such a region may not be available[19]. To tackle this, we *pretend* that a correlated allocation occurs in A by introducing the notion of a *virtual allocation* instruction, called $\texttt{alloc}_v$, in A. An $\texttt{alloc}_v$ instruction allocates a virtual region in A and a $\texttt{dealloc}_v$ instruction deallocates it.

Figure 2.10 shows the graph translations of the virtual (de)allocation instructions $\texttt{alloc}_v$ and $\texttt{dealloc}_v$ and fig. 2.11 shows the revised translations of other assembly instructions to incorporate the notion of virtual allocations. We update and annotate A with the translations and instructions in figs. 2.8, 2.10 and 2.11 to obtain Ä.

2.6.1 Virtual (de)allocations through $\texttt{alloc}_v$ and $\texttt{dealloc}_v$ instructions

An instruction '$p^j_{\text{Ä}} : \; v \coloneqq \texttt{alloc}_v \; e_w, a, zl$' non-deterministically chooses the start address (using choose instruction $\theta(\texttt{i}_{32})$) of a local variable zl of size e_w and required alignment a, performs a virtual allocation, and returns the start address in v. (AllocV)

[19]We use *may not* here instead of *is not* because in some situations, due to alignment requirements, it may be possible to find a large enough region in stack that is separate from other allocations but is not *used* for storing the local. We present an example of this later.

(AllocV)
$$\frac{p^j_{\ddot{A}} : \; v \coloneqq \texttt{alloc}_v \; e_w, \; a, \; zl}{\begin{array}{l}\texttt{wr}(\texttt{allocBegin}(zl, e_w, a));\\ v, w \coloneqq \theta(\texttt{i}_{32}), e_w;\\ \underline{\texttt{if}} \; (\neg \texttt{intrvlInSet}_a(v, v + w - 1_{\texttt{i}_{32}}, \texttt{comp}(\Sigma^{B \cup \{cv\}}_{\ddot{A}})))\\ \quad \texttt{halt}(\mathscr{W});\\ \Sigma^{zl}_{\ddot{A}}|^v \coloneqq \Sigma^{zl}_{\ddot{A}}|^v \cup [v]_w;\\ \texttt{wr}(\texttt{allocEnd}(zl, [v]_w, \pi_{[v]_w}(M_{\ddot{A}})));\end{array}}$$

(DeallocV)
$$\frac{p^j_{\ddot{A}} : \; \texttt{dealloc}_v \; zl}{\begin{array}{l}\underline{\texttt{if}} \; (\Sigma^{zl}_{\ddot{A}}|^s \neq \emptyset)\\ \quad \texttt{halt}(\mathscr{U});\\ \Sigma^{zl}_{\ddot{A}}|^v \coloneqq \emptyset;\\ \texttt{wr}(\texttt{dealloc}(zl));\end{array}}$$

Figure 2.10: Translation rules for converting the $\texttt{alloc}_v$ and $\texttt{dealloc}_v$ instructions instructions to graph instructions.

in fig. 2.10 shows the graph translation of $\texttt{alloc}_v$. The chosen start address v, together with interval e_w, is *assumed* to satisfy the desired WF constraints of separation (no overlap) and alignment (through $\texttt{intrvlInSet}_a(\ldots)$); error $\mathscr{W}$ is triggered otherwise. Notice that this is in contrast to $\texttt{alloc}_s$ where error $\mathscr{U}$ is triggered on WF violation to indicate that it is the compiler's responsibility to ensure the satisfaction of WF constraints. Unlike a stack allocation where the compiler chooses the allocated region (and the validator identifies it through an $\texttt{alloc}_s$ annotation), a virtual allocation is only a validation construct (the compiler is not involved) that is used only to enforce a lockstep correlation of allocation events. By triggering $\mathscr{W}$ on a failure during a virtual allocation, we effectively assume that allocation through $\texttt{alloc}_v$ satisfies the required WF conditions.

We put two restrictions on an $\texttt{alloc}_v$ annotation to keep our automatic algorithm (for construction of a witness of refinement) simple and tractable.

1. **$\texttt{alloc}_v$ restricted to local variable declarations**

   We support virtual allocations only for a variable declaration $zl \in Z_l$. Thus, we expect a call to `alloca()` at allocation site $za \in Z_a$ to always be stack-allocated in Ä.

   In our modeling of Ä, we replace the single variable $\Sigma^{zl}_{\ddot{A}}$ for address set of zl with two variables $\Sigma^{zl}_{\ddot{A}}|^s$ and $\Sigma^{zl}_{\ddot{A}}|^v$ that represent the address sets corresponding to the stack-allocations and virtual-allocations due to allocation site zl respectively. We compute $\Sigma^{zl}_{\ddot{A}} = \Sigma^{zl}_{\ddot{A}}|^s \cup \Sigma^{zl}_{\ddot{A}}|^v$ but do not maintain a separate variable $\Sigma^{zl}_{\ddot{A}}$. For convenience, we define $\Sigma^{Z_l}_{\ddot{A}}|^v = \bigcup_{zl \in Z_l}(\Sigma^{zl}_{\ddot{A}}|^v)$. In (AllocV) and (DeallocV), we use zl (instead of z) and a virtual (de)allocation updates the address set $\Sigma^{zl}_{\ddot{A}}|^v$ (instead of $\Sigma^{zl}_{\ddot{A}}$).

2. **A local variable may either be stack-allocated or virtual-allocated, not both**

   We do not tackle path-specializing transformations that may require, for a single variable declaration zl, a stack-allocation on one assembly path and a virtual-allocation on another. Thus, we assume that a variable declaration zl in C may either correlate with only stack-allocations (through $\texttt{alloc}_s$) or only virtual-allocations (through $\texttt{alloc}_v$) in Ä, i.e., $\Sigma^{zl}_{Ä}|^s \cap \Sigma^{zl}_{Ä}|^v = \emptyset$ holds at all times and if $\Sigma^{zl}_{Ä}|^s \neq \emptyset$ (resp. $\Sigma^{zl}_{Ä}|^v$) at any point, then $\Sigma^{zl}_{Ä}|^v = \emptyset$ (resp. $\Sigma^{zl}_{Ä}|^s$) holds throughout Ä's execution. This assumption simplifies the SMT encoding of proof obligations generated by our algorithm.

   Note that while this restriction may appear quite constraining at first glance, it does not impose substantial practical limitations. This is because if a variable zl is *not* address-taken, then it can be virtually-allocated, as its address is never observed (this is possible even if zl is allocated on stack by the compiler, i.e., there exists a region in stack of A that can be associated with zl).

We discuss the limitations arising from these restrictions in detail in section 2.6.3.

An instruction '$p^j_{Ä}$: $\texttt{dealloc}_v$ zl' in Ä empties the address set $\Sigma^{zl}_{Ä}|^v$ and produces an observable event through `wr` instruction. (DeallocV) in fig. 2.10 shows the graph translation of $\texttt{dealloc}_v$. An execution of $\texttt{dealloc}_v$ where $\Sigma^{zl}_{Ä}|^s$ is non-empty triggers error $\mathcal{U}$, i.e., we require an error-free execution of $\texttt{dealloc}_v$ to "empty" the address set $\Sigma^{zl}_{Ä}$ (defined as $\Sigma^{zl}_{Ä} = \Sigma^{zl}_{Ä}|^s \cup \Sigma^{zl}_{Ä}|^v$). Thus, we ensure the emptiness of $\Sigma^{zl}_{Ä}$ before producing the observable trace for deallocation of zl (similar to `dealloc` in C). Unlike $\texttt{dealloc}_s$, the deallocation of a virtually-allocated memory region, does not return the freed memory to stack stk (it instead goes back to the implicitly-defined region `free`).

Effectively, a lockstep correlation of virtual allocations in Ä with allocations in C ensures that the allocation states of both procedures always agree for regions $B \cup \{cv\}$.

Figure 2.1c shows an assembly procedure with annotated $\texttt{alloc}_v$ and $\texttt{dealloc}_v$ instructions. Instruction $\texttt{A3}^1$: $\texttt{v}_{\texttt{I1}} \coloneqq \texttt{alloc}_v$ 4, 4, `I1` performs virtual-allocation of a region of size 4 identified by `I1` and returns the start address in $\texttt{v}_{\texttt{I1}}$. Because $\texttt{alloc}_v$ is a validator-only construct, the return address is not used anywhere in rest of the procedure. The allocated region is deallocated by instruction $\texttt{A19}^2$: $\texttt{dealloc}_v$ `I1`.

2.6.2 Revised semantics for assembly procedure instructions

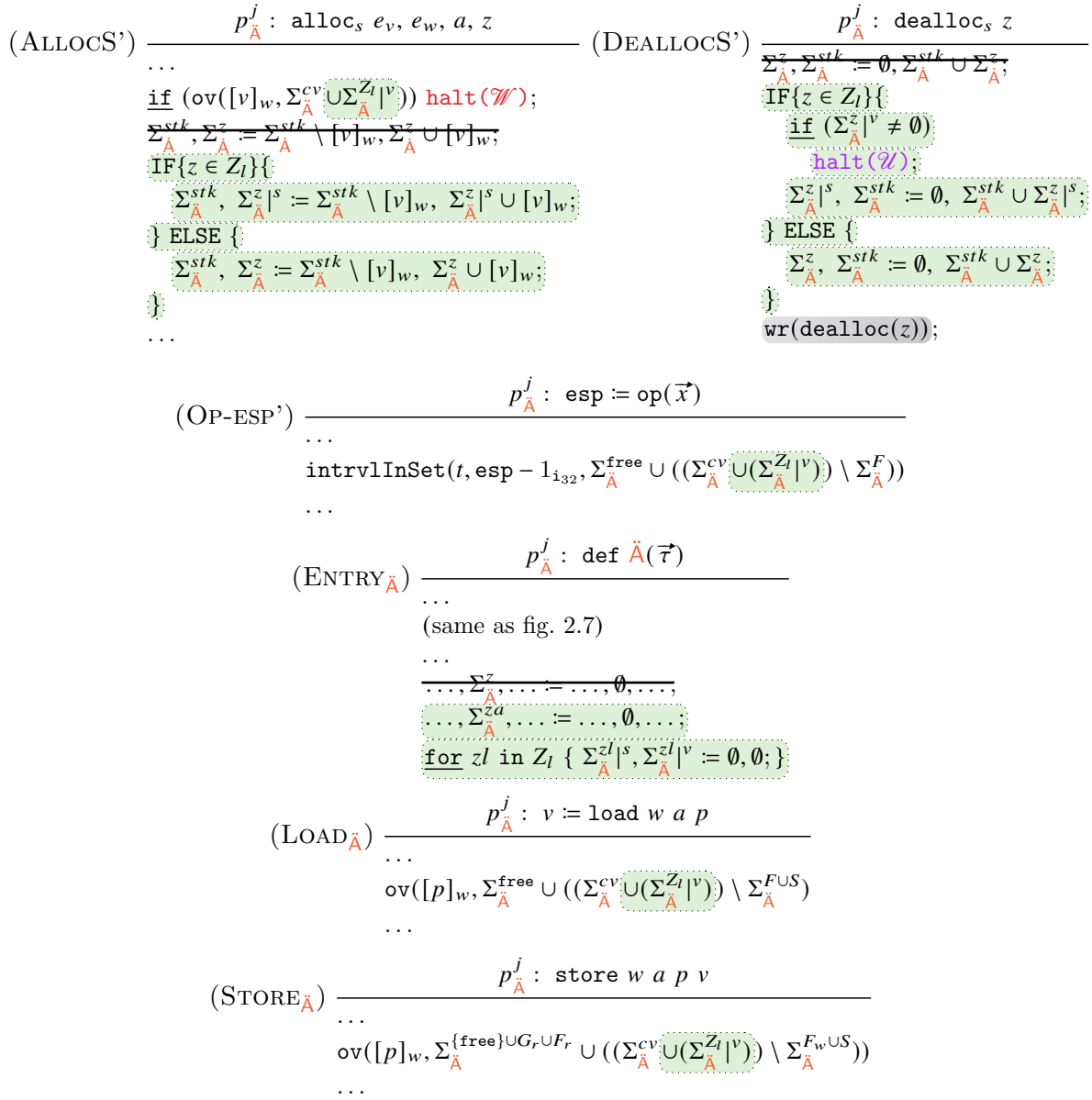


Figure 2.11: Revised translation rules for converting pseudo-assembly instructions to graph instructions. The IF{$z \in Z_l$}{...}ELSE{...} construct selects one of the translation depending on the result of syntactic predicate $z \in Z_l$.

Figure 2.11 shows the revised semantics for the procedure entry ((Entry$_{\ddot{A}}$)), alloc$_s$ and dealloc$_s$ instructions ((AllocS') and (DeallocS')), load and store instructions ((Load$_{\ddot{A}}$) and (Store$_{\ddot{A}}$)), and esp-modifying instruction ((Op-Esp')) of the assembly

procedure. Instead of reproducing the full translations, we only show the changes with appropriate context: the additions have a highlighted background and deletions are ~~striked out~~.

The execution semantics of Ä maintain the important invariant of separation of a virtual region from other common (with C) regions $B \cup \{cv\}$, thereby mirroring the execution semantics of C. However, a virtual region may potentially overlap with assembly-only regions $F \cup S$. This is because virtual-allocation is a validator-only construct, used by the validator solely for identifying a lockstep correlation of allocation states; a virtual-allocated region is never accessed otherwise in Ä. Thus, in the revised semantics of a stackpointer updating instruction, shown in (OP-ESP') of fig. 2.11, a stack push is allowed to overstep a virtually-allocated region (represented through $\Sigma_{Ä}^{Z_l}|^v$[20]). Notice the similarity in treatment of $\Sigma_{Ä}^{cv}$ and $\Sigma_{Ä}^{Z_l}|^v$, we will expound on this in a bit.

The revised semantics of the $\texttt{alloc}_s$ instruction in (ALLOCS') *assume* that the stack-allocated local memory is separate from virtually-allocated regions (and region $\Sigma_{Ä}^{cv}$), similar to separation assumption in $\texttt{alloc}_v$, error $\mathscr{W}$ is triggered otherwise. Further, both revised rules (ALLOCS') and (DEALLOCS') now use variable $\Sigma_{Ä}^{zl}|^s$ (instead of $\Sigma_{Ä}^{zl}$) for an address set of region $zl \in Z_l$, reflecting the separate tracking of stack-allocated and virtual-allocated address sets of region zl. Similarly to $\texttt{dealloc}_v$, $\texttt{dealloc}_s$ triggers $\mathscr{U}$ if $\Sigma_{Ä}^{zl}|^v$ $(zl \in Z_l)$ is non-empty, ensuring the execution of $\texttt{dealloc}_s$ empties $\Sigma_{Ä}^{zl}$ $(= \Sigma_{Ä}^{zl}|^s \cup \Sigma_{Ä}^{zl}|^v)$.

The revised $(\text{ENTRY}_{Ä})$ rule initializes both address sets $\Sigma_{Ä}^{zl}|^s$ and $\Sigma_{Ä}^{zl}|^v$ for each $zl \in Z_l$ to empty.

The revised semantics of memory access instructions $(\text{LOAD}_{Ä})$ and $(\text{STORE}_{Ä})$ enforce that a virtually-allocated region must never be accessed in Ä, unless it also happens to belong to the assembly-only regions $F \cup S$. Notice that this is similar to treatment of region cv which is similarly inaccessible in Ä.

The purpose of the cv or callers' virtual region should be clear now: cv or callers's virtual region of an assembly procedure Ä is the set of virtually-allocated addresses in Ä's call chain. At a procedure-call, the address set $\Sigma_{Ä}^{cv}$ for a callee is computed as $\Sigma_{Ä}^{cv} \cup \Sigma_{Ä}^{Z_l}|^v$. The lockstep correlation of allocation states (due to observation of (de)allocation) enables us to define Σ_{C}^{cv} for a callee in C using $\Sigma_{Ä}^{cv}$. As a virtual allocation is supposed to correspond to a register-allocated or an eliminated local, region cv is assumed to be

[20]Recall that $\Sigma_{Ä}^{Z_l}|^v = \bigcup_{zl \in Z_l}(\Sigma_{Ä}^{zl}|^v)$.

inaccessible in the callee[21]. This is sound because the set of observable regions for a callee constitute an observable in the caller and the equality of observables is required for establishing refinement.

2.6.3 Refinement Definition with both stack-allocated and register-allocated or eliminated locals

We define refinement in the presence of both stack-allocated and register-allocated or eliminated locals through the existence of an annotation $\ddot{A}$ of A such that $\ddot{A}$ refines C. $\ddot{A}$ is obtained through addition of $\texttt{(de)alloc}_v$ and $\texttt{(de)alloc}_s$ instructions (with semantics as described in sections 2.5.1 and 2.6.1) and annotation of a procedure-call instruction (with semantics as described in section 2.5.2) and use of revised semantics for other assembly instructions in A (as described in section 2.6.2).

Definition 2.6.1 (Refinement with stack and virtually-allocated locals)**.** $C \ddot{\sqsupseteq} A$ *iff:* $\exists \ddot{A} : C \sqsupseteq \ddot{A}$

Recall that $C \sqsupseteq \ddot{A}$ requires that *for all* non-deterministic choices of a virtually allocated local variable address in $\ddot{A}$ (v in (AllocV)), there *exists* a non-deterministic choice for the correlated local variable address in C (v in (Alloc)) such that: if $\ddot{A}$'s execution is well-formed (does not trigger $\mathscr{W}$), and C's execution is UB-free (does not trigger $\mathscr{U}$), then the two allocated intervals are identical (the observable values created through `allocBegin` and `allocEnd` value constructors must be equal).

In the presence of potentially register-allocated and eliminated local variables, $C \ddot{\sqsupseteq} A$ implies a correct translation from C to A. If all local variables are allocated in stack, $C \ddot{\sqsupseteq} A$ reduces to $C \dot{\sqsupseteq} A$ with $\ddot{A} = \dot{A}$. Figure 2.1c is an example of an annotated $\ddot{A}$.

Because our execution model observes each (de)allocation event (due to the `wr` instruction), a successful refinement check ensures that the allocation states of $\ddot{A}$ and C are identical at every correlated callsite for each procedure $C \in \mathbb{C}$. A coinductive argument over $\mathbb{C}$ and $\mathbb{A}$ is thus used to show that the address sets for the callers' locals — identified by cl and cv — are identical at the beginning of each correlated pair of procedures C and A, as modeled through identical reads from the outside world in (Entry_P) ($P \in \{C, A\}$) of figs. 2.5 and 2.7. A successful refinement check for each procedure-pair

[21]For a caller local to be accessible in a callee, it should have its address taken. An address-taken local cannot be register-allocated or eliminated.

C,A including the `main` procedure enables a coinductive proof of refinement from $\mathbb{C}$ to $\mathbb{A}$.

Capabilities and Limitations of $\mathsf{C} \ddot{\sqsupseteq} \mathsf{A}$

$\mathsf{C} \ddot{\sqsupseteq} \mathsf{A}$ inherits the limitation of requirement of preservation of the relative order of allocations and procedure calls that reuse stack space and the inability to handle interprocedural transformations from $\mathsf{C} \dot{\sqsupseteq} \mathsf{A}$. Notably, the example presented in fig. 2.9 is still not admitted under the new definition. However, virtual allocations enable admitting transformations involving register-allocation or elimination of a local — transformations that an optimizing compiler may and, in most cases, does perform. In Figure 2.1, the assembly procedure $\ddot{\mathsf{A}}_{\texttt{fib}}$ has register-allocated the local `i`. The `(de)alloc`$_v$ annotations enable the annotated $\ddot{\mathsf{A}}_{\texttt{fib}}$ to produce identical observable events as $\mathsf{C}_{\texttt{fib}}$ and establish $\mathsf{C}_{\texttt{fib}} \ddot{\sqsupseteq} \mathsf{A}_{\texttt{fib}}$[22].

Recall that we imposed two restrictions on annotations for virtual allocations (section 2.6.1): (1) `alloc`$_v$ annotation may only be added for local variable declarations, (2) a local variable may exclusively be either stack-allocated or virtual-allocated. While these restrictions make the execution semantics and our automatic algorithm simpler, they preclude supporting certain transformations. In particular, (2) appears to be quite limiting. We argue below that this is not the case and that the scope of transformations prohibited by (2) is relatively limited.

Recall that we need to add `alloc`$_s$ only in cases where a variable zl is address-taken (`alloc`$_v$ can be added in rest of the cases). So, unless variable zl is address-taken, we can always "virtual-allocate" it. For the case where stack-allocation of zl through `alloc`$_s$ is required (the case of an address-taken zl), $\mathsf{C} \ddot{\sqsupseteq} \mathsf{A}$ prohibits stack space reuse transformations where the stack space reserved for zl is reallocated during its lifetime. Such kind of transformations may be introduced due to *live-range splitting* [11] which is utilized by optimizing compilers for improving register allocation. Thus, if a local variable is address-taken, and yet the compiler has performed live-range splitting on it, then this definition does not support it. Fortunately, such transformations on address-

[22]Curiously, in this particular example, it is possible to establish $\mathsf{C}_{\texttt{fib}} \sqsupseteq \ddot{\mathsf{A}}_{\texttt{fib}}$ with `(de)alloc`$_s$ annotations for `i`. Line `A3` allocates 12 bytes on stack (presumably for aligning `esp` by 16) which are not used in any manner later. The 4 bytes required for `i` can be "allocated" out of these 16 bytes, i.e., the annotation at `A3`[1] can be `alloc`$_s$ `esp, 4, 4, I1`. Similarly, the annotation at `A19`[2] can be replaced with `dealloc`$_s$ `I1`. The resulting annotated $\dot{\mathsf{A}}_{\texttt{fib}}$ will satisfy $\mathsf{C}_{\texttt{fib}} \sqsupseteq \dot{\mathsf{A}}_{\texttt{fib}}$, and consequently establish $\mathsf{C}_{\texttt{fib}} \dot{\sqsupseteq} \mathsf{A}_{\texttt{fib}}$.

taken variables are rare. It is also worth pointing out that Static Single Assignment (SSA) [12] is a form of live-range splitting but almost all compilers implement SSA only for pseudo-registers, which are not address-taken, and our definition supports such transformations.

We give examples of unsupported transformations below.

Refinement failure due to elimination of `alloca()`: Recall that we restrict $\texttt{alloc}_v$ annotation to allocations due to a local variable declaration. If an `alloca()` in C is eliminated in A, then it may not be possible to annotate A with $\texttt{alloc}_s$ to produce Ä such that $C \sqsupseteq Ä$ holds. The following example demonstrates such C and A where the observable behaviors of both procedures are identical but $C \ddot{\sqsupseteq} A$ does not hold.

```
int foo()
{
  int *p = alloca(sizeof(int)*10);
  return 0;
}
```

```
foo:
  eax = 0
  ret
```

The pointer `p` returned by `alloca()` is not used in the C procedure (shown on the left) and hence eliminated by the compiler in the generated assembly (shown on the right). Our execution semantics forbid use of $\texttt{alloc}_v$ for this eliminated local allocation and, because the assembly procedure does not allocate any stack space, an $\texttt{alloc}_s$ annotation such that $C_{\texttt{foo}} \sqsupseteq Ä_{\texttt{foo}}$ holds cannot be made either. Thus, $C_{\texttt{foo}} \ddot{\sqsupseteq} A_{\texttt{foo}}$ cannot hold in this case.

Because the alloca() operator is not a part of the C standard, its use is, arguably, rare; also compilers are usually not aggressive about register-allocating the memory allocated by alloca(); and so, this limitation is not practically significant.

Refinement failure due to live-range splitting of an address-taken variable: Consider the C source and assembly code shown below.

```
void bar()
{
  int x; // alloc
  scanf(/*...*/, &x);
  for (...) {
    // 'x' used
  }
  printf(/*...*/, x);
} // dealloc
```

```
bar:
    ...
    esp -= 4 ; 'x' stack-allocated
    ... call scanf ...
    eax = mem4[esp]; ; 'x' register-allocated
    esp += 4 ; 'x' stack-deallocated
    ... ; for loop
    push eax ; value of 'x' as argument to printf
    ... call printf ...
    ...
```

In the C source, the variable `x` is address-taken in the call to `scanf`, then used in a loop, and, at the end, in the call to `printf`. In the assembly code, `x` is first stack-allocated so that the stack address can be passed to `scanf`. Then, before the loop, `x` is register-allocated to register `eax` and the allocated stack space is reclaimed. Lastly, `x`, present in register `eax`, is passed as argument to `printf`. Such an assembly code may be generated by a live-range splitting transformation where the live range of variable `x` is split into two variables $\texttt{x}_1$ and $\texttt{x}_2$ and $\texttt{x}_1$ is stack-allocated while $\texttt{x}_2$ is register-allocated with an assignment $\texttt{eax = mem}_4\texttt{[esp]}$ used to connect the two at the splitting point.

This procedure-pair cannot be admitted by our refinement definition because `x` requires an $\texttt{alloc}_s$ annotation (by virtue of being address-taken) but no $\texttt{(de)alloc}_s$ annotation can be made to produce $\ddot{\mathsf{A}}_{\texttt{bar}}$ such that $\mathsf{C}_{\texttt{bar}} \sqsupseteq \ddot{\mathsf{A}}_{\texttt{bar}}$ holds — the stack space for `x` is deallocated *before* the call to `printf` so a $\texttt{dealloc}_s$ must be inserted before the call but this would break the trace requirement because `dealloc` happens *after* the call to `printf` in $\mathsf{C}_{\texttt{bar}}$.

It is worth pointing out production compilers (Clang/LLVM, GCC) usually do not reclaim the stack space as eagerly as shown in the assembly above. Instead, the stack space for locals is reserved once in the prologue of the assembly procedure and reclaimed in the epilogue; in which case, our refinement definition will admit the transformation.

Refinement failure due to path specialization involving register-allocation on one path and stack-allocation on another for a local: Consider the C source and assembly code fragments shown below:

```
void baz()
{
  int x; // alloc
  if (...) {
    // used as '&x'
  } else {
    // used as 'x'
  }
} // dealloc
```

```
baz:
    ...
    if (...) jmp L1
L0: esp -= 4 ; allocation of 'x'
    ...
    esp += 4 ; deallocation of 'x'
    jmp L2
L1: ...      ; no stack allocation
    ...      ; on this branch
L2: ...
```

On the `if` branch, the address of the local `x` is taken such that a stack allocation would be required for it. But, on the `else` branch, the address of `x` is not taken so that a register allocation may suffice. The assembly on the right implements this strategy. Because our execution semantics disallow using both $\texttt{alloc}_s$ and $\texttt{alloc}_v$ annotations for `x`, it is impossible to annotate the assembly procedure such that refinement holds.

2.7 Towards A More General Refinement Definition and Execution Semantics

In this section, we explore execution semantics and a corresponding refinement definition to remove the limitations described in previous section. In these new semantics we retain the observation of (de)allocation in C and A, but define revised semantics for $\texttt{alloc}_v$ and replace $\texttt{(de)alloc}_s$ with new instructions.

We eliminate $\texttt{alloc}_s$ and $\texttt{dealloc}_s$, and instead introduce instructions `v2s` and `s2v` that do not produce non-silent trace events. We make minor changes to the semantics of $\texttt{alloc}_v$ and no longer restrict $\texttt{alloc}_v$ to an allocation due to local variable declaration. Figure 2.12 shows graph translations for these new instructions ((V2S) and (S2V)) and revised semantics of $\texttt{alloc}_v$ ((AllocV')) and procedure entry ((Entry$_{\ddot{A}}$)). Let Ä be obtained by annotating A using `v2s`, `s2v`, and $\texttt{(de)alloc}_v$ instructions, using revised semantics for procedure entry, and annotating a procedure-call with semantics as described in section 2.5.2.

(AllocV') shows the revised semantics for $\texttt{alloc}_v$ instruction. An instruction '$v :=$ $\texttt{alloc}_v$ e_w, a, z' now additionally sets three new ghost variables: $\texttt{lstVSz}.z$, $\texttt{align}.z$,

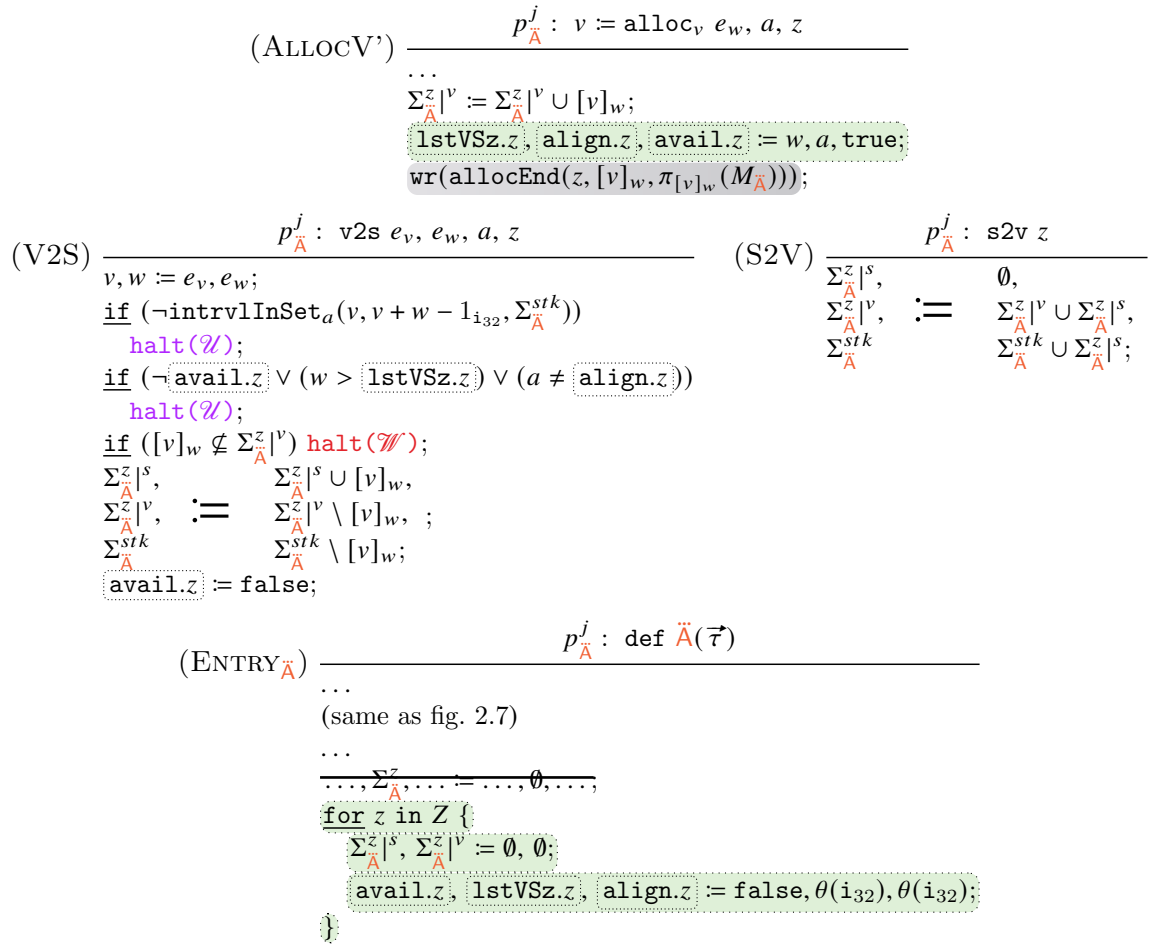


Figure 2.12: Translation rules for the converting pseudo-assembly instructions to graph instructions for Ä. (AllocV') is derived from (AllocV) in fig. 2.10.

and avail.z. These ghost variables keep track of the parameters of the *last* virtual allocation.

- lstVSz.z tracks the size of last virtual allocation due to z. In (AllocV'), lstVSz.z is assigned the size w ($= e_w$) of allocation. Each execution of $\texttt{alloc}_v$ thus updates lstVSz.z to the new value of allocation size.
- align.z tracks the required alignment of an allocation due to z — unlike lstVSz.z, the required alignment does not change across multiple executions of same $\texttt{alloc}_v$ instruction. Thus, the value of align.z, once assigned, does not change as Ä executes.
- avail.z tracks the availability of virtual region for *materialization* into a stack region. An execution of an $\texttt{alloc}_v$ instruction sets avail.z to true and an execution of an

`v2s` instruction resets it (shown in (V2S)).

Similar to the semantics defined in (AllocV), an error-free execution of $\texttt{alloc}_v$ requires the allocated interval to be separate from common regions $B \cup \{cv\}$ (but overlap with assembly-only regions $F \cup S$ is permitted).

An instruction '`v2s` e_v, e_w, a, z' converts (part of) virtual allocation region due to z to a (partial) stack-allocation of z. The parameters e_v, e_w, a, z of `v2s` have identical semantics as $\texttt{alloc}_s$. The interval of size e_w, starting at address e_v, must be aligned by a and belong to stack $\Sigma_{\ddot{A}}^{stk}$. Additionally, the execution of `v2s` must be sequenced after an $\texttt{alloc}_v$ instruction without any intervening `v2s` instruction for the same z ($\neg$ `avail.`z) such that the interval size is no greater than last virtual allocation size ($w >$ `lstVSz.`z) and alignment is exactly same as last virtual allocation ($a \neq$ `align.`z). A failure in meeting any of the above requirements triggers $\mathscr{U}$ indicating a translation error in this stack materialization of z, i.e., the compiler failed to meet the WF constraints for this stack allocation of z.

A valid execution of `v2s` requires the allocated interval to belong to *both* $\Sigma_{\ddot{A}}^{stk}$ and $\Sigma_{\ddot{A}}^{z}|^v$ so that the address set of stack allocations due to z, $\Sigma_{\ddot{A}}^{z}|^s$, is populated by transferring addresses from $\Sigma_{\ddot{A}}^{stk}$ and $\Sigma_{\ddot{A}}^{z}|^v$ to $\Sigma_{\ddot{A}}^{z}|^s$ (recall that $\Sigma_{\ddot{A}}^{stk}$ and $\Sigma_{\ddot{A}}^{z}|^v$ are allowed to overlap). We trigger $\mathscr{W}$ if the allocated interval does not belong $\Sigma_{\ddot{A}}^{z}|^v$, i.e, we assume that the non-deterministic choice made in $\texttt{alloc}_v$ is always such that a valid stack materialization is contained inside it. We define $\Sigma_{\ddot{A}}^{Z}|^v = \bigcup_{z \in Z} \Sigma_{\ddot{A}}^{z}|^v$ and $\Sigma_{\ddot{A}}^{z} = \Sigma_{\ddot{A}}^{z}|^s \cup \Sigma_{\ddot{A}}^{z}|^v$ for a $z \in Z$ (as done in section 2.6.1 for $zl \in Z_l$). An execution of `v2s` does not change $\Sigma_{\ddot{A}}^{z}$.

An instruction '`s2v` z' transfers *all* the stack allocations for a local z back to the virtual region and returns the corresponding addresses to stack. Unlike $\texttt{dealloc}_s$, the address set $\Sigma_{\ddot{A}}^{z}$ is not emptied due to execution of `s2v`; on the contrary, as with `v2s`, the execution of `s2v` does not affect $\Sigma_{\ddot{A}}^{z}$ — instead, it transfers the addresses in $\Sigma_{\ddot{A}}^{z}|^s$ back to $\Sigma_{\ddot{A}}^{z}|^v$ and $\Sigma_{\ddot{A}}^{stk}$. Similar to `v2s`, an execution of `s2v` does not change $\Sigma_{\ddot{A}}^{z}$. Neither `v2s` nor `s2v` produces any (non-silent) observable trace event.

(Entry$_{\ddot{A}}$) shows the revised semantics for procedure entry: the revision is limited to initializing address sets $\Sigma_{\ddot{A}}^{z}|^s$ and $\Sigma_{\ddot{A}}^{z}|^v$ to empty, setting ghost variable `avail.`z to `false`, and assigning non-deterministic values to ghost variables `lstVSz.`z and `align.`z, for each $z \in Z$.

As with previous definitions, refinement in this case is also defined through the existence

of an annotated $\ddot{A}$, annotated with (de)$\texttt{alloc}_v$, `s2v` and `v2s` instructions with semantics as described above and procedure-call with annotations as described in section 2.5.2, such that $C \sqsupseteq \ddot{A}$ holds.

Definition 2.7.1 (Refinement with stack and virtually-allocated locals)**.** $C \dddot{\sqsupseteq} A$ *iff:* $\exists \ddot{A} : C \sqsupseteq \ddot{A}$

2.7.1 Comparison with $C \dddot{\sqsupseteq} A$

$C \dddot{\sqsupseteq} A$ admits strictly more procedure-pairs than $C \ddot{\sqsupseteq} A$

We claim that $C \dddot{\sqsupseteq} A$ admits at least as many procedure-pairs as $C \ddot{\sqsupseteq} A$.

Let $\ddot{A}$ be obtained by annotating A as described in section 2.6. Let $\ddot{\ddot{A}}$ be obtained from $\ddot{A}$ by the following substitution:

- Substitute an '$\texttt{alloc}_s\ e_v, e_w, a, z$' instruction with instruction sequence '$\texttt{alloc}_v\ e_w, a, z$; `v2s` e_v, e_w, a, z'.
- Substitute a '$\texttt{dealloc}_s\ z$' instruction with instruction sequence '`s2v` z; $\texttt{dealloc}_v\ z$'.

Let the semantics defined in fig. 2.12 be used for the instructions in $\ddot{\ddot{A}}$.

Assume machine states σ and $\dot{\sigma}$ for $\ddot{A}$ and $\ddot{\ddot{A}}$ at nodes $n_{\ddot{A}} \in \mathcal{N}_{\ddot{A}}^{\cancel{\mathcal{U}\mathcal{W}}}$ and $n_{\ddot{\ddot{A}}} \in \mathcal{N}_{\ddot{\ddot{A}}}^{\cancel{\mathcal{U}\mathcal{W}}}$ respectively such that $n_{\ddot{A}}$ and $n_{\ddot{\ddot{A}}}$ are not error nodes. Let $=_\delta$ be a relation between σ and $\dot{\sigma}$ such that $\sigma =_\delta \dot{\sigma}$ holds iff:

- The state elements present in both σ and $\dot{\sigma}$ have identical values.
- $\Sigma^z_{\ddot{A}} = \Sigma^z_{\ddot{\ddot{A}}}$, $\Sigma^z_{\ddot{A}}|^s = \Sigma^z_{\ddot{\ddot{A}}}|^s$, and $\Sigma^z_{\ddot{A}}|^v = \Sigma^z_{\ddot{\ddot{A}}}|^v$ hold where:
  - $\Sigma^{zl}_{\ddot{A}} = \Sigma^{zl}_{\ddot{\ddot{A}}}|^s \cup \Sigma^{zl}_{\ddot{\ddot{A}}}|^v$ for $zl \in Z_l$.
  - $\Sigma^{za}_{\ddot{A}}|^s = \Sigma^{za}_{\ddot{\ddot{A}}}$ and $\Sigma^{za}_{\ddot{A}}|^v = \emptyset$ for $za \in Z_a$.
  - $\Sigma^z_{\ddot{\ddot{A}}} = \Sigma^z_{\ddot{\ddot{A}}}|^s \cup \Sigma^z_{\ddot{\ddot{A}}}|^v$ for $z \in Z$.

Lemma 2.7.2. *If $\sigma =_\delta \dot{\sigma}$ holds, and an execution of '$\texttt{alloc}_s\ e_v, e_w, a, z$' on σ triggers error, then an execution of instructions (in sequence) '$\texttt{alloc}_v\ e_w, a, z$; `v2s` e_v, e_w, a, z' on $\dot{\sigma}$ will also trigger an error (for some sequence of non-deterministic choices).*

Further, if the erroneous execution on σ does not trigger error $\mathcal{U}$, then the erroneous execution on $\dot{\sigma}$ also does not trigger error $\mathcal{U}$.

Proof. An execution of $\texttt{alloc}_s$ on σ may trigger either error $\mathcal{U}$ due to evaluation of

$\neg$`intrvlInSet`(...) check ((AllocS)) to `true`, call it case 1. Or, it may terminate with error $\mathscr{W}$ due to evaluation of `ov`(...) check ((AllocS')) to `true`, call it case 2.

1. *When error $\mathscr{U}$ is triggered:* The $\neg$`intrvlInSet`$_a(v, v+w-1_{i_{32}}, \Sigma_{\ddot{A}}^{stk})$ check due to `alloc`$_s$ ((AllocS)) is structurally identical to the $\mathscr{U}$-triggering check due to `v2s` ((V2S)). Further, the arguments to both instructions `alloc`$_s$ and `v2s` above are identical and due to $\sigma =_\delta \dot{\sigma}$, $\Sigma_{\ddot{A}}^{stk} = \Sigma_{\ddot{A}}^{stk}$ holds. Also, the execution of `alloc`$_v$ does not affect $\Sigma_{\ddot{A}}^{stk}$. Therefore, if execution of `alloc`$_v$ completes without triggering error $\mathscr{W}$, then execution of `v2s` will terminate with error $\mathscr{U}$ due to evaluation of $\neg$`intrvlInSet`$_a$(...) check to `false`.
2. *When error $\mathscr{W}$ is triggered:* An execution of instruction `alloc`$_v$ on $\dot{\sigma}$ may trigger $\mathscr{W}$ due to the non-deterministic choice for v in (AllocV') — to distinguish the v in (AllocV') from the v in (V2S), we will refer to the former as v_v and the latter as simply v [23].
   If the execution of `alloc`$_s$ on σ triggers $\mathscr{W}$ due to evaluation of `ov`$([v]_w, \Sigma_{\ddot{A}}^{cv} \cup \Sigma_{\ddot{A}}^{Z_l}|^v)$ check to `false`, then, for some sequence of non-deterministic choices, execution of `alloc`$_v$; `v2s` on $\dot{\sigma}$ may also trigger $\mathscr{W}$. We prove this by contradiction. Assume that the execution of `alloc`$_v$ on $\dot{\sigma}$ does not trigger $\mathscr{W}$, then consider the subsequent execution of `v2s` on resulting $\dot{\sigma}$. The evaluation of $\neg$`intrvlInSet`(...) on $\dot{\sigma}$ will not result in triggering $\mathscr{U}$ as it did not trigger $\mathscr{U}$ in execution of σ (same reasoning as case 1 above). If the execution of `v2s` on $\dot{\sigma}$ does not trigger $\mathscr{W}$ due to evaluation of $([v]_w \notin \Sigma_{\ddot{A}}^{z})$ to `false`, then we have $[v]_w \subseteq \Sigma_{\ddot{A}}^{z}$ or $[v]_w \subseteq \Sigma_{\ddot{A}}^{z}|^s \cup \Sigma_{\ddot{A}}^{z}|^v$ at the end of `alloc`$_v$'s execution (recall that v refers to the v in (V2S) here which holds identical value as v at the end of `alloc`$_s$'s execution). Due to $\sigma =_\delta \dot{\sigma}$ and the fact that execution of `alloc`$_s$ did not change $\Sigma_{\ddot{A}}^{z}|^s$ (due to early termination), we can conclude $[v]_w \subseteq (\Sigma_{\ddot{A}}^{z}|^v \cup [v_v]_w) \cup \Sigma_{\ddot{A}}^{z}|^s$ (notice the change from $\Sigma_{\ddot{A}}^{z}|^v$ to $\Sigma_{\ddot{A}}^{z}|^v$). As execution of `v2s` did not terminate with error $\mathscr{U}$, $[v]_w \subseteq \Sigma_{\ddot{A}}^{stk}$ must hold prior to execution of `v2s` and, therefore, $[v]_w \notin \Sigma_{\ddot{A}}^{z}|^s$ must hold at the same point [24]. Consequently, $[v]_w \subseteq \Sigma_{\ddot{A}}^{z}|^v \cup [v_v]_w$. If execution on σ triggered $\mathscr{W}$, then `ov`$([v]_w, \Sigma_{\ddot{A}}^{cv} \cup \Sigma_{\ddot{A}}^{Z}|^v)$ must hold. As v in `v2s` has identical value as v in `alloc`$_s$, `ov`$(\Sigma_{\ddot{A}}^{z}|^v \cup [v_v]_w, \Sigma_{\ddot{A}}^{cv} \cup \Sigma_{\ddot{A}}^{Z}|^v)$ must hold. $\neg$`ov`$([v_v]_w, \Sigma_{\ddot{A}}^{cv} \cup \Sigma_{\ddot{A}}^{Z}|^v)$ holds for error-free execution of `alloc`$_v$ and $\neg$`ov`$(\Sigma_{\ddot{A}}^{z}|^v, \Sigma_{\ddot{A}}^{cv})$ holds for σ at error-free node, yielding `ov`$(\Sigma_{\ddot{A}}^{z}|^v, \Sigma_{\ddot{A}}^{Z}|^v)$. Recall that Ä restricts `alloc`$_v$ to local variable declaration z and for a local z, Ä may either have

[23] We do not similarly distinguish between w because, unlike v, any execution over $\dot{\sigma}$ will have identical values for both w.

[24] This is due to the semantics of `v2s` and `s2v`, which prohibit an address α from simultaneously satisfying $\alpha \in \Sigma_{\ddot{A}}^{z}|^s$ and $\alpha \in \Sigma_{\ddot{A}}^{stk}$.

$\texttt{alloc}_s$ or $\texttt{alloc}_v$ annotation but not both. This gives us $\Sigma_{\ddot{A}}^{z}|^{v} = \emptyset$, resulting in the contradiction $\texttt{ov}(\emptyset, \Sigma_{\ddot{A}}^{Z}|^{v})$. Thus, either of our assumption of error-free execution of $\texttt{alloc}_v$ or `v2s` was incorrect and execution of $\dot{\sigma}$ must also terminate with error $\mathscr{W}$.

Due to case 2, it follows that the erroneous execution on $\dot{\sigma}$ will trigger $\mathscr{W}$ (and not $\mathscr{U}$) if the erroneous execution on σ triggers $\mathscr{W}$. □

Lemma 2.7.3. *If $\sigma =_{\delta} \dot{\sigma}$ holds, and an execution of '$\texttt{alloc}_s\ e_v, e_w, a, z$' on σ completes without triggering error, then there exists a sequence of non-deterministic choices such that an execution of instructions (in sequence) '$\texttt{alloc}_v\ e_w, a, z$; $\texttt{v2s}\ e_v, e_w, a, z$' on $\dot{\sigma}$ also completes without triggering error.*

Proof. If the execution on σ completed without triggering error, then neither of the conditions $\neg\texttt{intrvlInSet}(\ldots)$ and $\texttt{ov}(\ldots)$ in (AllocS') evaluated to `true` during σ's execution. Thus, both $\neg\texttt{ov}([v]_w, \Sigma_{\ddot{A}}^{cv} \cup \Sigma_{\ddot{A}}^{Z}|^{v})$ and $\neg\texttt{ov}([v]_w, \Sigma_{\ddot{A}}^{B} \setminus (\Sigma_{\ddot{A}}^{cv} \cup \Sigma_{\ddot{A}}^{Z}|^{v}))$ [25] should hold on σ for $v = e_v$ and $w = e_w$ before execution of $\texttt{alloc}_s$ begins, yielding $\neg\texttt{ov}([v]_w, \Sigma_{\ddot{A}}^{B\cup\{cv\}})$ or $\neg\texttt{ov}([v]_w, \Sigma_{\ddot{A}}^{B\cup\{cv\}})$ (due to $\sigma =_{\delta} \dot{\sigma}$). During the execution on $\dot{\sigma}$, choose v in (AllocV') to be identical to v above. It can be observed that the execution on $\dot{\sigma}$ will complete without triggering error. □

Lemma 2.7.4. *If $\sigma =_{\delta} \dot{\sigma}$ holds, then after an error-free execution of instruction '$\texttt{alloc}_s\ e_v, e_w, a, z$' on σ to obtain σ' and an error-free execution of instructions (in sequence) '$\texttt{alloc}_v\ e_w, a, z$; $\texttt{v2s}\ e_v, e_w, a, z$' on $\dot{\sigma}$ to obtain $\dot{\sigma}'$, $\sigma' =_{\delta} \dot{\sigma}'$ holds.*

As $\Omega_{\ddot{A}} \in \sigma'$ and $\Omega_{\ddot{A}} \in \dot{\sigma}'$, the executions must produce identical traces as well.

Proof. An error-free execution of $\texttt{alloc}_s$ on σ will mutate the address sets: $\Sigma_{\ddot{A}}^{stk}$ to $\Sigma_{\ddot{A}}^{stk} \setminus [v]_w$ and either of $\Sigma_{\ddot{A}}^{z}$ to $\Sigma_{\ddot{A}}^{z} \cup [v]_w$ (if $z \in Z_a$) or $\Sigma_{\ddot{A}}^{z}|^{s}$ to $\Sigma_{\ddot{A}}^{z}|^{s} \cup [v]_w$ (if $z \in Z_l$). Similarly, an error-free execution of $\texttt{alloc}_v$; `v2s` in sequence will mutate the address sets: $\Sigma_{\ddot{A}}^{stk}$ to $\Sigma_{\ddot{A}}^{stk} \setminus [v_s]_w$, $\Sigma_{\ddot{A}}^{z}|^{s}$ to $\Sigma_{\ddot{A}}^{z}|^{s} \cup [v_s]_w$, and $\Sigma_{\ddot{A}}^{z}|^{v}$ to $(\Sigma_{\ddot{A}}^{z}|^{v} \cup [v_v]_w) \setminus [v_s]_w$, where v_s refers to the v in (V2S) (which is identical to the v in $\texttt{alloc}_s$ above) and v_v refers to the v in (AllocV'). Choosing v_v (of (AllocV')) to be identical to v_s, it can be observed that $\sigma' =_{\delta} \dot{\sigma}'$ will hold. Proof of lemma 2.7.3 shows that this choice is always feasible for an error-free execution. □

[25] Due to semantics of (Op-esp'), (AllocS') and (DeallocS'), and (AllocV) and (DeallocV), because of which $\neg\texttt{ov}(\Sigma_{\ddot{A}}^{stk}, \Sigma_{\ddot{A}}^{B} \setminus (\Sigma_{\ddot{A}}^{cv} \cup \Sigma_{\ddot{A}}^{Z}|^{v}))$ holds.

It is easy to state and prove similar claims as lemmas 2.7.2 to 2.7.4 for '$\texttt{dealloc}_s\ z$' in Ä and instruction sequence '$\texttt{s2v}\ z; \texttt{dealloc}_v\ z$' in Ä̈. Similarly, for instruction '$\texttt{alloc}_v$' (resp. $\texttt{dealloc}_v$) in Ä and instruction '$\texttt{alloc}_v$' (resp. $\texttt{dealloc}_v$) in Ä̈, where the semantics are identical (barring the ghost variables $\texttt{lstVSz}.z$, $\texttt{align}.z$, and $\texttt{avail}.z$).

Theorem 2.7.5. *If there exists an annotation* Ä *such that* C ⊒ Ä *holds, then it is possible to construct an annotation* Ä̈ *such that* C ⊒ Ä̈ *holds.*

Proof. Construct Ä̈ as described above by replacing the $\texttt{alloc}_s$ and $\texttt{dealloc}_s$, instructions in Ä with instruction sequences '$\texttt{alloc}_v$; $\texttt{v2s}$' and '$\texttt{s2v}$; $\texttt{dealloc}_v$' respectively.

The proof follows from induction on lockstep execution of Ä and Ä̈ and lemmas 2.7.2 to 2.7.4. □

Recall that C ⫅̈ A does not admit certain procedure-pairs that have identical observable behavior according to the C standard (section 2.6.3). We demonstrate below that those procedure-pairs will be admitted under C ⫅̈ A.

- Elimination of `alloca()`:

```
int foo()
{
  int *p = alloca(sizeof(int)*10);
  return 0;
}
```

```
foo:
  allocv 40, ...
  eax = 0
  deallocv
  ret
```

  Without the restriction on placement of $\texttt{alloc}_v$, it is easy to add the required $\texttt{(de)alloc}_v$ annotations (shown in red) such that $\text{C}_{\texttt{foo}} \sqsupseteq \text{Ä}_{\texttt{foo}}$ holds.

- Live-range splitting of variable so that it is both stack-allocated and register-allocated in same path:

```
void bar()
{
  int x; // alloc
  scanf(/*...*/, &x);
  for (...) {
    // 'x' used
  }
  printf(/*...*/, x);
} // dealloc
```

```
bar:
    alloc_v 4, ...
    esp -= 4 ; 'x' stack-allocated
    v2s esp, 4, ...
    ... call scanf ...
    eax = mem_4[esp]; ; 'x' register-allocated
    s2v ...
    esp += 4 ; 'x' stack-deallocated
    ... ; for loop
    push eax ; value of 'x' as argument to printf
    ... call printf ...
    ...
    dealloc_v
```

The v2s and s2v annotations in $\ddot{A}_{bar}$ trace the stack-allocation and register-allocation performed by the compiler. As neither v2s nor s2v produce any observable trace event thats needs to be correlated with C_{bar}, an identical behavior in the ... fragments in both C_{bar} and A_{bar} with annotations as shown in red, $C_{bar} \sqsupseteq \ddot{A}_{bar}$ will hold.

- Path specialization involving register-allocation on one path and stack-allocation on another:

```
void baz()
{
  int x; // alloc
  if (...) {
    // used as '&x'
  } else {
    // used as 'x'
  }
} // dealloc
```

```
baz:
    ...
    alloc_v 4, ...
    if (...) jmp L1
L0: esp -= 4  ; true branch
    v2s esp, 4, ...
    ...
    s2v ...
    esp += 4
    jmp L2
L1: ...       ; false branch: no stack
    ...       ;              allocation
L2: ...       ; meet point
    dealloc_v
```

The use of v2s and s2v annotations in $\ddot{A}_{baz}$ enable treatment of a stack region as

belonging to a previously "allocated" (using $\texttt{alloc}_v$) local without producing any observable trace event that needs to be correlated with $\mathsf{C}_{\texttt{baz}}$. Assuming identical behaviors in the ... fragments in both $\mathsf{C}_{\texttt{baz}}$ and $\mathsf{A}_{\texttt{baz}}$ and annotations as shown in red, $\mathsf{C}_{\texttt{baz}} \sqsupseteq \ddot{\mathsf{A}}_{\texttt{baz}}$ will hold.

Thus, $\mathsf{C} \dddot{\sqsupseteq} \mathsf{A}$ admits strictly more procedure-pairs than $\mathsf{C} \ddot{\sqsupseteq} \mathsf{A}$.

Practicality of $\mathsf{C} \dddot{\sqsupseteq} \mathsf{A}$

While $\mathsf{C} \dddot{\sqsupseteq} \mathsf{A}$ is strictly more *powerful* than $\mathsf{C} \ddot{\sqsupseteq} \mathsf{A}$, the restrictions on $\mathsf{C} \ddot{\sqsupseteq} \mathsf{A}$ make it more amenable to a *simpler* algorithmic construction such that $\mathsf{C} \ddot{\sqsupseteq} \mathsf{A}$ can be witnessed (our construction is detailed in chapters 3 and 4). The two restrictions defined over annotation of $\texttt{alloc}_v$ (section 2.6.1) are consequential in realizing an efficient SMT encoding of verification conditions over C and $\ddot{\mathsf{A}}$. In particular, the restrictions on limiting annotation of $\texttt{alloc}_v$ to a variable declaration enable use of an address interval (defined by ghost variables $\texttt{lb}.z$ and $\texttt{ub}.z$ in C) for tracking $\Sigma^{zl}_{\ddot{\mathsf{A}}}|^v$. Use of such *interval encoding* results in measurable performance improvements in runtime of our algorithm (section 6.2). The restriction on exclusivity (using exclusively either an annotation of $\texttt{alloc}_s$ or an annotation $\texttt{alloc}_v$) enables a rewrite of the predicate $(\alpha \in \Sigma^{zl}_{\ddot{\mathsf{A}}})$ to either $(\alpha \in \Sigma^{zl}_{\ddot{\mathsf{A}}}|^s)$ or $(\alpha \in \Sigma^{zl}_{\ddot{\mathsf{A}}}|^v)$, depending on whether an $\texttt{alloc}_s$ or an $\texttt{alloc}_v$ annotation is used respectively. $(\alpha \in \Sigma^{z}_{\ddot{\mathsf{A}}}|^v)$ can be encoded (in SMT) using simple (bitvector) comparisons by taking advantage of the first restriction. We leave the exploration of similar construction for $\mathsf{C} \dddot{\sqsupseteq} \mathsf{A}$ for future work.

Chapter 3

Witnessing Refinement through a Determinized Cross-Product

In the previous chapter, we defined refinement between a source procedure C and an annotated assembly procedure Ä as a relation over their traces. In this chapter, we present our proof method for witnessing refinement. Our proof method involves constructing a cross-product or product program that puts C and Ä in lockstep. While product program construction is a well-known technique for establishing bisimulation, we propose a *determinized product program*, an extension that accommodates non-determinism and thus can be used for witnessing refinement. We describe a set of requirements over a determinized product program between C and Ä such that the existence of a program meeting these requirements implies the refinement relation $\mathsf{C} \sqsupseteq \ddot{\mathsf{A}}$.

The chapter is organized as follows: we start by defining program paths in section 3.1; in section 3.2, we define the product program as a determinized product graph and in section 3.3, we state the requirements over the determinized product graph that enable it to witness refinement. In sections 3.4 and 3.5, we describe new *callers' virtual smallest* and *safety-relaxed* semantics for C and A that enable a more efficient SMT encoding of verification conditions.

3.1 Program Paths

Let $P \in \{\mathsf{C}, \ddot{\mathsf{A}}\}$. Let $e_P = (n_P \rightarrow n_P^t) \in \mathcal{E}_P$ represent an edge from node n_P to node n_P^t, both drawn from nodes $\mathcal{N}_P$ of P. A *path* ξ_P from n_P to n_P^t, written $\xi_P = (n_P \twoheadrightarrow n_P^t)$, is a finite sequence of $m \geq 0$ edges $(e_P^1, e_P^2, \ldots, e_P^m)$ with $\forall_{1 \leq j \leq m} : e_P^j = (n_P^{f,j} \rightarrow n_P^{t,j}) \in \mathcal{E}_P$, such that $n_P^{f,1} = n_P$, $n_P^{t,m} = n_P^t$, and $\bigwedge_{j=1}^{m-1} (n_P^{t,j} = n_P^{f,j+1})$. An empty sequence ($m = 0$), written ϵ, represents the *empty path*. Nodes n_P and n_P^t are called the *source* and *sink* nodes of ξ_P respectively. ξ_P is said to originate at n_P and end at n_P^t. Edge e_P^j (for some $1 \leq j \leq m$) is said to be present in ξ_P, written $e_P^j \in \xi_P$.

A path $\xi_P^x = (e_P^{x_1}, e_P^{x_2}, \ldots, e_P^{x_m})$ ($m \geq 0$) is a prefix of path $\xi_P^y = (e_P^{y_1}, e_P^{y_2}, \ldots, e_P^{y_n})$ ($n \geq 0$), written $\xi_P^x \preceq \xi_P^y$, iff $m \leq n$ and $\bigwedge_{i=1}^{m} (e_P^{x_i} = e_P^{y_i})$.

Definition 3.1.1 (Mutually exclusive paths)**.** *Two paths, $\xi_P^1 = (n_P \twoheadrightarrow n^{t_1})$ and $\xi_P^2 = (n_P \twoheadrightarrow n_P^{t_2})$, both originating at node n_P are* ***mutually-exclusive****, written $\xi_P^1 \nsim \xi_P^2$, iff neither is a prefix of the other, i.e., both $\neg(\xi_P^1 \preceq \xi_P^2)$ and $\neg(\xi_P^2 \preceq \xi_P^1)$ hold.*

Definition 3.1.2 (Pathset)**.** *A* ***pathset*** *$\langle \xi \rangle_P$ is a set of pairwise mutually-exclusive paths $\langle \xi \rangle_P = \{\xi_P^1, \xi_P^2, \ldots, \xi_P^m\}$ originating at the same node n_P, i.e., $\forall_{1 \leq j \leq m} : \xi_P^j = (n_P \twoheadrightarrow n_P^j)$ and $\forall_{1 \leq j_1 < j_2 \leq m} : (\xi_P^{j_1} \nsim \xi_P^{j_2})$.*

The execution of a path $\xi_P = (n_P \twoheadrightarrow n_P^t) = (e_P^1, e_P^2, \ldots, e_P^m)$ ($m \geq 0$) is the sequential execution of edges $e_P^1, e_P^2, \ldots, e_P^m$ starting at node n_P. The *path condition* of a path ξ_P, written $pathcond(\xi_P)$, is a conjunction of the edge conditions of the constituent edges. Starting at n_P, $pathcond(\xi_P)$ represents the condition that ξ_P executes to completion.

Definition 3.1.3 (I/O path)**.** *A sequence of edges corresponding to a* shaded *statement in the translations (figs. 2.4 to 2.8 and 2.11) is distinguished and identified as an* ***I/O path****. An I/O path must contain either a single* `rd` *or a single* `wr` *instruction.*

A `rd`/`wr` instruction is always part of an I/O path. The sequence of edges corresponding to "$\texttt{wr}(\texttt{fcall}(\rho, \vec{x}, \beta^*, \pi_{\Sigma_\mathsf{C}^{\beta^*}}(M_\mathsf{C})))$" and "$M_\mathsf{C} \coloneqq \texttt{upd}_{\Sigma_\mathsf{C}^{\beta^* \setminus G_r}}(M_\mathsf{C}, \texttt{rd}(\texttt{i}_{32} \rightarrow \texttt{i}_8))$" in ($\textsc{Call}_\mathsf{C}$) of fig. 2.5 refer to two separate I/O paths. A path without any `rd` or `wr` instructions is called an *I/O-free path*.

3.2 Determinized Product Graph as a Transition Graph

A product program, represented as a *determinized product graph*, also called a comparison graph or a cross-product, $\mathsf{X} = \ddot{\mathsf{A}} \boxtimes \mathsf{C} = (\mathcal{N}_\mathsf{X}, \mathcal{E}_\mathsf{X}, \mathcal{D}_\mathsf{X})$, is a directed multigraph with finite sets of nodes $\mathcal{N}_\mathsf{X}$ and edges $\mathcal{E}_\mathsf{X}$ and a *deterministic choice map* $\mathcal{D}_\mathsf{X}$. X is used to encode a lockstep execution of $\ddot{\mathsf{A}}$ and C, such that $\mathcal{N}_\mathsf{X} \subseteq \mathcal{N}_{\ddot{\mathsf{A}}} \times \mathcal{N}_\mathsf{C}$. The start node of X is $n^s_\mathsf{X} = (n^s_{\ddot{\mathsf{A}}}, n^s_\mathsf{C})$ and all nodes in $\mathcal{N}_\mathsf{X}$ must be reachable from n^s_X. A node $n_\mathsf{X} = (n_{\ddot{\mathsf{A}}}, n_\mathsf{C})$ is an error node iff either $n_{\ddot{\mathsf{A}}}$ or n_C is an error node[1]. $\mathcal{N}^{\mathcal{UW}}_\mathsf{X}$ denotes the set of error-free nodes in X, such that $n_\mathsf{X} = (n_{\ddot{\mathsf{A}}}, n_\mathsf{C}) \in \mathcal{N}^{\mathcal{UW}}_\mathsf{X} \Leftrightarrow (n_{\ddot{\mathsf{A}}} \in \mathcal{N}^{\mathcal{UW}}_{\ddot{\mathsf{A}}} \wedge n_\mathsf{C} \in \mathcal{N}^{\mathcal{UW}}_\mathsf{C})$. A node $n_\mathsf{X} = (n_{\ddot{\mathsf{A}}}, n_\mathsf{C})$ is a terminating node iff either of $n_{\ddot{\mathsf{A}}}$ or n_C is a terminating node. An error node is always a terminating node.

Let $n_\mathsf{X} = (n_{\ddot{\mathsf{A}}}, n_\mathsf{C})$ and $n^t_\mathsf{X} = (n^t_{\ddot{\mathsf{A}}}, n^t_\mathsf{C})$ be nodes in $\mathcal{N}_\mathsf{X}$. Let $\xi_{\ddot{\mathsf{A}}} = (n_{\ddot{\mathsf{A}}} \twoheadrightarrow n^t_{\ddot{\mathsf{A}}})$ be a path in $\ddot{\mathsf{A}}$ and let $\xi_\mathsf{C} = (n_\mathsf{C} \twoheadrightarrow n^t_\mathsf{C})$ be a path in C. Each edge, $e_\mathsf{X} = (n_\mathsf{X} \xrightarrow{\xi_{\ddot{\mathsf{A}}};\xi_\mathsf{C}} n^t_\mathsf{X}) \in \mathcal{E}_\mathsf{X}$, is defined as a sequential execution of $\xi_{\ddot{\mathsf{A}}}$ followed by ξ_C. The execution of e_X transfers control of X from n_X to n^t_X.

The machine state σ_X of X is the concatenation of the machine states of $\ddot{\mathsf{A}}$ and C. The outside world of X, written Ω_X, is a pair of the outside worlds of $\ddot{\mathsf{A}}$ and C, i.e., $\Omega_\mathsf{X} = (\Omega_{\ddot{\mathsf{A}}}, \Omega_\mathsf{C})$. Similarly, the trace generated by X, written T_X, is a pair of the traces generated by $\ddot{\mathsf{A}}$ and C, i.e., $T_\mathsf{X} = (T_{\ddot{\mathsf{A}}}, T_\mathsf{C})$.

Let $n_\mathsf{X} = (n_{\ddot{\mathsf{A}}}, n_\mathsf{C}) \in \mathcal{N}_\mathsf{X}$ and $n^t_\mathsf{X} = (n^t_{\ddot{\mathsf{A}}}, n^t_\mathsf{C}) \in \mathcal{N}_\mathsf{X}$. During an execution of $e_\mathsf{X} = (n_\mathsf{X} \xrightarrow{\xi_{\ddot{\mathsf{A}}};\xi_\mathsf{C}} n^t_\mathsf{X}) \in \mathcal{E}_\mathsf{X}$, let $\vec{x}_{\ddot{\mathsf{A}}}$ be variables in $\ddot{\mathsf{A}}$ just at the end of the execution of path $\xi_{\ddot{\mathsf{A}}}$ (at $n^t_{\ddot{\mathsf{A}}}$) but before the execution of path ξ_C (recall, $\xi_{\ddot{\mathsf{A}}}$ executes before ξ_C). $\mathcal{D}_\mathsf{X} : (\mathcal{E}_\mathsf{X} \times \mathcal{E}_\mathsf{C} \times \mathbb{N}) \rightarrow \texttt{ExprList}$, called a *deterministic choice map*, is a partial function that maps edge $e_\mathsf{X} \in \mathcal{E}_\mathsf{X}$, and the n^{th} (for $n \in \mathbb{N}$) occurrence of an edge $e^\theta_\mathsf{C} \in \xi_\mathsf{C}$ labeled with instruction '$\vec{v} := \theta(\vec{\tau})$' to a list of expressions $E(\vec{x}_{\ddot{\mathsf{A}}})$. The semantics of $\mathcal{D}_\mathsf{X}$ are such that, if $\mathcal{D}_\mathsf{X}(e_\mathsf{X}, e^\theta_\mathsf{C}, n)$ is defined, then during an execution of e_X, an execution of the n^{th} occurrence of edge $e^\theta_\mathsf{C} \in \xi_\mathsf{C}$ labeled with '$\vec{v} := \theta(\vec{\tau})$' is semantically equivalent to an execution of '$\vec{v} := \mathcal{D}_\mathsf{X}(e_\mathsf{X}, e^n_\mathsf{C}, n)$'; otherwise, the original semantics of θ are used.

$\mathcal{D}_\mathsf{X}$ determinizes (or refines) the non-deterministic choices in C. For example, in a product graph X that correlates the programs in fig. 2.1b and fig. 2.1c, let $e^2_\mathsf{X} \in \mathcal{E}_\mathsf{X}$ correlate

[1] Recall that there are two error nodes in $P \in \{\mathsf{C}, \ddot{\mathsf{A}}\}$: $\mathscr{U}_P$ and $\mathscr{W}_P$ (section 2.2.7).

single instructions I2 and A5[1] (corresponding to alloc and $\texttt{alloc}_s$ respectively). Let $e_{\mathsf{C}}^{\texttt{I2},\theta_a}$ represent the edge labeled with '$\alpha_b \coloneqq \theta(\texttt{i}_{32})$' as a part of the translation of the alloc instruction at I2, as seen in (Alloc) (fig. 2.5). Then, $\mathcal{D}_{\mathsf{X}}(e_{\mathsf{X}}^2, e_{\mathsf{C}}^{\texttt{I2},\theta_a}, 1) = (\texttt{esp})$ is identified by the first operand of the annotated $\texttt{alloc}_s$ instruction at A5[1]. Similarly, if another edge $e_{\mathsf{C}}^{\texttt{I2},\theta_m}$ (in the translation of alloc at I2) is labeled with $\theta(\texttt{i}_{32} \rightarrow \texttt{i}_8)$ (due to '$M_{\mathsf{C}} \coloneqq \texttt{upd}_{[\alpha_b,\alpha_e]}(M_{\mathsf{C}}, \theta(\texttt{i}_{32} \rightarrow \texttt{i}_8))$'), then $\mathcal{D}_{\mathsf{X}}(e_{\mathsf{X}}^2, e_{\mathsf{C}}^{\texttt{I2},\theta_m}, 1) = (M_{\ddot{\mathsf{A}}})$, i.e., the initial contents of the newly-allocated region in C are based on the contents of the correlated uninitialized stack region in Ä. Similarly, let $e_{\mathsf{X}}^1 \in \mathcal{E}_{\mathsf{X}}$ correlate single instructions I1 and A3[1] so that $\mathcal{D}_{\mathsf{X}}(e_{\mathsf{X}}^1, e_{\mathsf{C}}^{\texttt{I1},\theta_a}, 1) = (\texttt{v}_{\texttt{I1}})$ and $\mathcal{D}_{\mathsf{X}}(e_{\mathsf{X}}^1, e_{\mathsf{C}}^{\texttt{I1},\theta_m}, 1) = (M_{\ddot{\mathsf{A}}})$.

Definition 3.2.1 (Determinized Path)**.** *For a path ξ_{C} in C, edge $e_{\mathsf{X}} = (n_{\mathsf{X}} \xrightarrow{\xi_{\ddot{\mathsf{A}}};\xi_{\mathsf{C}}} n'^{t}_{\mathsf{X}}) \in \mathcal{E}_{\mathsf{X}}$ and deterministic choice map $\mathcal{D}_{\mathsf{X}}$, $[\xi_{\mathsf{C}}]^{e_{\mathsf{X}}}_{\mathcal{D}_{\mathsf{X}}}$ denotes a **determinized path** that is identical to ξ_{C} except that: if $\mathcal{D}_{\mathsf{X}}(e_{\mathsf{X}}, e_{\mathsf{C}}^{\theta}, n)$ is defined, then the n^{th} occurrence of edge $e_{\mathsf{C}}^{\theta} \in \xi_{\mathsf{C}}$, labeled with '$\vec{v} \coloneqq \theta(\vec{\tau})$', is replaced with a new edge $e_{\mathsf{C}}^{\theta'_n}$ labeled with '$\vec{v} \coloneqq \mathcal{D}_{\mathsf{X}}(e_{\mathsf{X}}, e_{\mathsf{C}}^{\theta}, n)$'.*

Execution of a product graph X must begin at node n^s_{X} in an initial machine state where $\Omega_{\ddot{\mathsf{A}}} = \Omega_{\mathsf{C}}$ and $T_{\ddot{\mathsf{A}}} =_{st} T_{\mathsf{C}}$ hold. Execution of an edge $e_{\mathsf{X}} = (n_{\mathsf{X}} \xrightarrow{\xi_{\ddot{\mathsf{A}}};\xi_{\mathsf{C}}} n'^{t}_{\mathsf{X}}) \in \mathcal{E}_{\mathsf{X}}$ is execution of $\xi_{\ddot{\mathsf{A}}}$ followed by execution of potentially determinized (using $\mathcal{D}_{\mathsf{X}}$) ξ_{C}. Thus, X is a transition graph with its execution semantics derived from the semantics of Ä and C, and the map $\mathcal{D}_{\mathsf{X}}$.

3.3 Analysis of the determinized product graph

Let $\mathsf{X} = \ddot{\mathsf{A}} \boxtimes \mathsf{C} = (\mathcal{N}_{\mathsf{X}}, \mathcal{E}_{\mathsf{X}}, \mathcal{D}_{\mathsf{X}})$ be a determinized product graph. At each error-free node $n_{\mathsf{X}} \in \mathcal{N}_{\mathsf{X}}^{\mathcal{UW}}$, we infer an inductively-provable node invariant $\phi_{n_{\mathsf{X}}}$ which is a first-order logic predicate over state elements of X at node n_{X} that holds for all possible executions of X. A node invariant $\phi_{n_{\mathsf{X}}}$ relates the values of state elements of C and Ä that can be observed at n_{X}. Let Φ_{X} be an *invariant network* that maps n_{X} to its node invariant $\phi_{n_{\mathsf{X}}}$ so that $\Phi_{\mathsf{X}}(n_{\mathsf{X}}) = \phi_{n_{\mathsf{X}}}$.

A path $\xi_{\ddot{\mathsf{A}}}$ (similarly, ξ_{C}) is said to originate at node $n_{\mathsf{X}} = (n_{\ddot{\mathsf{A}}}, n_{\mathsf{C}}) \in \mathcal{N}_{\mathsf{X}}$ iff $\xi_{\ddot{\mathsf{A}}}$ (ξ_{C}) originates at $n_{\ddot{\mathsf{A}}}$ (n_{C}).

Definition 3.3.1 (Hoare triple)**.** *Let $n_{\mathsf{X}} = (n_{\ddot{\mathsf{A}}}, n_{\mathsf{C}}) \in \mathcal{N}_{\mathsf{X}}^{\mathcal{UW}}$. Let $\xi_{\ddot{\mathsf{A}}} = (n_{\ddot{\mathsf{A}}} \twoheadrightarrow n'^{t}_{\ddot{\mathsf{A}}})$ and $\xi_{\mathsf{C}} = (n_{\mathsf{C}} \twoheadrightarrow n'^{t}_{\mathsf{C}})$ be paths in Ä and C. A **Hoare triple**, written $\{pre\}(\xi_{\ddot{\mathsf{A}}};\xi_{\mathsf{C}})\{post\}$, denotes the statement: if execution starts at node n_{X} in state σ such that predicate $pre(\sigma)$ holds, and if paths $\xi_{\ddot{\mathsf{A}}};\xi_{\mathsf{C}}$ are executed in sequence to completion finishing in*

state σ'*, then predicate* $post(\sigma')$ *holds.*

We define *path infeasibility* and *path cover* in terms of Hoare triple(s).

Definition 3.3.2 (Path infeasibility)**.** *At a node* $n_{\mathsf{X}} = (n_{\ddot{\mathsf{A}}}, n_{\mathsf{C}}) \in \mathcal{N}_{\mathsf{X}}^{\mathcal{UW}}$*, a path* $\xi_{\ddot{\mathsf{A}}} = (n_{\ddot{\mathsf{A}}} \twoheadrightarrow n_{\ddot{\mathsf{A}}}^{t})$ *is* ***infeasible*** *at* n_{X} *iff the Hoare triple* $\{\phi_{n_{\mathsf{X}}}\}(\xi_{\ddot{\mathsf{A}}}; \epsilon)\{\texttt{false}\}$ *holds.*

Definition 3.3.3 (Path cover)**.** *At a node* $n_{\mathsf{X}} = (n_{\ddot{\mathsf{A}}}, n_{\mathsf{C}}) \in \mathcal{N}_{\mathsf{X}}^{\mathcal{UW}}$*, for a path* $\xi_{\ddot{\mathsf{A}}} = (n_{\ddot{\mathsf{A}}} \twoheadrightarrow n_{\ddot{\mathsf{A}}}^{t})$*, let* $\forall_{1 \le j \le m} : e_{\mathsf{X}}^{j} = (n_{\mathsf{X}} \xrightarrow{\xi_{\ddot{\mathsf{A}}}; \xi_{\mathsf{C}}^{j}} n_{\mathsf{X}}^{t_j})$ *be all edges in* $\mathcal{E}_{\mathsf{X}}$*, such that* $n_{\mathsf{X}}^{t_j} = (n_{\ddot{\mathsf{A}}}^{t}, n_{\mathsf{C}}^{t_j})$*. The set of edges* $\{e_{\mathsf{X}}^{1}, e_{\mathsf{X}}^{2}, \ldots, e_{\mathsf{X}}^{m}\}$ ***covers path*** $\xi_{\ddot{\mathsf{A}}}$*, written* $\{e_{\mathsf{X}}^{1}, e_{\mathsf{X}}^{2}, \ldots, e_{\mathsf{X}}^{m}\}\langle \mathcal{D}_{\mathsf{X}}, \xi_{\ddot{\mathsf{A}}} \rangle$*, iff the Hoare triple* $\{\phi_{n_{\mathsf{X}}}\}(\xi_{\ddot{\mathsf{A}}}; \epsilon)\{\bigvee_{j=1}^{m} pathcond([\xi_{\mathsf{C}}^{j}]_{\mathcal{D}_{\mathsf{X}}}^{e_{\mathsf{X}}^{j}})\}$ *holds.*

In other words, the set of edges $\{e_{\mathsf{X}}^{1}, \ldots, e_{\mathsf{X}}^{j}, \ldots, e_{\mathsf{X}}^{m}\}$ covers the path $\xi_{\ddot{\mathsf{A}}}$ iff: whenever an execution starting at n_{X} in state σ such that $\phi_{n_{\mathsf{X}}}(\sigma)$ holds completes the execution of $\xi_{\ddot{\mathsf{A}}}$, a subsequent execution starting at n_{C} must execute at least one of the determinized path $[\xi_{\mathsf{C}}^{j}]_{\mathcal{D}_{\mathsf{X}}}^{e_{\mathsf{X}}^{j}}$ $(1 \le j \le m)$ to completion. Thus, if path cover at n_{X} for path $\xi_{\ddot{\mathsf{A}}}$ holds, then at least one of the outgoing edges at n_{X} will execute to completion.

3.3.1 X requirements

Let _ be a wildcard character for a node. We define the following requirements on X so it may witness C ⊒ Ä:

1. (MutexÄ): For each node $n_{\mathsf{X}} \in \mathcal{N}_{\mathsf{X}}$, with *all* outgoing edges $\{e_{\mathsf{X}}^{1}, e_{\mathsf{X}}^{2}, \ldots, e_{\mathsf{X}}^{m}\}$ such that $e_{\mathsf{X}}^{j} = (n_{\mathsf{X}} \xrightarrow{\xi_{\ddot{\mathsf{A}}}^{j}; \xi_{\mathsf{C}}^{j}} n_{\mathsf{X}}^{t_j}) \in \mathcal{E}_{\mathsf{X}}$ and $n_{\mathsf{X}}^{t_j} \in \mathcal{N}_{\mathsf{X}}$ (for $1 \le j \le m$), the following holds:
$$\forall_{1 \le j_1, j_2 \le m} : (\xi_{\ddot{\mathsf{A}}}^{j_1} = \xi_{\ddot{\mathsf{A}}}^{j_2}) \vee (\xi_{\ddot{\mathsf{A}}}^{j_1} \fatsemi \xi_{\ddot{\mathsf{A}}}^{j_2})$$
In other words, two Ä paths $\xi_{\ddot{\mathsf{A}}}^{j_1}$, $\xi_{\ddot{\mathsf{A}}}^{j_2}$ $(1 \le j_1, j_2 \le m)$ originating at node n_{X} are either identical or mutually exclusive.

2. (MutexC): At each node $n_{\mathsf{X}} = (n_{\ddot{\mathsf{A}}}, n_{\mathsf{C}}) \in \mathcal{N}_{\mathsf{X}}$, for some path $\xi_{\ddot{\mathsf{A}}} = (n_{\ddot{\mathsf{A}}} \twoheadrightarrow n_{\ddot{\mathsf{A}}}^{t})$, let $\{e_{\mathsf{X}}^{1}, e_{\mathsf{X}}^{2}, \ldots, e_{\mathsf{X}}^{m}\}$ be a set of *all* outgoing edges such that $e_{\mathsf{X}}^{j} = (n_{\mathsf{X}} \xrightarrow{\xi_{\ddot{\mathsf{A}}}; \xi_{\mathsf{C}}^{j}} n_{\mathsf{X}}^{t_j}) \in \mathcal{E}_{\mathsf{X}}$ (for $1 \le j \le m$) and $n_{\mathsf{X}}^{t_j} = (n_{\ddot{\mathsf{A}}}^{t}, n_{\mathsf{C}}^{t_j}) \in \mathcal{N}_{\mathsf{X}}$. Then, the set $\{\xi_{\mathsf{C}}^{1}, \xi_{\mathsf{C}}^{2}, \ldots, \xi_{\mathsf{C}}^{m}\}$ must be a pathset, i.e., the paths are pairwise mutually-exclusive.

   In other words, the set of paths correlated with a path $\xi_{\ddot{\mathsf{A}}}$ originating at n_{X} are pairwise mutually exclusive. Because mutually-exclusive paths cannot be executed simultaneously (by virtue of their pairwise complementary path conditions), together

(MutexÄ) and (MutexC) make execution of X *deterministic* such that at a node n_X, at most one outgoing edge e^j_X may be executed to completion[2].

3. (Termination) For each error-free node $n_X = (n_Ä, n_C) \in \mathcal{N}_X^{\mathcal{U}\mathcal{W}}$, $n_Ä$ is a terminating node iff n_C is a terminating node.

   (Termination) ensures error-free termination of both executions (C and Ä) happens simultaneously.

4. (SingleIO): For each edge $e_X = (n_X \xrightarrow{\xi_Ä;\xi_C} n^t_X) \in \mathcal{E}_X$, either both $\xi_Ä$ and ξ_C are I/O paths or both $\xi_Ä$ and ξ_C are I/O-free.

   (SingleIO) enforces lockstep execution of non-silent trace-producing instructions in $\xi_Ä$ and ξ_C.

5. (Similar-speed): Let $(e^1_X, e^2_X, \ldots, e^m_X)$ be a cyclic path, so that $\forall_{1\leq j\leq m} : e^j_X = (n^{f,j}_X \xrightarrow{\xi^j_Ä;\xi^j_C} n^{t,j}_X) \in \mathcal{E}_X$; $n^{f,1}_X = n^{t,m}_X$; and $\bigwedge_{j=1}^{m-1}(n^{t,j}_X = n^{f,j+1}_X)$. For each cyclic path, $\neg \bigwedge_{j=1}^{m}(\xi^j_Ä = \epsilon)$ and $\neg \bigwedge_{j=1}^{m}(\xi^j_C = \epsilon)$ holds.

   (SimilarSpeed) enforces divergence preservation: Ä diverges (i.e., does not terminate) iff C diverges.

6. (Well-formedness): If a node of the form $n_X = (\_, \mathcal{W}_C)$ exists in $\mathcal{N}_X$, then n_X must be $(\mathcal{W}_Ä, \mathcal{W}_C)$.

   (Well-formedness) ensures that if a well-formedness (WF) constraint is violated in C (indicated by transition to $\mathcal{W}_C$), then Ä must transition to $\mathcal{W}_Ä$ — recall that $\mathcal{W}$ is used to signal violation of a WF constraint in C, and violation of a condition that can be assumed to never happen, e.g. stack overflow, in Ä (section 2.2.4). In other words, an error-free execution in Ä must never falsify the WF constraints.

7. (Safety): If a node of the form $n_X = (\mathcal{U}_Ä, \_)$ exists in $\mathcal{N}_X$, then n_X must be $(\mathcal{U}_Ä, \mathcal{U}_C)$.

   (Safety) ensures that Ä triggers $\mathcal{U}$ only if C triggers $\mathcal{U}$ — recall that $\mathcal{U}$ is used to signal occurrence of an undefined behavior (UB) in C, and occurrence of UB or translation error in Ä (section 2.2.4). With (Safety), we ensure that Ä may have translation errors (or may trigger UB) only if a lockstep execution of C triggers UB.

[2]Recall that execution of X edge e^j_X is defined as execution of $\xi_Ä$ followed by execution of ξ^j_C.

8. (CoverageÄ): For each error-free node $n_X = (n_Ä, n_C) \in \mathcal{N}_X^{\mathcal{UW}}$ and for each possible outgoing path $\xi_Ä^o = (n_Ä \twoheadrightarrow n_Ä^o)$, either $\xi_Ä^o$ is infeasible at n_X or there exists $e_X = (n_X \xrightarrow{\xi_Ä;\xi_C} n_X^t) \in \mathcal{E}_X$ such that either $\xi_Ä$ is a prefix of $\xi_Ä^o$ or $\xi_Ä^o$ is a prefix of $\xi_Ä$.

   (CoverageÄ) ensures all executable paths of Ä are present or *covered* in X.

9. (CoverageC): At each node $n_X = (n_Ä, n_C) \in \mathcal{N}_X$, for some $\xi_Ä = (n_Ä \twoheadrightarrow n_Ä^t)$, let $\{e_X^1, e_X^2, \ldots, e_X^m\}$ be the set of *all* outgoing edges such that $e_X^j = (n_X \xrightarrow{\xi_Ä;\xi_C^j} n_X^{t_j}) \in \mathcal{E}_X$ (for $1 \leq j \leq m$) and $n_X^{t_j} = (n_Ä^t, n_C^{t_j}) \in \mathcal{N}_X$. Then, $\{e_X^1, e_X^2, \ldots, e_X^m\}\langle \mathcal{D}_X, \xi_Ä \rangle$ holds.

   (CoverageC) requires that at least one of the outgoing edge e_X^j must execute to completion. (MutexÄ), (MutexC), and (CoverageC) together ensure that at a node n_X exactly one outgoing edge (if any) may be executed to completion.

10. (Inductive): For each error-free edge (an edge with error-free destination node) $e_X = (n_X \xrightarrow{\xi_Ä;\xi_C} n_X^t) \in \mathcal{E}_X$ such that $n_X, n_X^t \in \mathcal{N}_X^{\mathcal{UW}}$, the Hoare triple $\{\phi_{n_X}\}(\xi_Ä; [\xi_C]_{\mathcal{D}_X}^{e_X})\{\phi_{n_X^t}\}$ holds.

    (Inductive) ensures that the invariant network Φ_X is inductively provable starting at start node $n_X^s \in \mathcal{N}_X$

11. (Equivalence): For each error-free node $n_X = (n_Ä, n_C) \in \mathcal{N}_X^{\mathcal{UW}}$, $\Omega_Ä = \Omega_C$ must belong to ϕ_{n_X}.

    (Equivalence) ensures that Ä and C produce identical non-silent traces in an error-free execution. Because our execution semantics observe (de)allocation in C and A (figs. 2.5, 2.8 and 2.10), (Equivalence) ensures that the allocation state (address sets) of common regions B in C and Ä is identical.

12. (Memory Access Correspondence) or (MAC): For each edge $e_X = (n_X \xrightarrow{\xi_Ä;\xi_C} n_X^t) \in \mathcal{E}_X$, such that $n_X^t \neq (\_, \mathcal{U}_C) \in \mathcal{N}_X$, the following Hoare triples hold:

    (a) $\{\phi_{n_X} \wedge (\boxed{\Sigma_Ä^{\texttt{rd}}} = \boxed{\Sigma_C^{\texttt{rd}}} = \emptyset)\}(\xi_Ä; [\xi_C]_{\mathcal{D}_X}^{e_X})\{(\boxed{\Sigma_Ä^{\texttt{rd}}} \setminus \boxed{\Sigma_C^{\texttt{rd}}}) \subseteq \Sigma_Ä^{G \cup F} \cup [\texttt{esp}, \boxed{\texttt{stk}_e}]\}$

    (b) $\{\phi_{n_X} \wedge (\boxed{\Sigma_Ä^{\texttt{wr}}} = \boxed{\Sigma_C^{\texttt{wr}}} = \emptyset)\}(\xi_Ä; [\xi_C]_{\mathcal{D}_X}^{e_X})\{(\boxed{\Sigma_Ä^{\texttt{wr}}} \setminus \boxed{\Sigma_C^{\texttt{wr}}}) \subseteq \Sigma_Ä^{G_w \cup F_w} \cup [\texttt{esp}, \boxed{\texttt{stk}_e}]\}$

    Recall that the ghost address sets $\boxed{\Sigma_Ä^{\texttt{rd}}}$ ($\boxed{\Sigma_C^{\texttt{rd}}}$) and $\boxed{\Sigma_Ä^{\texttt{wr}}}$ ($\boxed{\Sigma_C^{\texttt{wr}}}$) accumulate the memory accesses performed by Ä(C) during its execution. (MAC) effectively requires that for every (unique) memory access made to address α belonging to region $r \in \{hp, cl, cs\}$ during execution of path $\xi_Ä$, there exists an access to α of the same read/write type on determinized path $[\xi_C]_{\mathcal{D}_X}^{e_X}$ that also executes to completion.

This requirement enables a sound over-approximation of the set of addresses belonging to hp, cl, and cs for a faster SMT encoding (described later in theorem 3.5.3 and section 5.3.2). For (MAC) to be meaningful, $\Sigma^{\mathtt{rd}}_{\ddot{A},C}$ and $\Sigma^{\mathtt{wr}}_{\ddot{A},C}$ must not be included in X's state elements over which a node invariant ϕ_{n_X} is inferred.

13. (MemEq): For each error-free node $n_X \in \mathcal{N}_X^{\mathcal{UW}}$, $M_{\ddot{A}} =_{\Sigma^B_{\ddot{A}} \setminus (\Sigma^{Z_l|^v}_{\ddot{A}})} M_C$ must belong to ϕ_{n_X}.

    (MemEq) requires the memory state of common regions modulo virtually-allocated locals ($\Sigma^B_{\ddot{A}} \setminus \Sigma^{Z_l|^v}_{\ddot{A}}$)[3] of C and A to be identical at a node n_X so that these regions are mutated identically in a lockstep fashion in both C and Ä.

    This requirement enables an efficient search algorithm which trades some completeness (by rejecting *sound* product graphs which do not respect this requirement) for a more efficient incremental exploration for the required product graph.

The first seven requirements are constraints on the graph structure of X and are referred to as *structural requirements*. The remaining six require discharge of proof obligations (in the form of Hoare triples) and are referred to as *semantic requirements*.

The first eleven requirements are *soundness requirements* that are required for witnessing refinement through X (section 3.3.2). The first twelve requirements are *fast-encoding requirements* that enable a faster SMT encoding (chapter 5). All thirteen are *search-algorithm requirements* that enable product graph search optimizations. Excluding (CoverageÄ) and (CoverageC), the remaining eleven are called *non-coverage requirements*.

3.3.2 Soundness of X requirements

Let $X = \ddot{A} \boxtimes C = (\mathcal{N}_X, \mathcal{E}_X, \mathcal{D}_X)$ be a determinized product graph that satisfies the soundness requirements (first eleven requirements in section 3.3.1).

Lemma 3.3.4 (X's execution)**.** *The following holds for an execution of* X*:* [4]

$$\begin{aligned} \forall \Omega, T'_{\ddot{A}}, T'_C : (X \downarrow_\Omega (T'_{\ddot{A}}, T'_C)) \Rightarrow \quad & T'_{\ddot{A}} =_{st} T'_C \\ & \vee ((e(T'_{\ddot{A}}) = \mathscr{W}) \wedge (\tilde{e}(T'_{\ddot{A}}) \leq_{st} T'_C)) \\ & \vee ((e(T'_C) = \mathscr{U}) \wedge (\tilde{e}(T'_C) \leq_{st} T'_{\ddot{A}})) \end{aligned} \tag{3.1}$$

[3] Recall that $\Sigma^{Z_l|^v}_{\ddot{A}}$ is defined as the union of address sets of virtually-allocated locals in Ä (section 2.6.1)

[4] The relations ($\downarrow$), $=_{st}$, and $\leq_{st}$ are defined in section 2.4.

Proof. The proof proceeds through a coinduction on the number of edges executed by X. We prove that the execution of a single edge $e_{\mathsf{X}} = (n_{\mathsf{X}} \xrightarrow{\xi_{\ddot{\mathsf{A}}};\xi_{\mathsf{C}}} n^t_{\mathsf{X}}) \in \mathcal{E}_{\mathsf{X}}$, starting at an error-free node $n_{\mathsf{X}} \in \mathcal{N}^{\not\mathcal{U}\not\mathcal{W}}_{\mathsf{X}}$ in a state that satisfies $T'_{\ddot{\mathsf{A}}} =_{st} T'_{\mathsf{C}}$, either reaches a terminating node n^t_{X}, such that final state satisfies the RHS of the $\Rightarrow$ in eq. (3.1), or reaches a non-terminating node n^t_{X}, such that $T'_{\ddot{\mathsf{A}}} =_{st} T'_{\mathsf{C}}$ holds at the end of execution of e_{X}.

Let $\{e^1_{\mathsf{X}}, e^2_{\mathsf{X}}, \ldots, e^m_{\mathsf{X}}\}$ be the set of *all* outgoing edges at error-free node $n_{\mathsf{X}} \in \mathcal{N}^{\not\mathcal{U}\not\mathcal{W}}_{\mathsf{X}}$ such that $\forall_{1 \le j \le m} : e^j_{\mathsf{X}} = (n_{\mathsf{X}} \xrightarrow{\xi^j_{\ddot{\mathsf{A}}};\xi^j_{\mathsf{C}}} n^j_{\mathsf{X}}) \in \mathcal{E}_{\mathsf{X}}$. Due to (SingleIO) there can be two cases:

1. **$\xi^j_{\ddot{\mathsf{A}}}$ and ξ^j_{C} are I/O paths.** Because I/O paths are straight-line sequences of instructions (with no branching), it must be true that $j = m = 1$. Further, an I/O path can only end at an error-free node n^j_{X} that must satisfy (Equivalence) requirement. (Equivalence) (through $\Omega_{\ddot{\mathsf{A}}} = \Omega_{\mathsf{C}}$) implies production of identical non-silent trace events. Therefore, the claim holds.

2. **$\xi^j_{\ddot{\mathsf{A}}}$ and ξ^j_{C} are I/O-free.** Due to (Mutex$\ddot{\mathsf{A}}$) and (Coverage$\ddot{\mathsf{A}}$), it must be possible to execute a path $\xi^j_{\ddot{\mathsf{A}}}$ to completion. Due to (Coverage$\mathsf{C}$), there exists some outgoing edge $e^j_{\mathsf{X}} = (n_{\mathsf{X}} \xrightarrow{\xi^j_{\ddot{\mathsf{A}}};\xi^j_{\mathsf{C}}} n^j_{\mathsf{X}}) \in \mathcal{E}_{\mathsf{X}}$ that is executed to completion. Further, due to (Mutex$\mathsf{C}$), such an edge e^j_{X} must be unique. The execution of $\xi^j_{\ddot{\mathsf{A}}}$ followed by execution of ξ^j_{C} effectively causes X to execute e^j_{X} and reach node $n^j_{\mathsf{X}} = (n^j_{\ddot{\mathsf{A}}}, n^j_{\mathsf{C}})$.

   The execution of $\xi^j_{\ddot{\mathsf{A}}}$ may end at either: (1) the error node $\mathcal{W}_{\ddot{\mathsf{A}}}$, (2) the error node $\mathcal{U}_{\ddot{\mathsf{A}}}$, (3) an error-free node $n^j_{\ddot{\mathsf{A}}}$.

   - In case (1), the execution ends at the error node $\mathcal{W}_{\ddot{\mathsf{A}}}$. Because the traces were stuttering equivalent before the execution of e^j_{X} (coinduction hypothesis), and the execution of $\xi^j_{\ddot{\mathsf{A}}}$ must only produce the $\mathcal{W}$ trace event (as $\xi^j_{\ddot{\mathsf{A}}}$ is I/O-free and cannot contain `rd`/`wr` instructions), $(e(T'_{\ddot{\mathsf{A}}}) = \mathcal{W} \wedge \tilde{e}(T_{\ddot{\mathsf{A}}}) \le_{st} T'_{\mathsf{C}})$ will hold.
   - In case (2), due to the (Safety) requirement, execution of e^j_{X} must reach node $n^{t_j}_{\mathsf{X}} = (\mathcal{U}_{\ddot{\mathsf{A}}}, \mathcal{U}_{\mathsf{C}})$. Moreover, the execution $\xi^j_{\ddot{\mathsf{A}}}$ and ξ^j_{C} must only generate the error code $\mathcal{U}$ as a trace event (as both $\xi^j_{\ddot{\mathsf{A}}}$ and ξ^j_{C} are I/O-free). Because the traces were stuttering equivalent before the execution of e^j_{X} (coinduction hypothesis), $(e(T'_{\mathsf{C}}) = \mathcal{U} \wedge \tilde{e}(T'_{\mathsf{C}}) \le_{st} T'_{\ddot{\mathsf{A}}})$ will hold.
   - In case (3), we analyze each possibility of n^j_{X} separately. n^j_{X} must be of one of the following forms: (a) $(n^j_{\ddot{\mathsf{A}}}, \mathcal{W}_{\mathsf{C}})$, (b) $(n^j_{\ddot{\mathsf{A}}}, \mathcal{U}_{\mathsf{C}})$, or (c) an error-free node $(n^j_{\ddot{\mathsf{A}}}, n^j_{\mathsf{C}})$, where n^j_{C} is an error-free node (recall that $n^j_{\ddot{\mathsf{A}}}$ is also an error-free node in this case).

Case (a) cannot occur due to the (Well-formedness) requirement. In case (b), $(e(T'_{\mathsf{C}}) = \mathscr{U} \wedge \tilde{e}(T'_{\mathsf{C}}) \leq_{st} T'_{\ddot{\mathsf{A}}})$ holds due to $\xi^j_{\ddot{\mathsf{A}}}$ and ξ^j_{C} being I/O-free and coinduction hypothesis (similar reasoning as case (2) above). In case (c), due to $\xi^j_{\ddot{\mathsf{A}}}$ and ξ^j_{C} being I/O-free, their execution cannot produce any non-silent trace event. Thus, due to coinduction hypothesis, $T'_{\ddot{\mathsf{A}}} =_{st} T'_{\mathsf{C}}$ must hold at n^j_{X}.

Finally, consider the case when ξ^j_{C} is ϵ and $\xi^j_{\ddot{\mathsf{A}}}$ is not. Due to (Similar-speed), there exists a finite sequence of edges $(e^{x_1}_{\mathsf{X}}, e^{x_2}_{\mathsf{X}}, \ldots, e^{x_n}_{\mathsf{X}})$ such that $\forall_{1\leq i\leq n} : e^{x_i}_{\mathsf{X}} = (n^{f,i}_{\mathsf{X}} \xrightarrow{\xi^{x_i}_{\ddot{\mathsf{A}}};\xi^{x_i}_{\mathsf{C}}} n^{t,i}_{\mathsf{X}}) \in \mathcal{E}_{\mathsf{X}}$, $e^j_{\mathsf{X}} = e^{x_1}_{\mathsf{X}}$, $\forall_{1\leq i<n} : \xi^{x_i}_{\mathsf{C}} = \epsilon$, and $\xi^{x_n}_{\mathsf{C}} \neq \epsilon$. Similar argument can be used when $\xi^j_{\ddot{\mathsf{A}}}$ is ϵ and ξ^j_{C} is not. (Similar-speed) thus ensures that the silent events in both traces differ only by a finite amount, thereby upholding $T'_{\ddot{\mathsf{A}}} =_{st} T'_{\mathsf{C}}$. □

Lemma 3.3.5 (X's trace is derived from C's trace)**.** *The following holds for an execution of* X*:*

$$\begin{aligned}\forall \Omega, T'_{\ddot{\mathsf{A}}}, T'_{\mathsf{C}} : (\mathsf{X} \downarrow_{\Omega} (T'_{\ddot{\mathsf{A}}}, T'_{\mathsf{C}})) \Rightarrow \exists T_{\mathsf{C}} : \quad & (\mathsf{C} \downarrow_{\Omega} T_{\mathsf{C}}) \\ & \wedge (\quad T'_{\mathsf{C}} =_{st} T_{\mathsf{C}} \\ & \quad \vee ((e(T'_{\ddot{\mathsf{A}}}) = \mathscr{W}) \wedge (\tilde{e}(T'_{\ddot{\mathsf{A}}}) \leq_{st} T_{\mathsf{C}})))\end{aligned}$$

Proof. The proof proceeds through a coinduction on the number of edges executed by X. Suppose X and C start execution with states $\sigma_{\mathsf{X}} = (\sigma_{\ddot{\mathsf{A}}}, \sigma_{\mathsf{C}})$ and σ_{C} at error-free nodes $n_{\mathsf{X}} = (n_{\ddot{\mathsf{A}}}, n_{\mathsf{C}}) \in \mathcal{N}^{\mathscr{U}\mathscr{W}}_{\mathsf{X}}$ and $n_{\mathsf{C}} \in \mathcal{N}^{\mathscr{U}\mathscr{W}}_{\mathsf{C}}$ respectively, such that $T_{\mathsf{C}} =_{st} T'_{\mathsf{C}}$, where $T_{\mathsf{C}} \in \sigma_{\mathsf{C}}$ and $(T'_{\ddot{\mathsf{A}}}, T'_{\mathsf{C}}) \in \sigma_{\mathsf{X}}$, holds.

Consider the execution of edge $e_{\mathsf{X}} = (n_{\mathsf{X}} \xrightarrow{\xi_{\ddot{\mathsf{A}}};\xi_{\mathsf{C}}} n^t_{\mathsf{X}}) \in \mathcal{E}_{\mathsf{X}}$, starting at n_{X} on state σ_{X}. If ξ_{C} is executed, as part of e_{X}'s execution, using some sequence of non-deterministic choices determined by $\mathcal{D}_{\mathsf{X}}$, the same path ξ_{C} can be executed in C for the same sequence of non-deterministic choices. As both executions start in identical states, they will produce identical sequence of trace events till execution reaches the sink node n^t_{C} where $T_{\mathsf{C}} =_{st} T'_{\mathsf{C}}$ will hold (note that execution of $\xi_{\ddot{\mathsf{A}}}$ may not modify the state elements of C in σ_{X} as both have disjoint state space).

If ξ_{C} is ϵ, then, due to (Similar-speed), there exists a finite sequence of edges $(e^1_{\mathsf{X}}, e^2_{\mathsf{X}}, \ldots, e^m_{\mathsf{X}})$ such that $e_{\mathsf{X}} = e^1_{\mathsf{X}}$, $\forall_{1\leq j\leq m} : e^j_{\mathsf{X}} = (n^{f,j}_{\mathsf{X}} \xrightarrow{\xi^j_{\ddot{\mathsf{A}}};\xi^j_{\mathsf{C}}} n^{t,j}_{\mathsf{X}}) \in \mathcal{E}_{\mathsf{X}}$, $\forall_{1\leq j<m} : \xi^j_{\mathsf{C}} = \epsilon$ and $\xi^m_{\mathsf{C}} \neq \epsilon$. Let n^t_{X} be $n^{t,m}_{\mathsf{X}}$ in this case.

If $n^t_{\mathsf{X}} = (n^t_{\ddot{\mathsf{A}}}, n^t_{\mathsf{C}})$ is a non-terminating node, then the claim holds due to the coinduction

hypothesis. Similarly, if both n^t_{X} and n^t_{C} are terminating nodes, then the claim holds by definition.

Consider the case when $n^t_{\mathsf{X}} = (n^t_{\ddot{\mathsf{A}}}, n^t_{\mathsf{C}})$ is a terminating node (due to $n^t_{\ddot{\mathsf{A}}}$ being a terminating node) but n^t_{C} is not a terminating node. There are three possibilities for $n^t_{\ddot{\mathsf{A}}}$ in this case:

- $n^t_{\ddot{\mathsf{A}}} = \mathscr{W}_{\ddot{\mathsf{A}}}$: Due to (Equivalence), $T'_{\ddot{\mathsf{A}}} =_{st} T'_{\mathsf{C}}$ holds at n_{X}. Further, due to (SingleIO), $\xi_{\ddot{\mathsf{A}}}$ cannot produce any non-silent trace event other than $\mathscr{W}$. Hence, $T'_{\ddot{\mathsf{A}}} \leq_{st} T_{\mathsf{C}}$ holds due to coinduction hypothesis.

- $n^t_{\ddot{\mathsf{A}}} = \mathscr{U}_{\ddot{\mathsf{A}}}$: Due to (Safety), $n^t_{\mathsf{X}} = (n^t_{\ddot{\mathsf{A}}}, n^t_{\mathsf{C}})$ must be of the form $(\mathscr{U}_{\ddot{\mathsf{A}}}, \mathscr{U}_{\mathsf{C}})$. However, this violates the assumption that n^t_{C} is a non-terminating node. Therefore, this case is not possible.

- $n^t_{\ddot{\mathsf{A}}}$ is an error-free terminating node: This case is not possible due to (Termination) requiring n^t_{C} to be error-free terminating node whenever $n^t_{\ddot{\mathsf{A}}}$ is an error-free terminating node.

□

Lemma 3.3.6 ($\ddot{\mathsf{A}}$'s traces are in X)**.** *The following holds for an execution of* $\ddot{\mathsf{A}}$*:*

$$\begin{aligned}\forall \Omega, T_{\ddot{\mathsf{A}}} : (\ddot{\mathsf{A}} \downarrow_{\Omega} T_{\ddot{\mathsf{A}}}) \Rightarrow \exists T'_{\ddot{\mathsf{A}}}, T'_{\mathsf{C}} : \quad & (\mathsf{X} \downarrow_{\Omega} (T'_{\ddot{\mathsf{A}}}, T'_{\mathsf{C}})) \\ & \wedge \big(\quad T_{\ddot{\mathsf{A}}} =_{st} T'_{\ddot{\mathsf{A}}} \\ & \quad \vee ((e(T'_{\mathsf{C}}) = \mathscr{U}) \wedge (e(T'_{\ddot{\mathsf{A}}}) \neq \mathscr{W}) \wedge (\tilde{e}(T'_{\mathsf{C}}) \leq_{st} T_{\ddot{\mathsf{A}}}))\big)\end{aligned} \tag{3.2}$$

Proof. Consider an execution of X that is currently at an error-free node $n_{\mathsf{X}} = (n_{\ddot{\mathsf{A}}}, n_{\mathsf{C}}) \in \mathcal{N}^{\mathcal{U}\mathscr{W}}_{\mathsf{X}}$. We show by coinduction on the number of edges executed in $\ddot{\mathsf{A}}$, starting at $n_{\ddot{\mathsf{A}}}$, that eq. (3.2) holds. The proof of the lemma follows by using $n_{\mathsf{X}} = n^s_{\mathsf{X}} = (n^s_{\ddot{\mathsf{A}}}, n^s_{\mathsf{C}}) \in \mathcal{N}_{\mathsf{X}}$.

Due to (Mutex$\ddot{\mathsf{A}}$), (Mutex$\mathsf{C}$), (Coverage$\ddot{\mathsf{A}}$), and (Coverage$\mathsf{C}$), there exists exactly one $e_{\mathsf{X}} = (n_{\mathsf{X}} \xrightarrow{\xi_{\ddot{\mathsf{A}}};\xi_{\mathsf{C}}} n^t_{\mathsf{X}}) \in \mathcal{E}_{\mathsf{X}}$ such that $\xi_{\ddot{\mathsf{A}}}$ and ξ_{C} execute to completion to reach $n^t_{\mathsf{X}} = (n^t_{\ddot{\mathsf{A}}}, n^t_{\mathsf{C}}) \in \mathcal{N}_{\mathsf{X}}$. Consider the following two cases for $\xi_{\ddot{\mathsf{A}}}$:

- $\xi_{\ddot{\mathsf{A}}} \neq \epsilon$: If n^t_{X} is an error-free node, then for each non-deterministic choice that an execution of $\ddot{\mathsf{A}}$ can make to execute to completion, an execution of $\ddot{\mathsf{A}}$ as part of X can make as well. Thus, due to coinduction hypothesis, both executions can produce

identical observables events such that $T_{\ddot{A}} =_{st} T'_{\ddot{A}}$ holds. If $n^t_C = \mathscr{W}_C$, then $n^t_{\ddot{A}}$ must also be $\mathscr{W}_{\ddot{A}}$ due to (Well-formedness), and $T_{\ddot{A}} =_{st} T'_{\ddot{A}}$ holds due to (SingleIO) and coinduction hypothesis. If $n^t_C = \mathscr{U}_C$ and $n^t_{\ddot{A}} = \mathscr{W}_{\ddot{A}}$, $T_{\ddot{A}} =_{st} T'_{\ddot{A}}$ holds due to (SingleIO) and coinduction hypothesis. If $n^t_C = \mathscr{U}_C$ and $n^t_{\ddot{A}} \neq \mathscr{W}_{\ddot{A}}$, then $\tilde{e}(T'_C) \leq_{st} T_{\ddot{A}}$ holds due to (SingleIO) and coinduction hypothesis. $n^t_C \neq \mathscr{U}_C$ and $n^t_{\ddot{A}} = \mathscr{U}_{\ddot{A}}$ is not possible due to (Safety).

- $\xi_{\ddot{A}} = \epsilon$: Execute k edges in X before a non-ϵ path is encountered, where k is the length of the longest sequence of edges in X such that an edge $e_X = (n_X \xrightarrow{\xi_{\ddot{A}};\xi_C} n^t_X)$ with $\xi_{\ddot{A}} \neq \epsilon$ is reached; then repeat the coinductive step above. Due to (Similar-speed), k must be defined.

□

Theorem 3.3.7 (X witnesses $C \sqsupseteq \ddot{A}$)**.** *If there exists* $X = \ddot{A} \boxtimes C$ *that satisfies the soundness requirements, then* $C \sqsupseteq \ddot{A}$ *holds.*

Proof. Consider an execution of $\ddot{A}$ under world Ω. Using lemma 3.3.6, we have:

$$\begin{aligned}\forall \Omega, T_{\ddot{A}} : (\ddot{A} \downarrow_\Omega T_{\ddot{A}}) \Rightarrow \exists T'_{\ddot{A}}, T'_C : \quad & (X \downarrow_\Omega (T'_{\ddot{A}}, T'_C)) \\ & \wedge (\quad T_{\ddot{A}} =_{st} T'_{\ddot{A}} \\ & \quad \vee ((e(T'_C) = \mathscr{U}) \wedge (e(T'_{\ddot{A}}) \neq \mathscr{W}) \wedge (\tilde{e}(T'_C) \leq_{st} T_{\ddot{A}})))\end{aligned} \tag{3.3}$$

Instantiating lemma 3.3.5, we obtain,

$$\begin{aligned}\forall \Omega, T_{\ddot{A}} : (\ddot{A} \downarrow_\Omega T_{\ddot{A}}) \Rightarrow \exists T'_{\ddot{A}}, T'_C : \quad & (X \downarrow_\Omega (T'_{\ddot{A}}, T'_C)) \\ & \wedge (\quad T_{\ddot{A}} =_{st} T'_{\ddot{A}} \\ & \quad \vee ((e(T'_C) = \mathscr{U}) \wedge (e(T'_{\ddot{A}}) \neq \mathscr{W}) \wedge (\tilde{e}(T'_C) \leq_{st} T_{\ddot{A}}))) \\ & \wedge (\exists T_C : \quad (C \downarrow_\Omega T_C) \\ & \qquad \wedge (\quad T'_C =_{st} T_C \\ & \qquad \quad \vee ((e(T'_{\ddot{A}}) = \mathscr{W}) \wedge (\tilde{e}(T'_{\ddot{A}}) \leq_{st} T_C))))\end{aligned} \tag{3.4}$$

We consider each minterm in the sum-of-products representation of the following terms

in the RHS of eq. (3.4):

$$\begin{aligned}
&(\quad T_{\ddot{A}} =_{st} T'_{\ddot{A}} \\
&\quad \vee ((e(T'_C) = \mathcal{U}) \wedge (e(T'_{\ddot{A}}) \neq \mathcal{W}) \wedge (\tilde{e}(T'_C) \leq_{st} T_{\ddot{A}}))) \\
\wedge\ &(\quad T'_C =_{st} T_C \\
&\quad \vee ((e(T'_{\ddot{A}}) = \mathcal{W}) \wedge (\tilde{e}(T'_{\ddot{A}}) \leq_{st} T_C)))
\end{aligned}$$

1. $(T_{\ddot{A}} =_{st} T'_{\ddot{A}}) \wedge (T'_C =_{st} T_C)$ holds.

   Instantiating lemma 3.3.4 in eq. (3.4), there are three cases:

   (a) $T'_{\ddot{A}} =_{st} T'_C$ holds.

   Due to $=_{st}$ being an equivalence relation, we have $T_{\ddot{A}} =_{st} T_C$ and, therefore, $C \sqsupseteq \ddot{A}$ holds.

   (b) $(e(T'_{\ddot{A}}) = \mathcal{W}) \wedge (\tilde{e}(T'_{\ddot{A}}) \leq_{st} T'_C)$ holds.

   As $=_{st}$ is congruent with respect to $\leq_{st}$, we have $(e(T_{\ddot{A}}) = \mathcal{W}) \wedge (\tilde{e}(T_{\ddot{A}}) \leq_{st} T_C)$, which is equivalent to $W^{\Omega,T_{\ddot{A}}}_{\mathtt{pre}}(C)$. Therefore, $C \sqsupseteq \ddot{A}$ holds.

   (c) $(e(T'_C) = \mathcal{U}) \wedge (e(T'_{\ddot{A}}) \neq \mathcal{W}) \wedge (\tilde{e}(T'_C) \leq_{st} T'_{\ddot{A}})$ holds.

   Using congruence of $=_{st}$ with respect to $\leq_{st}$, we have $(e(T_C) = \mathcal{U}) \wedge (\tilde{e}(T_C) \leq_{st} T_{\ddot{A}})$, which is equivalent to $U^{\Omega,T_{\ddot{A}}}_{\mathtt{pre}}(C)$. Therefore, $C \sqsupseteq \ddot{A}$ holds.

2. $(T_{\ddot{A}} =_{st} T'_{\ddot{A}}) \wedge ((e(T'_{\ddot{A}}) = \mathcal{W}) \wedge (\tilde{e}(T'_{\ddot{A}}) \leq_{st} T_C))$ holds.

   Using definition of $=_{st}$ and congruence of $=_{st}$ with respect to $\leq_{st}$, we have $(e(T_{\ddot{A}}) = \mathcal{W}) \wedge (\tilde{e}(T_{\ddot{A}}) \leq_{st} T_C)$, which is equivalent to $W^{\Omega,T_{\ddot{A}}}_{\mathtt{pre}}(C)$. Therefore, $C \sqsupseteq \ddot{A}$ holds.

3. $((e(T'_C) = \mathcal{U}) \wedge (\tilde{e}(T'_C) \leq_{st} T_{\ddot{A}})) \wedge (T'_C =_{st} T_C)$ holds.

   Using definition of $=_{st}$ and congruence of $=_{st}$ with respect to $\leq_{st}$, we have $(e(T_C) = \mathcal{U}) \wedge (\tilde{e}(T_C) \leq_{st} T_{\ddot{A}})$, which is equivalent to $U^{\Omega,T_{\ddot{A}}}_{\mathtt{pre}}(C)$. Therefore, $C \sqsupseteq \ddot{A}$ holds.

4. $((e(T'_C) = \mathcal{U}) \wedge (e(T'_{\ddot{A}}) \neq \mathcal{W}) \wedge (\tilde{e}(T'_C) \leq_{st} T_{\ddot{A}})) \wedge ((e(T'_{\ddot{A}}) = \mathcal{W}) \wedge (\tilde{e}(T'_{\ddot{A}}) \leq_{st} T_C))$ holds.

   This case is not possible due to the mutually unsatisfiable clauses $\ldots \wedge (e(T'_{\ddot{A}}) \neq \mathcal{W}) \wedge \ldots \wedge (e(T'_{\ddot{A}}) = \mathcal{W}) \wedge \ldots$.

□

3.3.3 Global Invariants in C, Ä, and X

Definition 3.3.8 (Non-entry Node)**.** *Let $P \in \{Ä, C\}$. A node $n_P \in \mathcal{N}_P$ is called a* ***non-entry node*** *iff it does not correspond to a node due to* (Entry$_C$) *and* (Entry$_Ä$) *(figs. 2.5 and 2.7) in P. A node $n_X = (n_Ä, n_C) \in \mathcal{N}_X$ is called a non-entry node iff both $n_Ä$ and n_C are non-entry nodes.*

Due to the execution semantics of Ä and C, certain invariants hold by construction in Ä and C. We call these invariants *global invariants* as they hold at each error-free, non-entry node.

Theorem 3.3.9 (Global Invariants in Ä)**.** *The following invariants hold at each error-free, non-entry node $n_Ä \in \mathcal{N}_Ä^{\mathcal{UW}}$:*

- *(*$\boxed{\texttt{em.}f}$ *tracks emptiness)* $\Sigma_Ä^f = \emptyset \Leftrightarrow \boxed{\texttt{em.}f}$*, for $f \in F$. Note that because $|\Sigma_Ä^f| > 0$ by definition,* $\boxed{\texttt{em.}f} = \texttt{false}$ *holds for all $f \in F$.*
- *(*$\boxed{\texttt{sz.}f}$ *tracks size)* $\boxed{\texttt{sz.}f} = |\Sigma_Ä^f| = \texttt{sz}(\texttt{T}(f))$ *for $f \in F$* [5]*.*
- *(Address sets of F are intervals)* $(\boxed{\texttt{em.}f} \vee [\boxed{\texttt{lb.}f}, \boxed{\texttt{ub.}f}] = \Sigma_Ä^f)$*, for $f \in F$. Or, using* $\boxed{\texttt{em.}f} = \texttt{false}$*,* $[\boxed{\texttt{lb.}f}, \boxed{\texttt{ub.}f}] = \Sigma_Ä^f$*.*
- *(Alignment of f)* $\texttt{aligned}_{\texttt{algnmnt}(f)}(\boxed{\texttt{lb.}f})$*, for $f \in F$* [6]*.*
- *(Stack bounds)* $\Sigma_Ä^{\{stk\}\cup Y} \cup (\Sigma_Ä^Z \setminus (\Sigma_Ä^{Z_l}|^v)) = [\texttt{esp}, \boxed{\texttt{stk}_e}]$*.*
- *(cs and cl)* $\Sigma_Ä^{\{cs,cl\}} = [\boxed{\texttt{stk}_e} + 1, \boxed{\texttt{cs}_e}]$
- *(Heap subset)* $\Sigma_Ä^{hp} \subseteq \texttt{comp}(\Sigma_Ä^{G\cup F\cup\{cv\}} \cup \Sigma_Ä^{Z_l}|^v \cup [\texttt{esp}, \boxed{\texttt{cs}_e}])$
- *(Disjoint regions in Ä)* $\neg\texttt{ov}(\Sigma_Ä^{hp}, \Sigma_Ä^{cl}, \Sigma_Ä^{cv}, \Sigma_Ä^{\texttt{vrdc}}, \ldots, \Sigma_Ä^{g}, \ldots, \Sigma_Ä^{y}, \ldots, \Sigma_Ä^{z}, \ldots)$ *and* $\neg\texttt{ov}(\Sigma_Ä^{hp}, \Sigma_Ä^{cl}, \Sigma_Ä^{\texttt{vrdc}}, \ldots, \Sigma_Ä^{g}, \ldots, \Sigma_Ä^{y}, \ldots, \Sigma_Ä^{z}|^s, \ldots, \Sigma_Ä^{f}, \ldots, \Sigma_Ä^{stk}, \Sigma_Ä^{cs})$*.*
- *(Read-only memory in Ä)* $M_Ä =_{i_Ä^r} \texttt{ROM}_Ä^r(i_Ä^r)$ *for $r \in F_r$.*

Proof sketch: By induction on the number of transitions executed in Ä with the base case defined by the first transition out of (Entry$_Ä$) in fig. 2.11. □

[5] Recall that $\texttt{sz}(\texttt{T}(r))$ gives the size in bytes of the variable named r (table 2.1).

[6] Recall that $\texttt{aligned}_{\texttt{algnmnt}(r)}(\boxed{\texttt{lb.}r})$ holds iff $\boxed{\texttt{lb.}r}$ is aligned by the alignment of variable r (table 2.1).

Theorem 3.3.10 (Global Invariants in C)**.** *The following invariants hold at each error-free, non-entry node* $n_C \in \mathcal{N}_C^{\mathcal{UW}}$*:*

- *(*$\boxed{\texttt{em.}r}$ *tracks emptiness)* $\Sigma_C^r = \emptyset \Leftrightarrow \boxed{\texttt{em.}r}$*, for* $r \in G \cup Y \cup Z$*. Note that* $\Sigma_C^r = \emptyset$ *may hold only for* $r \in \{\texttt{vrdc}\} \cup Z$*.*
- *(*$\boxed{\texttt{sz.}r}$ *tracks size)* $\boxed{\texttt{sz.}r} = |\Sigma_C^r|$ *for* $r \in G \cup Y$*.* $\boxed{\texttt{sz.}r} = \texttt{sz}(\texttt{T}(r))$*, for* $r \in G \cup (Y \setminus \{\texttt{vrdc}\})$*.*
- *(*$\boxed{\texttt{lstSz.}zl}$ *tracks size)* $\boxed{\texttt{lstSz.}zl} = |\Sigma_C^{zl}|$ *for* $zl \in Z_l$*.*
- *(*$\boxed{\texttt{lb.}r}$*,* $\boxed{\texttt{ub.}r}$ *track bounds)* $\boxed{\texttt{em.}r} \vee (\boxed{\texttt{lb.}r} = \texttt{lb}(\Sigma_C^r) \wedge \boxed{\texttt{ub.}r} = \texttt{ub}(\Sigma_C^r))$*, for* $r \in G \cup Y \cup Z$*.*
- *(Address sets of* G, Y, Z_l *are intervals)* $\boxed{\texttt{em.}r} \vee ([\boxed{\texttt{lb.}r}, \boxed{\texttt{ub.}r}] = \Sigma_C^r)$*, for* $r \in G \cup Y \cup Z_l$*. As a consequence, we have:* $\boxed{\texttt{em.}r} \vee ((\boxed{\texttt{lb.}r} \leq_u \boxed{\texttt{ub.}r}) \wedge (\boxed{\texttt{ub.}r} = \boxed{\texttt{lb.}r} + \boxed{\texttt{sz.}r} - 1_{\texttt{i32}}))$*, for* $r \in G \cup Y$ *and* $\boxed{\texttt{em.}zl} \vee ((\boxed{\texttt{lb.}zl} \leq_u \boxed{\texttt{ub.}zl}) \wedge (\boxed{\texttt{ub.}zl} = \boxed{\texttt{lb.}zl} + \boxed{\texttt{lstSz.}zl} - 1_{\texttt{i32}}))$ *for* $zl \in Z_l$*.*
- *(Alignment of* g *and* y*)* $\texttt{aligned}_{\texttt{algnmnt}(r)}(\boxed{\texttt{lb.}r})$*, for* $r \in G \cup (Y \setminus \{\texttt{vrdc}\})$*.*
- *(Disjoint regions in* C*)* $\neg\texttt{ov}(\Sigma_C^{hp}, \Sigma_C^{cl}, \Sigma_C^{cv}, \Sigma_C^{\texttt{vrdc}}, \ldots, \Sigma_C^{g}, \ldots, \Sigma_C^{y}, \ldots, \Sigma_C^{z}, \ldots)$*.*
- *(Read-only memory in* C*)* $M_C =_{i_C^r} \texttt{ROM}_C^r(i_C^r)$ *for* $r \in G_r$*.*

Proof sketch. By induction on the number of transitions executed in C with the base case defined by the first transition out of (ENTRY$_C$) in fig. 2.5 □

Theorem 3.3.11 (Global Invariants in X)**.** *Let* $X = Ä \boxtimes C$ *be a product graph that satisfies the search-algorithm requirements of section 3.3.1. The following invariants hold at an error-free, non-entry node* $n_X = (n_Ä, n_C) \in \mathcal{N}_X^{\mathcal{UW}}$*.*

1. *The invariants stated in theorems 3.3.9 and 3.3.10.*
2. *(Stack subset)* $\Sigma_Ä^{stk} \subseteq \Sigma_C^{\{cv,\texttt{free}\}} \cup \Sigma_Ä^{Z_l}|^v$

Proof sketch. Item 1 follows because n_X is an error-free, non-entry node iff both $n_Ä$ and n_C are error-free, non-entry nodes.

Item 2 follows from (Disjoint regions in Ä) of item 1 and the (Equivalence) requirement. □

3.4 Callers' Virtual Smallest Semantics

We are going to introduce two different semantics for C and A: callers' virtual smallest semantics (this section) and safety-relaxed semantics (section 3.5). These semantics are amenable to a faster SMT encoding and we separately prove that the theorems proved using these new semantics translate to theorems proved with the original semantics.

We construct C′ and A′ from C and A by using new *callers' virtual smallest semantics* where the cv region is made empty, i.e., $\Sigma^{cv}_{\mathsf{C}'} = \Sigma^{cv}_{\mathsf{A}'} = \emptyset$. With an empty cv, we compute the address set of region `free` as $\Sigma^{\texttt{free}}_P = \texttt{comp}(\Sigma^{B \cup F \cup S}_P)$ for $P \in \{\mathsf{C}', \mathsf{A}'\}$.

Formally, we obtain C′ and A′ from C and A by removing assignments to Σ^{cv}_{C} and Σ^{cv}_{A} due to (Entry$_{\mathsf{C}}$) and (Entry$_{\mathsf{A}}$) respectively (figs. 2.5 and 2.7) and replacing uses of Σ^{cv}_{C} and Σ^{cv}_{A} due to (Entry$_{\mathsf{C}}$), (Entry$_{\mathsf{A}}$), (Op-esp'), (Load$_{\ddot{\mathsf{A}}}$), (Store$_{\ddot{\mathsf{A}}}$), (AllocS'), and (AllocV) (figs. 2.4 and 2.11) with $\emptyset$:

1. In (Entry$_{\mathsf{C}}$) and (Entry$_{\mathsf{A}}$), $\texttt{addrSetsAreWF}(\Sigma^{hp}_P, \Sigma^{cl}_P, \Sigma^{cv}_P, \ldots, i^{g}_P, \ldots, \Sigma^{f}_P, \ldots, i^{y}_P, \ldots, \Sigma^{\texttt{vrdc}}_P)$ is replaced with $\texttt{addrSetsAreWF}(\Sigma^{hp}_P, \Sigma^{cl}_P, \ldots, i^{g}_P, \ldots, \Sigma^{f}_P, \ldots, i^{y}_P, \ldots, \Sigma^{\texttt{vrdc}}_P)$ for $P \in \{\mathsf{C}, \mathsf{A}\}$.

2. In (Op-esp'), $\texttt{intrvlInSet}(t, \texttt{esp} - 1_{\texttt{i32}}, \Sigma^{\{\texttt{free}\}}_{\ddot{\mathsf{A}}} \cup ((\Sigma^{cv}_{\ddot{\mathsf{A}}} \cup \Sigma^{Z_l|^v}_{\ddot{\mathsf{A}}}) \setminus \Sigma^{F}_{\ddot{\mathsf{A}}}))$ is replaced with $\texttt{intrvlInSet}(t, \texttt{esp} - 1_{\texttt{i32}}, \Sigma^{\texttt{free}}_{\ddot{\mathsf{A}}} \cup (\Sigma^{Z_l|^v}_{\ddot{\mathsf{A}}} \setminus \Sigma^{F}_{\ddot{\mathsf{A}}}))$.

3. In (Load$_{\ddot{\mathsf{A}}}$), $\texttt{ov}([p]_w, \Sigma^{\texttt{free}}_{\ddot{\mathsf{A}}} \cup ((\Sigma^{cv}_{\ddot{\mathsf{A}}} \cup (\Sigma^{Z_l|^v}_{\ddot{\mathsf{A}}})) \setminus \Sigma^{F \cup S}_{\ddot{\mathsf{A}}}))$ is replaced with $\texttt{ov}([p]_w, \Sigma^{\texttt{free}}_{\ddot{\mathsf{A}}} \cup ((\Sigma^{Z_l|^v}_{\ddot{\mathsf{A}}}) \setminus \Sigma^{F \cup S}_{\ddot{\mathsf{A}}}))$.

4. In (Store$_{\ddot{\mathsf{A}}}$), $\texttt{ov}([p]_w, \Sigma^{\{\texttt{free}\} \cup G_r \cup F_r}_{\ddot{\mathsf{A}}} \cup ((\Sigma^{cv}_{\ddot{\mathsf{A}}} \cup (\Sigma^{Z_l|^v}_{\ddot{\mathsf{A}}})) \setminus \Sigma^{F_w \cup S}_{\ddot{\mathsf{A}}}))$ is replaced with $\texttt{ov}([p]_w, \Sigma^{\{\texttt{free}\} \cup G_r \cup F_r}_{\ddot{\mathsf{A}}} \cup ((\Sigma^{Z_l|^v}_{\ddot{\mathsf{A}}}) \setminus \Sigma^{F_w \cup S}_{\ddot{\mathsf{A}}}))$.

5. In (AllocS'), $\texttt{ov}([v]_w, \Sigma^{cv}_{\ddot{\mathsf{A}}} \cup \Sigma^{Z_l|^v}_{\ddot{\mathsf{A}}})$ is replaced with $\texttt{ov}([v]_w, \Sigma^{Z_l|^v}_{\ddot{\mathsf{A}}})$.

6. In (AllocV), $\texttt{intrvlInSet}_a(v, v+w-1_{\texttt{i32}}, \texttt{comp}(\Sigma^{B \cup \{cv\}}_{\ddot{\mathsf{A}}}))$ is replaced with $\texttt{intrvlInSet}_a(v, v + w - 1_{\texttt{i32}}, \texttt{comp}(\Sigma^{B}_{\ddot{\mathsf{A}}}))$.

The callers' virtual smallest semantics are useful because they allow an over-approximation of the heap (hp) region in C′, which can be efficiently encoded in SMT to achieve faster discharge (as described in chapter 5).

3.4.1 Soundness of Callers' Virtual Smallest semantics

Let A and C be transition graphs obtained due to original semantics described in figs. 2.4 to 2.8, 2.10 and 2.11. Let A′ and C′ be obtained from A and C respectively by applying the callers' virtual smallest semantics described in previous section. Let Ä′ be obtained by annotating A′ as described in section 2.4. Let Ä be obtained by annotating A such that annotations made in Ä′ and Ä are identical. Let $\mathsf{X}' = Ä' \boxtimes C' = (\mathcal{N}_{\mathsf{X}'}, \mathcal{E}_{\mathsf{X}'}, \mathcal{D}_{\mathsf{X}'})$ be a product graph such that X′ satisfies the search-algorithm requirements. We prove that there exists a product graph $\mathsf{X} = Ä \boxtimes C = (\mathcal{N}_{\mathsf{X}}, \mathcal{E}_{\mathsf{X}}, \mathcal{D}_{\mathsf{X}})$ such that X satisfies the search-algorithm requirements. In other words, we show that existence of a product graph between procedures with callers' virtual smallest semantics implies existence of a product graph between the same procedures without callers' virtual smallest semantics.

Definition 3.4.1 ((CoverageC) holds for $\xi_{Ä}$ at n_{X} in X)**.** *At a node $n_{\mathsf{X}} \in \mathcal{N}_{\mathsf{X}}$, let $\{e^1_{\mathsf{X}}, e^2_{\mathsf{X}}, \ldots, e^m_{\mathsf{X}}\}$ be the set of* all *outgoing edges such that $e^j_{\mathsf{X}} = (n_{\mathsf{X}} \xrightarrow{\xi_{Ä}; \xi^j_{C}} (n^t_{Ä}, n^{t_j}_{C}))$ (for $1 \leq j \leq m$). Then,* (CoverageC) holds for $\xi_{Ä}$ at n_{X} in X *iff* $\{e^1_{\mathsf{X}}, e^2_{\mathsf{X}}, \ldots, e^m_{\mathsf{X}}\}\langle \mathcal{D}_{\mathsf{X}}, \xi_{Ä} \rangle$ *holds.*

Notice that this definition is identical to the (CoverageC) definition in section 3.3.1, except that it defines (CoverageC) for a specific path $\xi_{Ä}$ starting at a specific node n_{X}. We define (CoverageÄ) at node n_{X} analogously.

Theorem 3.4.2 (Soundness of Callers' Virtual Smallest Semantics)**.** *Given $\mathsf{X}' = Ä' \boxtimes C' = (\mathcal{N}_{\mathsf{X}'}, \mathcal{E}_{\mathsf{X}'}, \mathcal{D}_{\mathsf{X}'})$ that satisfies the search-algorithm requirements (section 3.3.1) , it is possible to construct $\mathsf{X} = Ä \boxtimes C = (\mathcal{N}_{\mathsf{X}}, \mathcal{E}_{\mathsf{X}}, \mathcal{D}_{\mathsf{X}})$ that also satisfies the search-algorithm requirements.*

Proof. Construct X = X′. Add extra edges in X to nodes $(\mathscr{W}_{Ä}, n_C)$ where n_C is an error-free node such that (MutexÄ) is not violated. These extra edges help in ensuring (CoverageÄ) in X.

As the use of callers' virtual smallest semantics does not affect the graph structure of A and C (recall that the changes were limited to modifications to instructions of an edge), the seven structural requirements, (MutexÄ), (MutexC), (Termination), (SingleIO), (Well-formedness), (Safety), and (Similar-speed), should continue to hold for X.

Let $n_{\mathsf{X}'} = (n_{Ä'}, n_{C'}) \in \mathcal{N}_{\mathsf{X}'}$ be a node in X′ and let $n_{\mathsf{X}} = (n_{Ä}, n_C) \in \mathcal{N}_{\mathsf{X}}$ be its corresponding node in X. Let $\xi_{Ä'}$ be an outgoing path at $n_{Ä'}$ in Ä′ and let $\xi_{Ä}$ be its structurally similar

path originating at $n_{\ddot{A}}$ in $\ddot{A}$. Let $\{e^1_{X'}, \ldots, e^m_{X'}\}$ be the set of all outgoing edges at $n_{X'}$ such that $\forall_{1 \leq j \leq m} : e^j_{X'} = (n_{X'} \xrightarrow{\xi_{\ddot{A}'}; \xi^j_{C'}} n^t_{X'}) \in \mathcal{E}_{X'}$. Let the set $\{e^1_{X'}, \ldots, e^m_{X'}\}$ be defined analogously for X. Our proof completes by induction on the number of edges executed in X, starting at n_X.

We analyze the instructions in $\ddot{A}$ and C affected by the semantics change and consider the case when an edge $e_{\ddot{A}} \in \xi_{\ddot{A}}$ or $e_C \in \xi^j_{C'}$ corresponds to it [7].

- (Entry$_C$) and (Entry$_{\ddot{A}}$): The ¬`addrSetsAreWF`(...) condition is weaker in $\ddot{A}$ and C than $\ddot{A}'$ and C' respectively. Consequently, the path condition for paths $\xi_{\ddot{A}} = (n_{\ddot{A}} \twoheadrightarrow n^{\not\mathscr{W}}_{\ddot{A}})$ (where $n^{\not\mathscr{W}}_{\ddot{A}} \in \mathcal{N}_{\ddot{A}} \setminus \mathscr{W}_{\ddot{A}}$) and $\xi_C = (n_C \twoheadrightarrow n^{\not\mathscr{W}}_C)$ (where $n^{\not\mathscr{W}}_C \in \mathcal{N}_C \setminus \mathscr{W}_C$) that do not go to $\mathscr{W}_{\ddot{A}}$ and $\mathscr{W}_C$ respectively is stronger in $\ddot{A}$ and C than $\ddot{A}'$ and C' respectively.

  Because the address sets returned by the `rd` instruction are arbitrary and identical across C and $\ddot{A}$, due to (Equivalence), (Coverage$C$) holds by construction in this case.

  As the results of the `rd` instruction are arbitrary, the difference in infeasibility of $\xi_{\ddot{A}'} = (n_{\ddot{A}'} \twoheadrightarrow \mathscr{W}_{\ddot{A}'})$ and structurally similar $\xi_{\ddot{A}} = (n_{\ddot{A}} \twoheadrightarrow \mathscr{W}_{\ddot{A}})$ can only be due to the address set of regions in F (see definition of `addrSetsAreWF`(...) in table 2.1) As $\Sigma^F_{\ddot{A}'} = \Sigma^F_{\ddot{A}}$, (Coverage$\ddot{A}$) at n_X should continue to hold in this case.

- (Alloc), (AllocV), and (AllocS'): As $(\Sigma^{cv}_{\ddot{A}} = \Sigma^{cv}_C) \supseteq (\Sigma^{cv}_{\ddot{A}'} = \Sigma^{cv}_{C'} = \emptyset)$, the ¬`intrvlInSet`$_a$(...) condition of (Alloc) and (AllocV) and `ov`(...) condition of (AllocS') is weaker in $\ddot{A}$ and C than $\ddot{A}'$ and C' respectively. Consequently, similarly to previous case, the path condition for paths that do not go to $\mathscr{W}_{\ddot{A}}$ and $\mathscr{W}_C$ respectively is stronger in $\ddot{A}$ and C than $\ddot{A}'$ and C' respectively.

  Due to (SingleIO), the nodes $n_{\ddot{A}}$ and n_C must either correspond to PCs due to: (1) (AllocV) and (Alloc); or (2) (AllocS') and (Alloc). Due to (Equivalence), $\Sigma^{\texttt{comp}(B \cup \{cv\})}_{\ddot{A}} = \Sigma^{\texttt{comp}(B \cup \{cv\})}_C = \Sigma^{\texttt{free}}_C$ must hold at n_X. As, for $P \in \{\ddot{A}, \ddot{A}', C, C'\}$, $\Sigma^{\{hp,cl\}}_P$ is assigned arbitarily at entry, the set of possible values for $\Sigma^{\texttt{comp}(B \cup \{cv\})}_P$ (note $\Sigma^{cv}_{\ddot{A}'} = \Sigma^{cv}_{C'} = \emptyset$) remain identical in P at an error-free node n_X and $n_{X'}$. Thus, in case (1), the affected ¬`intrvlInSet`$_a$(...) condition should have identical semantics in both X' and X and (Coverage$\ddot{A}$) and (Coverage$C$) should continue to hold.

  In case (2), a path $\xi_{\ddot{A}'} = (n_{\ddot{A}'} \twoheadrightarrow \mathscr{W}_{\ddot{A}'})$ with an edge with the `ov`(...) condition could be provably infeasible at $n_{X'}$ in X' but a similarly structured path $\xi_{\ddot{A}}$ could

[7] Note that (Load$_C$), (Store$_C$), (CallV), and (Call$_C$), are not affected as the cv region is inaccessible in C and cannot be returned by $\beta(x)$ for any variable x and $\beta_M(r)$ for any region r.

potentially be feasible at n_{X} in X — e.g., when $\Sigma^{Z_l}_{\ddot{\mathsf{A}}'}|^{v} = \emptyset$. To ensure (Coverage$\ddot{\mathsf{A}}$), we introduce edge $e'_{\mathsf{X}} = ((n_{\ddot{\mathsf{A}}}, n_{\mathsf{C}}) \xrightarrow{\xi_{\ddot{\mathsf{A}}};\epsilon} (\mathscr{W}_{\ddot{\mathsf{A}}}, n_{\mathsf{C}}))$ for each such path $\xi_{\ddot{\mathsf{A}}}$ in X. Notice that (Coverage$\mathsf{C}$) holds for $\xi_{\ddot{\mathsf{A}}}$ at n_{X}. Because $\xi_{\ddot{\mathsf{A}}}$ does not contain any memory access, introduction of e'_{X} would not disturb (MAC).

For a path $(n_{\ddot{\mathsf{A}}} \twoheadrightarrow n^{\mathscr{W}}_{\ddot{\mathsf{A}}})$ (where $n^{\mathscr{W}}_{\ddot{\mathsf{A}}} \in \mathcal{N}_{\ddot{\mathsf{A}}} \setminus \{\mathscr{W}_{\ddot{\mathsf{A}}}\}$), (Coverage$\mathsf{C}$) holds due to (Stack subset) invariant (theorem 3.3.11) and by using identical reasoning as case (1) above.

- (Op-esp'): The condition `intrvlInSet()` is not affected by the semantics change as the address sets $\Sigma^{\mathtt{free}}_{\ddot{\mathsf{A}}} \cup ((\Sigma^{cv}_{\ddot{\mathsf{A}}} \cup \Sigma^{Z_l}_{\ddot{\mathsf{A}}}|^{v}) \setminus \Sigma^{F}_{\ddot{\mathsf{A}}})$ and $\Sigma^{\mathtt{free}}_{\ddot{\mathsf{A}}'} \cup (\Sigma^{Z_l}_{\ddot{\mathsf{A}}'}|^{v} \setminus \Sigma^{F}_{\ddot{\mathsf{A}}'})$ must evaluate to identical values (on states σ and σ' at nodes n_{X} and $n_{\mathsf{X}'}$ in X and X' resp. such that $\phi_{n_{\mathsf{X}}}(\sigma)$ and $\phi_{n_{\mathsf{X}'}}(\sigma')$ hold) due to new definition of $\Sigma^{\mathtt{free}}_{\ddot{\mathsf{A}}'}$ in $\ddot{\mathsf{A}}'$.

- (Load$_{\ddot{\mathsf{A}}}$) and (Store$_{\ddot{\mathsf{A}}}$): Identical reasoning as (Op-esp') case; the address set expressions should evaluate to identical values. Hence, no change in semantics for this case too.

As the path condition to an error-free node is only stronger (or equivalent) in $\ddot{\mathsf{A}}$ and C, the remaining semantic requirements, (Inductive), (Equivalence), (MAC), and (MemEq) should also continue to hold in X.

□

3.5 Safety-Relaxed Semantics

We define new *safety-relaxed semantics* for the assembly procedure A with callers' virtual smallest semantics. These semantics relax the memory-safety checks in A; the soundness is retained in the context of product graph because of the (MAC) requirement.

Under the safety-relaxed semantics, we construct A' from A such that

(a) a $\varphi_l = \mathtt{ov}([p]_w, \Sigma^{\mathtt{free}}_{\ddot{\mathsf{A}}} \cup ((\Sigma^{Z_l}_{\ddot{\mathsf{A}}}|^{v}) \setminus \Sigma^{F \cup S}_{\ddot{\mathsf{A}}}))$ check due to (Load$_{\ddot{\mathsf{A}}}$) in A is replaced with $\varphi'_l = \mathtt{ov}([p]_w, (\Sigma^{Z_l}_{\ddot{\mathsf{A}}}|^{v}) \setminus (\Sigma^{F}_{\ddot{\mathsf{A}}} \cup [\mathtt{esp}, \boxed{\mathtt{cs}_e}]))$ in A'.

(b) a $\varphi_s = \mathtt{ov}([p]_w, \Sigma^{\{\mathtt{free}\} \cup G_r \cup F_r}_{\ddot{\mathsf{A}}} \cup ((\Sigma^{Z_l}_{\ddot{\mathsf{A}}}|^{v}) \setminus \Sigma^{F_w \cup S}_{\ddot{\mathsf{A}}}))$ check due to (Store$_{\ddot{\mathsf{A}}}$) in A is replaced with $\varphi'_s = \mathtt{ov}([p]_w, (\Sigma^{Z_l}_{\ddot{\mathsf{A}}}|^{v}) \setminus (\Sigma^{F_w}_{\ddot{\mathsf{A}}} \cup [\mathtt{esp}, \boxed{\mathtt{cs}_e}]))$ in A'.

(c) a $\varphi_r = \neg(\boxed{M^{cs}} =_{\Sigma^{cs}_{\ddot{\mathsf{A}}}} M_{\mathsf{A}})$ check due to (Ret$_{\mathsf{A}}$) in A is replaced with $\varphi'_r = \mathtt{false}$ in A'.

We call this construction a *safety-relaxed rewrite* and call A' the *safety-relaxed version* of A.

The callers' virtual smallest and safety-relaxed semantics are useful in enabling an efficient SMT encoding where the *hp* region in C' and A' is over-approximated and the *cs* region in A' is under-approximated (as described in chapter 5).

3.5.1 Soundness of Safety-Relaxed Semantics

Let $\ddot{\mathsf{A}}'$ be obtained by annotating A' as described in section 2.6. Let $\ddot{\mathsf{A}}$ be the annotated version of A such that the annotations made in $\ddot{\mathsf{A}}$ and $\ddot{\mathsf{A}}'$ are identical. Let C be the corresponding unoptimized IR procedure with the callers' virtual smallest semantics. Let $\mathsf{X}' = \ddot{\mathsf{A}}' \boxtimes \mathsf{C} = (\mathcal{N}_{\mathsf{X}'}, \mathcal{E}_{\mathsf{X}'}, \mathcal{D}_{\mathsf{X}'})$ be a product graph that satisfies the search-algorithm requirements. We prove that it is possible to construct $\mathsf{X} = \ddot{\mathsf{A}} \boxtimes \mathsf{C} = (\mathcal{N}_{\mathsf{X}}, \mathcal{E}_{\mathsf{X}}, \mathcal{D}_{\mathsf{X}})$ such that X also satisfies search-algorithm requirements (theorem 3.5.3). In other words, we prove that the existence of a product graph between procedures with safety-relaxed semantics implies the existence of a product graph between procedures without safety-relaxed semantics.

Lemma 3.5.1 (Paths containing memory accesses do not modify allocation state of common regions)**.** *Let* $e_{\mathsf{X}} = (n_{\mathsf{X}} \xrightarrow{\xi_{\ddot{\mathsf{A}}}; \xi_{\mathsf{C}}} n'_{\mathsf{X}}) \in \mathcal{E}_{\mathsf{X}}$. *If* $\xi_{\ddot{\mathsf{A}}}$ *contains an edge corresponding to* ($\textsc{Load}_{\ddot{\mathsf{A}}}$) *or* ($\textsc{Store}_{\ddot{\mathsf{A}}}$) *(i.e., a* `load` *or* `store` *instruction), then* $\xi_{\ddot{\mathsf{A}}}$ *does not modify the address sets corresponding to regions in* B*:* $\Sigma^{g}_{\ddot{\mathsf{A}}}$ *(for each* $g \in G$*),* $\Sigma^{hp}_{\ddot{\mathsf{A}}}$, $\Sigma^{cl}_{\ddot{\mathsf{A}}}$, $\Sigma^{y}_{\ddot{\mathsf{A}}}$ *(for each* $y \in Y$*), and* $\Sigma^{z}_{\ddot{\mathsf{A}}}$ *(for each* $z \in Z$*).*

Proof. Once initialized in ($\textsc{Entry}_{\mathsf{A}}$) in an I/O path that does not contain any `load` or `store` instruction (fig. 2.7), the address sets corresponding to regions $B \setminus Z$ are not modified during the entire execution of $\ddot{\mathsf{A}}$.

The address set corresponding to a region $z \in Z$ may only be modified by the `(de)alloc`$_{\mathtt{s,v}}$ instructions. Due to (SingleIO) requirement, these `(de)alloc`$_{\mathtt{s,v}}$ instructions cannot exist as a part of longer paths that may contain `load` or `store` instructions (as evident from translations given in figs. 2.8, 2.10 and 2.11). □

Note the the lemma holds for both original and safety-relaxed version of $\ddot{\mathsf{A}}$. As a corollary, due to (SingleIO), ξ_{C} also does not modify the address sets corresponding to regions in B.

Lemma 3.5.2 ($\pi_{\Sigma^{cs}_{Ä'}}(M_{Ä'})$ is not modified in X′)**.** *Let* $X' = Ä' \boxtimes C$ *be a product graph for a lockstep execution between* Ä′ *(with safety-relaxed semantics) and* C*. If* X′ *satisfies the search-algorithm requirements, then* $\boxed{M^{cs}} =_{\Sigma^{cs}_{Ä'}} M_{Ä'}$ *holds at each error-free, non-entry node* $n_{X'} \in \mathcal{N}^{\overline{UW}}_{X'}$.

Proof. For simplicity, let's first assume that there is only one outgoing edge $e^s_{X'} = (n^s_{X'} \xrightarrow{\xi^s_{Ä'};\xi^s_C} n^{s2}_{X'})$ from the start node $n^s_{X'}$ to an error-free node $n^{s2}_{X'}$ such that $\xi^s_{Ä'}$ and ξ^s_C represent the program paths corresponding to (Entry$_{Ä}$) and (Entry$_C$) respectively. Let's call this the *start-edge* assumption.

The proof proceeds by induction over the number of edges executed in X′ starting from $n^{s2}_{X'}$.

$\boxed{M^{cs}} =_{\Sigma^{cs}_{Ä'}} M_{Ä'}$ holds at $n^{s2}_{X'}$ due to (Entry$_{Ä}$), which forms our base case.

Consider a node $n_{X'}$ such that $\boxed{M^{cs}} =_{\Sigma^{cs}_{Ä'}} M_{Ä'}$ holds at $n_{X'}$, and let $e_{X'} = (n_{X'} \xrightarrow{\xi_{Ä'};\xi_C} n^t_{X'}) \in \mathcal{E}_{X'}$ such that $n^t_{X'} = (n^t_{Ä'}, n^t_C) \in \mathcal{N}^{\overline{UW}}_{X'}$ is an error-free node. Two cases for $\xi_{Ä'}$:

1. If $\xi_{Ä'}$ does not contain a `store` instruction, then $\boxed{M^{cs}} =_{\Sigma^{cs}_{Ä'}} M_{Ä'}$ holds trivially at $n^t_{X'}$.

2. If $\xi_{Ä'}$ contains a `store` instruction, then it cannot modify the allocation state of common regions (B) in Ä′ (due to lemma 3.5.1). Similarly, ξ_C also cannot modify the allocation state of common memory regions in C (corollary of lemma 3.5.1).

   Let α be an address such that a `store` is performed to α in $\xi_{Ä'}$. If $\alpha \in \Sigma^{cs}_{Ä'}$, then due to (MAC), there must be a `store` to the same address in C before execution may reach n^t_C. Then, due to the global invariants (Disjoint regions in Ä) and (cs and cl) (section 3.3.3) and requirement (Equivalence), $\Sigma^{cs}_{Ä'} \subseteq (\Sigma^{Z_l}_{Ä'}|^v \cup \Sigma^{\texttt{free}}_C) \cap [\boxed{\texttt{stk}_e} + 1, \boxed{\texttt{cs}_e}]$ must hold during the execution of $e_{X'}$. So, $\alpha \in (\Sigma^{Z_l}_{Ä'}|^v \cup \Sigma^{\texttt{free}}_C) \cap [\boxed{\texttt{stk}_e} + 1, \boxed{\texttt{cs}_e}]$. However, $\alpha \in (\Sigma^{Z_l}_{Ä'}|^v) \setminus (\Sigma^{F_w}_{Ä'} \cup [\texttt{esp}, \boxed{\texttt{cs}_e}])$ is not possible for an error-free node going $\xi_{Ä'}$ due to (Store$_{Ä}$) with the safety-relaxed semantics. Thus, $\alpha \in (\Sigma^{\texttt{free}}_C \cap [\boxed{\texttt{stk}_e} + 1, \boxed{\texttt{cs}_e}])$ must hold. However, this is not possible for an error-free node going ξ_C due to (Store$_C$). Thus, by contradiction, a `store` to address $\alpha \in \Sigma^{cs}_{Ä'}$ is infeasible in $\xi_{Ä'}$. Thus, $\boxed{M^{cs}} =_{\Sigma^{cs}_{Ä'}} M_{Ä'}$ holds at $n^t_{X'}$.

To generalize beyond the start-edge assumption, we only need to show that for an outgoing edge $e_{X'} = (n_{X'} \xrightarrow{\xi_{Ä'};\xi_C} n^t_{X'}) \in \mathcal{E}_{X'}$ such that $n_{X'}$ is not a non-entry node but

$n^t_{\mathsf{X}'} = (n^t_{\ddot{\mathsf{A}}'}, n^t_{\mathsf{C}})$ is a non-entry node, $M^{cs} =_{\Sigma^{cs}_{\ddot{\mathsf{A}}'}} M_{\ddot{\mathsf{A}}'}$ holds at $n^t_{\mathsf{X}'}$. We observe that there must exist a node $n'_{\ddot{\mathsf{A}}'}$ in $\xi_{\ddot{\mathsf{A}}'}$ where $M^{cs} =_{\Sigma^{cs}_{\ddot{\mathsf{A}}'}} M_{\ddot{\mathsf{A}}'}$ holds due to ($\textsc{Entry}_{\ddot{\mathsf{A}}}$). The rest of the argument remains identical for the path $\xi'_{\ddot{\mathsf{A}}'} = (n'_{\ddot{\mathsf{A}}'} \twoheadrightarrow n^t_{\ddot{\mathsf{A}}'})$. □

Theorem 3.5.3 (Soundness of Safety-Relaxed Semantics)**.** *Given* $\mathsf{X}' = \ddot{\mathsf{A}}' \boxtimes \mathsf{C}$ *that satisfies the search-algorithm requirements, it is possible to construct* $\mathsf{X} = \ddot{\mathsf{A}} \boxtimes \mathsf{C}$ *that also satisfies the search-algorithm requirements.*

Proof. Construct $\mathsf{X} = \mathsf{X}'$ with some extra edges from nodes in X to the error-node $(\mathcal{U}_{\ddot{\mathsf{A}}}, \mathcal{U}_{\mathsf{C}})$ such that (Mutex$\ddot{\mathsf{A}}$), (Mutex$\mathsf{C}$) and (SingleIO) are not violated. We later describe what edges are added to X and why X continues to satisfy the search-algorithm requirements even after the addition of these edges. It is already possible to see that the structural requirements viz., (Termination), (Similar-speed), (Well-formedness), and (Safety), will hold for X even after the addition of such edges.

Let $\xi_{\ddot{\mathsf{A}}}$ be a path in $\ddot{\mathsf{A}}$ on which there exists an overlap check $\varphi_l = \mathtt{ov}([p]_w, \Sigma^{\mathtt{free}}_{\ddot{\mathsf{A}}} \cup ((\Sigma^{Z_l}_{\ddot{\mathsf{A}}}|^v) \setminus \Sigma^{F \cup S}_{\ddot{\mathsf{A}}}))$ (for triggering $\mathcal{U}$) due to a ($\textsc{Load}_{\ddot{\mathsf{A}}}$) instruction [8], In $\ddot{\mathsf{A}}'$, φ_l is replaced by $\varphi'_l = \mathtt{ov}([p]_w, (\Sigma^{Z_l}_{\ddot{\mathsf{A}}}|^v) \setminus (\Sigma^{F}_{\ddot{\mathsf{A}}} \cup [\mathtt{esp}, \boxed{\mathtt{cs}_e}]))$ [9] to obtain $\xi_{\ddot{\mathsf{A}}'}$.

Recall that A's translation for ($\textsc{Load}_{\ddot{\mathsf{A}}}$) has "$\mathtt{if}\ \varphi_l\ \mathtt{then\ halt}(\mathcal{U})$" [10] while A''s translation has "$\mathtt{if}\ \varphi'_l\ \mathtt{then\ halt}(\mathcal{U})$" [11]. Because $\varphi'_l \Rightarrow \varphi_l$, $\ddot{\mathsf{A}}$ may trigger $\mathcal{U}$ when $\ddot{\mathsf{A}}'$ would simply execute the *error-free path* (the path that does not end at an error node) in ($\textsc{Load}_{\ddot{\mathsf{A}}}$). Conversely, if $\ddot{\mathsf{A}}$ executes an error-free path (of ($\textsc{Load}_{\ddot{\mathsf{A}}}$)) on an initial state σ, then $\ddot{\mathsf{A}}'$ will also execute the same error-free path on σ [12].

Similarly, let $\varphi_r = \neg(M^{cs} =_{\Sigma^{cs}_{\ddot{\mathsf{A}}}} M_{\ddot{\mathsf{A}}})$ be a check in $\ddot{\mathsf{A}}$ (due to ($\textsc{Ret}_{\mathsf{A}}$)), that has been replaced with $\varphi'_r = \mathtt{false}$ in $\ddot{\mathsf{A}}'$. Again, if $\ddot{\mathsf{A}}$ executes an error-free path of ($\textsc{Ret}_{\mathsf{A}}$) on an initial state σ, then $\ddot{\mathsf{A}}'$ will also execute the same error-free path on σ.

Thus, it can be shown through induction that four of the six semantic requirements — (Inductive), (Equivalence), (MAC), (MemEq) — hold on X if they hold on X' with $\mathbf{\Phi}_{\mathsf{X}} = \mathbf{\Phi}_{\mathsf{X}'}$. The common argument in this part of the proof is that the path condition of an error-free path in X (containing $\neg\varphi_{l,s,r}$ for ($\textsc{Load}_{\ddot{\mathsf{A}}}$), ($\textsc{Store}_{\ddot{\mathsf{A}}}$), and ($\textsc{Ret}_{\mathsf{A}}$)) is always stronger than the path condition of an error-free path in X' (containing $\neg\varphi'_{l,s,r}$).

[8] Or, an overlap check $\varphi_s = \mathtt{ov}([p]_w, \Sigma^{\mathtt{free}}_{\ddot{\mathsf{A}}} \cup ((\Sigma^{Z_l}_{\ddot{\mathsf{A}}}|^v) \setminus \Sigma^{F_w \cup S}_{\ddot{\mathsf{A}}}))$ (for triggering $\mathcal{U}$) due to a ($\textsc{Store}_{\ddot{\mathsf{A}}}$) instruction.

[9] $\varphi'_s = \mathtt{ov}([p]_w, (\Sigma^{Z_l}_{\ddot{\mathsf{A}}}|^v) \setminus (\Sigma^{F_w}_{\ddot{\mathsf{A}}} \cup [\mathtt{esp}, \boxed{\mathtt{cs}_e}]))$ in case of a ($\textsc{Store}_{\ddot{\mathsf{A}}}$).

[10] "$\mathtt{if}\ \varphi_s\ \mathtt{then\ halt}(\mathcal{U})$" for ($\textsc{Store}_{\ddot{\mathsf{A}}}$)

[11] "$\mathtt{if}\ \varphi'_s\ \mathtt{then\ halt}(\mathcal{U})$" for ($\textsc{Store}_{\ddot{\mathsf{A}}}$)

[12] $\varphi'_s \Rightarrow \varphi_s$ in case of ($\textsc{Store}_{\ddot{\mathsf{A}}}$)) — rest of the argument remains identical.

We next show that if (CoverageC) holds for path $\xi_{\ddot{A}'}$ starting at node $n_{X'}$ in X', (CoverageC) also holds for corresponding path $\xi_{\ddot{A}}$ starting at corresponding node n_X in X (note: using definition 3.4.1 here). For an edge $e^j_X = (n_X \xrightarrow{\xi_{\ddot{A}};\xi^j_C} (n^t_{\ddot{A}}, n^{t_j}_C)) \in \mathcal{E}_X$ $(1 \leq j \leq m)$, if $\xi_{\ddot{A}}$ ends at a node $n^t_{\ddot{A}} \neq \mathcal{U}_{\ddot{A}}$, then this is easy to show by induction on the number of edges executed on a path: because the path condition of $\xi_{\ddot{A}}$ in $\ddot{A}$ is always equal or stronger than the path condition of a corresponding (structurally identical) path $\xi_{\ddot{A}'}$ in $\ddot{A}'$. If (CoverageC) holds for $\xi_{\ddot{A}'}$ at a node n'_X in X', it must also hold for $\xi_{\ddot{A}}$ at the corresponding node n_X in X. We next show that (CoverageC) holds for a path $\xi_{\ddot{A}}$ terminating in $\mathcal{U}_{\ddot{A}}$ $(n^t_{\ddot{A}} = \mathcal{U}_{\ddot{A}})$.

Consider a path $\xi_{\ddot{A}}$ in $\ddot{A}$ and the corresponding path $\xi_{\ddot{A}'}$ in $\ddot{A}'$. If on a machine state σ both paths $\xi_{\ddot{A}}$ and $\xi_{\ddot{A}'}$ transition to $\mathcal{U}_{\ddot{A}}$ and $\mathcal{U}_{\ddot{A}'}$ respectively, then because X' satisfies (CoverageC), σ must execute one of ξ^j_C (for $1 \leq j \leq m$) to completion, thus satisfying (CoverageC) in X in this case. Thus, we only need to cater to the following two situations where execution on $\ddot{A}$ may deviate from $\ddot{A}'$ (i.e., execution does not complete for $\ddot{A}$ but completes for $\ddot{A}'$)

- (Ret$_A$): Let $\varphi_r = \neg(\boxed{M^{cs}} =_{\Sigma^{cs}_{\ddot{A}}} M_{\ddot{A}})$ be the check in $\ddot{A}$ (due to (Ret$_A$)), that has been replaced with $\varphi'_r =$ `false` in $\ddot{A}'$. We show that φ_r must evaluate to `false` in X at procedure return. In other words, the $\ddot{A}$ path "`if` φ_r `then halt`$(\mathcal{U})$" is infeasible and so $\ddot{A}$ does not deviate from $\ddot{A}'$ in this case.

  By lemma 3.5.2, $\boxed{M^{cs}} =_{\Sigma^{cs}_{\ddot{A}'}} M_{\ddot{A}'}$ holds at every error-free node $n_{X'} \in \mathcal{N}^{\mathcal{UW}}_{X'}$ and therefore $n_X \in \mathcal{N}_X$. Further, using the (MAC) requirement at the error-free terminating node `exit`, this can be generalized to show that $\boxed{M^{cs}} =_{\Sigma^{cs}_{\ddot{A}}} M_{\ddot{A}}$ holds at the beginning of the path corresponding to (Ret$_A$) in $\ddot{A}$. Thus, because the $\ddot{A}$ path "`if` φ_r `then halt`$(\mathcal{U})$" is infeasible, (CoverageC) holds trivially for this path at n_X in X.

- (Load$_{\ddot{A}}$) or (Store$_{\ddot{A}}$): Let $\xi^U_{\ddot{A}} = (n_{\ddot{A}} \twoheadrightarrow \mathcal{U}_{\ddot{A}})$ be a path that terminates with $\mathcal{U}_{\ddot{A}}$.

  Lemma 3.5.4. *Let σ be a state at an error-free node $n_X = (n_{\ddot{A}}, n_C) \in \mathcal{N}^{\mathcal{UW}}_X$ such that $\phi_{n_X}(\sigma)$ holds and σ executes $\xi^U_{\ddot{A}} = (n_{\ddot{A}} \twoheadrightarrow \mathcal{U}_{\ddot{A}})$ to completion. Then σ must execute some path $\xi_C = (n_C \twoheadrightarrow \mathcal{U}_C)$ to completion in C.*

  Proof. Consider the execution of σ on X' starting at $n_{X'} = (n_{\ddot{A}'}, n_C)$, such that $n_{X'}$ in X' is structurally identical to n_X in X. Due to (MutexÄ) and (CoverageÄ), there can be only two cases:

1. σ executes some path $\xi^x_{\ddot{A}'} = (n_{\ddot{A}'} \twoheadrightarrow \mathcal{U}_{\ddot{A}})$ to completion in $\ddot{A}'$. In this case, due to (Coverage$C$) and (Safety), some $\xi^x_C = (n_C \twoheadrightarrow \mathcal{U}_C)$ must be executed to completion on σ in C. In this case, the lemma holds with $\xi_C = \xi^x_C$.

2. σ executes some path $\xi^x_{\ddot{A}'} = (n_{\ddot{A}'} \twoheadrightarrow n^x_{\ddot{A}'})$ to completion in $\ddot{A}'$, where $n^x_{\ddot{A}'} \neq \mathcal{U}_{\ddot{A}'}$ and $e^{x_v}_{X'} = (n_{X'} \xrightarrow{\xi^x_{\ddot{A}'};\xi^{x_v}_C} n^{x_v}_{X'}) \in \mathcal{E}_{X'}$ (for $1 \leq v \leq w$) are $w \geq 1$ edges in X', where $n^{x_v}_{X'} = (n^x_{\ddot{A}'}, n^{x_v}_C)$. Because X' satisfies (Coverage$C$), σ must execute a path $\xi^{x_v}_C = (n_C \twoheadrightarrow n^{x_v}_C)$ to completion in C, for some $1 \leq v \leq w$. We show by contradiction that $\forall_{1 \leq v \leq w} : n^{x_v}_C = \mathcal{U}_C$ must hold.

   Assume $n^{x_v}_C \neq \mathcal{U}_C$. Let memory access instructions $d_1, d_2, \ldots, d_k$ exist on path $\xi^x_{\ddot{A}'}$, such that $\xi^x_{\ddot{A}'}$ deviates from $\xi^U_{\ddot{A}}$ on one of these memory access instructions d_r $(1 \leq r \leq k)$, so that $\xi^U_{\ddot{A}}$ transitions to $\mathcal{U}_{\ddot{A}}$ due to φ evaluating to `true` in a check "`if` φ `halt`($\mathcal{U}$)" in a (Load$_{\ddot{A}}$) or (Store$_{\ddot{A}}$) in $\ddot{A}$, while $\xi^x_{\ddot{A}'}$ continues execution to reach $n^x_{\ddot{A}'} \neq \mathcal{U}_{\ddot{A}'}$ due to φ' (safety-relaxed rewrite of φ) evaluating to `false` in a corresponding check "`if` φ' `halt`($\mathcal{U}$)" in $\ddot{A}'$.

   Let $[p]_w$ represent the addresses being accessed by the memory access instruction d_r. It must be true that $\exists \alpha \in [p]_w : \alpha \in \texttt{comp}(\Sigma^{B \cup F \cup S}_{\ddot{A}'})$ if d_r is a `load` instruction and $\exists \alpha \in [p]_w : \alpha \in \texttt{comp}(\Sigma^{(B \setminus G_r) \cup F_w \cup S}_{\ddot{A}'})$ if d_r is a `store` instruction; this is because φ' evaluates to `false` but φ evaluates to `true` (for `load` and `store` instructions). Because X' satisfies (MAC), the execution of σ starting at n_C must cause all addresses in $[p]_w$ to be accessed before execution can reach $n^{x_v}_C$ in C (and $n^{x_v}_{X'}$ in X'); this is because $\alpha \notin \Sigma^{G \cup F}_{\ddot{A}'} \cup [\texttt{esp}, \texttt{stk}_e]$. Further, because $\xi^x_{\ddot{A}'}$ contains a memory access instruction, due to lemma 3.5.1, both $\xi^x_{\ddot{A}'}$ and ξ^x_C cannot modify the address sets of common regions B. Thus, during the execution of σ starting at n_C, the `accessIsSafeC`$_{\tau,a}()$ check must necessarily evaluate to `false` and the execution must transition to $\mathcal{U}_C$. This is a contradiction, and so it must be true that $n^{x_v}_C = \mathcal{U}_C$. Hence, the lemma holds in this case with $\xi_C = \xi^x_C = (n_C \twoheadrightarrow \mathcal{U}_C)$.

□

Using lemma 3.5.4, we enumerate all such paths $\xi_C = (n_C \twoheadrightarrow \mathcal{U}_C)$ that can be executed in C if $\xi^U_{\ddot{A}} = (n_{\ddot{A}} \twoheadrightarrow \mathcal{U}_{\ddot{A}})$ is executed in $\ddot{A}$ starting at node $n_X \in \mathcal{N}_X$. As described in the proof of lemma 3.5.4, there are only a finite number of such paths. For each such path ξ_C, we add an edge $e^x_X = (n_X \xrightarrow{\xi^U_{\ddot{A}};\xi_C} (\mathcal{U}_{\ddot{A}}, \mathcal{U}_C))$ to $\mathcal{E}_X$ if it does not exist already. (Coverage$C$) thus follows from lemma 3.5.4. Further, (Coverage$\ddot{A}$) also holds for X because all assembly paths that exist in X' also exist in X and additional paths, only potentially feasible in $\ddot{A}$, are added.

□

Using theorems 3.4.2 and 3.5.3, hereafter, we will use only the safety-relaxed and callers' virtual smallest semantics of the unoptimized IR and assembly procedures. We will continue to refer to the unoptimized IR with the callers' virtual smallest semantics and assembly procedure with the safety-relaxed and callers' virtual smallest semantics as C and A respectively. The corresponding annotated procedure of A will be referred as $\ddot{\mathsf{A}}$.

Chapter 4

Automatic Construction of a Product-Program

In the previous chapter, we established that a product program $\mathsf{X} = \ddot{\mathsf{A}} \boxtimes \mathsf{C}$ can be used as a witness of refinement from an unoptimized IR procedure C to an annotated assembly procedure $\ddot{\mathsf{A}}$. In this chapter, we describe our algorithm, called Dynamo, for simultaneous automatic annotation of A, to produce $\ddot{\mathsf{A}}$, and construction of X. The product program X produced by our algorithm is guaranteed to satisfy the thirteen requirements that enable it to be used as a witness of refinement between an input unoptimized IR procedure C and the annotated assembly procedure $\ddot{\mathsf{A}}$, also produced by our algorithm from the input assembly procedure A.

Dynamo uses a set of internal heuristics for discovering the required annotation for A — we call this *blackbox* setting. In the other *whitebox* setting, Dynamo is capable of utilizing external *untrusted* hints for annotating A. These hints may be *untrusted* because the annotation is validated by the algorithm. As the hints need not be trusted, they could be generated through lightweight compiler instrumentation which may be incorrect or sourced from potentially inaccurate sources such as debug headers.

4.1 The Dynamo algorithm

The Dynamo algorithm takes the transition graphs corresponding to the $\texttt{LLVM}_d$ and assembly procedures (C and A resp.) and an *unroll factor* μ as input and returns, if successful, an annotated $\ddot{\mathsf{A}}$ and a product graph $\mathsf{X} = \ddot{\mathsf{A}} \boxtimes \mathsf{C} = (\mathcal{N}_\mathsf{X}, \mathcal{E}_\mathsf{X}, \mathcal{D}_\mathsf{X})$ as output. In

addition, it also identifies an inductive invariant network Φ_X that maps each error-free node $n_X \in \mathcal{N}_X^{\mathcal{UW}}$ to its node invariant ϕ_{n_X}.

Given enough computational time, DYNAMO is guaranteed to find the required $(\ddot{A}, X)$ if:

(a) A is a translation of C through, potentially path-specializing, bisimilar transformations up to a maximum unrolling of μ.

(b) For two or more allocations or procedure calls that reuse stack space in A, their relative order in C is preserved in A.

(c) An allocation due to `alloca()` is always stack-allocated in A.

(d) An allocation-containing path is not specialized by the compiler, such that on one specialization stack-allocation is performed and on another register-allocation is performed.

(e) The desired annotation to $\ddot{A}$ is identifiable either through search heuristics (blackbox mode) or through user-supplied and/or compiler hints (whitebox).

(f) Our invariant inference procedure is able to identify the required invariant network Φ_X that captures the compiler transformations from C to A.

The restriction on non-bisimilar and de-specializing transformations stated in clause (a) stems from the limitations of the COUNTER algorithm [17] of which our algorithm is a derivative of — the former is fundamental to the algorithm. The restriction on de-specializing transformations is a performance trade-off and can potentially be lifted at the cost of runtime (see §4.4 of [17]). Clause (b) is simply a restatement of the limitation of our refinement definition (section 2.5.4), and clauses (c) and (d) are due to restrictions on a virtual-allocation annotation (section 2.6). We describe our blackbox annotation algorithm that dictates clause (e) in section 4.1.3 and our invariant inference algorithm (for clause (f)) in section 4.2.

DYNAMO constructs the solution incrementally, by relying on the property that for a non-coverage requirement to hold for a fully-annotated $\ddot{A}$ and a fully-constructed X, it must also hold for a partially-annotated $\ddot{A}$ and a partially-constructed subgraph of X rooted at its entry node n_X^s. A partially-constructed X (based on a partially-annotated $\ddot{A}$), constructed from the entry node n_X^s, that does not meet the non-coverage requirements may be safely discarded without affecting the completeness of the algorithm.

Recall that we defined callers' virtual smallest and safety-relaxed semantics for the procedures C and A (sections 3.4 and 3.5) — we assume that the transition graphs C and A passed to the algorithm have these semantics. Before beginning the construction of X (and annotation of A), we run an intraprocedural, flow-sensitive, field-insensitive points-to dataflow analysis [3] to compute over-approximate states of the β and β_M maps for each node $n_C \in \mathcal{N}_C$ [1]. These sound but over-approximate values, computed at each node n_C, are substituted in to replace all references to β and β_M in C's graph. After this substitution, the assignments to β and β_M (e.g., in (LOAD$_C$), (STORE$_C$), etc.) become vacuous.

Algorithm 1 presents the pseudo-code of the algorithm. The algorithm has two phases. In the first phase (line 4 in algorithm 1), it attempts to correlate the paths in A with the paths in C while simultaneously identifying the required annotation for A. At the successful completion of the first phase, all paths in the original, non-annotated A are correlated. However, recall that the annotation instructions, $\texttt{alloc}_s$ and $\texttt{alloc}_v$, have additional paths to error nodes $\mathscr{U}_{\ddot{A}}$ and $\mathscr{W}_{\ddot{A}}$ (figs. 2.8, 2.10 and 2.11). These paths to error nodes are not correlated in the first phase. The second phase of the algorithm (line 6 in algorithm 1) correlates these additionally introduced (error) paths.

The sub-procedure *constructX()*, used in both phases, identifies the required correlations and annotation and builds the product program X incrementally. It assumes the availability of an oracle **chooseFrom** operator, such that $\rho \leftarrow\!\shortmid$ **chooseFrom** $\vec{\rho}$ chooses a quantity ρ from a finite set $\vec{\rho}$, such that the algorithm is able to complete the refinement proof, if such a choice exists. If the search space is limited, an exhaustive search could be used to implement **chooseFrom**. For larger search spaces, a counterexample-guided best-first search procedure (described in section 4.1.9) is employed to approximate **chooseFrom**. At a high-level, *constructX()* systematically visits each uncorrelated path $\xi_{\ddot{A}}$ in A and tries to identify a pathset $\langle\xi\rangle_C$ in C that can be correlated with $\xi_{\ddot{A}}$ such that the non-coverage requirements are not violated. If required, $\xi_{\ddot{A}}$ is annotated on-the-fly. We discuss the algorithm in detail in the following sections.

[1] Recall that $\beta(x)$ returns the set of regions that a state variable x in C may point to and $\beta_M(r)$ returns the set of regions that some pointer stored in region r may point to.

Algorithm 1: Automatic construction of X

```
1  Function DYNAMO(A, C, μ)
2      Ä ↤ A;        C ↤ pointsToAnalysis(C);
3      N_X ↤ {(n^s_Ä, n^s_C)};        E_X ↤ ∅;        D_X ↤ ∅;        Φ_X ↤ {(n^s_Ä, n^s_C) ↦ (Ω_Ä = Ω_C)};
4      if ¬constructX(Ä, C, μ, N_X, E_X, D_X, Φ_X, CORRELATE_AND_ANNOTATE) then
5          return Failure
6      if ¬constructX(Ä, C, μ, N_X, E_X, D_X, Φ_X, CORRELATE_NEW_ERROR_PATHS) then
7          return Failure
8      if ¬checkCoverageReqs(N_X, E_X, D_X, Φ_X, Ä, C) then
9          return Failure
10     return Success(Ä, (N_X, E_X, D_X), Φ_X)
11 end
12 Function constructX(Ä, C, μ, N_X, E_X, D_X, Φ_X, phase)
13     Q_Ä ↤ getCutPointsInRPO(Ä);
14     foreach q_Ä in Q_Ä do
15         foreach q^t_Ä in cutPointSuccessorsRPO(q_Ä, Q_Ä, Ä) do
16             foreach ξ_Ä in getAllSimplePathsBetweenCutPoints(q_Ä, q^t_Ä, Ä) do
17                 if pathExists(ξ_Ä, E_X) then
18                     continue
19                 if pathIsInfeasible(ξ_Ä, N_X, Φ_X) then
20                     continue
21                 ⟨ξ⟩_C ↤ chooseFrom correlatedPathsInCOpts(ξ_Ä, μ, N_X, E_X, Ä, C);
22                 foreach ξ_C in ⟨ξ⟩_C do
23                     if phase = CORRELATE_AND_ANNOTATE then
24                         (Ä, ξ_Ä) ↤ chooseFrom asmAnnotOpts(ξ_Ä, ξ_C, Ä, C);
25                     end
26                     ξ⃗'_Ä ↤ breakIntoSingleIOPaths(ξ_Ä);
27                     ξ⃗'_C ↤ breakIntoSingleIOPaths(ξ_C);
28                     ξ⃗*_Ä, ξ⃗*_C ↤ trimToMatchPathToErrorNode(ξ⃗'_Ä, ξ⃗'_C);
29                     if ¬haveSimilarStructure(ξ⃗*_Ä, ξ⃗*_C) then
30                         return Failure
31                     foreach ξ'_Ä = (n_Ä ↠ n^t_Ä), ξ'_C = (n_C ↠ n^t_C) in zip(ξ⃗*_Ä, ξ⃗*_C) do
32                         e_X ↤ (ξ'_Ä; ξ'_C);        n^t_X ↤ (n^t_Ä, n^t_C);
33                         if addingEdgeWillCreateEmptyCCycle(N_X, E_X, e_X) then
34                             return Failure
35                         E_X ↤ E_X ∪ {e_X};        N_X ↤ N_X ∪ {n^t_X};
36                         D_X ↤ addDetMappings(e_X, D_X);
37                         Φ_X ↤ inferInvariantsAndCounterexamples(n^t_X, N_X, E_X, D_X, Φ_X, Ä, C);
38                         if ¬checkSemanticReqsExceptCoverage(N_X, E_X, D_X, Φ_X, Ä, C) then
39                             return Failure
40                     end
41                 end
42             end
43         end
44     end
45     return Success
46 end
```

4.1.1 Enumerating A paths

Cut-points and Simple Paths

We first define some useful predicates. Let $P \in \{\mathsf{C}, \ddot{\mathsf{A}}\}$.

Definition 4.1.1 ($\texttt{io}(n_P)$)**.** $\texttt{io}(n_P)$ *evaluates to* `true` *iff* n_P *is either a source or sink node of an I/O path.*

Definition 4.1.2 ($\texttt{term}(n_P)$)**.** $\texttt{term}(n_P)$ *evaluates to* `true` *iff* n_P *is a terminating node.*

We define an ordered set of nodes $Q_P \subseteq \mathcal{N}_P$, called the *cut-points* in procedure P, such that $Q_P \supseteq \{n_P : n_P \in \mathcal{N}_P \wedge (n_P = n_P^s \vee \texttt{io}(n_P) \vee \texttt{term}(n_P))\}$ and the maximum length of a path between two nodes in Q_P (not containing any other intermediate node that belongs to Q_P) is finite. The algorithm to identify Q_P for a procedure P, *getCutPointsInRPO(P)*, first initializes $Q_P \coloneqq \{n_P : n_P \in \mathcal{N}_P \wedge (n_P = n_P^s \vee \texttt{io}(n_P) \vee \texttt{term}(n_P))\}$, and then identifies all cycles in the transition graph P that do not already contain a cut-point; for each such cycle, the first node belonging to that cycle in reverse postorder is added to Q_P. The cut-points Q_P returned by *getCutPointsInRPO(P)* are arranged in a reverse postorder (RPO).

A *simple path* ($q_P \twoheadrightarrow q_P^t$) is a non-empty path connecting two cut-points $q_P, q_P^t \in Q_P$ and not containing any other cut-point as an intermediate node; q_P^t is called a *cut-point successor* of q_P. By definition, a simple path must be finite.

The *cutPointSuccessorsRPO(q_P, Q_P, P)* function returns the cut-point successors of q_P ordered in reverse postorder. The error node successors of q_P, if any, are ordered after the error-free successors. Further, the error node successor $\mathscr{W}_P$ (if it exists) is arranged at the very end, after all other cut-point successors. This property is expected during path enumeration.

The *getAllSimplePathsBetweenCutPoints(q_P, q_P^t, P)* function returns *all* simple paths of the form $q_P \twoheadrightarrow q_P^t$, for $q_P, q_P^t \in Q_P$. The returned paths are mutually exclusive by construction.

Reverse postorder enumeration of A paths

The procedure *constructX()* first identifies a set of cut-points $Q_{\ddot{\mathsf{A}}}$ of procedure $\ddot{\mathsf{A}}$ using `getCutPointsInRPO`($\ddot{\mathsf{A}}$) (line 13 in algorithm 1). Then, for each cut-point $q_{\ddot{\mathsf{A}}} \in Q_{\ddot{\mathsf{A}}}$,

Algorithm 2: Pseudo-code of the *pathIsInfeasible()* procedure.

```
1 Function pathIsInfeasible(ξ_Ä = (q_Ä ↠ q^t_Ä), N_X, Φ_X)
2     n_X ↤ getXNode(q_Ä, N_X) ;        // unique X node corresponding to source of ξ_Ä
3     if Hoare triple {φ_{n_X}}(ξ_Ä; ϵ){false} holds then
4         return true
5     else
6         return false
7     end
8 end
```

visited in reverse postorder, it identifies the set of cut-point successors of $q^t_Ä$ using `cutPointSuccessorsRPO`$(q_Ä, Q_Ä, Ä)$ (line 15 in algorithm 1). These cut-point successors are visited in reverse postorder, and all simple paths $\vec{\xi}_Ä$ between $q_Ä$ and its cut-point successor $q^t_Ä$ are enumerated through `getAllSimplePathsBetweenCutPoints` $(q_Ä, q^t_Ä, Ä)$ (line 16 in algorithm 1).

The paths $\vec{\xi}_Ä$ are enumerated in reverse postorder (i.e., the source and sink nodes are enumerated in reverse postorder) so that annotation (if any) over a path $\xi_Ä = (q_Ä \twoheadrightarrow q^t_Ä) \in \vec{\xi}_Ä$ is performed *before* the address sets (potentially modified by annotation) are used in successor instructions of $q^t_Ä$. This property ensures consistency in the invariant network Φ_X and enables incremental building of the product program X.

Given a simple path $\xi_Ä = (q_Ä \twoheadrightarrow q^t_Ä) \in \vec{\xi}_Ä$, *pathIsInfeasible*$(\xi_Ä, \mathcal{N}_X, \Phi_X)$ returns `true` iff $\xi_Ä$ is infeasible at every node $n_X = (q_Ä, \_) \in \mathcal{N}_X$; our construction ensures there can be at most one $n_X = (q_Ä, \_)$ for each $q_Ä \in Q_Ä$. An infeasible path is not considered for correlation (line 19 in algorithm 1). Algorithm 2 shows the pseudo-code of *pathIsInfeasible()*.

For an Ä path $\xi_Ä$, *pathExists*$(\xi_Ä, \mathcal{E}_X)$ returns `true` iff $\xi_Ä$ is already correlated with some $\xi_C = (q_C \twoheadrightarrow q^t_C)$ in $\mathcal{E}_X$ (i.e., $\exists e_X : e_X = ((q_Ä, q_C) \xrightarrow{\xi_Ä;\xi_C} (q^t_Ä, q^t_C)) \in \mathcal{E}_X$ holds). Because the same *constructX()* procedure is invoked in both phases, the use of *pathExists()* in line 17 of algorithm 1 is an optimization to avoid correlating the same paths again in the second phase. In the first call to *constructX()*, when *phase* = `CORRELATE_AND_ANNOTATE`, *pathExists*$(\xi_Ä, \mathcal{E}_X)$ will always return `false` as the enumeration returns fresh, uncorrelated paths. However, in the second call to *constructX()*, when *phase* = `CORRELATE_NEW_ERROR_PATHS`, the algorithm is only correlating the newly introduced error paths, and *pathExists*$(\xi_Ä, \mathcal{E}_X)$ will return `false` only for the uncorrelated paths to error nodes introduced due to annotation of Ä.

Trade-off between completeness and efficiency

DYNAMO attempts to correlate each cut-point in $Q_{Ä}$ with a cut point in Q_{C}. To allow maximum transformations, $Q_{Ä}$ should be as small as possible. On the other hand, a smaller $Q_{Ä}$ could result in potentially longer paths between cut-points; a Hoare triple over longer paths could be potentially harder for SMT solvers to reason about due to larger expression sizes. Our algorithm of *getCutPointsInRPO*(P) returns the minimum set of cut-points for a procedure P.

4.1.2 Correlating C paths

Given an Ä path $\xi_{Ä} = (q_{Ä} \twoheadrightarrow q^{t}_{Ä})$, *correlatedPathsInCOpts*($\xi_{Ä}, \mu, \dots$) identifies a non-empty set of *options* for candidate pathset[2] $[\langle\xi\rangle_{C}]$ in C that can potentially be correlated with $\xi_{Ä}$. The **chooseFrom** operator is used to choose a pathset $\langle\xi\rangle_{C}$ from the returned options $[\langle\xi\rangle_{C}]$. All paths in $\langle\xi\rangle_{C}$ must originate at a unique error-free cut-point q_{C} such that $(q_{Ä}, q_{C}) \in \mathcal{N}_{X}$. The enumeration by *correlatedPathsInCOpts()* ensures that there will be exactly one such $(q_{Ä}, q_{C})$ in $\mathcal{N}_{X}$. Paths in $\langle\xi\rangle_{C}$ may have different sinks, however. Unlike $\xi_{Ä}$, a path $\xi_{C} \in \langle\xi\rangle_{C}$ need not be a simple path, and can visit a node $n_{C} \in \mathcal{N}_{C}$ up to μ times — this enables DYNAMO to handle transformations such as loop peeling, unrolling, etc.

Before we describe *correlatedPathsInCOpts()*, we define a $\mathcal{U}$-maximal pathset.

Definition 4.1.3 ($\mathcal{U}$-maximal pathset)**.** *A pathset $\langle\xi\rangle_{P}$ of procedure P is $\mathcal{U}$-maximal iff whenever there exists a path $\xi'_{P} = \epsilon$ or $\xi'_{P} = (q_{P} \twoheadrightarrow q^{t}_{P})$ such that $\xi'_{P} \preceq \xi_{P}$ [3] for $\xi_{P} \in \langle\xi\rangle_{P}$ and $\xi^{u}_{P} = \xi'_{P} \cdot (q^{t}_{P} \rightarrow \mathcal{U}_{P})$ for some edge $(q^{t}_{P} \rightarrow \mathcal{U}_{P}) \in \mathcal{E}_{P}$, $\xi^{u}_{P} \in \langle\xi\rangle_{P}$ holds.*

Informally, a $\mathcal{U}$-maximal pathset $\langle\xi\rangle_{P}$ is a set of paths where the $\mathcal{U}_{P}$-going path of a potentially $\mathcal{U}$-producing instruction is included (as part of a path) in $\langle\xi\rangle_{P}$, if the instruction is included (as part of a path) in $\langle\xi\rangle_{P}$. The union of two $\mathcal{U}$-maximal pathsets, if it is a pathset, is also a $\mathcal{U}$-maximal pathset.

The *correlatedPathsInCOpts()* procedures enumerates $\mathcal{U}$-maximal pathsets as candidate options. Enumerating $\mathcal{U}$-maximal pathsets is essential for a fast SMT encoding (see chapter 5). The key property that $\mathcal{U}$-*maximality* enables is that the pathset condition

[2] Recall that a pathset is a set of pairwise mutually-exclusive paths originating at the same node (definition 3.1.2).

[3] Recall that $\xi'_{P} \preceq \xi_{P}$ iff ξ'_{P} is a prefix of ξ_{P} (section 3.1).

(disjunction of the path conditions of the individual paths) does not depend on the $\mathcal{U}$-triggering condition of an instruction in the pathset.

The DYNAMO algorithm requires *correlatedPathsInCOpts()* to behave differently based on the sink node $q^t_{\ddot{A}}$ of $\xi_{\ddot{A}}$. We present the associated requirements first and then later discuss our implementation.

- **If** $q^t_{\ddot{A}} \notin \{\mathcal{U}_{\ddot{A}}, \mathcal{W}_{\ddot{A}}\}$, i.e. $q^t_{\ddot{A}}$ is an error-free cut-point node, *correlatedPathsInCOpts()* must return candidates where each candidate $\langle\xi\rangle_C$ is a $\mathcal{U}$-maximal non-empty pathset such that each path $\xi_C \in \langle\xi\rangle_C$ starts at q_C and either:

  (a) ends at a unique error-free cut-point node, say q^t_C, i.e., all paths $\xi_C \in \langle\xi\rangle_C$ ending at an error-free node end at q^t_C, or

  (b) ends at error node $\mathcal{U}_C$.

  Because we restrict all paths in $\langle\xi\rangle_C$ with an error-free sink node to have the same sink q^t_C, for a $q^t_{\ddot{A}}$ node, this creates exactly one node $n^t_X = (q^t_{\ddot{A}}, q^t_C) \in \mathcal{N}_X$.

  If $\epsilon \in \langle\xi\rangle_C$, then no other path to an error-free node can be present in $\langle\xi\rangle_C$. In other words, ϵ is treated as a path that starts and ends at q_C.

- **If** $q^t_{\ddot{A}} = \mathcal{U}_{\ddot{A}}$, *correlatedPathsInCOpts()* must return candidates where each candidate $\langle\xi\rangle_C$ is a $\mathcal{U}$-maximal non-empty pathset such that each path $\xi_C \in \langle\xi\rangle_C$ starts at q_C and ends at $\mathcal{U}_C$.

  correlatedPathsInCOpts() must return a pathset option containing the empty path, $\langle\xi\rangle_C = \{\epsilon\}$, if the aforementioned enumeration is not possible, e.g., if no path from q_C to $\mathcal{U}_C$ exists in C. Notice that this choice is never a valid correlation option for a $\mathcal{U}_{\ddot{A}}$-going path because it does not satisfy the (Safety) requirement. However, because *correlatedPathsInCOpts()* must return a non-empty set of options, we simply choose $\langle\xi\rangle_C = \{\epsilon\}$ as a sentinel value so that DYNAMO may fail as early as possible.

- **If** $q^t_{\ddot{A}} = \mathcal{W}_{\ddot{A}}$, *correlatedPathsInCOpts()* must return candidates where each candidate $\langle\xi\rangle_C$ is a $\mathcal{U}$-maximal non-empty pathset such that each path $\xi_C \in \langle\xi\rangle_C$ either has an error-free sink or has the error node $\mathcal{U}_C$ as sink. Unlike the $q^t_{\ddot{A}} \notin \{\mathcal{U}_{\ddot{A}}, \mathcal{W}_{\ddot{A}}\}$ case, the paths to error-free nodes are not required to have the same, unique sink node. In this case, not restricting the enumeration to a unique error-free sink node facilitates a simultaneous proof of (MAC) and (Coverage$C$). We demonstrate this through an example.

  Consider the C and (abstracted) assembly code fragments shown below:

```
q_C :    ...
         // load/store to hp/cl
         ...
         ...
         if (cc) {
q_C^t1 :     fcall1() ...
         } else {
q_C^t2 :     fcall2() ...
         }
```

```
q_Ä :    ...
         // load/store to hp/cl
         ...
q_Ä^t :  esp = ... // ... halt(𝒲)
         if (ca) {
q_Ä^t1 :     fcall1() ...
         } else {
q_Ä^t2 :     fcall2() ...
         }
```

Assume that DYNAMO has already correlated $(q_C, q_Ä)$, $(q_C^{t_1}, q_Ä^{t_1})$, and $(q_C^{t_2}, q_Ä^{t_2})$, and is now trying to correlate $\xi_Ä = (q_Ä \twoheadrightarrow q_Ä^t)$ such that $q_Ä^t = \mathcal{W}_Ä$. Consider the different possibilities for a pathset $\langle\xi\rangle_C$, identified by *correlatedPathsInCOpts()*:

- If $\langle\xi\rangle_C = \{\epsilon, (q_C \twoheadrightarrow \mathcal{U}_C)\}$, such that $\xi_C = \epsilon$ is correlated with $\xi_Ä$ producing $e_X = ((q_Ä, q_C) \xrightarrow{\xi_Ä;\epsilon} (\mathcal{W}_Ä, q_C))$, then e_X would fail to uphold the (MAC) condition because the memory access to *hp* or *cl* (*load/store to hp/cl*) would remain unmatched in $\xi_Ä$ (recall that (MAC) requires a *hp* or *cl* access in $\xi_Ä$ to be *matched* in ξ_C; see also the `checkMAC()` algorithm in section 4.1.6).
- If $\langle\xi\rangle_C = \{(q_C \twoheadrightarrow q_C^{t_1}), (q_C \twoheadrightarrow \mathcal{U}_C)\}$ or $\langle\xi\rangle_C = \{(q_C \twoheadrightarrow q_C^{t_2}), (q_C \twoheadrightarrow \mathcal{U}_C)\}$, i.e., $\langle\xi\rangle_C$ includes paths from either of the two already correlated pathsets (but not both), then (CoverageC) would fail to hold because the correlated path $\xi_C = (q_C \twoheadrightarrow q_C^{t_1})$ (or $(q_C \twoheadrightarrow q_C^{t_2})$) may not execute to completion when $\xi_Ä = (q_Ä \twoheadrightarrow q_Ä^t)$ is executed to completion, i.e., `cc` (or `¬cc`) may not necessarily hold even if $\xi_Ä$, which terminates before reaching corresponding `if` statement, executes to completion.
- If $\langle\xi\rangle_C = \{(q_C \twoheadrightarrow q_C^{t_1}), (q_C \twoheadrightarrow q_C^{t_2}), (q_C \twoheadrightarrow \mathcal{U}_C)\}$, i.e., $\langle\xi\rangle_C$ includes paths with different sink nodes ($q_C^{t_1}$ and $q_C^{t_2}$), then it is easy to see that both (MAC) and (CoverageC) can simultaneously hold.

Note that both (MAC) and (CoverageC) can simultaneously hold when $\xi_Ä = (q_Ä \twoheadrightarrow q_Ä^{t_1})$ is correlated with $\xi_C = (q_C \twoheadrightarrow q_C^{t_1})$ or $\xi_Ä = (q_Ä \twoheadrightarrow q_Ä^{t_2})$ is correlated with $\xi_C = (q_C \twoheadrightarrow q_C^{t_2})$. In both cases, (CoverageC) can hold if `cc` (`¬cc`) and `ca` (`¬ca`) evaluate to identical value in a lockstep execution of ξ_C and $\xi_Ä$. Thus, not restricting the candidates in $\langle\xi\rangle_C$ to a single sink node is required for satisfying (MAC) in the particular case of $q_Ä^t = \mathcal{W}_Ä$.

As *correlatedPathsInCOpts()* identifies options of a pathset, which, by definition, is a set of mutually-exclusive paths, the (MutexC) requirement is satisfied by construction.

Further, because a path $\xi_C = (q_C \twoheadrightarrow \mathscr{W}_C)$ is never returned as part of any candidate, (Well-formedness) is also satisfied.

With an appropriate unroll-factor μ, this path enumeration strategy supports path specializing compiler transformations like loop peeling, unrolling, splitting, unswitching, etc., but does not support a path de-specializing transformation like loop re-rolling. A path de-specialization transformation requires a single $\xi_{Ä} = (q_{Ä} \twoheadrightarrow q^t_{Ä})$ to be correlated with a set of paths $\{\xi^1_C, \xi^2_C, \ldots, \xi^m_C\}$ with potentially different end-points such that $\forall_{1 \leq j \leq m} : \xi^j_C = (q_C \twoheadrightarrow q^j_C)$ and $\forall_{1 \leq j_1 < j_2 \leq m} : q^{j_1}_C \neq q^{j_2}_C$. Such a construction would require an error-free cut-point in Ä to be correlated with multiple error-free cut-points in C — recall that we correlate an error-free cut-point in Ä with exactly one error-free cut-point in C. For comparison, a path specialization transformation requires a cut-point q_C in C to be correlated with multiple cut-points in Ä (which our algorithm supports).

correlatedPathsInCOpts() algorithm

Algorithm 3 shows the pseudo-code of the *correlatedPathsInCOpts()* procedure. The procedure accepts the path $\xi_{Ä} = (q_{Ä} \twoheadrightarrow q^t_{Ä})$, unroll-factor μ, nodes $\mathcal{N}_X$ and edges $\mathcal{E}_X$ of X, and the two procedures Ä and C as input parameters and produces a set of pathsets (pathset options) as output.

correlatedPathsInCOpts() presumes the existence of a pathset enumeration sub-procedure `enumPathsetsTillUnroll()`. The `enumPathsetsTillUnroll`(q_P, q^t_P, μ, P) procedure returns a set of pathsets $\overrightarrow{\langle \xi \rangle}_P$ such that for each pathset $\langle \xi \rangle_P \in \overrightarrow{\langle \xi \rangle}_P$ and each path $\xi_P \in \langle \xi \rangle_P$, ξ_P has q_P as source node and q^t_P as sink node, i.e., $\xi_P = (q_P \twoheadrightarrow q^t_P)$ and maximum unrolling of a node in ξ_P is bounded by μ. Thus, `enumPathsetsTillUnroll()` returns options for a mutually exclusive set of paths such that each path in the set starts at q_P and ends at q^t_P and a node appears at most μ times in a path. The `getFullPathsetAtAllDelta`(q_P, q^t, P, μ) procedure from COUNTER (§3.11 in [17]) satisfies the requirements for `enumPathsetsTillUnroll()` and is used in our implementation. `getFullPathsetAtAllDelta`(q_P, q^t, P, μ) returns a set of (μ, δ)-*unrolled full pathsets* that are constructed such that paths in a returned pathset are disjoint from paths in rest of the pathsets. We will elaborate on (μ, δ)-unrolled full pathset when we talk about representation of a pathset later in this section.

correlatedPathsInCOpts() begins by identifying the unique node q_C such that $n_X = (q_{Ä}, q_C) \in \mathcal{N}_X$ (recall that q_C is guaranteed to exist and be unique). It then constructs

Algorithm 3: Pseudo-code of the *correlatedPathsInCOpts()* algorithm.

```
1  Function correlatedPathsInCOpts(ξ_Ä = (q_Ä ↠ q^t_Ä), μ, N_X, E_X, Ä, C)
2      n_X = (q_Ä, q_C) ↤ getXNode(q_Ä, N_X) ;        // unique X node corresponding to q_Ä
3      ret ↤ {mkPathset({ϵ})} ;      // pathset with empty path is always an option
4      if q^t_Ä = 𝒰_Ä then                    // enumerate options for 𝒰_C-going paths
5          ⟨ξ⟩⃗_C ↤ enumPathsetsTillUnroll(q_C, 𝒰_C, μ, C);
6          foreach ⟨ξ⟩_C ∈ ⟨ξ⟩⃗_C do
7              ret ↤ ret ∪ getAllUMaximalSubsets(⟨ξ⟩_C, C);
8          end
9      else if q^t_Ä = 𝒲_Ä then   // enumerate combinations of already correlated paths
10         e⃗^o_X ↤ getOutgoingEdges(n_X, E_X);
11         ⟨ξ⟩⃗_C ↤ prjCPathsets(e⃗^o_X) ;    // project C pathsets out of outgoing edges
12         ret ↤ ret ∪ ⟨ξ⟩⃗_C;
           /* include the maximal combinations of the pathsets as well          */
13         foreach ⟨ξ⟩_C ∈ ⟨ξ⟩⃗_C do
14             r ↤ ⟨ξ⟩_C;
15             foreach ⟨ξ⟩'_C ∈ ⟨ξ⟩⃗_C do
16                 if unionRemainsMutex(r, ⟨ξ⟩'_C) then
17                     r ↤ unionPathsets(r, ⟨ξ⟩'_C);
18             end
19             ret ↤ ret ∪ r;
20         end
21     else // q^t_Ä ∉ {𝒰_Ä, 𝒲_Ä}:  enumerate μ-unrolled paths to reachable cut-points
22         foreach q^t_C ∈ getAllNonErrorReachableCutPoints(q_C, C) do
23             ⟨ξ⟩⃗_C ↤ enumPathsetsTillUnroll(q_C, q^t_C, μ, C);
24             foreach ⟨ξ⟩^t_C ∈ ⟨ξ⟩⃗_C do
25                 ret ↤ ret ∪ getAllUMaximalSubsets(⟨ξ⟩^t_C, C);
26             end
27         end
28     return ret
29 end
30 Function getAllUMaximalSubsets(⟨ξ⟩_P, P)
31     ret ↤ ∅;
32     foreach ⟨ξ⟩'_P ∈ powerset(⟨ξ⟩_P) do
33         ret ↤ ret ∪ mkUMaximal(⟨ξ⟩'_P, P);
34     end
35     return ret
36 end
```

(using `mkPathset({ϵ})`) the pathset containing the empty path ϵ (line 3 in algorithm 3) as an output option. This ensures that the procedure returns at least one candidate option — even though it may not the correct candidate in all cases, e.g., when $q^t_{Ä} = \mathcal{U}_{Ä}$. The other output options returned by *correlatedPathsInCOpts()* are determined by $q^t_{Ä}$.

When $q^t_{Ä} = \mathcal{U}_{Ä}$ (line 4 in algorithm 3), the procedure uses `enumPathsetsTillUnroll()` to enumerate a set of pathsets $\overrightarrow{\langle\xi\rangle}_C$ containing μ-unrolled paths to $\mathcal{U}_C$. For a pathset

$\langle\xi\rangle_C \in \overrightarrow{\langle\xi\rangle}_C$, the *getAllUMaximalSubsets*$(\langle\xi\rangle_C, C)$ sub-procedure (defined in lines 30 to 36 of algorithm 3) computes a $\mathcal{U}$-maximal pathset for each subset of $\langle\xi\rangle_C$ — computing over each subset of $\langle\xi\rangle_C$ instead of just $\langle\xi\rangle_C$ increases generality at the cost of exponential increase in the number of returned candidates. The ranking and pruning strategies proposed by COUNTER [17] have been demonstrated to help in effectively navigating such large search-spaces. The `mkUMaximal`$(\langle\xi\rangle_P, P)$ function, used in line 33 of algorithm 3, returns the smallest $\mathcal{U}$-maximal pathset $\langle\xi\rangle_P^U$ such that $\langle\xi\rangle_P \subseteq \langle\xi\rangle_P^U$. The computed $\mathcal{U}$-maximal pathsets are added as candidate output options.

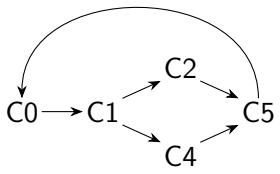
When $q_{\ddot{A}}^t = \mathcal{W}_{\ddot{A}}$ (line 9 in algorithm 3), the procedure returns a set of pathset options derived from the already correlated paths in $\mathcal{E}_X$. Recall that *cutPointSuccessorsRPO*$(q_P, \ldots)$ orders the cut-point successor $\mathcal{W}_{\ddot{A}}$ of q_P after all other cut-point successors of q_P. Due to this, $\xi_{\ddot{A}} = (q_{\ddot{A}} \twoheadrightarrow \mathcal{W}_{\ddot{A}})$ is considered for correlation only after all paths of the form $\xi_{\ddot{A}}^{\not{\mathcal{W}}} = (q_{\ddot{A}} \twoheadrightarrow q_{\ddot{A}}^{\not{\mathcal{W}}})$ (for $q_{\ddot{A}}^{\not{\mathcal{W}}} \neq \mathcal{W}_{\ddot{A}}$) have been correlated. A pathset $\langle\xi\rangle_C^{\not{\mathcal{W}}}$ already correlated with $\xi_{\ddot{A}}^{\not{\mathcal{W}}}$ $((\xi_{\ddot{A}}^{\not{\mathcal{W}}}; \langle\xi\rangle_C^{\not{\mathcal{W}}}) \in \mathcal{E}_X)$ is used as a candidate option.

To increase coverage, the procedure also includes the maximal mutually-exclusive combinations of the pathsets $\overrightarrow{\langle\xi\rangle}_C^{\not{\mathcal{W}}}$ — computed using iterative merging in lines 13 to 20 of algorithm 3 — as candidate options. For example, if there are two pathsets $\overrightarrow{\langle\xi\rangle}_C^{\not{\mathcal{W}}\,1}$ and $\overrightarrow{\langle\xi\rangle}_C^{\not{\mathcal{W}}\,2}$ correlated with paths of the form $\xi_{\ddot{A}}^{\not{\mathcal{W}}}$ (as described above), such that a union of $\overrightarrow{\langle\xi\rangle}_C^{\not{\mathcal{W}}\,1}$ and $\overrightarrow{\langle\xi\rangle}_C^{\not{\mathcal{W}}\,2}$ is also a pathset $\overrightarrow{\langle\xi\rangle}_C^{\not{\mathcal{W}}\,u}$, then all three pathsets $\overrightarrow{\langle\xi\rangle}_C^{\not{\mathcal{W}}\,1}$, $\overrightarrow{\langle\xi\rangle}_C^{\not{\mathcal{W}}\,2}$, and $\overrightarrow{\langle\xi\rangle}_C^{\not{\mathcal{W}}\,u}$ will be included in the candidate options. An example where this is useful was presented earlier in this section when requirements associated with *correlatedPathsInCOpts()* were described. Note that each of the returned candidate pathset will be $\mathcal{U}$-maximal by construction.

When $q^t \notin \{\mathcal{U}_{\ddot{A}}, \mathcal{W}_{\ddot{A}}\}$ (line 21 in algorithm 3), the procedure first identifies all error-free cut-points $Q_C^{\mathcal{UW}}$ $(Q_C^{\mathcal{UW}} \cap \{\mathcal{U}_C, \mathcal{W}_C\} = \emptyset)$ reachable from q_C using the function `getAllNonErrorReachableCutPoints()` (line 22 in algorithm 3). For each cut-point $q_C^t \in Q_C^{\mathcal{UW}}$, pathsets $\overrightarrow{\langle\xi\rangle}_C$ with paths from q_C to q_C^t are enumerated using `enumPathsetsTillUnroll()`. For each enumerated pathset $\langle\xi\rangle_C^t \in \overrightarrow{\langle\xi\rangle}_C$, the procedure identifies a $\mathcal{U}$-maximal pathset for each subset of $\langle\xi\rangle_C^t$ and adds it to the set of candidate options (lines 24 to 26 in algorithm 3).

```
C0:  for (i = 0; i < n; ++i) {
C1:    if (c[i]) {
C2:      x += a[i];
C3:    } else {
C4:      x += b[i];
C5:    }
C6:  }
```

(a) C source fragment.



(b) Abbreviated graph for fig. 4.1a.

Figure 4.1: C source fragment and its abbreviated control-flow graph.

Representation of a pathset

The number of enumerated paths can be exponential in the number of procedure nodes and unroll factor μ. COUNTER [17] suggests use of a compact series-parallel digraph representation, called *SP-graph* representation in [42], for efficiently representing a (μ, δ)*-unrolled full pathset* — a (μ, δ)-unrolled full pathset is a maximal set of pairwise mutually-exclusive paths such that all paths have same source node and same sink node and no node in a path is repeated more than μ times in a path and the sink node repeats exactly δ times in each path ($\delta \leq \mu$). The SP-graph representation of a pathset is a structured serial ($\cdot$) and parallel ($+$) combination defined by the grammar $SP ::= \epsilon \mid e \mid SP \cdot SP \mid SP + SP$, where e is an edge and ϵ represents the empty path. The paths represented by an SP-graph can be enumerated by a recursive traversal of its structure. For example, consider the C procedure fragment and its abbreviated control-flow graph in fig. 4.1. A $(1, 1)$-unrolled full pathset of loop path from `C0`, say $\mathtt{FP}^{1,1}_{\mathtt{C0}\rightsquigarrow\mathtt{C0}}$, has paths corresponding to both branches of `if` at `C1`, i.e., $\mathtt{FP}^{1,1}_{\mathtt{C0}\rightsquigarrow\mathtt{C0}} = \{\mathtt{C0} \to \mathtt{C1} \to \mathtt{C2} \to \mathtt{C5} \to \mathtt{C0},\ \mathtt{C0} \to \mathtt{C1} \to \mathtt{C4} \to \mathtt{C5} \to \mathtt{C0}\}$. The SP-graph of $\mathtt{FP}^{1,1}_{\mathtt{C0}\rightsquigarrow\mathtt{C0}}$ is $(\mathtt{C0} \to \mathtt{C1}) \cdot (((\mathtt{C1} \to \mathtt{C2}) \cdot (\mathtt{C2} \to \mathtt{C5})) + ((\mathtt{C1} \to \mathtt{C4}) \cdot (\mathtt{C4} \to \mathtt{C5}))) \cdot (\mathtt{C5} \to \mathtt{C0})$, or, after abbreviating serial combinations, $\mathtt{FP}^{1,1}_{\mathtt{C0}\rightsquigarrow\mathtt{C0}} = (\mathtt{C0} \to \mathtt{C1}) \cdot (\mathtt{C1} \to \mathtt{C2} \to \mathtt{C5} + \mathtt{C1} \to \mathtt{C4} \to \mathtt{C5}) \cdot (\mathtt{C5} \to \mathtt{C0})$. An another example, the $(4, 4)$-unrolled full pathset of loop path from `C0`, say $\mathtt{FP}^{4,4}_{\mathtt{C0}\rightsquigarrow\mathtt{C0}}$, contains 16 paths and is represented by a serial concatenation of four $\mathtt{FP}^{0,0}_{\mathtt{C0}\rightsquigarrow\mathtt{C0}}$: $\mathtt{FP}^{4,4}_{\mathtt{C0}\rightsquigarrow\mathtt{C0}} = ((\mathtt{C0} \to \mathtt{C1}) \cdot (\mathtt{C1} \to \mathtt{C2} \to \mathtt{C5} + \mathtt{C1} \to \mathtt{C4} \to \mathtt{C5}) \cdot (\mathtt{C5} \to \mathtt{C0}))^4$, where the repetition is indicated by the exponent.

COUNTER correlates a (μ, δ)-unrolled full pathset in a single step (such that a product graph edge contains a (μ, δ)-unrolled full pathset). The primary advantage of using (μ, δ)-unrolled full pathset is that its SP-graph representation enables an efficient SMT

encoding of a Hoare triple over such a pathset. Using such SP-graph representation, it is possible to have linear-sized (in number of nodes in the SP-graph) SMT proof obligations even if the SP-graph represents an exponential number of paths. SMT solvers are able to discharge these linear-sized SMT obligations faster than time taken to discharge the exponential number of proof obligations for individual paths. For example, the *weakest precondition*[14] of predicate $\texttt{x} = 0$ over $\texttt{FP}^{4,4}_{\texttt{C0}\rightsquigarrow\texttt{C0}}$ is a linear sized expression

$$\begin{aligned}\texttt{x} + (\texttt{c[i]} \;\; ?\; \texttt{a[i]} \;\; : \texttt{b[i]})\\ + (\texttt{c[i+1]} ?\; \texttt{a[i+1]} : \texttt{b[i+1]})\\ + (\texttt{c[i+2]} ?\; \texttt{a[i+2]} : \texttt{b[i+2]})\\ + (\texttt{c[i+3]} ?\; \texttt{a[i+3]} : \texttt{b[i+3]}) = 0\end{aligned}$$

where the C-like ternary operator '?:' is used as a shorthand for the 'if-then-else' operator of SMT. SMT solvers are able to discharge one such proof obligation faster than 16 obligations (for each path in $\texttt{FP}^{4,4}_{\texttt{C0}\rightsquigarrow\texttt{C0}}$) performed over individual paths.

Recall that the set of paths in a pathset $\langle\xi\rangle_{\texttt{C}}$ must be mutually exclusive by definition. Further, a candidate pathset $\langle\xi\rangle_{\texttt{C}}$ enumerated for a path $\xi_{\ddot{\texttt{A}}} = (q_{\ddot{\texttt{A}}} \twoheadrightarrow q^{\mathcal{UW}}_{\ddot{\texttt{A}}})$ $(q^{\mathcal{UW}}_{\ddot{\texttt{A}}} \notin \{\mathcal{U}_{\ddot{\texttt{A}}}, \mathcal{W}_{\ddot{\texttt{A}}}\})$ can be partitioned into two pathsets $\langle\xi\rangle^t_{\texttt{C}}$ and $\langle\xi\rangle^u_{\texttt{C}}$ where the pathset $\langle\xi\rangle^t_{\texttt{C}}$ contains paths with an error-free sink node $q^t_{\texttt{C}}$ and the pathset $\langle\xi\rangle^u_{\texttt{C}}$ contains paths with sink node $\mathcal{U}_{\texttt{C}}$. If the $\langle\xi\rangle^t_{\texttt{C}}$ pathset is a (μ,δ)-unrolled full pathset, i.e., the `enumPathsetsTillUnroll()` function uses the `getFullPathsetAtAllDelta()` from [17] (as we do in our implementation) then the SP-graph representation can be used for representing $\langle\xi\rangle^t_{\texttt{C}}$.

4.1.3 Identifying A annotation

For each feasible simple path $\xi_{\ddot{\texttt{A}}}$ and each (potentially non-simple) path $\xi_{\texttt{C}} \in \langle\xi\rangle_{\texttt{C}}$, the *asmAnnotOpts()* procedure enumerates the options for annotating $\xi_{\ddot{\texttt{A}}}$ with $\texttt{alloc}_{s,v}$, $\texttt{dealloc}_{s,v}$ instructions and operands for assembly `call` instructions.

An annotation option includes the positions and the operands of the (de)allocation instructions:

- For an $\texttt{alloc}_s$ instruction '$p^j_{\ddot{\texttt{A}}}$: $\texttt{alloc}_s$ e_v, e_w, a, z', an annotation option would have $p^j_{\ddot{\texttt{A}}}$ as the position, e_v as the start address, e_w as the allocation size, a as the

alignment, and $z \in Z$ as the allocation site.

- For an $\texttt{alloc}_v$ instruction '$p^j_{Ä}$: $\texttt{alloc}_v$ e_w, a, zl', an annotation option would have $p^j_{Ä}$ as the position, e_w as the allocation size, a as the alignment, and $zl \in Z_l$ as the allocation site.

Recall that we restrict $\texttt{alloc}_v$ annotation to allocation sites due to a declaration of a local variable $zl \in Z_l$, excluding allocation sites due to `alloca()` (section 2.6). *asmAnnotOpts()* thus limits enumeration of annotation options for $\texttt{alloc}_v$ to allocation sites Z_l.

For an assembly call instruction '$p^j_{Ä}$: `call` ρ', an annotation option would have the return type γ, the types and values of arguments $(\vec{\tau}\ \vec{x})$, and the set of callee-observable regions β^*. The annotated `call` instruction would be '$p^j_{Ä}$: `call` γ $\rho(\vec{\tau}\,\vec{x})$ β^*'.

asmAnnotOpts() algorithm

Algorithm 4 shows the pseudo-code for the *asmAnnotOpts()* procedure. The procedure takes two correlated paths $\xi_{Ä}$ and ξ_{C} and procedures Ä and C as input and returns the *options* for (potentially) annotated Ä and modified path $\xi^*_{Ä}$ as output.

(SingleIO) requirement enforces a lockstep correlation between (de)allocation and procedure-call instructions in $\xi_{Ä}$ and ξ_{C}. *asmAnnotOpts()* thus annotates a $\texttt{(de)alloc}_{s,v}$, `call` instruction in $\xi_{Ä}$ only if a corresponding `(de)alloc`, `call` instruction is present in ξ_{C}. The call to `collectAllocDeallocCallInsns()` at line 2 in algorithm 4 returns, in $\vec{✤}_{C}$, the list of `(de)alloc` and `call` instructions in input ξ_{C}. *asmAnnotOpts()* attempts to generate annotation options, represented using $\vec{\pi}$, for each returned instruction $✤_{C} \in \vec{✤}_{C}$ (lines 4 to 24 in algorithm 4). The Cartesian product of the generated annotation options $\vec{\pi}$ (for each instruction $✤_{C} \in \vec{✤}_{C}$) forms the annotation candidates $\vec{✏}$ for $\xi_{Ä}$ and each annotation candidate $✏ \in \vec{✏}$, which is a sequence of annotation options (one for each $✤_{C} \in \vec{✤}_{C}$), is separately applied to input Ä to generate an output annotated Ä (and the updated $\xi_{Ä}$, $\xi^*_{Ä}$) option.

In the whitebox setting (lines 6 to 9 in algorithm 4), annotation hints are available that can be used for precisely identifying the required annotation π. These annotation hints are derived from multiple sources, including hints from instrumented compiler (see section 6.1.4 for an example), debug headers of the executable, or manually provided by the user. Further, they annotation hints are of varied quality, for example, hints harvested from debug headers of an highly optimized executable are not very reliable

Algorithm 4: Pseudo-code of the *asmAnnotOpts()* procedure.

```
1  Function asmAnnotOpts(ξ_Ä, ξ_C, Ä, C)
2      ✤⃗_C ↤ collectAllocDeallocCallInsns(ξ_C, C);
3      ✏⃗⃗ ↤ {∅};
4      foreach ✤_C in ✤⃗_C do
5          π⃗ ↤ ∅;
6          if hints are available for ✤_C then // whitebox setting
7              foreach hint h do
8                  π⃗ ↤ π⃗ ∪ genAnnotOptUsingHint(ξ_Ä, ✤_C, ξ_C, h, Ä, C);
9              end
           // blackbox enumeration
10         if ✤_C is alloc then
11             if alloc insn not already present for ✤_C in Ä then
12                 π⃗ ↤ genAnnotOptsForAlloc(ξ_Ä, ✤_C, ξ_C, ✏⃗⃗, Ä, C);
13         else if ✤_C is dealloc then
14             π⃗ ↤ genAnnotOptsForDealloc(ξ_Ä, ✤_C, ξ_C, ✏⃗⃗, Ä, C);
15         else // procedure call
16             π⃗ ↤ genAnnotOptsForFcallUsingCallConvAndCPath(ξ_Ä, ✤_C, ξ_C, ✏⃗⃗, Ä, C);
17         ✏⃗⃗′ = ∅;
           // cross-product all possibilities with existing options
18         foreach ✏ in ✏⃗⃗ do
19             foreach π in π⃗ do
20                 ✏⃗⃗′ ↤ ✏⃗⃗′ ∪ {✏ · π};
21             end
22         end
23         ✏⃗⃗ ↤ ✏⃗⃗′;
24     end
25     ret ↤ ∅;
       // generate an annotated Ä for each option
26     foreach ✏ in ✏⃗⃗ do
27         Ä′, ξ*_Ä ↤ applyAnnots(Ä, ξ_Ä, ✏);
28         ret ↤ ret ∪ (Ä′, ξ*_Ä);
29     end
30     return ret
31 end
```

[26]. We use all available hints for generating (multiple) annotation options. A best-first search implementation (see section 4.1.9) may choose to rank options from certain sources (e.g., hints from instrumented compiler) ahead of others.

If no annotation hints are available, the correlated `(de)alloc/call` instruction $✤_C$ in ξ_C can be utilized for deducing partial annotation. The allocation size, alignment, and allocation site operands of an $\texttt{(de)alloc}_{s,v}$ instruction are uniquely identified from the correlated `(de)alloc` instruction $✤_C$. Similarly, the arguments' count and types, the return type, and the callee-observable regions for a `call` instruction in $\xi_Ä$ are identified using a correlated `call` instruction $✤_C$; the deduction of arguments'

count further enables identification of the addresses for the arguments of `call` using calling conventions (`genAnnotOptsForFcallUsingCallConvAndCPath()` at line 16 in algorithm 4).

After the deduction of other parameters, only the position and start address[4] of an (de)$\texttt{alloc}_{s,v}$ instruction need to be determined (or guessed) for a complete annotation. For a (de)$\texttt{alloc}_{s}$ due to a procedure call argument, the position and start address are determined based on argument order and calling conventions. For rest of the (de)$\texttt{alloc}_{s,v}$ instructions, we reduce the search space for the position and start address (at the cost of reduced generality) using the following three restrictions:

1. An $\texttt{alloc}_{s,v}$ ($\texttt{dealloc}_{s,v}$) annotation is added only after (before) an instruction that updates the stackpointer `esp`.
2. For an $\texttt{alloc}_{s}$ instruction, the stackpointer value `esp` after the update is used as the annotation for the start address expression.
3. For a single allocation site in C, at most one $\texttt{alloc}_{s,v}$ instruction (but potentially multiple $\texttt{dealloc}_{s,v}$ instructions) is added to Ä (line 11 in algorithm 4).

Thus, in a blackbox setting, due to the third restriction, a refinement proof may fail if the compiler specializes a path containing a local variable allocation. Due to the first and second restriction, a refinement proof may fail for certain (arguably rare) types of (de)allocation order preserving stack reallocation and stack merging performed by the compiler. An example is discussed in section 6.2.4. Note that these limitations hold only for the blackbox setting.

An annotation option π, obtained either from whitebox hints or blackbox enumeration, is accumulated in each sequence of annotation options $✏ \in \vec{✏}$ enumerated so far (lines 18 to 22 in algorithm 4). Once all instructions in $\vec{✤}_C$ have been considered, $\vec{✏}$ contains the Cartesian product of annotation options enumerated for each $✤_C \in \vec{✤}_C$ and each $✏ \in \vec{✏}$ forms an annotation candidate for $\xi_{Ä}$.

Each enumerated annotation candidate $✏ \in \vec{✏}$ is applied separately to A to incrementally construct Ä (`applyAnnots()` at line 27 in algorithm 4). The application of an annotation may potentially update the path $\xi_{Ä}$ to $\xi^*_{Ä}$ due to addition of edges for (de)$\texttt{alloc}_{s,v}$ and annotated `call`. *asmAnnotOpts()* returns the options for the updated Ä and $\xi^*_{Ä}$.

[4]start address is only required for the $\texttt{alloc}_{s}$ instruction

Algorithm 5: Pseudo-code of the *trimToMatchPathToErrorNode()* procedure.

```
1 Function trimToMatchPathToErrorNode(ξ⃗_Ä, ξ⃗_C)
2     l ↤ min(|ξ⃗_Ä|, |ξ⃗_C|);
3     ξ⃗*_Ä, ξ⃗*_C ↤ take(l, ξ⃗_Ä), take(l, ξ⃗_C); ; // take first l elements from ξ⃗_Ä and ξ⃗_C
4     if sink(last(ξ⃗*_Ä)) ∈ {𝒰_Ä, 𝒲_Ä} or sink(last(ξ⃗*_C)) = 𝒰_C then // return trimmed
        sequences if either ends with error node
5         return (ξ⃗*_Ä, ξ⃗*_C)
6     else // otherwise, return the original sequences
7         return ξ⃗_Ä, ξ⃗_C
8 end
```

4.1.4 Validating structure of identified paths

asmAnnotOpts() produces a set of options for annotation and the **chooseFrom** operator chooses one such that (if possible) the annotated instructions generate identical traces in Ä and C.

It must be emphasized here that due to these annotated instructions, extra paths to error nodes $\mathcal{U}_{Ä}$ and $\mathcal{W}_{Ä}$ are added to Ä — recall the multi-line graph instructions translations of (de)alloc$_{s,v}$ instructions, with checks for overlap and alignment, presented in (AllocS'), (DeallocS'), (AllocV), and (DeallocV). These paths are *not* a part of the annotated $\xi^*_{Ä}$ that contains the error-free sub-paths of the annotated (de)alloc$_{s,v}$ instructions. Instead, the correlation of these extra paths to error nodes happens in second phase of the algorithm (when *phase* = CORRELATE_NEW_ERROR_PATHS; recall that the algorithm operates in two phases).

After annotation, the annotated path $\xi^*_{Ä}$ may become a non-simple path[5] due to the extra I/O instructions introduced by the annotation. The (potentially non-simple) path $\xi^*_{Ä}$ is therefore broken into a sequence of constituent paths $\vec{\xi}'_{Ä}$ using *breakIntoSingleIOPaths()* (line 26 of algorithm 1) so that each I/O path appears by itself (and not as a sub-path of a longer constituent path) — this caters to the (SingleIO) requirement (section 3.3.1). *breakIntoSingleIOPaths()* is similarly used on ξ_C to obtain a sequence of simple paths $\vec{\xi}'_{C}$.

The (SingleIO) requirement requires that each I/O path $\xi'_{Ä} \in \vec{\xi}'_{Ä}$ is correlated separately with an I/O path $\xi'_{C} \in \vec{\xi}'_{C}$ of similar kind. However, the sequences of simple paths $\vec{\xi}'_{Ä}$ and $\vec{\xi}'_{C}$ obtained after *breakIntoSingleIOPaths()* may not have identical lengths. For

[5]Recall that a simple path cannot have a cut-point as an intermediate node.

Algorithm 6: Pseudo-code of the *haveSimilarStructure()* procedure.

```
1  Function haveSimilarStructure(ξ⃗_Ä, ξ⃗_C)
2      if |ξ⃗_Ä| ≠ |ξ⃗_C| then
3          return false
4      (n_Ä, n^t_Ä) ↤ src(first(ξ⃗_Ä)), sink(last(ξ⃗_Ä));
5      (n_C, n^t_C) ↤ src(first(ξ⃗_C)), sink(last(ξ⃗_C));
6      if n^t_Ä = 𝒰_Ä ∧ n^t_C ≠ 𝒰_C then // (Safety)
7          return false
8      if n^t_Ä ∉ {𝒰_Ä, 𝒲_Ä} ∧ n^t_C ∉ {𝒰_C, 𝒲_C} ∧ term(n^t_Ä) ≠ term(n^t_C) then // (Termination)
9          return false
       // (SingleIO) --- each I/O path-pair is of same kind
10     foreach (ξ_Ä, ξ_C) in zip(ξ⃗_Ä, ξ⃗_C) do
11         if isIOPath(ξ_Ä) ≠ isIOPath(ξ_C) then
12             return false
13         if isIOPath(ξ_Ä) ∧ ¬IOPathsOfSameKind(ξ_Ä, ξ_C) then
14             return false
15     end
16     return true
17 end
```

example, if $\xi_{Ä}$ is a path to an error-free node $n^t_{Ä}$ and ξ_C is a path to the error node $\mathcal{U}_C$, then the sequence $\vec{\xi}'_{Ä}$ may be larger than $\vec{\xi}'_C$. In such a scenario, because $\mathcal{U}_C$ is a terminating node, the path sequence $\vec{\xi}'_{Ä}$ can be *trimmed* to make the sequence lengths identical. This trimming is permissible because under the refinement definition the generated traces are required to be identical only till C halts with error $\mathcal{U}$ or Ä halts with error $\mathcal{W}$ (section 2.4). The procedure *trimToMatchPathToErrorNode()* (line 28 of algorithm 1) attempts to make the lengths of $\vec{\xi}'_{Ä}$ and $\vec{\xi}'_C$ identical by trimming $\vec{\xi}'_{Ä}$ and $\vec{\xi}'_C$ in this fashion. Algorithm 5 shows the pseudo-code of *trimToMatchPathToErrorNode()*.

Next, the *haveSimilarStructure()* procedure validates the structure of $\vec{\xi}_{Ä}$ and $\vec{\xi}_C$ obtained after *trimToMatchPathToErrorNode()* (line 29 in algorithm 1). *haveSimilarStructure*($\vec{\xi}_{Ä}$, $\vec{\xi}_C$) returns true iff the sequence of paths $\vec{\xi}_{Ä}$ and $\vec{\xi}_C$ have identical lengths and are *similarly structured*, where structural similarity is defined with respect to the structural requirements of X. Algorithm 6 shows the pseudo-code of *haveSimilarStructure()*. Let $\mathtt{pos}(\xi, \vec{\xi})$ represent the position of path ξ in a sequence of paths $\vec{\xi}$. Let $\xi^j_{Ä}$ and ξ^j_C denote paths such that $\xi^j_{Ä} \in \vec{\xi}_{Ä}$, $\xi^j_C \in \vec{\xi}_C$ and $\mathtt{pos}(\xi_C, \vec{\xi}_C) = \mathtt{pos}(\xi_{Ä}, \vec{\xi}_{Ä}) = j$; we will refer to the pair $(\xi^j_{Ä}, \xi^j_C)$ as a "coupled path-pair". *haveSimilarStructure()* ensures that, i.e., it returns `true` if, if the partially-constructed X satisfies the structural requirements of section 3.3.1, then it will continue to satisfy the (SingleIO), (Safety), and (Termination) requirements after adding edges corresponding to coupled path-pairs $e^j_X = (\xi^j_{Ä}, \xi^j_C)$ (for all j) to X. Let $\xi_{Ä}$ and ξ_C be the last paths in the path sequences

$\vec{\xi}_{Ä}$ and $\vec{\xi}_{C}$ respectively such that $\texttt{pos}(\xi_{Ä}, \vec{\xi}_{Ä}) = \texttt{pos}(\xi_{C}, \vec{\xi}_{C}) = |\vec{\xi}_{Ä}|$. (Safety) requires that if the sink node $n^t_{Ä}$ of $\xi_{Ä}$ is $\mathcal{U}_{Ä}$, then the sink node n^t_{C} of ξ_{C} must be $\mathcal{U}_{C}$ and (Termination) requires that if both $n^t_{Ä}$ and n^t_{C} are error-free, then they must agree on terminating status, i.e., $\texttt{term}(n^t_{Ä}) = \texttt{term}(n^t_{C})$. The (SingleIO) check obligations include ensuring that a coupled I/O path-pair $(\xi^j_{Ä}, \xi^j_{C})$ is an I/O path-pair of *same kind*. This is expressed through `IOPathsOfSameKind()` in algorithm 6 wherein $\texttt{IOPathsOfSameKind}(\xi^j_{Ä}, \xi^j_{C})$ returns true iff $\xi^j_{Ä}$ and ξ^j_{C} are either both reads or both writes for the same type of value (implemented as syntactic checks on the read/written value[6]).

4.1.5 Incremental construction of (Ä, X)

At this point (line 31 in algorithm 1), we have two sequences of paths $\vec{\xi}^*_{Ä}$ and $\vec{\xi}^*_{C}$ that have identical number of elements. In its next step, the DYNAMO algorithm constructs an X edge $e_X = (\xi'_{Ä} = (n_{Ä} \twoheadrightarrow n^t_{Ä}), \xi'_{C} = (n_{Ä} \twoheadrightarrow n^t_{Ä}))$ from a coupled path-pair $(\xi'_{Ä}, \xi'_{C})$ (through `zip()` in line 31 of algorithm 1). To ensure (Similar-Speed) requirement (section 3.3.1), the algorithm checks that the addition of e_X does not create an empty cycle of C paths in $\mathcal{E}_X$ (*addingEdgeWillCreateEmptyCCycle()* in algorithm 1). We only need to perform the check on C paths because our enumeration guarantees that $\xi'_{Ä}$ is never empty — recall that *correlatedPathsInCOpts()* allows a path ξ_{C} in a pathset $\langle\xi\rangle_{C}$ to be empty (so that ξ'_{C} can be empty) but $\xi_{Ä}$, as returned by *getAllSimplePathsBetweenCutPoints()*, can never be empty. The constructed edge $e_X = (\xi'_{Ä}; \xi'_{C})$ is added to $\mathcal{E}_X$ and its destination node $(n^t_{Ä}, n^t_{C})$ is added to $\mathcal{N}_X$, if not already present.

If ξ'_{C} contains an edge labeled with a choose instruction ('$\vec{v} := \theta(\vec{\tau})$'), then we update the deterministic choice map $\mathcal{D}_X$ to include determinized mappings for each choose instruction in ξ'_{C} through *addDetMappings()*; algorithm 7 shows the pseudo-code of *addDetMappings()*. For example, if ξ'_{C} represents a path between $\texttt{wr}(\texttt{allocBegin}(\ldots),)$ and $\texttt{wr}(\texttt{allocEnd}(\ldots),)$ for an `alloc` instruction in C ((ALLOC) in fig. 2.5), and $\xi'_{Ä}$ is a corresponding path due to an $\texttt{alloc}_{s,v}$ instruction, and edges $e^{\theta_a}_{C}$ and $e^{\theta_m}_{C}$ in ξ'_{C} are labeled with instructions '$\alpha_b := \theta(\texttt{i}_{32})$' and '$\theta(\texttt{i}_{32} \rightarrow \texttt{i}_8)$' respectively due to (ALLOC), we add mappings $\mathcal{D}_X(e_X, e^{\theta_a}_{C}, 1) = v$ and $\mathcal{D}_X(e_X, e^{\theta_m}_{C}, 1) = M_{Ä}$, where v is the address defined in $\xi'_{Ä}$ due to either $\texttt{alloc}_s$ ((ALLOCS)) or $\texttt{alloc}_v$ ((ALLOCV)). In algorithm 7, the $e^{\theta_a}_{C}$ edge is identified using `identifyAllocAddr`θ`Edge()` and the

[6]Recall that we use different value constructors for different instructions; see item 4 in section 2.2.7

Algorithm 7: Pseudo-code of the *addDetMappings()* and *inferInvariantsAndCounterexamples()* procedures.

```
1  Function addDetMappings(e_X, D_X)
2      D_X' ↤ D_X;
3      (ξ_Ä, ξ_C) ↤ e_X;
4      if e_C^{θa} ↤ identifyAllocAddrθEdge(ξ_C) then
5          e_C^{θm} ↤ identifyAllocMemθEdge(ξ_C);
6          D_X'(e_X, e_C^{θa}, 1) ↤ identifyAllocAddr(ξ_Ä); ; // v in (AllocS),(AllocV)
7          D_X'(e_X, e_C^{θm}, 1) ↤ M_Ä;
8      else if e'_C ↤ identifyEntryMemθEdge(ξ_C) then
9          D_X'(e_X, e'_C, 1) ↤ M_Ä;
10     return D_X'
11 end
12 Function inferInvariantsAndCounterexamples(n_X, N_X, E_X, D_X, Φ_X, Ä, C)
13     if n_X is an error node then
14         return
15     ☆_Ä(n_X) ↤ computeInterestingExprsAtNodeForA(Ä, N_X, E_X, Φ_X);
16     ☆_C(n_X) ↤ computeInterestingExprsAtNodeForC(C, N_X, E_X, Φ_X);
17     Φ_X ↤ inferInvariantsForExprs(☆_Ä, ☆_C, N_X, E_X, D_X, Φ_X, Ä, C);
18 end
```

$e_C^{\theta_m}$ edge is identified using `identifyAllocMemθEdge()`. Notice that our algorithm only populates $\mathcal{D}_X(e_X, e_C^{\theta}, n)$ for $n = 1$, even though section 3.2 defines $\mathcal{D}_X$ more generally. Algorithm 7 also populates $\mathcal{D}_X$ for an edge e'_C that corresponds to the '$\theta(i_{32} \rightarrow i_8)$' edge due to procedure-entry ((Entry$_C$) in fig. 2.5 and (Entry$_A$) in fig. 2.7), identified using `identifyEntryMemθEdge()` in algorithm 7.

Recall that Dynamo maintains the invariant that all non-coverage requirements hold over the partially-constructed X. So far, the algorithm has ensured that each structural requirement is satisfied by the newly added edge $e_X = (n_X \rightarrow (n^t_Ä, n^t_C))$. If the destination node $(n^t_Ä, n^t_C)$ is an error-free node, the invariant network Φ_X may no longer be inductive due to the addition of the new edge, thereby violating the (Inductive) requirement (see section 3.3.1). The *inferInvariantsAndCounterexamples()* procedure, shown in algorithm 7, updates Φ_X to ensure its inductivity. To keep invariant inference tractable, the state elements and expressions participating in invariant inference at a node $n_X = (n_Ä, n_C)$ are restricted to *interesting expressions* at n_X — which include at least the live registers, ghost variables, and stack slots at $n_Ä$ in Ä (shown as $☆_Ä(n_X)$ in algorithm 7) and a subset of all defined variables (including ghost variables) for C (shown as $☆_C(n_X)$ in algorithm 7). These choices are similar to the ones considered in [42]. We defer a detailed discussion of the invariant inference, including a description of the candidate invariant grammar, to section 4.2.

Algorithm 8: Pseudo-code of the *checkSemanticReqsExceptCoverage()* procedure.

```
1  Function checkSemanticReqsExceptCoverage(N_X, E_X, D_X, Φ_X, Ä, C)
2      if ¬invariantsAreInductive(N_X, E_X, D_X, Φ_X, Ä, C) then
3          return false
4      if ¬checkEquivalence(N_X, E_X, D_X, Φ_X, Ä, C) then
5          return false
6      if ¬checkMAC(N_X, E_X, D_X, Φ_X, Ä, C) then
7          return false
8      if ¬checkMemEq(N_X, E_X, D_X, Φ_X, Ä, C) then
9          return false
10     return true
11 end
```

4.1.6 Checking requirements on partial X

Once the invariant network has been updated, the non-coverage semantic requirements are checked through *checkSemanticReqsExceptCoverage()* procedure. A partially constructed X that fails to satisfy the requirements check is discarded[7]. The pseudo-code of *checkSemanticReqsExceptCoverage()*, shown in algorithm 8, performs checks for the following semantic requirements on the partial X: (Inductive), (Equivalence), (Memory Access Correspondence) or (MAC), and (MemEq). The (Inductive) check is realized through `invariantsAreInductive()`, which fails if the invariant network Φ_{X} is not inductive. Similarly, the (Equivalence), (MemEq), and (Memory Access Correspondence) or (MAC) checks are realized through `checkEquivalence()`, `checkMemEq()`, and `checkMAC()` respectively. The former two, `checkEquivalence()` and `checkMemEq()`, simply involve checking if the set of invariants $\phi_{n_{\mathsf{X}}}$, inferred at each error-free node $n_{\mathsf{X}} \in \mathcal{N}_{\mathsf{X}}$, include the required invariants, $\Omega_{\mathsf{C}} = \Omega_{\ddot{\mathsf{A}}}$ and $M_{\ddot{\mathsf{A}}} =_{\Sigma^{B}_{\ddot{\mathsf{A}}} \setminus (\Sigma^{Z_l}_{\ddot{\mathsf{A}}}|^{\nu})} M_{\mathsf{C}}$ respectively[8].

(Memory Access Correspondence) or (MAC) check

The pseudo-code for the `checkMAC()` sub-procedure is shown in algorithm 8. Recall that the (MAC) requirement check (section 3.3.1) entails ensuring that for each edge $e_{\mathsf{X}} = (n_{\mathsf{X}} \xrightarrow{\xi_{\ddot{\mathsf{A}}}; \xi_{\mathsf{C}}} n^{t}_{\mathsf{X}}) \in \mathcal{E}_{\mathsf{X}}$, such that $n^{t}_{\mathsf{X}} \neq (\_, \mathscr{U}_{\mathsf{C}})$ and for each memory access to interval $[\alpha]_w$ in $\xi_{\ddot{\mathsf{A}}}$, either:

[7] Recall that the **chooseFrom** operator chooses option such that this check does not fail if the required X can be constructed using DYNAMO.

[8] As we will see later in section 4.2, these are instantiations of the invariant shapes `WEq` and `MemEq` respectively.

Algorithm 9: Pseudo-code of the *checkMAC()* procedure.

Function *checkMAC*($\mathcal{N}_{\mathsf{X}}, \mathcal{E}_{\mathsf{X}}, \mathcal{D}_{\mathsf{X}}, \Phi_{\mathsf{X}}, \ddot{\mathsf{A}}, \mathsf{C}$)
  foreach $\xi_{\ddot{\mathsf{A}}}, \xi_{\mathsf{C}}$ *such that* $e_{\mathsf{X}} = (n_{\mathsf{X}} \xrightarrow{\xi_{\ddot{\mathsf{A}}}; \xi_{\mathsf{C}}} n'_{\mathsf{X}}) \in \mathcal{E}_{\mathsf{X}}$ *and* $n'_{\mathsf{X}} \neq (\_, \mathcal{U}_{\mathsf{C}})$ **do**
    $memAcc_{\mathsf{C}} \leftarrow$ `collectMemAccessesInPath`(ξ_{C});
    $isSafe_{\mathsf{C}}$(`rd`) $\leftarrow \bigwedge_{(\xi^j_{\mathsf{C}}, \alpha_j, w_j, \mathtt{rd}) \in memAcc_{\mathsf{C}}} \mathtt{WP}_{[\xi^j_{\mathsf{C}}]^{ex}_{\mathcal{D}_{\mathsf{X}}}}(accessIsSafe(\alpha_j, w_j, \mathtt{rd}, \mathsf{C}))$;
    $isSafe_{\mathsf{C}}$(`wr`) $\leftarrow \bigwedge_{(\xi^j_{\mathsf{C}}, \alpha_j, w_j, \mathtt{wr}) \in memAcc_{\mathsf{C}}} \mathtt{WP}_{[\xi^j_{\mathsf{C}}]^{ex}_{\mathcal{D}_{\mathsf{X}}}}(accessIsSafe(\alpha_j, w_j, \mathtt{wr}, \mathsf{C}))$;
    foreach $(\xi^j_{\ddot{\mathsf{A}}}, \alpha_j, w_j, k_j) \in$ `collectMemAccessesInPath`($\xi_{\ddot{\mathsf{A}}}$) **do**
      $isSafe_{\ddot{\mathsf{A}}} \leftarrow accessIsSafe(\alpha_j, w_j, k_j, \ddot{\mathsf{A}})$;
      if *Hoare triple* $\{\phi_{n_{\mathsf{X}}} \wedge pathcond([\xi_{\mathsf{C}}]^{ex}_{\mathcal{D}_{\mathsf{X}}})\}(\xi^j_{\ddot{\mathsf{A}}}; \epsilon)\{isSafe_{\mathsf{C}}(k_j) \implies isSafe_{\ddot{\mathsf{A}}}\}$ *does not hold* **then**
        return `false`
    end
  end
  return `true`
end
Function *accessIsSafe*(α, w, k, P)
  `// page size parametric access-is-safe condition generator`
  `pgMask` $\leftarrow \sim$(`PAGE_SIZE` $- 1$);
  $un \leftarrow$ *unique variable different from all state variables in* C *and* $\ddot{\mathsf{A}}$;
  $unL \leftarrow un$ & `pgMask`; $unH \leftarrow unL +$ `PAGE_SIZE` $- 1$;
  $l, h \leftarrow \alpha, \alpha + w - 1$;
  $inUnalloc \leftarrow$ `ov`$([l, h], [unL, unH])$;
  if $P = \ddot{\mathsf{A}}$ **then**
    if $k =$ `rd` **then**
      $inGFS \leftarrow [l, h] \subseteq (\Sigma^{G \cup F}_{\ddot{\mathsf{A}}} \cup [\mathtt{esp}, \boxed{\mathtt{stk}_e}])$;
    else `//` $k =$ `wr`
      $inGFS \leftarrow [l, h] \subseteq (\Sigma^{G_w \cup F_w}_{\ddot{\mathsf{A}}} \cup [\mathtt{esp}, \boxed{\mathtt{stk}_e}])$;
    return $inUnalloc \implies inGFS$)
  else `//` $P = \mathsf{C}$
    return $\neg inUnalloc$
end

- A corresponding memory access (a corresponding read or a corresponding write) to identical interval is present in ξ_{C}. In other words, the access is *matched* in ξ_{C}.

- Or, for a read access, $[\alpha]_w \subseteq (\Sigma^{G \cup F}_{\ddot{\mathsf{A}}} \cup [\mathtt{esp}, \boxed{\mathtt{stk}_e}])$, and for a write access, $[\alpha]_w \subseteq (\Sigma^{G_w \cup F_w}_{\ddot{\mathsf{A}}} \cup [\mathtt{esp}, \boxed{\mathtt{stk}_e}])$.

In the `checkMAC()` procedure, the first clause is generalized to *page-granularity* where an access in $\xi_{\ddot{\mathsf{A}}}$ is deemed to be matched if it remains within the same memory page as an access in ξ_{C}, i.e., an access interval $[\alpha]_w$ ($[\alpha]_w \subseteq \boxed{\Sigma^{\mathtt{rd}}_{\ddot{\mathsf{A}}}}$ or $[\alpha]_w \subseteq \boxed{\Sigma^{\mathtt{wr}}_{\ddot{\mathsf{A}}}}$) is matched

iff the following holds:

$$\forall_p : \begin{pmatrix} \texttt{aligned}_{\texttt{PAGE\_SIZE}}(p) \\ \wedge\ \texttt{ov}([\alpha]_w, [p]_{\texttt{PAGE\_SIZE}}) \end{pmatrix} \implies \exists_{\alpha', w'} : \begin{pmatrix} [\alpha']_{w'} \subseteq \boxed{\Sigma_{\mathsf{C}}^{\texttt{rd/wr}}} \\ \wedge\ \texttt{ov}([\alpha']_{w'}, [p]_{\texttt{PAGE\_SIZE}}) \end{pmatrix}$$

where $\boxed{\Sigma_{\mathsf{C}}^{\texttt{rd/wr}}} = \boxed{\Sigma_{\mathsf{C}}^{\texttt{rd}}}$ if $[\alpha]_w \subseteq \boxed{\Sigma_{\ddot{\mathsf{A}}}^{\texttt{rd}}}$ and $\boxed{\Sigma_{\mathsf{C}}^{\texttt{rd/wr}}} = \boxed{\Sigma_{\mathsf{C}}^{\texttt{wr}}}$ if $[\alpha]_w \subseteq \boxed{\Sigma_{\ddot{\mathsf{A}}}^{\texttt{wr}}}$, and PAGE_SIZE is the size of the page (e.g., PAGE_SIZE = 4096). This page granular version is equivalent to the original requirement when PAGE_SIZE = 1.

The checkMAC() implementation in algorithm 8 formulates the (MAC) check as a search for an arbitrary *unallocated* page of size PAGE_SIZE starting at an (page-aligned) address *un* such that the access interval $[\alpha]_w$ (in $\ddot{\mathsf{A}}$) overlaps with it but none of the memory accesses in ξ_{C} overlap with it.

Let $e_{\mathsf{X}} = ((n_{\ddot{\mathsf{A}}}, n_{\mathsf{C}}) \xrightarrow{\xi_{\ddot{\mathsf{A}}};\xi_{\mathsf{C}}} n_{\mathsf{X}}^t)$ such that $n_{\mathsf{X}}^t \neq (\_, \mathcal{U}_{\mathsf{C}})$. A memory access on $\xi_{\ddot{\mathsf{A}}}$ to address $\alpha_{\ddot{\mathsf{A}}}$, of size $w_{\ddot{\mathsf{A}}}$, and kind $k_{\ddot{\mathsf{A}}} \in \{\texttt{rd}, \texttt{wr}\}$ ($k_{\ddot{\mathsf{A}}} = \texttt{rd}$ for memory read and $k_{\ddot{\mathsf{A}}} = \texttt{wr}$ for memory write) reachable after traversing path $\xi'_{\ddot{\mathsf{A}}}$ starting from $n_{\ddot{\mathsf{A}}}$ is represented by tuple $(\xi'_{\ddot{\mathsf{A}}}, \alpha_{\ddot{\mathsf{A}}}, w_{\ddot{\mathsf{A}}}, k_{\ddot{\mathsf{A}}})$. Similarly, $(\xi_{\mathsf{C}}^j, \alpha_{\mathsf{C}}^j, w_{\mathsf{C}}^j, k_{\mathsf{C}}^j)$ for $1 \leq j \leq m$ represents jth memory access (out of m) on ξ_{C} of same kind as $k_{\ddot{\mathsf{A}}}$, i.e., $\forall_{1 \leq j \leq m} : k_{\mathsf{C}}^j = k_{\ddot{\mathsf{A}}}$. Let $\texttt{WP}_{\xi_P}(\alpha)$ represent *weakest precondition*[14] of α after executing ξ_P. An access $(\xi'_{\ddot{\mathsf{A}}}, \alpha_{\ddot{\mathsf{A}}}, w_{\ddot{\mathsf{A}}}, k_{\ddot{\mathsf{A}}})$ is *safe* iff the following holds:

$$\begin{pmatrix} \phi_{(n_{\ddot{\mathsf{A}}}, n_{\mathsf{C}})} \\ \wedge\ pathcond(\xi'_{\ddot{\mathsf{A}}}) \wedge pathcond([\xi_{\mathsf{C}}]_{\mathcal{D}_{\mathsf{X}}}^{e_{\mathsf{X}}}) \\ \wedge\ \bigwedge_{j=1}^{m} \texttt{WP}_{\xi_{\mathsf{C}}^j}(\neg\texttt{ov}([un]_{\texttt{PAGE\_SIZE}}, [\alpha_{\mathsf{C}}^j]_{w_{\mathsf{C}}^j})) \end{pmatrix} \implies \texttt{WP}_{\xi'_{\ddot{\mathsf{A}}}}\Big(\begin{array}{l} \texttt{ov}([un]_{\texttt{PAGE\_SIZE}}, [\alpha_{\ddot{\mathsf{A}}}]_{w_{\ddot{\mathsf{A}}}}) \\ \implies [\alpha_{\ddot{\mathsf{A}}}]_{w_{\ddot{\mathsf{A}}}} \subseteq \Sigma^{GFS} \end{array} \Big) \tag{4.1}$$

where $\Sigma^{GFS} = \Sigma_{\ddot{\mathsf{A}}}^{G \cup F} \cup [\texttt{esp}, \boxed{\texttt{stk}_e}]$ if $k_{\ddot{\mathsf{A}}} = \texttt{rd}$, and $\Sigma^{GFS} = \Sigma_{\ddot{\mathsf{A}}}^{G_w \cup F_w} \cup [\texttt{esp}, \boxed{\texttt{stk}_e}]$ if $k_{\ddot{\mathsf{A}}} = \texttt{wr}$ (see procedure *accessIsSafe()* in algorithm 9).

It is easy to see that eq. (4.1) will not hold for an access $[\alpha_{\ddot{\mathsf{A}}}]_{w_{\ddot{\mathsf{A}}}}$ if it is neither matched by any of the accesses ($[\alpha_{\mathsf{C}}^j]_{w_{\mathsf{C}}^j}$ for any j) in ξ_{C} nor does it belong to the address set Σ^{GFS}.

4.1.7 Correlating paths to error nodes due to annotated instructions

Recall that the DYNAMO operates in two phases: in the first phase all paths in the original, non-annotated A are annotated and correlated, and in the second phase the extra error-going paths introduced due to annotation are correlated.

The correlation of these extra paths follow the same procedure, shown as call to the same phase-parametric *constructX()* procedure at line 6 in algorithm 1. As the required annotation is assumed to have been performed in the first phase, in the second call to *constructX()* with *phase* = `CORRELATE_NEW_ERROR_PATHS`, the procedure skips the call to the *asmAnnotOpts()* procedure (line 23 in algorithm 1). Similarly, as an optimization, the procedure avoids correlating an already correlated $\xi_{\ddot{A}}$ (line 17 in algorithm 1).

4.1.8 Soundness of DYNAMO algorithm

When all feasible simple paths between the cut points of Ä are exhaustively correlated (including the paths introduced due to annotation), the (CoverageÄ) requirement must be satisfied. The *checkCoverageReqs()* procedure further checks the satisfaction of (CoverageC) before returning `Success` (line 8 in algorithm 1).

Recall that the path enumeration ensures (MutexÄ), (MutexC) and (Well-formedness) are satisfied by construction, the *addingEdgeWillCreateEmptyCCycle()* procedure ensures that (Similar-Speed) holds, and the *haveSimilarStructure()* procedure ensures that the remaining three structural requirements are also satisfied. The four non-coverage semantic requirements are checked in *checkSemanticReqsExceptCoverage()*. DYNAMO is sound because it returns `Success` only if all the thirteen search-algorithm requirements are satisfied.

4.1.9 Counterexample Guided Best-First Search

The **chooseFrom** operator must attempt to maximize the chances of returning `Success`, even if only a fraction of the search space has been explored. DYNAMO uses the counterexamples generated when a proof obligation is falsified (e.g., during invariant inference) to guide the search towards the more promising options. A counterexample is a proxy for the machine states of C and Ä that may appear at a node n_X during the

lockstep execution encoded by X. Thus, if at any step during the construction of X, the execution of a counterexample for a candidate partial solution $(\ddot{\mathsf{A}}, \mathsf{X})$ results in the violation of a non-coverage requirement (e.g., (MemEq)) that candidate is discarded and the search backtracks (instead of failing). The counterexample-guided best-first search exhausts the entire search space of correlations before returning `Failure`.

Our (MemEq) requirement is generalization of the heap relation[9] pruning criteria of COUNTER [17]. In COUNTER, a candidate that does not have matching heap states at the end of counterexample execution is discarded. DYNAMO, through (MemEq), extends this to matching of memory states of stack-allocated locals as well. If the assembly procedure $\ddot{\mathsf{A}}$ does not have store-sinking like memory optimizations that cause divergence in the *non-stack* memory state of the two procedures, (MemEq) effectively prunes the search space without any loss in completeness guarantee.

The execution of counterexamples opportunistically weakens the node invariants in X. Like COUNTER [17], we use the number of live registers in $\ddot{\mathsf{A}}$ related through the current invariants in Φ_{X} to rank the enumerated partial candidate solutions to implement a best-first search. This ranking criteria is key to (relative) scaling of COUNTER and DYNAMO algorithms.

4.2 Invariant Inference

Our invariant inference procedure for identifying node invariants is a counterexample-guided algorithm, similar to COUNTER [17]. Candidate invariants of a partial product graph X are formed by conjoining the predicates drawn from the predicate grammar shown in fig. 4.2. A candidate invariant $\boxed{\phi}$ at node n_{X} is validated using Hoare triples of the form: $\{\phi_{n_{\mathsf{X}}^j}\}(\xi_{\ddot{\mathsf{A}}}^{j_k}; \xi_{\mathsf{C}}^{j_k})\{\boxed{\phi}\}$ for all $n_{\mathsf{X}}^j \in \mathcal{N}_{\mathsf{X}}$ and $(\xi_{\ddot{\mathsf{A}}}^{j_k}, \xi_{\mathsf{C}}^{j_k})$ such that $e_{\mathsf{X}}^{j_k} = (n_{\mathsf{X}}^j \xrightarrow{\xi_{\ddot{\mathsf{A}}}^{j_k}; \xi_{\mathsf{C}}^{j_k}} n_{\mathsf{X}}) \in \mathcal{E}_{\mathsf{X}}$.

The predicate grammar includes shape for inferring affine relations ($\boxed{\texttt{affine}}$) between *interesting* bitvector expressions of $\ddot{\mathsf{A}}$ and C — recall that the set of expressions participating in invariant inference are drawn from a set of *interesting* expressions in $\ddot{\mathsf{A}}$ and C (*inferInvariantsAndCounterexamples()* in algorithm 7). Apart from $\boxed{\texttt{affine}}$, we also infer inequality relations through $\boxed{\texttt{ineq}}$ and $\boxed{\texttt{ineqC}}$ over these interesting

[9] Unlike DYNAMO, COUNTER does not distinguish between different global variables in the program and treat all memory as a single "heap".

affine $\sum_i c_i v_i = c$ ineqC $v \odot c$ ineq $v_1 \odot v_2$

MemEq $M_{\mathsf{C}} =_{\Sigma^B_{\ddot{\mathsf{A}}} \setminus (\Sigma^{z_l|^v}_{\ddot{\mathsf{A}}})} M_{\ddot{\mathsf{A}}}$ AllocEq $\forall_{r \in B} : \Sigma^r_{\mathsf{C}} = \Sigma^r_{\ddot{\mathsf{A}}}$ WEq $\Omega_{\ddot{\mathsf{A}}} = \Omega_{\mathsf{C}}$

spOrd $\mathtt{sp}.p^{j_1}_{\ddot{\mathsf{A}}} \leq_u (\mathtt{sp}.p^{j_2}_{\ddot{\mathsf{A}}} - v^*)$ zEmpty $\{\Sigma^z_{\mathsf{C}}, \Sigma^{zl}_{\ddot{\mathsf{A}}}|^s, \Sigma^{zl}_{\ddot{\mathsf{A}}}|^v\}\ \{=, \neq\}\ \emptyset$

spzBd $\mathtt{em}.z \vee (\mathtt{sp}.p^j_{\ddot{\mathsf{A}}} \odot \{\mathtt{lb}.z, \mathtt{ub}.z\})$ spzBd' $\mathtt{em}.z \vee (\mathtt{sp}.p^j_{\ddot{\mathsf{A}}} \leq_u (\mathtt{lb}.z - v^*))$

Figure 4.2: Predicate grammar for constructing candidate invariants. v represents a bitvector variable (registers, stack slots, and ghost variables), c represents a bitvector constant. $\odot \in \{\leq_{s,u}, <_{s,u}, >_{s,u}, \geq_{s,u}\}$. v^* represents a bitvector value drawn from a restricted grammar (explained in text).

expressions. An efficient counterexample-guided algorithm for computing affine relations over bitvectors in an incremental setting such as ours is described in [42].

The candidate invariants include shapes MemEq and AllocEq for equality of memory and allocation state of common regions across Ä and C and the WEq shape for capturing the equality of outside world states. MemEq and WEq cater to the (MemEq) and (Equivalence) requirements respectively. Note that AllocEq follows from DYNAMO's $\mathcal{D}_{\mathsf{X}}$ construction (*addDetMappings()* in algorithm 1) and the execution semantics that observe each (de)allocation event. If the (de)allocations in both Ä and C were unobservable (and thus allowed to potentially differ indefinitely), invariant inference would become significantly harder, especially when equating memory regions of locals.

Recall that in our graph representations we save stackpointer value at the boundary of a stackpointer updating instruction at PC $p^j_{\ddot{\mathsf{A}}}$ in ghost variable $\mathtt{sp}.p^j_{\ddot{\mathsf{A}}}$ ((OP-ESP) in fig. 2.6). These ghost variables make it convenient to express relationships between stack-allocated local regions. To prove separation between different local variables (allocated by different stackpointer decrements), we require invariants that lower-bound the gap between two ghost variables, say $\mathtt{sp}.p^{j_1}_{\ddot{\mathsf{A}}}$ and $\mathtt{sp}.p^{j_2}_{\ddot{\mathsf{A}}}$, by some value v^* that depends on the allocation size operand of an $\mathtt{alloc}_s$ instruction (spOrd).

To capture the various relations between lower bounds, upper bounds, region sizes, and $\mathtt{sp}.p^j_{\ddot{\mathsf{A}}}$, the guessing grammar includes shapes spzBd and spzBd' that are of the form: "either a local variable region is empty or its bounds are related to $\mathtt{sp}.p^j_{\ddot{\mathsf{A}}}$ in these possible ways". The "a local variable region is empty" part caters to the case of a conditional `alloca()` where n_{X} may have some incoming paths where allocation did not happen. Similar to spOrd, the value v^* in spzBd' is derived from the allocation

$\boxed{\texttt{Empty}}\ \forall_{r\in G\cup F\cup Y\cup Z} : (\Sigma^{r}_{\ddot{A}} = \emptyset \Leftrightarrow \boxed{\texttt{em.}r})$

$\boxed{\texttt{gfySz}}\ \forall_{r\in G\cup F\cup Y\setminus\{\texttt{vrdc}\}} : (\boxed{\texttt{sz.}r} = \texttt{sz}(\texttt{T}(r)))$ $\qquad$ $\boxed{\texttt{vrdcSz}}\ (\boxed{\texttt{em.vrdc}} \Leftrightarrow \boxed{\texttt{sz.vrdc}} = 0)$

$\boxed{\texttt{gfyIntvl}}\ \underset{r\in G\cup F\cup Y}{\forall} : \left[\boxed{\texttt{em.}r} \vee \left(\begin{array}{l}(\boxed{\texttt{lb.}r} \leq_u \boxed{\texttt{ub.}r}) \wedge (\boxed{\texttt{lb.}r} + \boxed{\texttt{sz.}r} - 1_{\texttt{i32}} = \boxed{\texttt{ub.}r}) \\ \wedge\ ([\boxed{\texttt{lb.}r}, \boxed{\texttt{lb.}r}] = \Sigma^{r}_{\ddot{A}})\end{array}\right)\right]$

$\boxed{\texttt{zlIntvl}}\ \boxed{\texttt{em.}zl} \vee \left(\begin{array}{l}(\boxed{\texttt{lb.}zl} \leq_u \boxed{\texttt{ub.}zl}) \wedge (\boxed{\texttt{lb.}zl} + \boxed{\texttt{lstSz.}zl} - 1_{\texttt{i32}} = \boxed{\texttt{ub.}zl}) \\ \wedge\ ([\boxed{\texttt{lb.}zl}, \boxed{\texttt{lb.}zl}] = \Sigma^{zl}_{\ddot{A}})\end{array}\right)$

$\boxed{\texttt{zaBd}}\ \boxed{\texttt{em.}za} \vee \left(\begin{array}{l}(\boxed{\texttt{lb.}za} \leq_u \boxed{\texttt{ub.}za}) \wedge (\boxed{\texttt{lb.}za} + \boxed{\texttt{lstSz.}za} - 1_{\texttt{i32}} \leq_u \boxed{\texttt{ub.}za}) \\ \wedge\ (\boxed{\texttt{lb.}za} = \texttt{lb}(\Sigma^{za}_{\ddot{A}}) \wedge \boxed{\texttt{ub.}za} = \texttt{ub}(\Sigma^{za}_{\ddot{A}}))\end{array}\right)$

$\boxed{\texttt{StkBd}}\ \Sigma^{\{stk\}\cup Y}_{\ddot{A}} \cup (\Sigma^{Z}_{\ddot{A}} \setminus (\Sigma^{Z_l}_{\ddot{A}}|^{v})) = [\texttt{esp}, \boxed{\texttt{stk}_e}]$ $\qquad$ $\boxed{cs\texttt{Bd}}\ \Sigma^{\{cs,cl\}}_{\ddot{A}} = [\boxed{\texttt{stk}_e} + 1, \boxed{\texttt{cs}_e}]$

$\boxed{\texttt{NoOverlapC}}\ \neg\texttt{ov}(\Sigma^{hp}_{\ddot{A}}, \Sigma^{cl}_{\ddot{A}}, \Sigma^{\texttt{vrdc}}_{\ddot{A}}, \ldots, i^{g}_{\ddot{A}}, \ldots, i^{y}_{\ddot{A}}, \ldots, \Sigma^{z}_{\ddot{A}})$

$\boxed{\texttt{NoOverlapA}}\ \neg\texttt{ov}(\Sigma^{\{hp,cl\}\cup G\cup Y}_{\ddot{A}}, \ldots, \Sigma^{z}_{\ddot{A}}|^{s}, \ldots, i^{f}_{\ddot{A}}, \ldots, \Sigma^{stk}_{\ddot{A}}, \Sigma^{cs}_{\ddot{A}})$

$\boxed{\texttt{ROMA}}\ \forall_{r\in F_r} : (M_{\ddot{A}} =_{i^{r}_{\ddot{A}}} \texttt{ROM}^{r}_{\ddot{A}}(i^{r}_{\ddot{A}}))$ $\qquad$ $\boxed{\texttt{ROMC}}\ \forall_{r\in G_r} : (M_{\texttt{C}} =_{i^{r}_{\texttt{C}}} \texttt{ROM}^{r}_{\texttt{C}}(i^{r}_{\texttt{C}}))$

Figure 4.3: Global invariants that hold at each non-entry, error-free node $n_{\mathsf{X}} \in \mathcal{N}^{\mathcal{U}\mathcal{W}}_{\mathsf{X}}$.

size operand of an $\texttt{alloc}_s$ instruction. $\boxed{\texttt{zEmpty}}$ tracks the emptiness of the address set of a local region z in C and the address sets $\Sigma^{zl}_{\ddot{A}}|^{s}$ and $\Sigma^{zl}_{\ddot{A}}|^{v}$ for $zl \in Z_l$ in Ä. We need to track the emptiness of the latter address sets in Ä to prove the infeasibility of the $\mathcal{U}$-going paths due to (DeallocV) and (DeallocS') (sections 2.6.1 and 2.6.2).

Together, the predicate shapes $\boxed{\texttt{spOrd}}$, $\boxed{\texttt{spzBd}}$, $\boxed{\texttt{spzBd'}}$, $\boxed{\texttt{zEmpty}}$ (and $\boxed{\texttt{affine}}$ and $\boxed{\texttt{ineq}}$ for relations between $\boxed{\texttt{sp.}p^{j}_{\ddot{A}}}$), enable disambiguation between stack writes involving spilled pseudo-registers and stack-allocated locals.

4.2.1 Global Invariants

Recall that due to our execution semantics certain *global invariants* hold by construction at each non-entry, error-free node $n_{\mathsf{X}} \in \mathcal{N}^{\mathcal{U}\mathcal{W}}_{\mathsf{X}}$ in X (section 3.3.3). We add these global invariants to the set of node invariants $\phi_{n_{\mathsf{X}}}$ at a node n_{X}, along with the inferred invariants described in previous section. We list the predicate expressions for these global invariants in fig. 4.3 and discuss each below.

- $\boxed{\texttt{Empty}}$ asserts that the ghost variable $\boxed{\texttt{em.}r}$ (for tracking the emptiness of region) for $r \in G \cup F \cup Y \cup Z$ tracks the emptiness of its address set.

- gfySz equates the ghost variable sz.r (for tracking size of region $r \in G \cup F \cup (Y \setminus \{\texttt{vrdc}\})$) to the size of the variable named r (recall that a region identifier is also the name of the variable)[10].

  vrdcSz encodes that the sz.vrdc is zero iff em.vrdc holds for the variadic parameter $\texttt{vrdc} \in Y$.

- gfyIntvl encodes that the address set of region $r \in G \cup F \cup Y$ is an interval of size sz.r bounded by ghost variables lb.r and ub.r. Note that only the vrdc region can potentially be empty such that em.r holds for $r = \texttt{vrdc}$.

- zlIntvl captures the property that a local variable region $zl \in Z_l$, if non-empty, must be an interval of size lstSz.zl bounded by ghost variables lb.zl and ub.zl.

  zlIntvl is encoding the same "r is an interval" property presented in gfyIntvl for $r \in Z_l$.

- zaBd captures a weaker property (than zlIntvl) for a local region $za \in Z_a$ (recall that za is a local allocated using alloca()): if non-empty, this region must be bounded by its ghost variables (lb.zl and ub.zl) and must be at least lstSz.za large.

  The difference from zlIntvl is that the region za need not be an interval. This matches the expectation from intuition that multiple stack decrements in Ä corresponding to executions of an alloca() in C need not be contiguous.

- StkBd encodes the invariant that the interval [esp, $\texttt{stk}_e$] represents the union of the address sets of stk, regions in Y, and stack-allocated local regions ($\Sigma^{Z}_{\ddot{A}} \setminus (\Sigma^{Z_l}_{\ddot{A}}|^{v})$).

- csBd is similarly shaped as StkBd and encodes that the interval [$\texttt{stk}_e + 1$, $\texttt{cs}_e$] represents the union of the address sets of regions cs and cl.

- NoOverlapC encodes the disjointedness of all regions $r \in B$ (recall that B denote the common regions present in both C and A).

- NoOverlapA encodes the disjointedness of all regions in Ä except virtually-allocated regions. Note that NoOverlapA does not encode disjointedness of regions $\{hp, cl\} \cup G \cup Y$ — we rely on AllocEq and NoOverlapC for this.

[10] Note that $\texttt{sz}(\texttt{T}(r)) > 0$ for all r because a variable can never have zero size in the C programming language.

- $\boxed{\boxed{\texttt{ROM}\mathsf{C}}}$ and $\boxed{\boxed{\texttt{ROM}\mathsf{A}}}$ encode the preservation of memory contents of read-only regions in C and Ä.

Let $\boxed{\boxed{\phi_{\mathsf{X}}}}$ represent the conjunction of the global invariants listed in fig. 4.3. Let $\boxed{\phi_{n_{\mathsf{X}}}}$ represent the conjunction of the inductively provable invariants drawn from grammar shown in fig. 4.2. The invariants $\phi_{n_{\mathsf{X}}}$ at an error-free node $n_{\mathsf{X}} \in \mathcal{N}_{\mathsf{X}}$ are obtained through conjunction of $\boxed{\boxed{\phi_{\mathsf{X}}}}$ and $\boxed{\phi_{n_{\mathsf{X}}}}$, i.e., $\phi_{n_{\mathsf{X}}} = \boxed{\boxed{\phi_{\mathsf{X}}}} \wedge \boxed{\phi_{n_{\mathsf{X}}}}$.

4.3 Running Example of the Algorithm

Figure 4.5 shows the abbreviated Transition Graphs of the unoptimized IR and fully annotated assembly procedure of the `fib` procedure from fig. 2.1, reproduced for convenience in fig. 4.4. A node in figs. 4.5a and 4.5b is identified by its PC and we use a subscript notation for the PCs due to constituent edges of an unoptimized IR/assembly instruction, e.g., the procedure call at `I13` corresponds to graph edges $\texttt{I13} \rightarrow \texttt{I13}_1$, $\texttt{I13}_1 \rightarrow \texttt{I13}_2$, $\texttt{I13}_2 \rightarrow \texttt{I13}_3$, and $\texttt{I13}_3 \rightarrow \texttt{I14}$. The abbreviated graphs retain only a subset of the nodes — roughly, we retain nodes corresponding to the PCs of the statements in the listings in fig. 4.4. We omit almost all edge labels except for the `rd` and `wr` instructions. We show transitions to error nodes in an abbreviated edge using edge labels, e.g., in fig. 4.5a, the transition to error node $\mathscr{W}_{\mathsf{C}}$ due to (Entry$_{\mathsf{C}}$) is shown as the label $\rightarrow\mathscr{W}_{\mathsf{C}}$ on the edge $\texttt{I0}_1 \rightarrow \texttt{I0}_2$. For an abbreviated edge due to (de)`alloc` and (de)$\texttt{alloc}_{s,v}$ instruction, we show the constituent (still abbreviated) edges using an exploded view, e.g., the abbreviated edge $\texttt{I1} \rightarrow \texttt{I2}$ corresponding to `alloc` instruction at `I1` in fig. 4.4 is exploded into (still abbreviated) edges $\texttt{I1} \rightarrow \texttt{I1}_1$, $\texttt{I1}_1 \rightarrow \texttt{I1}_2$, $\texttt{I1}_2 \rightarrow \texttt{I1}_3$, $\texttt{I1}_3 \rightarrow \texttt{I1}_4$, and $\texttt{I1}_4 \rightarrow \texttt{I2}$ in fig. 4.5a. In the exploded view for edges due to `alloc` at `I9` and `I10` in fig. 4.4b and due to $\texttt{alloc}_s$ at $\texttt{A17}^1$ and $\texttt{A17}^2$, we omit the nodes as well and just retain the labels. Because the graphs are abbreviated we will refer both the unoptimized IR and assembly procedures in fig. 4.4 and the graphs in fig. 4.5 in our discussion below.

We show the execution of the Dynamo algorithm on the two graphs in fig. 4.5 at unroll-factor $\mu = 2$. Note that the Ä graph in fig. 4.5b shows the fully annotated graph. During our exposition, we will treat the graph as if it were not annotated till the point it becomes annotated in our discussion i.e., we will ignore the annotated edges until they are *inserted* by our execution of the algorithm. The algorithm begins with the points-to analysis on C that over-approximates the β and β_M sets at each PC of C.

```
    int printf(const char*, ...);

C0: int fib(int n, int m) {
C1:   int v[n+2];
C2:   v[0]=0; v[1]=1;
C3:   for(int i=2; i<=m; ++i)
C4:     v[i]=v[i-1]+v[i-2];
C5:   printf("fib(%d) = %d", m, v[m]);
C6:   return v[m];
C7: }
```

(a) C program with VLA.

```
I0: int fib(int* n, int* m):
I1:   i=alloc 1, int, 4;
I2:   v=alloc *n+2, int, 4;
I3:   v[0]=0; v[1]=1;
I4:   *i=2;
I5:   if(*i >_s *m) goto I9;
I6:     v[*i]=v[*i-1]+v[*i-2];
I7:     ++(*i);
I8:     goto I5;
I9:   p_I9=alloc 1, char*, 4;
I10:  p_I10=alloc 1, struct{int;int;}, 4;
I11:  *p_I9=__S__;
I12:  *p_I10=*m; *(p_I10 + 4)=v[*m];
I13:  t=call int printf(p_I9, p_I10);
I14:  dealloc I10;
I15:  dealloc I9;
I16:  r=v[*m];
I17:  dealloc I2;
I18:  dealloc I1;
I19:  ret r;
```

(b) (Abstracted) Unoptimized IR.

```
A0: fib:
A1:   push ebp; ebp = esp;
A2:   push {edi, esi, ebx};
A3:   esp -= 12;
A3¹:  v_I1 = alloc_v 4, 4, I1;
A4:   eax = mem4[ebp+8]; ebx = mem4[ebp+12];
A5:   esp -= 0xFFFFFFF0 & (4*(eax+2)+15));
A5¹:  alloc_s esp, 4*(eax+2), 4, I2;
A6:   esi = ((esp+3)>>2)*4;
A7:   mem4[esi] = 0; mem4[esi+4] = 1;
A8:   if(ebx ≤_s 1) jmp A15;
A9:   edi = 0; edx = 1; eax = 2;
A10:    ecx = edx+edi;
A11:    edi = edx; edx = ecx;
A12:    mem4[esi+4*eax] = ecx;
A13:    ++eax;
A14:    if(eax ≤_s ebx) jmp A10;
A15:  edi = mem4[esi+4*ebx];
A16:  esp -= 4;
A17:  push {edi, ebx, __S__};
A17¹: alloc_s esp,  4, 4, I9;
A17²: alloc_s esp+4, 8, 4, I10;
A18:  call int printf
          (<char*> esp,
           <struct{int; int;}> esp+4)
          {hp, cl, I9, I10};
A18¹: dealloc_s I10;
A18²: dealloc_s I9;
A19:  eax = edi;
A19¹: dealloc_s I2;
A19²: dealloc_v I1;
A20:  esp = ebp-12;
A21:  pop {ebx, esi, edi, ebp};
A22:  ret;
```

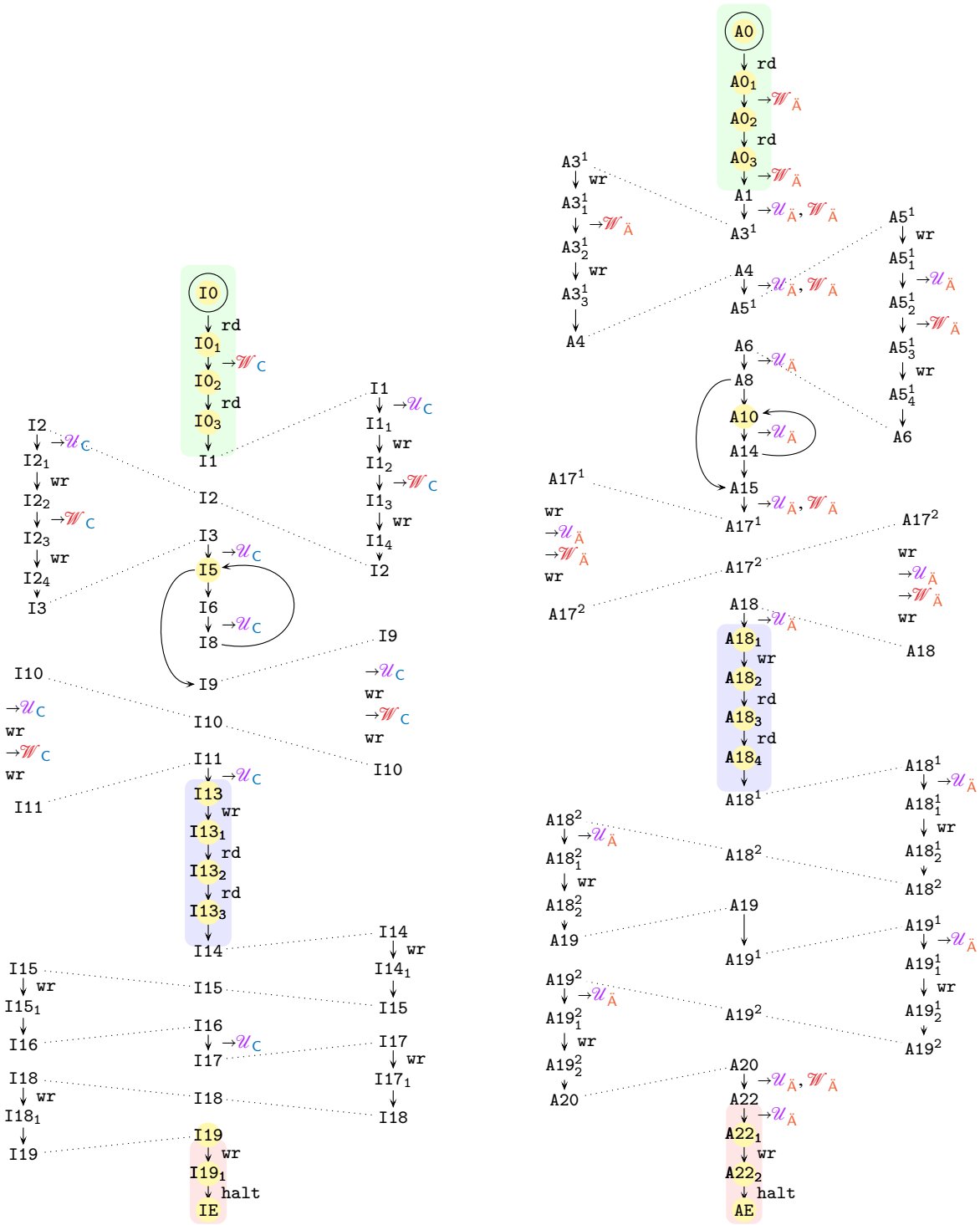
(c) (Abstracted) 32-bit x86 assembly code.

Figure 4.4: Reproduced C program and its unoptimized IR and assembly from fig. 2.1.

For example, at node `I3` (or PC `I3` in fig. 4.4b), $\beta(\texttt{n}) = \{\texttt{n}\}$, $\beta(\texttt{m}) = \{\texttt{m}\}$, $\beta(\texttt{i}) = \{\texttt{I1}\}$, $\beta(\texttt{v}) = \{\texttt{I2}\}$ and $\beta_M(\texttt{n}) = \beta_M(\texttt{m}) = \{hp, cl\}$, $\beta_M(\texttt{I1}) = \beta_M(\texttt{I2}) = \emptyset$, where `n` and `m` refer to both the state variables and regions $\texttt{n}, \texttt{m} \in Y$ and $\texttt{I1}, \texttt{I2} \in Z_l$.

Enumerating A paths

After the points-to analysis, DYNAMO begins its first phase with the call to the *constructX()* procedure with parameter *phase* = `CORRELATE_AND_ANNOTATE`. *constructX()*



(a) Abbreviated Transition Graph for the unoptimized IR of the `fib` procedure from fig. 4.4b.

(b) Abbreviated Transition Graph for the compiler-generated assembly of the `fib` procedure from fig. 4.4c.

Figure 4.5: Abbreviated Transition Graphs for the unoptimized IR and assembly of the `fib` procedure from fig. 4.4.

identifies the set of cut-points in fig. 4.5b. A subset of cut-points nodes in figs. 4.5a and 4.5b are highlighted with background[11]. The cut-points nodes identified by *getCutPointsInRPO()* are: $\texttt{A0}$, $\texttt{A0}_1$, $\texttt{A0}_2$, $\texttt{A0}_3$, $\texttt{A10}$, $\texttt{A18}_1$, $\texttt{A18}_2$, $\texttt{A18}_3$, $\texttt{A18}_4$, $\texttt{A22}_1$, $\texttt{A22}_2$, and $\texttt{AE}$[12]. The correlation of (abbreviated) edges, $\texttt{A0} \rightarrow \texttt{A0}_1$, $\texttt{A0}_1 \rightarrow \texttt{A0}_2$, and $\texttt{A0}_2 \rightarrow \texttt{A0}_3$, due to ($\textsc{Entry}_{\ddot{\text{A}}}$), is rather straightforward and we will instead consider the case when the cut-point node is $\texttt{A0}_3$, i.e., when $\mathcal{N}_{\text{X}} = \{(\texttt{A0}, \texttt{I0}), (\texttt{A0}_1, \texttt{I0}_1), (\texttt{A0}_2, \texttt{I0}_2), (\texttt{A0}_3, \texttt{I0}_3)\}$ and $\mathcal{E}_{\text{X}} = \{((\texttt{A0}, \texttt{I0}) \xrightarrow{(\texttt{A0} \twoheadrightarrow \texttt{A0}_1);(\texttt{I0} \twoheadrightarrow \texttt{I0}_1)} (\texttt{A0}_1, \texttt{I0}_1)), ((\texttt{A0}_1, \texttt{I0}_1) \xrightarrow{(\texttt{A0}_1 \twoheadrightarrow \texttt{A0}_2);(\texttt{I0}_1 \twoheadrightarrow \texttt{I0}_2)} (\texttt{A0}_2, \texttt{I0}_2)), ((\texttt{A0}_2, \texttt{I0}_2) \xrightarrow{(\texttt{A0}_2 \twoheadrightarrow \texttt{A0}_3);(\texttt{I0}_2 \twoheadrightarrow \texttt{I0}_3)} (\texttt{A0}_3, \texttt{I0}_3))\}$.

The (ordered) cut-point successors of $\texttt{A0}_3$, as identified by *cutPointSuccessorsRPO()*, are: $\texttt{A10}$, $\texttt{A18}_1$, $\mathscr{U}_{\ddot{\text{A}}}$, and $\mathscr{W}_{\ddot{\text{A}}}$[13]. The cut-point $\texttt{A10}$, due to the cycle $\texttt{A10} \rightarrow \texttt{A14} \rightarrow \texttt{A10}$, is considered first as it precedes others in the order. The only simple path between $\texttt{A0}_3$ and $\texttt{A10}$ is the path $(\texttt{A0}_3 \twoheadrightarrow \texttt{A10}) = \texttt{A0}_3 \rightarrow \texttt{A1} \rightarrow \texttt{A4} \rightarrow \texttt{A6} \rightarrow \texttt{A8} \rightarrow \texttt{A10}$. As $(\texttt{A0}_3 \twoheadrightarrow \texttt{A10})$ is not demonstrably infeasible, it is considered for the next step where its C pathset options are enumerated.

Correlating C paths

The *correlatedPathsInCOpts()* procedure (algorithm 3) is called with arguments $\xi_{\ddot{\text{A}}} = (\texttt{A0}_3 \twoheadrightarrow \texttt{A10})$ and $\mu = 2$. The unique X node for $\texttt{A0}_3$ in $\mathcal{N}_{\text{X}}$ is $(\texttt{A0}_3, \texttt{I0}_3)$. As the sink $\texttt{A10}$ of $(\texttt{A0}_3 \twoheadrightarrow \texttt{A10})$ is an error-free node, the third case in *correlatedPathsInCOpts()* is triggered (line 21 in algorithm 3):

1. First, the set of *all* error-free cut-points reachable from $\texttt{I0}_3$ are identified (*getAllNonErrorReachableCutPoints()*). In fig. 4.5a, these are: $\texttt{I1}_1$, $\texttt{I1}_2$, $\texttt{I1}_3$, $\texttt{I1}_4$, $\texttt{I2}_1$, $\texttt{I2}_2$, $\texttt{I2}_3$, $\texttt{I2}_4$, $\texttt{I5}$, $\texttt{I13}$, $\texttt{I13}_1$, $\texttt{I13}_2$, $\texttt{I13}_3$, $\texttt{I19}$, $\texttt{I19}_1$, $\texttt{IE}$, ... (omitting the ones due to `(de)alloc` instructions at $\texttt{I9}$, $\texttt{I10}$, $\texttt{I14}$, $\texttt{I15}$, $\texttt{I17}$, and $\texttt{I18}$).

2. For each cut-point q^t_{C} enumerated in previous step, the set of pathsets from $\texttt{I0}_3$ to q^t_{C} with unrolling up to $\mu = 2$ are enumerated using `enumPathsetsTillUnroll()`. For $q^t_{\mathsf{C}} = \texttt{I5}$, the possible pathsets returned by `enumPathsetsTillUnroll()` are the singleton sets $\{\texttt{I0}_3 \rightarrow \texttt{I1} \rightarrow \texttt{I2} \rightarrow \texttt{I3} \rightarrow \texttt{I5}\}$ and $\{\texttt{I0}_3 \rightarrow \texttt{I1} \rightarrow \texttt{I2} \rightarrow \texttt{I3} \rightarrow \texttt{I5} \rightarrow \texttt{I6} \rightarrow \texttt{I8} \rightarrow \texttt{I5}\}$[14] — notice that the latter has two unrolling of node $\texttt{I5}$.

[11] All cut-point nodes except those due to `(de)alloc` and $\texttt{(de)alloc}_{\text{s,v}}$ instructions are highlighted in figs. 4.5a and 4.5b.

[12] As established earlier, we will ignore the inserted edges at this point.

[13] Recall that the nodes between $\texttt{A3}^1$ and $\texttt{A4}$ and between $\texttt{A5}^1$ and $\texttt{A6}$ will not be considered because they are inserted later due to annotation.

[14] Here and henceforth, we will omit the edges due to `(de)alloc` for brevity.

3. For each pathset $\langle\xi\rangle^t_{\mathsf{C}}$ returned by `enumPathsetsTillUnroll()`, set of its all $\mathcal{U}$-maximal subsets are added as candidate options. A candidate pathset added due to the singleton pathset $\langle\xi\rangle^t_{\mathsf{C}} = \{\mathtt{I0_3} \rightarrow \mathtt{I1} \rightarrow \mathtt{I2} \rightarrow \mathtt{I3} \rightarrow \mathtt{I5}\}$ includes the paths: $\mathtt{I0_3} \rightarrow \mathtt{I1} \rightarrow \mathcal{U}_{\mathsf{C}}$, $\mathtt{I0_3} \rightarrow \mathtt{I1} \rightarrow \mathtt{I2} \rightarrow \mathcal{U}_{\mathsf{C}}$, $\mathtt{I0_3} \rightarrow \mathtt{I1} \rightarrow \mathtt{I2} \rightarrow \mathtt{I3} \rightarrow \mathcal{U}_{\mathsf{C}}$, and $\mathtt{I0_3} \rightarrow \mathtt{I1} \rightarrow \mathtt{I2} \rightarrow \mathtt{I3} \rightarrow \mathtt{I5}$, where the $\mathcal{U}_{\mathsf{C}}$ going paths due to the statement `I3` in fig. 4.4b have been appreviated to $\mathtt{I0_3} \rightarrow \mathtt{I1} \rightarrow \mathtt{I2} \rightarrow \mathtt{I3} \rightarrow \mathcal{U}_{\mathsf{C}}$ for brevity.

In *constructX()*, let the **chooseFrom** operator choose the pathset containing the paths $\{\mathtt{I0_3} \rightarrow \mathtt{I1} \rightarrow \mathcal{U}_{\mathsf{C}}, \mathtt{I0_3} \rightarrow \mathtt{I1} \rightarrow \mathtt{I2} \rightarrow \mathcal{U}_{\mathsf{C}}, \mathtt{I0_3} \rightarrow \mathtt{I1} \rightarrow \mathtt{I2} \rightarrow \mathtt{I3} \rightarrow \mathcal{U}_{\mathsf{C}}, \mathtt{I0_3} \rightarrow \mathtt{I1} \rightarrow \mathtt{I2} \rightarrow \mathtt{I3} \rightarrow \mathtt{I5}\}$ as the candidate pathset $\langle\xi\rangle_{\mathsf{C}}$ for correlation with $\xi_{\ddot{\mathsf{A}}} = (\mathtt{A0_3} \twoheadrightarrow \mathtt{A10})$. Each path $\xi_{\mathsf{C}} \in \langle\xi\rangle_{\mathsf{C}}$ is considered separately; we demonstrate the correlation of $\xi_{\mathsf{C}} = \mathtt{I0_3} \rightarrow \mathtt{I1} \rightarrow \mathtt{I2} \rightarrow \mathtt{I3} \rightarrow \mathtt{I5}$. As *phase* = `CORRELATE_AND_ANNOTATE`, *constructX()* calls *asmAnnotOpts()* for potentially annotating $\xi_{\ddot{\mathsf{A}}} = (\mathtt{A0_3} \twoheadrightarrow \mathtt{A10})$ (line 23 in algorithm 1).

Identifying A annotation

The *asmAnnotOpts()* procedure begins by collecting the sequence of (edges due to) `alloc`, `dealloc`, and `call` instructions in $\xi_{\mathsf{C}} = \mathtt{I0_3} \rightarrow \mathtt{I1} \rightarrow \mathtt{I2} \rightarrow \mathtt{I3} \rightarrow \mathtt{I5}$. The (abbreviated) edges $\mathtt{I1} \rightarrow \mathtt{I2}$ and $\mathtt{I2} \rightarrow \mathtt{I3}$ correspond to the `alloc` instructions at `I1` and `I2` respectively in fig. 4.4b. For each `alloc` instruction, *asmAnnotOpts()* enumerates $\mathtt{alloc}_{s,v}$ annotation options in the path $\xi_{\ddot{\mathsf{A}}} = \mathtt{A0_3} \rightarrow \mathtt{A1} \rightarrow \mathtt{A4} \rightarrow \mathtt{A6} \rightarrow \mathtt{A8} \rightarrow \mathtt{A10}$.

In the blackbox mode, the enumeration is dictated by the restrictions described in section 4.1.3. The first restriction confines the position of an $\mathtt{alloc}_{s,v}$ annotation to just after a stackpointer updating instruction — in our case (fig. 4.4c), the candidates are the PCs just after the statements `A1`, `A2`, `A3`, and `A5`. The second restriction confines the options for start address in an $\mathtt{alloc}_s$ annotation to the stackpointer value `esp` at the respective position — for each of the positions enumerated above, the corresponding stackpointer values are represented by the ghost variables `sp.A1`, `sp.A2`, `sp.A3`, and `sp.A5`[15]. The other parameters for the $\mathtt{alloc}_{s,v}$ instructions are inferred from the corresponding `alloc` instruction, e.g. for the `alloc` instruction at `I2` (fig. 4.4b), the size of allocation is inferred to be `4*(eax+2)` (inferred using the relation $\mathtt{eax} = (*\mathtt{n}) = \mathtt{sel}_4(\mathtt{mem}, \mathtt{ebp} + 8)$[16]), the required alignment to be 4 (from alignment of

[15]Recall that the translation rule (OP-ESP) for a stackpointer updating instruction at PC $p^j_{\ddot{\mathsf{A}}}$ stores the stackpointer value at the end of the update to a ghost variable $\mathtt{sp}.p^j_{\ddot{\mathsf{A}}}$ identified uniquely by $p^j_{\ddot{\mathsf{A}}}$.

[16]A more general method is to build a $\mathcal{D}_{\mathsf{X}}$-like map for enabling propagation of information from C to Ä. Such a construction is enabled due to lockstep correlation of `alloc` and $\mathtt{alloc}_{s,v}$ in X.

int), and the region identifier to be I2. Thus, the annotation '$\texttt{alloc}_v$ 4, 4, I1' after A3 for alloc at I1 and '$\texttt{alloc}_s$ sp.A5, 4 ∗ (eax + 2), 4, I2' after A5 for alloc at I2 will be enumerated by the blackbox enumeration procedure (genAnnotOptsForAlloc() in algorithm 4). The *asmAnnotOpts()* procedure considers all combinations of individual annotation options, one of which will include the combined annotation '$\texttt{alloc}_v$ 4, 4, I1' after A3 and '$\texttt{alloc}_s$ sp.A5, 4 ∗ (eax + 2), 4, I2' after A5 as an annotation candidate. The application of the above annotation candidate inserts the edges $\texttt{A3}^1 \rightarrow \texttt{A3}^1_1 \rightarrow \texttt{A3}^1_2 \rightarrow \texttt{A3}^1_3 \rightarrow \texttt{A4}$ and $\texttt{A5}^1 \rightarrow \texttt{A5}^1_1 \rightarrow \texttt{A5}^1_2 \rightarrow \texttt{A5}^1_3 \rightarrow \texttt{A5}^1_4 \rightarrow \texttt{A6}$ to $\ddot{\text{A}}$ as shown in fig. 4.5b. Note that the updated $\xi_{\ddot{\text{A}}}$, $\xi^*_{\ddot{\text{A}}} = \texttt{A0}_3 \rightarrow \texttt{A1} \rightarrow \texttt{A3}^1 \rightarrow \texttt{A4} \rightarrow \texttt{A5}^1 \rightarrow \texttt{A6} \rightarrow \texttt{A8} \rightarrow \texttt{A10}$[17], obtained after including the annotated edges, does not include the $\mathscr{U}_{\ddot{\text{A}}}$ and $\mathscr{W}_{\ddot{\text{A}}}$ going paths added due to annotation — these paths are correlated in the second phase (when *phase* = CORRELATE_NEW_ERROR_PATHS).

Transforming identified paths to correlation paths

After annotation, *constructX()* breaks $\xi_{\ddot{\text{A}}}$ and ξ_{C} into constituent paths using *breakIntoSingleIOPaths()*, so that each I/O path (section 3.1) appears by itself and not as sub-path of some larger path. The path sequences obtained after both calls to *breakIntoSingleIOPaths()* are $\vec{\xi}'_{\ddot{\text{A}}} = ((\texttt{A0}_3 \twoheadrightarrow \texttt{A3}^1), (\texttt{A3}^1 \twoheadrightarrow \texttt{A3}^1_1), (\texttt{A3}^1_1 \twoheadrightarrow \texttt{A3}^1_2), (\texttt{A3}^1_2 \twoheadrightarrow \texttt{A3}^1_3), (\texttt{A3}^1_3 \twoheadrightarrow \texttt{A5}^1), (\texttt{A5}^1 \twoheadrightarrow \texttt{A5}^1_1), (\texttt{A5}^1_1 \twoheadrightarrow \texttt{A5}^1_3), (\texttt{A5}^1_3 \twoheadrightarrow \texttt{A5}^1_4), (\texttt{A5}^1_4 \twoheadrightarrow \texttt{A10}))$ and $\vec{\xi}'_{\text{C}} = ((\texttt{I0}_3 \twoheadrightarrow \texttt{I1}_1), (\texttt{I1}_1 \twoheadrightarrow \texttt{I1}_2), (\texttt{I1}_2 \twoheadrightarrow \texttt{I1}_3), (\texttt{I1}_3 \twoheadrightarrow \texttt{I1}_4), (\texttt{I1}_4 \twoheadrightarrow \texttt{I2}_1), (\texttt{I2}_1 \twoheadrightarrow \texttt{I2}_2), (\texttt{I2}_2 \twoheadrightarrow \texttt{I2}_3), (\texttt{I2}_3 \twoheadrightarrow \texttt{I2}_4), (\texttt{I2}_4 \twoheadrightarrow \texttt{I5}))$.

As neither ξ_{C} nor $\xi_{\ddot{\text{A}}}$ ends at an error node, the call to *trimToMatchPathToErrorNode()* (algorithm 5) will not perform any trimming. For demonstrating *trimToMatchPathToErrorNode()*, we consider the case when for the same $\xi_{\ddot{\text{A}}}$, $\xi_{\text{C}} = \texttt{I0}_3 \rightarrow \texttt{I1} \rightarrow \texttt{I2} \rightarrow \mathscr{U}_{\text{C}}$ is correlated so that $\vec{\xi}'_{\text{C}}$ (obtained after *breakIntoSingleIOPaths()*) comes out to be $((\texttt{I0}_3 \twoheadrightarrow \texttt{I1}_1), (\texttt{I1}_1 \twoheadrightarrow \texttt{I1}_2), (\texttt{I1}_2 \twoheadrightarrow \texttt{I1}_3), (\texttt{I1}_3 \twoheadrightarrow \texttt{I1}_4), (\texttt{I1}_4 \twoheadrightarrow \mathscr{U}_{\text{C}}))$. In this case, $\vec{\xi}'_{\text{C}}$ ends at error node $\mathscr{U}_{\text{C}}$ and the two paths sequences have difference lengths ($|\vec{\xi}'_{\ddot{\text{A}}}| = 9$ and $|\vec{\xi}'_{\text{C}}| = 5$). *trimToMatchPathToErrorNode()* trims $\vec{\xi}'_{\ddot{\text{A}}}$ to match the length of the error going path sequence $\vec{\xi}'_{\text{C}}$, i.e., we obtain $\vec{\xi}^*_{\ddot{\text{A}}} = ((\texttt{A0}_3 \twoheadrightarrow \texttt{A3}^1), (\texttt{A3}^1 \twoheadrightarrow \texttt{A3}^1_1), (\texttt{A3}^1_1 \twoheadrightarrow \texttt{A3}^1_2), (\texttt{A3}^1_2 \twoheadrightarrow \texttt{A3}^1_3), (\texttt{A3}^1_3 \twoheadrightarrow \texttt{A5}^1))$ (dropping the last four path components from $\vec{\xi}'_{\ddot{\text{A}}}$) and $\vec{\xi}^*_{\text{C}}$ remains identical to $\vec{\xi}'_{\text{C}}$. Effectively, with *trimToMatchPathToErrorNode()*, the lockstep execution of either of the error-free path in an X edge is restricted to the point where the other path encounters error.

[17] Omitting the constituent edges between $\texttt{A3}^1$ and A4, and $\texttt{A5}^1$ and A6.

Validating the structure of the paths

constructX() validates the structure of $\vec{\xi}^{*}_{\ddot{A}}$ and $\vec{\xi}^{*}_{C}$ using *haveSimilarStructure()*. The *haveSimilarStructure()* predicate verifies that both path sequences $\vec{\xi}^{*}_{\ddot{A}}$ and $\vec{\xi}^{*}_{C}$ have identical number of paths and each coupled path-pair satisfies the (SingleIO), (Safety), and (Termination) structural requirements. With $\vec{\xi}^{*}_{\ddot{A}} = ((\texttt{A0}_3 \twoheadrightarrow \texttt{A3}^1), (\texttt{A3}^1 \twoheadrightarrow \texttt{A3}^1_1), (\texttt{A3}^1_1 \twoheadrightarrow \texttt{A3}^1_2), (\texttt{A3}^1_2 \twoheadrightarrow \texttt{A3}^1_3), (\texttt{A3}^1_3 \twoheadrightarrow \texttt{A5}^1), (\texttt{A5}^1 \twoheadrightarrow \texttt{A5}^1_1), (\texttt{A5}^1_1 \twoheadrightarrow \texttt{A5}^1_3), (\texttt{A5}^1_3 \twoheadrightarrow \texttt{A5}^1_4), (\texttt{A5}^1_4 \twoheadrightarrow \texttt{A10}))$ and $\vec{\xi}^{*}_{C} = ((\texttt{I0}_3 \twoheadrightarrow \texttt{I1}_1), (\texttt{I1}_1 \twoheadrightarrow \texttt{I1}_2), (\texttt{I1}_2 \twoheadrightarrow \texttt{I1}_3), (\texttt{I1}_3 \twoheadrightarrow \texttt{I1}_4), (\texttt{I1}_4 \twoheadrightarrow \texttt{I2}_1), (\texttt{I2}_1 \twoheadrightarrow \texttt{I2}_2), (\texttt{I2}_2 \twoheadrightarrow \texttt{}_3), (\texttt{I2}_3 \twoheadrightarrow \texttt{I2}_4), (\texttt{I2}_4 \twoheadrightarrow \texttt{I5}))$, the (Safety) and (Termination) requirements hold trivially and it is easy to see that the I/O paths of same kind are coupled together for the (SingleIO) requirement, e.g., the I/O path $(\texttt{A3}^1 \twoheadrightarrow \texttt{A3}^1_1)$ that corresponds to the `wr(allocBegin())` instruction due to $\texttt{alloc}_v$ (see (AllocV)) is paired with the path $(\texttt{I1}_1 \twoheadrightarrow \texttt{I1}_2)$ that also corresponds to a `wr(allocBegin())` instruction due to `alloc` (see (Alloc)).

Adding X edges and updating $\mathcal{D}_X$

After validating the structural requirements, the identically-sized path sequences $\vec{\xi}^{*}_{\ddot{A}}$ and $\vec{\xi}^{*}_{C}$ are deconstructed into path pairs that form an X edge and added to $\mathcal{E}_X$. The *addingEdgeWillCreateEmptyCCycle()* check, performed for each path pair, holds for each path pair in our example as none of the paths are empty. For the path pairs, $((\texttt{A3}^1_1 \twoheadrightarrow \texttt{A3}^1_2), (\texttt{I1}_2 \twoheadrightarrow \texttt{I1}_3))$ and $((\texttt{A5}^1_1 \twoheadrightarrow \texttt{A5}^1_3), (\texttt{I2}_2 \twoheadrightarrow \texttt{I2}_3))$ that contain the *choose* instruction (θ), the *addDetMappings()* procedure updates $\mathcal{D}_X$ to include the mappings for the choose instructions in $(\texttt{I1}_2 \twoheadrightarrow \texttt{I1}_3)$ and $(\texttt{I2}_2 \twoheadrightarrow \texttt{I2}_3)$. For the C path $(\texttt{I1}_2 \twoheadrightarrow \texttt{I1}_3)$, two mappings are added, one for the address and another for memory, $\mathcal{D}_X(((\texttt{A3}^1_1 \twoheadrightarrow \texttt{A3}^1_2), (\texttt{I1}_2 \twoheadrightarrow \texttt{I1}_3)), (\texttt{I1}_2 \rightarrow \texttt{I1}_3{}^a), 1) = (\texttt{v}_{\texttt{I1}})$ and $\mathcal{D}_X(((\texttt{A3}^1_1 \twoheadrightarrow \texttt{A3}^1_2), (\texttt{I1}_2 \twoheadrightarrow \texttt{I1}_3)), (\texttt{I1}_2 \rightarrow \texttt{I1}_3{}^m), 1) = (M_{\ddot{A}})$ where $(\texttt{I1}_2 \rightarrow \texttt{I1}_3{}^a)$ is the edge for $\theta(\texttt{i}_{32})$ instruction (in the abbreviated edge $(\texttt{I1}_2 \rightarrow \texttt{I1}_3)$) and $(\texttt{I1}_2 \rightarrow \texttt{I1}_3{}^m)$ is the edge for $\theta(\texttt{i}_{32} \rightarrow \texttt{i}_8)$ instruction. Similarly, for $(\texttt{I2}_2 \twoheadrightarrow \texttt{I2}_3)$, the following mappings are added: $\mathcal{D}_X(((\texttt{A5}^1_1 \twoheadrightarrow \texttt{A5}^1_3), (\texttt{I2}_2 \twoheadrightarrow \texttt{I2}_3)), (\texttt{I2}_2 \rightarrow \texttt{I2}_3{}^a), 1) = (\texttt{sp.A5})$ and $\mathcal{D}_X(((\texttt{A5}^1_1 \twoheadrightarrow \texttt{A5}^1_3), (\texttt{I2}_2 \twoheadrightarrow \texttt{I2}_3)), (\texttt{I2}_2 \rightarrow \texttt{I2}_3{}^m), 1) = (M_{\ddot{A}})$.

Invariant inference and checking semantic requirements

Next, Dynamo performs invariant inference to update the invariant network Φ_X. Table 4.1 shows some of the inferred invariants for the fully-constructed X. We discuss some of the key invariants below.

n_X	ϕ_{n_X}		
$(\texttt{A3}^1_3, \texttt{I1}_4)$	① $\texttt{n} = \boxed{\texttt{lb.n}} = \boxed{\texttt{sp.}\mathit{entry}} + 4$ ④ $\texttt{m} = \boxed{\texttt{lb.m}} = \boxed{\texttt{sp.}\mathit{entry}} + 8$ ⑦ $\texttt{i} = \boxed{\texttt{lb.I1}} = \texttt{v}_{\texttt{I1}}$ ⑩ $\Sigma^{\texttt{I1}}_{\ddot{\texttt{A}}}\vert^s = \Sigma^{\texttt{I2}}_{\texttt{C}} = \Sigma^{\texttt{I9}}_{\texttt{C}} = \Sigma^{\texttt{I10}}_{\texttt{C}} = \emptyset$ ⑬ $M_{\texttt{C}} =_{\Sigma^B_{\ddot{\texttt{A}}} \setminus (\Sigma^{Z_l}_{\ddot{\texttt{A}}}\vert^v)} M_{\ddot{\texttt{A}}}$	② $\boxed{\texttt{sp.}\mathit{entry}} = \texttt{ebp} + 4$ ⑤ $\texttt{esp} * 2^{28} = 0$ ⑧ $\boxed{\texttt{lstSz.I1}} = 4$ ⑪ $\boxed{\texttt{stk}_e} = \texttt{ebp} + 15$ ⑭ $\forall_{r \in B} : \Sigma^r_{\texttt{C}} = \Sigma^r_{\ddot{\texttt{A}}}$	③ $\boxed{\texttt{sp.A3}} = \texttt{esp}$ ⑥ $\boxed{\texttt{sp.A3}} = \texttt{ebp} - 24$ ⑨ $\boxed{\texttt{em.I1}} = \texttt{false}$ ⑫ $\texttt{esp} \leq_u \boxed{\texttt{stk}_e}$ ⑮ $\Omega_{\ddot{\texttt{A}}} = \Omega_{\texttt{C}}$
$(\texttt{A5}^1, \texttt{I2}_1)$	① $\texttt{n} = \boxed{\texttt{lb.n}} = \boxed{\texttt{sp.}\mathit{entry}} + 4$ ④ $\texttt{m} = \boxed{\texttt{lb.m}} = \boxed{\texttt{sp.}\mathit{entry}} + 8$ ⑦ $\boxed{\texttt{sp.}\mathit{entry}} = \texttt{ebp} + 4$ ⑩ $\Sigma^{\texttt{I1}}_{\ddot{\texttt{A}}}\vert^s = \Sigma^{\texttt{I2}}_{\texttt{C}} = \Sigma^{\texttt{I9}}_{\texttt{C}} = \Sigma^{\texttt{I10}}_{\texttt{C}} = \emptyset$ ⑫ $M_{\texttt{C}} =_{\Sigma^B_{\ddot{\texttt{A}}} \setminus (\Sigma^{Z_l}_{\ddot{\texttt{A}}}\vert^v)} M_{\ddot{\texttt{A}}}$	② $(\boxed{\texttt{sp.}\mathit{entry}} + 4) * 2^{28} = 0$ ⑤ $\texttt{esp} * 2^{28} = 0$ ⑧ $\boxed{\texttt{sp.A3}} = \texttt{ebp} - 24$ ⑪ $\boxed{\texttt{sp.A5}} \leq_u \boxed{\texttt{sp.A3}} - 4 * (\texttt{eax} + 2)$ ⑬ $\forall_{r \in B} : \Sigma^r_{\texttt{C}} = \Sigma^r_{\ddot{\texttt{A}}}$	③ $\texttt{eax} = *\texttt{n}$ ⑥ $\texttt{ebx} = *\texttt{m}$ ⑨ $\texttt{esp} = \boxed{\texttt{sp.A5}}$ ⑭ $\Omega_{\ddot{\texttt{A}}} = \Omega_{\texttt{C}}$
$(\texttt{A10}, \texttt{I5})$	① $\texttt{v} = \boxed{\texttt{lb.I2}} = \texttt{esi}$ ④ $\texttt{m} = \boxed{\texttt{lb.m}}$ ⑦ $\Sigma^{\texttt{I1}}_{\ddot{\texttt{A}}}\vert^s = \Sigma^{\texttt{I2}}_{\ddot{\texttt{A}}}\vert^v = \Sigma^{\texttt{I9}}_{\texttt{C}} = \Sigma^{\texttt{I10}}_{\texttt{C}} = \emptyset$ ⑩ $M_{\texttt{C}} =_{\Sigma^B_{\ddot{\texttt{A}}} \setminus (\Sigma^{Z_l}_{\ddot{\texttt{A}}}\vert^v)} M_{\ddot{\texttt{A}}}$	② $\texttt{v}[*\texttt{i} - 1] = \texttt{edi}$ ⑤ $\texttt{ebx} = *\texttt{m}$ ⑧ $\boxed{\texttt{sp.}\mathit{entry}} = \texttt{ebp} + 4$ ⑪ $\forall_{r \in B} : \Sigma^r_{\texttt{C}} = \Sigma^r_{\ddot{\texttt{A}}}$	③ $\texttt{v}[*\texttt{i} - 2] = \texttt{edx}$ ⑥ $\texttt{eax} = *\texttt{i}$ ⑨ $\texttt{esp} * 2^{28} = 0$ ⑫ $\Omega_{\ddot{\texttt{A}}} = \Omega_{\texttt{C}}$
$(\texttt{A18}^2_2, \texttt{I15}_1)$	① $\Sigma^{\texttt{I1}}_{\ddot{\texttt{A}}}\vert^s = \Sigma^{\texttt{I2}}_{\ddot{\texttt{A}}}\vert^v = \Sigma^{\texttt{I9}}_{\texttt{C}} = \Sigma^{\texttt{I10}}_{\texttt{C}} = \emptyset$ ④ $M_{\texttt{C}} =_{\Sigma^B_{\ddot{\texttt{A}}} \setminus (\Sigma^{Z_l}_{\ddot{\texttt{A}}}\vert^v)} M_{\ddot{\texttt{A}}}$	② $\boxed{\texttt{sp.}\mathit{entry}} = \texttt{ebp} + 4$ ⑤ $\forall_{r \in B} : \Sigma^r_{\texttt{C}} = \Sigma^r_{\ddot{\texttt{A}}}$	③ $\texttt{v}[*\texttt{m}] = \texttt{edi}$ ⑥ $\Omega_{\ddot{\texttt{A}}} = \Omega_{\texttt{C}}$

Table 4.1: Some of the inferred inductive node invariants for the product graph X of the two procedures shown in fig. 4.5. $*v$ is short for $\texttt{sel}_4(M_{\texttt{C}}, v)$, e.g., $*\texttt{i} = \texttt{sel}_4(M_{\texttt{C}}, \texttt{i})$, $*\texttt{m} = \texttt{sel}_4(M_{\texttt{C}}, \texttt{m})$ and so on.

- At node $(\texttt{A5}^1, \texttt{I2}_1)$ (that corresponds to start of `alloc` and $\texttt{alloc}_v$ in unoptimized IR and assembly respectively), the invariants, ⑨ $\texttt{esp} = \boxed{\texttt{sp.}A5}$, ⑪ $\boxed{\texttt{sp.}A5} \leq_u \boxed{\texttt{sp.}A3} - 4 * (\texttt{eax} + 2)$, and ③ $\texttt{eax} = *\texttt{n}$ are consequential in proof of the clause $[v, v + w - 1_{\texttt{i32}}] \in \Sigma^{stk}_{\ddot{\texttt{A}}}$ of $\texttt{intrvlInSet}_a(v, v + w - 1_{\texttt{i32}}, \Sigma^{stk}_{\ddot{\texttt{A}}})$ check due to (AllocS). The proof of $\phi = \texttt{intrvlInSet}_a(v, v + w - 1_{\texttt{i32}}, \Sigma^{stk}_{\ddot{\texttt{A}}})$ makes the path $(\texttt{A5}^1_1 \twoheadrightarrow \mathcal{U}_{\texttt{A}})$ with path condition $\neg\phi$ infeasible.

- At node $(\texttt{A10}, \texttt{I5})$ (that corresponds to loop head in both unoptimized IR and assembly), the invariant ⑨ $\texttt{esp} * 2^{28} = 0$, which implies that the stackpointer `esp` is aligned by 16, helps in falsifying the check $\neg\texttt{aligned}_{16}(\texttt{esp})$ due to procedure call at `A18` in fig. 4.4c (see $(\text{Call}_{\ddot{\texttt{A}}})$ in fig. 2.8), thereby making the $\mathcal{U}_{\ddot{\texttt{A}}}$-going path $(\texttt{A18} \twoheadrightarrow \mathcal{U}_{\texttt{A}})$ infeasible.

- At node $(\texttt{A18}^2_2, \texttt{I15}_1)$ (that corresponds to deallocation of first argument to `printf`), the invariant ① $\Sigma^{\texttt{I2}}_{\ddot{\texttt{A}}}\vert^v = \emptyset$ helps in proving the infeasibility of the $\mathcal{U}_{\ddot{\texttt{A}}}$-going path

due to (AllocS').

After invariant inference, the non-coverage semantics requirements are checked through *checkSemanticReqsExceptCoverage()*. The checks for (Equivalence) and (MemEq) involve ensuring that the predicates corresponding to instantiations of $\boxed{\texttt{WEq}}$ and $\boxed{\texttt{MemEq}}$ are present in the inferred invariants ϕ_{n_X}. For example, table 4.1 contains instantiations of $\boxed{\texttt{WEq}}$ and $\boxed{\texttt{MemEq}}$ at each node; the presence of these invariant shapes is sufficient to ensure (Equivalence) and (MemEq). For the (Memory Address Correspondence) or (MAC) check, we take the example of the path pair $((\texttt{A3}_3^1 \twoheadrightarrow \texttt{A5}^1), (\texttt{I1}_4 \twoheadrightarrow \texttt{I2}_1),)$ that has two memory accesses in Ä (at A4 in fig. 4.4c): $\texttt{mem}_4(\texttt{ebp}+8)$ and $\texttt{mem}_4(\texttt{ebp}+12)$. There are no memory accesses on the path $(\texttt{I1}_4 \twoheadrightarrow \texttt{I2}_1)$, thus, for (MAC) it must be established that these memory accesses belong to the address set $\Sigma_{Ä}^{G \cup F} \cup [\texttt{esp}, \boxed{\texttt{stk}_e}]$. This is easily provable because $[\texttt{ebp}+8, \texttt{ebp}+15] \subseteq \Sigma_{Ä}^{\{n,m\}} \subseteq [\texttt{esp}, \boxed{\texttt{stk}_e}]$ is provable over the path $(\texttt{A3}_3^1 \twoheadrightarrow \texttt{A5}^1)$ due to invariants $\texttt{esp} = \boxed{\texttt{sp.A3}} = \texttt{ebp} - 24$, $\boxed{\texttt{stk}_e} = \texttt{ebp}+15$, and $\texttt{esp} \leq_u \boxed{\texttt{stk}_e}$ at $(\texttt{A3}_3^1, \texttt{I1}_4)$.

Once the first phase has finished, Dynamo invokes the second phase where the error-going paths that were added due to annotation are correlated. In fig. 4.5b, these are the $\mathcal{U}_{Ä}$ and $\mathcal{W}_{Ä}$-going paths originating from nodes $\texttt{A3}_1^1, \texttt{A5}_1^1, \texttt{A5}_2^1, \texttt{A17}^1, \texttt{A17}^2, \texttt{A18}^1, \texttt{A18}^2, \texttt{A19}^1$, and $\texttt{A19}^2$.

In the last step, Dynamo checks the (CoverageC) requirement for the added edges. We take the example of the loop path correlation here. The Ä loop path $\xi_{Ä} = \texttt{A10} \rightarrow \texttt{A14} \rightarrow \texttt{A10}$ is correlated with the paths $\xi_C^1 = \texttt{I5} \rightarrow \texttt{I6} \rightarrow \texttt{I8} \rightarrow \texttt{I5}$ and $\xi_C^2 = \texttt{I5} \rightarrow \texttt{I6} \rightarrow \mathcal{U}_C$[18], to produce the two edges, $e_X^1 = ((\texttt{A10}, \texttt{I5}) \xrightarrow{\texttt{A10} \rightarrow \texttt{A14} \rightarrow \texttt{A10}; \texttt{I5} \rightarrow \texttt{I6} \rightarrow \texttt{I8} \rightarrow \texttt{I5}} (\texttt{A10}, \texttt{I5}))$ and $e_X^2 = ((\texttt{A10}, \texttt{I5}) \xrightarrow{\texttt{A10} \rightarrow \texttt{A14} \rightarrow \texttt{A10}; \texttt{I5} \rightarrow \texttt{I6} \rightarrow \mathcal{U}_C} (\texttt{A10}, \mathcal{U}_C))$. The (CoverageC) obligation involves proving that the path cover $\{e_X^1, e_X^2\}\langle \mathcal{D}_X, \xi_{Ä} \rangle$, which is equivalent to the Hoare triple $\{\phi_{(\texttt{A10},\texttt{I5})}\}(\xi_{Ä}; \epsilon)\{pathcond([\xi_C^1]_{\mathcal{D}_X}^{e_X^1}) \vee pathcond([\xi_C^2]_{\mathcal{D}_X}^{e_X^2})\}$, holds. The invariants, ⑤ $\texttt{ebx} = *\texttt{m}$ and ⑥ $\texttt{eax} = *\texttt{i}$ at $(\texttt{A10}, \texttt{I5})$ in table 4.1 are adequate to successfully discharge this Hoare triple.

Table 4.2 shows the edges $\mathcal{E}_X$ in the final product graph X for the graphs in fig. 4.5 — $\texttt{alloc}_{s,v}$ paths not shown in fig. 4.5b have been omitted and the Ä and C paths have been abbreviated for brevity as done in fig. 4.5b. Notice that certain paths to $\mathcal{U}_{Ä}$ in Ä are not in $\mathcal{E}_X$ because they were proven infeasible, e.g. $(\texttt{A1} \twoheadrightarrow \mathcal{U}_A)$, $(\texttt{A5}_1^1 \twoheadrightarrow \mathcal{U}_A)$, $(\texttt{A18} \twoheadrightarrow \mathcal{U}_A)$, $(\texttt{A20} \twoheadrightarrow \mathcal{U}_A)$, $(\texttt{A22} \twoheadrightarrow \mathcal{U}_A)$ and so on.

[18] Representing all $\mathcal{U}_C$-going paths using a single $\texttt{I5} \rightarrow \texttt{I6} \rightarrow \mathcal{U}_C$ path.

n_X	n_X^t	$\xi_{\bar{A}}$	ξ_C
$(A0, I0)$	$(A0_1, I0_1)$	$A0 \to A0_1$	$I0 \to I0_1$
$(A0_1, I0_1)$	$(A0_2, I0_2)$	$A0_1 \to A0_2$	$I0_1 \to I0_2$
$(A0_1, I0_1)$	$(\mathscr{W}_A, I0_1)$	$A0_1 \to \mathscr{W}_A$	ϵ
$(A0_2, I0_2)$	$(A0_3, I0_3)$	$A0_2 \to A0_3$	$I0_2 \to I0_3$
$(A0_3, I0_3)$	$(A3^1, I1_1)$ $(A3^1, \mathscr{U}_C)$	$A0_3 \to A1 \to A3^1$	$I0_3 \to I1 \to I1_1$ $I0_3 \to I1 \to \mathscr{U}_C$
$(A0_3, I0_3)$	$(\mathscr{W}_A, I0_3)$	$A0_3 \to \mathscr{W}_A$ $A0_3 \to A1 \to \mathscr{W}_A$	ϵ
$(A3^1, I1_1)$	$(A3^1_1, I1_2)$	$A3^1 \to A3^1_1$	$I1_1 \to I1_2$
$(A3^1_1, I1_2)$	$(A3^1_2, I1_3)$	$A3^1_1 \to A3^1_2$	$I1_2 \to I1_3$
$(A3^1_1, I1_2)$	$(\mathscr{W}_A, I1_2)$	$A3^1_1 \to \mathscr{W}_A$	ϵ
$(A3^1_2, I1_3)$	$(A3^1_3, I1_4)$	$A3^1_2 \to A3^1_3$	$I1_3 \to I1_4$
$(A3^1_3, I1_4)$	$(A5^1, I2_1)$ $(A5^1, \mathscr{U}_C)$	$A3^1_3 \to A4 \to A5^1$	$I1_4 \to I2 \to I2_1$ $I1_4 \to I2 \to \mathscr{U}_C$
$(A3^1_3, I1_4)$	$(\mathscr{W}_A, I2_1)$ $(\mathscr{W}_A, \mathscr{U}_C)$	$A3^1_3 \to A4 \to \mathscr{W}_A$	$I1_4 \to I2 \to I2_1$ $I1_4 \to I2 \to \mathscr{U}_C$
$(A3^1_3, I1_4)$	$(\mathscr{U}_A, \mathscr{U}_C)$	$A3^1_3 \to A4 \to \mathscr{U}_A$	$I1_4 \to I2 \to \mathscr{U}_C$
$(A5^1, I2_1)$	$(A5^1_1, I2_2)$	$A5^1 \to A5^1_1$	$I2_1 \to I2_2$
$(A5^1_1, I2_2)$	$(A5^1_3, I2_3)$	$A5^1_1 \to A5^1_3$	$I2_2 \to I2_3$
$(A5^1_1, I2_2)$	$(\mathscr{W}_A, I2_2)$	$A5^1_1 \to \mathscr{W}_A$	ϵ
$(A5^1_3, I2_3)$	$(A5^1_4, I2_4)$	$A5^1_3 \to A5^1_4$	$I2_3 \to I2_4$
$(A5^1_4, I2_4)$	$(A10, I5)$ $(A10, \mathscr{U}_C)$	$A5^1_4 \to A6 \to A8 \to A10$	$I2_4 \to I3 \to I5$ $I2_4 \to I3 \to \mathscr{U}_C$
$(A10, I5)$	$(A10, I5)$ $(A10, \mathscr{U}_C)$	$A10 \to A14 \to A10$	$I5 \to I6 \to I8 \to I5$ $I5 \to I6 \to \mathscr{U}_C$
$(A10, I5)$	$(\mathscr{U}_A, \mathscr{U}_A)$	$A10 \to \mathscr{U}_A$	$I5 \to I6 \to \mathscr{U}_A$
$(A10, I5)$	$(A17^1, I9)$ $(A17^1, \mathscr{U}_C)$	$A10 \to A14 \to A15 \to A17^1$	$I5 \to I6 \to I8 \to I5 \to I9$ $I5 \to I6 \to \mathscr{U}_C$
$(A10, I5)$	$(\mathscr{W}_A, I9)$ $(\mathscr{W}_A, \mathscr{U}_C)$	$A10 \to A14 \to A15 \to \mathscr{W}_A$	$I5 \to I6 \to I8 \to I5 \to I9$ $I5 \to I6 \to \mathscr{U}_C$
$(A18, I11)$	$(A18_1, I13)$ $(A18_1, \mathscr{U}_C)$	$A18 \to A18_1$	$I11 \to I13$ $I11 \to \mathscr{U}_C$
$(A18_1, I13)$	$(A18_2, I13_1)$	$A18_1 \to A18_2$	$I13 \to I13_1$
$(A18_2, I13_1)$	$(A18_3, I13_2)$	$A18_2 \to A18_3$	$I13_1 \to I13_2$
$(A18_3, I13_2)$	$(A18_4, I13_3)$	$A18_3 \to A18_4$	$I13_2 \to I13_3$
$(A18_4, I13_3)$	$(A18^1_1, I14)$	$A18_4 \to A18^1_1$	$I13_3 \to I14$
$(A18^1_1, I14)$	$(A18^1_2, I14_1)$	$A18^1_1 \to A18^1_2$	$I14 \to I14_1$
$(A18^1_2, I14_1)$	$(A18^2_1, I15_1)$	$A18^1_2 \to A18^2_1$	$I14_1 \to I15$
$(A18^2_1, I15)$	$(A18^2_2, I15_1)$	$A18^2_1 \to A18^2_2$	$I15 \to I15_1$
$(A18^2_2, I15_1)$	$(A19^1_1, I17)$ $(A19^1_1, \mathscr{U}_C)$	$A18^2_2 \to A19^1_1$	$I15_1 \to I17$ $I15_1 \to \mathscr{U}_C$
$(A19^1_1, I17)$	$(A19^1_2, I17_1)$	$A19^1_1 \to A19^1_2$	$I17 \to I17_1$
$(A19^1_2, I17_1)$	$(A19^2_1, I18)$	$A19^1_2 \to A19^2_1$	$I17_1 \to I18$
$(A19^2_1, I18)$	$(A19^2_2, I18_1)$	$A19^2_1 \to A19^2_2$	$I18 \to I18_1$
$(A19^2_2, I18_1)$	$(A22_1, I19)$	$A19^2_2 \to A20 \to A22 \to A22_1$	$I18_1 \to I19$
$(A19^2_2, I18_1)$	$(\mathscr{W}_A, I19)$	$A19^2_2 \to A20 \to \mathscr{W}_A$	$I18_1 \to I19$
$(A22_1, I19)$	$(A22_2, I19_1)$	$A22_1 \to A22_2$	$I19 \to I19_1$
$(A22_2, I19_1)$	(AE, IE)	$A22_2 \to AE$	$I19_1 \to IE$

Table 4.2: $\mathcal{E}_X$ of X for the procedures shown in fig. 4.5. Each row represents an X edge $e_X = (n_X \xrightarrow{\xi_{\bar{A}};\xi_C} n_X^t) \in \mathcal{E}_X$.

Chapter 5

SMT Encoding

In the previous chapter, we described our algorithm DYNAMO for simultaneous automatic construction of a product graph X and automatic inference of an annotation $\ddot{\mathsf{A}}$ of procedure A. The DYNAMO algorithm generates verification conditions or proof obligations in the form of Hoare triples during its execution. These generated Hoare triples are discharged using off-the-shelf SMT solvers. In this chapter, we describe our encoding for translating a Hoare triple over X into an SMT format compatible with solvers. Our primary contribution here is an efficient SMT representation of an address set and encoding of various relations over an address set.

We organize the chapter as follows. In section 5.1, we discuss some preliminary steps we take before the SMT encoding. In section 5.2, we present an *allocation state array* representation of address sets and encoding of address set relations for this representation. In section 5.3, we describe a faster *interval encoding* for address set relations and present its proof of soundness. We conclude in section 5.4 with description of an alternate semantics for $\ddot{\mathsf{A}}$ that are amenable to a simpler SMT encoding.

5.1 Preliminary Steps

At an error-free node $n_{\mathsf{X}} = (n_{\ddot{\mathsf{A}}}, n_{\mathsf{C}}) \in \mathcal{N}_{\mathsf{X}}^{\mathcal{UW}}$ of $\mathsf{X} = (\mathcal{N}_{\mathsf{X}}, \mathcal{E}_{\mathsf{X}}, \mathcal{D}_{\mathsf{X}})$, DYNAMO may generate a proof obligation O in the form of a Hoare triple $\{\phi_{n_{\mathsf{X}}}\}(\xi_{\ddot{\mathsf{A}}}; [\xi_{\mathsf{C}}]_{\mathcal{D}_{\mathsf{X}}}^{e_{\mathsf{X}}})\{post\}$ — recall that both path cover (used in (Coverage$\mathsf{C}$)) and path infeasibility (used in (Coverage$\ddot{\mathsf{A}}$)) conditions are also represented as Hoare triples with $\xi_{\mathsf{C}} = \epsilon$. A Hoare triple O is encoded as a first-order logic predicate in theories of bitvector, arrays,

Predicate	First-order logic encoding using $\alpha \in \Sigma^r_P$
$\alpha \in \Sigma^{\vec{r}}_P \quad \vec{r} \subseteq R$	$\bigvee_{r \in \vec{r}} \alpha \in \Sigma^r_P$
$\forall_{r \in B} : \Sigma^r_{\mathsf{C}} = \Sigma^r_{\ddot{\mathsf{A}}}$	$\forall_\alpha : (\alpha \in \Sigma^B_{\mathsf{C}} \Leftrightarrow \alpha \in \Sigma^B_{\ddot{\mathsf{A}}})$
$\Sigma^r_P = \emptyset$	$\forall_\alpha : \neg(\alpha \in \Sigma^r_P)$
$(\boxed{\texttt{lb}.z} = \texttt{lb}(\Sigma^z_{\mathsf{C}}) \wedge \boxed{\texttt{ub}.z} = \texttt{ub}(\Sigma^z_{\mathsf{C}}))$	$(\forall_\alpha : \alpha \in \Sigma^z_{\mathsf{C}} \Rightarrow (\boxed{\texttt{lb}.z} \leq_u \alpha \leq_u \boxed{\texttt{ub}.z})) \wedge \boxed{\texttt{lb}.z} \in \Sigma^z_{\mathsf{C}} \wedge \boxed{\texttt{ub}.z} \in \Sigma^z_{\mathsf{C}}$
$\texttt{ov}([\alpha_b, \alpha_e], \Sigma^{\vec{r}}_P) \quad \vec{r} \subseteq R$	$\exists_\alpha : (\alpha_b \leq_u \alpha \leq_u \alpha_e) \wedge \alpha \in \Sigma^{\vec{r}}_P$
$[\alpha_b, \alpha_e] \subseteq \Sigma^{\vec{r}}_P \quad \vec{r} \subseteq R$	$\forall_\alpha : (\alpha_b \leq_u \alpha \leq_u \alpha_e) \Rightarrow \alpha \in \Sigma^{\vec{r}}_P$
$[\alpha_b, \alpha_e] = \Sigma^r_P$	$\forall_\alpha : (\alpha_b \leq_u \alpha \leq_u \alpha_e) \Leftrightarrow \alpha \in \Sigma^r_P$
$\Sigma^{\{stk\} \cup Y}_{\ddot{\mathsf{A}}} \cup (\Sigma^Z_{\ddot{\mathsf{A}}} \setminus (\Sigma^{Z_l}_{\ddot{\mathsf{A}}}\vert^v)) = [\texttt{esp}, \boxed{\texttt{stk}_e}]$	$\forall_\alpha : (\alpha \in \Sigma^{\{stk\} \cup Y}_{\ddot{\mathsf{A}}} \vee (\alpha \in \Sigma^Z_{\ddot{\mathsf{A}}} \wedge \neg(\alpha \in \Sigma^{Z_l}_{\ddot{\mathsf{A}}}\vert^v))) \Leftrightarrow (\texttt{esp} \leq_u \alpha \leq_u \boxed{\texttt{stk}_e})$
$\Sigma^{\{cs,cl\}}_{\ddot{\mathsf{A}}} = [\boxed{\texttt{stk}_e} + 1, \boxed{\texttt{cs}_e}]$	$\forall_\alpha : (\alpha \in \Sigma^{\{cs,cl\}}_{\ddot{\mathsf{A}}}) \Leftrightarrow (\boxed{\texttt{stk}_e} + 1 \leq_u \alpha \leq_u \boxed{\texttt{cs}_e})$

Table 5.1: Encodings of address set predicates using the address set membership predicate. R is the set of all region identifiers.

and uninterpreted functions over the state elements at n_{X} and discharged using an off-the-shelf SMT solver. The state elements of a procedure $P \in \{\mathsf{C}, \ddot{\mathsf{A}}\}$ are represented using bitvectors for a register/variable/ghost variable, arrays for memory M_P, and uninterpreted functions for $\texttt{read}_{\vec{v}}(\Omega_P)$ and $\texttt{io}(\Omega_P, \texttt{rw}, \vec{v})$[1].

For address sets, we describe encoding for the set-membership predicate $\alpha \in \Sigma^r_P$ for an arbitrary address α and region identifier r. All other address set predicates can be expressed in terms of the set-membership predicate; see table 5.1 for a construction. To simplify the encodings, we rely on the correct-by-construction global invariants $\boxed{\phi_{\mathsf{X}}} \subseteq \phi_{n_{\mathsf{X}}}$ (section 4.2.1) and assume that the partially-constructed X (and, therefore, node n_{X}) satisfies the (Equivalence), (MAC), and (MemEq) requirements. The (Equivalence) requirement implies the predicate $\boxed{\texttt{AllocEq}}$ and (MemEq) implies the predicate $\boxed{\texttt{MemEq}}$ (both defined in section 4.2) so that $\boxed{\texttt{AllocEq}} \in \phi_{n_{\mathsf{X}}}$ and $\boxed{\texttt{MemEq}} \in \phi_{n_{\mathsf{X}}}$.

Sets of stack-allocated and virtually-allocated locals at a node

Recall that for an allocation site $zl \in Z_l$ due to a variable declaration (or procedure parameter), at a node $n_{\mathsf{X}} \in \mathcal{N}_{\mathsf{X}}$, we use $\Sigma^{zl}_{\ddot{\mathsf{A}}}\vert^s$ and $\Sigma^{zl}_{\ddot{\mathsf{A}}}\vert^v$ to represent the address sets corresponding to the stack and virtual allocations performed in $\ddot{\mathsf{A}}$ for zl (section 2.6).

[1]Recall that `read` and `io` are the uninterpreted functions used for representing interaction with the outside world Ω_P (section 2.2.7).

Let $Zls = \{zl \mid zl \in Z_l \wedge \Sigma^{zl}_{\ddot{A}}|^s \neq \emptyset\}$ and $Zlv = \{zl \mid zl \in Z_l \wedge \Sigma^{zl}_{\ddot{A}}|^v \neq \emptyset\}$ represent the set of non-empty stack-allocated and virtually-allocated locals at n_X respectively. Recall that we restrict ourselves to only those compiler transformations that ensure the validity of $Zls \cap Zlv = \emptyset$ at each n_X (section 2.6). This enables us to work with Zls and Zlv instead of Z_l, which simplifies the encoding of $\alpha \in \Sigma^{zl}_{\ddot{A}}$ to either $\alpha \in \Sigma^{zl}_{\ddot{A}}|^s$ (for $zl \in Zls$) or $\alpha \in \Sigma^{zl}_{\ddot{A}}|^v$ (for $zl \in Zlv$) or `false` (if $zl \notin Zls \cup Zlv$). Further, because relations $\Sigma^{zl}_{\ddot{A}}|^s$ $\{=, \neq\}$ $\emptyset$ and $\Sigma^{zl}_{\ddot{A}}|^v$ $\{=, \neq\}$ $\emptyset$ are tracked through invariant shape `zEmpty` (fig. 4.2), Zls and Zlv can be identified through syntactic pattern matching over ϕ_{n_X}.

Using invariants for simplifying encoding

We use `AllocEq` (assumed to be in ϕ_{n_X} due to (Equivalence)) to replace $\alpha \in \Sigma^r_{\ddot{A}}$ with $\alpha \in \Sigma^r_C$ for $r \in B$. For example, the `MemEq` predicate $M_C =_{\Sigma^B_{\ddot{A}} \setminus (\Sigma^{Z_l}_{\ddot{A}}|^v)} M_{\ddot{A}}$ is encoded as:

$$\forall_\alpha : \alpha \in \Sigma_C^{G \cup Y \cup Zls \cup Z_a \cup \{hp, cl\}} \Rightarrow \mathtt{sel}_1(M_C, \alpha) = \mathtt{sel}_1(M_{\ddot{A}}, \alpha)$$

5.2 Representing address sets using allocation state array

Let $\mathcal{L}_P : \mathtt{i}_{32} \rightarrow R$ be an *allocation state array* that maps an address to a region identifier in procedure $P \in \{\ddot{A}, C\}$. We add a state element $\mathcal{L}_P$ to procedure P for tracking the allocation state, address sets Σ^r_P for all $r \in R \setminus \{\mathtt{free}\}$, in procedure P. The set-membership predicate $\alpha \in \Sigma^r_P$ for $r \in R$, which can be used for encoding all other predicates over address set Σ^r_P, (table 5.1), is encoded using $\mathcal{L}_P$ as follows.

- For $r \notin Zlv$, $\alpha \in \Sigma^r_P$ is encoded as $\mathtt{sel}_1(\mathcal{L}_P, \alpha) = r$.
- For $r \in Zlv$, both $\alpha \in \Sigma^r_C$ and $\alpha \in \Sigma^r_{\ddot{A}}$ are encoded as $\mathtt{sel}_1(\mathcal{L}_C, \alpha) = r$, i.e., the encodings for both $\ddot{A}$ and C use $\mathcal{L}_C$ for virtually-allocated locals (by relying on the `AllocEq` invariant that is guaranteed to hold at n_X).

Thus, $\mathcal{L}_{\ddot{A}}$ is not used to track the virtually-allocated locals; instead, an address belonging to a virtually allocated-region maps to one of $\{\mathtt{free}, stk, cs\} \cup F$ regions in $\mathcal{L}_{\ddot{A}}$.

An array-based encoding that maps a unique region to an address is possible in C because of global invariant `NoOverlapC` that forbids any overlap among all regions in

#	Instruction	SMT Encoding using $\mathcal{L}_P$
1	$\Sigma_P^r \coloneqq \Sigma_P^r \cup [\alpha_b, \alpha_e]; \quad r \in \{stk\} \cup Z$	$\mathcal{L}_P{}' = \texttt{cwrite}(\mathcal{L}_P, \lambda x.x \in [\alpha_b, \alpha_e], r)$
2	$\Sigma_{\ddot{A}}^{stk} \coloneqq \Sigma_{\ddot{A}}^{stk} \setminus [\alpha_b, \alpha_e];$	$\mathcal{L}_{\ddot{A}}{}' = \texttt{cwrite}(\mathcal{L}_{\ddot{A}}, \lambda x.x \in [\alpha_b, \alpha_e], \texttt{free})$
3	$\Sigma_P^z \coloneqq \emptyset;$	$\mathcal{L}_P{}' = \texttt{cwrite}(\mathcal{L}_P, \lambda x.\texttt{sel}_1(\mathcal{L}_P, x) = z, \texttt{free})$
4	$\Sigma_{\ddot{A}}^{stk} \coloneqq \{[\texttt{esp}, \boxed{\texttt{stk}_e}]\} \setminus \Sigma_{\ddot{A}}^{Y};$	$\mathcal{L}_{\ddot{A}}{}' = \texttt{cwrite}(\mathcal{L}_{\ddot{A}}, \lambda x.x \in [\texttt{esp}, \boxed{\texttt{stk}_e}] \wedge \bigwedge_{y \in Y} x \notin \Sigma_{\ddot{A}}^{y}, stk)$

Table 5.2: SMT encoding of address set updating instructions using allocation state array $\mathcal{L}_P$. $P \in \{C, \ddot{A}\}$. $\mathcal{L}_P{}'$ is the allocation state array obtained after executing the instruction.

C at an error-free node $n_C \in \mathcal{N}_C^{\overline{UW}}$. In $\ddot{A}$, $\boxed{\texttt{NoOverlap}_{\ddot{A}}}$ permits overlap of $\Sigma_{\ddot{A}}^{Zlv}$ with $\Sigma_{\ddot{A}}^{\{stk,cs\}\cup F}$ and, consequently, $\mathcal{L}_{\ddot{A}}$ is only used for tracking regions other than Zlv.

As $\mathcal{L}_P$ is an array state element, similar to M_P, it is directly encode-able in SMT using the theory of arrays.

5.2.1 Encoding of address set updating instructions

An address set updating instruction of the form '$\Sigma_P^r \coloneqq e(\ldots)$' updates the address set of region $r \in R$. In an allocation state array $\mathcal{L}_P$ representation, an update '$\Sigma_P^r \coloneqq e(\ldots)$' produces a new allocation state array $\mathcal{L}_P{}'$. Table 5.2 shows $\mathcal{L}_P$ based SMT encoding of the graph instructions that update address sets. We list the encoding for the four kinds of address set updating instructions that appear in our translations (figs. 2.5 to 2.8, 2.10 and 2.11).

Table 5.2 uses an auxiliary "conditional write" operator `cwrite` to encode the update of $\mathcal{L}_P$. If $\mathcal{L}_P{}' = \texttt{cwrite}(\mathcal{L}_P, \lambda x.c, v)$, then the following holds:

$$\forall_\alpha : \quad (\lambda x.c)(\alpha) \Rightarrow \texttt{sel}_1(\mathcal{L}_P{}', \alpha) = v$$
$$\wedge\, \neg(\lambda x.c)(\alpha) \Rightarrow \texttt{sel}_1(\mathcal{L}_P{}', \alpha) = \texttt{sel}_1(\mathcal{L}_P, \alpha)$$

Here, $(\lambda x.c)$ represents a function that takes an address x and returns a boolean evaluated through expression c, and $(\lambda x.c)(\alpha)$ represents the application of this function to address α. Thus, $\texttt{cwrite}(\mathcal{L}_P, \lambda x.c, v)$ represents the modification of allocation state array $\mathcal{L}_P$ to value v for all addresses α that satisfy the boolean condition c. In other words, $\texttt{cwrite}(\mathcal{L}_P, \lambda x.c, v)$ is equivalent to $\texttt{st}_1(\ldots \texttt{st}_1(\ldots \texttt{st}_1(\mathcal{L}_P, \alpha_1, v), \ldots, \alpha_i, v), \ldots, \alpha_n, v)$

for all $\alpha_1, \ldots, \alpha_i, \ldots, \alpha_n$ where the predicate c holds [2].

We discuss the specific cases of allocation and deallocation below:

- Allocation of an interval $[\alpha_b, \alpha_e]$ to region r through instruction '$\Sigma_P^r \coloneqq \Sigma_P^r \cup [\alpha_b, \alpha_e]$' is encoded as $\mathcal{L}_P{}' = \texttt{cwrite}(\mathcal{L}_P, \lambda x.x \in [\alpha_b, \alpha_e], r)$ which translates to "mark the addresses in interval $[\alpha_b, \alpha_e]$ as belonging to region r in $\mathcal{L}_P{}'$" (shown in row 1 of table 5.2 with $r \in \{stk\} \cup Z$).
- Similarly, deallocation of an interval $[\alpha_b, \alpha_e]$ from region r through instruction '$\Sigma_P^r \coloneqq \Sigma_P^r \setminus [\alpha_b, \alpha_e]$' is encoded as $\mathcal{L}_P{}' = \texttt{cwrite}(\mathcal{L}_P, \lambda x.x \in [\alpha_b, \alpha_e], \texttt{free})$ which translates to "mark the addresses in interval $[\alpha_b, \alpha_e]$ as belonging to region `free` in $\mathcal{L}_P{}'$" (shown in row 2 of table 5.2 with $r = stk$).
  The deallocation of a region r through instruction '$\Sigma_P^r \coloneqq \emptyset$' is encoded as $\mathcal{L}_P{}' = \texttt{cwrite}(\mathcal{L}_P, \lambda x.\texttt{sel}_1(\mathcal{L}_P, x) = r, \texttt{free})$ where the addresses to be set to `free` are identified using the predicate $\texttt{sel}_1(\mathcal{L}_P, x)$ (instead of a range check $x \in [\alpha_b, \alpha_e]$ as done in case of deallocation of an interval from r).

As $\mathcal{L}_{\ddot{A}}$ does not track virtually-allocated locals, the (de)allocation instructions $\Sigma_{\ddot{A}}^{zlv}|^v \coloneqq \Sigma_{\ddot{A}}^{zlv}|^v \cup [v]_w$ and $\Sigma_{\ddot{A}}^{zlv}|^v \coloneqq \emptyset$ ((AllocV) and (DeallocV) in fig. 2.10) for $zlv \in Zlv$ become vacuous in $\ddot{A}$, i.e., they do not change any state element in $\ddot{A}$.

5.2.2 Full-array encoding

We call this allocation state arrays $\mathcal{L}_C$ and $\mathcal{L}_{\ddot{A}}$ based representation of address sets and corresponding encoding of address set relations a *full-array encoding*. In a full-array encoding, the address sets of C are tracked using $\mathcal{L}_C$ and the address set of $\ddot{A}$ are tracked using a combination of $\mathcal{L}_C$ (for a region $r \in Zlv$) and $\mathcal{L}_{\ddot{A}}$ (otherwise). A predicate $\alpha \in \Sigma_P^r$ for a region r is encoded as an SMT *array select* operation $\texttt{select}(\mathcal{L}_P, \alpha)$ over the respective array $\mathcal{L}_P$.

A proof obligation encoded using full-array encoding contains constructs with quantifiers over SMT arrays (tables 5.1 and 5.2). Such a proof obligation, with quantifiers over arrays, can be (relatively) slow to discharge using SMT solvers. In the subsequent sections, we describe an *interval encoding* that makes use of the global invariants for a more performant encoding attainable under certain conditions. We confirm the relative better performance of our interval encoding in our experiments (section 6.2).

[2] Recall that $\texttt{st}_1(\mathcal{L}_P, \alpha_1, v)$ is our size-associated *store* operation which is equivalent to SMT expression $\texttt{store}(\mathcal{L}_P, \alpha_1, v)$ in this particular case.

5.3 Interval Encoding

5.3.1 Interval encoding for $r \in G \cup F \cup Y \cup Z_l \cup \{stk\}$

Recall that the address set Σ_P^r of a region $r \in G \cup F \cup Y \setminus \{\texttt{vrdc}\}$ is an interval and the address set Σ_P^r of a region $r \in Z_l \cup \{\texttt{vrdc}\}$ is either empty or an interval (section 4.2.1) — the global invariants $\boxed{\texttt{gfyIntvl}}$ and $\boxed{\texttt{zlIntvl}}$ encode this in ϕ_{n_X}. We use these invariants (and $\boxed{\texttt{AllocEq}}$) for a more performant *interval encoding*. In the interval encoding, we encode $\alpha \in \Sigma_P^r$ for $r \in G \cup F \cup Y \cup Z_l$ as[3]

$$\neg\,\texttt{em.}r \wedge (\texttt{lb.}r \leq_u \alpha \leq_u \texttt{ub.}r)$$

Moreover, if there are no local variables allocated due to the `alloca()` operator (i.e., $\Sigma_P^{Z_a} = \emptyset$), then all local variables are contiguous and thus amenable for an interval encoding. In this case, due to $\boxed{\texttt{StkBd}}$, the *stk* region can be identified as $[\texttt{esp}, \texttt{stk}_e] \setminus \Sigma_{\ddot{A}}^{Y \cup Zls}$.

As the interval encoding utilizes ghost variables ($\texttt{em.}z$, $\texttt{lb.}z$, $\texttt{ub.}z$), that are updated during an (de)allocation, the SMT encoding of an address set mutating instruction becomes vacuous.

5.3.2 Interval encoding for $r \in \{hp, cl, cs\}$

Even though the regions *hp* (heap), *cl* (callers' locals), and *cl* (callers' stack) can be discontiguous in general, we over-approximate these regions to their contiguous covers to be able to soundly encode them using intervals. Recall that our proof obligation O is a Hoare triple of the form $\{\phi_{n_X}\}(\xi_{\ddot{A}}; [\xi_C]^{e_X}_{\mathcal{D}_X})\{post\}$ such that $n_X = (n_{\ddot{A}}, n_C) \in \mathcal{N}_X^{\mathcal{UW}}$. Here, if $\xi_{\ddot{A}}$ is an I/O path, its execution interacts with the outside world, and so an over-approximation of an externally-visible address set is unsound. We thus restrict ourselves to an I/O-free $\xi_{\ddot{A}}$ (consequently, I/O-free ξ_C due to (SingleIO)) for our interval encoding.

Let $n_{\ddot{A}}^1, n_{\ddot{A}}^2, \ldots, n_{\ddot{A}}^m$ be the nodes on path $\xi_{\ddot{A}} = (n_{\ddot{A}} \twoheadrightarrow n_{\ddot{A}}^t)$, such that $n_{\ddot{A}}^1 = n_{\ddot{A}}$ and $n_{\ddot{A}}^m = n_{\ddot{A}}^t$. Let $SP_{min}(\xi_{\ddot{A}})$ represent the minimum value of stackpointer `esp` observed at any node $n_{\ddot{A}}^j$ $(1 \leq j \leq m)$ visited during the execution of path $\xi_{\ddot{A}}$. Similarly, let $Zlv_U(\xi_{\ddot{A}})$ be the union of the values of address set $\Sigma_{\ddot{A}}^{Zlv}$ observed at any $n_{\ddot{A}}^j$ $(1 \leq j \leq m)$ visited during

[3] Recall that $\texttt{em.}r$, $\texttt{lb.}r$, and $\texttt{ub.}r$ are ghost variables that capture the emptiness of region r, lower bound of region r, and upper bound of region r respectively.

$\xi_{\ddot{A}}$'s execution. Intuitively, $SP_{min}(\xi_{\ddot{A}})$ is the lowest possible value of `esp` [4] and $Zlv_U(\xi_{\ddot{A}})$ is the largest possible value of set $\Sigma_{\ddot{A}}^{Zlv}$ seen during $\xi_{\ddot{A}}$'s execution starting at n_X.

Let $HP(\xi_{\ddot{A}})$, $CL(\xi_{\ddot{A}})$, and $CS(\xi_{\ddot{A}})$ be defined as shown below.

$$\begin{aligned} HP(\xi_{\ddot{A}}) &= \texttt{comp}(\Sigma_{\ddot{A}}^{G \cup F} \cup Zlv_U(\xi_{\ddot{A}}) \cup [SP_{min}(\xi_{\ddot{A}}), \boxed{\texttt{cs}_e}]) \\ CL(\xi_{\ddot{A}}) &= [\boxed{\texttt{stk}_e} + 1_{\texttt{i32}}, \boxed{\texttt{cs}_e}] \setminus Zlv_U(\xi_{\ddot{A}}) \\ CS(\xi_{\ddot{A}}) &= [\boxed{\texttt{stk}_e} + 1_{\texttt{i32}}, \boxed{\texttt{cs}_e}] \cap Zlv_U(\xi_{\ddot{A}}) \end{aligned}$$

Intuitively, $HP(\xi_{\ddot{A}})$ and $CL(\xi_{\ddot{A}})$ are the largest possible values for $\Sigma_{\ddot{A}}^{hp}$ and $\Sigma_{\ddot{A}}^{cl}$ respectively such that ϕ_{n_X} is satisfied and $\xi_{\ddot{A}}$ is executed to completion; as cs and cl are complementary (due to $\boxed{cs\texttt{Bd}}$), this makes $CS(\xi_{\ddot{A}})$ the smallest possible value under the same conditions.

Theorem 5.3.1. *Let $O = \{pre\}(\xi_{\ddot{A}}; [\xi_C]^{ex}_{\mathcal{D}_X})\{post\}$ be a proof obligation generated by* DYNAMO. *Let O' be obtained from O by strengthening precondition pre to $pre' = pre \wedge (\Sigma_{\ddot{A}}^{hp} = HP(\xi_{\ddot{A}})) \wedge (\Sigma_{\ddot{A}}^{cl} = CL(\xi_{\ddot{A}})) \wedge (\Sigma_{\ddot{A}}^{cs} = CS(\xi_{\ddot{A}}))$. If $\xi_{\ddot{A}}$ is I/O-free, $O \Leftrightarrow O'$ holds.*

Proof sketch. $O \Rightarrow O'$ is trivial. The proof for $O' \Rightarrow O$ relies on the limited shapes of predicates that may appear in pre and $post$, the enumeration of $\mathcal{U}$-maximal pathsets for C (section 4.1.2), and use of safety-relaxed semantics for Ä (section 3.5). For I/O-free $\xi_{\ddot{A}}$, pre and $post$ shapes are limited by our invariant grammar (fig. 4.2), and the edge conditions appearing in our execution semantics (figs. 2.4 to 2.8, 2.10 and 2.11). The full proof is available in section 5.3.3. □

Using theorem 5.3.1, we rewrite $\alpha \in \Sigma_P^{hp}$ to $\alpha \in HP(\xi_{\ddot{A}})$, $\alpha \in \Sigma_P^{cl}$ to $\alpha \in CL(\xi_{\ddot{A}})$, and $\alpha \in \Sigma_P^{cs}$ to $\alpha \in CS(\xi_{\ddot{A}})$ in proof obligation O. If $\Sigma_P^{Z_a} = \emptyset$ holds at n_X, we encode all non-free regions using intervals — we call this a *full-interval encoding*; else, we encode regions in $Y \cup Z_a \cup Zls \cup \{stk\}$ using an allocation state array, and $G \cup F \cup Zlv \cup \{hp, cl, cs\}$ using intervals — we call this a *partial-interval encoding*.

Table 5.3 shows the SMT encoding of $\alpha \in \Sigma_P^r$ in *full-array*, *partial-interval*, and *full-interval* encoding. The column selects the encoding and the row selects the region. We groups regions with common encoding into a single row, e.g., the third row gives the encoding for a region $r \in G \cup Zlv$. We merge cells with common entries for

[4] Recall that stack is allocated by decrementing `esp`.

<table>
<tr><td rowspan="2">$\alpha \in \Sigma^r_P$</td><td colspan="2">Full-array encoding</td><td>Partial-interval encoding</td><td>Full-interval encoding</td></tr>
<tr><td>$P = \mathsf{C}$</td><td>$P = \mathsf{A}$</td><td>$(\Sigma^{Z_a}_P \neq \emptyset)$</td><td>$(\Sigma^{Z_a}_P = \emptyset)$</td></tr>
<tr><td>$r = hp$</td><td colspan="2" rowspan="4">$\mathtt{sel}_1(\mathcal{L}_\mathsf{C}, \alpha) = r$</td><td colspan="2">$\alpha \notin (\Sigma^{G \cup F}_{\ddot{\mathsf{A}}} \cup Zlv_U(\xi_{\ddot{\mathsf{A}}}) \cup [SP_{min}(\xi_{\ddot{\mathsf{A}}}), \boxed{\mathtt{cs}_e}])$</td></tr>
<tr><td>$r = cl$</td><td colspan="2">$\alpha \in [\boxed{\mathtt{stk}_e} + 1, \boxed{\mathtt{cs}_e}] \wedge \alpha \notin Zlv_U(\xi_{\ddot{\mathsf{A}}})$</td></tr>
<tr><td>$r \in G \cup Zlv$</td><td colspan="2" rowspan="2">$\neg \boxed{\mathtt{em}.r} \wedge (\boxed{\mathtt{lb}.r} \leq_u \alpha \leq_u \boxed{\mathtt{ub}.r})$</td></tr>
<tr><td>$r \in Y \cup Z_a \cup Zls$</td></tr>
<tr><td>$r \in F$</td><td rowspan="3">$\mathtt{false}$</td><td rowspan="3">$\mathtt{sel}_1(\mathcal{L}_{\ddot{\mathsf{A}}}, \alpha) = r$</td><td colspan="2" rowspan="2">$\alpha \in [\boxed{\mathtt{stk}_e} + 1, \boxed{\mathtt{cs}_e}] \wedge \alpha \in Zlv_U(\xi_{\ddot{\mathsf{A}}})$</td></tr>
<tr><td>$r = cs$</td></tr>
<tr><td>$r = stk$</td><td>$\alpha \in [\mathtt{esp}, \boxed{\mathtt{stk}_e}] \wedge \bigwedge_{r \in Y \cup Zls}(\alpha \notin \Sigma^r_{\ddot{\mathsf{A}}})$</td></tr>
</table>

Table 5.3: SMT encoding of $\alpha \in \Sigma^r_P$ for DYNAMO's proof obligation O with outgoing assembly path $\xi_{\ddot{\mathsf{A}}}$.

clearer presentation, e.g., instead of repeating $\mathtt{sel}_1(\mathcal{L}_\mathsf{C}, \alpha) = r$ for full-array encoding of $r \in \{hp, cl\} \cup G \cup Zlv \cup Y \cup Z_a \cup Zls$, we merge the cells for each of these entries into a single cell. As $\Sigma^r_\mathsf{C} = \emptyset$ for $r \in F \cup \{cs, stk\}$, $\alpha \in \Sigma^r_\mathsf{C}$ is encoded as $\mathtt{false}$.

5.3.3 Soundness of Interval Encoding

Let the Hoare triple representation of a proof obligation O generated by DYNAMO be $\{pre\}(\xi_{\ddot{\mathsf{A}}}; [\xi_\mathsf{C}]^{e_\mathsf{X}}_{\mathcal{D}_\mathsf{X}})\{post\}$, where $\xi_{\ddot{\mathsf{A}}} = (n_{\ddot{\mathsf{A}}} \twoheadrightarrow n^t_{\ddot{\mathsf{A}}})$ and either $\xi_\mathsf{C} = \epsilon$ or $\xi_\mathsf{C} = (n_\mathsf{C} \twoheadrightarrow n^t_\mathsf{C})$; both $\xi_{\ddot{\mathsf{A}}}$ and ξ_C are I/O-free execution paths in $\ddot{\mathsf{A}}$ and C respectively; $n_\mathsf{X} = (n_{\ddot{\mathsf{A}}}, n_\mathsf{C}) \in \mathcal{N}^{\mathcal{UW}}_\mathsf{X}$ is an error-free node; if $\xi_\mathsf{C} = (n_\mathsf{C} \twoheadrightarrow n^t_\mathsf{C})$, then $e_\mathsf{X} = (n_\mathsf{X} \xrightarrow{\xi_{\ddot{\mathsf{A}}}; \xi_\mathsf{C}} n^t_\mathsf{X}) \in \mathcal{E}_\mathsf{X}$ and $n^t_\mathsf{X} = (n^t_{\ddot{\mathsf{A}}}, n^t_\mathsf{C}) \in \mathcal{N}_\mathsf{X}$.

Let $HP(\xi_{\ddot{\mathsf{A}}})$, $CL(\xi_{\ddot{\mathsf{A}}})$, and $CS(\xi_{\ddot{\mathsf{A}}})$ be defined as described in previous section.

Let $O' = \{pre\}(\xi_{\ddot{\mathsf{A}}}; \xi_\mathsf{C})\{post\}$ be obtained by strengthening precondition pre to $pre' = pre \wedge (\Sigma^{hp}_{\ddot{\mathsf{A}}} = HP(\xi_{\ddot{\mathsf{A}}})) \wedge (\Sigma^{cl}_{\ddot{\mathsf{A}}} = CL(\xi_{\ddot{\mathsf{A}}})) \wedge (\Sigma^{cs}_{\ddot{\mathsf{A}}} = CS(\xi_{\ddot{\mathsf{A}}}))$ in O'. We need to show that $O \Leftrightarrow O'$ holds.

($\Rightarrow$) Proving $O \Rightarrow O'$ is trivial, as O' requires a stronger precondition than O (with everything else identical).

($\Leftarrow$) Assume that O' holds. We are interested in showing that O holds. Assume a machine state σ of product program X that satisfies the weaker precondition pre, and

executes to completion over $\xi_{\ddot{A}}$ and ξ_C. We are interested in showing that σ satisfies the postcondition *post* after completing the execution.

We define "error-free execution" to be the case where the execution on a state σ across $(\xi_{\ddot{A}}; \xi_C)$ does not end at an error node in X.

Lemma 5.3.2 ($HP(\xi_{\ddot{A}})$,$CL(\xi_{\ddot{A}})$ overapproximate *hp*,*cl*)**.** $(\Sigma_{\ddot{A}}^{hp} \subseteq HP(\xi_{\ddot{A}})) \wedge (\Sigma_{\ddot{A}}^{cl} \subseteq CL(\xi_{\ddot{A}}))$ *holds on* σ *for an error-free execution.*

Proof. Recall that $pre \Rightarrow \phi_{n_X}$. If $\Sigma_{\ddot{A}}^{hp} \supset HP(\xi_{\ddot{A}})$ or $\Sigma_{\ddot{A}}^{cl} \supset CL(\xi_{\ddot{A}})$, then either at least one of $\boxed{\texttt{NoOverlap}A}$ or $\boxed{\texttt{NoOverlap}C}$ will evaluate to `false` in ϕ_{n_X} (and *pre*), or during the execution of path $\xi_{\ddot{A}}$; error $\mathscr{W}$ will be triggered in $\ddot{A}$ because either the allocation of stack space through stackpointer decrement will overstep $\Sigma_{\ddot{A}}^{\{hp,cl\}}$ (Op-esp'), or the virtual allocation of a local variable will overstep $\Sigma_{\ddot{A}}^{\{hp,cl\}}$ (AllocV). However, by assumption, σ satisfies *pre* (and ϕ_{n_X}) and executes $\xi_{\ddot{A}}$ and ξ_C to completion to an error-free node; thus proved by contradiction. □

Lemma 5.3.3 ($CS(\xi_{\ddot{A}})$ underapproximates *cs*)**.** $(\Sigma_{\ddot{A}}^{cs} \supseteq CS(\xi_{\ddot{A}}))$ *holds on* σ *for an error-free execution.*

Proof. Follows from lemma 5.3.2 and $\boxed{csBd}$ $(\Sigma_{\ddot{A}}^{cs} = [\boxed{\texttt{stk}_e} + 1_{i32}, \boxed{\texttt{cs}_e}] \setminus \Sigma_{\ddot{A}}^{cl})$. □

Lemma 5.3.4 ($HP(\xi_{\ddot{A}})$ and $CL(\xi_{\ddot{A}})$ borrow from the `free` and *cs* regions)**.** *The following hold on* σ *for an error-free execution.*

1. $(HP(\xi_{\ddot{A}}) \setminus \Sigma_{\ddot{A}}^{hp}) \subseteq \Sigma_{\ddot{A}}^{\texttt{free}} \subseteq \Sigma_C^{\texttt{free}}$
2. $(CL(\xi_{\ddot{A}}) \setminus \Sigma_{\ddot{A}}^{cl}) \subseteq \Sigma_{\ddot{A}}^{cs} \subseteq \Sigma_C^{\texttt{free}}$

Proof. The proof follows from the definition of $HP(\xi_{\ddot{A}})$ and $CL(\xi_{\ddot{A}})$, as these sets are not allowed to overlap with $\Sigma_{\ddot{A}}^{B \cup F \cup S}$ or $\Sigma_C^{B \cup F \cup S}$. □

Construct a state σ' that is identical to σ with the following modifications made in sequence:

1. The region identified by addresses (that would belong to region `free` in C by lemma 5.3.4) $(HP(\xi_{\ddot{A}}) \cup CL(\xi_{\ddot{A}})) \setminus \Sigma_{\ddot{A}}^{\{hp,cl\}}$ in σ''s M_C is updated through $M_C := \texttt{upd}_{(HP(\xi_{\ddot{A}}) \cup CL(\xi_{\ddot{A}})) \setminus \Sigma_{\ddot{A}}^{\{hp,cl\}}}(M_C, M_{\ddot{A}})$.

2. The address sets $\Sigma^{hp}_{Ä}$, $\Sigma^{cl}_{Ä}$, Σ^{hp}_{C}, and Σ^{cl}_{C} are expanded and the address set $\Sigma^{cs}_{Ä}$ is shrunk so that $\Sigma^{hp}_{Ä} = \Sigma^{hp}_{C} = HP(\xi_{Ä})$, $\Sigma^{cl}_{Ä} = \Sigma^{cl}_{C} = CL(\xi_{Ä})$, and $\Sigma^{cs}_{Ä} = CS(\xi_{Ä})$ (this involves the transfer of addresses from the `free` region to *hp* and *cl* regions in C, and from the `free` and *cs* regions to *hp* and *cl* regions respectively in Ä (lemma 5.3.4)).

The constructed state σ' thus satisfies the stronger precondition pre'.

Let Σ^{hp}_{σ} ($\Sigma^{hp}_{\sigma'}$), Σ^{cl}_{σ} ($\Sigma^{cl}_{\sigma'}$), Σ^{cs}_{σ} ($\Sigma^{cs}_{\sigma'}$), and $\Sigma^{\texttt{free}}_{\sigma}$ ($\Sigma^{\texttt{free}}_{\sigma'}$) denote the values of $\Sigma^{hp}_{Ä}$, $\Sigma^{cl}_{Ä}$, $\Sigma^{cs}_{Ä}$, and $\Sigma^{\texttt{free}}_{Ä}$ in state σ (σ') respectively. Similarly, let $M^{\sigma}_{Ä}$ (M^{σ}_{C}) and $M^{\sigma'}_{Ä}$ ($M^{\sigma'}_{C}$) represent the state of procedure Ä's (C's) memory $M_{Ä}$ (M_{C}) in machine states σ and σ' respectively.

To relate σ and σ', we define relation $sim(\sigma, \sigma')$ as the conjunction of the following conditions:

1. (*hp* subset in σ) $\Sigma^{hp}_{\sigma} \subseteq \Sigma^{hp}_{\sigma'}$.
2. (*cl* subset in σ) $\Sigma^{cl}_{\sigma} \subseteq \Sigma^{cl}_{\sigma'}$.
3. (*cs* superset in σ) $\Sigma^{cs}_{\sigma} \supseteq \Sigma^{cs}_{\sigma'}$.
4. (`free` superset in σ) $\Sigma^{\texttt{free}}_{\sigma} \supseteq \Sigma^{\texttt{free}}_{\sigma'}$.
5. (Ä's memory states are equal) $M^{\sigma}_{Ä} = M^{\sigma'}_{Ä}$
6. (C's memory states are equal except at the updated regions) $M^{\sigma}_{C} =_{\texttt{comp}(\Sigma^{\{hp,cl\}}_{\sigma'} \setminus \Sigma^{\{hp,cl\}}_{\sigma})} M^{\sigma'}_{C}$.
7. The remaining state elements have equal values in σ and σ'.

By construction, $sim(\sigma, \sigma')$ holds.

Lemma 5.3.5 ($sim(\sigma, \sigma')$ is preserved for error-free execution across all non-I/O edges in $\mathcal{E}_{Ä}$)**.** *If a non-I/O edge $e_{Ä} \in \mathcal{E}_{Ä}$ is executed on both machine states σ and σ', and if $sim(\sigma, \sigma')$ holds before the execution, and if the execution on σ completes without error, then there exists a sequence of non-deterministic choices during the execution on σ' such that the execution is error-free and $sim(\sigma, \sigma')$ holds at the end of both error-free executions.*

Proof. For each non-I/O Ä instruction that does not refer to the $\{hp, cl, cs, \texttt{free}\}$ regions ((Op-Nesp),(AllocS), (DeallocS), (Call$_{Ä}$), (Ret$_{A}$), (DeallocV)), the execution will have identical behaviour on both σ and σ', as identical values will be observed in σ and σ'. Thus, if an execution on σ' makes the same non-deterministic choice as the execution on σ, the execution on σ' will complete without error and

$sim(\sigma, \sigma')$ will hold at the end of both executions.

We consider each remaining non-I/O instruction in $\ddot{A}$ below:

- ($\text{ENTRY}_{\ddot{A}}$). Consider the overlap conditions $\Upsilon_1 = \texttt{ov}(\Sigma^{hp}_{\ddot{A}}, \Sigma^{cl}_{\ddot{A}}, \ldots, i^{g}_{\ddot{A}}, \ldots, \Sigma^{f}_{\ddot{A}}, \ldots, i^{y}_{\ddot{A}}, \ldots, \Sigma^{\texttt{vrdc}}_{\ddot{A}})$ (due to $\neg\texttt{addrSetsAreWF}$), $\Upsilon_2 = \texttt{ov}([\texttt{esp}, \texttt{esp} + 3_{\texttt{i}_{32}}], \Sigma^{B\cup F}_{\ddot{A}})$, $\Upsilon_3 = \texttt{ov}([\boxed{\texttt{stk}_e} + 1_{\texttt{i}_{32}}, \boxed{\texttt{cs}_e}], \Sigma^{\{hp\}\cup G\cup F}_{\ddot{A}})$, and $\Upsilon_4 = \texttt{ov}(\Sigma^{cl}_{\ddot{A}}, \texttt{comp}([\boxed{\texttt{stk}_e} + 1_{\texttt{i}_{32}}, \boxed{\texttt{cs}_e}]))$ (due to $\texttt{stkIsWF}$). During an execution on σ, all four conditions must evaluate to $\texttt{false}$, as we assume an error-free execution on σ. For the same non-deterministic choices made in both executions (over σ and σ'), by the definitions of $HP(\xi_{\ddot{A}})$ and $CL(\xi_{\ddot{A}})$, Υ_1, Υ_2, Υ_3, and Υ_4 will also evaluate to $\texttt{false}$ for an execution on σ' — recall that $HP(\xi_{\ddot{A}})$ cannot overlap with $[\texttt{esp}, \boxed{\texttt{cs}_e}]$ (which includes the arguments) and global variable regions (due to lemma 5.3.4); and $CL(\xi_{\ddot{A}})$ is a subset of $[\boxed{\texttt{stk}_e} + 1_{\texttt{i}_{32}}, \boxed{\texttt{cs}_e}]$ (by definition). Further, because all other state elements observed during the execution of the non-I/O edges in ($\text{ENTRY}_{\ddot{A}}$) are identical in both σ and σ', $sim(\sigma, \sigma')$ will hold at the end of error-free executions.

- (OP-ESP). The negated subset check $\Upsilon = \neg([t, \texttt{esp} - 1_{\texttt{i}_{32}}] \subseteq \Sigma^{\texttt{free}}_{\ddot{A}} \cup \Sigma^{Z_l|^v}_{\ddot{A}})$ (due to $\neg\texttt{intrvlInSet}(t, \texttt{esp} - 1_{\texttt{i}_{32}}, \Sigma^{\texttt{free}}_{\ddot{A}} \cup \Sigma^{Z_l|^v}_{\ddot{A}})$) depends (indirectly) on the addresses of the set $\Sigma^{\{hp,cl\}}_{\ddot{A}}$ (as $\texttt{free}$ is defined as complement of the allocated region). The execution on σ must evaluate Υ to $\texttt{false}$ as we assume an error-free execution. By the definitions of $HP(\xi_{\ddot{A}})$ and $CL(\xi_{\ddot{A}})$, for the same non-deterministic choices made in both executions (over σ and σ'), Υ will also evaluate to $\texttt{false}$ for an execution on σ' — recall that $(HP(\xi_{\ddot{A}}) \cup CL(\xi_{\ddot{A}}))$ cannot overlap with $[SP_{min}(\xi_{\ddot{A}}), \boxed{\texttt{stk}_e}]$, and the latter includes $[t, \texttt{esp} - 1_{\texttt{i}_{32}}]$. All other state elements observed in the other instructions of (OP-ESP) are identical in both σ, σ' and $sim(\sigma, \sigma')$ will hold at the end of error-free executions.

- (ALLOCV). Consider the negated subset check $\Upsilon = \neg([v]_w \subseteq \Sigma^{\texttt{comp}(B)}_{\ddot{A}})$ (due to $\neg\texttt{intrvlInSet}_a(v, v + w - 1_{\texttt{i}_{32}}, \Sigma^{\texttt{comp}(B)}_{\ddot{A}})$). The execution on σ must evaluate Υ to $\texttt{false}$ as we assume an error-free execution. By the definitions of $HP(\xi_{\ddot{A}})$ and $CL(\xi_{\ddot{A}})$, for the same non-deterministic choices made in both executions (over σ and σ'), Υ will also evaluate to $\texttt{false}$ for an execution on σ' — recall that $(HP(\xi_{\ddot{A}}) \cup CL(\xi_{\ddot{A}}))$ cannot overlap with $Zlv_U(\xi_{\ddot{A}})$, and the latter includes the interval $[v]_w$. All other state elements observed in the other instructions of (ALLOCV) are identical in both σ, σ' and $sim(\sigma, \sigma')$ will hold at the end of error-free executions.

- ($\text{LOAD}_{\ddot{A}}$) and ($\text{STORE}_{\ddot{A}}$). The overlap checks, $\texttt{ov}([p]_w, (\Sigma^{Z_l|^v}_{\ddot{A}}) \setminus (\Sigma^{F}_{\ddot{A}} \cup [\texttt{esp}, \boxed{\texttt{cs}_e}]))$

for (Load$_{\ddot{A}}$) and $\texttt{ov}([p]_w, (\Sigma_{\ddot{A}}^{Z_l}|^v) \setminus (\Sigma_{\ddot{A}}^{F_w} \cup [\texttt{esp}, \boxed{\texttt{cs}_e}]))$ for (Store$_{\ddot{A}}$), in the modified semantics of (Load$_{\ddot{A}}$) and (Store$_{\ddot{A}}$) will evaluate to `false` for σ due to the assumption of error-free execution. As these checks do not refer to the potentially modified regions $\{hp, cl, cs, \texttt{free}\}$, σ' must also evaluate the check to `false` (for the same sequence of non-deterministic choices). Notice that this reasoning relies on the safety-relaxed semantics, and would not hold on the original semantics. All other state elements observed in the other instructions of (Load$_{\ddot{A}}$) and (Store$_{\ddot{A}}$) are identical in both σ, σ' and $sim(\sigma, \sigma')$ will hold at the end of error-free executions.

□

Recall that the Dynamo algorithm populates the deterministic choice map $\mathcal{D}_X$ such that the result of the *choose* instruction ($\theta(\texttt{i}_{32})$) for α_b in an `alloc` instruction in ξ_C matches the address v in an $\texttt{alloc}_{s,v}$ instruction in $\xi_{\ddot{A}}$ and the result of the *choose* instruction for memory contents ($\theta(\texttt{i}_{32} \rightarrow \texttt{i}_8)$) of the freshly allocated interval $[\alpha_b, \alpha_e]$ matches the memory contents of the interval $[v]_w$ (in the `alloc` and $\texttt{alloc}_{s,v}$ instructions respectively). We use this fact in the following theorem on the execution of $[\xi_C]^{e_X}_{\mathcal{D}_X}$.

Lemma 5.3.6 ($sim(\sigma, \sigma')$ is preserved for error-free execution across all non-I/O edges in $\mathcal{E}_C$)**.** *If a non-I/O edge $e_C \in \mathcal{E}_C$ in the path $[\xi_C]^{e_X}_{\mathcal{D}_X}$ is executed on both machine states σ and σ', and if $sim(\sigma, \sigma')$ holds before the execution, and if the execution on σ, with non-deterministic choices determinized by $\mathcal{D}_X$, completes without error, then, for the same sequence of non-deterministic choices, the execution on σ' completes without error and $sim(\sigma, \sigma')$ holds at the end of both error-free executions.*

Proof. For a non-I/O C instruction that does not refer to the $\{hp, cl, cs, \texttt{free}\}$ regions ((Op), (AssignConst), (Dealloc), (VaStartPtr), (CallV), (Call$_C$), (Ret$_C$), (RetV)), the execution will have identical behaviour on both σ and σ' as identical values will be observed in both σ and σ'. Thus, if an execution on σ' makes the same non-deterministic choice as the execution on σ, the execution on σ' will complete without error and $sim(\sigma, \sigma')$ will hold at the end of both executions.

We consider each remaining non-I/O instruction in C below:

- (Entry$_C$) Consider the overlap check $\Upsilon = \texttt{ov}(\Sigma_C^{hp}, \Sigma_C^{cl}, \ldots, i_C^g, \ldots, \Sigma_C^f, \ldots, i_C^y, \ldots, \Sigma_C^{\texttt{vrdc}})$ (due to $\neg\texttt{addrSetsAreWF}$). During an execution on σ, this condition must evaluate to `false`, as we assume an error-free execution on σ. For the same non-deterministic choices made in both executions (over σ and σ'), by the definitions

of $HP(\xi_{\ddot{A}})$ and $CL(\xi_{\ddot{A}})$, Υ will also evaluate to `false` for an execution on σ' — recall that $(HP(\xi_{\ddot{A}}) \cup CL(\xi_{\ddot{A}}))$ cannot overlap with other allocated regions (due to lemma 5.3.4). Further, because all other state elements observed during the execution of the non-I/O edges in (ENTRY$_C$) are identical in both σ and σ', $sim(\sigma, \sigma')$ will hold at the end of error-free executions.

- (ALLOC) Consider the negated subset check $\Upsilon = \neg([\alpha_b, \alpha_e] \subseteq \Sigma_C^{\texttt{free}})$ (due to $\neg\texttt{intrvlInSet}_a(\alpha_b, \alpha_e, \Sigma_C^{\texttt{free}})$). The execution on σ must evaluate Υ to `false` as we assume an error-free execution. By the definitions of $HP(\xi_{\ddot{A}})$ and $CL(\xi_{\ddot{A}})$, for the same non-deterministic choices made in both executions (over σ and σ'), Υ will also evaluate to `false` for an execution on σ' — recall that during execution on σ, the deterministic choice map $\mathcal{D}_X$ will be used for the non-deterministic choices of address α_b and memory $\pi_{[\alpha_b,\alpha_e]}(M_C)$ such that the freshly allocated interval $[\alpha_b, \alpha_e]$ matches (in both address and data) the allocated interval $[v]_w$ in an $\texttt{alloc}_{s,v}$ instruction in $\xi_{\ddot{A}}$; because the same $\mathcal{D}_X$ is used in both σ and σ' executions, Υ will also evaluate to `false` in σ'. All other state elements observed in the other instructions of (ALLOC) are identical in both σ, σ'.

- (LOAD$_C$) and (STORE$_C$). An $\texttt{accessIsSafeC}_{\tau,a}()$ check must evaluate to `true` for σ due to the assumption of error-free execution. Because the allocated space Σ_C^B can only be bigger in σ' (by lemma 5.3.2), the `accessIsSafeC` check will also evaluate to `true` for σ' (for the same sequence of non-deterministic choices). Further, for an execution on σ, the contents of the memory region $\pi_{\Sigma_{\sigma'}^{\{hp,cl\}} \setminus \Sigma_{\sigma}^{\{hp,cl\}}}(M_C^{\sigma})$ cannot be observed on an error-free path; and because all other state elements observed in (LOAD$_C$) and (STORE$_C$) are identical in both σ and σ', the contents of the memory region $\pi_{(\Sigma_{\sigma'}^{\{hp,cl\}} \setminus \Sigma_{\sigma}^{\{hp,cl\}})}(M_C^{\sigma'})$ will also remain unobserved during an execution on σ' (that uses the same sequence of non-deterministic choices as an execution on σ). All other state elements observed in the other instructions of (LOAD$_C$) and (STORE$_C$) are identical in both σ, σ'.

□

Lemma 5.3.7 ($sim(\sigma, \sigma')$ is preserved for error-free execution across $\xi_{\ddot{A}}; \xi_C$)**.** *Recall that $\xi_{\ddot{A}}$ contains only non-I/O instructions (by assumption). Thus, due to the (SingleIO) requirement, ξ_C also contains only non-I/O instructions.*

If $\xi_{\ddot{A}}$ is executed on machine states σ and σ', and if the execution of σ completes without error, then there exists a sequence of non-deterministic choices during the

execution of σ' such that the execution is error-free and $sim(\sigma, \sigma')$ holds at the end of both error-free executions.

Similarly, if ξ_C is next executed on machine states σ and σ', and if the execution of σ completes without error, then there exists a sequence of non-deterministic choices during the execution of σ' such that the execution is error-free and $sim(\sigma, \sigma')$ holds at the end of both error-free executions.

Proof. To show this, we execute the sequence of paths $(\xi_{\ddot{A}}; \xi_C)$ in lockstep on both σ and σ', i.e., in a single step, one instruction is executed on both states modifying the states in place. The proof proceeds by induction on the number of steps. The base case holds by assumption. For the inductive step, we rely on lemmas 5.3.5 and 5.3.6. □

Lemma 5.3.8 (σ and σ' execute the same path in $\ddot{A}$)**.** *If $\xi_{\ddot{A}}$ executes to completion on state σ, it will also execute to completion on σ'.*

Proof. By case analysis on all edge conditions in figs. 2.6 to 2.8, 2.10 and 2.11. For $\xi_{\ddot{A}} = n_{\ddot{A}} \twoheadrightarrow \mathcal{U}_{\ddot{A}}$ due to (Load$_{\ddot{A}}$) and (Store$_{\ddot{A}}$), the proof relies on the safety-relaxed semantics, and would not hold on the original semantics. □

Lemma 5.3.9 (σ and σ' execute the same non-$\mathcal{U}$ path in C)**.** *If ξ_C does not terminate in $\mathcal{U}_C$, and σ executes ξ_C to completion, then σ' will also execute ξ_C to completion.*

Proof. By case analysis on all edge conditions in figs. 2.4 and 2.5 with same arguments as used in lemma 5.3.6. □

Lemma 5.3.10 ($post(\sigma') \wedge sim(\sigma, \sigma') \Rightarrow post(\sigma)$ holds for an error-free node $(n^t_{\ddot{A}}, n^t_C)$)**.** *For two states σ and σ' at node $(n^t_{\ddot{A}}, n^t_C)$, where $n^t_{\ddot{A}}$ and n^t_C are error-free nodes, $post(\sigma') \wedge sim(\sigma, \sigma') \Rightarrow post(\sigma)$ holds.*

Proof. The *post* condition that may appear in a Hoare triple proof obligation generated by Dynamo can be one of the following:

- (CoverageC) where $post = \bigvee_{1 \le j \le m} pathcond([\xi^j_C]^{e^j_X}_{\mathcal{D}_X})$ for $e^j_X = (n_X \xrightarrow{\xi_{\ddot{A}}; \xi^j_C} (n^t_{\ddot{A}}, n^t_C)) \in \mathcal{E}_X$ $(1 \le j \le m)$.
- (Inductive) where *post* is one of the predicate shapes listed in fig. 4.2. Note that the `MemEq` shape in fig. 4.2 represents the proof obligation for the (MemEq) requirement.

- (Equivalence) where *post* is either $\Omega_{\ddot{A}} = \Omega_C$ or $T_{\ddot{A}} =_{st} T_C$. I/O free paths do not mutate world states so $\Omega_{\ddot{A}} = \Omega_C$ holds trivially for these cases. Further, the only I/O free paths that may modify trace must contain `halt` instruction, appearing as the last edge of the sequence. As the generated trace event for `halt` does not observe any procedure state variable, we ignore this case.

- (MAC) where *post* checks the address of each memory access in Ä against the addresses of a set of memory accesses in C for equality. Also, (MAC) checks if a memory access overlaps with address regions $\Sigma_{\ddot{A}}^{G \cup F} \cup [\texttt{esp}, \texttt{stk}_e]$ or $\Sigma_{\ddot{A}}^{G_w \cup F_w} \cup [\texttt{esp}, \texttt{stk}_e]$.

Case: When post is one of the predicate shapes in fig. 4.2 or is a (MAC) proof obligation.

- The predicate shapes `affine`, `ineqC`, `ineq`, `spOrd`, `zEmpty`, `spzBd`, `spzBd'`, and a (MAC) proof obligation do not involve operations over address sets $\{hp, cl, cs, \texttt{free}\}$ or memory operations in the updated region $\Sigma_{\sigma'}^{\{hp,cl\}} \setminus \Sigma_{\sigma'}^{\{hp,cl\}}$. Thus, $post(\sigma') \wedge sim(\sigma, \sigma') \Rightarrow post(\sigma)$ holds in this case.

- Consider the case when *post* is `AllocEq`. Due to (Equivalence), `AllocEq` is guaranteed to be in *pre* and therefore $\Sigma_{\ddot{A}}^{hp} = \Sigma_C^{hp}$ and $\Sigma_{\ddot{A}}^{cl} = \Sigma_C^{cl}$ hold for σ'. Due to $sim(\sigma, \sigma')$, σ and σ' agree on the remaining state elements, including the address sets for each region $z \in Z$. Thus, $post(\sigma') \wedge sim(\sigma, \sigma') \Rightarrow post(\sigma)$ holds in this case.

- Consider the case when *post* is `MemEq`. $sim(\sigma, \sigma')$ ensures that the address sets of regions $\{hp, cl\}$ in σ are a subset of respective address sets in σ'. Further, due to $sim(\sigma, \sigma')$, the memory states of A in σ and σ' are identical, $M_{\ddot{A}}^{\sigma} = M_{\ddot{A}}^{\sigma'}$, and the memory states of C in σ and σ' disagree only over the updated (expanded) address sets, $M_C^{\sigma} =_{\texttt{comp}(\Sigma_{\sigma'}^{\{hp,cl\}} \setminus \Sigma_{\sigma}^{\{hp,cl\}})} M_C^{\sigma'}$. Because the allocated regions in σ do not belong to these (expanded) addresses, $post(\sigma)$ follows from $post(\sigma')$.

Case: When post is a proof obligation for (CoverageC). In this case, *post* must be of the form $\bigvee_{1 \le j \le m} pathcond([\xi_C^j]_{\mathcal{D}_X}^{e_X^j})$ for $e_X^j = ((n_{\ddot{A}}, n_C) \xrightarrow{\xi_{\ddot{A}}; \xi_C^j} (n_{\ddot{A}}^t, n_C^t)) \in \mathcal{E}_X$ $(1 \le j \le m)$. The edge conditions in C are independent of the regions $\{hp, cl, cs, \texttt{free}\}$, except for (LoadC) and (StoreC). If the edge condition is independent of these address regions, then $post(\sigma)$ follows trivially from $post(\sigma')$. Consider the other case now: for an error-free node, the $\mathcal{U}$-maximal set of paths $\{\xi_C^1, \ldots, \xi_C^m\}$ includes both the paths that evaluate $\texttt{accessIsSafeC}_{\tau,a}$ to `true` and `false` respectively. Thus, even in this case,

$post(\sigma)$ holds if $post(\sigma')$ holds.

□

Lemma 5.3.11 ($post(\sigma') \Rightarrow post(\sigma)$ for $n^t_{\ddot{A}} = \mathscr{W}_{\ddot{A}}$). *For two states σ and σ' at node $(\mathscr{W}_{\ddot{A}}, n^t_C)$, $post(\sigma') \Rightarrow post(\sigma)$ holds.*

Proof. The *post* condition of this type may appear in a Hoare triple proof obligation generated by DYNAMO for one of the following:

- (CoverageC) where $post = \bigvee_{1 \le j \le m} pathcond([\xi^j_C]^{e^j_X}_{\mathcal{D}_X})$ for $e^j_X = (n_X \xrightarrow{\xi_{\ddot{A}};\xi^j_C} (n^t_{\ddot{A}}, n^t_C)) \in \mathcal{E}_X$ $(1 \le j \le m)$.
- (MAC) where *post* checks the address of each memory access in $\ddot{A}$ against the addresses of a set of memory accesses in C for equality. Also, (MAC) checks if a memory access overlaps with address regions $\Sigma^{G \cup F}_{\ddot{A}} \cup [\texttt{esp}, \boxed{\texttt{stk}_e}]$ or $\Sigma^{G_w \cup F_w}_{\ddot{A}} \cup [\texttt{esp}, \boxed{\texttt{stk}_e}]$.

The proof arguments for both these cases are identical to the ones made in the proof for lemma 5.3.10. □

Lemma 5.3.12 ($post(\sigma') \Rightarrow post(\sigma)$ for $n^t_{\ddot{A}} = \mathscr{U}_{\ddot{A}}$). *For two states σ and σ' at node $(\mathscr{U}_{\ddot{A}}, n^t_C)$, $post(\sigma') \Rightarrow post(\sigma)$ holds.*

Proof. The *post* condition of this type may appear in only one type of proof obligation generated by DYNAMO:

- (CoverageC) where $post = \bigvee_{1 \le j \le m} pathcond([\xi^j_C]^{e^j_X}_{\mathcal{D}_X})$ for $e^j_X = (n_X \xrightarrow{\xi_{\ddot{A}};\xi^j_C} (\mathscr{U}_{\ddot{A}}, n^t_C)) \in \mathcal{E}_X$ $(1 \le j \le m)$.

Let the (CoverageC) proof obligation be $\{\phi_{n_X}\}(\xi_{\ddot{A}}; \epsilon)\{\bigvee_{1 \le j \le m} pathcond([\xi^j_C]^{e^j_X}_{\mathcal{D}_X})\}$. Due to (Safety), each path ξ^j_C must end at $\mathscr{U}_C$.

From the semantics in figs. 2.4 and 2.5, if the path condition for ξ^j_C evaluates to `true` on σ' (for some j), it must also evaluate to `true` on σ — in other words, whenever σ' transitions to $\mathscr{U}_C$, σ is guaranteed to transition to $\mathscr{U}_C$. This is because the edge conditions in C will evaluate either identically on σ and σ' (due to $\{\xi^1_C, \ldots, \xi^m_C\}$ being a $\mathscr{U}$-maximal set), or in the case of $\neg\texttt{accessIsSafeC}_{\tau,a}()$, the edge condition will evaluate to `true` on σ if it evaluates to `true` on σ'.

Thus, if $post(\sigma')$ evaluates to `true`, $post(\sigma)$ also evaluates to `true`. □

Proof for (⇐). Follows from lemmas 5.3.7 to 5.3.12. □

Proof of theorem 5.3.1. Follows from (⇒) and (⇐). □

5.4 Semantics with Simpler SMT Encoding for *stk* Region of Ä

In our implementation, we use a slightly revised semantics for Ä for a more efficient SMT encoding. We define a new region stk^+ such that stk^+ is large enough to contain the stk region at every point during Ä's execution. A key property that stk^+ satisfies is that $\alpha \in \Sigma_{Ä}^{stk}$ can be rewritten in terms of $\alpha \in \Sigma_{Ä}^{stk^+}$ while $\alpha \in \Sigma_{Ä}^{stk^+}$ itself having simpler SMT encoding than $\Sigma_{Ä}^{stk}$.

(Entry$_{Ä}$'), (Op-esp"), (AllocS"), and (DeallocS") in fig. 5.1 present the new semantics for procedure-entry, a stackpointer updating instruction, and $\texttt{alloc}_s$ and $\texttt{dealloc}_s$ instructions respectively. As with previous presentations, we only show the changes (with respect to figs. 2.6 to 2.8 and 2.11) with appropriate context in fig. 5.1; additions are highlighted and deletions are ~~canceled.~~

(Entry$_{Ä}$') initializes the address set $\Sigma_{Ä}^{stk^+}$ such that $\Sigma_{Ä}^{stk} \subseteq \Sigma_{Ä}^{stk^+}$ holds and $\Sigma_{Ä}^{stk^+}$ does not overlap with other allocated regions in Ä. The lower bound of $\Sigma_{Ä}^{stk^+}$ is defined by a ghost variable $\texttt{stk}_b$ which is required to be below `esp` at all times for an error-free execution. Intuitively, $\texttt{stk}_b$ corresponds to the lowest stackpointer value seen during the execution of Ä[5].

(Op-esp") shows the updated semantics for a stackpointer (`esp`) updating instruction. Our modification is limited to the address set argument (third argument) of the $\{\mathscr{W},\mathscr{U}\}$-guarding $\neg\texttt{intrvlInSet}(\ldots)$ predicate in both stack allocation and deallocation cases of the instruction. $\texttt{intrvlInSet}(\alpha_b, \alpha_e, i)$ returns `false` if the interval $[\alpha_b, \alpha_e]$ does not completely lie within the address set i. In the stack allocation case (under `if (isPush(...))`), the address set argument $\Sigma_{Ä}^{\texttt{free}} \cup ((\Sigma_{Ä}^{cv} \cup (\Sigma_{Ä}^{Z_l}|^v)) \setminus \Sigma_{Ä}^{F})$ to $\texttt{intrvlInSet}(\ldots)$ is replaced with $\Sigma_{Ä}^{stk^+}$. This ensures that the stk^+ region is always big enough to accommodate a stack push and, consequently, imply that $\Sigma_{Ä}^{stk} \subseteq \Sigma_{Ä}^{stk^+}$

[5]Note the similarity with $SP_{min}(\xi_{Ä})$ from section 5.3.2

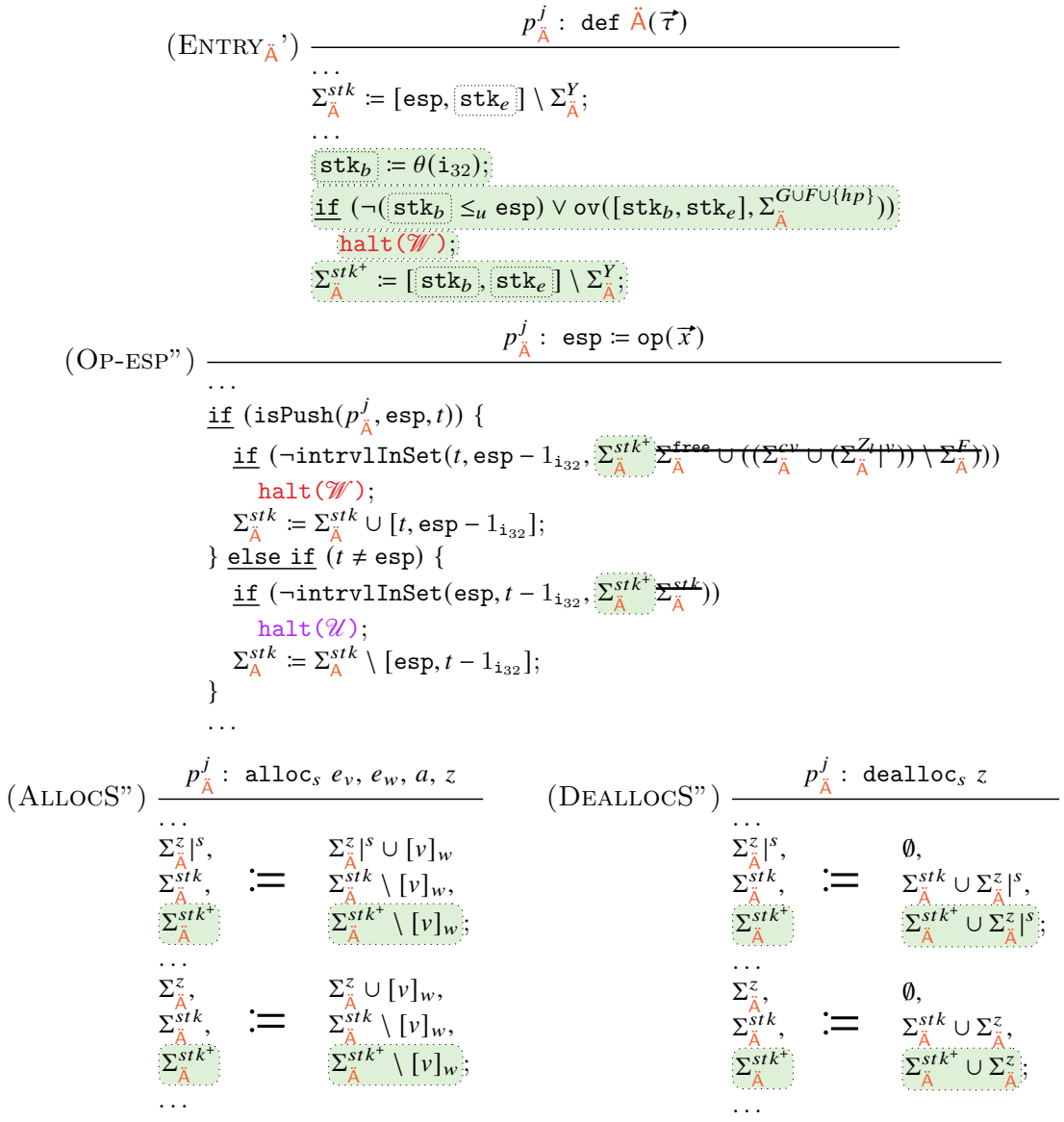


Figure 5.1: Revised translation rules for the new stk^+-based semantics for Ä.

continues to hold after a stack allocation. Similarly, in the stack deallocation case (under elseif ($t \neq$ esp)), $\Sigma^{stk}_Ä$ is replaced with $\Sigma^{stk^+}_Ä$. Our modifications are carefully designed so that this modified check (for stack deallocation case) will evaluate identically to the original check in an execution of Ä.

(ALLOCS") and (DEALLOCS") shows the updated semantics for alloc$_s$ and dealloc$_s$ instructions. In both cases, $\Sigma^{stk^+}_Ä$ is updated identically to $\Sigma^{stk}_Ä$ for a local (de)allocation from(to) stack.

Theorem 5.4.1. *The following property holds at every error-free, non-entry node* $n_{\ddot{A}} \in \mathcal{N}_{\ddot{A}}^{\not{UW}}$ *in an execution of* Ä *with the semantics presented in fig. 5.1 (and semantics presented earlier in figs. 2.5, 2.6, 2.8, 2.10 and 2.11).*

$$\alpha \in \Sigma_{\ddot{A}}^{stk} \Leftrightarrow \alpha \in \Sigma_{\ddot{A}}^{stk^+} \wedge (\alpha \geq_u \texttt{esp})$$

Proof sketch: By induction on the number of transitions executed in Ä with the base case defined by the first transition out of (ENTRY$_{\ddot{A}}$') in fig. 5.1. □

Using the above property, the SMT encoding of a verification condition over Ä is rewritten to replace references to $\Sigma_{\ddot{A}}^{stk}$ with references to $\Sigma_{\ddot{A}}^{stk^+}$. All assignments to $\Sigma_{\ddot{A}}^{stk}$ are made vacuous and only $\Sigma_{\ddot{A}}^{stk^+}$ is tracked in the allocation state of Ä. Because $\Sigma_{\ddot{A}}^{stk^+}$ is not updated due to a stackpointer updating instruction, the resulting SMT expressions are simpler.

Chapter 6

Evaluation

This chapter discusses the implementation specifics and evaluation of a prototype tool based on the DYNAMO algorithm described in the previous chapters.

6.1 Implementation of DYNAMO

6.1.1 System components

Figure 6.1 shows the high-level component blocks and the flowchart of our tool, also named DYNAMO. The two programs, C source code and its (potentially optimized) x86 assembly, are translated to their respective graph representations before they are passed as input to the tool. The C source is first translated to $\texttt{LLVM}_d$ (which is based on unoptimized LLVM IR; see section 2.1.1) using a modified Clang/LLVM [48]. The explicit `dealloc` instructions, present only in $\texttt{LLVM}_d$, are also added during this translation using the `stacksave` and `stackrestore` intrinsics — these intrinsics are generated for the purpose of stack related code generation, we use them for deriving scope information of allocated locals. The generated $\texttt{LLVM}_d$ program is then lowered to its graph representation C (described in section 2.3.1) and a may-point-to analysis is performed (described in section 4.1). The x86 assembly is similarly translated to its graph representation A (described in section 2.3.2); the translation uses the safety-relaxed semantics (described in section 3.5). Both C and A use the callers' virtual smallest semantics (described in section 3.4).

The various components blocks of the tool are shown in fig. 6.1. The tool interfaces

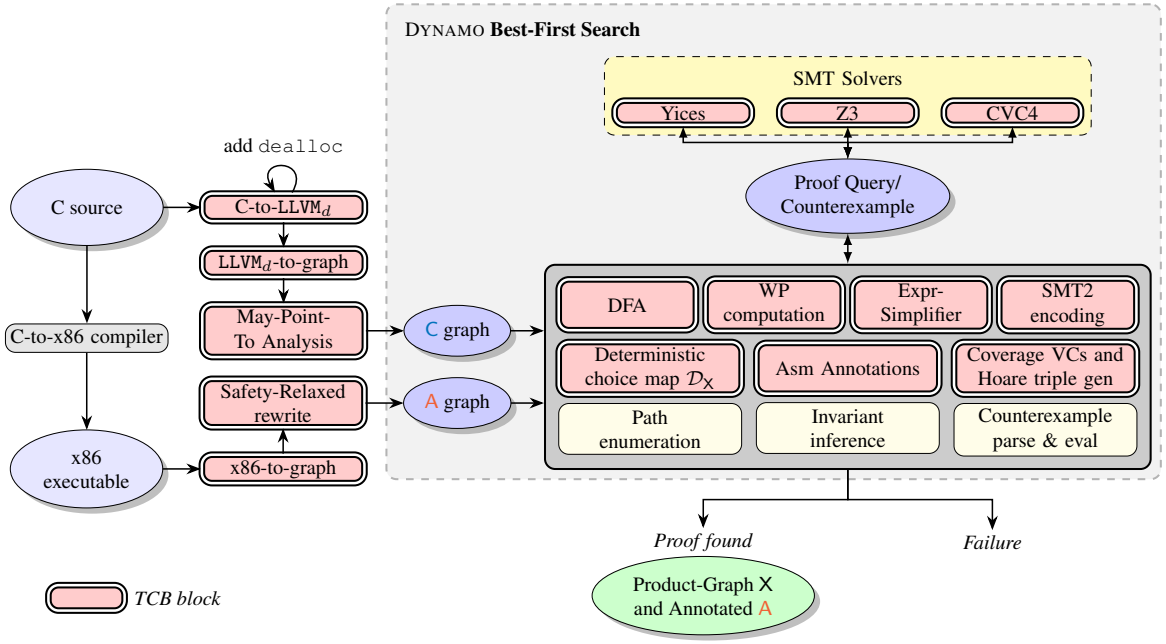


Figure 6.1: High-level components of DYNAMO implementation. Trusted Code Base (TCB) blocks have double border and a red background.

with third-party SMT solvers ("SMT Solvers" in the diagram) for discharging its proof obligations. The "SMT2 encoding" component encodes a Hoare triple into SMT-LIB2 format [7] that is used as a uniform input format for the SMT solvers. A model or counterexample returned by a solver is parsed, evaluated, and translated for guiding the correlation and invariant inference ("Counterexample parse & eval"). The other prominent components of the system shown in the diagram are: a common data-flow analysis (DFA) framework ("DFA") a weakest-precondition computation module ("WP computation"), a syntactic expression simplifier ("Expr-Simplifier"), and a verification condition generator ("Coverage VCs and Hoare triple gen"). The modules for the deterministic choice map (described in section 3.2) and assembly annotations are shown separately as well ("Deterministic choice map $\mathcal{D}_X$" and "Asm Annotations").

Trusted Code Base

The soundness of a verification effort is critically dependent on the correctness of the *trusted computing base* (TCB) of the verifier — in fig. 6.1 the blocks belonging to the TCB are marked with double borders and have a red background. An error in the TCB may result in a soundness error.

DYNAMO is around 400K Source Lines of Code (SLOC) in C/C++ that has been in development for over a decade and used in prior work [4, 13, 42]. Out of the 400K, the TCB is around 70K SLOC. Within the TCB, around 20K SLOC is due to the expression handling and simplification logic, less than 10K for SMT encoding, less than 10K SLOC for the graph representation and the weakest-precondition logic and less than 1K SLOC for a common data-flow analysis framework. The x86-to-graph translation is around 18K SLOC of C code (for disassembly) and 5K SLOC of OCaml code (for logic encoding), and the IR-to-graph translation, including the addition of `dealloc` and the may-point-to analysis (section 2.3), is around 6K SLOC of C++ code. We rely on the Clang framework for the C-to-IR translation — one can imagine replacing this with a verified frontend, such as CompCert's [30]. The modeling of the deterministic choice map, generation of Hoare triple and coverage verification conditions is relatively simple (less than 1K SLOC total).

6.1.2 Discharging Proof Obligations

Recall that we use quantified logic over bitvectors, arrays, and uninterpreted functions (AUFBV in SMT-LIB [7]) in our SMT encoding. For procedures without local (de)allocations and procedure calls, it is possible to use a quantifier-free encoding by modeling the stack as a separate memory array. Using a lightweight analysis on the A graph, each memory access may be labeled as either to stack region or *non-stack* (other than stack) regions. Utilizing the full-interval encoding (section 5.3), the various address set relations can then be rewritten in quantifier-free logic. Prior work [39, 42] has used similar approach for achieving quantifier-free encoding.

Use of multiple solvers: DYNAMO uses four SMT solvers running in parallel for discharging proof obligations: `Z3-4.8.7`, `Z3-4.8.14`, `Yices2-45e38fc`, and `CVC4-1.7`. Though `Yices2` does not support quantified logic (and cannot be used in the general setting), we use it fruitfully in the quantifier-free setting described in previous paragraph. For instance, in our experiments, we observed that it was consistently faster than the others. Because both our best-first search and invariant inference algorithm may potentially converge faster with a larger set of counterexamples, we invoke two versions of the `Z3` SMT solver: `Z3-4.8.7` and `Z3-4.8.14`. This decision was guided by an empirical observation where we noted that the two solvers have different performance characteristics and produce *sufficiently different* counterexamples that contribute to faster convergence of the invariant inference algorithm [42].

Handling difficult queries: As an optimization, our implementation makes use of the *query-decomposition* technique and simplification rewrites described in [18] for discharging queries difficult for the SMT solvers. We invoke query-decomposition only after the SMT solvers fail to discharge the proof query within a specified timeout period (we provide the timeout period used in each experiment in section 6.2).

6.1.3 Pseudo-register allocation in $\texttt{LLVM}_d$

Before checking refinement, if the address of a local variable l is never taken in C, we transform C to register-allocate l using LLVM's `mem2reg` pass. This reduces the proof effort, at the cost of having to trust the pseudo-register allocation logic. However, `mem2reg` does not register-allocate local arrays and aggregates (`structs`) in $\texttt{LLVM}_d$, even though an optimizing compiler may register-allocate them in assembly — virtual allocations help validate such translations. Another case where virtual allocations are helpful is described in section 6.2.4 where a `va_list` variable (used in variadic procedures) is not (pseudo-)register allocated in C by `mem2reg` but is register-allocated in optimized assembly. Thus, after `mem2reg`, virtual allocations are only required for validating register-allocation of arrays, aggregates, and `va_list` variables.

6.1.4 Instrumentation of Clang/LLVM for generating annotation hints

We instrumented the Clang/LLVM v12.0.0 compiler to generate annotation hints for the whitebox setting. The instrumentation, which required only a couple of lines of code, prints the constant offset (with respect to the stackpointer at entry of the procedure) of the allocated stack slots for *non-VLA* allocations — for VLA allocations we still rely on blackbox enumeration. These offsets do not map to a named local in C and are only used as additional address options for an $\texttt{alloc}_s$ annotation (section 4.1.3).

6.2 Experiments

We design our experiments to validate the capabilities and explore the limitations of the Dynamo algorithm in our prototype tool. In particular, the experiments explore the following aspects of the tool (and the algorithm):

1. Handling of different constructs available for local allocation in the C language and

common extensions (section 6.2.1).

2. Performance of the full-interval and partial-interval SMT encoding in comparison to the naive full-array SMT encoding (section 6.2.1).

3. Overhead of identifying the required annotation for modeling local (de)allocations (section 6.2.2).

4. Performance of the tool on a real-world program (section 6.2.3).

6.2.1 Evaluating efficacy of DYNAMO

We first evaluate the efficacy of DYNAMO to handle the diverse programming patterns seen with local allocations in C. Table 6.1 lists the programming patterns we test in this experiment. Each programming pattern corresponds to one benchmark (i.e., a C procedure) except `vsl`N and `vil`N where we substitute N with 1, 2, 3 to obtain `vsl1`, `vsl2`, `vsl3` and `vil1`, `vil2`, `vil3` respectively in each case. The programming patterns we include in this experiment have:

- use of address-taken local variables and parameters in benchmarks `ats`, `atss`, `atps`, `atpss`, and `atc`.
- use of constant- and variable-length arrays in benchmarks `ata`, `atail`, `fib`, `vin`, `vcu`, `vsl1`, `vsl2`, and `vsl3`.
- variadic procedures in benchmarks `vwl`, `vwc`, `mp`, and `ms`.
- use of variable-length arrays (VLAs) allocated inside loops in benchmarks `vil1`, `vil2`, `vil3`, `vilcc`, and `vilce`.
- unconditional and conditional use of `alloca()` in `as`, `ac`, and `ams`.
- use of `alloca()` to create a linked-list in `all`.
- mixed use of constant-sized and variable-length array in `rod`.

There are total 27 benchmarks and we use three compilers, Clang/LLVM v12.0.0, GCC v8.4.0, and ICC v2021.8.0, to generate 32-bit x86 executables at `O3` optimization level to create 81 procedure-pairs. We manually disable interprocedural and unrolling/vectorization optimizations in each invocation using the compiler's source-level `pragma`s and/or command-line flags[1]. A refinement check is performed for each of the 81 procedure-pairs. We use an unroll-factor of one $\mu = 1$ for all benchmarks except `all` for GCC and ICC, and `fib` for Clang/LLVM where we use an unroll-factor of two $\mu = 2$. Each refinement

[1]The full command-line used for generating the assembly in each case is provided in section B.1.

#	Name	Programming pattern
1	ats	Address-taken local scalar — `int ats() { int ret; foo(&ret); return ret; }`
2	atss	Address-taken local struct — `int atss() { struct Point p1, p2; foo(&p1, &p2); return ...; }`
3	atps	Address-taken scalar parameter — `int atps(int a) { char x; scanf("%c %d", &x, &a); return ...; }`
4	atpss	Address-taken struct parameter — `int atpss(struct Point p1, ...) { ...; scanf(..., &p1.x); ... }`
5	atc	Address taken conditionally — `int atc(int* p) { int x; if (!p) p = &x; foo(p); return *p }`
6	ata	Local array — `int ata() { char ret[8]; foo(ret); return bar(ret, 0, 16); }`
7	atail	Local array alloc. in loop — `int atail(..){..for(..){ char a[4096]; f(a..); b(a..);...}...}`
8	fib	Program from fig. 2.1
9	vin	VLAs in nested scopes — `void vin(int n, int m){int v1[n]; { int v2[m]; foo(v2); } bar(v1); }`
10	vcu	VLA conditional use — `int vcu(int n,int k){ int a[n]; if (...) { /*rd/wr to a*/}...}`
11	vsl$N$	N VLA(s) — `int vsl`N`(..){ .. int v1[n], ... v`N`[n]; for(i=0;i<n;++i) { v1[i]=..a[i]..; .... v`N`[i]=..a[i]..; } return foo`N`(...); }`
12	vil$N$	N VLA(s) in a loop — `int vil`N`(..){..for(i=1;i<n;++i) { int v1[4*i], ... v`N`[4*i]; foo`N`(..); ..}`
13	vilcc	VLA in loop with continue — `int vilcc(..){..while(i<n){ char v[i];...if(..) continue;..}..}`
14	vilce	VLA in loop with break — `int vilce(..){..while(i<n){ char v[i];...if(..) break;..}..}`
15	vwl	Variadic procedure — `int vwl(int n,...){ va_list a; va_start(a, n); for(..){/* va_arg(a,int) */}..}`
16	vwc	Variadic procedure using va_copy() — `vwc(int n,...) { va_list a, b; va_start(a,n); va_copy(b,a); for(...){ /* va_arg(b) */ } va_copy(b,a); foo(b); ...}`
17	mp	minprintf procedure adapted from K&R [24]
18	ms	minscanf procedure similar to minprintf
19	as	alloca() — `int as(int n){...int* p=alloca(n*sizeof(n)); for(...){/*write to p*/}...}`
20	ac	alloca() conditional use — `int ac(char*a) {..if (!a) a=alloca(n); for(...)/*r/w to a*/}`
21	all	alloca() in a loop to form a linked list — `all() { ...hd=NULL; for(...){ ...n=alloca(..); n->nxt=hd; hd=n; } while(...){ /* traverse the list starting at hd */ }}`
22	ams	alloca() & malloc() — `int ams(..){..if(..){p=alloca(..);}else{p=malloc(..);}/* r/w to p ..*/...}`
23	rod	A local char array initialized using a string and a VLA and a for loop — Available in ??.

Table 6.1: Benchmarks and their programming patterns. N in vsl$N$ and vil$N$ is substituted to obtain vsl1, vsl2, vsl3 and vil1, vil2, vil3 respectively. Full program listings available in chapter C.

check is run with a refinement check timeout of two hours and a per SMT query timeout of 120 seconds, i.e., the refinement check is terminated automatically after two hours, resulting in a refinement failure, and, similarly, an SMT solver spawned for discharging an SMT query is automatically terminated after 120 seconds, yielding a "timeout" result which is interpreted as a failure in the discharge of the corresponding proof obligation — a failed proof obligation may eventually result in refinement failure.

Figure 6.2 shows graph of the refinement check run times for the 81 procedure-pairs. The X-axis lists the benchmarks and the Y-axis represents the total time taken in seconds (using a log scale) for a refinement check. To study the performance implications, we

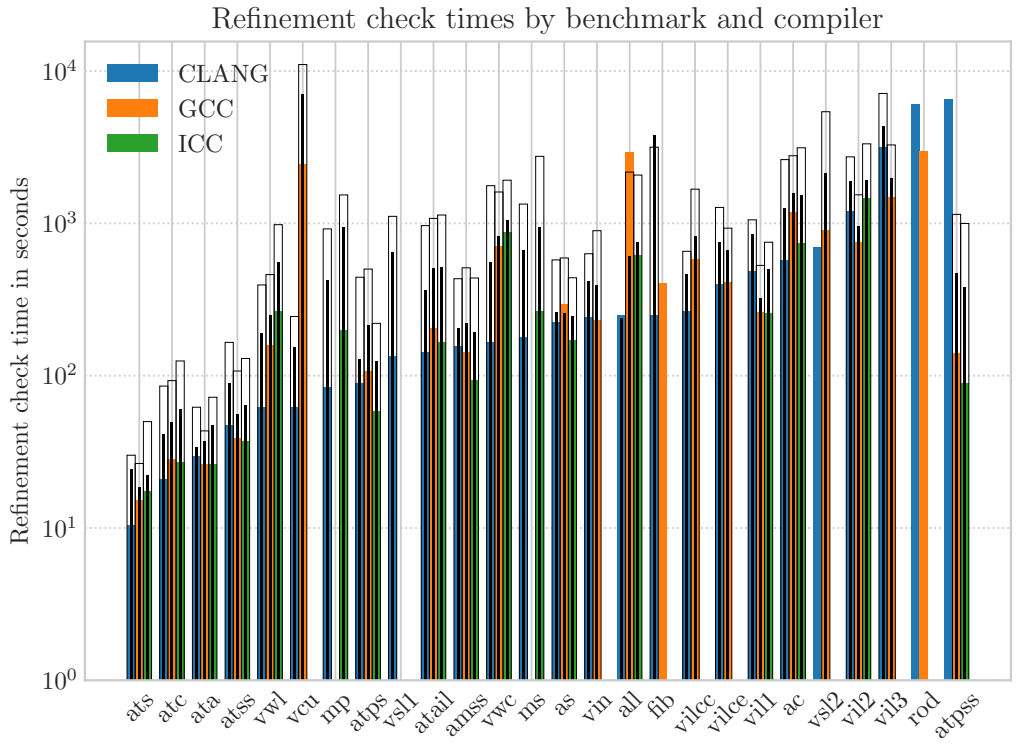


Figure 6.2: Comparison of running times of procedures in table 6.1 with full-interval (filled bars), partial-interval (thick black lines), and full-array (empty bars) encoding. Y-axis is logarithmically scaled.

run a check with all three encodings for these benchmarks:

- A filled bar ( ) represents the time taken with, if it was possible to use, a full-interval encoding.
- A thick black line centered on a bar ( ) represents the time taken with a *forced* partial-interval encoding, i.e., partial-interval encoding is forced even if a full-interval encoding is possible.
- An empty bar ( ) represents represents the time taken with a *forced* full-array encoding, i.e., a full-array encoding is forced even if a full-interval or a partial-interval encoding is possible.
- A missing bar or a missing thick line (for partial-interval encoding) represents a failure in proof search with respective encoding.

Of the 81 procedure pairs, our tool is able to find a refinement proof for 66, 61, and

60 procedure-pairs while using full-interval, partial-interval, and full-array encodings respectively. For benchmarks where a refinement check succeeds for all encodings, the full-interval encoding performs 2.07 and 3.88 times faster on average (for each compiler) than the partial-interval and full-array encodings respectively.

Encoding	# of successful refinement checks			
	Clang/LLVM	GCC	ICC	**Total**
Full-interval encoding	26	23	17	66
Partial-interval encoding	23	21	17	61
Full-array encoding	22	21	17	60

Figure 6.3: Summary of refinement check results for the programs in table 6.1.

Figure 6.3 summarizes the results of this experiment, showing the number of successful refinement checks for each compiler and encoding pair. DYNAMO is unable to find refinement a proof in 15 cases for the full-interval encoding. We analyze these failures in detail in section 6.2.4. A failure particular to partial-interval or full-array encoding (i.e., the refinement check succeeded with a full-interval encoding) is either due to SMT solver timeout causing failure of a crucial proof obligation or a refinement check timeout in that invocation.

Two benchmarks, `vilcc` and `vilce`, require multiple $\texttt{dealloc}_s$ instructions to be added to A for a single `dealloc` in C. An $\texttt{alloc}_v$ annotation is required for the '`va_list a`' variable in the GCC and ICC compilations of `vwl` and `vwc` (see table 6.1) — while GCC and ICC register-allocate `a`, it is allocated in memory using `alloc` in $\texttt{LLVM}_d$ even after `mem2reg` (section 6.1.3). The time spent in constructing the correct product graph forms around 70-80% of the total search time.

6.2.2 Evaluating modeling cost of local allocations

Our next experiment evaluates the cost of modeling local allocations. Towards this, we run DYNAMO on the TSVC suite of vectorization benchmarks with arrays and loops [33]. The benchmarks in this suite are used to evaluate the vectorization capabilities of optimizing compilers and typically have C functions with (nested) loops and array accesses. A variation of this suite, where the floating-point types are replaced with integer types, was used in prior work on translation validation [8, 17] and we adopt the same modified version for our use — we select 25 procedures (as used in [17]) for our experiments.

```
int a[4000]; // global array -- 'a' is locally allocated in 'locals' version
int b[4000];
int s122(int n1, int n3) {
    int j, k;
    j = 1;
    k = 0;
    // int a[4000]; // this is uncommented in 'locals' version
    init_local1(a); // this call initializes 'a' in 'locals' version
    for (int i = n1-1; i < 4000; i += n3) {
        k += j;
        a[i] += b[4000 - k];
    }
    print_local1(a); // this call observes (outputs) 'a' in 'locals' version
    return 0;
}
```

Figure 6.4: Procedure `s122` from '`globals`' version of (modified) TSVC suite.

We create two versions of this suite for our experiments:

1. '`globals`' where global variables are used for storing the output array values.
2. '`locals`' where local array variables are used for storing the output array values.

In both versions, a procedure call is added at the end of the procedure body to print the contents of the modified output array variables. Further, if the output array variable is being read before being assigned, we add a procedure call before the read to initialize the variable. Figure 6.4 shows an example.

We use Clang/LLVM v12.0.0 at optimization level `O3` with vectorization enabled using `-msse 4.2` flag for compiling the 25 C procedures in each version. The compiler performs the same vectorizing transformations on both versions. Unlike `globals`, the benchmarks in `locals` version additionally require the automatic identification of required annotation. We use an unroll-factor of 64, a global timeout of two hours, and a per SMT query timeout of one minute for each run of Dynamo on the 50 procedure-pairs.

Figure 6.5 shows the execution times of Dynamo for these two versions of the TSVC benchmark. Dynamo is able to successfully validate these compilations. Similar to fig. 6.2, we show execution times for both full-interval and *forced* partial-interval encodings for the '`locals`' benchmarks using filled bars and empty bars respectively (we omit the *forced* full-array encoding numbers in this case). Compared to `globals`,

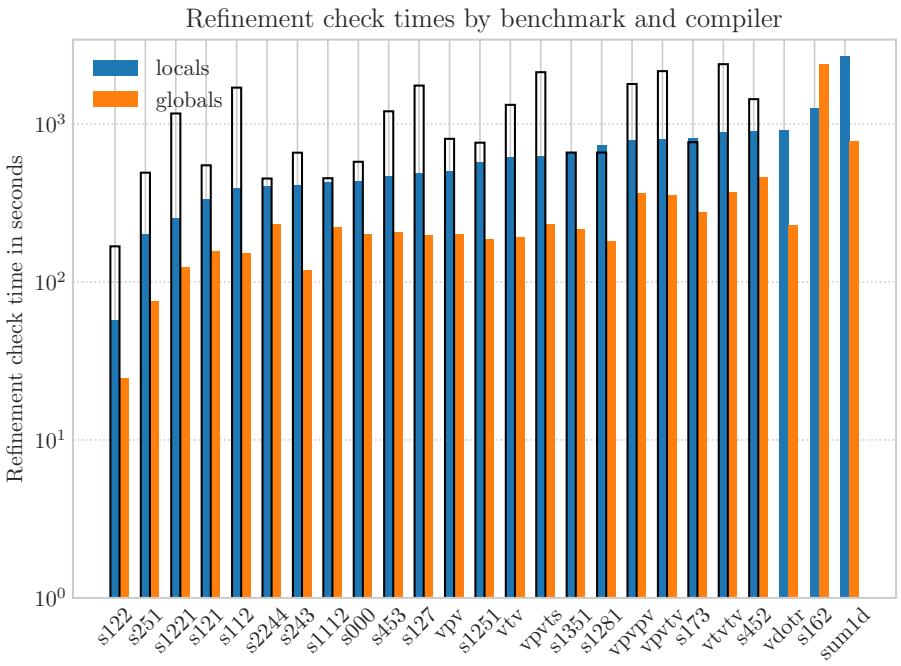


Figure 6.5: Comparison of running times of TSVC benchmarks with exactly same code modulo allocation strategy. Y-axis is logarithmically scaled.

refinement checks are 2.5x slower for `locals` (on average) due to the extra overhead of identifying the required annotations. The full-interval encoding is 1.9 times faster (on average) than partial-interval encoding. This relative speed-up of full-interval encoding over partial-interval encoding is similar to the number obtained in previous experiment (section 6.2.1). With a forced partial-interval encoding, Dynamo fails to validate 3 (out of 25) benchmarks within the assigned time budget.

6.2.3 Evaluating Dynamo on a real-world program

Our last experiment is on SPEC CPU2000's `bzip2`[20] program compiled using Clang/LLVM v12.0.0 at three optimization levels: `O1`, `O2`, and `O1-`. `O1-` is a custom optimization level configured by us that enables all optimizations at `O1` *except* the following.

(a) merging of multiple procedure calls on different paths into a single call,
(b) early-CSE (common subexpression elimination),
(c) loop-invariant code motion at both LLVM IR and Machine IR,
(d) dead-argument elimination,

(e) interprocedural sparse conditional constant propagation,
(f) dead-code elimination of procedure calls.

These passes were chosen based on our knowledge of the limitations of our invariant inference algorithm. `bzip2` runs 2% slower with `O1-` than with `O1`; this is still 5% faster than the executable produced by CompCert, for example. Of all 72 procedures in `bzip2`, DYNAMO successfully validates the translations for 64, 63, and 57 procedures at `O1-`, `O1`, and `O2` respectively at unroll-factor $\mu = 2$ and a per SMT query timeout of three minutes. At `O1-`, DYNAMO takes around 8.7 CPU hours to compute refinement proofs for the 64 procedures. DYNAMO times out for the remaining 8 procedures, all of which are bigger than 142 ALOC.

Name	SLOC	ALOC	$\#_{al}$	$\#_{call}$	T	$\lvert\mathcal{N}_X\rvert$	$\lvert\mathcal{E}_X\rvert$	EXP	BT	$\#_q$	Avg. qT
`generateMTFValues`	76	144	1	1	4k	14	30	60	16	3860	0.56
`recvDecodingTables`	70	199	2	10	3k	38	66	102	15	5611	0.21
`undoReversible-Transformation_fast`	116	221	1	6	2k	21	34	43	6	2998	0.23

Table 6.2: Statistics obtained by running DYNAMO on procedures in the `bzip2` program.

Three of `bzip2`'s procedures for which refinement proofs are successfully computed at both `O1-` and `O1` optimization levels contain at least one local array, and table 6.2 presents statistics for the `O1-` validation experiments for these procedures. For each procedure, we show the number of source lines of code in C (`SLOC`), the number of assembly instructions in A (`ALOC`), the number of local variables ($\#_{al}$), and the number of procedure calls ($\#_{call}$). The `T` column shows the validation times (in seconds). The $|\mathcal{N}_X|$ and $|\mathcal{E}_X|$ columns show the number of nodes and edges in the final product graph, and `BT` and `EXP` is the number of best-first search backtrackings and the number of (partial) candidate product graphs explored by best-first search in DYNAMO respectively. $\#_q$ is the total number of SMT queries discharged, and Avg. `qT` is the average time taken by an SMT query in seconds for the refinement check.

In the experiment with `O2` optimization level some of the loops in these procedures are vectorized. With an unroll-factor $\mu = 8$, DYNAMO is able to validate `undoReversible-Transformation_fast`'s assembly implementation within ≈ 38 minutes. However, the BFS search diverges away from the correct solution for the other `bzip2` functions, eventually resulting in a timeout. These experiments indicate the need for more robust search algorithms and/or faster logical models/solvers in the presence of aggressive

optimizations on large programs.

In a separate experiment, we manually split the large procedures in `bzip2` into smaller procedures, so that DYNAMO successfully validates the `O1-` compilation of the full modified `bzip2` program in ≈16 hours: the splitting disables some compiler transformations and also reduces the correlation search space.

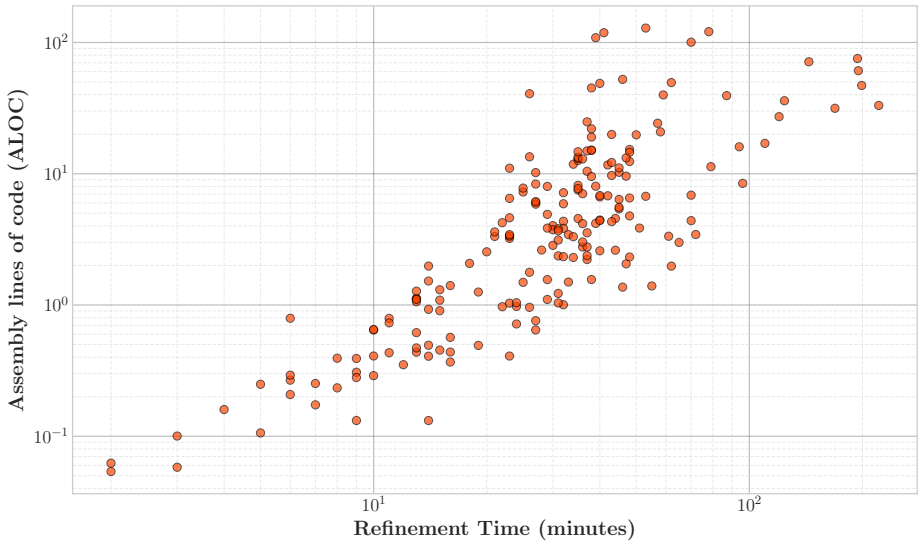


Figure 6.6: Scatter plot of refinement time (in minutes) vs assembly lines of code (ALOC). Both axes are logarithmically scaled.

Figure 6.6 shows the scatter plot of refinement time (in minutes) versus the assembly lines of code (ALOC) of all procedure-pairs considered so far. While the relationship is almost linear, the range of values is quite wide, especially at higher ALOC.

6.2.4 Analysis of Failures

Table 6.3 lists the 15 refinement check failures with full-interval encoding for the refinement checks in fig. 6.2. We have grouped the reasons for these failures into three categories: (1) limitation of blackbox annotation; (2) incompleteness due to affine invariant inference grammar; and (3) incompleteness in affine invariant inference due to chosen set of procedure variables. We discuss each reason in detail in following subsections.

#	Benchmark	Compiler	Failure reason
1	`vsl1`	GCC	Limitation of $\texttt{dealloc}_s$ annotation
2	`vsl1`	ICC	Non-affine invariant required
3	`vsl2`		
4	`vil3`		
5	`vilcc`		
6	`vilce`		
7	`fib`		
8	`rod`		
9	`vin`		
10	`vcu`		
11	`vsl3`		
12	`vsl3`	Clang	
13	`vsl3`	GCC	
14	`mp`	GCC	Incompleteness in affine invariant inference due to the chosen set of procedure variables
15	`ms`		

Table 6.3: Failure reasons for the refinement checks shown in fig. 6.2.

Limitation of the $\texttt{alloc}_s$/$\texttt{dealloc}_s$ annotation algorithm in the blackbox setting

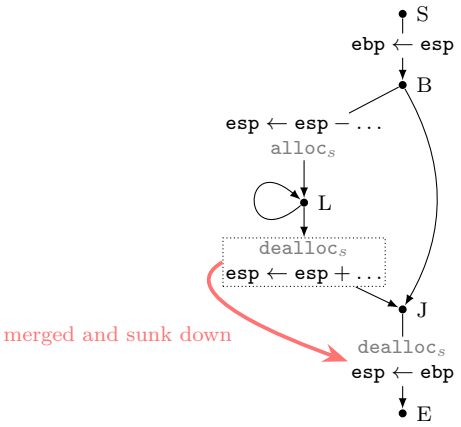
Recall that in the blackbox setting of Dynamo, when hints from the compiler are not available, the annotation algorithm (*asmAnnotOpts()*) limits the insertion of a $\texttt{dealloc}_s$ instruction to only those PCs that occur just before an instruction that updates the stackpointer register `esp`. This limitation may cause a refinement check to fail in some (not all) of the situations where a compiler implements merging of multiple allocations (resp. deallocations) into a single stackpointer decrement (resp. increment) instruction. This is the reason for the failure to validate GCC's compilation of `vsl1`.

Figure 6.7 shows the `vsl1` procedure (fig. 6.7a) and a sketch of the control-flow graph (CFG) of the assembly procedure generated by GCC at `O3` optimization level (fig. 6.7b). The assembly path $S \rightarrow B \rightarrow J \rightarrow E$ represents the case when ($\texttt{n} \leq 0$) and the procedure exits early (without allocating any local variable). PC with label L represents the loop head in the CFG, and the allocation and deallocation of the VLA `v` is supposed to happen before entering the loop and after exiting the loop respectively.

On the assembly CFG path $L \rightarrow J$, the assembly instruction that reclaims stack space (by incrementing the stackpointer) for deallocating `v` has been merged by the compiler with an instruction that restores the stackpointer to its value at the beginning of

```
int vsl1(int *a, int n)
{
  if (n <= 0)
    return 0;
  int v1[n];
  for (int i = 0; i < n; ++i)
  {
    v1[i] = a[i]+a[i];
  }
  return foo1(v1);
}
```

(a) C source code



(b) Abbreviated CFG of GCC compiled assembly

Figure 6.7: `vsl1` procedure from table 6.1 (appears as `vsl`N) and the control-flow graph (CFG) of its GCC compiled assembly.

the function (stored in register `ebp`). Thus, while the original stackpointer increment instruction would have been at the end of the $L \rightarrow J$ path, the merged instruction is sunk by the compiler to lie within the path $J \rightarrow E$. As can be seen, this transformation saves an extra instruction to update the stack pointer on the path $L \rightarrow J \rightarrow E$.

In the absence of compiler hints (blackbox setting), our tool considers the annotation of a $\texttt{dealloc}_s$ instruction in assembly only at a PC that immediately precedes an instruction that updates the stackpointer. In this example, the only candidate PC for annotating $\texttt{dealloc}_s$ (considered by our blackbox algorithm) is on the path $J \rightarrow E$. However, the required position of the $\texttt{dealloc}_s$ instruction was at the end of the path $L \rightarrow J$ (which is not considered because there is no instruction that updates the stackpointer at the end of the path $L \rightarrow J$). Thus, our blackbox algorithm cannot find a refinement proof. On the other hand, providing a manual hint to the tool that it should consider annotating a $\texttt{dealloc}_s$ instruction at the end of the $L \rightarrow J$ path causes the algorithm to successfully return a refinement proof for GCC's compilation of `vsl1`.

For completeness, let us consider what happens when the tool annotates a $\texttt{dealloc}_s$ instruction just before the instruction that updates the stackpointer to `ebp` on the $J \rightarrow E$ path. Such an annotation violates the (stuttering) trace equivalence condition on the procedure path $S \rightarrow B \rightarrow J \rightarrow E$: in the C procedure, there is no deallocation (or allocation) on the early exit path (when $\texttt{n} \leq 0$), but this annotation will cause a $\texttt{dealloc}_s$ instruction to be executed on the correlated path ($S \rightarrow B \rightarrow J \rightarrow E$) in the

assembly procedure. Because a $\texttt{dealloc}_s$ instruction generates a (non-silent) trace event through the `wr` instruction, this candidate annotation therefore fails to show the equivalence of traces on at least one pair of correlated paths. Thus, this candidate annotation is discarded by our algorithm.

Requirement of a non-affine invariant in some ICC compiled benchmarks

The ICC compiler (and in two cases GCC and Clang as well) generates a certain pattern of assembly code in some (not all) cases of VLA containing C source code that necessitate need of a non-affine invariant shape for completing the refinement proof.

In some cases, ICC uses the following pattern of instruction sequence for the allocation of a VLA of size $vlaSz$:

$$\begin{aligned} reg_1 &\leftarrow vlaSz \\ reg_2 &\leftarrow (reg_1 + \texttt{C}) \,\&\, \sim \texttt{C} \\ \texttt{esp} &\leftarrow \texttt{esp} - reg_2 \end{aligned}$$

Here, reg_1 and reg_2 are assembly registers other than `esp`, `esp` is the stackpointer register, `C` is a bitvector constant, and $\sim$ denotes the bitwise complement operator. After execution of the instruction sequence, the value in reg_1 matches the allocation size of the VLA in the C procedure. For example, for a VLA declaration `int v[n]`, reg_1 would have value `n*4` [2]. The value in reg_2 is the allocation amount after adjusting for alignment requirements, e.g., `v` (declared as `int v[n]`) would have an alignment of at least 4 in 32-bit x86. The last instruction shifts `esp` reg_2 bytes below its original value, thus *allocating* the VLA[3].

At time of deallocation, the stackpointer register is simply incremented by the same value as it was decremented by during the allocation:

$$\begin{aligned} reg_1 &\leftarrow vlaSz \\ reg_2 &\leftarrow (reg_1 + \texttt{C}) \,\&\, \sim \texttt{C} \\ \texttt{esp} &\leftarrow \texttt{esp} + reg_2 \end{aligned}$$

[2]Recall that 4 is the size of an `int` in our 32-bit configuration

[3]`esp` must have been originally aligned by the required alignment for the resulting value to be correctly aligned.

Notice that the two assembly sequences are identical except for the last instruction, where in the deallocation sequence + is used in place of −.

Recall that the execution of a stackpointer increment instruction may trigger $\mathcal{U}$ if the deallocated interval does not belong to the stack region ((Op-esp) in fig. 2.6). Thus, in order to prove that the assembly procedure does not trigger $\mathcal{U}$ in the above deallocation sequence, we must have an invariant stating that the original stackpointer value (just before executing the above deallocation sequence), was at least reg_2 bytes (where reg_2 is derived from $vlaSz$) below some address in the stack region *and* the interval spanned by the two stackpointer values belongs entirely to the stack region. The global invariants `StkBd`, `zlIntvl`, and `zaBd` (fig. 4.3) and, invariants `affine` and `ineq` (fig. 4.2) over the ghost variables for stackpointer ($\texttt{sp}.p^{j}_{\ddot{\text{A}}}$) and local regions (`lb`.z, `ub`.z, `lstSz`.z) help in proving separation of intervals defined by stackpointer ghost variables and stack-allocated local regions, and are usually sufficient to discharge the latter check. Proving the former condition, that the gap between two stackpointer values is larger than a value derived from $vlaSz$, however, requires an invariant that cannot be inferred by our predicate grammar.

Let us look at the allocation and deallocation instruction sequences again. At the end of the allocation sequence, the new stackpointer value is related to the old stackpointer value by ($\texttt{esp}_a = \texttt{esp}_b - reg_2$) where $\texttt{esp}_b$ and $\texttt{esp}_a$ are the stackpointer values just before and after executing the stackpointer decrement instruction respectively. Due to translation rule (Op-esp) in fig. 2.6, both $\texttt{esp}_b$ and $\texttt{esp}_a$ will have ghost variables, say, $\texttt{sp}.p^{j_1}_{\ddot{\text{A}}}$ and $\texttt{sp}.p^{j_2}_{\ddot{\text{A}}}$, holding value equal to them. Thus, the relation between the two stackpointer values can be expressed as $\texttt{sp}.p^{j_2}_{\ddot{\text{A}}} = \texttt{sp}.p^{j_1}_{\ddot{\text{A}}} - reg_2$, an affine shape that is a member of `affine`. On the other hand, the relation $reg_2 = (vlaSz + \texttt{C}) \;\&\; \sim\texttt{C}$ is not an affine shape and cannot be captured by any other shape in our predicate grammar. Further, the alignment adjustment computation is not performed in C so there is no C variable that can be related to reg_2 through an affine relation. Consequently, instead of a precise affine invariant relating reg_2 and $vlaSz$, an imprecise inequality invariant of the form $\texttt{sp}.p^{j_2}_{\ddot{\text{A}}} \leq_u \texttt{sp}.p^{j_1}_{\ddot{\text{A}}} - \texttt{vlaSz}$, is inferred (`spOrd` in fig. 4.2). This inferred invariant is sufficient for proving the well-formedness of the `alloc` instruction (so that C does not go to $\mathscr{W}_{\text{C}}$) and proving that Ä does not go to $\mathcal{U}_{\ddot{\text{A}}}$ in (AllocS).

In the deallocation sequence, instead of reusing reg_2 or resetting the stackpointer to older value (as happens in some compilations of GCC), the required value for stackpointer increment is recomputed again using $vlaSz$. At this point, the product graph invariants

do not imply a precise relationship between $vlaSz$ and the reg_2. Thus, the stackpointer is incremented by a value which, according to the inferred invariants, is completely unrelated to the value by which stackpointer was decremented. Consequently, the attempt to prove that the deallocated interval belongs to the stack region fails and we are unable to prove that Ä does not go to $\mathcal{U}_{\text{Ä}}$ in the stackpointer increment instruction.

It can be observed that the necessary invariant required for the proof to go through is $(reg_2 = (vlaSz + \texttt{C})\ \&\ {\sim}\texttt{C})$, where $vlaSz$ can be substituted with the ghost variable for the size of the VLA (`lstSz.`z for allocation site z of the VLA). This invariant shape, though required in this case, is rather specific and may not be useful otherwise. On the other hand, choosing to include it in the predicate grammar is highly likely to cause an increase in the runtime of the tool. We make the choice of omitting it, choosing to sacrifice completeness in favor of (relatively better) runtimes — our choice is informed by casual observations on scalability bottlenecks due to inference of non-affine invariant shapes.

Failure due to choice of program variables for invariant inference

In the `affine` $\sum_i c_i v_i = c$ invariant shape of the predicate grammar (fig. 4.2), the program variables v_i are drawn from a set V that includes the pseudo-registers in C and registers and stack slots in A (section 4.1.5). The candidate variables for correlation in V do not include "memory slots" of shape $\texttt{sel}_{\texttt{sz}}(M_{\text{C}}, \alpha)$ (little-endian concatenation of `sz` bytes starting at α in the array M_{C}) to avoid an explosion in the number of candidate invariants and, consequently, the running time of the algorithm.

This causes a failure while validating the GCC compilations (at `O3`) of the variadic `mp` and `ms` benchmarks in table 6.1. GCC register-allocates the `va_list` variable (that maintains the current position in the variadic argument). On the other hand, the $\texttt{LLVM}_d$ IR maintains this pointer value in a local variable (allocated using an `alloc` instruction) — the loads and stores to this local variable $\langle\!|ap|\!\rangle$ can be seen in fig. 2.3. Thus, for a refinement proof to succeed, a validator must relate the assembly register's value with the value stored inside the local variable's memory region ($\texttt{sel}_4(M_{\text{C}}, \langle\!|ap|\!\rangle)$). Because our invariant inference algorithm does not consider memory slots in C, this required relation is not identified, resulting in a proof failure.

It may be worth asking the question: why does our choice of program variables work for the other benchmarks? Due to the `mem2reg` pass used in C before computing equivalence,

the only memory slots that remain in procedure C pertain to potentially address-taken variables. Our requirements on the product graph X ensure that the memory regions corresponding to address-taken local variables (and global variables) of C and A are equated in X. Thus, relating the *addresses* of potential memory accesses in C and A using affine invariants and considering only the memory slots from A largely suffices for invariant inference to validate most compilations (but not for GCC's compilation of `mp` and `ms`).

6.3 Other Applications

A translation validator has applications beyond compiler validation. This section explores some applications of our tool to domains other than translation validation.

Applicability in a Superoptimizer

An end-to-end translation validation is a crucial ingredient in a synthesis-driven superoptimizer [5, 9, 38] which attempts to generate "optimal" code for a particular program (or specification). In a superoptimizer, a synthesis tool proposes candidate solutions and a translation validation tool validates them against a high-level input specification. The efficacy of superoptimization depends both on the synthesis tool, through its ability to generate effective solutions, and the translation validation tool, through its ability to validate the proposed solutions. With advances in synthesis, e.g. through Large Language Models (LLMs), the burden of effective superoptimization will inevitably shift towards translation validation where DYNAMO-like tools will find application.

Checking enforcement of calling conventions

Consider the case of alignment used for certain types. Compilers often use higher alignment factors than those necessitated by the C standard, e.g., the "`long long`" type is often aligned at eight-byte boundaries to reduce cache misses. Our tool can easily check use of such higher than required alignments by changing the well-formedness condition for alignment (section 2.1.3) to reflect the higher alignment value. By using `long long` type in our first set of benchmarks (containing different programming patterns) in section 6.2.1, we validated that all the three production compilers (Clang/LLVM, GCC, ICC) ensure that `long long` variables are eight-byte aligned for these benchmarks. In contrast, we found that the ACK compiler [47] only ensures the mandated four-byte

alignment.

Use in fuzzing

A translation validator can also be used as a part of compiler-fuzzing tools such as CSmith [52] and EMI [27]. Validation usually takes much longer than testing, yet provides greater (maximum) coverage. It remains to be seen if validation can be used in conjunction with fuzzing to uncover more compiler bugs. We confirmed that our validator is able to detect the previous bugs (involving local memory variables) reported by these compiler fuzzing tools. These bugs were hitherto not possible to uncover through validation due to the lack of support for local variable modeling in prior work.

Detecting performance anomaly

The validator also helped us identify a small performance anomaly in the code generated by a recent version of GCC. It turns out that in the presence of VLAs, GCC emits assembly code to redundantly align an already aligned pointer. This is easily checked by a lightweight static checker on the final product graph that checks if an alignment operation (characterized by the `and` opcode) occurs at an assembly PC where the product graph invariants already ensure that alignment.

Bug in SMT solver

Through our experiments, we uncovered and reported a bug in recent versions of `z3`, including `z3-4.8.14` and `z3-4.12.5`, where for an input satisfiability query Q, the SMT solver returns an unsound model (counterexample) that evaluates Q to false [53]. When a modern SMT solver is used to validate compilations produced by a mature compiler, a bug may be found on either side.

Chapter 7

Conclusion

The overarching goal of this work is to investigate the applicability of translation validation as an alternative to verified compilation. Toward this, we identify the modeling of dynamic allocation and deallocation of local memory as an important unaddressed sub-problem and make multiple contributions within this space.

7.1 Summary

We formalize the refinement from an unoptimized representation (IR) of a C procedure to a 32-bit x86 assembly procedure as a relation over the observable events (termination, procedure-call invocation, (de)allocation, etc.) produced by the execution of the respective procedures (section 2.4). A key aspect of this formalization is the observation of a (de)allocation in the IR procedure as a distinct (observable) event. This enables the identification of a similar (de)allocation event in the assembly procedure through a verifiable *annotation* instruction (sections 2.5.1 and 2.6.1). The identification of correlated events provides a basis for a *lockstep correlation*-based proof method, such as product program construction, where the execution of both procedures matches step-by-step, ensuring event correspondence. We demonstrate that, unlike prior work, this annotation-based modeling is not tied to a particular local (de)allocation strategy and is applicable in different settings, including validation of programs with dynamic variable-sized or constant-sized allocations, e.g., through C99 VLAs or the `alloca()` operator (sections 2.5.4 and 2.6.3).

We generalize the product program-based proof method, used in prior work for estab-

lishing equivalence, to a determinized product program that is applicable for correlating two programs in the presence of non-determinism and, thus, can cater to a generalized refinement setting (section 3.2). We identify the key requirements for witnessing refinement through a determinized product program and present an automatic algorithm for constructing the same (section 3.3).

Our automatic algorithm is based on prior heuristic-guided best-first search that constructs the required product program incrementally (section 4.1). The automatic algorithm supports external untrusted hints for guiding the construction and thus can make use of information from sources such as compiler instrumentation (sections 4.1.3 and 6.1.4). We provide an efficient SMT encoding that is up to 4 times faster than a naive encoding for discharging the generated proof obligations (section 5.3). We show that our prototype implementation is able to validate compilations by production optimizing compilers, Clang/LLVM, GCC, and ICC (section 6.2.1). We also evaluate our prototype on SPEC2000's `bzip2` program, where we are able to successfully validate an optimized compilation of procedures with up to 142 assembly lines of code, failing to validate just 8 procedures out of 72 (section 6.2.3).

7.2 Limitations and Directions for Future Work

While our work represents a significant step toward realizing a practical translation validator, it has several limitations. Although our execution model and refinement definition admit a broad range of transformations and allocation strategies, our (determinized) product program construction and SMT encoding operate under specific constraints. In particular, we assume:

- In allocations and procedure calls that reuse stack space, their relative order is preserved in both the original and compiled programs (section 2.5.4).
- An `alloca()` is always stack-allocated (section 2.6.1).
- The compiler does not specialize allocation-containing paths such that one specialization uses stack allocation and another uses register allocation (section 2.6.1).

While the generalized execution model and refinement definition presented in section 2.7 lift these constraints, they leave the challenge of efficient product program construction and SMT encoding to future work.

Additionally, our over-approximate modeling of procedure calls, which treats them as arbitrary mutations of callee-observable state, lacks the precision necessary for validating interprocedural transformations (section 2.5.4).

Our experiments on the `bzip2` benchmark highlight the need for a more robust and scalable search algorithm and/or SMT encoding. Addressing these scalability challenges presents a promising direction for future work toward a fully realized translation validator.

Appendices

Appendix A

Soundness and Completeness Implications of `isPush()` Choice

An update to the stackpointer `esp` in the assembly procedure A can be through any arbitrary instruction, such as `esp` $\coloneqq \mathbb{Y}$. If the previous `esp` value, just before this instruction was executed, was $\mathbb{X}$, then the stackpointer update distance is $\mathbb{D} = \mathbb{X} - \mathbb{Y}$. In general, it is impossible to tell whether this instruction intends a stack growth by $\mathbb{D}$ bytes (push) or a shrink by $(2^{32} - \mathbb{D})$ bytes (pop). The modeling for the two cases is different: for stack push, an overlap of the interval representing the push with non-stack region causes a $\mathscr{W}$ error, while for stack pop, the stackpointer going outside stack region causes $\mathscr{U}$ error. Refinement is trivially proven if A terminates with $\mathscr{W}$ error. Unfortunately, this seems impossible to disambiguate just by looking at the assembly code – to tackle this dilemma, we assume an oracle function, $\texttt{isPush}(p^j_{\mathsf{A}}, \mathbb{X}, \mathbb{Y})$, that returns `true` iff the assembly instruction at PC p^j_{A} represents a stack push.

In section 2.3.2, we define an $\texttt{isPush}(p^j_{\mathsf{A}}, \mathbb{X}, \mathbb{Y})$ operator for an assembly instruction at p^j_{A} based on thresholding of the update distance $\mathbb{D} = \mathbb{X} - \mathbb{Y}$ by a threshold value $\mathbb{K} = 2^{31} - 1$:

$$\texttt{isPush}(p^j_{\mathsf{A}}, \mathbb{X}, \mathbb{Y}) \Leftrightarrow \mathbb{X} - \mathbb{Y} \leq_u \mathbb{K}$$

Here, $\mathbb{K}$ represents the threshold value for the stack update distance $\mathbb{X} - \mathbb{Y}$, below which we consider the update to be a push.

If $\mathbb{K}$ is smaller than required, then we risk misclassifying stack pushes (stack growth) as stack pops (stack shrink). On the other hand, if $\mathbb{K}$ is bigger than required, then we

risk misclassifying stack pops (stack shrink) as stack pushes (stack growth). In the latter case (when $\mathbb{K}$ is bigger than required), we would incorrectly trigger $\mathscr{W}$, instead of $\mathscr{U}$, and that would cause the refinement proof to complete incorrectly (soundness problem). In the extreme case, if $\mathbb{K} = 2^d - 1$ (where the address space has size 2^d), then even 4-byte stack pops (e.g., through the x86 `pop` instruction) would be considered as stack pushes (growth), and we would incorrectly trigger in every situation where $\mathscr{U}$ was expected, and the refinement proof would complete trivially (and unsoundly).

On the other hand, if $\mathbb{K}$ is smaller than required, we may incorrectly count some stack growth operations as stack pops. In these cases, we will have show to absence of $\mathscr{U}$ (as part of (Safety)) for a stack pop for which a stack push never happened. This would result in an refinement failure (completeness problem).

A.1 $\mathbb{K}$ needs to be at least 2^{d-1} in the presence of VLAs

Consider a VLA declaration, "`char v[n]`" in C. In this case, `n` could be any positive integer $\leq_u$`INT_MAX`; this upper bound of `INT_MAX` comes from the variable size limits imposed by the C language. The corresponding allocation statement in assembly code would be something like "p^j_{A}: `esp` $\coloneqq$ `esp` $-$ `n`". The resulting condition for not triggering $\mathscr{U}$ is (from (OP-ESP) of fig. 2.6):

$$\begin{aligned}\neg(\quad & \neg\texttt{isPush}(p^j_{\mathsf{A}}, \texttt{esp}, \texttt{esp} - \texttt{n}) \\ & \wedge \texttt{esp} \neq \texttt{esp} - \texttt{n} \\ & \wedge \neg\texttt{intrvlInSet}(\texttt{esp}, \texttt{esp} - \texttt{n}, \Sigma^{stk}_{\mathsf{A}}))\end{aligned}$$

or equivalently,

$$\begin{aligned}(\texttt{n} >_u \mathbb{K}) \Rightarrow (\quad & \texttt{n} = 0_{\texttt{i}32} \\ & \vee (\quad \texttt{esp} \neq 0_{\texttt{i}32} \\ & \quad \wedge (\texttt{esp} \leq_u \texttt{esp} - \texttt{n}) \\ & \quad \wedge [\texttt{esp}, \texttt{esp} - \texttt{n}] \subseteq \Sigma^{stk}_{\mathsf{A}}))\end{aligned} \tag{A.1}$$

Now, if $\mathbb{K}$ is smaller than the biggest possible value of `n`, then there exist values of `n` where the left clause (left of $\Rightarrow$) of eq. (A.1) would evaluate to `true`. Consequently, there exist values of `n` for which the right clause has to be proven `true`, i.e., prove that

the stack region is at least $2^d - \mathtt{n}$ bytes large. It may not be possible to prove such strong conditions in all cases and thus we get false refinement check failures. Because the C language constrains `n` to be $\leq_u$ `INT_MAX`$(= 2^{d-1} - 1)$, $\mathbb{K} \geq_u 2^{d-1} - 1$ seems sufficient to be able to validate such translations.

However, $\mathbb{K} = 2^{d-1} - 1$ is also insufficient, because typically the code generated by a compiler for "`char v[n]`" also aligns `n` using a rounding factor $r = 2^i$: "`esp` $\coloneqq$ `esp` $-(\lceil \frac{\mathtt{n}}{r} \rceil \cdot r)$". In this scenario, even though $\mathtt{n} \leq_u (2^{d-1} - 1)$, it is possible for $\mathbb{D} = \lceil \frac{\mathtt{n}}{r} \rceil \cdot r$ to be greater than $(2^{d-1} - 1)$. Thus, if $\mathbb{K} = 2^{d-1} - 1$, there exist legal values of `n` for which stack region is at least $2^d - \mathtt{n}$ bytes large has to be proven to demonstrate absence of $\mathcal{U}$. The choice $\mathbb{K} = 2^{d-1}$ allows for such alignment padding, and thus allows the refinement proof to be completed in these situations.

A.2 $\mathbb{K} = 2^{d-1}$ can still lead to completeness problems

If a single stack update allocates two VLAs at once, we can incorrectly classify a stack growth as a stack shrink.

Consider two C statements in sequence, "`char v1[m]; char v2[n];`". In this case both `m` and `n` can individually be as large as $2^{d-1} - 1$. If the compiler decides to use a single assembly instruction to allocate both these variables, then it is possible for a single stack update distance $\mathbb{D}$ to be greater than $\mathbb{K} = 2^{d-1}$. Thus, in these cases, the refinement proof may fail if we are not able to prove that stack is large enough to contain $2^d - \mathbb{D}$ bytes (for the classified stack pop). This is a completeness problem.

A.3 $\mathbb{K} = 2^{d-1}$ can also lead to soundness problems

If a single stack update deallocates two VLAs at once, we can incorrectly classify a stack shrink as a stack growth.

Consider two C statements in sequence, "`char v1[`$2^{d-1} - 1$`]; char v2[2];`". If during deallocation, the compiler decides to use a single instruction to deallocate both the arrays, e.g., "`esp` $\coloneqq$ `esp + (`2^{d-1}`-1) + 2`" for a total update distance of:

$$\mathbb{D} = -((2^{d-1} - 1) + 2) = 2^{d-1} - 1 \qquad (\mathtt{mod}\ 2^d)$$

Here, because $2^{d-1} - 1 \leq_u \mathbb{K}$ we will classify this "deallocation" as a stack push

(allocation) of $(2^{d-1} - 1)$ bytes and trigger $\mathscr{W}$ if allocation of $(2^{d-1} - 1)$ bytes is not possible. This is a soundness problem because triggering $\mathscr{W}$ under such a weaker condition may lead the refinement proof to succeed incorrectly.

A.4 Solution

Thus, it seems impossible in general to be able to distinguish a push from a pop in a sound manner. This problem is unavoidable in the presence of VLAs. CompCert side-stepped this problem by disabling VLA support and thus being able to statically bound the overall stack size. For a bounded stack, it becomes possible to distinguish pushes from pops. But it is not possible to bound the stack in the presence of a VLA.

Thus we propose that the compiler must explicitly emit trustworthy information that distinguishes a push from a pop. Hence, `isPush()` can simply leverage this information emitted by the compiler.

As explained in section 2.3.2, in our work, we use a threshold of $2^{31} - 1$ on the update distance to disambiguate stack pushes from pops. We rely on manual verification for soundness.

Appendix B

More details of the experiments

B.1 Command-line used for compiling benchmarks in experiments

1. Programs in table 6.1

   - Clang/LLVM v12.0.0

     ```
     clang -m32 -S -no-integrated-as -g -Wl,--emit-relocs -fdata-sections -g
          -fno-builtin -fno-strict-aliasing -fno-optimize-sibling-calls -
         fwrapv -fno-strict-overflow -ffreestanding -fno-jump-tables -fcf-
         protection=none -fno-stack-protector -fno-inline -fno-inline-
         functions -D_FORTIFY_SOURCE=0 -D__noreturn__=__no_reorder__ -I/usr/
         include/x86_64-linux-gnu/c++/9/32 -I/usr/include/x86_64-linux-gnu/c
         ++/9  -mllvm -enable-tail-merge=false -mllvm -nomerge-calls -std=c11
          -O3 <file.c> -o <file.s>
     ```

   - GCC v8.4.0

     ```
     gcc-8 -m32 -S -g -Wl,--emit-relocs -fdata-sections -g -no-pie -fno-pie -
         fno-strict-overflow -fno-unit-at-a-time -fno-strict-aliasing -fno-
         optimize-sibling-calls -fkeep-inline-functions -fwrapv -fno-reorder-
         blocks -fno-jump-tables -fno-caller-saves -fno-inline -fno-inline-
         functions -fno-inline-small-functions -fno-indirect-inlining -fno-
         partial-inlining -fno-inline-functions-called-once -fno-early-
         inlining -fno-whole-program -fno-ipa-sra -fno-ipa-cp -fcf-protection
     ```

```
=none -fno-stack-protector -fno-stack-clash-protection -
D_FORTIFY_SOURCE=0 -D__noreturn__=__no_reorder__ -fno-builtin-printf
 -fno-builtin-malloc -fno-builtin-abort -fno-builtin-exit -fno-
builtin-fscanf -fno-builtin-abs -fno-builtin-acos -fno-builtin-asin
-fno-builtin-atan2 -fno-builtin-atan -fno-builtin-calloc -fno-
builtin-ceil -fno-builtin-cosh -fno-builtin-cos -fno-builtin-exp -
fno-builtin-fabs -fno-builtin-floor -fno-builtin-fmod -fno-builtin-
fprintf -fno-builtin-fputs -fno-builtin-frexp -fno-builtin-isalnum -
fno-builtin-isalpha -fno-builtin-iscntrl -fno-builtin-isdigit -fno-
builtin-isgraph -fno-builtin-islower -fno-builtin-isprint -fno-
builtin-ispunct -fno-builtin-isspace -fno-builtin-isupper -fno-
builtin-isxdigit -fno-builtin-tolower -fno-builtin-toupper -fno-
builtin-labs -fno-builtin-ldexp -fno-builtin-log10 -fno-builtin-log
-fno-builtin-memchr -fno-builtin-memcmp -fno-builtin-memcpy -fno-
builtin-memset -fno-builtin-modf -fno-builtin-pow -fno-builtin-
putchar -fno-builtin-puts -fno-builtin-scanf -fno-builtin-sinh -fno-
builtin-sin -fno-builtin-snprintf -fno-builtin-sprintf -fno-builtin-
sqrt -fno-builtin-sscanf -fno-builtin-strcat -fno-builtin-strchr -
fno-builtin-strcmp -fno-builtin-strcpy -fno-builtin-strcspn -fno-
builtin-strlen -fno-builtin-strncat -fno-builtin-strncmp -fno-
builtin-strncpy -fno-builtin-strpbrk -fno-builtin-strrchr -fno-
builtin-strspn -fno-builtin-strstr -fno-builtin-tanh -fno-builtin-
tan -fno-builtin-vfprintf -fno-builtin-vsprintf -fno-builtin -I/usr/
include/x86_64-linux-gnu/c++/9/32 -I/usr/include/x86_64-linux-gnu/c
++/9  -fno-tree-tail-merge --param max -tail-merge-comparisons=0 --
param max-tail-merge-iterations=0  -std=c11 -O3 <file.c> -o <file.s>
```

- ICC v2021.8.0

```
icc -m32 -D_Float32=__Float32 -D_Float64=__Float64 -D_Float32x=
    __Float32x -D_Float64x=__Float64x -S -g -Wl,--emit-relocs -fdata-
    sections -g -no-ip -fno-optimize-sibling-calls -fargument-alias -no-
    ansi-alias -falias -fno-jump-tables -fno-omit-frame-pointer -fno-
    strict-aliasing -fno-strict-overflow -fwrapv -fabi-version=1 -nolib-
    inline -inline-level=0 -fno-inline-functions -finline-limit=0 -no-
    inline-calloc -no-inline-factor=0 -fno-builtin-printf -fno-builtin-
    malloc -fno-builtin-abort -fno-builtin-exit -fno-builtin-fscanf -fno
    -builtin-abs -fno-builtin-acos -fno-builtin-asin -fno-builtin-atan2
```

-fno-builtin-atan -fno-builtin-calloc -fno-builtin-ceil -fno-builtin -cosh -fno-builtin-cos -fno-builtin-exp -fno-builtin-fabs -fno-builtin-floor -fno-builtin-fmod -fno-builtin-fprintf -fno-builtin-fputs -fno-builtin-frexp -fno-builtin-isalnum -fno-builtin-isalpha -fno-builtin-iscntrl -fno-builtin-isdigit -fno-builtin-isgraph -fno-builtin-islower -fno-builtin-isprint -fno-builtin-ispunct -fno-builtin-isspace -fno-builtin-isupper -fno-builtin-isxdigit -fno-builtin-tolower -fno-builtin-toupper -fno-builtin-labs -fno-builtin-ldexp -fno-builtin-log10 -fno-builtin-log -fno-builtin-memchr -fno-builtin-memcmp -fno-builtin-memcpy -fno-builtin-memset -fno-builtin-modf -fno-builtin-pow -fno-builtin-putchar -fno-builtin-puts -fno-builtin-scanf -fno-builtin-sinh -fno-builtin-sin -fno-builtin-snprintf -fno-builtin-sprintf -fno-builtin-sqrt -fno-builtin-sscanf -fno-builtin-strcat -fno-builtin-strchr -fno-builtin-strcmp -fno-builtin-strcpy -fno-builtin-strcspn -fno-builtin-strlen -fno-builtin -strncat -fno-builtin-strncmp -fno-builtin-strncpy -fno-builtin-strpbrk -fno-builtin-strrchr -fno-builtin-strspn -fno-builtin-strstr -fno-builtin-tanh -fno-builtin-tan -fno-builtin-vfprintf -fno-builtin-vsprintf -fno-builtin -D_FORTIFY_SOURCE=0 -D__noreturn__=__no_reorder__ -qno-opt-multi-version-aggressive -ffreestanding -unroll0 -no-vec -I/usr/include/x86_64-linux-gnu/c++/9/32 -I/usr/include/x86_64-linux-gnu/c++/9 -std=c11 -O3 <file.c> -o <file.s>

2. TSVC

clang -m32 -S -no-integrated-as -g -Wl,--emit-relocs -fdata-sections -g -fno-builtin -fno-strict-aliasing -fno-optimize-sibling-calls -fwrapv -fno-strict-overflow -ffreestanding -fno-jump-tables -fcf-protection=none -fno-stack-protector -fno-inline -fno-inline-functions -D_FORTIFY_SOURCE=0 -D__noreturn__=__no_reorder__ -I/usr/include/x86_64-linux-gnu/c++/9/32 -I/usr/include/x86_64-linux-gnu/c++/9 -msse4.2 -mllvm -enable-tail-merge=false -mllvm -nomerge-calls -std=c11 -O3 <file.c> -o <file.s>

3. bzip2 O1-

clang -m32 -S -g Wl,--emit-relocs -fno-unroll-loops -fdata-sections -fno-inline -fno-inline-functions -fcf-protection=none -fno-stack-protector

-mllvm -enable-tail-merge=false -O1 -mllvm -nomerge-calls -mllvm -no-early-cse -mllvm -no-licm -mllvm -no-machine-licm -mllvm -no-dead-arg-elim -mllvm -no-ip-sparse-conditional-constant-prop -mllvm -no-dce-fcalls -mllvm -replexitval=never -std=c11 -fno-builtin -fno-strict-aliasing -fno-optimize-sibling-calls -fwrapv -fno-strict-overflow -ffreestanding -fno-jump-tables -D_FORTIFY_SOURCE=0 -D__noreturn__=__no_reorder__ -fno-builtin-printf -fno-builtin-malloc -fno-builtin-abort -fno-builtin-exit -fno-builtin-fscanf -fno-builtin-abs -fno-builtin-acos -fno-builtin-asin -fno-builtin-atan2 -fno-builtin-atan -fno-builtin-calloc -fno-builtin-ceil -fno-builtin-cosh -fno-builtin-cos -fno-builtin-exp -fno-builtin-fabs -fno-builtin-floor -fno-builtin-fmod -fno-builtin-fprintf -fno-builtin-fputs -fno-builtin-frexp -fno-builtin-isalnum -fno-builtin-isalpha -fno-builtin-iscntrl -fno-builtin-isdigit -fno-builtin-isgraph -fno-builtin-islower -fno-builtin-isprint -fno-builtin-ispunct -fno-builtin-isspace -fno-builtin-isupper -fno-builtin-isxdigit -fno-builtin-tolower -fno-builtin-toupper -fno-builtin-labs -fno-builtin-ldexp -fno-builtin-log10 -fno-builtin-log -fno-builtin-memchr -fno-builtin-memcmp -fno-builtin-memcpy -fno-builtin-memset -fno-builtin-modf -fno-builtin-pow -fno-builtin-putchar -fno-builtin-puts -fno-builtin-scanf -fno-builtin-sinh -fno-builtin-sin -fno-builtin-snprintf -fno-builtin-sprintf -fno-builtin-sqrt -fno-builtin-sscanf -fno-builtin-strcat -fno-builtin-strchr -fno-builtin-strcmp -fno-builtin-strcpy -fno-builtin-strcspn -fno-builtin-strlen -fno-builtin-strncat -fno-builtin-strncmp -fno-builtin-strncpy -fno-builtin-strpbrk -fno-builtin-strrchr -fno-builtin-strspn -fno-builtin-strstr -fno-builtin-tanh -fno-builtin-tan -fno-builtin-vfprintf -fno-builtin-vsprintf -fno-builtin -I/usr/include/x86_64-linux-gnu/c++/9/32 -I/usr/include/x86_64-linux-gnu/c++/9 bzip2.c -o bzip2.s

4. bzip2 O1

clang -m32 -S -no-integrated-as -g -Wl,--emit-relocs -fdata-sections -g -fno-builtin -fno-strict-aliasing -fno-optimize-sibling-calls -fwrapv -fno-strict-overflow -ffreestanding -fno-jump-tables -fcf-protection=none -fno-stack-protector -fno-inline -fno-inline-functions -D_FORTIFY_SOURCE=0 -D__noreturn__=__no_reorder__ -I/usr/include/x86_64-linux-gnu/c++/9/32 -I/usr/include/x86_64-linux-gnu/c++/9 -fno-unroll-loops -mllvm -enable-tail-merge=false -mllvm -nomerge-calls -std=c11 -

```
O1 bzip2.c -o bzip2.s
```

5. bzip2 O2

```
clang -m32 -S -no-integrated-as -g -Wl,--emit-relocs -fdata-sections -g -
    fno-builtin -fno-strict-aliasing -fno-optimize-sibling-calls -fwrapv -
    fno-strict-overflow -ffreestanding -fno-jump-tables -fcf-protection=
    none -fno-stack-protector -fno-inline -fno-inline-functions -
    D_FORTIFY_SOURCE=0 -D__noreturn__=__no_reorder__ -I/usr/include/x86_64-
    linux-gnu/c++/9/32 -I/usr/include/x86_64-linux-gnu/c++/9 -fno-unroll-
    loops -mllvm -enable-tail-merge=false -mllvm -nomerge-calls -std=c11 -
    O2 bzip2.c -o bzip2.s
```

B.2 Full results for the bzip2 experiment

Table B.1 shows the full list of bzip2 procedures with their assembly lines of code (ALOC) and validation times (in seconds) for the three Clang/LLVM compiler configurations (O1-, O1, O2).

Name	ALOC			Validation time (seconds)		
	O1-	O1	O2	O1-	O1	O2
allocateCompressStructures	47	47	51	43.2	47.2	50.6
badBGLengths	13	13	13	23.2	25.3	28.5
badBlockHeader	13	13	13	21.9	23.0	27.3
bitStreamEOF	13	13	13	22.7	21.2	26.1
blockOverrun	13	13	13	23.5	25.3	27.2
bsFinishedWithStream	22	22	25	24.0	23.2	26.0
bsGetInt32	4	4	4	6.0	6.0	7.2
bsGetIntVS	6	6	6	7.3	8.1	9.8
bsGetUChar	5	5	5	6.0	7.4	7.1
bsGetUInt32	24	24	24	13.3	16.9	20.6
bsPutInt32	6	6	6	6.6	7.6	9.0
bsPutIntVS	6	6	6	9.6	9.8	11.1
bsPutUChar	8	8	8	7.8	8.4	10.9
bsPutUInt32	32	32	32	30.8	30.7	36.4
bsR	46	46	46	42.8	42.8	51.0
bsSetStream	9	9	9	3.7	3.7	4.7

bsW	31	32	36	34.4	33.8	40.0
cadvise	6	6	6	20.3	20.2	26.9
cleanUpAndFail	48	46	46	187.9	179.0	227.8
compressOutOfMemory	14	14	14	33.6	33.0	42.6
compressStream	124	124	124	342.0	369.2	402.6
compressedStreamEOF	16	16	16	21.5	24.0	27.6
crcError	15	15	15	31.4	30.9	36.6
debug_time	2	2	2	1.6	1.8	2.0
doReversibleTransformation	48	49	47	93.3	102.9	129.2
fullGtU	120	113	113	363.0	375.4	404.4
generateMTFValues	144	144	166	1909.3	10441.4	✗
getAndMoveToFrontDecode	299	296	305	✗	✗	✗
getFinalCRC	3	3	3	1.9	2.2	2.3
getGlobalCRC	2	2	2	2.1	1.8	2.2
getRLEpair	72	73	73	144.6	✗	✗
hbAssignCodes	37	37	37	296.4	325.4	330.7
hbCreateDecodeTables	94	94	107	1610.3	1622.3	✗
hbMakeCodeLengths	261	249	292	✗	✗	✗
indexIntoF	23	23	23	30.6	32.0	41.2
initialiseCRC	2	2	2	2.3	1.9	2.3
ioError	15	15	15	17.9	18.3	23.1
loadAndRLEsource	96	96	96	336.2	366.7	✗
main	190	132	183	✗	✗	✗
makeMaps	16	16	16	14.5	15.8	17.9
med3	14	14	14	3.8	4.1	4.2
moveToFrontCodeAndSend	9	9	9	15.1	16.0	15.5
mySIGSEGVorSIGBUScatcher	35	23	23	178.3	✗	✗
mySignalCatcher	10	10	10	16.9	18.9	25.2
panic	13	13	13	32.3	36.1	30.3
qSort3	297	297	363	✗	✗	✗
randomiseBlock	35	37	38	155.1	177.9	✗
recvDecodingTables	199	193	295	2539.8	2690.8	✗
sendMTFValues	691	692	832	✗	✗	✗
setDecompressStructureSizes	79	79	81	426.1	351.8	345.0
setGlobalCRC	3	3	3	2.8	3.0	3.1
showFileNames	8	8	8	15.6	18.4	17.0
simpleSort	194	185	215	✗	✗	✗
sortIt	409	406	421	✗	✗	✗
spec_compress	11	11	11	16.0	16.0	16.0
spec_getc	29	29	29	40.6	43.7	46.6
spec_init	48	49	49	120.4	134.1	123.7
spec_initbufs	9	9	9	11.0	9.6	11.0
spec_load	110	105	105	512.4	499.8	524.1

spec_putc	29	29	29	52.5	51.2	57.3
spec_read	44	46	46	133.5	131.2	166.7
spec_reset	16	16	16	21.8	20.1	23.4
spec_rewind	5	5	5	3.4	3.3	3.5
spec_uncompress	10	10	10	15.0	16.4	14.3
spec_ungetc	45	48	48	176.1	188.2	183.7
spec_write	34	34	34	73.3	77.0	73.8
testStream	195	194	196	1619.5	✗	✗
uncompressOutOfMemory	14	14	14	48.6	50.5	45.8
uncompressStream	169	174	176	1010.5	✗	✗
undoReversibleTransformation_fast	221	223	248	1794.0	1836.8	✗
undoReversibleTransformation_small	273	271	281	✗	✗	✗
vswap	27	27	27	63.3	61.1	54.0

Table B.1: List of `bzip2` procedures with their assembly lines of code (ALOC) and validation times (in seconds) for the three Clang/LLVM compiler configurations (`O1-`, `O1`, `O2`).✗ denotes validation failure for that procedure-compiler pair.

Appendix C

Full source code of the benchmarks

We provide the full source code of the benchmarks from table 6.1 in figs. C.1 to C.4 below (the source code for `fib` is already listed in fig. 2.1a).

The loops of validated `bzip2` benchmarks are shown in fig. C.5.

```
// substitute 𝒩 with 1, 2, 3
// to obtain vsl1, vsl2, vsl3
int vsl𝒩(int n)
{
  if (n <= 0)
  return 0;
  int v1[n], ..., v𝒩[n];
  for (int i = 0; i < n; ++i) {
    v1[i] = F1(a[i]);
    ...
    v𝒩[i] = F𝒩(a[i]);
  }
  return foo𝒩(v1, ...,v𝒩);
}
```

```
// substitute 𝒩 with 1, 2, 3
// to obtain vil1, vil2, vil3
int vil𝒩(unsigned n)
{
  int r = 0;
  for (unsigned i = 1; i < n; ++i) {
    int v1[4*i], ... v𝒩[4*i];
    r += foo𝒩(v1, ...,v𝒩, i);
  }
  return r;
}
```

```
int vcu(int n, int k)
{
   int a[n];
   if (k > 0 && k <= n) {
      a[0] = 0;
      a[k-1] = 10;
      return a[0];
   }
   return 0;
}
```

```
int vilcc(int n)
{
  int ret = 0;
  int i = 1;
  while (i < n) {
    char t[i];
    if (init(t, i) < 0)
      continue;
    ret += t[i-1];
    ++i;
  }
  return ret;
}
```

```
int vilce(int n)
{
  int ret = 0;
  int i = 1;
  while (i < n) {
    char t[i];
    if (init(t, i) < 0)
      break;
    ret += t[i-1];
    ++i;
  }
  return ret;
}
```

Figure C.1: Benchmarks with VLAs.

```c
#include <alloca.h>
int as(int n)
{
  if (n < 1) {
    return 0;
  }
  int* p = alloca(n*sizeof(n));
  for (int i = 0; i<n; ++i) {
    p[i] = i*i;
  }
  return p[0]*p[n-1];
}
```

```c
int ams(int n)
{
  if (n <= 0)
    return -1;
  int* p;
  if (n < 4096) {
    p = alloca(sizeof(int)*n);
  } else {
    p = malloc(sizeof(int)*n);
    if (!p) return -1;
  }
  foo(p);
  int ret = p[0]+p[n/2]+p[n-1];
  if (!(n < 4096))
    free(p);
  return ret;
}
```

```c
int ac(char* s, int fd, int* a)
{
  int n;
  if (!s || (n = strlen(s)) <= 0)
    return 0;
  if (!a) {
    a = alloca(sizeof(int)*n);
  }
  for (int i = 0; i < n; ++i) {
    a[i] = s[i] + 32;
  }
  return write(fd, a, n);
}
```

```c
#include <alloca.h>
int n;
int all()
{
  typedef struct lln {
    int data;
    struct lln* next;
  } Node;
  if (n > 4096)
    return 0;
  Node* hd = 0;
  for (int i = 0; i < n; ++i) {
    Node* t = alloca(sizeof(Node));
    t->data = next_data();
    t->next = hd; hd = t;
  }
  Node* t = hd;
  int ret = 0;
  while (t != 0) {
    ret += t->data;
    t = t->next;
  }
  return ret;
}
```

Figure C.2: Benchmarks with use of `alloca`

```
const int cts[] = { 0x66, 0x65, 0x67, 0x60 };
int rod(int n)
{
  char zz[] = "0123456789";
  printf("Scanning␣%d␣chars", n);
  char t[n];
  scanf("%s",t);
  int ret = 0;
  for (int i = 0, j = 0; i < n; ++i) {
    printf("Round␣#...\n", i);
    zz[j] ^= t[i];
    if (++j >= sizeof zz) j = 0;
  }
  ret += zz[0] + cts[n%((sizeof cts)/sizeof(cts[0]))];
  printf("Returning␣%d", ret);
  return ret;
}
```

Figure C.3: Benchmark `rod` with mixed use of VLA and address-taken variable.

```c
#include <stdarg.h>
void mp(char *fmt, ...)
{
  va_list ap;
  char *p, *sval;
  int ival;

  va_start(ap, fmt);
  for (p = fmt; *p; p++) {
    check(p);
    if (*p != '%') {
      putchar(*p);
      continue;
    }
    switch (*++p) {
      case 'd':
        ival = va_arg(ap, int);
        print_int(ival);
        break;
      case 's':
        for (sval = va_arg(ap, char*);
             *sval; sval++)
          putchar(*sval);
        break;
      default:
        break;
    }
  }
  va_end(ap);
}
```

```c
#include <stdarg.h>
int ms(char* fmt, ...)
{
  va_list ap;
  char *p;
  int ret = 0;

  va_start(ap, fmt);
  for (p = fmt; *p; ++p) {
    DBG();
    if (*p != '%') {
      if (!is_blank(*p)) {
        if (!match_char(*p))
          break;
      }
      continue;
    }
    switch (*++p) {
      case 'd' :
        *va_arg(ap, int*) = read_int();
        ++ret;
        break;
      case 's' :
        read_string(va_arg(ap, char*));
        ++ret;
        break;
      case '%':
        if (!match_char('%'))
          goto end;
        break;
      default:
        goto end;
    }
  }
end:
  va_end(ap);
  return ret;
}
```

Figure C.4: Benchmarks `mp` and `ms` with variable argument list. `mp` is adapted from `minprintf` of K&R[24]

```
void recvDecodingTables() {
  unsigned char inUse16[16];
  for (...) { /* write:inUse16 ... */ }
  for (...) { /* ... */ }
  for (...) { /* read:inUse16 ...*/
    for (...) { /* ... */ }
  }
  for (...) { while (...) {/* ... */ }  }
  { unsigned char pos[6];
    for (...) { /* write:pos ... */ }
    for (...) { /* read,write:pos ... */
      while (...) {/* ... */ }
    }
  }
  for (...) {
    for (...) {
      while (...) {/* ... */ }
    }
  }
  for (...) { for (...) { /* ... */ } }
}
```

(a) Loops in `recvDecodingTables()`

```
void generateMTFValues() {
  unsigned char yy[256];
  for (...) { /* ... */ }
  for (...) { /* write:yy ... */ }
  for (...) { /* read,write:yy ... */
    while (...) {/* ... */ }
    while (...) {/* ... */ }
  }
while (...) {/* ... */ }
}
```

(b) Loops in `generateMTFValues()`

```
void undoReversibleTransformation_fast() {
  int cftab[257];
  for (...) { /* write:cftab ... */ }
  for (...) { /* read,write:cftab ... */ }
  for (...) { /* read,write:cftab ... */ }
  if (...) { while (...) for (...) { /* ... */ } }
  else { while (...) for (...) { /* ... */ } }
}
```

(c) Loops in `undoReversibleTransformation_fast()`

Figure C.5: Structure of `bzip2`'s functions

List of Publications

[1] Shubhani Gupta, Abhishek Rose, and Sorav Bansal. "Counterexample-Guided Correlation Algorithm for Translation Validation". In: *Proc. ACM Program. Lang.* 4.OOPSLA (Nov. 2020). DOI: 10.1145/3428289. URL: https://doi.org/10.1145/3428289.

[2] Vaibhav Kiran Kurhe, Pratik Karia, Shubhani Gupta, Abhishek Rose, and Sorav Bansal. "Automatic Generation of Debug Headers through BlackBox Equivalence Checking". In: *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2022, pp. 144–154. DOI: 10.1109/CGO53902.2022.9741273.

[3] Abhishek Rose and Sorav Bansal. "Modeling Dynamic (De)Allocations of Local Memory for Translation Validation". In: *Proc. ACM Program. Lang.* 8.OOPSLA1 (Apr. 2024). DOI: 10.1145/3649863. URL: https://doi.org/10.1145/3649863.

Biography

Abhishek Rose is a PhD student in the Department of Computer Science and Engineering at IIT Delhi. He obtained his B. Tech. from MSIT, GGSIPU and his M. Tech. from IIT Kanpur.

Bibliography

[1] *'alloca' Instruction. LLVM Language Reference Manual*. URL: https://llvm.org/docs/LangRef.html#alloca-instruction.

[2] *alloca(3) Linux Programmer's Manual*. URL: https://man7.org/linux/man-pages/man3/alloca.3.html.

[3] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. Tech. rep. 1994.

[4] Sorav Bansal. "Peephole Superoptimization". PhD thesis. Stanford University, 2008. URL: https://sorav.compiler.ai/pubs/thesis.pdf.

[5] Sorav Bansal and Alex Aiken. "Automatic Generation of Peephole Superoptimizers". In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XII. San Jose, California, USA: ACM, 2006, pp. 394–403. ISBN: 1-59593-451-0. DOI: 10.1145/1168857.1168906. URL: http://doi.acm.org/10.1145/1168857.1168906.

[6] Clark Barrett, Yi Fang, Benjamin Goldberg, Ying Hu, Amir Pnueli, and Lenore Zuck. "TVOC: A Translation Validator for Optimizing Compilers". In: *Computer Aided Verification*. Ed. by Kousha Etessami and Sriram K. Rajamani. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 291–295. ISBN: 978-3-540-31686-2.

[7] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. https://www.smt-lib.org. 2016.

[8] Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. "Semantic Program Alignment for Equivalence Checking". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. Phoenix, AZ, USA: ACM, 2019, pp. 1027–1040. ISBN: 978-1-4503-6712-7. DOI: 10.1145/3314221.3314596. URL: http://doi.acm.org/10.1145/3314221.3314596.

[9] Berkeley Churchill, Rahul Sharma, JF Bastien, and Alex Aiken. "Sound Loop Superoptimization for Google Native Client". In: *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '17. Xi'an, China: ACM, 2017, pp. 313–326. ISBN: 978-1-4503-4465-4. DOI: 10.1145/3037697.3037754.

[10] *Clang C Language Family Frontend for LLVM*. URL: https://clang.llvm.org/.

[11] Keith Cooper. "Live Range Splitting in a Graph Coloring Register Allocator". In: (Jan. 1998). DOI: 10.1007/BFb0026430.

[12] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. "Efficiently computing static single assignment form and the control dependence graph". In: *ACM Trans. Program. Lang. Syst.* 13.4 (Oct. 1991), pp. 451–490. ISSN: 0164-0925. DOI: 10.1145/115372.115320. URL: https://doi.org/10.1145/115372.115320.

[13] Manjeet Dahiya. "Black-box Equivalence Checking across Compiler Transformations". PhD thesis. Indian Institute of Technology Delhi, 2018. URL: https://sorav.compiler.ai/pubs/manjeet_thesis.pdf.

[14] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976. ISBN: 013215871X.

[15] *GCC, the GNU Compiler Collection*. URL: %5Curl%7Bhttps://gcc.gnu.org/%7D.

[16] Dick Grune, Kees van Reeuwijk, Henri E. Bal, Ceriel J.H. Jacobs, and Koen Langendoen. *Modern Compiler Design*. 2nd. Springer Publishing Company, Incorporated, 2012. ISBN: 1461446988.

[17] Shubhani Gupta, Abhishek Rose, and Sorav Bansal. "Counterexample-Guided Correlation Algorithm for Translation Validation". In: *Proc. ACM Program. Lang.* 4.OOPSLA (Nov. 2020). DOI: 10.1145/3428289. URL: https://doi.org/10.1145/3428289.

[18] Shubhani Gupta, Aseem Saxena, Anmol Mahajan, and Sorav Bansal. "Effective Use of SMT Solvers for Program Equivalence Checking Through Invariant-Sketching and Query-Decomposition". In: *Theory and Applications of Satisfiability Testing – SAT 2018*. Ed. by Olaf Beyersdorff and Christoph M. Wintersteiger. Cham: Springer International Publishing, 2018, pp. 365–382. ISBN: 978-3-319-94144-8.

[19] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. "Defining the undefinedness of C". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. Portland, OR, USA: Association for Computing Machinery, 2015, pp. 336–345. ISBN: 9781450334686. DOI: 10.1145/2737924.2737979. URL: https://doi.org/10.1145/2737924.2737979.

[20] John L. Henning. "SPEC CPU2000: Measuring CPU performance in the new millenium". In: *IEEE Computer* 33.7 (July 2000), pp. 28–35.

[21] ISO. *ISO/IEC 9899:2011 Information technology — Programming languages — C*. Geneva, Switzerland: International Organization for Standardization, Dec. 2011, 683 (est.) URL: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853.

[22] Jeehoon Kang, Yoonseung Kim, Youngju Song, Juneyoung Lee, Sanghoon Park, Mark Dongyeon Shin, Yonghyun Kim, Sungkeun Cho, Joonwon Choi, Chung-Kil Hur, and Kwangkeun Yi. "Crellvm: Verified Credible Compilation for LLVM". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2018. Philadelphia, PA, USA: ACM, 2018, pp. 631–645. ISBN: 978-1-4503-5698-5. DOI: 10.1145/3192366.3192377. URL: http://doi.acm.org/10.1145/3192366.3192377.

[23] Theodoros Kasampalis, Daejun Park, Zhengyao Lin, Vikram S. Adve, and Grigore Roşu. "Language-Parametric Compiler Validation with Application to LLVM". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS 2021. Virtual, USA: Association for Computing Machinery, 2021, pp. 1004–1019. ISBN: 9781450383172.

[24] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. 2nd. Prentice Hall Professional Technical Reference, 1988. ISBN: 0131103709.

[25] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. "CakeML: A Verified Implementation of ML". In: *Principles of Programming Languages (POPL)*. ACM Press, Jan. 2014, pp. 179–191. DOI: 10.1145/2535838.2535841. URL: https://cakeml.org/popl14.pdf.

[26] Vaibhav Kiran Kurhe, Pratik Karia, Shubhani Gupta, Abhishek Rose, and Sorav Bansal. "Automatic Generation of Debug Headers through BlackBox Equivalence Checking". In: *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2022, pp. 144–154. DOI: 10.1109/CGO53902.2022.9741273.

[27] Vu Le, Mehrdad Afshari, and Zhendong Su. "Compiler Validation via Equivalence Modulo Inputs". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '14. Edinburgh, United

Kingdom: ACM, 2014, pp. 216–226. ISBN: 978-1-4503-2784-8. DOI: 10.1145/2594291.2594334. URL: http://doi.acm.org/10.1145/2594291.2594334.

[28] Juneyoung Lee, Dongjoo Kim, Chung-Kil Hur, and Nuno P. Lopes. "An SMT Encoding of LLVM's Memory Model for Bounded Translation Validation". In: *Computer Aided Verification*. Ed. by Alexandra Silva and K. Rustan M. Leino. Cham: Springer International Publishing, 2021, pp. 752–776. ISBN: 978-3-030-81688-9.

[29] Xavier Leroy. "Formal certification of a compiler back-end, or: programming a compiler with a proof assistant". In: *33rd ACM symposium on Principles of Programming Languages*. ACM Press, 2006, pp. 42–54. URL: http://gallium.inria.fr/~xleroy/publi/compiler-certif.pdf.

[30] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. "CompCert – A Formally Verified Optimizing Compiler". In: *ERTS 2016: Embedded Real Time Software and Systems*. SEE, 2016. URL: http://xavierleroy.org/publi/erts2016_compcert.pdf.

[31] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. "Alive2: Bounded Translation Validation for LLVM". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021, pp. 65–79. ISBN: 9781450383912. DOI: 10.1145/3453483.3454030. URL: https://doi.org/10.1145/3453483.3454030.

[32] H.J. Lu, David L Krietzer, Milind Girkar, and Zia Ansari. *System V Application Binary Interface. Intel386 Architecture Processor Supplement. Version 1.0*. https://uclibc.org/docs/psABI-i386.pdf. Feb. 2015.

[33] Saeed Maleki, Yaoqing Gao, Maria J. Garzarán, Tommy Wong, and David A. Padua. "An Evaluation of Vectorizing Compilers". In: *Proceedings of the 2011*

International Conference on Parallel Architectures and Compilation Techniques. PACT '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 372–382. ISBN: 978-0-7695-4566-0. DOI: 10.1109/PACT.2011.68. URL: https://doi.org/10.1109/PACT.2011.68.

[34] David Menendez, Santosh Nagarakatte, and Aarti Gupta. "Alive-FP: Automated Verification of Floating Point Based Peephole Optimizations in LLVM". In: Sept. 2016, pp. 317–337. ISBN: 978-3-662-53412-0. DOI: 10.1007/978-3-662-53413-7_16.

[35] KedarS. Namjoshi and LenoreD. Zuck. "Witnessing Program Transformations". English. In: *Static Analysis*. Ed. by Francesco Logozzo and Manuel Fähndrich. Vol. 7935. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 304–323. ISBN: 978-3-642-38855-2. DOI: 10.1007/978-3-642-38856-9_17. URL: http://dx.doi.org/10.1007/978-3-642-38856-9_17.

[36] George C. Necula. "Translation Validation for an Optimizing Compiler". In: *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*. PLDI '00. Vancouver, British Columbia, Canada: ACM, 2000, pp. 83–94. ISBN: 1-58113-199-2. DOI: 10.1145/349299.349314. URL: http://doi.acm.org/10.1145/349299.349314.

[37] Amir Pnueli, Michael Siegel, and Eli Singerman. "Translation Validation". In: *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*. TACAS '98. London, UK, UK: Springer-Verlag, 1998, pp. 151–166. ISBN: 3-540-64356-7. URL: http://dl.acm.org/citation.cfm?id=646482.691453.

[38] Raimondas Sasnauskas, Yang Chen, Peter Collingbourne, Jeroen Ketema, Jubi Taneja, and John Regehr. "Souper: A Synthesizing Superoptimizer". In: *CoRR*

abs/1711.04422 (2017). arXiv: 1711.04422. URL: http://arxiv.org/abs/1711.04422.

[39] Thomas Sewell. "Translation Validation for Verified, Efficient and Timely Operating Systems". PhD thesis. Sydney, Australia: UNSW, July 2017. URL: https://trustworthy.systems/publications/papers/Sewell%3Aphd.pdf.

[40] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. "Translation Validation for a Verified OS Kernel". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '13. Seattle, Washington, USA: Association for Computing Machinery, 2013, pp. 471–482. ISBN: 9781450320146. DOI: 10.1145/2491956.2462183. URL: https://doi.org/10.1145/2491956.2462183.

[41] Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. "Data-driven Equivalence Checking". In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA '13. Indianapolis, Indiana, USA: ACM, 2013, pp. 391–406. ISBN: 978-1-4503-2374-1. DOI: 10.1145/2509136.2509509. URL: http://doi.acm.org/10.1145/2509136.2509509.

[42] Shubhani. "Counterexample-Guided Equivalence Checking". PhD thesis. Indian Institute of Technology Delhi, 2023. URL: https://sorav.compiler.ai/pubs/shubhani_thesis.pdf.

[43] Bjarne Steensgaard. "Points-to analysis in almost linear time". In: *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1996, pp. 32–41.

[44] Michael Stepp, Ross Tate, and Sorin Lerner. "Equality-based Translation Validator for LLVM". In: *Proceedings of the 23rd International Conference on Computer Aided Verification*. CAV'11. Snowbird, UT: Springer-Verlag, 2011, pp. 737–742.

ISBN: 978-3-642-22109-5. URL: http://dl.acm.org/citation.cfm?id=2032305.2032364.

[45] Chengnian Sun, Vu Le, and Zhendong Su. "Finding Compiler Bugs via Live Code Mutation". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2016. Amsterdam, Netherlands: Association for Computing Machinery, 2016, pp. 849–863. ISBN: 9781450344449. DOI: 10.1145/2983990.2984038. URL: https://doi.org/10.1145/2983990.2984038.

[46] Chengnian Sun, Vu Le, Qirun Zhang, and Zhendong Su. "Toward understanding compiler bugs in GCC and LLVM". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ISSTA 2016. Saarbrücken, Germany: Association for Computing Machinery, 2016, pp. 294–305. ISBN: 9781450343909. DOI: 10.1145/2931037.2931074. URL: https://doi.org/10.1145/2931037.2931074.

[47] Andrew S. Tanenbaum, Hans van Staveren, E. G. Keizer, and Johan W. Stevenson. "A Practical Tool Kit for Making Portable Compilers". In: *Commun. ACM* 26.9 (Sept. 1983), pp. 654–660. ISSN: 0001-0782. DOI: 10.1145/358172.358182. URL: https://doi.org/10.1145/358172.358182.

[48] The LLVM developers. *The LLVM Compiler Infrastructure*. URL: https://llvm.org/.

[49] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. "Evaluating Value-graph Translation Validation for LLVM". In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. San Jose, California, USA: ACM, 2011, pp. 295–305. ISBN: 978-1-4503-0663-8. DOI: 10.1145/1993498.1993533. URL: http://doi.acm.org/10.1145/1993498.1993533.

[50] Jean-Baptiste Tristan and Xavier Leroy. "Formal verification of translation validators: A case study on instruction scheduling optimizations". In: *Proceedings of the 35th ACM Symposium on Principles of Programming Languages (POPL'08)*. ACM Press, Jan. 2008, pp. 17–27. URL: http://xavierleroy.org/publi/validation-scheduling.pdf.

[51] Xi Wang, Haogang Chen, Alvin Cheung, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. "Undefined behavior: what happened to my code?" In: *Proceedings of the Asia-Pacific Workshop on Systems*. APSYS '12. Seoul, Republic of Korea: Association for Computing Machinery, 2012. ISBN: 9781450316699. DOI: 10.1145/2349896.2349905. URL: https://doi.org/10.1145/2349896.2349905.

[52] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. "Finding and Understanding Bugs in C Compilers". In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. San Jose, California, USA: ACM, 2011, pp. 283–294. ISBN: 978-1-4503-0663-8. DOI: 10.1145/1993498.1993532. URL: http://doi.acm.org/10.1145/1993498.1993532.

[53] *Z3 bug report for an unsound model*. https://github.com/Z3Prover/z3/issues/7132. 2024.

[54] Anna Zaks and Amir Pnueli. "CoVaC: Compiler Validation by Program Analysis of the Cross-Product". In: *Proceedings of the 15th International Symposium on Formal Methods*. FM '08. Turku, Finland: Springer-Verlag, 2008, pp. 35–51. ISBN: 978-3-540-68235-6. DOI: 10.1007/978-3-540-68237-0_5. URL: http://dx.doi.org/10.1007/978-3-540-68237-0_5.

[55] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. "Formal Verification of SSA-based Optimizations for LLVM". In: *Proceedings*

of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '13. Seattle, Washington, USA: ACM, 2013, pp. 175–186. ISBN: 978-1-4503-2014-6. DOI: 10.1145/2491956.2462164. URL: http://doi.acm.org/10.1145/2491956.2462164.

[56] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. "Formalizing the LLVM Intermediate Representation for Verified Program Transformations". In: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '12. Philadelphia, PA, USA: ACM, 2012, pp. 427–440. ISBN: 978-1-4503-1083-3. DOI: 10.1145/2103656.2103709. URL: http://doi.acm.org/10.1145/2103656.2103709.

[57] Lenore Zuck, Amir Pnueli, Yi Fang, and Benjamin Goldberg. "VOC: A Methodology for the Translation Validation of Optimizing Compilers". In: 9.3 (Mar. 28, 2003), pp. 223–247.