IMPERIAL

MEng Individual Project

Imperial College London

Department of Computing

From State to Solver: Sound SMT Encodings for Compositional Symbolic Execution

Author:
Shivanandan Tamil Kumaran

Supervisors:
Dr. Andreas Lööw
Prof. Phillipa Gardner

June 13, 2025

Acknowledgements

I would like to first thank Andreas Lööw for his attentive support throughout this project. No doubt it would not have been possible without him. His enthusiasm and curiosity about the topic inspired the same in me, and I will truly miss the thoughtful discussions we shared. I would also like to thank Philippa Gardner for introducing me to the world of software verification, and sparking my passion for this subject. Her advice and guidance, have been invaluable.

To Amma, Appa and Vaishu, I thank you all for your endless love and support. You're the reason I am able to follow my dreams – for this I am eternally grateful.

Lastly, I want to thank my friends, who have always been by my side for the past four years. It has been an incredible journey, and there's no one else I would rather have done it with.

Contents

1. Introduction **7**
1.1. Contributions . . . 8

2. Background **10**
2.1. Separation Logic (SL) . . . 10
2.2. Incorrectness Separation Logic (ISL) . . . 11
2.3. Symbolic Execution and Symbolic States . . . 11
2.4. Symbolic Execution and Satisfiability . . . 12
2.4.1. OX Satisfiability . . . 12
2.4.2. UX Satisfiability . . . 13
2.5. Compositional Symbolic Execution . . . 13
2.5.1. CSE and Entailment . . . 14
2.6. CSE Tools . . . 14
2.6.1. The Gillian Framework . . . 14
2.6.2. Viper . . . 15
2.6.3. VeriFast . . . 15
2.6.4. CN . . . 15
2.7. The SMT-LIB Standard . . . 15
2.7.1. SMT-LIB Theories . . . 15
2.7.2. Function Definitions and Termination . . . 17
2.7.3. Quantification . . . 17
2.7.4. ADTs . . . 18
2.7.5. SMT Solvers . . . 20

3. A Survey of CSE Tools **21**
3.1. Key Challenges . . . 21
3.1.1. Typing . . . 21
3.1.2. Partiality . . . 21
3.1.3. User-Defined Functions and Data Types . . . 21
3.1.4. Performance Considerations . . . 22
3.2. Survey Methodology . . . 22
3.3. Approaches to Handle Typing . . . 24
3.4. Approaches to Handle User-Defined Datatypes . . . 24
3.4.1. Gillian . . . 24
3.4.2. CN . . . 24
3.4.3. VeriFast . . . 25

3.4.4. Viper . . . 25
3.4.5. Axiomatising ADTs . . . 27
3.5. Approaches to Handle User-Defined Functions . . . 28
3.5.1. CN . . . 28
3.5.2. VeriFast . . . 29

4. Formalising the Encoding of Symbolic State into SMT-LIB . . . 31
4.1. Formalising CSE . . . 31
4.1.1. Concrete State . . . 31
4.1.2. Logical Expressions . . . 32
4.1.3. Assertions . . . 35
4.1.4. Symbolic State . . . 35
4.1.5. Typing Environment and Symbolic Execution . . . 37
4.1.6. Untypable Logical Expressions . . . 37
4.1.7. Consume and Produce Operations . . . 39
4.2. Formalising SMT-LIB . . . 40
4.2.1. Term Language . . . 41
4.2.2. Signatures . . . 41
4.2.3. Structures . . . 42
4.2.4. Valuations . . . 42
4.2.5. Theories . . . 42
4.2.6. Satisfiability . . . 42
4.3. Formalising the Encoding . . . 43
4.3.1. Background Signature . . . 43
4.3.2. Encoding Logical Expressions . . . 44
4.3.3. Background Assertions . . . 45
4.3.4. Encoding Path Conditions and Symbolic Stores . . . 46
4.3.5. Encoding Symbolic Memories . . . 47
4.3.6. Encoding OX Satisfiability . . . 47
4.3.7. Encoding UX Satisfiability . . . 48
4.3.8. Unfolding Symbolic Predicates . . . 49
4.3.9. Encoding Entailment . . . 49
4.4. Bridging CSE and SMT-LIB . . . 50
4.4.1. Typing Environments and the Background Signature . . . 50
4.4.2. LVal and $\Sigma_{\hat{T}}$-structures . . . 50
4.4.3. Logical Variable Substitutions and Valuations . . . 51
4.5. Correctness . . . 52
4.5.1. Well-Sortedness . . . 52
4.5.2. Soundness . . . 53
4.6. Findings from Theory . . . 54
4.6.1. Typing Environment . . . 55
4.6.2. Partiality . . . 55
4.6.3. UX Encoding . . . 57

4.6.4. Towards Language Parametricity 57

5. Extending Gillian with User-Defined Datatypes and Functions 59
5.1. Syntax 59
5.2. Matching Plans 60
5.3. Reduction 61
5.4. Encoding 61
5.4.1. Partiality in Selectors 61

6. Evaluation 63
6.1. Theoretical Evaluation 63
6.1.1. Learnings from Theory 63
6.1.2. Limitations 64
6.2. Practical Evaluation 64
6.2.1. Ease of Verification 64
6.2.2. Performance 68

7. Conclusion 70
7.1. Further Work 70
Declarations 74

A. Formalising CSE 75
A.1. Assertions 75
A.2. Symbolic State 75
A.3. Untypable Logical Expressions 75
A.3.1. Proof of Lemma 2 77

B. Formalising SMT-LIB 78
B.1. Well Sorted Terms 78
B.2. Term Interpretations 78

C. Formalising the Encoding 79
C.1. Proof of Lemma 6 79
C.2. Definition of encTys 79
C.3. Definition of $\mathsf{encLExp}_{\hat{T}}$ 80
C.4. Proof of Lemma 7 82

D. Bridging CSE and SMT-LIB 84
D.1. $\Sigma_{\hat{T}}$-structures and LVal 84
D.2. Logical Variable Substitutions and Valuations 84

E. Correctness 86
E.1. Proof of Lemma 11 86
E.2. Proof of Lemma 14 87
E.3. Proof of Lemma 15 91

E.4. Proof of Lemma 16 93
E.5. Proof of Lemma 12 94
E.6. Proof of Lemma 13 94
E.7. Proof of Theorem 8 95
E.8. Proof of Theorem 9 96
E.9. Additional Lemmas and Proofs 97

1. Introduction

As software becomes increasingly integral to the most critical areas of society, ensuring the correctness of the software we develop is paramount. The complexity of modern systems, coupled with the stakes involved in their failure, underscores the need for methods of formal verification that go beyond traditional testing and debugging techniques.

One such method for semi-automatically verifying the correctness of software is *symbolic execution*. Symbolic execution is a powerful technique that allows us to explore the behavior of a program across all possible inputs and execution paths. In contrast to *concrete execution*, i.e. traditional testing and debugging techniques, where a program is run with a specific input and a single control flow path is explored, symbolic execution operates on symbolic states – states that represent a broad set of potential concrete states rather than a single concrete state. By manipulating symbolic states instead of concrete ones, symbolic execution can explore multiple paths of execution simultaneously. This makes it a particularly valuable tool for soundly analyzing programs for correctness properties, such as safety and security guarantees [1].

While symbolic execution has demonstrated great potential, recent advancements have focused on addressing the scalability challenges that arise when using these techniques on large, complex software systems. One promising direction is the development of *functionally compositional* symbolic execution [13]. The key idea behind this approach is that software verification can be modularized, allowing for the verification of smaller program components (e.g., functions) independently. This is particularly useful for managing large programs, as it enables symbolic execution engines to reason about and check the correctness of individual components before composing them together to reason about the entire system.

Central to the compositionality of symbolic execution is the notion of local reasoning, especially when dealing with functions that manipulate heap resources. A function specification should describe only the partial state or resources that the function accesses or modifies, rather than requiring a full description of the entire program state. This allows us to verify the function's correctness in isolation, and then reuse this verified specification when the function is invoked in different contexts with potentially larger or more complex states. This modular approach to verification is supported by the framework of *separation logic* [18, 24], which allows for reasoning about heap-manipulating programs in a way that is compositional and sound. Separation logic has been extended in recent work to *incorrectness separation logic* (ISL) [20], which enables under-approximate reasoning and underpins true bug-finding.

It becomes necessary during compositional symbolic execution to query the satisfiability of a symbolic state by some concrete state. In over-approximate reasoning (i.e. verification), cutting unsatisfiable states mitigates path explosion. In under-approximate reasoing (i.e. bug-finding), bugs must only be reported if they occur in states that are realisable. Typically, satisfiability modulo theories (SMT) solvers are used for the purpose of answering these satisfiability questions.

An SMT solver is a tool that determines whether logical formulas are satisfiable with respect to certain background theories like arithmetic, arrays, or bit-vectors. It extends SAT solvers by handling more complex expressions involving these theories. Most modern SMT solvers follow the SMT-LIB standard [3], which defines a common syntax and semantics for expressing queries.

However, the encoding of symbolic state into queries that can be processed by SMT solvers is non-trivial and presents several challenges:

- **Typing** – Tools like *Gillian* [6, 16], for example, are designed to work with dynamically typed languages, whereas the SMT-LIB standard enforces static typing.
- **Partiality** – SMT-LIB only supports *total* functions. Thus care must be taken when encoding partial expressions, in order to ensure that the semantics are preserved in the translation.
- **User-Defined Functions and Data Types** – Predicates often involve user-defined functions and data types, which add another layer of difficulty. For instance, reasoning about algorithms that manipulate linked lists requires a logical representation of lists to specify the behavior of such functions. Tools such as *CN* [19], *Viper* [17], and *VeriFast* [11] allow users to define logical data structures using *Algebraic Data Types* (ADTs) and to specify pure logical functions that operate on them. These constructs, however, need to be encoded into the SMT-LIB standard.
- **Performance Considerations** – SMT solvers can be a major bottleneck in symbolic execution. As such it is necessary to optimise our encoding and consider performance factors.
- **Soundness** – The soundness of the symbolic execution tool as a whole is dependent on the soundness of the encoding of symbolic state into SMT-LIB.

While prior work has focused on the operational semantics and soundness of compositional symbolic execution [13, 10, 28, 4], these formalisms are largely based on being able to query the satisfiability of symbolic states, abstracting away the specifics of how symbolic state is encoded into SMT-LIB queries. These formalisms do not handle user-defined functions or datatypes. They also do not handle the measures taken to handle dynamic typing and partiality in the encoding. They assume the existence of a sound encoding, but no work has formalised or proved the soundness of an encoding of symbolic state into SMT-LIB. While initially it may seem a straightforward translation, there are various pitfalls and edge cases in the encoding process. As such it is vital that we develop a rigorous formalism to reason about the soundness of the interaction between SMT solvers and CSE tools.

1.1. Contributions

This work addresses this gap by formalising the interface between CSE tools and SMT solvers, with a focus on how symbolic state is encoded and communicated between these systems. Concretely, we make the following contributions:

- In Chapter 3, we survey existing approaches of encoding symbolic state into SMT-LIB queries, across CSE tools. We explore some practical considerations of these encodings. In particular, we explore in Section 3.3 the approach taken by Gillian to handle dynamic typing. We explore the approaches taken by various CSE tools to encode user defined datatypes and functions in Section 3.4 and Section 3.5 respectively. We discover interestingly that different CSE tools are employing vastly different approaches, particularly in the encoding of user defined datatypes.

- In Chapter 4, we formally capture approaches taken by CSE tools in encoding symbolic state into SMT-LIB, proving a soundness result. In Section 4.1, we extend prior formalisations of CSE to capture the handling of typing during symbolic execution. We give also a formal semantics for an expression language that includes user-defined datatypes and functions. In Section 4.3, we give a formal encoding of symbolic state into SMT-LIB, exploring the handling of partiality, dynamic typing, user-defined datatypes and functions in this encoding. In Section 4.4, we bridge between the theory of CSE and the formal semantics of SMT-LIB, defining the machinery that allows us to reason about our encoding. Importantly, in Section 4.5, we prove our formal encoding to be both OX and UX sound. Finally, in Section 4.6 we apply our theory to Gillian, identifying unexpected design choices and potential soundness.

- In Chapter 5, we bring Gillian in-line with other modern CSE tools, adding support in Gillian for user-defined data types and functions, exploring details of our implementation.

2. Background

2.1. Separation Logic (SL)

An early formal method for proving program properties is *Hoare logic* [8]. Hoare logic provides a framework for reasoning about programs by defining a set of axioms and inference rules that determine the validity of proof derivations. These derivations allow us to establish the correctness of *Hoare triples*, which are expressed in the form:

$$\left\{P\right\} C \left\{Q\right\}$$

This notation means that "for any state satisfying the precondition P, the execution of the program C results in a state satisfying the postcondition Q." For example, a valid Hoare triple might be:

$$\left\{\mathsf{x} = x\right\} \mathsf{x} := \mathsf{x} + 1 \left\{\mathsf{x} = x + 1\right\}$$

It is important to note that Hoare logic is an over-approximating (OX) framework. This means that the postcondition of a Hoare triple represents a *superset* of all possible states that could be reached during the concrete execution of the program, starting from a state that satisfies the precondition. To illustrate this, consider the following valid Hoare triple:

$$\left\{\mathsf{x} = x\right\} \mathsf{x} := \mathsf{x} + 1 \left\{\mathsf{x} > x\right\}$$

Here, the postcondition describes states that include those achievable by concrete execution, but it also includes states that cannot actually occur in practice. In the context of verification, this is desirable. We want all possible terminating states to be encompassed in the post-condition. However, in the context of bug-finding this is undesirable. A bug may only occur in a subset of the terminating states – over-approximating frameworks don't allow us to specify / prove this behaviour.

A key limitation of Hoare logic becomes evident when reasoning about programs that interact with the heap. Suppose the heap is a simple mapping of natural numbers to values. Imagine we define a predicate, $\mathsf{list}(x)$, which represents the presence of a linked list starting at address x in the heap. It is difficult in Hoare logic to express that two lists are *distinct* and do not share any memory. For example, consider the assertion $\mathsf{list}(x) \wedge \mathsf{list}(y)$. There is no mechanism within Hoare logic to enforce that these are two separate lists, ensuring that the memory cells they occupy are disjoint. This limitation complicates reasoning about programs where the absence memory overlap is a critical property.

Separation logic [18, 24] is an extension of Hoare logic which addresses these issues. Much like Hoare logic which preceded it, separation logic is over-approximating. It introduces the

separating conjunction, $\star$. The assertion $P \star Q$ is satisfied by a heap which can be split into two *disjoint* heaps, which individually satisfy P and Q separately. This allows us to write assertions like $\mathsf{list}(x) \star \mathsf{list}(y)$ which captures the disjoint nature of the two heaps. It expresses that the two lists do not occupy any of the same memory. In addition, separation logic introduces spatial relations, such as the "points to" relation, where $x \mapsto y$ denotes that y is stored at address x in the heap.

The satisfiability judgement $\theta, s, \mu \models A$ denotes that a separation logic assertion A is satisfied by logical environment θ, variable store s, and a heap μ. The logical environment is a mapping from logical variables to values, whereas the variable store is a mapping from program variables to values. For example, the satisfiability of $\star$ and $\mapsto$ is given by:

$$\begin{aligned} \theta, s, \mu \models A_1 \star A_2 \Leftrightarrow & \quad \exists \mu_1, \mu_2 \in \mathsf{Heap}.\, \mu = \mu_1 \uplus \mu_2 \wedge \theta, s, \mu_1 \models A_1 \wedge \theta, s, \mu_2 \models A_2 \\ \theta, s, \mu \models E_1 \mapsto E_2 \Leftrightarrow & \quad h = \{[\![E_1]\!]_{\theta,s} \mapsto [\![E_2]\!]_{\theta,s}\} \end{aligned}$$

where $\uplus$ denotes the disjoint union of two heaps, i.e. their domain is disjoint, and $[\![E]\!]_{\theta,s}$ is the evaluation of a logical expression under the logical environment θ and variable store s.

A key result of separation logic is the frame rule. Since a separation logic triple guarantees that no heap resource, outside of the state described by the pre- and post-conditions, is modified or accessed by the program, we can arbitrarily add on additional heap resource:

$$\textsc{Frame}\ \frac{\{P\}\ C\ \{Q\}}{\{P \star F\}\ C\ \{Q \star F\}}$$

2.2. Incorrectness Separation Logic (ISL)

Whilst separation logic allows us to prove correctness properties, i.e. the *absence* of bugs, it is desirable in the context of bug-catching to be able to prove the *presence* of bugs. This motivates the development of *incorrectness separation logic* [20]. Contrary to separation logic, which is over-approximate, separation logic is under-approximate (UX). *ISL triples* have the form,

$$[P]\ C\ [\epsilon :\ Q]$$

where ϵ is the exit condition, which is either normal, ok, or erroneous, err. Since ISL is under-approximate, the result, Q, describes a subset of the states which are reachable from the presumption, P, by executing C. Intuitively, this allows us to specify and prove the presence of bugs - bugs don't occur in all possible executions of a program, but rather in a specific subset.

2.3. Symbolic Execution and Symbolic States

Symbolic execution is a method of automating this verification process [1] - deriving proofs manually is labourious and tedious, and in order to scale to large codebases automation is key. Symbolic execution operates over symbolic values as opposed to concrete values. Symbolic values are abstract, and represent a set of possible concrete values. As execution proceeds, restrictions

on these symbolic values are accumulated, for example on conditional branches.

A symbolic state typically consists of, $\hat{\sigma} = (\hat{s}, \hat{\mu}, \hat{\pi})$, consists of:

- A symbolic variable store $\hat{s}$, mapping program variables to symbolic values
- A symbolic heap $\hat{\mu}$, mapping symbolic values representing heap addresses, to symbolic values representing the contents of heap cells
- A symbolic path constraint, $\hat{\pi}$, which holds the accumulated constraints during symbolic execution

A symbolic state is said to be *satisfiable* if there is a concrete interpretation of the symbolic state for which the symbolic store and heap are well-formed and the path constraint is true:

$$\theta, s \models_{\mathsf{Sto}} \hat{s} \iff \forall v \in \mathsf{dom}(\hat{s}).\ [\![\hat{s}(v)]\!]_\theta = s(v)$$
$$\theta, \mu \models_{\mathsf{Mem}} \{a_1 \mapsto e_1, \ldots, a_n \mapsto e_n\} \iff \mu = \uplus_{i=1}^{n}\{[\![a_i]\!]_\theta \mapsto [\![e_i]\!]_\theta\}$$
$$\theta, s, \mu \models \hat{s}, \hat{\mu}, \hat{\pi} \iff \theta, s \models_{\mathsf{Sto}} \hat{s} \text{ and } \theta, \mu \models_{\mathsf{Mem}} \hat{\mu} \text{ and } [\![\hat{\pi}]\!]_\theta = \mathsf{true}$$

were θ is a mapping from symbolic variables to concrete values as before.

During symbolic execution, various commands may cause execution to *branch*. For example, consider a simple program:

$$\texttt{if}\ (\mathsf{c} >= 42)\ \{\ldots\}\ \texttt{else}\ \{\texttt{error}\}$$

This branches execution into three separate paths:

- A path where c is not a number, i.e. $\mathsf{c} \notin \mathsf{Int}$, in which execution results in an error.
- A path where $\mathsf{c} \geq 42$, in which the `if` branch is explored.
- A path where $\mathsf{c} < 42$, in which the `else` branch is explored, also resulting in a program error.

In contrast with concrete execution, in symbolic execution, all three branches are explored. The path constraint is updated accordingly within each execution branch.

2.4. Symbolic Execution and Satisfiability

Satisfiability checks are a key component of symbolic execution, though their role and requirements differ depending on the verification goal.

2.4.1. OX Satisfiability

In OX-style verification, it is sound to explore infeasible (unrealisable) paths, but desirable to prune clearly unsatisfiable branches to mitigate path explosion. Here, satisfiability checking serves a performance optimisation role rather than a soundness-critical one. Accordingly, we may use a fast over-approximation: the check may conservatively return `SAT` even when the state is actually unsatisfiable. However, the check must be cheap – if it takes longer than simply

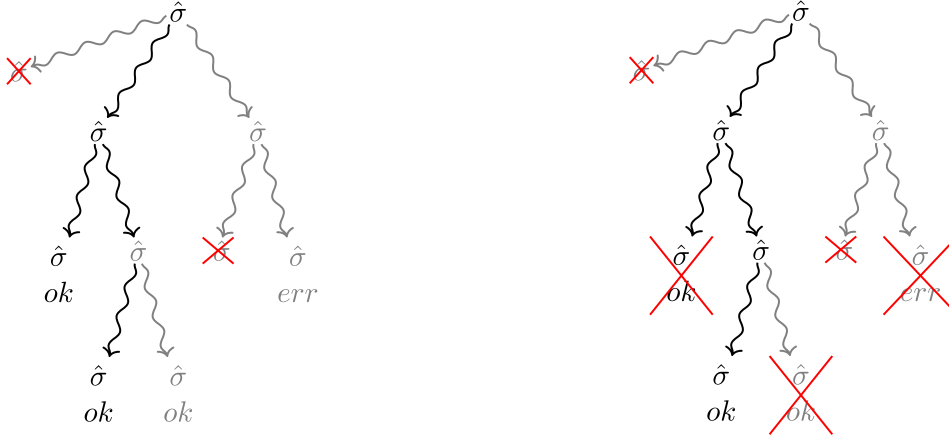

Figure 2.1. Left: OX satisfiability check to prevent path explosion. Right: UX satisfiability check to prevent unrealisable error reports. Grey states are not satisfiable, and crossed states are cut by SAT checks.

exploring the unsatisfiable branch, it undermines its purpose. In practice, such approximations might, for example, check only the satisfiability of the path condition. For the check to be OX sound, it must however return SAT for all satisfiable symbolic states.

2.4.2. UX Satisfiability

In contrast, in UX-style bug-finding, only realistic execution paths can justify a reported bug. That is, before reporting a bug, we must verify that the path leading to it is realisable. Here, satisfiability checking is required for soundness, not merely performance. As such, we need a sound under-approximation: if the check reports the state as satisfiable, it must genuinely be so. This requires examining the entire symbolic state, not just fragments like the path condition. As an example, consider the three branches of the program snippet presented above. The states which result in errors (i.e. where $\mathsf{c} \notin \mathsf{Int}$ or $\mathsf{c} < 42$) must be satisifiable before these errors can be soundly reported.

2.5. Compositional Symbolic Execution

A key problem with symbolic execution that hinders scalability is *path explosion* [1], with each branch potentially doubling the number of paths that need to be explored. One approach to overcoming path explosion is functional compositionality [13]. If a function specification is verified once, this specification can then be reused at all call-sites. This drastically reduces the number of paths that need to be explored - we need not symbolically execute every function within every possible path which calls it. Furthermore, it allows us to modularly verify our program - we can verify functions independently of the rest of the code base, which makes it more practical to use.

Most compositional symbolic execution engines use a *consume / produce* model in order to use SL / ISL function specifications in symbolic execution [13]. The consume operation removes from the symbolic state the necessary state to satisfy an assertion, e.g. a function pre-condition. Conversely, the produce operation adds to the symbolic state an assertion's (e.g. a function

post-condtion's) symbolic state. During these operations, it is again necessary to verify the satisfiability of different symbolic states.

Since function specifications may produce or consume *predicates*, the symbolic state is now modified to carry around predicates which describe

2.5.1. CSE and Entailment

During compositional symbolic execution, particularly when performing the consume operation, it is often necessary to determine whether a symbolic state entails a given logical expression. We say that a symbolic state $\hat{\sigma}$ entails a logical expression E, written $\hat{\sigma} \models E$, if E holds under all models of $\hat{\sigma}$:

$$\theta, \sigma \models \hat{\sigma} \implies [\![E]\!]_\theta = \mathsf{true}$$

This form of entailment is required in OX reasoning when consuming pure assertions. A pure assertion may only be consumed if the symbolic state guarantees that the assertion holds. Entailment checking can be reframed as a satisfiability problem: we ask whether there exists a model that satisfies the symbolic state but falsifies the assertion. If such a model exists, then the entailment does not hold.

For soundness, it is crucial that if the procedure reports that $\hat{\sigma} \models E$, then this must truly be the case.

2.6. CSE Tools

We briefly detail a few relevant CSE tools.

2.6.1. The Gillian Framework

Gillian [6, 16] is a language-independent platform for the development of compositional symbolic execution tools. Gillian is parametric on the memory model of the target language. Gillian supports whole program symbolic testing, OX verification and UX bug-detection. Gillian must be instantiated on a target language, by instantiating the parametric memory model for that target language, as well as providing a compiler from the target language to *GIL*, the small goto language which underpins symbolic execution in Gillian. Currently there are Gillian instantiations for C and JavaScript, as well as for *WISL*, a simple while-like language with a linear heap memory model.

Crucially, encoding of symbolic state occurs in both the core part of Gillian, as well as in the insantiated part. Consider the $\models_{\text{Mem}}$ relation introduced in Section 2.3. This relation is parametric on the memory model – it varies between different memory models. In order to determine the satisfiability of a symbolic state, we must be able to determine whether the symbolic memory is satisfiable. Since the semantics of satisfiability of symbolic memory differs between instantiations, this encoding of symbolic memory into SMT queries must occur within each instantiation.

2.6.2. Viper

Viper [17] is a verification infrastructure aimed at supporting permission-based reasoning for concurrent and sequential programs. It provides a rich intermediate verification language and tools for encoding high-level verification conditions. Viper supports compositional reasoning via separation logic and permission logics. Its architecture facilitates backend reuse across different source languages through front-end translators.

2.6.3. VeriFast

VeriFast [11] is a tool for the modular verification of safety properties in single- and multithreaded C and Java programs. It relies on symbolic execution in combination with separation logic and user-supplied annotations to verify memory safety and functional correctness. VeriFast performs compositional symbolic execution by interpreting specifications of procedures as contracts, enabling scalable analysis of large programs.

2.6.4. CN

CN (short for Compositional Nominal) [19] is a tool for compositional symbolic execution of heap-manipulating programs. It introduces a novel technique for representing and manipulating heap summaries using nominal techniques, facilitating modular verification. CN is designed to support reasoning about programs with dynamically allocated, recursive data structures.

2.7. The SMT-LIB Standard

CSE tools typically make use of satisfiability modulo theory (SMT) solvers for the purpose of querying the satisfiability of symbolic states. The SMT-LIB standard [3] is a standardized language and framework designed to facilitate the specification and exchange of problems in the field of satisfiability modulo theories. SMT-LIB is based on many-sorted first-order logic and supports a range of theories such as linear arithmetic, bit-vectors, arrays, and uninterpreted functions. It allows for the seamless encoding of formulas that combine multiple theories, making it versatile for a wide range of applications. The language employs a Lisp-like syntax, using symbolic expressions (sexpressions) to define sorts (types), declare variables, and construct formulas. At a high level, a query is a sequence of declarations of constants, functions, etc. followed by asserting a list of constraints over these symbols. When the `check-sat` command is called, the solver will find an interpretation of the declared symbols which satisfies the constraints imposed. Calling `get-model` will return this interpretation.

2.7.1. SMT-LIB Theories

The standard supports a range of theories. We briefly cover the relevant theories.

Uninterpreted Functions

The basic building blocks of SMT formulas are functions. Even constants are functions which take no arguements. Functions in SMT-LIB are total and pure. An uninterpreted function is one

which has no interpretation attached to it. We can make assertions about these uninterpreted functions, detailing it's behaviour. The solver will attempt to interpret this function under the constraints imposed on it. If no such interpreted function exists, the solver returns `unsat`.

```
(declare-sort A)
(declare-fun f (A) A)
(declare-const x A)
(assert (= (f (f x)) x))
(assert (= (f (f (f x))) x))
(check-sat)
(get-model)

(assert (not (= (f x) x)))
(check-sat)
```

In this example, the first satisfiability check returns `sat`, whereas the second return `unsat`.

Arithmetic

SMT-LIB supports the theory of integers, and the theory of reals:

```
(declare-const a Int)
(declare-const b Int)
(declare-const c Int)
(declare-const d Real)
(declare-const e Real)
(assert (< a (+ b 2)))
(assert (= a (+ (* 2 c) 10)))
(assert (= (+ c b) 1000))
(assert (= d e))
(check-sat)
(get-model)
```

One important consideration to note is that division, like all functions, is total. Thus division by zero is allowed, but the result is not specified. For example:

```
(declare-const a Real)
(assert (= (/ x 0.0) 0.0))
(check-sat)

(assert (= (/ x 0.0) 1.0))
(check-sat)
```

The first call to `check-sat` returns `sat`, as the solver will interpret division by zero as being defined to equal `0.0`. The second satisfiability check, however, will return `unsat`, since `x` divided by zero can't equal both `0.0` and `1.0`.

Bitvectors

Similar to the theory of integers, SMT-LIB supports the theory of fixed size bitvectors. It supports a number of arithmetic and bitwise operators on bitvectors.

2.7.2. Function Definitions and Termination

While the SMT-LIB standard defines syntax to declare *uninterpreted functions*, it also provides syntax to define functions with some interpretation:

```
(define-fun foo ((x Int) (y Int)) Int (x + y))
```

A different syntax is used to define mutually recursive functions:

```
(define-funs-rec
  ((foo (Int Int) Int) (bar (Int Int) Int))
  ( * Definition of foo * )
  ( * Definition of bar * ))
```

However, this is semantically equivalent to the following:

```
(declare-fun foo (Int Int) Int)
(declare-fun bar (Int Int) Int)
(forall ((x Int) (y Int)) (= (foo x y) ( * Definition of foo * )))
(forall ((x Int) (y Int)) (= (bar x y) ( * Definition of bar * )))
```

A side-effect of this semantics of function definitions is that the notion of non-termination is meaningless. To illustrate this, consider the following function, which simply calls itself:

```
(define-funs-rec
  ((non-terminating (x Bool) Bool) (non-terminating x))
```

While this function might seem non-terminating, this is in fact equivalent to:

```
(declare-fun non-terminating (Bool) Bool)
(forall (x Bool) (= (non-terminating x) (non-terminating x)))
```

As such, any interpretation of the function satisfies the assertion above, since it is a validity.

2.7.3. Quantification

We observe above that the use of mutually recursive functions necessitates universal quantification. Quantification – both universal and existential – introduces expressiveness at the cost of solver performance and predictability. While quantifiers are necessary, e.g. to encode recursive functions, they pose significant challenges for SMT solvers. In particular, automated reasoning with quantifiers is undecidable in general, and solvers rely on heuristic techniques [21].

The most common of these heuristics is *E-matching* [25]. It works by identifying characteristic sub-expressions of quantified formulae, named *triggers*, which are matched during proof search

on ground terms to discover relevant instantiations of the quantified formula. E-matching is used in a number of solvers, including Z3 [5] and CVC5 [2].

To give a concrete example, consider this query involving universal quantification:

```
(declare-fun f (Int) Int)
(declare-fun g (Int) Int)
(declare-const a Int)
(declare-const b Int)
(declare-const c Int)
(assert (forall ((x Int))
              (! (= (f (g x)) x)
                 :pattern ((f (g x))))))
(assert (= (g a) c))
(assert (= (g b) c))
(assert (not (= a b)))
(check-sat)
```

The universally quantified assertion states that the function `f` is the inverse of `g`. The `:pattern` annotation allows us to manually specify the trigger. This query should not be satisfiable, since `g` is not injective, and so does not have an inverse. However, because of our choice of trigger, the quantification is only instantiated whenever the term `(f (g x))` appears. Since that term does not appear in our assertions, the quantification is not instantiated, and so the query is in fact satisfiable. If we instead choose the trigger `(g x)`, the quantification is instantiated over both `a` and `b`, thus rendering the query unsatisfiable.

This illustrates the difficulty (in general) of selecting good triggers for universal quantification. A trigger that is more restrictive (e.g. `(f (g x))` in our example) will mean the quantification is instantiated a fewer number of times, potentially increasing the performance of the query, but also making the solution less complete. Triggers can be specified manually, as seen in the example above, or inferred by the solver.

2.7.4. ADTs

SMT-LIB supports natively syntax for defining algebraic datatypes:

```
(declare-datatypes (T) ((Lst nil (cons (hd T) (tl Lst)))))
```

These can be recursive, or even mutually recursive. Here, `cons` and `nil` are referred to as *constructors*, whereas `hd` and `tl` are selectors. There is also syntax for recognizing terms that were built by some constructor:

```
(declare-const l Lst)
(assert ((_ is nil) l))
```

This asserts that the list `l` is empty.

Lazy Approach to ADTs

Most mainstream SMT solvers, including Z3 and CVC5, handle algebraic data types (ADTs) using a lazy approach [22]. In this setting, reasoning about ADTs is done incrementally and on demand. The solver keeps the formula in its original ADT form and invokes a specialised theory solver for ADTs as needed during the solving process. This theory solver uses techniques such as congruence closure, unification, and acyclicity checks to reason about ADT constructors, selectors, and recognisers. The key idea in the lazy approach is delayed instantiation: instead of expanding all ADT-related constraints upfront, the solver waits until it becomes necessary – for example, when a conflict is detected or a decision needs to be propagated. This helps avoid unnecessary work but comes at the cost of unpredictability and scalability issues, especially in deeply recursive or complex instances. Shah, Mora, and Seshia [26] found that state-of-the-art lazy solvers struggled to solve many challenging queries even with generous timeouts.

Eager Approach to ADTs

By contrast, the approach introduced by Shah, Mora, and Seshia [26] is fundamentally eager. Their prototype solver, Algaroba, tackles ADT queries by entirely eliminating ADT constructs before solving begins. Instead of using ADT-specific reasoning during solving, it reduces the problem to an equivalent one in a simpler theory – specifically, the theory of uninterpreted functions (UF). This is done through a systematic translation that replaces constructors, selectors, and recognisers with uninterpreted function symbols, and encodes the necessary ADT axioms (such as acyclicity) as quantifier-free constraints. A novel aspect of their work is the use of bounded unfolding based on the query depth. They prove that for any quantifier-free ADT query, there is a finite unfolding depth sufficient to encode all necessary properties – enabling them to avoid universal quantifiers altogether. This allows them to preserve decidability and use existing UF solvers without requiring specialised ADT-handling machinery. We note that their solver, Algaroba, does not support many useful basic theories including arithmetic, nor does it support quantification. It supports only the theory of ADTs.

Non-Standard Theories

There are a number of non-standard theories that are relevant to the encoding of symbolic state into SMT-LIB:

- **Sequences** – This theory adds support for reasoning about ordered sequences of elements.
- **Separation Logic** – This theory adds support for reasoning about separation logic assertions. Our work interacts with the encoding of spatial assertions into SMT-LIB. Developing a separation logic theory in SMT-LIB is an area that has been studied previously [23], with support for this theory in mainstream SMT solvers such as CVC5 [2]. However, we highlight a key distinction between our works, in that the prior works are not memory-model parametric, typically supporting only “points-to” spatial assertions (i.e. a linear memory model).

These non-standard theories aren't supported by SMT-LIB, but may have support from specific SMT solvers.

2.7.5. SMT Solvers

There are a number of existing solvers which adhere to the SMT-LIB standard. These solvers differ in their supported theories and optimizations. Relevant SMT solvers include: Z3 [5], developed by Microsoft Research, is a high-performance SMT solver supporting a wide range of theories, including non-standard theories such as sequences; CVC5 [2], the successor to CVC4, is an efficient and highly configurable SMT solver supporting the theories of both sequences and separation logic; and Algaroba [27] a solver that focuses on a novel, more performant aproach to handling ADTs. It does this by reducing ADTs to the theory of uninterpreted functions.

3. A Survey of CSE Tools

In this chapter, we survey existing approaches employed by various CSE tools in the encoding of SMT-LIB. We begin by identifying specifically what makes the encoding into SMT-LIB complicated.

3.1. Key Challenges

3.1.1. Typing

In order to support symbolic execution of dynamically typed languages, some tools such as Gillian are dynamically typed. As detailed above, SMT-LIB is statically typed. This complicates the encoding strategy.

3.1.2. Partiality

All functions in SMT-LIB are total. As such, care must be taken when encoding functions which are partial, such as division. The semantics of our source expression language must be preserved in the encoded SMT-LIB query. For example consider a symbolic state $\hat{\sigma}$ whose path condition is $x/0 = 1$. There is no interpretation of the logical variable x under which this expression evaluates to true. Thus, by our definition of symbolic state satisfiability, this symbolic state is not satisfiable. A naive encoding to SMT-LIB might not consider this partiality. However, because of the semantics of SMT-LIB, this leads to a satisfiable query. While it is OX sound to incorrectly classify symbolic states as satisfiable, this is not UX sound.

3.1.3. User-Defined Functions and Data Types

To reason about datastructures, we must be able to model their contents logically. Consider for example, a mathematical list, given by the grammar:

$$\alpha ::= \epsilon \mid a : \alpha$$

This allows us to define a separation logic predicate which describes a singly linked list,

$$\mathsf{list}(E, \alpha) \stackrel{\text{def}}{=} (E = \texttt{null} \star \alpha = \epsilon) \ \vee (\exists a, \alpha', y.\, E \mapsto a, y \star \mathsf{list}(y, \alpha') \star \alpha = a : \alpha')$$

We may also wish to define functions over our logical datastructures, for example,

$$\begin{aligned} \mathsf{append}(\epsilon, a) &= a : \epsilon \\ \mathsf{append}(a' : \alpha, a) &= a' : \mathsf{append}(\alpha, a) \end{aligned}$$

In order to maximise flexibility, tools such as CN, VeriFast and Viper allow users to define their own logical datastructures, through algebraic datatypes (ADTs). These tools also allow users to define pure functions which operate on these ADTs. These user-defined ADTs and functions may appear in predicates, and so it is necessary to be able to encode these into SMT-LIB.

As a further complication, we must consider that these user-defined functions may be partial. For example, consider this simple function:

$$\mathsf{div}(x, y) = x/y$$

Clearly this function is not well-defined for $y = 0$. As discussed in Section 3.1.2, all SMT-LIB functions must be total, so partiality in user-defined functions must be accounted for.

3.1.4. Performance Considerations

When emitting SMT-LIB commands to a solver, it is vital that we consider and optimise for performance. This includes considering various factors which affect performance, such as the size of the query, as well as the theories we utilise in our encoding. For example, it is desirable to avoid using ADTs where possible. Another consideration is that the use of quantifiers in our encoding decreases performance, and makes the query undecidable.

3.2. Survey Methodology

We have investigated current approaches taken by existing tools to address the challenges we identified. We have done this by intercepting SMT-LIB commands generated by various symbolic execution tools, including Gillian, CN, Viper and VeriFast. Throughout this investigation, we worked with a portable example, attempting to verify some basic operations on singly linked lists. The type signatures of these operations are given below:

```c
typedef struct ln {
    int data;
    struct ln *next;
} SLL;

int listLength(SLL *x);
SLL *listAppend(SLL *x, int v);
SLL *listPrependV(SLL *x, int v);
SLL *listCopy(SLL *x);
SLL *listConcat(SLL *x, SLL *y);
```

A summary of the existing tools we have investigated, as well as aspects of them that are relevant to the encoding of symbolic state into SMT-LIB is detailed in Table 3.1. By in-built data types, we refer to datatypes that the tools provides natively, without the user having to provide a definition.

None of the tools that support user-defined functions enforce totality. VeriFast and CN permit operations like division within function definitions, which can introduce partiality. Viper, on the

Table 3.1. Relevant Aspects of Existing Tools

	Gillian	**CN**	**Viper**	**VeriFast**
Typing	Dynamic	Static	Static	Static
User-Def. Functions	✗	✓	✓	✓
User-Def. Data Types	✗	ADTs	ADTs / *Domains*	ADTs
In-Built Data Types	Lists, Sets	✗	Lists, Sets, Maps, ...	Lists, Options, Pairs, ...

other hand, allows preconditions to be specified for functions, ensuring that a function cannot be invoked unless its preconditions are satisfied.

Viper is unique in that it supports a construct called *domains*. Domains enable the definition of additional types, mathematical functions, and axioms that provide their properties . Syntactically, domains consist of a name (for the newly-introduced type), and a block in which a number of function declarations and axioms can be introduced. For example, we may define a list using a domain as follows:

```
domain List[T] {
  /* Constructors */
  function Nil(): List[T]
  function Cons(head: T, tail: List[T]): List[T]
  /* Destructors */
  function head_Cons(xs: List[T]): T // requires is_Cons(xs)
  function tail_Cons(xs: List[T]): List[T] // requires is_Cons(xs)
  /* Recognisers */
  function is_Nil(xs: list): Bool
  function is_Cons(xs: list): Bool
  /* Constructor types */
  function type(xs: List[T]): Int
  unique function type_Nil(): Int
  unique function type_Cons(): Int

  function length(xs: list): Int

  /* Axioms */
  axiom destruct_over_construct_Cons {
    forall head: Int, tail: list :: {Cons(head, tail)}
         head_Cons(Cons(head, tail)) == head
      && tail_Cons(Cons(head, tail)) == tail
  }

  /* More axioms ... */
}
```

Viper also supports ADTs, but this is syntactic sugar for these domains.

VeriFast provides several built-in data types, all of which are defined internally using ADT syntax in a prelude that is automatically imported. The encoding of these in-built data types

aren't treated specially – they are encoded as user-defined ADTs.

3.3. Approaches to Handle Typing

Tools such as CN, Viper and VeriFast which are statically typed have an advantage, in that typing is not something that presents a significant challenge in the encoding process. The drawback with this statically typed approach is that it is restrictive – these tools can only target statically typed languages.

Gillian takes two approaches to bridge the disparity between its dynamically typed environment, and SMT-LIB's statically typed one. The first, more naive, approach it takes is that it defines an ADT within SMT-LIB to represent a literal:

```
(declare-datatype GIL_Literal
 ((Undefined) (Null) (Empty) (Bool (bValue Bool)) (Int (iValue Int))
  (Num (nValue Real)) (String (sValue Int)) (Loc (locValue Int))
```

We remark that this is analagous to the σ_{LVal} ADT introduced in the formal account of the encoding. The use of ADTs undesirable for performance reasons, so wherever possible Gillian avoids using `GIL_Literals`. If possible, it is desirable to declare constants and functions using native types to improve performance. In order to achieve this, Gillian performs some type inference. Wherever the type can be infered, it is used. We see many examples of this in the SMT-LIB commands which Gillian generated:

```
(declare-const _lvar_80 Int)
```

3.4. Approaches to Handle User-Defined Datatypes

3.4.1. Gillian

Gillian does not currently support user-defined ADTs or functions. It does however have in-built support for reasoning with lists and sets. These are encoded using Z3's in-built `Seqs`:

```
(declare-const _lvar_176 (Seq GIL_Literal))
```

We also observe how it makes use of the `GIL_Literal` ADT described above in order to achieve a dynamically typed list.

3.4.2. CN

CN on the other hand, does support user defined datatypes. They can be expressed in the following syntax.

```
datatype List {
  Nil {},
  Cons {i32 Head, datatype List Tail}
}
```

The ADTs are encoded directly using SMT-LIB's ADTs:

```
(declare-datatypes ((List 0))
 (((Nil)
   (Cons (Head (_ BitVec 32)) (Tail List)))))
```

3.4.3. VeriFast

Interestingly, VeriFast takes a different approach to encoding user-defined ADTs. Jacobs, Smans, and Piessens [9] detail how the canonical list ADT would be encoded as an uninterpreted function with the signature:

$$\mathsf{nil} : \mathsf{List}$$
$$\mathsf{cons} : \mathbb{Z} \times \mathsf{List} \to \mathsf{List}$$

along with a set of axioms to ensure that the functions behave like constructors for an ADT. This approach is reminiscent of the theory reduction detailed by Shah, Mora, and Seshia [27]. Whilst the VeriFast paper likely took this approach because it was written prior to the introduction of ADTs into SMT-LIB, there may be performance related motivation. One key difference between this and the eager approach proposed by Shah, Mora, and Seshia [27], is that VeriFast uses universal quantifiers to encode the axioms of ADTs, whereas Shah, Mora, and Seshia [27] use a novel approach to avoid quantification completely.

3.4.4. Viper

Viper takes a similar approach when encoding domains and ADTs. Again, they encode ADTs as uninterpreted functions, along with a set of axioms, making use of quantifiers to encode these axioms. Consider again the list domain introduced above. As with VeriFast, these functions are encoded as uninterpreted functions. We note in particular the use of quantifiers, along with triggers, to encode the axioms.

```
(declare-const Nil<list> list)
(declare-fun Cons<list> (Int list) list)
;; ...
(assert (distinct type_Nil<Int> type_Cons<Int>))
(assert (forall ((head Int) (tail list)) (!
  (and
    (= (head_Cons<Int> (Cons<list> head tail)) head)
    (= (tail_Cons<list> (Cons<list> head tail)) tail))
  :pattern ((Cons<list> head tail))
  :qid |prog.destruct_over_construct_Cons|)))
;; ...
```

Viper also supports syntax for ADTs:

```
adt list {
   Nil()
   Cons(value: Int, tail: list)
}

function length(l: list): Int {
  l.isNil ? 0 : 1 + length(l.tail)
}
```

Internally these are translated into domains as presented above. As such, the encoding of ADTs into SMT-LIB follows a similar approach:

```
(declare-const Nil<list> list)
(declare-fun Cons<list> (Int list) list)
(declare-fun get_list_value<Int> (list) Int)
(declare-fun get_list_tail<list> (list) list)
(declare-fun list_tag<Int> (list) Int)
(assert (forall ((value Int) (tail list)) (!
  (= value (get_list_value<Int> (Cons<list> value tail)))
  :pattern ((Cons<list> value tail))
  )))
(assert (forall ((value Int) (tail list)) (!
  (= tail (get_list_tail<list> (Cons<list> value tail)))
  :pattern ((Cons<list> value tail))
  )))
(assert (forall ((t list)) (!
  (or
    (= t (as Nil<list> list))
    (= t (Cons<list> (get_list_value<Int> t) (get_list_tail<list> t))))
  :pattern ((list_tag<Int> t))
  :pattern ((get_list_value<Int> t))
  :pattern ((get_list_tail<list> t))
  )))
(assert (= (list_tag<Int> (as Nil<list> list)) 1))
(assert (forall ((value Int) (tail list)) (!
  (= (list_tag<Int> (Cons<list> value tail)) 0)
  :pattern ((Cons<list> value tail))
  )))
```

Viper automatically generates functions and axioms to implement the ADT. Viper also supports in-built datastructures. For example, we could use Viper's in-built `Seq` (list) data structure to specify our `listLength` function:

```
function listContents(this: Ref): Seq[Int]
/* ... */

function listLength(this: Ref): Int
```

```
  requires acc(list(this))
  ensures result == |listContents(this)|
{
  unfolding acc(list(this)) in
  this == null ? 0 :
                1 + listLength(this.next)
}
```

Viper, like Gillian, uses Z3's in-built `Seqs` to encode this:

```
(declare-fun listContents ($Snap $Ref) Seq<Int>)
```

3.4.5. Axiomatising ADTs

Table 3.2. ADT Axioms by Tool

	VeriFast	Viper	Z3	Algaroba	SMT-LIB Standard
Disjointness	✓	✓	✓	✓	✓
Injectivity	✓	✓	✓	✓	✓
Exhaustiveness	✗	✓	✓	✓	✗
Acyclicity	✗	✗	✓	✓	✗
Selector / Tester Axioms		✓	✓	✓	✓

We observe through our survey that various CSE tools attempt to encode ADTs as uninterpreted functions, along with some axioms to ensure that these functions behave as ADTs do. However, despite the common approach, each tool appears to encode different axioms. Table 3.2 summarises which axioms are captured in which contexts. The ADT axioms that appear across the various contexts are:

- **Disjointness** – This that the ranges of different constructors of the same ADT are disjoint. In the example of lists:

$$\forall x, \vec{x}.\ (\mathsf{nil} \neq \mathsf{cons}(x, \vec{x}))$$

- **Injectivity** – Constructors map distinct inputs to distinct outputs:

$$\forall x, y, \vec{x}, \vec{y}.\ \mathsf{cons}(x, \vec{x}) = \mathsf{cons}(y, \vec{y}) \implies x = y \land \vec{x} = \vec{y}$$

- **Acyclicity** – A constructor cannot be equal to one of it's "children". For example:

$$\mathsf{cons}(x, \mathsf{cons}(y, z)) \neq z$$

- **Selector / Tester Axioms** – These are axioms that govern the behaviour of *selectors* and

testers. A selector destructs a constructor:

$$\mathsf{head}(\mathsf{cons}(x, \vec{x})) = x$$
$$\mathsf{tail}(\mathsf{cons}(x, \vec{x})) = \vec{x}$$

A tester (interchangably refered to as a recogniser) returns a boolean value:

$$\mathsf{isCons}(\mathsf{cons}(x, \vec{x})) = \mathsf{true} \qquad \mathsf{isCons}(\mathsf{nil}) = \mathsf{false}$$
$$\mathsf{isNil}(\mathsf{nil}) = \mathsf{true} \qquad \mathsf{isNil}(\mathsf{cons}(x, \vec{x})) = \mathsf{false}$$

We see in the intercepted Viper SMT queries that it encodes exhaustiveness, disjointness, and selector / tester axioms. These selector axioms imply injectivity. VeriFast encode disjointness and injectivity [9][1]. Surprisingly, the SMT-LIB standard does not semantically enforce acyclicity or exhaustiveness using the `declare-datatypes` syntax. Despite this, Algaroba [27] and Z3 (in our experimentation) do indeed adhere to all axioms [2].

Importantly, all of these axioms are required to exactly model the behaviour of ADTs. While it is OX-sound to encode some subset of these axioms, it is not UX-sound to do so.

3.5. Approaches to Handle User-Defined Functions

3.5.1. CN

In addition to user-defined datatypes, CN supports user-defined functions. They can be defined with the following syntax:

```
function [rec] (datatype List) Append(datatype List L1, datatype List L2) {
  match L1 {
    Nil {} => {
      L2
    }
    Cons {Head : H, Tail : T} => {
      Cons {Head: H, Tail: Append(T, L2)}
    }
  }
}
```

An interesting point to note is that recursive functions in CN must be manually unfolded. An example of this can be seen in this example, where we verify a list concatenation function.

```
SLL *listConcat(SLL *x, SLL *y)
/*@ requires take L1 = SLList_At(x); take L2 = SLList_At(y);
    ensures take L3 = SLList_At(return); L3 == Append(L1, L2);
@*/
```

[1] Jacobs, Smans, and Piessens [9] do not clarify whether their system supports selectors and testers, or whether it encodes the corresponding axioms.

[2] We have not succeeded in finding documentation about Z3s behaviour, and its divergence from the standard here.

```
{
    SLL *r;
    if (x == NULL) {
        /*@ unfold Append(L1, L2); @*/
        r = y;
    } else {
        /*@ unfold Append(L1, L2); @*/
        SLL *c = listConcat(x->next, y);
        x->next = c;
        r = x;
    };
    return r;
}
```

The encoding of the append function in SMT-LIB is as follows:

```
(declare-fun Append (List List) List)
(declare-fun H () (_ BitVec 32))
(declare-fun T () List)
(declare-fun default_uf () List)
(assert
 (= (Append L1 L2)
  (let ((match L1))
   (ite ((_ is Nil) match) L2
    (ite ((_ is Cons) true)
     (let ((H (Head match)) (T (Tail match)))
      (Cons H (Append T L2)))
     default_uf)))))
```

We notice that this function's definition is only instantiated for specific lists, `L1` and `L2`. This is a consequence of the manual unfolding of functions in CN, and this method of encoding cannot be applied generally.

3.5.2. VeriFast

VeriFast takes a similar approach to CN in encoding user-defined functions. However instead of instantiating the definition for specific inputs, the definition is instantiated across all possible inputs with universal quantification. For example, the list length function in VeriFast syntax is given by:

```
fixpoint int length(list l) {
  switch(l) {
    case nil: return 0;
    case cons(h, t):
      return 1 + length(t);
  }
}
```

This example would be encoded as the two axioms over an uninterpreted function:

$$\mathsf{length}(\mathsf{nil}) = 0$$
$$\forall h : \mathsf{Int}, t : \mathsf{List}.\ \mathsf{length}(\mathsf{cons}(h, t)) = 1 + \mathsf{length}(t)$$

We note in particular the use of quantification here.

VeriFast enforces structural recursion in its “fixpoint” definitions [9]. It enforces that the fixpoint’s body is a switch statement over one of its parameters, called the *inductive parameter*. It requries that the value of the inductive parameter of a recursive call is a constructor argument of the value of the inductive parameter of the caller. Further, it restricts fixpoints to direct recursion, not supporting mutually recursive functions. In doing so, it ensures that fixpoints are terminating. We recall that a similar restriction is imposed in our formalism. We recall that this is an important restriction, since termination is a non-sensical concept in the semantics of SMT-LIB functions.

4. Formalising the Encoding of Symbolic State into SMT-LIB

In this chapter, we develop a formal account of how symbolic program states are encoded into SMT-LIB in order to determine their satisfiability. This formalisation is designed to be general enough to capture the approaches used in existing CSE tools, including Gillian. Lööw et al. [14] provide a formal foundation for memory-model parametric CSE platforms, inspired by Gillian. This formalisation is dependent on being able to determine the satisfiability of a symbolic state, denoted $\mathsf{SAT}(\hat{\sigma})$. We extend this formalisation to give an encoding of symbolic state into SMT-LIB proving soundness of this encoding with respect to the logical semantics of SMT-LIB.

4.1. Formalising CSE

We begin by covering the relevant aspects of the formalism from past works.

4.1.1. Concrete State

The CSE engine presented in [14] analyses a memory-model parametric version of a standard "while language". The semantics of this language can be defined in terms of its *concrete state*.

We define Val to be the set of values which are manipulated by programs in this language:

$$v \in \mathsf{Val} \quad ::= \quad n \in \mathsf{Int} \mid b \in \mathsf{Bool}^1$$

and denote program variables as $\mathsf{x} \in \mathsf{PVar}$.

A program state is a pair $\sigma = (s, \mu)$, where $s : \mathsf{PVar} \rightharpoonup_{fin} \mathsf{Val}$ [2] is a variable store mapping program variables to values, and $\mu \in \mathsf{CMem}$ represents the heap. To achieve memory-model parametricity, the nature of the heap is defined by each instantiation:

Instance Definition 1. *We require a tuple* $(\mathsf{CMem}, \mu_\emptyset, \cdot, \mathcal{W}f)$*, where* CMem *is a set of memories,* $\mu_\emptyset \in \mathsf{CMem}$ *the empty-memory element,* $\cdot : \mathsf{CMem} \times \mathsf{CMem} \rightharpoonup \mathsf{CMem}$ *is a memory composition operator, and* $\mathcal{W}f \subseteq \mathsf{CMem}$ *is a well-formedness predicate. The empty memory must be well-formed and composition must maintain well-formedness.*

Instance Definition Example 1. *To illustrate what instance definitions might look like in practice, we introduce a running example of a simple, linear memory model. In our linear model,* CMem *is* $\mathsf{Int} \rightharpoonup_{fin} (\mathsf{Val} \uplus \{\varnothing\})$*, where the symbol* $\varnothing$ *records that a memory cell has been freed. The empty*

[1] We differ from Lööw et al. [14] in our definition of Val – we work with a subset set of program values (e.g. we don't include the rationals Rats), in order to simplify the presentation of this formalism.

[2] $X \rightharpoonup_{fin} Y$ denotes partial functions from X to Y with finite support.

memory $\mu_\emptyset$ is the empty function and the composition of two memories μ and μ' is their disjoint union $\mu \uplus \mu'$. All memories are well-formed, i.e., $\mathcal{W}f = \mathsf{CMem}$.

4.1.2. Logical Expressions

We detail here the logical expression language. Importantly, we extend prior formalisations [13, 10, 28, 4] by supporting ADTs and user-defined functions.

Logical Values

During symbolic execution, the CSE engine operates on logical expressions. Logical expressions evaluate to logical values:

$$v \in \mathsf{LVal} ::= u \in \mathsf{Val} \mid [\vec{v}] \mid C(\vec{v})$$

Logical values extend the set of concrete values to include logical constructs, such as lists or datatype constructors. We differ from Lööw et al. [14] in our separation of logical and concrete values. We highlight that concrete program execution does not involve logical values. Logical values only arise during symbolic execution, as a result of specifications which involve these logical constructs.

Types

Logical expressions and values can be assigned a type. We write $v \in \tau$ to mean that the logical value v has a type τ. The set of types Tys is given by:

$$\tau \in \mathsf{Tys} ::= \mathsf{Int} \mid \mathsf{Bool} \mid \mathsf{List} \mid \tau \in \mathsf{Datatype} \mid \mathsf{LVal}$$

We extend the types presented by Lööw et al. [14] to align with our modified expression language. One key difference is the introduction of the type LVal, representing that no type restriction exists.

User-Defined Datatypes

We also extend the logical expression language presented by Lööw et al. [14] to support algebraic datatypes (ADTs). An algebraic datatype $\tau \in \mathsf{Datatype}$ is constructed by some set of constructors $C(\vec{\tau}) : \tau \in \mathsf{Constructor}$, where Constructor is the set of datatype constructor declarations. The logical value $C(v_1, \ldots, v_n)$ has some type $\tau \in \mathsf{Datatype}$ iff $C(\tau_1, \ldots, \tau_n) : \tau \in \mathsf{Constructor}$ and $v_i \in \tau_i$ for each $i = 1, \ldots, n$.

Logical Expressions

Logical expressions can then be defined as:

$$
\begin{aligned}
E \in \mathsf{LExp} ::= {} & v \in \mathsf{LVal} \mid x \in \mathsf{LVar} \mid \mathsf{x} \in \mathsf{PVar} \\
& \mid E \oplus E \mid \ominus E \mid E \in \tau \\
& \mid E : E \mid E[E] \\
& \mid C(\vec{E}) \mid \mathsf{get}C_i(E) \mid \mathsf{is}C(E) \\
& \mid f(\vec{E}) \mid E \mathbin{?} E \,:\, E
\end{aligned}
$$

Logical expressions involve logical values, logical variables, and program variables. Logical variables only occur in logical expressions, and are not part of the semantics of concrete execution. Logical expressions involve a standard set of numerical and boolean binary / unary operators. The logical expression $E \in \tau$ checks the type of a logical expression, evaluating to true or false. In addition, logical expressions include some list operations, including prepending and access. Importantly, we extend the logical expressions of [14] to include operations on ADTs. $C(\vec{E})$ represents the application of an ADT constructor, $\mathsf{get}C_i(E)$ is a *selector*, and $\mathsf{is}C(E)$ is a *recognizer*.

User-Defined Functions

Our formalism also extends past work to support user-defined functions. As such, logical expressions also involve applications of user-defined functions $f(\vec{E})$, where f is defined by $f(\vec{x} : \vec{\tau})\{E\} \in \mathsf{Func}$, and Func is the set of user-defined function definitions. This denotes a function with name $f \in \mathsf{Str}$, with the parameters $\vec{x} \in \vec{\mathsf{LVar}}$, where the parameter's types are restricted to $\tau_i \in \mathsf{Tys}$. When a parameter's type is given as $x : \mathsf{LVal}$, this is taken to mean there are no type restrictions on this parameter. Lastly, we add *if-then-else* expressions, denoted $E \mathbin{?} E_1 \,:\, E_2$.

Logical Expression Evaluation

The semantics of logical expression evaluation is defined with respect to an interpretation of logical variables $\theta : \mathsf{LVar} \rightarrow \mathsf{LVal}$ [3], mapping logical variables to logical values, and a concrete variable store $s : \mathsf{PVar} \rightarrow \mathsf{Val}$, mapping program variables to values. We use $[\![E]\!]_{\theta,s}$ to denote the evaluation of a logical expression under some interpretation θ and variable store s, resulting in a value $v \in \mathsf{LVal} \cup \{\lightning\}$. We illustrate a few interesting cases of its definition:

$$
\begin{aligned}
[\![v]\!]_{\theta,s} &= v \\
[\![\mathsf{x}]\!]_{\theta,s} &= s(\mathsf{x}) \\
[\![x]\!]_{\theta,s} &= \theta(x) \\
[\![E \in \tau]\!]_{\theta,s} &= \begin{cases} \mathsf{true} & [\![E]\!]_{\theta,s} \in \tau \\ \mathsf{false} & \text{otherwise} \end{cases}
\end{aligned}
$$

[3] Note that we differ from Lööw et al. [14] in that they present θ as a partial function. We instead choose to strengthen this to make θ total, in order to better align with the logical semantics of SMT-LIB. In particular, *valuation functions* in SMT-LIB (presented below) are total, since SMT-LIB closely resembles a many-sorted logic.

$$\llbracket f(E_1,\,\ldots,\,E_n)\rrbracket_{\theta,s} = \begin{cases} \llbracket E\rrbracket_{\theta',s} & \begin{array}{l} f(\vec{x}:\vec{\tau})\{E\} \in \mathsf{Func} \\ \llbracket\vec{E}\rrbracket_{\theta,s} = \vec{v} \\ \vec{v} \in \vec{\tau} \\ \theta' = \theta[\vec{x} \mapsto \vec{v}] \end{array} \\ \lightning & \text{otherwise} \end{cases}$$

$$\llbracket C(E_1,\,\ldots,\,E_n)\rrbracket_{\theta,s} = \begin{cases} C(v_1,\,\ldots,\,v_n) & \begin{array}{c} C(\tau_1,\ldots,\tau_n) : \tau \in \mathsf{Constructor} \\ \bigwedge_{i=1}^{n}(\llbracket E_i\rrbracket_{\theta,s} = v_i \wedge v_i \in \tau_i) \end{array} \\ \lightning & \text{otherwise} \end{cases}$$

$$\llbracket \mathsf{is}C(E)\rrbracket_{\theta,s} = \begin{cases} \mathsf{true} & \llbracket E\rrbracket_{\theta,s} = C(\vec{v}) \\ \mathsf{false} & \text{otherwise} \end{cases}$$

$$\llbracket \mathsf{get}C_i(E)\rrbracket_{\theta,s} = \begin{cases} v_i & \llbracket E\rrbracket_{\theta,s} = C(v_1,\,\ldots,\,v_n) \wedge 1 \le i \le n \\ \lightning & \text{otherwise} \end{cases}$$

$$\llbracket E\,?\,E_1 \,:\, E_2\rrbracket_{\theta,s} = \begin{cases} \llbracket E_1\rrbracket_{\theta,s} & \llbracket E\rrbracket_{\theta,s} = \mathsf{true} \\ \llbracket E_2\rrbracket_{\theta,s} & \llbracket E\rrbracket_{\theta,s} = \mathsf{false} \\ \lightning & \text{otherwise} \end{cases}$$

In particular cases, we omit the store and write $\llbracket E\rrbracket_{\theta}$ instead of $\llbracket E\rrbracket_{\theta,s}$ when the store irrelevant to the outcome of evaluating the logical expression. In the definition of $\llbracket E \in \tau\rrbracket_{\theta,s}$, we note that when $\tau = \mathsf{LVal}$, $\llbracket E\rrbracket_{\theta,s} \in \mathsf{LVal} \Leftrightarrow \llbracket E\rrbracket_{\theta,s} \neq \lightning$. We also draw attention to the evaluation of function / constructor application – in particular, the evaluation of these logical expressions results in $\lightning$ if the arguments do not respect the type restrictions imposed on the parameters.

Termination

We note that it is possible to define *non-terminating* expressions. A simple example is $f(\mathsf{true})$ where $f(x : \mathsf{Bool})\{f(x)\} \in \mathsf{Func}$. We make the simplifying assumption that the evaluation of all expressions terminates:

Property 1 (Logical Expression Evaluation Termination)**.** *For all logical expressions E:*

$$\exists v \in \mathsf{LVal}.\ \llbracket E\rrbracket_{\theta} = v$$

This is not a restrictive assumption in practice. Termination can be enforced, for example, by requiring that functions are *structurally recursive*. A structurally recursive function operates on inductively defined data types, such as lists or trees, and makes recursive calls only on direct subcomponents of its input. For functions that require more sophisticated termination arguments – such as those that recurse on non-obvious substructures or involve mutual recursion – it is well

known how to use *measures* to prove termination.

4.1.3. Assertions

The CSE engine reasons about function specifications, with function pre/post-conditions expressed in the assertion language:

$$A \in \mathsf{Asrt} ::= E \in \mathsf{LExp} \mid \mathsf{True} \mid A_1 \Rightarrow A_2 \mid A_1 \vee A_2 \mid \\ \exists x.\, A \mid \mathsf{emp} \mid A_1 \star A_2 \mid r(\vec{E_1}; \vec{E_2}) \mid p(\vec{E_1}; \vec{E_2})$$

Assertions include Boolean assertions E, several first-order connectives and quantifiers, the empty memory assertion emp, assertions built using the separating conjunction $\star$, resource assertions $r(\vec{E_1}; \vec{E_2})$, and user-defined-predicate assertions $p(\vec{E_1}; \vec{E_2})$. The parameters of resource and user-defined-predicate assertions are split into *in-parameters* and *out-parameters*. This is for automation purposes: in the CSE engine, the consume operation requires the in-parameters *to be known* before consumption and *learns* the out-parameters during consumption (see Lööw et al. [15] for further details).

An assertion can be satisfied by some concrete state, and an interpretation of the logical variables in the assertion, $\theta : \mathsf{LVar} \rightarrow \mathsf{Val}$. The interesting cases for the satisfaction relation, denoted by $\theta, \sigma \models A$, are as follows (remaining cases are given in Appendix A.1):

$$\theta, (s, \mu) \models A_1 \star A_2 \;\Leftrightarrow\; \exists \mu_1, \mu_2.\, \mu = \mu_1 \cdot \mu_2 \text{ and } \theta, (s, \mu_1) \models A_1 \text{ and } \theta, (s, \mu_2) \models A_2 \\ r(\vec{E_1}; \vec{E_2}) \Leftrightarrow \llbracket \vec{E_1} \rrbracket_{\theta,s} = \vec{v_1} \text{ and } \llbracket \vec{E_2} \rrbracket_{\theta,s} = \vec{v_2} \text{ and } \mu \models_{\mathsf{Res}} r(\vec{v_1}; \vec{v_2})$$

As is standard for abstract separation logics, the semantics of $\star$ is defined in terms of the composition operator $\cdot$ from the memory instance data (introduced in the previous section). The meaning of resource assertions $r(\vec{E_1}; \vec{E_2})$ is also defined by instance data:

Instance Definition 2. *A satisfaction relation for resource assertions* $\mu \models_{\mathsf{Res}} r(\vec{v_1}; \vec{v_2})$.

Instance Definition Example 2. *For our running example linear model, there are two types of resources: the positive cell assertion, (prettified)* $E_1 \mapsto E_2$*, and the negative cell assertion,* $E \mapsto \varnothing$*. For the positive cell assertion,* E_1 *is an in-parameter and* E_2 *is an out-parameter. For the negative cell assertion,* E *is an in-parameter. The satisfaction relation is as follows:*

$$\mu \models_{\mathsf{Res}} v_1 \mapsto v_2 \Leftrightarrow \mu = \{v_1 \mapsto v_2\} \\ v_1 \mapsto \varnothing \Leftrightarrow \mu = \{v_1 \mapsto \varnothing\}$$

We also introduce the concept of semantic entailment of assertions, writing $A_1 \models A_2$, meaning all models satisfying A_1 must satisfy A_2.

$$\forall \theta, \sigma.\, \theta, \sigma \models A_1 \implies \theta, \sigma \models A_2$$

4.1.4. Symbolic State

A symbolic state $\hat{\sigma}$ of our CSE engine consists of structures of logical expressions LExp without program variables. To separate symbolic definitions from other logical definitions, we denote

symbolic definitions using hat-notation, e.g., $\hat{\sigma}$ for symbolic state.

Symbolic Memory

The most interesting component of symbolic states is their symbolic memory component:

Instance Definition 3. *A symbolic memory model is a pair* $(\mathsf{SMem}, \hat{\mu}_\emptyset)$*, where* SMem *is a set of symbolic memories and* $\hat{\mu}_\emptyset \in \mathsf{SMem}$ *is an empty-memory element.*

Instance Definition Example 3. *For our running example linear model, we have that* SMem *is* $\mathsf{LExp} \rightharpoonup_{fin} (\mathsf{LExp} \uplus \{\varnothing\})$ *and* $\hat{\mu}_\emptyset$ *is the empty function.*

Typing Environment

We extend the symbolic state presented by Lööw et al. [14] and introduce a typing environment, $\hat{T} : \mathsf{LVar} \rightharpoonup_{fin} \mathsf{Tys}$. The typing environment maps logical variables x to their inferred type $\hat{T}(x) = \tau$. We may write $x : \tau \in \hat{T}$ to mean $\hat{T}(x) = \tau$. We write $\theta \models_{\mathsf{Type}} \hat{T}$ to mean that θ satisfies the typing environment $\hat{T}$:

$$\forall x \in \mathsf{dom}(\hat{T}).\ x : \tau \in \hat{T} \implies \theta(x) \in \tau$$

Full Symbolic State

In full, a symbolic state $\hat{\sigma}$ has the form $(\hat{s}, \hat{\mu}, \hat{\mathcal{P}}, \hat{\pi}, \hat{T})$ where: $\hat{s} : \mathsf{PVar} \rightharpoonup_{fin} \mathsf{LExp}$ is a symbolic store; $\hat{\mu}$ is the symbolic memory introduced above; $\hat{\mathcal{P}}$ is a multiset of symbolic predicates, where a symbolic predicate is an expression $p(\vec{E}_1; \vec{E}_2)$ where $p \in \mathsf{Str}$ is a predicate name and $\vec{E}_1, \vec{E}_2 \in \vec{\mathsf{LExp}}$; $\hat{\pi} \in \mathsf{LExp}$ is a path condition (that captures constraints imposed during execution); and, lastly, $\hat{T}$ is the typing environment.

We use the syntax $\hat{\sigma}.\mathsf{mem}$, $\hat{\sigma}.\mathsf{pc}$, $\hat{\sigma}.\mathsf{te}$ etc. to access components of symbolic state. We use $\mathtt{lv}(\hat{\sigma})$ to refer to the logical variables of a symbolic state $\hat{\sigma}$.

To define the symbolic state satisfaction relation $\theta, (s, \mu) \models \hat{\sigma}$ we first need to define satisfaction relations for symbolic stores $\theta, s \models_{\text{Sto}} \hat{s}$, symbolic memories $\theta, \mu \models_{\text{Mem}} \hat{\mu}$, and symbolic predicates $\theta, \mu \models_{\text{Pred}} \hat{\mathcal{P}}$. The satisfaction relations for symbolic stores and symbolic predicates are as expected (see Appendix A.2) and satisfaction relation for symbolic memories is given by instance data:

Instance Definition 4. *A symbolic memory satisfaction relation* $\theta, \mu \models_{Mem} \hat{\mu}$*. The empty symbolic memory and empty concrete memory must be related as follows:* $\theta, \mu \models_{Mem} \hat{\mu}_\emptyset \iff \mu = \mu_\emptyset$*.*

Instance Definition Example 4. *The symbolic memory satisfaction relation for our running example linear model is defined as follows, where we for succinct presentation say* $[\![\varnothing]\!]_\theta = \varnothing$*:*

$$\theta, \mu \models_{Mem} \{E_a^1 \mapsto E_e^1, \ldots, E_a^n \mapsto E_e^n\} \Leftrightarrow \mu = \uplus_{i=1}^n \{[\![E_a^i]\!]_\theta \mapsto [\![E_e^i]\!]_\theta\}$$

Now, the symbolic state satisfaction relation $\theta, s, \mu \models (\hat{s}, \hat{\mu}, \hat{\mathcal{P}}, \hat{\pi}, \hat{T})$ is defined as follows: $\exists \mu_1, \mu_2.\ \mu = \mu_1 \cdot \mu_2$ and $\theta, s \models_{\text{Sto}} \hat{s}$ and $\theta, \mu_1 \models_{\text{Mem}} \hat{\mu}$ and $\theta, \mu_2 \models_{\text{Pred}} \hat{\mathcal{P}}$ and $[\![\hat{\pi}]\!]_\theta = \mathsf{true}$ and $\theta \models_{\text{Type}} \hat{T}$.

We say that a symbolic state $\hat{\sigma}$ is satisfiable, denoted $\mathsf{SAT}(\hat{\sigma})$, when $\exists \theta, s, \mu.\ \theta, s, \mu \models \hat{\sigma}$. We also write $\hat{\sigma} \models E$ to mean that a logical expression E is true under all models of a symbolic state $\hat{\sigma}$:

$$\forall \theta, s, \mu.\ \theta, (s, \mu) \models \hat{\sigma} \implies [\![E]\!]_{\theta,s} = \mathsf{true}$$

4.1.5. Typing Environment and Symbolic Execution

We note that adding the type information $x : \tau \in \hat{T}$ to the typing environment is semantically equivalent to adding the condition $x \in \tau$ to the path condition of a symbolic state – both symbolic states are satisfied by exactly the same (θ, σ). [4] In particular, this has the effect of restricting the set of concrete states which satisfy the symbolic state. While we don't cover the specifics of how the typing environment can be maintained throughout symbolic execution, we can use this observation to note that if our symbolic state implies that $x \in \tau$, $x : \tau$ can be soundly added to the typing environment without changing the set of states which satisfy the symbolic state. This fact can then be used to extend the operational semantics of symbolic execution presented by Lööw et al. [14] to maintain a typing environment along with the pre-existing symbolic state.

In practice, this can be achieved, for example, by running a standard type inference algorithm on the path condition $\hat{\sigma}.\mathsf{pc}$ of a symbolic state. This type inference can be as maximal or minimal as desired, without impact on the formalism, with the most naive approach being no type inference at all. This of course has performance trade offs, in that the encoded SMT query will be better optimised the more type information is known.

4.1.6. Untypable Logical Expressions

We highlight a key distinction between logical expressions whose type is not knowable (e.g. when accessing a value from a list, or applying a logical function) and expressions which are *untypable*. A simple example of an untypable expression is $1 + \mathsf{true}$. We write $\hat{T} \nvdash E : \tau$ to mean that E cannot be typed as τ in the typing environment $\hat{T}$. Further, we write $\hat{T} \nvdash E$ to mean that E cannot be typed as any type. The semantics of $\hat{T} \nvdash E : \tau$ are detailed in Appendix A.3. We also write $\hat{T} \vdash E$ / $\hat{T} \vdash E : \tau$ to mean $\neg \hat{T} \nvdash E$. Formally, by "cannot be typed as", we mean:

Lemma 1. *For all $E \in \mathsf{LExp}$ and typing environments $\hat{T}$ and $\tau \in \mathsf{Tys}$:*

$$\hat{T} \nvdash E : \tau \implies \forall \theta : \mathsf{LVar} \to \mathsf{LVal}.\ \left(\theta \models_{\mathsf{Type}} \hat{T} \implies [\![E]\!]_\theta \notin \tau\right)$$

Proof. See Appendix A.3 ☐

As a consequence of Lemma 1, we have:

Lemma 2. *The evaluation of untypable expressions must result in $\lightning$. Formally, for all $E \in \mathsf{LExp}$ and typing environments $\hat{T}$:*

$$\hat{T} \nvdash E \implies \forall \theta.\ \left(\theta \models_{\mathsf{Type}} \hat{T} \implies [\![E]\!]_\theta = \lightning\right)$$

[4] This can be shown trivially, and follows from the semantics of $[\![\cdot]\!]_\theta$ – $[\![x \in \tau]\!]_\theta = \mathsf{true}$ iff $[\![x]\!]_\theta \in \tau$ iff $\theta(x) \in \tau$.

Proof. This follows as a result of Lemma 1. Take arbitrary typing environment $\hat{T}$ and $E \in \mathsf{LExp}$. Suppose then that $\hat{T} \not\vdash E$. It must then be the case that $\hat{T} \not\vdash E : \mathsf{LVal}$. Take arbitrary θ and assume that $\theta \models_{\mathsf{Type}} \hat{T}$. From Lemma 1, it follows that $[\![E]\!]_\theta \notin \mathsf{LVal}$, and so it must be that $[\![E]\!]_\theta = \lightning$. □

Untypable Function Bodies

Property 2. *We assume that function bodies are well-typed and well-formed:*

$$\forall f(x_1 : \tau, \ldots, x_n : \tau)\{E\} \in \mathsf{Func}.\ \mathtt{lv}(E) \subseteq \{x_i\}_{i=1}^n \wedge \hat{T}_f \vdash E$$

Where $x_i : \tau_i \in \hat{T}_f$ *for* $i = 1, \ldots, n$.

In practice, this can be checked statically.

Untypable Symbolic States

We extend this notion of untypability to symbolic states – a symbolic state is said to be untypable if it contains any untypable logical expression. We build up this definition of untypable symbolic states, by defining untypability individually for each component of symbolic state.

We say a symbolic store is untypable if it contains any untypable logical expressions in its range:

$$\hat{T} \not\vdash \hat{s} \stackrel{\text{def}}{=} \exists E \in \mathsf{rng}(\hat{s}).\ \hat{T} \not\vdash E$$

Lemma 3. *Untypable symbolic stores are not satisfiable:*

$$\theta \models_{\mathsf{Type}} \hat{T} \wedge \hat{T} \not\vdash \hat{s} \implies \neg\exists s.\ \theta, s \models_{\mathsf{Sto}} \hat{s}$$

Proof. Suppose for an arbitrary θ, $\hat{T}$ and $\hat{s}$ that $\theta \models_{\mathsf{Type}} \hat{T}$ and $\hat{T} \not\vdash \hat{s}$. We know then that $\hat{T} \not\vdash E$ for some $E \in \mathsf{rng}(\hat{s})$. It follows from Lemma 2 that $[\![E]\!]_\theta = \lightning$, and so $[\![E]\!]_\theta \notin \mathsf{Val}$. Suppose for a contradiction that $\theta, s \models_{\mathsf{Sto}} \hat{s}$ for some concrete store s. But then $[\![E]\!]_\theta \in \mathsf{rng}(s)$ and so $[\![E]\!]_\theta \in \mathsf{Val}$ – we have a contradiction. □

Similar to symbolic stores, we wish to define untypability for symbolic memories. However, since symbolic memories are defined by each instantiation, the definition of untypability for symbolic states must also be defined by the instantiation:

Instance Definition 5. *We require a definition for the untypability relation over symbolic memories,* $\hat{T} \not\vdash \hat{\mu}$.

Instance Property 1. *We require untypable symbolic memories to be unsatisfiable:*

$$\theta \models_{\mathsf{Type}} \hat{T} \wedge \hat{T} \not\vdash \hat{\mu} \implies \neg\exists \mu.\ \theta, \mu \models_{\mathsf{Mem}} \hat{\mu}$$

Instance Definition Example 5. *In our running example, symbolic memories* $\hat{\mu}$ *are partial finite mappings* $\mathsf{LExp} \rightharpoonup_{\mathit{fin}} \mathsf{LExp}$. *Thus, our untyability relation can be defined as follows:*

$$\hat{T} \not\vdash \hat{\mu} \stackrel{\text{def}}{=} \left(\exists E \in \mathsf{dom}(\hat{\mu}).\ \hat{T} \not\vdash E : \mathsf{Int}\right) \vee \left(\exists E \in \mathsf{rng}(\hat{\mu}).\ \hat{T} \not\vdash E\right)$$

It can be shown, by considering Lemma 2 and Lemma 1 that in our running example, the untypability relation over symbolic stores in Instance Definition Example 5 satisfies Instance Property 1:

Proof. Suppose that $\theta \models_{\mathsf{Type}} \hat{T}$ and $\hat{T} \not\vdash \hat{\mu}$. Suppose for a contradiction that $\theta, \mu \models_{\mathsf{Mem}} \hat{\mu}$ for some μ. We have then that since $\hat{T} \not\vdash \hat{\mu}$, either $\hat{T} \not\vdash E : \mathsf{Int}$ for some $E \in \mathsf{dom}(\hat{\mu})$, or $\hat{T} \not\vdash E$ for some $E \in \mathsf{rng}(\hat{\mu})$. In the case of the former, we have from Lemma 1 that $[\![E]\!]_\theta \notin \mathsf{Int}$, but since $[\![E]\!]_\theta \in \mathrm{dom}(\mu)$, it must be that $[\![E]\!]_\theta \in \mathsf{Int}$ – a contradiction. In the case of the latter, we have from Lemma 2 that $[\![E]\!]_\theta = \lightning$, but since $[\![E]\!]_\theta \in \mathrm{dom}(\mu)$, it must be that $[\![E]\!]_\theta \in \mathsf{Val} \cup \{\varnothing\}$. In both cases, we reach a contradiction. □

We can use these definitions to define when symbolic states are untypable. We write $\not\vdash \hat{\sigma}$ when this is the case, meaning:

$$\hat{T} \not\vdash \hat{\pi} : \mathsf{Bool} \;\vee\; \hat{T} \not\vdash \hat{s} \;\vee\; \hat{T} \not\vdash \hat{\mu}$$

where $\hat{\sigma} = (\hat{s}, \hat{\mu}, \hat{\mathcal{P}}, \hat{\pi}, \hat{T})$.

Lemma 4. *Untypable symbolic states are not satisfiable. Formally, for all symbolic states $\hat{\sigma}$:*

$$\not\vdash \hat{\sigma} \implies \neg\mathsf{SAT}(\hat{\sigma})$$

Proof. This follows as a result of Lemma 2, Lemma 3, Instance Property 1, and the definition of symbolic state satisfiability. □

4.1.7. Consume and Produce Operations

As discussed in Section 2.5, most modern CSE tools implement a consume / produce architecture. In short, the consume operation takes as input an assertion and removes (“consumes”) the corresponding symbolic state from the engine’s current symbolic state and the produce operation takes as input an assertion and adds (“produces”) the corresponding symbolic state to the current symbolic state. Lööw et al. [13, 14] present an axiomatic interface which describes the properties of the consume and produce operations. Of particular relevance to our work is the produce operation, and its properties.

First, we introduce symbolic substitutions, $\hat{\theta} : \mathsf{LVar} \rightharpoonup_{\mathit{fin}} \mathsf{LExp}$, which the produce operation uses to instantiate free logical variables. Now, the judgement of produce is:

$$\mathsf{produce}(A, \hat{\theta}, \hat{\sigma}) \rightsquigarrow \hat{\sigma}'$$

The judgement for produce states that the production of A with symbolic substitution $\hat{\theta}$ in state $\hat{\sigma}$ results in state $\hat{\sigma}'$ where the symbolic state corresponding to A has been added. Figure 4.1, illustrates an example of the produce operation being applied to a symbolic state $\hat{\sigma}$, and an assertion describing a singly linked list, A. This operation results in two symbolic states, $\hat{\sigma}_1$ and $\hat{\sigma}_2$. In the first case, $\hat{\sigma}_1$ is a symbolic state where the symbolic state of the empty list has been added to $\hat{\sigma}$. On the other hand, $\hat{\sigma}_2$ is a symbolic state where the symbolic state of a non-empty list has been added to $\hat{\sigma}$. We omit the specifics of the produce operation and its definition. It suffices for us to know that the following OX/UX properties hold for produce:

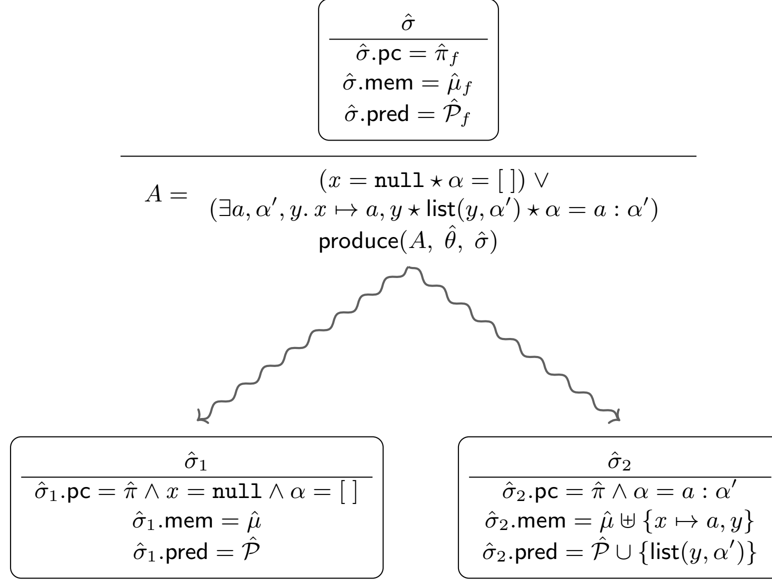

Figure 4.1. Produce Operation on a List Assertion

Property 3 (OX Soundness: produce)**.** *Let* $\texttt{lv}(A) \subseteq \mathrm{dom}(\hat{\theta})$*. If:*

$$\theta, s, \mu_A \models \hat{\theta}'(A)$$
$$\theta, s, \mu \models \hat{\sigma}$$

and $(\mu_A \cdot \mu_f)$ *is defined, then:*

$$\exists \theta', \hat{\sigma}'.\ \theta'|_{\texttt{lv}(\hat{\sigma})} = \theta \ \wedge\ \mathsf{produce}(P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow \hat{\sigma}' \ \wedge\ \theta', s, (\mu_A \cdot \mu_f) \models \hat{\sigma}'$$

Property 4 (UX Soundness: produce)**.** *If:*

$$\mathsf{produce}(A, \hat{\theta}, \hat{\sigma}) \rightsquigarrow \hat{\sigma}'$$
$$\theta, s, \mu \models \hat{\sigma}'$$

then there exists μ_A*,* μ_f *such that:*

$$\mu = \mu_A \cdot \mu_f$$
$$\theta, s, \mu_A \models \hat{\theta}(A)$$
$$\theta, s, \mu_f \models \hat{\sigma}$$

4.2. Formalising SMT-LIB

This section formalises a core subset of SMT-LIB to support our encoding. The presentation broadly follows the formal semantics provided in the SMT-LIB standard [3], with certain omissions and refinements for relevance and clarity.

4.2.1. Term Language

The syntax of SMT-LIB terms is defined by the following grammar:

$$t \in \mathsf{Term} ::= x \in \mathsf{LVar} \mid f\ t^* \mid \exists (x : \sigma)^+\ t \mid \forall (x : \sigma)^+\ t$$

Here, each term is associated with a *sort* (i.e., a type), reflecting the many-sorted nature of SMT-LIB. A term of sort σ_{Bool} is referred to as a formula.

4.2.2. Signatures

A *signature* defines the syntactic and type structure of terms. Formally, a signature Σ consists of:

- A set Σ^S of sort symbols, including σ_{Bool}. The set of all constructible sorts is denoted $\mathsf{Sort}(\Sigma)$.
- A set Σ^F of function symbols.
- Subsets of Σ^F designating constructors (Σ^C), selectors (Σ^G), and testers (Σ^T), with appropriate disjointness constraints.
- A total mapping $\mathsf{con}_\Sigma : \Sigma^S \to 2^{\Sigma^C}$ assigning constructors to sort symbols.
- A total mapping $\mathsf{sel}_\Sigma : \Sigma^C \to (\Sigma^G)^*$ assigning selector sequences to constructors, such that each selector is uniquely assigned.
- A bijective total mapping $\mathsf{tes}_\Sigma : \Sigma^C \to \Sigma^T$ pairing each constructor with a corresponding tester.
- A variable-sorting map $\mathsf{LVar} \to \mathsf{Sort}(\Sigma)$.
- A function ranking map $R : \Sigma^F \to (\mathsf{Sort}(\Sigma))^+$ providing type signatures for function symbols, satisfying:
  1. Each constructor $C \in \Sigma^C$ has a rank $\sigma_1 \dots \sigma_n \sigma$, where $C \in \mathsf{con}_\Sigma(\sigma)$.
  2. If $R(C) = \sigma_1 \dots \sigma_n \sigma$ and $\mathsf{sel}_\Sigma(C) = \mathsf{get}C_1 \dots \mathsf{get}C_n$, then for all i, $R(\mathsf{get}C_i) = \sigma\sigma_i$.
  3. If $R(C) = \vec{\sigma}\sigma$, and $\mathsf{is}C = \mathsf{tes}_\Sigma(C)$, then $R(\mathsf{is}C) = \sigma\sigma_{\mathsf{Bool}}$.

A sort is treated as an algebraic datatype if it has at least one constructor. We write $x : \sigma \in \Sigma$ if variable x is assigned sort σ in Σ. The well-sortedness judgment $\Sigma \vdash t : \sigma$ asserts that t has sort σ under signature Σ. This extends naturally to term sets T as:

$$\forall t \in T.\ \Sigma \vdash t : \sigma$$

A signature Ω is said to *extend* another signature Σ if:

- $\Sigma^S \subseteq \Omega^S$ and $\Sigma^F \subseteq \Omega^F$
- Sorts and function symbols shared between them preserve their assignments
- Variable typings agree: $x : \sigma \in \Sigma$ iff $x : \sigma \in \Omega$

A *variant* Σ' of Σ differs only in how it maps variables to sorts.

4.2.3. Structures

Given a signature Σ, a *structure* $\mathbf{A}$ provides its semantic interpretation. It consists of:

- A universe A partitioned into non-empty domains $\sigma^{\mathbf{A}} \subseteq A$ for each sort $\sigma \in \mathsf{Sort}(\Sigma)$, such that $A = \bigcup_\sigma \sigma^{\mathbf{A}}$.
- Each function $f : \sigma_1 \dots \sigma_n \sigma \in \Sigma$ is interpreted as a total function $f^{\mathbf{A}}$ of matching arity.
- $\sigma^{\mathbf{A}}_{\mathsf{Bool}} = \mathsf{Bool} = \{\mathsf{true}, \mathsf{false}\}$.
- Selectors act as projections: for any constructor C and arguments $(v_1, \dots, v_n)$,

$$\mathsf{get}C_i^{\mathbf{A}}\left(C^{\mathbf{A}}(v_1, \dots, v_n)\right) = v_i$$

- Testers check constructor origin: $\mathsf{is}C^{\mathbf{A}}(v) = \mathsf{true}$ iff v lies in the image of $C^{\mathbf{A}}$.
- All constructors are interpreted as *acyclic* functions[5].

If $\mathbf{B}$ is a structure over an expanded signature Ω, its *reduct* to Σ is the unique structure $\mathbf{A}$ interpreting the shared sorts and function symbols identically. In this case, $\mathbf{B}$ is an *extension* of $\mathbf{A}$.

4.2.4. Valuations

A *valuation* over $\mathbf{A}$ is a total function $\theta : \mathsf{LVar} \to A$ such that each $x : \sigma \in \Sigma$ maps to $\theta(x) \in \sigma^{\mathbf{A}}$.

Given a term t, its interpretation $[\![t]\!]_{\mathbf{A},\theta}$ assigns it a value in $\mathbf{A}$ under valuation θ, assuming the term is well-sorted. We record the preservation of sorting in the following lemma:

Lemma 5. *If $\Sigma \vdash t : \sigma$, then for every structure $\mathbf{A}$ and valuation θ into $\mathbf{A}$,*

$$[\![t]\!]_{\theta,\mathbf{A}} \in \sigma^{\mathbf{A}}$$

4.2.5. Theories

An *SMT-LIB theory* $\mathcal{T}$ consists of a signature Σ and a class of structures $\mathcal{M}(\mathcal{T})$ that satisfy certain axioms over that signature.

4.2.6. Satisfiability

Given a set of formulae Φ, we say that $\mathbf{A}, \theta \models_{\mathtt{SMT}} \Phi$ if:

$$\forall \varphi \in \Phi, \quad [\![\varphi]\!]_{\mathbf{A},\theta} = \mathsf{true}$$

A structure $\mathbf{A}$ satisfies Φ iff every valuation into it does:

$$\mathbf{A} \models_{\mathtt{SMT}} \Phi \quad \text{iff} \quad \forall \theta,\ \mathbf{A}, \theta \models_{\mathtt{SMT}} \Phi$$

[5] Acyclicity is an additional assumption beyond the SMT-LIB standard.

4.3. Formalising the Encoding

We have now covered the necessary background to present the encoding of symbolic state into SMT-LIB. Prior formalisations of CSE [13, 10, 28, 4] typically abstract over the mechanism for querying satisfiability, and do not address the encoding of symbolic state into SMT queries.

We work modulo some background theories $\mathcal{T}$, including the theory of seqs [6] and non-linear arithmetic. These theories have a combined signature $\Sigma_{\mathcal{T}}$ and a set of structures $\mathcal{M}(\mathcal{T})$ that adhere to our the axioms of $\mathcal{T}$.

4.3.1. Background Signature

We begin by extending $\Sigma_{\mathcal{T}}$ with SMT-LIB ADTs and functions relevant to the encoding. One key concept introduced here is the σ_{LVal} ADT. This is an ADT, which contains constructors for values of all possible concrete types. Consider for example the encoding of a list into SMT-LIB. The type of the list's elements is not knowable, due to the dynamic nature of typing in our expression language. To circumvent this, we instead encode the list as $(\sigma_{\mathsf{Seq}}\sigma_{\mathsf{LVal}})$, i.e. a sequence of σ_{LVal} ADTs. Thus, the list can contain an integer wrapped in the integer constructor, and simultaneously a boolean wrapped in the boolean constructor. Consider the signature resulting from extending $\Sigma_{\mathcal{T}}$ with:

- an ADT to represent a value, whose type is not known and so cannot be *sorted* in the encoding, including,
  - the sort σ_{LVal}
  - the constructor bool with rank $\sigma_{\mathsf{Bool}}\sigma_{\mathsf{LVal}}$, along with corresponding tester isBool and selector $\mathsf{getBool}$
  - the constructor int with rank $\sigma_{\mathsf{Int}}\sigma_{\mathsf{LVal}}$, along with corresponding tester isInt and selector getInt
  - the constructor seq with rank $(\sigma_{\mathsf{Seq}}\sigma_{\mathsf{LVal}})\sigma_{\mathsf{LVal}}$ [7], along with corresponding tester isSeq and selector getSeq
  - constructors $\mathsf{datatype}_\tau$ with rank $\sigma_\tau\sigma_{\mathsf{LVal}}$ for each $\tau \in \mathsf{Datatype}$, along with corresponding tester $\mathsf{isDatatype}_\tau$ and selectors $\mathsf{getDatatype}_\tau$;
  - a special nullary constructor $\lightning$ with rank σ_{LVal}, and the corresponding tester $\mathsf{is}\lightning$.
- ADTs for user-defined datatypes, including a sort σ_τ for each $\tau \in \mathsf{Datatype}$, and for each $C(\tau_1, \ldots, \tau_n) : \tau \in \mathsf{Constructor}$,
  - a corresponding constructor function symbol C, with rank $\sigma_1 \ldots \sigma_n \sigma_\tau$
  - a tester function $\mathsf{is}C$, with rank $\sigma_\tau\sigma_{\mathsf{Bool}}$
  - selector functions $\mathsf{get}C_i$, with rank $\sigma_\tau\sigma_i$

  where $\sigma_i = \mathsf{encTys}(\tau_i)$ for $i = 1 \ldots n$ (encTys is introduced below);

[6] The theory of seqs is a Z3 [5] specific theory, which generalises the standard theory of strings.

[7] The sort $(\sigma_{\mathsf{Seq}}\sigma_{\mathsf{LVal}})$ represents a sequences of values. Our formalisation omits details about polymorphism in SMT-LIB, but specifics can be found in the SMT-LIB standard [3].

- for each $f(x_1 : \tau_1, \ldots, x_n : \tau_n)\{E\} \in \mathsf{Func}$, a corresponding function symbol f with rank $\sigma_1 \ldots \sigma_n \sigma_{\mathsf{LVal}}$, where $\sigma_i = \mathsf{encTys}(\tau_i)$ for $i = 1 \ldots n$.

We label this expansion of $\Sigma_{\mathcal{T}}$ as Σ, and refer to it as the *background signature*. Further, we say that a Σ-structure $\mathbf{A} \in \mathcal{M}(\mathcal{T}^{\Sigma})$ when the reduct of $\mathbf{A}$ to $\Sigma_{\mathcal{T}}$ is in $\mathcal{M}(\mathcal{T})$.

Lemma 6. *The signature Σ is a well-formed expansion of $\Sigma_{\mathcal{T}}$.*

Proof. See Appendix C.1. □

4.3.2. Encoding Logical Expressions

In order to define our encoding, we first define two functions – $\mathsf{encLExp}_{\hat{T}} : \mathsf{LExp} \rightharpoonup \mathsf{Term} \times \mathsf{Sort}(\Sigma) \times 2^{\mathsf{Term}}$, and $\mathsf{encTys} : \mathsf{Tys} \to \mathsf{Sort}(\Sigma)$. encTys maps types in the expression language into sorts in the background signature. Whereas $\mathsf{encLExp}_{\hat{T}}(E) = (t,\ \sigma,\ \Phi)$ gives a tuple containing: the encoding of E as an SMT-LIB term t; the sort σ of the encoded term t; and, a set of assertions (i.e. SMT-LIB formulae) Φ which are satisfiable iff the evaluation of E is well-defined. The full definitions of encTys and $\mathsf{encLExp}_{\hat{T}}$ are given in Appendix C.2 and Appendix C.3 respectively, but we give below a few interesting cases of the definition of $\mathsf{encLExp}_{\hat{T}}$:

$$\mathsf{encLExp}_{\hat{T}}(x) = \begin{cases} (x,\ \ \mathsf{encTys}(\tau),\ \ \emptyset) & \text{if } x : \tau \in \hat{T} \\ (x,\ \ \sigma_{\mathsf{LVal}},\ \ \emptyset) & \text{otherwise} \end{cases} \qquad \text{iff } x \in \mathsf{LVar}$$

$$\mathsf{encLExp}_{\hat{T}}(E_1 / E_2) = (/\ t_1\ t_2,\ \ \sigma_{\mathsf{Int}},\ \ \Phi) \qquad \text{iff } \begin{cases} (t_1, \sigma_1, \Phi_1) = \mathsf{toInt}(\mathsf{encLExp}_{\hat{T}}(E_1)) \\ (t_2, \sigma_2, \Phi_2) = \mathsf{toInt}(\mathsf{encLExp}_{\hat{T}}(E_2)) \\ \Phi = \Phi_1 \cup \Phi_2 \cup \{(\neq\ \ t_2\ 0)\} \end{cases}$$

$$\mathsf{encLExp}_{\hat{T}}(C(E_1, \ldots, E_n)) = (C\ t_1\ \ldots\ t_n,\ \ \sigma_\tau,\ \ \Phi) \qquad \text{iff } \begin{cases} (t_1, \sigma_1, \Phi_1) = \mathsf{toSort}_{\tau_1}(\mathsf{encLExp}_{\hat{T}}(E_1)) \\ \vdots \\ (t_n, \sigma_n, \Phi_n) = \mathsf{toSort}_{\tau_n}(\mathsf{encLExp}_{\hat{T}}(E_n)) \\ C(\tau_1, \ldots, \tau_n) : \tau \in \mathsf{Constructor} \\ \Phi = \bigcup_{i=1}^{n} \Phi_i \end{cases}$$

$$\mathsf{encLExp}_{\hat{T}}(\mathsf{get}C_i(E)) = (\mathsf{get}C_i\ t,\ \ \sigma,\ \ \Phi \cup \{\mathsf{is}C\ t\}) \qquad \text{iff } \begin{cases} 1 \le i \le n \\ C(\tau_1, \ldots, \tau_n) : \tau \in \mathsf{Constructor} \\ (t, \sigma, \Phi) = \mathsf{toDatatype}_\tau(\mathsf{encLExp}_{\hat{T}}(E)) \\ \sigma = \mathsf{encTys}(\tau_i) \end{cases}$$

$$\mathsf{encLExp}_{\hat{T}}(\mathsf{is}C(E)) = (\mathsf{is}C\ t,\ \ \sigma_{\mathsf{Bool}},\ \ \Phi) \qquad \text{iff } \begin{cases} C(\tau_1, \ldots, \tau_n) : \tau \in \mathsf{Constructor} \\ (t', \sigma', \Phi') = \mathsf{encLExp}_{\hat{T}}(E) \\ \sigma' \in \{\sigma_\tau, \sigma_{\mathsf{LVal}}\} \\ (t, \sigma, \Phi) = \mathsf{toDatatype}_\tau(t', \sigma', \Phi') \end{cases}$$

$$\mathsf{encLExp}_{\hat{T}}(\mathsf{is}C(E)) = (\mathsf{false},\ \ \sigma_{\mathsf{Bool}},\ \ \emptyset) \qquad \text{iff } \begin{cases} C(\tau_1, \ldots, \tau_n) : \tau \in \mathsf{Constructor} \\ (t, \sigma, \Phi) = \mathsf{encLExp}_{\hat{T}}(E) \\ \sigma \notin \{\sigma_\tau, \sigma_{\mathsf{LVal}}\} \end{cases}$$

$$\mathsf{encLExp}_{\hat{T}}(f(E_1, \ldots, E_n)) = (t, \ \sigma_{\mathsf{LVal}}, \ \Phi) \quad \text{iff} \begin{cases} (t_1, \sigma_1, \Phi_1) = \mathsf{toSort}_{\tau_1}(\mathsf{encLExp}_{\hat{T}}(E_1)) \\ \vdots \\ (t_n, \sigma_n, \Phi_n) = \mathsf{toSort}_{\tau_n}(\mathsf{encLExp}_{\hat{T}}(E_n)) \\ f(x_1 : \tau_1, \ldots, x_n : \tau_n)\{E\} \in \mathsf{Func} \\ t = f \ t_1 \ \ldots \ t_n \\ \Phi = \left(\bigcup_{i=1}^{n} \Phi_i\right) \cup \{(\neq \ t \ \lightning)\} \end{cases}$$

We first note the use of the typing environment to encode (where known) variables in their native sort, resorting to σ_{LVal} otherwise. In the encoding of integer division, we see an example of how partiality is handled in the encoding. In addition to the assertions generated by the encoding of each sub-expression, the assertion $(\neq \ t_2 \ 0)$ is generated. This guarantees that the encoding is well-defined under the restrictions of the generated assertions. Another example can be seen in the case of user-defined function application – the assertion $(\neq \ (f \ t_1 \ \ldots \ t_n) \ \lightning)$ is generated. In addition, the encoding function carries around the sort of the encoded term. It uses auxiliary functions to convert between sorts. For example, toInt is defined as:

$$\mathsf{toInt}(t, \sigma, \Phi) = \begin{cases} (t, \ \sigma_{\mathsf{Int}}, \ \Phi) & \text{iff } \sigma = \sigma_{\mathsf{Int}} \\ (\mathsf{getInt} \ t, \ \sigma_{\mathsf{Int}}, \ \Phi \cup \{(\mathsf{isInt} \ t)\}) & \text{iff } \sigma = \sigma_{\mathsf{LVal}} \end{cases}$$

unwrapping terms from the σ_{LVal} ADT where necessary. We draw attention to the assertion generated in the second case: $(\mathsf{isInt} \ t)$. This is necessary since selector functions, like all functions in SMT-LIB, are total.

Constructors of user-defined datatypes are encoded with their corresponding SMT-LIB constructors, and user-defined functions are encoded using their corresponding function symbols. Both of these are declared in our background signature, Σ.

We note that $\mathsf{encLExp}_{\hat{T}}(E)$ is not well-defined for some logical expressions E. This becomes clear when we examine toInt, for example. There is no sensible way to convert a logical expression encoded as σ_{Bool} to σ_{Int}. However, we can guarantee that $\mathsf{encLExp}_{\hat{T}}(E)$ is well-defined if E is typable in $\hat{T}$:

Lemma 7. *For all typing environments $\hat{T}$, $E \in \mathsf{LExp}$:*

$$\hat{T} \vdash E \implies \exists t, \sigma, \Phi. \ \mathsf{encLExp}_{\hat{T}}(E) = (t, \sigma, \Phi)$$

Proof. See Appendix C.4. □

4.3.3. Background Assertions

With this logical expression encoding defined, we can proceed with the presentation of our encoding. To give meaning to user-defined function symbols declared in the background signature, we generate a set of *background assertions* $\Psi \in 2^{\mathsf{Term}}$. Each formula $\psi \in \Psi$ has the form

$$\forall x_1 : \sigma_1 \ \ldots \ x_n : \sigma_n. \, f \ x_1 \ \ldots \ x_n = t$$

for some term t.

Let $\hat{T}_f$ be the typing environment for each $f(x_1 : \tau_1, \ldots, x_n : \tau_n)\{E\} \in \mathsf{Func}$, such that $x_i : \tau_i \in \hat{T}_f$ for $i = 1, \ldots, n$. We can use $\mathsf{encLExp}_{\hat{T}_f}$ to encode the body of a user-defined function into SMT-LIB, and so generate the background assertions Ψ. Because of our assumption that function bodies are well-typed (and Lemma 7), we know that $\mathsf{encLExp}_{\hat{T}_f}$ is well-defined for function bodies. This allows us to define the encoding of user-defined functions:

$$\Psi = \left\{ \forall \vec{x} : \vec{\sigma}.\ f\ \vec{x} = \mathsf{ite}\ (\bigwedge \Phi)\ \ t\ \ \lightning \;\middle|\; \begin{array}{c} f(\vec{x} : \vec{\tau})\{E\} \in \mathsf{Func} \\ \vec{\sigma} = \mathsf{encTys}(\vec{\tau}) \\ (t, \sigma, \Phi) = \mathsf{toVal}(\mathsf{encLExp}_{\hat{T}_f}(E)) \end{array} \right\}$$

For each $f(\vec{x} : \vec{\tau})\{E\} \in \mathsf{Func}$, we encode the body of function $\mathsf{encLExp}_{\hat{T}_f}(E)$ under the typing environment $\hat{T}_f$. We then wrap this term in the σ_{LVal} ADT. This results in a term t and a set of assertions Φ under which this encoded term is well-defined. We assert that for all inputs $\vec{x}$ to the function, the result of applying this function is equal to t if the conjunction of Φ holds, and $\lightning$ otherwise.

4.3.4. Encoding Path Conditions and Symbolic Stores

We can also use $\mathsf{encLExp}_{\hat{T}}$ to encode path conditions $\hat{\pi}$ and symbolic stores $\hat{s}$:

$$\mathsf{encPC}_{\hat{T}}(\hat{\pi}) = \{\varphi\} \cup \Phi \quad \text{where } (\varphi, \sigma, \Phi) = \mathsf{toBool}(\mathsf{encLExp}_{\hat{T}}(\hat{\pi}))$$

$$\mathsf{encSto}_{\hat{T}}(\hat{s}) = \bigcup \left\{ \Phi' \;\middle|\; \begin{array}{c} E \in \mathsf{rng}(\hat{s}) \\ (\varphi', \sigma', \Phi') = \mathsf{encLExp}_{\hat{T}}(E) \end{array} \right\}$$

When encoding symbolic stores, we simply assert that all logical expressions in the store are well-defined (i.e. don't evaluate to $\lightning$).

Lemma 8. *The encoding* $\mathsf{encPC}_{\hat{T}}(\hat{\pi})$ *is well-defined when* $\hat{T} \vdash \hat{\pi} : \mathsf{Bool}$*:*

$$\hat{T} \vdash \hat{\pi} : \mathsf{Bool} \implies \exists \Phi.\ \mathsf{encPC}_{\hat{T}}(\hat{\pi}) = \Phi$$

Proof. Suppose that $\hat{T} \vdash \hat{\pi} : \mathsf{Bool}$. Then, as a result of Lemma 7, we know that $\mathsf{encLExp}_{\hat{T}}(\hat{\pi})$ is well defined. In particular, we know as a result of Lemma 17 (see Appendix C.4) that $\mathsf{encLExp}_{\hat{T}}(\hat{\pi}) = (\varphi, \sigma, \Phi)$ where $\sigma = \sigma_{\mathsf{Bool}}$ or $\sigma = \sigma_{\mathsf{LVal}}$. In both cases, $\mathsf{toBool}(\varphi, \sigma, \Phi)$ is well-defined, and so we have that $\mathsf{toBool}(\mathsf{encLExp}_{\hat{T}}(\hat{\pi}))$ is well-defined. □

Lemma 9. *The encoding* $\mathsf{encSto}_{\hat{T}}(\hat{s})$ *is well-defined when* $\hat{T} \vdash \hat{s}$*:*

$$\hat{T} \vdash \hat{s} \implies \exists \Phi.\ \mathsf{encSto}_{\hat{T}}(\hat{s}) = \Phi$$

Proof. Suppose $\hat{T} \vdash \hat{s}$. Then, $\forall E \in \mathsf{dom}(\hat{s}).\ \hat{T} \vdash E$. It follows from Lemma 7 that $\mathsf{encLExp}_{\hat{T}}(E)$ is well-defined for all $E \in \mathsf{rng}(\hat{s})$. As such $\mathsf{encSto}_{\hat{T}}(\hat{s})$ is well-defined. □

4.3.5. Encoding Symbolic Memories

In a similar manner to the above encodings of symbolic stores and path conditions, we wish to encode the satisfiability of symbolic memories $\hat{\mu}$. Since symbolic memories are instance specific, their encoding must also be so:

Instance Definition 6. *We require a partial function* $\mathsf{encSMem}_{\hat{T}} : \mathsf{SMem} \rightharpoonup 2^{\mathsf{Term}}$. *The encoding* $\mathsf{encSMem}_{\hat{T}}(\hat{\mu}) = \Phi$ *encodes a symbolic memory* $\hat{\mu}$ *under some typing environment* $\hat{T}$, *returning a set of assertions* Φ *which determine the satisfiability of the symbolic memory.*

Instance Definition Example 6. *In our running example, recalling that symbolic memories are mappings* $\hat{\mu} : \mathsf{LExp} \rightharpoonup_{fin} \mathsf{LExp}$, *we define* $\mathsf{encSMem}_{\hat{T}}$ *as:*

$$\mathsf{encSMem}_{\hat{T}}(\hat{\mu}) = \begin{array}{l} \left(\bigcup\{\Phi' \mid (t', \sigma', \Phi') = \mathsf{toInt}(\mathsf{encLExp}_{\hat{T}}(E)),\ E \in \mathsf{dom}(\hat{\mu})\}\right) \cup \\ \left(\bigcup\{\Phi' \mid (t', \sigma', \Phi') = \mathsf{encLExp}_{\hat{T}}(E),\ E \in \mathsf{rng}(\hat{\mu})\}\right) \cup \\ \left\{ (\neq\ t_1\ t_2) \,\middle|\, \begin{array}{c} (t_1, \sigma_1, \Phi_1) = \mathsf{toInt}(\mathsf{encLExp}_{\hat{T}}(E_1)), \\ (t_2, \sigma_2, \Phi_2) = \mathsf{toInt}(\mathsf{encLExp}_{\hat{T}}(E_2)), \\ E_1 \in \mathsf{dom}(\hat{\mu}),\ \ E_2 \in \mathsf{dom}(\hat{\mu}),\ \ E_1 \neq E_2 \end{array} \right\} \end{array}$$

Instance Property 2. *We require that* $\mathsf{encSMem}_{\hat{T}}(\hat{\mu})$ *is well-defined when* $\hat{T} \vdash \hat{\mu}$:

$$\hat{T} \vdash \hat{\mu} \implies \exists \Phi, T.\ \mathsf{encSMem}_{\hat{T}}(\hat{\mu}) = (\Phi, T)$$

4.3.6. Encoding OX Satisfiability

We have now established the necessary definitions to enable us to define our over-approximate satisfiability encoding. We define a function:

$$\mathsf{encSAT}^{\mathsf{OX}} : \mathsf{SState} \to 2^{\mathsf{Term}}$$

which gives a set of assertions Φ. Informally, $\mathsf{encSAT}^{\mathsf{OX}}(\hat{\sigma})$ gives a set of assertions that are satisfiable only if the symbolic state $\hat{\sigma}$ is satisfiable. It is defined as follows:

$$\mathsf{encSAT}^{\mathsf{OX}}(\hat{\sigma}) = \begin{cases} \{\mathsf{false}\} & \nvdash \hat{\sigma} \\ \mathsf{encPC}_{\hat{T}}(\hat{\pi}) \cup \mathsf{encSto}_{\hat{T}}(\hat{s}) \cup \mathsf{encSMem}_{\hat{T}}(\hat{\mu}) & \text{otherwise} \end{cases}$$

where $\hat{\sigma} = (\hat{\pi}, \hat{s}, \hat{\mu}, \hat{\mathcal{P}})$.

We know that the encoding is well-defined because our encoding handles cases of untypable expressions in the symbolic state:

Theorem 1. *The OX SAT encoding is well-defined. Formally, for all* $\hat{\sigma}$:

$$\exists \Phi \in 2^{\mathsf{Term}}.\ \mathsf{encSAT}^{\mathsf{OX}}(\hat{\sigma}) = \Phi$$

Proof. We have that either $\vdash \hat{\sigma}$ or $\nvdash \hat{\sigma}$. In the case of the latter, $\mathsf{encSAT}^{\mathsf{OX}}(\hat{\sigma}) = \{\mathsf{false}\}$ is clearly well-defined.

Suppose that $\vdash \hat{\sigma}$ and $\hat{\sigma} = (\hat{\pi}, \hat{s}, \hat{\mu}, \hat{\mathcal{P}}, \hat{T})$. We then have that:

$$\hat{T} \vdash \hat{\pi} : \mathsf{Bool} \ \wedge \ \hat{T} \vdash \hat{s} \ \wedge \ \hat{T} \vdash \hat{\mu}$$

Then, from Lemma 8, Lemma 9 and Instance Property 2 we know that $\mathsf{encSAT}^{\mathsf{OX}}$ is well-defined. □

Handling Symbolic Predicates in OX Encoding

We note that our OX encoding function does not take symbolic predicates into account. This effectively assumes that, for any symbolic state, there exists some concrete memory satisfying its symbolic predicates. While this assumption may not hold for every symbolic state, it is acceptable in the OX setting – returning SAT for an unsatisfiable symbolic state is permissible in this context. More generally, we remark that encoding each aspect of symbolic state is not strictly necessary. The most naive OX encoding would be one which assumes that *all* symbolic states are satisfiable.

4.3.7. Encoding UX Satisfiability

We define similarly

$$\mathsf{encSAT}^{\mathsf{UX}} : \mathsf{SState} \to 2^{\mathsf{Term}}$$

which is the under-approximate equivalent of $\mathsf{encSAT}^{\mathsf{OX}}$. Its definition is identical to $\mathsf{encSAT}^{\mathsf{OX}}$, with the only difference being its handling of the symbolic predicates:

$$\mathsf{encSAT}^{\mathsf{UX}}(\hat{\sigma}) = \begin{cases} \{\mathsf{false}\} & \nvdash \hat{\sigma} \\ \{\mathsf{false}\} & \hat{\mathcal{P}} \neq \emptyset \\ \mathsf{encPC}_{\hat{T}}(\hat{\pi}) \cup \mathsf{encSto}_{\hat{T}}(\hat{s}) \cup \mathsf{encSMem}_{\hat{T}}(\hat{\mu}) & \text{otherwise} \end{cases}$$

where $\hat{\sigma} = (\hat{\pi}, \hat{s}, \hat{\mu}, \hat{\mathcal{P}})$.

Theorem 2. *The UX SAT encoding is well-defined. For all $\hat{\sigma}$:*

$$\exists \Phi \in 2^{\mathsf{Term}}.\ \mathsf{encSAT}^{\mathsf{UX}}(\hat{\sigma}) = \Phi$$

Proof. This proof is almost identical to that of Theorem 1. □

Handling Symbolic Predicates in UX Encoding

As with our OX encoding, we do not directly encode symbolic predicates in the UX encoding. However, unlike in the OX setting, it is not sound in UX to assume that any symbolic state with symbolic predicates has a satisfying concrete memory. In the UX setting, if the encoding is satisfiable, then the symbolic state itself must truly be satisfiable. Therefore, we conservatively assume that any symbolic state with a non-empty set of symbolic predicates is unsatisfiable.

We remark that the most naive — and ultimately unhelpful — UX encoding would treat all symbolic states as unsatisfiable, regardless of their structure.

4.3.8. Unfolding Symbolic Predicates

This conservative approach to handling symbolic predicates in the UX encoding is too restrictive in practice. For example, in the context of bug-finding, an error can only be reported if the corresponding symbolic state is under-approximately satisfiable. If we were to treat all states with symbolic predicates as unsatisfiable, this would prevent many genuine bugs from being reported.

To address this limitation, we introduce an unfolding mechanism for symbolic predicates. Recall the produce operation from Section 4.1.7. We apply this operation to unfold symbolic predicates in the hope that doing so will eliminate them. As a result of the UX soundness property of the produce operation (Property 4), if any of the resulting states after unfolding are satisfiable, then the original symbolic state must also be satisfiable.

We formalise this process by defining a judgement unfold, which takes a symbolic state and systematically unfolds its predicates:

$$\frac{}{\mathsf{unfold}(\hat{\sigma}) \rightsquigarrow \hat{\sigma}} \qquad \frac{\begin{array}{c}\mathsf{unfold}(\hat{\sigma}) \rightsquigarrow \hat{\sigma}'' \\ \hat{\mathcal{P}} = \hat{\sigma}''.\mathsf{pred} \\ p(\vec{E}_1; \vec{E}_2) \in \hat{\mathcal{P}} \\ \hat{\mathcal{P}}' = \hat{\mathcal{P}} \setminus \{p(\vec{E}_1; \vec{E}_2)\} \\ \hat{\sigma}''' = \hat{\sigma}''[\mathsf{pred} := \hat{\mathcal{P}}'] \\ p(\vec{x}_1; \vec{x}_2)\{A\} \in \mathsf{Preds} \\ \hat{\theta} = \{\vec{x}_1 \mapsto \vec{E}_1,\ \vec{x}_2 \mapsto \vec{E}_2\} \\ \mathsf{produce}(A, \hat{\theta}, \hat{\sigma}''') \rightsquigarrow \hat{\sigma}'\end{array}}{\mathsf{unfold}(\hat{\sigma}) \rightsquigarrow \hat{\sigma}'}$$

When checking the satisfiability of a symbolic state in the UX setting, we proceed by applying the unfolding operation to generate a set of new symbolic states. We then check whether any of these unfolded states are satisfiable. It is permissible to explore an arbitrary number of such unfolded states during this process. Importantly, in a UX context it is always sound to return `UNSAT`. Therefore, if we fail to discover any satisfiable unfolded symbolic state, we conservatively conclude that the original symbolic state is unsatisfiable and terminate the check.

4.3.9. Encoding Entailment

Lastly, we define an encoding which (under-approximately) determines the satisfiability of the entailment of logical expressions:

$$\mathsf{encImpl} : (\mathsf{SState} \times \mathsf{LExp}) \to 2^{\mathsf{Term}}$$

If $\mathsf{encImpl}(\hat{\sigma}, E)$ is unsatisfiable, then the entailment $\hat{\sigma} \models E$ holds. It is defined as:

$$\mathsf{encImpl}(\hat{\sigma}, E) = \begin{cases} \mathsf{encSAT}^{\mathsf{OX}}(\hat{\sigma}) & \hat{T} \not\vdash E : \mathsf{Bool} \\ \mathsf{encSAT}^{\mathsf{OX}}(\hat{\sigma}) \ \cup\ \{\neg(\varphi \wedge \bigwedge \Phi)\} & \text{otherwise} \end{cases}$$

where $(\varphi, \sigma, \Phi) = \mathsf{toBool}(\mathsf{encLExp}_{\hat{T}}(E))$.

To gain an informal understanding of this encoding, we consider each case of its definition in turn. In the first case of the definition, where $\hat{T} \not\vdash E : \mathsf{Bool}$, we know that $[\![E]\!]_\theta \notin \mathsf{Bool}$ and so

can never be true. Then, the entailment $\hat{\sigma} \models E$ can only hold vaccuously, if $\hat{\sigma}$ is unsatisfiable. The OX encoding guarantees that the encoding is unsatisfiable only if $\hat{\sigma}$ is unsatisfiable.

In the other case of the definition, consider a scenario in which the encoding is UNSAT, but $\hat{\sigma}$ is SAT. Because of the OX encoding of $\hat{\sigma}$, it must then be the case that $\neg E$ is unsatisfiable, and so the entailment must hold

Theorem 3. *The entailment encoding is well-defined. For all $\hat{\sigma}$, $E \in \mathsf{LExp}$:*

$$\exists \Phi \in 2^{\mathsf{Term}}.\ \mathsf{encImpl}(\hat{\sigma}, E) = \Phi$$

Proof. We note that $\mathsf{encSAT}^{\mathsf{OX}}$ is total. Suppose that $\hat{T} \nvdash E : \mathsf{Bool}$. We have trivially that our encoding is well defined in this case. On the other hand, if $\hat{T} \vdash E : \mathsf{Bool}$, it then follows from Lemma 7 that $\mathsf{encLExp}_{\hat{T}}(E)$ is defined. In particular, from Lemma 17 (Appendix C.4), we have that $\mathsf{encLExp}_{\hat{T}}(E) = (\varphi, \sigma, \Phi)$ where $\sigma = \sigma_{\mathsf{Bool}}$ or $\sigma = \sigma_{\mathsf{LVal}}$. In both cases, we know $\mathsf{toBool}(\varphi, \sigma, \Phi)$ is well-defined. Therefore, we conclude that $\mathsf{encImpl}(\hat{\sigma})$ is well-defined. □

4.4. Bridging CSE and SMT-LIB

With our encoding established, we aim to prove its correctness. In order to achieve this, we first introduce the formalisms required to marry the formalisations of SMT-LIB and CSE, in order to reason about our encoding.

4.4.1. Typing Environments and the Background Signature

We begin by considering the background signature Σ (Section 4.3.1). Our encoding functions (in particular $\mathsf{encLExp}_{\hat{T}}$) work under the assumption that logical variables are sorted in alignment with the typing environment $\hat{T}$. Formally, the semantics of our encodings are derived from *variants* (Section 4.2.2) of the background signature, where variables are sorted in alignment with the typing environment. We write $\Sigma_{\hat{T}}$ as a shorthand to represent the variant of Σ with logical variables sorted according to the typing environment $\hat{T}$:

$$\Sigma_{\hat{T}} = \Sigma[x : \mathsf{encTys}(\tau) \mid x : \tau \in \hat{T}][x : \sigma_{\mathsf{LVal}} \mid x \notin \mathsf{dom}(\hat{T})]$$

Informally, this sets the sort of a variable according to the type environment, if the type of the variable is known, and defaults the sort of a variable to σ_{LVal} otherwise. We sometimes write $\Sigma_{\hat{\sigma}}$ to mean $\Sigma_{\sigma.\mathsf{te}}$, representing the signature with logical variables sorted according to the typing environment of the symbolic state $\hat{\sigma}$.

4.4.2. LVal and $\Sigma_{\hat{T}}$-structures

Let $\Sigma_{\hat{T}}$ be a variant of the background signature Σ. Since $\Sigma_{\hat{T}}$ differs from Σ only by the sort it assigns to variables, we know that the set of Σ-structures is equivalent to the set of $\Sigma_{\hat{T}}$-structures. In particular, from the definition of our background signature Σ, we have that every $\Sigma_{\hat{T}}$-structure $\mathbf{A}$ consists of:

- a universe A such that:

$$A = \sigma_{\mathsf{Int}}^{\mathbf{A}} \cup \sigma_{\mathsf{Bool}}^{\mathbf{A}} \cup \sigma_{\mathsf{LVal}}^{\mathbf{A}} \cup \bigcup\{\sigma_{\tau}^{\mathbf{A}} \mid \tau \in \mathsf{Datatype}\} \cup \bigcup\{(\sigma_{\mathsf{Seq}}\sigma)^{\mathbf{A}} \mid \sigma \in \mathsf{Sort}(\Sigma)\}$$

- interpretations of all function symbols in Σ – $f^{\mathbf{A}}$.

We note that the universe of $\mathbf{A}$ is not equal to our set of logical values, LVal, primarily because the universe contains additional elements for the σ_{LVal} ADT. We can however, define a mapping $[\![\cdot]\!]_{\mathbf{A}}$, which maps values from the universe of the $\Sigma_{\hat{T}}$-structure $\mathbf{A}$, to the set of logical values LVal (Appendix D.1). The mapping is defined obviously, unwrapping values from σ_{LVal} ADT, and so removing all occurences of the ADT.

4.4.3. Logical Variable Substitutions and Valuations

Another consequence of this disparity is that the logical variable substitutions $\theta : \mathsf{LVar} \to \mathsf{LVal}$ of CSE are not valid valuation into $\Sigma_{\hat{T}}$-structures.

To circumvent this disparity between the formalisms, we introduce the total mapping $[\![\cdot]\!]_{\Sigma_{\hat{T}},\mathbf{A}}$, which, given a logical variable substitution $\theta : \mathsf{LVar} \to \mathsf{LVal}$, returns a valuation into the $\Sigma_{\hat{T}}$-structure $\mathbf{A}$. The specifics of this mapping are detailed in Appendix D.2. The mapping itself is uninteresting, simply wrapping logical values in the σ_{LVal} ADT where necessary. In doing so, it ensures that the resulting valuation adheres to the sorting of variables within the signature $\Sigma_{\hat{T}}$:

Lemma 10. *$[\![\theta]\!]_{\Sigma_{\hat{T}},\mathbf{A}}$ results in a valid valuation into the $\Sigma_{\hat{T}}$-structure $\mathbf{A}$. Formally, let $\Sigma_{\hat{T}}$ be a variant of the background signature Σ, and $\mathbf{A}$ be a $\Sigma_{\hat{T}}$-structure with universe A. For all $\theta : \mathsf{LVar} \to \mathsf{LVal}$, let $\theta' = [\![\theta]\!]_{\Sigma_{\hat{T}},\mathbf{A}}$. Then:*

$$\theta' : \mathsf{LVar} \to A$$

and for all $x \in \mathsf{LVar}$:

$$x : \sigma \in \Sigma_{\hat{T}} \implies \theta'(x) \in \sigma^{\mathbf{A}}$$

For brevity of notation, we introduce $\theta \models_{\Sigma_{\hat{T}}} \Phi$ to mean:

$$\exists \mathbf{A} \in \mathcal{M}(\mathcal{T}^{\Sigma_{\hat{T}}}).\ [\![\theta]\!]_{\Sigma_{\hat{T}},\mathbf{A}} = \theta' \wedge \theta', \mathbf{A} \models_{\mathtt{SMT}} \Phi$$

and $\mathsf{SAT}_{\Sigma_{\hat{T}}}$, meaning:

$$\exists \theta,\ \mathbf{A} \in \mathcal{M}(\mathcal{T}^{\Sigma_{\hat{T}}}).\ \theta, \mathbf{A} \models_{\mathtt{SMT}} \Phi$$

4.5. Correctness

We now prove various correctness results about our encoding.

4.5.1. Well-Sortedness

To prove soundness of our encoding, we first show that all generated SMT-LIB terms are well-sorted.

Theorem 4 (Background Assertions Well-Sorted). *All formulae in the background assertions are well-sorted. For all variants of the background signature, $\Sigma_{\hat{T}}$:*

$$\Sigma_{\hat{T}} \vdash \Psi : \sigma_{\mathsf{Bool}}$$

Theorem 5 (Encodings Well-Sorted). *All formulae generated by* $\mathsf{encSAT}^{\mathsf{OX}}$ / $\mathsf{encSAT}^{\mathsf{UX}}$ / $\mathsf{encImpl}$ *are well sorted. Let* $\Phi = \mathsf{encSAT}^{\mathsf{OX}}(\hat{\sigma})$ *(resp.* $\mathsf{encSAT}^{\mathsf{UX}}(\hat{\sigma})$ *and* $\mathsf{encImpl}(\hat{\sigma}, E)$*). Then:*

$$\Sigma_{\hat{\sigma}} \vdash \Phi : \sigma_{\mathsf{Bool}}$$

In order for these theorems to hold, we require that encoding functions provided by the instantiation are well-sorted:

Instance Property 3. *We require* $\mathsf{encSMem}_{\hat{T}}$ *to return well sorted terms / formulae. If* $\Phi = \mathsf{encSMem}_{\hat{T}}(\hat{\mu})$*, then:*

$$\Sigma_{\hat{T}} \vdash \Phi : \sigma_{\mathsf{Bool}}$$

To prove these well-sortedness theorems, we show that $\mathsf{encLExp}_{\hat{T}}$ produces well-sorted terms / formulae.

Lemma 11. *The encoding of logical expressions, produces well-sorted terms and formulae. Suppose* $\mathsf{encLExp}_{\hat{T}}(E) = (t, \sigma, \Phi)$*. Then:*

$$\Sigma_{\hat{T}} \vdash t : \sigma \ \wedge \ \Sigma_{\hat{T}} \vdash \Phi : \sigma_{\mathsf{Bool}}$$

Proof. See Appendix E.1. □

It can be shown (from Lemma 11) that $\mathsf{encPC}_{\hat{T}}$ and $\mathsf{encSto}_{\hat{T}}$ produce well-sorted formulae. Thus, Theorem 4 and Theorem 5 can be shown to be true.

4.5.2. Soundness

We now show our encoding to be both OX and UX sound. We begin by stating our top-level soundness theorems:

Theorem 6 (OX SAT Soundness). *Let* $\mathsf{encSAT}^{\mathsf{OX}}(\hat{\sigma}) = \Phi$*, then:*

$$\mathtt{SAT}(\hat{\sigma}) \implies \mathtt{SAT}_{\Sigma_{\hat{\sigma}}}(\Phi \cup \Psi)$$

Theorem 7 (UX SAT Soundness). *Let* $\mathsf{encSAT}^{\mathsf{UX}}(\hat{\sigma}) = \Phi$*, then:*

$$\mathtt{SAT}_{\Sigma_{\hat{\sigma}}}(\Phi \cup \Psi) \implies \mathtt{SAT}(\hat{\sigma})$$

Theorem 8 (UX Unfolding Soundness). *Unfolding symbolic predicates is UX sound:*

$$\mathsf{unfold}(\hat{\sigma}) \rightsquigarrow \hat{\sigma}' \,\wedge\, \mathtt{SAT}(\hat{\sigma}') \implies \mathtt{SAT}(\hat{\sigma})$$

Theorem 9 (Entailment Soundness). *Let* $\mathsf{encImpl}(\hat{\sigma}, E) = \Phi$*, then:*

$$\mathtt{UNSAT}_{\Sigma_{\hat{\sigma}}}(\Phi \cup \Psi) \implies \hat{\sigma} \models E$$

We prove Theorem 6 & Theorem 7 by proving stronger lemmas, which states that the same model of logical variables satisfies both the the symbolic state, and its encoding:

Lemma 12. *Let* $\mathsf{encSAT}^{\mathsf{OX}}(\hat{\sigma}) = \Phi$*:*

$$\forall \theta, \mu, s.\ \theta, (\mu, s) \models \hat{\sigma} \implies \theta \models_{\Sigma_{\hat{\sigma}}} (\Phi \cup \Psi)$$

Lemma 13. *Let* $\mathsf{encSAT}^{\mathsf{UX}}(\hat{\sigma}) = \Phi$*:*

$$\forall \theta.\ \theta \models_{\Sigma_{\hat{\sigma}}} (\Phi \cup \Psi) \implies \exists \mu, s.\ \theta, (\mu, s) \models \hat{\sigma}$$

In order to prove these lemmas, we must prove that our encoding of logical expressions is correct. We formulate the following lemma about the semantics of logical expression encodings, to enable us to prove the top-level lemmas:

Lemma 14. *Suppose that:*

$$\begin{gathered}(t, \sigma, \Phi) = \mathsf{encLExp}_{\hat{T}}(E) \\ \mathbf{A} \in \mathcal{M}(\mathcal{T}^{\Sigma_{\hat{T}}}) \\ \mathbf{A} \models_{\mathtt{SMT}} \Psi \\ [\![E]\!]_{\theta} = v \\ [\![\theta]\!]_{\Sigma_{\hat{T}}, \mathbf{A}} = \theta'\end{gathered}$$

Then:

$$\begin{gathered}v \neq \lightning \,\wedge\, \theta \models_{\mathsf{Type}} \hat{T} \\ \implies \\ \theta', \mathbf{A} \models_{\mathtt{SMT}} \Phi \,\wedge\, \exists v' \in A.\ [\![t]\!]_{\theta', \mathbf{A}} = v' \,\wedge\, [\![v']\!]_{\mathbf{A}} = v\end{gathered}$$

and also:

$$\theta', \mathbf{A} \models_{\texttt{SMT}} \Phi \ \wedge \ [\![t]\!]_{\theta', \mathbf{A}} = v'$$
$$\Longrightarrow$$
$$[\![v']\!]_{\mathbf{A}} = v \ \wedge \ v \neq \lightning \ \wedge \ \theta \models_{\mathsf{Type}} \hat{T}$$

Proof. This is an interesting proof, that uses *functional induction* over the definition of logical expression evaluation, $[\![\cdot]\!]_\theta$. The most interesting case is that of function application. Functional induction is crucial here, as it allows us to formulate an induction hypothesis for the body of the function definition. Details of the proof can be found in Appendix E.2. □

In order to prove soundness results, we also require some instantiation lemmas, about the semantics of $\mathsf{encSMem}_{\hat{T}}$:

Instance Property 4. *Suppose that* $\Phi = \mathsf{encSMem}_{\hat{T}}(\hat{\mu})$*. We require that for all* $\mathbf{A} \in \mathcal{M}(\mathcal{T}^{\Sigma_{\hat{T}}})$ *where* $\mathbf{A} \models_{\texttt{SMT}} \Psi$*, and* $[\![\theta]\!]_{\Sigma_{\hat{T}}, \mathbf{A}} = \theta'$*:*

$$\exists \mu.\ \theta, \mu \models_{\mathsf{Mem}} \hat{\mu}$$
$$\Longleftrightarrow$$
$$\theta', \mathbf{A} \models_{\texttt{SMT}} \Phi$$

We can also prove the following lemmas about $\mathsf{encPC}_{\hat{T}}$ and $\mathsf{encSto}_{\hat{T}}$:

Lemma 15. *Suppose that* $\Phi = \mathsf{encPC}_{\hat{T}}(\hat{\pi})$*. Then, that for all* $\mathbf{A} \in \mathcal{M}(\mathcal{T}^{\Sigma_{\hat{T}}})$ *where* $\mathbf{A} \models_{\texttt{SMT}} \Psi$*, and* $[\![\theta]\!]_{\Sigma_{\hat{T}}, \mathbf{A}} = \theta'$*:*

$$[\![\hat{\pi}]\!]_\theta = \mathsf{true} \ \wedge \ \theta \models_{\mathsf{Type}} \hat{T}$$
$$\Longleftrightarrow$$
$$\theta', \mathbf{A} \models_{\texttt{SMT}} \Phi$$

Proof. This follows mostly as a result of Lemma 14. Details of the proof can be found in Appendix E.3 □

Lemma 16. *Suppose that* $\Phi = \mathsf{encSto}_{\hat{T}}(\hat{s})$*. Then, that for all* $\mathbf{A} \in \mathcal{M}(\mathcal{T}^{\Sigma_{\hat{T}}})$ *where* $\mathbf{A} \models_{\texttt{SMT}} \Psi$*, and* $[\![\theta]\!]_{\Sigma_{\hat{T}}, \mathbf{A}} = \theta'$*:*

$$\exists s.\ \theta, s \models_{\mathsf{Sto}} \hat{s} \ \wedge \ \theta \models_{\mathsf{Type}} \hat{T}$$
$$\Longleftrightarrow$$
$$[\![\theta]\!]_{\Sigma_{\hat{T}}, \mathbf{A}} = \theta' \ \wedge \ \theta', \mathbf{A} \models_{\texttt{SMT}} \Phi$$

Proof. Again, this lemma follows mostly as a result of the soundness result for logical expression encodings Lemma 14. Details of this proof can be found in Appendix E.4 □

Our soundness results follow almost directly from Lemma 15, Lemma 16 and Instance Property 4. Details of these proofs can be found in Appendix E.5, Appendix E.6, Appendix E.7 and Appendix E.8.

4.6. Findings from Theory

Although our formalism aims for general applicability to compositional symbolic execution (CSE) tools, its development was intimately informed by Gillian's practical implementation. We will

highlight in this section the key findings from our formalism's development that offer concrete improvements for Gillian in practice.

4.6.1. Typing Environment

As discussed in Section 3.3, Gillian encodes variables using their native SMT-LIB sort, if the type of this variable is known, and uses an ADT to wrap variables whose type is not known. This is achieved, as in our formalism, by maintaing a typing environment during symbolic execution, along with the remaining symbolic state. We recall from our formalism that adding information to the typing environment is equivalent to adding constraints to the path condition. As such, care must be taken in maintaining the typing environment during symbolic execution.

Currently, typing information is inferred into the typing environment by running a standard type inference algorithm on logical expressions. This occurs mainly through two routes:

1. **Expression Evaluation** – when expressions (e.g. in the program body) are evaluated, type information is inferred.
2. **Path Condition** – type information is inferred during symbolic execution from the path condition of the symbolic state.

The first route (expression evaluation) is unexpected, and doesn't adhere to our principle of only inferring typing information that is implied by the symbolic state. For an illustration of how this leads to OX unsoundness, consider this example:

```
// ...
if (x == 0) {
  // x : Int
} else {
  // x : Int
}
// x : Int
```

Suppose that we know our program variable x is equal to a logicale variable x. When the expression $\mathsf{x} = 0$ is evaluated, the typing information $x : \mathsf{Int}$ is added to the typing environment. However, this information is not removed when evaluating the else branch, or even after the if / else statement. This has the effect of adding this constraint to the symbolic state, in effect restricting our symbolic state to represent a subset of concrete states that are realisable in practice. For example, a valid concrete execution of the program occurs where $\mathsf{x} \in \mathsf{Bool}$, and the else branch is executed. This execution path is not explored by Gillian, because of this over-eager inference of types.

4.6.2. Partiality

A major source of potential unsoundness is that partiality is not considered at all in the encoding of symbolic state into SMT-LIB. In practice, this does not lead to unsoundness in most cases, because this is typically handled at the level of compilation into GIL. We recall that Gillian targets an intermediate verification language, called GIL. This is a small goto language, which

program specifications from the instantiation are compiled into. Currently, partiality is handled in this compilation process. For example, when compiling division operations into GIL in the C-instantiation, the following procedure is generated:

```
proc i__binop_div(v1, v2) {
       goto [ l-nth(v1, 0i) = "int" ] llon unde;
  llon: goto [ l-nth(v2, 0i) = "int" ] blon unde;
       assert (! (l-nth(v2, 1i) == 0i));
  blon: ret := {{ "int", l-nth(v1, 1i) i/ l-nth(v2, 1i) }};
       return;
  unde: fail[operator]("Using /lu operator for non-int elements")
};
```

In particular, we note that this procedure which wraps division, asserts that the divisor is non-zero.

We argue that this is not the correct approach to take in the designing of verification frameworks. The intermediate verification language should provide a simple and clean interface, abstracting over the specific semantics of SMT-LIB. Handling partiality during symbolic execution, as opposed to handling it at compile time, poses numerous benefits:

- **Separation of Concerns** – Handling partiality within the symbolic execution engine, rather than during compilation to the intermediate verification language (IVL), ensures a cleaner architectural separation. This also keeps the IVL semantics simpler and closer to a minimal core language. The semantics of the IVL should not be constrained by the semantics of SMT-LIB.

- **Decreased Burden on Instantiation Developers** – Requiring every instantiation to manually inject partiality guards during translation to GIL leads to duplicated logic, inconsistencies, and potential omissions. By centralising partiality handling in the symbolic execution engine, developers writing new frontends need not encode every low-level semantic check themselves. This improves modularity and makes instantiations faster and less error-prone to develop.

- **Smaller Unverified Gap** – Currently, correctness of the partiality handling relies on the instantiation-specific compiler producing correct and complete partiality-guards during translation. However, these compilers are typically unverified. By contrast, the symbolic execution engine and its core semantics are often the focus of formalisation and proof. Moving partiality handling into the symbolic semantics reduces the portion of the system that must be trusted but is not formally verified. This strengthens the overall soundness argument of the verification framework.

- **Edge Cases** – There are some instances in which partiality in the encoding cannot be handled at compile time. For example, recall how Gillian uses an ADT to wrap values whose type cannot be inferred. Recall also that ADT selectors are partial functions - they are ill-defined for values which were not constructed by the corresponding constructor. The manipulation of values in this ADT is not exposed to instantiation developers, and

introduces partiality into the encoding. This partiality must be handled during symbolic execution, as opposed to at compile time.

4.6.3. UX Encoding

The final source of unsoundness in Gillian is that the encoding only encodes the path condition, neglecting other aspects of the symbolic state. While this is sound in OX, and even desirable in order to ensure that SAT checks are cheap, in a UX setting this leads to unsoundness.

4.6.4. Towards Language Parametricity

The next generation of Gillian aims to be parametric on not just the memory model, but also the target language itself. For context, the process of compiling a target language into GIL does not occur in a single pass [6]. Instead, current instantiations of Gillian rely on verified and trusted compilers to lower the source language to an intermediate representation that is closer in abstraction to GIL. This intermediate representation is then translated into GIL via an additional compilation phase. For example, the Gillian-C instantiation leverages the formally verified *CompCert* compiler [12] to compile C programs into *C# Minor*, a lower-level intermediate language. Similarly, the JavaScript instantiation utilises *JaVerT's* verified compiler [7], which translates JavaScript into an intermediate representation called *JSIL*. These verified compilers provide strong guarantees about the correctness of their respective translations.

However, while the first compilation step – transforming the source language into the intermediate representation – is formally verified, the subsequent step of compiling this intermediate representation into GIL is not. This introduces a significant source of unsoundness in the overall verification process, forming a major bottleneck in the certification of Gillian's symbolic execution framework. Although it would be possible to prove the soundness of the compilation phase for each individual instantiation of Gillian, this approach does not scale well. Each new language instantiation would require an independent proof of soundness for its respective compilation process, which becomes increasingly costly as more languages are incorporated into the platform.

This issue is compounded by the inherent difference in expression languages. It then becomes the burden of the instantiation developer to reconcile these differences, whilst preserving the semantics of the target language. This approah is complex, and error-prone.

One potential approach to overcoming this challenge is to eliminate the need for the unverified compilation step entirely by making Gillian parametric on the language. Rather than translating programs into GIL and executing them symbolically, Gillian could instead operate directly on the intermediate representations, such as C# Minor and JSIL. Since these representations already have verified compilers from their respective source languages, this approach would remove the need for an additional, potentially unsound translation step. Concretely, this would involve designing symbolic interpreters for these intermediate representations, allowing Gillian to reason about programs at this level while maintaining the guarantees provided by the verified compilers.

Our formal work provides useful insights into how the encoding of SMT might work in a language-parametric setting. Recall our three encoding functions: $\mathsf{encSAT}^{\mathsf{OX}}$, $\mathsf{encSAT}^{\mathsf{UX}}$ and $\mathsf{encImpl}$. These are all underpinned by the encoding of logical expressions: $\mathsf{encLExp}_{\hat{T}}$. In a language-parametric setting, this encoding of logical expressions will also be instantiation specific,

and so must be provided by each instantiation. Our formalism has allowed us describe precisely the properties this encoding must satisfy, in order to achieve OX and UX soundness.

5. Extending Gillian with User-Defined Datatypes and Functions

In this chapter we extend Gillian with support for user-defined datatypes and functions, exploring and motivating details of the implementation.

We first motivate the need for user-defined datatypes and functions. In order to reason about data structures, it is often necessary to maintain a logical representation of their contents. For example, when verifying programs that manipulate list-like structures, we may wish to express and prove properties such as sortedness, membership, or the preservation of certain invariants across operations. To support this, Gillian currently provides built-in logical support for lists and sets, which can be used to abstractly represent the contents of memory-based structures during verification.

However, this support is limited. As soon as we wish to reason about more complex or recursive data structures – such as binary search trees, graphs, or abstract syntax trees – the lack of user-defined datatypes becomes a serious limitation. While it is sometimes possible to encode the logical state of a binary search tree, for example, using a list of its pre-order traversal, this approach is indirect, cumbersome, and fragile. Moreover, it obscures the natural recursive structure of the data, complicating specifications and proofs. These issues are exacerbated in domains that rely on structured recursive data, such as the verification of compilers or interpreters, where abstract syntax trees are central. In such settings, the inability to define custom algebraic data types restricts expressiveness and makes specifications less natural and more error-prone. By extending Gillian to support user-defined datatypes and functions, we allow users to define logical representations that mirror the structure of the data they work with. This enables more modular, concise, and robust specifications, and broadens the applicability of Gillian to a wider range of verification problems.

Our implementation efforts focus on targetting the core Gillian platform, adding support for the handling of user-defined datatypes and functions. We also extend the WISL frontend to add support for these constructs. We detail briefly some interesting aspects of our implementation.

5.1. Syntax

Our implementation begins with extending the syntax of GIL to support user-defined datatypes and functions. We first extend the AST with datatype and function declarations. In addition, we extend the GIL expression AST with:

- **Constructor Application** – We allow the application of datatype constructors on sub-expressions.

- **Function Application** – We allow the application of user-defined functions on sub-expressions.
- **Case Expressions** – We allow basic pattern matching on datatype constructors, by implementing case expressions.

Furthermore, we extend the set of types to allow us to represent user-defined datatypes.

We similarly extend the WISL AST, and extend the compilation of WISL to GIL to support these additions. Our additions to the syntax allow us to write (in WISL) the following user-defined datatype and function declarations:

```
datatype MyList {
  Nil;
  Cons(Any, MyList)
}
pure function append(xs : MyList, x) {
  case xs {
    Nil -> 'Cons(x, 'Nil);
    Cons(y, ys) -> 'Cons(y, append(ys, x))
  }
}
```

We note in particular that we allow users to optionally restrict the type that can be passed into functions and constructors. For example, the second argument to `Cons` must be a `MyList`. Also, the first argument to `append` must be a `MyList`. This type information is entirely optional, and we assume that there are no restrictions imposed if this is not provided. This design choice allows us to remain true to the dynamic nature of Gillian, while still benefiting from type information where it is provided to us.

GIL expressions form a core part of the AST, and extending it requires us to extend all aspects of the symbolic interpreter which interact with expressions. For example, we extend type inference to handle constructor and function applications. For brevity, we do not detail the specifics of our additions in these cases, as they are largely uninteresting.

5.2. Matching Plans

One interesting aspect of our implementation interacts with matching plans. During the consume operation, the assertion being consumed may have some logical variables which are known, and others (which are existentially quantified) whose value we must infer based on the current symbolic state. Matching plans facilitate this, by planning an order in which to match parts of the assertion. With each step, more information is gained, allowing us to match further.

One key aspect of the matching plan algorithm is the learning of unknown logical variables value given an equality that holds in the current symbolic state. For a simple example, consider a scenario where I have a known variable, x, and an unknown variable, y. We are given the equality $x = y - 1$. In order to *learn* the value of y, we can use the laws of arithmetic to obtain that $x + 1 = y$. We analagously extend this algorithm to allow us to learn unknown values from

constructor applications. Consider the equality $x = C(y_1, \ldots, y_n)$. For some constructor C. It must then be the case that $x = C(x_1, \ldots, x_n)$ for some $x_1, \ldots, x_n$, and $y_i = x_i$. We can use this insight to invert the equality using case expressions:

```
y_1 = case x {
  C(x_1, ..., x_n) -> x_1
}
```

We extend the matching plan algorithm using this method. This allows greater flexibility and increased automation in verification involving user-defined datatypes. We note however, that there is no obvious way to invert a function, as such we are not able to learn from expressions involving function application.

5.3. Reduction

Before queries are dispatched to the SMT solver, all logical expressions are first reduced. We extend this reduction engine to eliminate where possible constructor applications. This is important, as it significantly improves the performance of our queries when ADTs are not involved. We encode manually a number of obvious reductions. For example, the boolean expression $C(\vec{x}) = C(\vec{y})$ is reduced to $\vec{x} = \vec{y}$.

5.4. Encoding

Most importantly, we implement an encoding of these constructs into SMT. We defer the majority of the reasoning about these constructs to the SMT solver, apart from the minor reductions detailed above.

We observed in Section 3.4 that CN makes use of the native `declare-datatypes` syntax in order to declare its datatypes. On the other hand, Viper and VeriFast are declaring uninterpreted functions, along with some axioms to ensure their behaviour as ADTs. However, we observed that it is difficult to completely capture all of the axioms necessary to capture the behaviour of ADTs. In particular, encoding acyclicity is difficult. We require a UX sound encoding, and so it is vital that our encoding captures all of the axioms. From our experimentation, Z3 enforces all axioms, when using the `declare-datatypes` syntax. Since Gillian uses Z3 as its SMT backend, we chose to encode datatypes using this native method, favouring soundness over any potential performance benefits of other encodings. One added benefit of this approach is that we can cleanly use SMT-LIB's in-built `match` expressions to encode our case statements. These `match` statements allow us to pattern match on ADTs within SMT-LIB. Lastly, in order to encode user-defined functions, we use the `define-funs-rec` syntax.

5.4.1. Partiality in Selectors

We recall from our encoding that datatype selectors are partial functions, and as such must be handled with care in the encoding. While Gillian does not handle partiality in other aspects of

its encoding, we make efforts to rectify this, by ensuring partiality is correctly handled in the handling of ADTs.

6. Evaluation

This work aimed to target both theoretical and practical gaps in the current state of compositional symbolic execution. Theoretically, we aimed to formalise the encoding of symbolic state into SMT-LIB, and prove soundness of this encoding. Practically, we aimed to survey exisiting approaches taken by CSE tools. We also aimed to extend Gillian with support for user-defined datatypes and functions. In this chapter, we evaluate the extent to which these aims were met.

6.1. Theoretical Evaluation

This work has achieved its primary goal of formalising, and proving sound, the encoding of symbolic state into SMT-LIB. The formalisation reflects accurately the challenges faced in practice. In particular, we have accurately captured:

- **Partiality** – Our formal encoding presents a sound method of handling partiality in the encoding, be it through partiality in the expression language, partiality in user-defined functions, or partiality in the manipulation of ADTs.
- **Typing** – We capture and formalise approaches to handle dynamic typing during symbolic execution. Our formalism provides a precise semantics for the typing environment. It also captures approaches taken to encode dynamically typed expressions within SMT-LIB. In particular, this aspect of our formalism models accurately the approach taken within Gillian.
- **User Defined Functions and Datatypes** – Our formalism handles encoding of user-defined functions and datatypes, and adds support for these constructs in the expression language. This is aspect has not been formalised in past works.
- **Memory Model Parametricity** – Our formalism is designed to be memory model parametric, increasing the applicability of our theory to a broader range of CSE tools.
- **Soundness** – Most importantly, our formal encoding has beeen proven sound.

Our theoretical work is general, and can be applied to a wide range of CSE tools, while still accurately capturing all non-trivial aspects of encoding into SMT-LIB.

6.1.1. Learnings from Theory

In addition to accurately capturing practical approaches to encoding into SMT, our formalism has allowed us to gain useful insights that can be applied in practice. This includes:

- **Unexpected Behaviour in Gillian** – Our formalism has allowed us to spot potentially unsound and unexpected behaviour in Gillian. This includes inconcistencies in Gillian's approach to type inference during symbolic execution, potential UX unsoundness, and also a lack of handling of partiality in Gillian's encoding into SMT.
- **Towards Language Parametricity** – We've provided a precise formal description of a soundness property for logical expression encoding. This foundational work offers a clear and precise basis for future research aimed at developing truly language-parametric CSE tools.

6.1.2. Limitations

While our work has been successful given the time constraints, there are still areas in which future work could be done to make improvements. One key limitation is that our soundness result is not mechanised, and is a pen-and-paper proof. Future work is required to mechanise our results, and incorporate this with existing mechanised CSE theory. Another limitation is that our formalism's expression language is somewhat simplistic, and not representative of realistic expression languages. We made this choice given the time constraints of the project. For example, our expression language is limited to integers and booleans, and does not model the "case" expressions which we added to Gillian. While we predict the extension of our formalism to a larger expression language to be straightforward, there might be challenges we have not forseen.

6.2. Practical Evaluation

We aimed to extend Gillian, adding support for user-defined functions and datatypes, in order to increase the usability of the platform. We now evaluate the extent to which this has been achieved in our implementation. In order to evaluate our implementation, we compare verification tasks on two datastructures – singly linked lists (SLLs) and binary search trees (BSTs). Concretely, we write (in WISL) a number of standard algorithms manipulating each datastructure. We then attempt to verify these algorithms, with and without the use of ADTs.

6.2.1. Ease of Verification

Verifying SLLs is a fairly straightforward task, even without ADTs. We can use an in-built list to represent logically the contents of an SLL. As such, we define our SLL predicate as follows:

```
predicate SLL(+x, vs) {
  (x == null) * (vs == []);
  (x -b> #v, #next) * SLL(#next, #vs) *
  (vs == #v :: #vs)
}
```

This simply states that there is a singly linked list, starting at address `x`, with contents `vs`. It is analagous to the list predicate introduced in earlier sections.

One awkward consequence of the lack of user-defined functions is that we must use predicates to model functions. For example, we define a list membership function using predicates:

```
predicate list_member(+vs, +v, r : Bool){
  (vs == []) * (r == false);
  (vs == v :: #rest) * (r == true) * list_member(#rest, v, #mem);
  (vs == #v :: #rest) * (! (#v == v)) * list_member(#rest, v, r)
}
```

This is inconvenient and awkward for a number of reasons. When using predicates to model functions, we use an out-parameter to model the return value of the function. In general, this method of defining functions is confusing and difficult to understand. This defeats the purpose of function specifications – they should be easy to reason about and simple, such that we can be confident in the correctness functions they specify.

With the introduction of ADTs, we now can define our own logical representation of lists:

```
datatype MyList {
  Nil;
  Cons(Any, MyList)
}
```

With this definition in place, we translate the list predicate introduced above to make use of this ADT:

```
predicate SLL(+x, vs) {
  (x == null) * (vs == 'Nil);
  (x -b> #v, #next) * SLL(#next, #vs) *
  (vs == 'Cons(#v, #vs))
}
```

The real advantage in this scenario is that we can now use a user-defined function to represent list membership:

```
pure function list_member(xs : MyList, x) {
  case xs {
    Nil -> false;
    Cons(y, ys) -> (y == x) || list_member(ys, x)
  }
}
```

This definition is far more readable than the definition that was possible without user-defined functions.

The advantages of ADTs are much more apparent when we consider the case of BSTs. These are inherently a recursive datastructure, and while it is possible to represent their contents using only lists, it is difficult, awkward, and makes verification challenging. Without ADTs the BST predicate is defined as:

```
predicate bst(+x, vs : List) {
  (x == null) * (vs == []);
  (x -b> #v, #l, #r) * bst(#l, #vsl) * bst(#r, #vsr) *
  (vs == (#v::#vsl @ #vsr)) *
  list_member(#vsl, #v, false) *
  list_member(#vsr, #v, false) *
  list_gt(#vsl, #v) *
  list_lt(#vsr, #v)
}
```

The contents of the BST is flattened into a list, in order to represent its structure. This flattening of the tree into a list is not obvious, and doesn't reflect the natural structure of trees. While somewhat managable in this case, other more complicated recursive datastructures become exceedingly challenging to model purely with lists. Again, we argue that this goes against the spirit of verification – the complexity should be within the algorithms we verify, not in their specifications.

We note in particular that in the evaluation of our extension, we attempted – and failed – to verify an expression tree evaluation algorithm without ADTs. This demonstrates how inflexible supporting just lists is. Being unable to verify recursive datastructures prevents us from, for example, being able to cleanly verify compilers, which work extensively with ASTs.

In the example above, the BST invariants are encoded using the `list_gt( ... )` and `list_lt( ... )` predicates. These predicates, which should really be functions, are awkward for the reasons discussed above.

With the introduction of ADTs, our predicate becomes greatly simplified:

```
predicate bst(+x, t) {
  (x == null) * (t == 'Leaf);
  (x -b> #k, #l, #r) * bst(#l, #tl) * bst(#r, #tr) *
  (t == 'Node(#k, #tl, #tr)) *
  (bst_gt(#tl, #k) == true) * (bst_lt(#tr, #k) == true)
}
```

A simple metric by which we can quantitatively measure the ease with which our algorithms can be verified, is by counting the number of lines of code required in each case. Figure 6.1 visualises this, categorising each line in the specification as either:

- **Function Specification** – This includes the actual algorithms, specifications for pre- / post-conditions of the functions, and logical commands / tactics (e.g. unfolds).
- **Predicates** – These are the lines used to define the predicates, with which the function specifications are specified.
- **Lemmas** – Inductive facts cannot be inferred, and must be proven by the user for Gillian to be able to use. These proofs are referred to as lemmas.
- **User Defined Datatypes / Functions** – These are the new constructs which we extended Gillian to support.

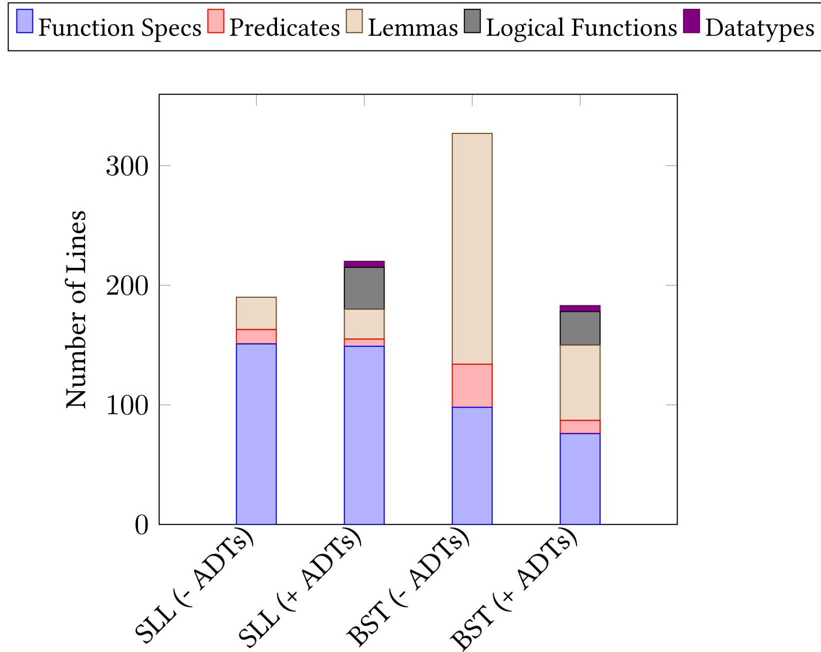

Figure 6.1. Number of lines required to verify various datastructures, with and without user-defined datatypes

In the case of SLLs, using ADTs slightly increases the number of lines of code required to verify the algorithms. Looking closer, we observe that this increase comes primarily from user-defined functions. This is because with ADTs, we no longer have access to in-built list operations, such as concatenation, and must define these for our user-defined datatypes manually. We notice however, that the use of ADTs does not necessitate us to prove more lemmas, in fact the exact lemmas required previously were required with ADTs. Overall, we argue that the ease of verification is similar in both cases.

In the case of BSTs, the difference is much more obvious, with the addition of ADTs decreasing the line count by approximately one third. Verification becomes significantly easier in the presence of ADTs in this case. There are two main factors for this:

- **Spatial Predicates for Functions** – Predicates exist at the spatial level in Gillian, and cannot be freely duplicated, even if they are pure. This requires lemmas to prove that they can be duplicated, and also it requires the application of these lemmas in the function specifications. This hinders automation of verification, and demonstrates clearly why the use of spatial predicates to model pure functions is problematic.

- **Additional Lemmas** – Since we use lists to model trees, we must prove a large number of lemmas in order to prove that the BST invariant is maintained across operations to these logical lists. These lemmas are complicated to formulate, and again increase the difficulty with which we are able to verify more complex datastructures.

We conclude that verification becomes significantly easier with the introduction of user-defined functions and datatypes. User-defined datatypes allow us to more closely model complex datastructures, meaning the reasoning is simplified. User-defined functions allow eliminate the awkwardness of using spatial predicates to model pure functions.

6.2.2. Performance

Another factor which impacts usability is the performance of the tool. Our extension cannot be considered usable unless it is sufficiently performant.

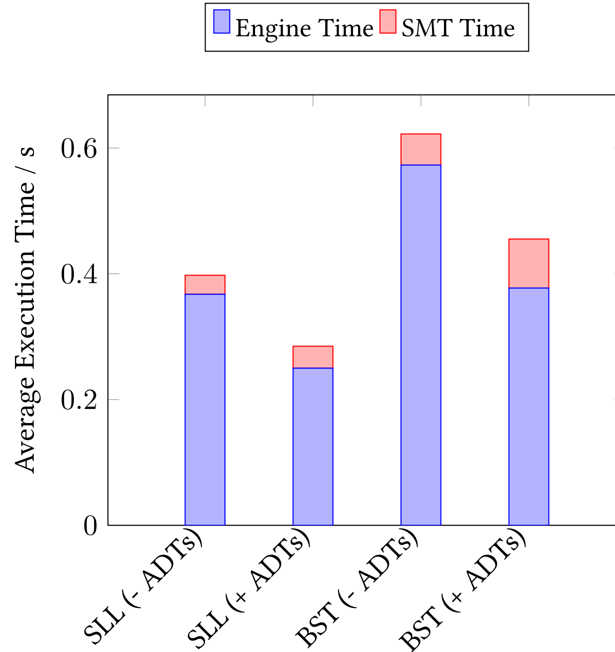

Figure 6.2. Execution time of Gillian in verifying various datastructures, with and without ADTs

Figure 6.2 compares the average execution times in the verification of the running example of BSTs and SLLs. We distinguish execution time between *engine time*, which is the time taken for Gillian to execute, and *SMT time*, which is the time taken for the SMT solver (which is Z3 in the case of our experiments) to solve queries. These results were obtained by running repeatedly ($n = 10$) Gillian on each verification task.

We notice that, surprisingly, ADTs and user-defined functions increase performance. While the SMT time increases slightly, the total execution time decreases, by approximately a quarter in both cases. While initally surprising, this makes sense when considering the transition between predicates and user-defined functions, to model pure functions. When using predicates, the engine must use its existing machinery, which is built to handle spatial reasoning, to unfold and fold recursive function definitions. With the introduction of user-defined functions, this burden is shifted to the solver, whose role it is to now handle the unfolding / folding of these functions.

We note that the introduction of user-defined, recursive functions into the SMT encoding introduces universal quantifiers. We have found in practice, the effects of this to be negligible. The SMT performance decrease is outweighed by the performance increase in the engine. Throughout our experimentation, we have only experienced one time-out as a result of user-defined functions / ADTs. When developing the SLL comparison, we wrote the following list reversal function:

```
pure function reverse(xs) {
  case xs {
    Nil -> 'Nil;
    Cons(x, xs) -> append(reverse(xs), x)
  }
}
```

In particular, no type hints were provided for `xs`, which is clearly a list. As such, the encoding assumes the input can take any type, and encodes the function as:

```
(define-funs-rec
 ((reverse ((xs Extended_GIL_Literal)) Extended_GIL_Literal))
 ((ite (and (isDatatypeMyList (singElem xs)) (isSingular xs))
   (match (MyListValue (singElem xs))
    (((Nil) (Elem (DatatypeMyList Nil)))
     ((Cons x xs)
      (append (MyListValue (singElem (reverse (Elem (DatatypeMyList xs)))))
       (Elem x)))
     (_ (Elem Undefined))))
   (Elem Undefined))))
```

Because the type of the input is not known, it is encoded as an `Extended_GIL_Literal`, which is an ADT. In the function body, we then access the `MyListValue` from this ADT, match it, then wrap it again in the ADT to pass into the recursive call. The function also returns an ADT, so the result of the recursive call must again be unwrapped. This deep nesting of ADT operations leads to the query timing out. This is an inherent drawback of dynamic typing, as it leads to these deeply nested ADTs. In practice, this can be easily avoided by supplying obvious type hints to the user-defined functions. In this case, we were matching on list constructors, so in fact, the only valid input to the function is a list. Adding the type hint to our function eliminates this timeout. In future work we could also implement more robust type inference on user-defined function bodies (e.g. inferring input parameter types / return type), reducing the likelihood with which these timeouts occur because of user error.

7. Conclusion

This project successfully addressed a key gap in the formal foundations of CSE, by developing and proving the soundness of an encoding of symbolic state into SMT-LIB. The theoretical contributions of this work are both general and practically relevant: the formalism accurately captures crucial challenges in encoding, such as dynamic typing, partiality, user-defined datatypes and functions. Importantly, we provided the first formal semantics for handling user-defined constructs within the encoding process, an aspect that prior formalisms have neglected.

A major achievement was the demonstration of soundness for both OX and UX reasoning in our encoding, thereby providing a robust foundation upon which future CSE tools can build. The formalism also yielded valuable insights applicable to real-world tools. In particular, we identified unexpected and potentially unsound behaviour in Gillian's handling of type inference and partiality, insights which can guide improvements in its implementation. Furthermore, the project advanced theoretical understanding by contributing a precise basis for the encoding of logical expressions, laying groundwork towards the development of language-parametric CSE tools.

From a practical standpoint, the extension of Gillian with support for user-defined datatypes and functions significantly enhanced its usability. Evaluation on standard data structures demonstrated not only improved expressiveness but also notable performance gains, as the burden of unfolding was shifted from the symbolic execution engine to the solver. These enhancements make verification of complex algorithms more natural and efficient within the Gillian framework.

7.1. Further Work

While we identified unexpected in behaviour in Gillian, we did not fix the issues that were identified. Further work is required in order to fully address and investigate these issues. Another important direction for further work is the mechanisation of our formalism and soundness result, as well as integration with the existing mechanised CSE theory.

Bibliography

[1] Roberto Baldoni et al. "A Survey of Symbolic Execution Techniques". In: *ACM Comput. Surv.* 51.3 (May 2018), 50:1–50:39. ISSN: 0360-0300. DOI: 10.1145/3182657.

[2] Haniel Barbosa et al. "Cvc5: A Versatile and Industrial-Strength SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Dana Fisman and Grigore Rosu. Cham: Springer International Publishing, 2022, pp. 415–442. ISBN: 978-3-030-99524-9. DOI: 10.1007/978-3-030-99524-9_24.

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.7*. Tech. rep. Department of Computer Science, The University of Iowa, 2025.

[4] Thibault Dardinier et al. "Formal Foundations for Translational Separation Logic Verifiers". In: *Formal Foundations for Translational Separation Logic Verifiers – Artifact* 9.POPL (Jan. 2025), 20:569–20:599. DOI: 10.1145/3704856.

[5] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer, 2008, pp. 337–340. ISBN: 978-3-540-78800-3. DOI: 10.1007/978-3-540-78800-3_24.

[6] José Fragoso Santos et al. "Gillian, Part i: A Multi-Language Platform for Symbolic Execution". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. New York, NY, USA: Association for Computing Machinery, June 2020, pp. 927–942. ISBN: 978-1-4503-7613-6. DOI: 10.1145/3385412.3386014.

[7] José Fragoso Santos et al. "JaVerT 2.0: Compositional Symbolic Execution for JavaScript". In: *Artifact accompanying the POPL'19 article "JaVerT 2.0: Compositional Symbolic Analysis for JavaScript"* 3.POPL (Jan. 2019), 66:1–66:31. DOI: 10.1145/3290379.

[8] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259.

[9] Bart Jacobs, Jan Smans, and Frank Piessens. *VeriFast: Imperative Programs as Proofs*. Workshop Handout (SecAppDev 2010). Katholieke Universiteit Leuven, Department of Computer Science, 2010.

[10] Bart Jacobs, Frédéric Vogels, and Frank Piessens. "Featherweight VeriFast". In: *Logical Methods in Computer Science* Volume 11, Issue 3 (Sept. 2015), p. 1595. ISSN: 1860-5974. DOI: 10.2168/LMCS-11(3:19)2015. arXiv: 1507.07697 [cs].

[11] Bart Jacobs et al. "VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java". In: *NASA Formal Methods*. Ed. by Mihaela Bobaru et al. Berlin, Heidelberg: Springer, 2011, pp. 41–55. ISBN: 978-3-642-20398-5. DOI: 10.1007/978-3-642-20398-5_4.

[12] Xavier Leroy et al. “CompCert - A Formally Verified Optimizing Compiler”. In: *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*. Jan. 2016.

[13] Andreas Lööw et al. “Compositional Symbolic Execution for Correctness and Incorrectness Reasoning”. In: *38th European Conference on Object-Oriented Programming (ECOOP 2024)*. Ed. by Jonathan Aldrich and Guido Salvaneschi. Vol. 313. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 25:1–25:28. ISBN: 978-3-95977-341-6. DOI: 10.4230/LIPIcs.ECOOP.2024.25.

[14] Andreas Lööw et al. “Compositional Symbolic Execution for the Next 700 Memory Models”. Unpublished manuscript provided by the author, 2025.

[15] Andreas Lööw et al. “Matching Plans for Frame Inference in Compositional Reasoning”. In: *38th European Conference on Object-Oriented Programming (ECOOP 2024)*. Ed. by Jonathan Aldrich and Guido Salvaneschi. Vol. 313. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 26:1–26:20. ISBN: 978-3-95977-341-6. DOI: 10.4230/LIPIcs.ECOOP.2024.26.

[16] Petar Maksimović et al. “Gillian, Part II: Real-World Verification for JavaScript and C”. In: *Computer Aided Verification*. Ed. by Alexandra Silva and K. Rustan M. Leino. Cham: Springer International Publishing, 2021, pp. 827–850. ISBN: 978-3-030-81688-9. DOI: 10.1007/978-3-030-81688-9_38.

[17] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. “Viper: A Verification Infrastructure for Permission-Based Reasoning”. In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Barbara Jobstmann and K. Rustan M. Leino. Berlin, Heidelberg: Springer, 2016, pp. 41–62. ISBN: 978-3-662-49122-5. DOI: 10.1007/978-3-662-49122-5_2.

[18] Peter O’Hearn, John Reynolds, and Hongseok Yang. “Local Reasoning about Programs That Alter Data Structures”. In: *Computer Science Logic*. Ed. by Laurent Fribourg. Berlin, Heidelberg: Springer, 2001, pp. 1–19. ISBN: 978-3-540-44802-0. DOI: 10.1007/3-540-44802-0_1.

[19] Christopher Pulte et al. “CN: Verifying Systems C Code with Separation-Logic Refinement Types”. In: *Proceedings of the ACM on Programming Languages* 7.POPL (Jan. 2023), pp. 1–32. ISSN: 2475-1421. DOI: 10.1145/3571194.

[20] Azalea Raad et al. “Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic”. In: *Computer Aided Verification*. Ed. by Shuvendu K. Lahiri and Chao Wang. Cham: Springer International Publishing, 2020, pp. 225–252. ISBN: 978-3-030-53291-8. DOI: 10.1007/978-3-030-53291-8_14.

[21] Andrew Reynolds. “Conflicts, Models and Heuristics for Quantifier Instantiation in SMT”. In: *Vampire 2016. Proceedings of the 3rd Vampire Workshop*, pp. 1–15. DOI: 10.29007/jmd3.

[22] Andrew Reynolds and Jasmin Christian Blanchette. “A Decision Procedure for (Co)Datatypes in SMT Solvers”. In: *Journal of Automated Reasoning* 58.3 (Mar. 2017), pp. 341–362. ISSN: 1573-0670. DOI: 10.1007/s10817-016-9372-6.

[23] Andrew Reynolds et al. “A Decision Procedure for Separation Logic in SMT”. In: *Automated Technology for Verification and Analysis*. Ed. by Cyrille Artho, Axel Legay, and Doron Peled. Cham: Springer International Publishing, 2016, pp. 244–261. ISBN: 978-3-319-46520-3. DOI: 10.1007/978-3-319-46520-3_16.

[24] J.C. Reynolds. “Separation Logic: A Logic for Shared Mutable Data Structures”. In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. July 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817.

[25] Philipp Rümmer. “E-Matching with Free Variables”. In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by Nikolaj Bjørner and Andrei Voronkov. Berlin, Heidelberg: Springer, 2012, pp. 359–374. ISBN: 978-3-642-28717-6. DOI: 10.1007/978-3-642-28717-6_28.

[26] Amar Shah, Federico Mora, and Sanjit A. Seshia. *An Eager Satisfiability Modulo Theories Solver for Algebraic Datatypes*. Oct. 2023. DOI: 10.48550/arXiv.2310.12234. arXiv: 2310.12234 [cs].

[27] Amar Shah, Federico Mora, and Sanjit A. Seshia. “An Eager Satisfiability Modulo Theories Solver for Algebraic Datatypes”. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 38.8 (Mar. 2024), pp. 8099–8107. ISSN: 2374-3468. DOI: 10.1609/aaai.v38i8.28649.

[28] Conrad Zimmerman, Jenna DiVincenzo, and Jonathan Aldrich. “Sound Gradual Verification with Symbolic Execution”. In: *Proc. ACM Program. Lang.* 8.POPL (Jan. 2024), 85:2547–85:2576. DOI: 10.1145/3632927.

Declarations

Use of Generative AI

I acknowledge the use of OpenAI's ChatGPT-3.5 and ChatGPT-4.0 language models (available at `https://chat.openai.com/`) to assist with identifying relevant prior work and performing minor LaTeX formatting tasks during the preparation of this report. No AI-generated content has been presented as original academic work or submitted in place of my own writing, analysis, or problem-solving.

Ethical Considerations

This work did not involve human participants, personal data, or potentially harmful outputs, and posed no significant ethical risks. All tools used were publicly available and open-source, and no proprietary or sensitive systems were analysed.

Sustainability

Given the formal and implementation-focused nature of this project, sustainability was primarily achieved by minimising computational overhead. The project made use of lightweight symbolic execution frameworks and ran all experiments locally.

Availability of Data and Materials

This report is for the most part self contained. Contributions to the Gillian Platform can be accessed on GitHub (available at `https://github.com/GillianPlatform/Gillian`).

A. Formalising CSE

A.1. Assertions

We define the semantics of the assertion satisfaction relation. Most of the assertions are defined straightforwardly, but we note that for the logical expression case, we assume the heap is empty. The memory resource case depends on the resource satisfaction relation introduced in Section 4.1.3. For predicates, we have a set Preds. The elements of $(p, \vec{x}_{\text{in}}, \vec{x}_{\text{out}}, A) \in \mathsf{Preds}$, denoted $p(\vec{x}_{\text{in}}; \vec{x}_{\text{out}})\ \{A\} \in \mathsf{Preds}$, have ty $(\mathsf{Str}, \vec{\mathsf{LVar}}, \vec{\mathsf{LVar}}, \mathsf{Asrt})$, where p is the predicate name, $\vec{x}_{\text{in}}$ the input parameters, $\vec{x}_{\text{out}}$ the output parameters, and A the predicate body with $\texttt{pv}(A) = \emptyset$ and $\texttt{lv}(A) \subseteq \vec{x}_{\text{in}} \cup \vec{x}_{\text{out}}$ of the form $A = \bigvee_i (\exists \vec{x}_i.\ A_i)$ where A_i, called the predicate bodies, does not contain further quantifiers or disjunctions and $\vec{x}_i \cap \vec{x}_{\text{out}} = \emptyset$.

$$
\begin{array}{lll}
\theta, (s, \mu) \models E & \Leftrightarrow & [\![E]\!]_{\theta,s} = \mathsf{true} \land \mu = \mu_\emptyset \\
\quad\mathsf{True} & \Leftrightarrow & \mathcal{W}f(\mu) \\
\quad A_1 \Rightarrow A_2 & \Leftrightarrow & \theta, (s, \mu) \models A_1 \Rightarrow \theta, (s, \mu) \models A_2 \\
\quad A_1 \lor A_2 & \Leftrightarrow & \theta, (s, \mu) \models A_1 \lor \theta, (s, \mu) \models A_2 \\
\quad \exists x.\ A & \Leftrightarrow & \exists v \in \mathsf{Val}.\ \theta[x \mapsto v], (s, \mu) \models A \\
\quad \mathsf{emp} & \Leftrightarrow & \mu = \mu_\emptyset \\
\quad A_1 \star A_2 & \Leftrightarrow & \exists \mu_1, \mu_2.\ \mu = \mu_1 \cdot \mu_2 \land \theta, (s, \mu_1) \models A_1 \land \theta, (s, \mu_2) \models A_2 \\
\quad r(\vec{E}_1; \vec{E}_2) & \Leftrightarrow & [\![\vec{E}_1]\!]_{\theta,s} = \vec{v_1} \land [\![\vec{E}_2]\!]_{\theta,s} = \vec{v_2} \land \mu \models_{\mathsf{Res}} r(\vec{v_1}; \vec{v_2}) \\
\quad p(\vec{E}_1; \vec{E}_2) & \Leftrightarrow & \theta, (s, \mu) \models A[\vec{E}_1/\vec{x}_{\text{in}}][\vec{E}_2/\vec{x}_{\text{out}}] \text{ for } p(\vec{x}_{\text{in}}; \vec{x}_{\text{out}})\ \{A\} \in \mathsf{Preds}
\end{array}
$$

A.2. Symbolic State

The symbolic store satisfaction relation is as follows:

$$\theta, s \models_{\mathsf{Sto}} \hat{s} \iff \forall v \in \text{dom}(\hat{s}).\ [\![\hat{s}(v)]\!]_\theta = s(v)$$

Let $\hat{\mathcal{P}}$ be a multiset of symbolic predicates $\{p^1(\vec{E}^1_{\text{in}}; \vec{E}^1_{\text{out}}), \dots, p^n(\vec{E}^n_{\text{in}}; \vec{E}^n_{\text{out}})\}$. Now, the symbolic predicate satisfaction relation is as follows:

$$\theta, \mu \models_{\mathsf{Pred}} \hat{\mathcal{P}} \iff \exists \mu_1, \dots, \mu_n.\ \mu = \mu_1 \cdot \ldots \cdot \mu_n \land \forall i \in \{1, \dots, n\}.\ \theta, \mu_i \models p^i(\vec{E}^i_{\text{in}}; \vec{E}^i_{\text{out}})$$

A.3. Untypable Logical Expressions

We define the notion of untypable expressions – $\hat{T} \not\vdash E$. In order to define this, we first introduce $\hat{T} \not\vdash E : \tau$, read "$E$ *cannot be* typed as τ in $\hat{T}$". We illustrate the definition of this relation in a

few cases:

$$\frac{\hat{T} \not\vdash E : \mathsf{LVal}}{\hat{T} \not\vdash E : \tau}$$

$$\frac{x : \tau' \in \hat{T} \qquad \tau' \neq \tau \qquad \tau \neq \mathsf{LVal} \qquad \tau' \neq \mathsf{LVal}}{\hat{T} \not\vdash x : \tau}$$

$$\frac{\tau \neq \mathsf{Int}}{\hat{T} \not\vdash E_1 + E_2 : \tau}$$

$$\frac{\hat{T} \not\vdash E_1 : \mathsf{Int}}{\hat{T} \not\vdash E_1 + E_2 : \mathsf{Int}} \qquad \frac{\hat{T} \not\vdash E_2 : \mathsf{Int}}{\hat{T} \not\vdash E_1 + E_2 : \mathsf{Int}}$$

$$\frac{f(x_1 : \tau_1, \ldots, x_n : \tau_n)\{E\} \in \mathsf{Func} \qquad \hat{T} \not\vdash E_i : \tau_i \text{ for some } i \in \{1, \ldots, n\}}{\hat{T} \not\vdash f(E_1, \ldots, E_n) : \mathsf{LVal}}$$

$$\frac{f(x_1 : \tau_1, \ldots, x_n : \tau_n)\{E\} \notin \mathsf{Func}}{\hat{T} \not\vdash f(E_1, \ldots, E_n) : \mathsf{LVal}}$$

$$\frac{C(\tau_1, \ldots, \tau_n) : \tau' \in \mathsf{Constructor} \qquad \tau \neq \tau'}{\hat{T} \not\vdash C(E_1, \ldots, E_n) : \tau}$$

$$\frac{C(\tau_1, \ldots, \tau_n) : \tau' \in \mathsf{Constructor} \qquad \hat{T} \not\vdash E_i : \tau_i \text{ for some } i \in \{1, \ldots, n\}}{\hat{T} \not\vdash C(E_1, \ldots, E_n) : \tau}$$

$$\frac{\tau \neq \mathsf{Bool}}{\hat{T} \not\vdash \mathsf{is}C(E) : \tau}$$

$$\frac{C(\tau_1, \ldots, \tau_n) : \tau' \in \mathsf{Constructor} \qquad \tau \neq \tau_i}{\hat{T} \not\vdash \mathsf{get}C_i(E) : \tau}$$

$$\frac{C(\tau_1, \ldots, \tau_n) : \tau \in \mathsf{Constructor} \qquad \hat{T} \not\vdash E : \tau}{\hat{T} \not\vdash \mathsf{get}C_i(E) : \tau_i}$$

Formally, by "cannot be typed as", we mean:

$$\hat{T} \not\vdash E : \tau \implies \forall \theta : \mathsf{LVar} \to \mathsf{LVal}.\ \left(\theta \models_{\mathsf{Type}} \hat{T} \implies [\![E]\!]_\theta \notin \tau\right)$$

as stated in Lemma 1. We outline a proof for this lemma:

Proof. We proceed by structural induction over LExp. We illustrate one interesting case (function application) of the proof. Take arbitrary $E_1, \ldots, E_n \in \mathsf{LExp}$.

Inductive Hypothesis. For $i = 1, \ldots, n$, for all typing environments $\hat{T}$, and $\tau \in \mathsf{Tys}$:

(IH) $\hat{T} \not\vdash E_i : \tau \implies \forall \theta : \mathsf{LVar} \to \mathsf{LVal}.\ \left(\theta \models_{\mathsf{Type}} \hat{T} \implies [\![E_i]\!]_\theta \notin \tau\right)$

To Show. For all typing environments $\hat{T}$ and $\tau \in \mathsf{Tys}$:

$$\hat{T} \not\vdash f(E_1, \ldots, E_n) : \tau \implies \\ \forall \theta : \mathsf{LVar} \to \mathsf{LVal}.\ \left(\theta \models_{\mathsf{Type}} \hat{T} \implies [\![f(E_1, \ldots, E_n)]\!]_\theta \notin \tau\right)$$

Take arbitrary typing environment $\hat{T}$ and $\tau \in \mathsf{Tys}$. Suppose that **(A1)** $\hat{T} \not\vdash f(E_1, \ldots, E_n) : \tau$.

Take arbitrary θ and assume that (A2) $\theta \models_{\mathsf{Type}} \hat{T}$. From (A1) and the definition of $\nvdash$, we have two cases.

(C1) $f(x_1 : \tau_1, \ldots, x_n : \tau_n)\{E\} \in \mathsf{Func}$ and $\hat{T} \nvdash E_i : \tau_i$ for some $i \in \{1, \ldots, n\}$.

(1)	$[\![E_i]\!]_\theta \notin \tau_i$	from (C1), (IH), (A2)
(2)	$[\![f(E_1, \ldots, E_n)]\!]_\theta = ⭍$	from (2) and def. logical expression evaluation, $[\![\cdot]\!]_\theta$
(3)	$[\![f(E_1, \ldots, E_n)]\!]_\theta \notin \tau$	from (3)

(C2) $f(x_1 : \tau_1, \ldots, x_n : \tau_n)\{E\} \notin \mathsf{Func}$

(4)	$[\![f(E_1, \ldots, E_n)]\!]_\theta = ⭍$	from (C2) and def. logical expression evaluation, $[\![\cdot]\!]_\theta$
(5)	$[\![f(E_1, \ldots, E_n)]\!]_\theta \notin \tau$	from (5)

In both cases, we have that (7) $[\![f(E_1, \ldots, E_n)]\!]_\theta \notin \tau$. □

This then allows us to define $\hat{T} \nvdash E$ to mean that for all $\tau \in \mathsf{Tys}$, $\hat{T} \nvdash E : \tau$.

A.3.1. Proof of Lemma 2

We prove Lemma 2 which states that the evaluation of untypable expressions must result in ⭍

Proof. This follows as a result of Lemma 1. Take arbitrary typing environment $\hat{T}$ and $E \in \mathsf{LExp}$. Suppose then that $\hat{T} \nvdash E$. It must then be the case that $\hat{T} \nvdash E : \mathsf{LVal}$. Take arbitrary θ and assume that $\theta \models_{\mathsf{Type}} \hat{T}$. From Lemma 1, it follows that $[\![E]\!]_\theta \notin \mathsf{LVal}$, and so it must be that $[\![E]\!]_\theta = ⭍$. □

B. Formalising SMT-LIB

B.1. Well Sorted Terms

We define the *well-sortedness* relation as follows:

$$\frac{}{\Sigma \vdash x : \sigma}$$

$$\frac{\Sigma \vdash t_1 : \sigma_1 \quad \dots \quad \Sigma \vdash t_n : \sigma_n \qquad f : \sigma_1 \dots \sigma_n \sigma \in \Sigma}{\Sigma \vdash f\ t_1\ \dots\ t_n : \sigma}$$

For $Q \in \{\exists, \forall\}$:

$$\frac{\Sigma[x : \sigma] \vdash t : \mathsf{Bool}}{\Sigma \vdash Q(x : \sigma).\ t : \mathsf{Bool}}$$

B.2. Term Interpretations

We also define the interpretation of a term, given a Σ-structure $\mathbf{A}$ and a valuation θ.

$$\begin{array}{rcl}
[\![x]\!]_{\mathbf{A},\theta} & = & \theta(x) \\
[\![f\ t_1\ \dots\ t_n]\!]_{\mathbf{A},\theta} & = & f^{\mathbf{A}}([\![t_1]\!]_{\mathbf{A},\theta}, \dots, [\![t_n]\!]_{\mathbf{A},\theta}) \\
[\![\exists x : \sigma.\, t]\!]_{\mathbf{A},\theta} & = & \mathsf{true} \quad \text{iff} \quad [\![t]\!]_{\mathbf{A},\theta'} = \mathsf{true} \\
 & & \text{for some} \quad \theta' = \theta[x \mapsto v] \quad \text{where} \quad v \in \sigma^{\mathbf{A}} \\
[\![\forall x : \sigma.\, t]\!]_{\mathbf{A},\theta} & = & \mathsf{true} \quad \text{iff} \quad [\![t]\!]_{\mathbf{A},\theta'} = \mathsf{true} \\
 & & \text{for all} \quad \theta' = \theta[x \mapsto v] \quad \text{where} \quad v \in \sigma^{\mathbf{A}}
\end{array}$$

C. Formalising the Encoding

C.1. Proof of Lemma 6

We detail the proof of Lemma 6, which states that the expansion of the structure $\Sigma_{\mathcal{T}}$, labelled Σ, is a valid expansion. The expansion is defined in Section 4.3.1.

Proof. The proof is straightforward, and follows from the definition of the background signature.

(1) $\Sigma^S \stackrel{\text{def}}{=} \Sigma^S_{\mathcal{T}} \cup \{\sigma_{\mathsf{LVal}}\} \cup \{\sigma_\tau \mid \tau \in \mathsf{Datatype}\}$ — from the definition of Σ^S

(2) $\Sigma^S_{\mathcal{T}} \subseteq \Sigma^S$ — from (1)

Similarly, (3) follows from the definition of Σ^F:

$$\Sigma^F \stackrel{\text{def}}{=} \Sigma^F_{\mathcal{T}} \cup \left\{ \begin{array}{rll} \mathsf{bool}, & \mathsf{getBool}, & \mathsf{isBool}, \\ \mathsf{int}, & \mathsf{getInt}, & \mathsf{isInt}, \\ \mathsf{seq}, & \mathsf{getSeq}, & \mathsf{isSeq}, \\ \mathsf{datatype}_\tau, & \mathsf{getDatatype}_\tau, & \mathsf{isDatatype}_\tau \end{array} \right\}$$
$$\cup \{C, \mathsf{is}C \mid C(\tau_1, \ldots, \tau_n) : \tau \in \mathsf{Constructor}\}$$
$$\cup \{\mathsf{get}C_i \mid C(\tau_1, \ldots, \tau_n) : \tau \in \mathsf{Constructor},\ i \in \{1, \ldots, n\}\}$$
$$\cup \{f \mid f(x_1 : \tau_1, \ldots, x_n : \tau_n)\{E\} \in \mathsf{Func}\}$$

and so, (4) $\Sigma^F_{\mathcal{T}} \subseteq \Sigma^F$ follows from (3). Also:

(5) $\forall \sigma \in \Sigma^S_{\mathcal{T}}.\ \mathsf{con}_{\Sigma_{\mathcal{T}}}(\sigma) = \mathsf{con}_{\Sigma}(\sigma)$ — new constructors are associated with new sorts

(6) $\forall C \in \Sigma^C_{\mathcal{T}}.\ \mathsf{sel}_{\Sigma_{\mathcal{T}}}(C) = \mathsf{sel}_{\Sigma}(C)$ — new selectors are associated with new constructors

(7) $\forall C \in \Sigma^C_{\mathcal{T}}.\ \mathsf{tes}_{\Sigma_{\mathcal{T}}}(C) = \mathsf{tes}_{\Sigma}(C)$ — new testers are associated with new constructors

(8) $x : \sigma \in \Sigma \iff x : \sigma \in \Sigma_{\mathcal{T}}$ — by def. Σ

(9) $f : \sigma_1 \ldots \sigma_n \in \Sigma \iff f : \sigma_1 \ldots \sigma_n \in \Sigma_{\mathcal{T}}$ — by def. Σ

From (2), (4), (5), (6), (7), (8) and (9), we have that Σ is an expansion of $\Sigma_{\mathcal{T}}$ □

C.2. Definition of encTys

The function $\mathsf{encTys} : \mathsf{Tys} \to \mathsf{Sort}(\Sigma)$ maps types in the expression language to sorts in our backgound signature, where LVal represents when there is no type restriction. Its definition is

straightforward:

$$\mathsf{encTys}(\tau) = \begin{cases} \sigma_{\mathsf{Int}} & \text{if } \tau = \mathsf{Int} \\ \sigma_{\mathsf{Bool}} & \text{if } \tau = \mathsf{Bool} \\ (\sigma_{\mathsf{Seq}}\sigma_{\mathsf{LVal}}) & \text{if } \tau = \mathsf{List} \\ \sigma_{\tau} & \text{if } \tau \in \mathsf{Datatype} \\ \sigma_{\mathsf{LVal}} & \text{if } \tau = \mathsf{LVal} \end{cases}$$

C.3. Definition of $\mathsf{encLExp}_{\hat{T}}$

In this section, we present the definition of $\mathsf{encLExp}_{\hat{T}} : \mathsf{LExp} \to \mathsf{Term} \times \mathsf{Sort}(\Sigma)$, which encodes a logical expression into an SMT-LIB term, returning also the sort of the SMT-LIB term:

$$\mathsf{encLExp}_{\hat{T}}(n) = (n,\ \sigma_{\mathsf{Int}},\ \emptyset) \qquad \text{iff } n \in \mathsf{Int}$$

$$\mathsf{encLExp}_{\hat{T}}(b) = (b,\ \sigma_{\mathsf{Bool}},\ \emptyset) \qquad \text{iff } b \in \mathsf{Bool}$$

$$\mathsf{encLExp}_{\hat{T}}(x) = \begin{cases} (x,\ \mathsf{encTys}(\tau),\ \emptyset) & \text{if } x : \tau \in \hat{T} \\ (x,\ \sigma_{\mathsf{LVal}},\ \emptyset) & \text{otherwise} \end{cases} \qquad \text{iff } x \in \mathsf{LVar}$$

$$\mathsf{encLExp}_{\hat{T}}(E_1 \oplus E_2) = (\oplus\ t_1\ t_2,\ \sigma_{\mathsf{Int}},\ \Phi_1 \cup \Phi_2) \qquad \text{iff } \begin{cases} \oplus \in \{+,\ -,\ \times\} \\ (t_1, \sigma_1, \Phi_1) = \mathsf{toInt}(\mathsf{encLExp}_{\hat{T}}(E_1)) \\ (t_2, \sigma_2, \Phi_2) = \mathsf{toInt}(\mathsf{encLExp}_{\hat{T}}(E_2)) \end{cases}$$

$$\mathsf{encLExp}_{\hat{T}}(E_1 / E_2) = (/\ t_1\ t_2,\ \sigma_{\mathsf{Int}},\ \Phi) \qquad \text{iff } \begin{cases} (t_1, \sigma_1, \Phi_1) = \mathsf{toInt}(\mathsf{encLExp}_{\hat{T}}(E_1)) \\ (t_2, \sigma_2, \Phi_2) = \mathsf{toInt}(\mathsf{encLExp}_{\hat{T}}(E_2)) \\ \Phi = \Phi_1 \cup \Phi_2 \cup \{(\neq\ t_2\ 0)\} \end{cases}$$

$$\mathsf{encLExp}_{\hat{T}}(E_1 = E_2) = (=\ t_1\ t_2,\ \sigma_{\mathsf{Bool}},\ \Phi_1 \cup \Phi_2) \qquad \text{iff } \begin{cases} (t_1, \sigma_1, \Phi_1) = \mathsf{encLExp}_{\hat{T}}(E_1) \\ (t_2, \sigma_2, \Phi_2) = \mathsf{encLExp}_{\hat{T}}(E_2) \end{cases}$$

$$\mathsf{encLExp}_{\hat{T}}(E_1 < E_2) = (<\ t_1\ t_2,\ \sigma_{\mathsf{Bool}},\ \Phi_1 \cup \Phi_2) \qquad \text{iff } \begin{cases} (t_1, \sigma_1, \Phi_1) = \mathsf{toInt}(\mathsf{encLExp}_{\hat{T}}(E_1)) \\ (t_2, \sigma_2, \Phi_2) = \mathsf{toInt}(\mathsf{encLExp}_{\hat{T}}(E_2)) \end{cases}$$

$$\mathsf{encLExp}_{\hat{T}}(\neg E) = (\neg\ t,\ \sigma_{\mathsf{Bool}},\ \Phi) \qquad \text{iff } (t, \sigma, \Phi) = \mathsf{toBool}(\mathsf{encLExp}_{\hat{T}}(E))$$

$$\mathsf{encLExp}_{\hat{T}}(E_1 \wedge E_2) = (\wedge\ t_1\ t_2,\ \sigma_{\mathsf{Bool}},\ \Phi_1 \cup \Phi_2) \qquad \text{iff } \begin{cases} (t_1, \sigma_1, \Phi_1) = \mathsf{toBool}(\mathsf{encLExp}_{\hat{T}}(E_1)) \\ (t_2, \sigma_2, \Phi_2) = \mathsf{toBool}(\mathsf{encLExp}_{\hat{T}}(E_2)) \end{cases}$$

$$\mathsf{encLExp}_{\hat{T}}(E_1 : E_2) = (t,\ (\sigma_{\mathsf{Seq}}\sigma_{\mathsf{LVal}}),\ \Phi_1 \cup \Phi_2) \qquad \text{iff } \begin{cases} (t_1, \sigma_1, \Phi_1) = \mathsf{toVal}(\mathsf{encLExp}_{\hat{T}}(E_1)) \\ (t_2, \sigma_2, \Phi_2) = \mathsf{toSeq}(\mathsf{encLExp}_{\hat{T}}(E_2)) \\ t = \mathsf{seq}.\!+\!\!+\ (\mathsf{seq.unit}\ t_1)\ t_2 \end{cases}$$

$$\mathsf{encLExp}_{\hat{T}}(E_1[E_2]) = (\mathsf{seq.nth}\ \ t_1\ t_2,\ \sigma_{\mathsf{LVal}},\ \Phi) \qquad \text{iff } \begin{cases} (t_1, \sigma_1, \Phi_1) = \mathsf{toSeq}(\mathsf{encLExp}_{\hat{T}}(E_1)) \\ (t_2, \sigma_2, \Phi_2) = \mathsf{toInt}(\mathsf{encLExp}_{\hat{T}}(E_2)) \\ \Phi = \Phi_1 \cup \Phi_2 \cup \{\leq\ \ 0\ t_2\} \cup \{<\ \ t_2\ (\mathsf{seq.len}\ t_1)\} \end{cases}$$

$$\mathsf{encLExp}_{\hat{T}}(C(E_1,\ \ldots,\ E_n)) = (C\ t_1\ \ldots\ t_n,\ \sigma_{\tau},\ \Phi) \qquad \text{iff } \begin{cases} (t_1, \sigma_1, \Phi_1) = \mathsf{toSort}_{\tau_1}(\mathsf{encLExp}_{\hat{T}}(E_1)) \\ \vdots \\ (t_n, \sigma_n, \Phi_n) = \mathsf{toSort}_{\tau_n}(\mathsf{encLExp}_{\hat{T}}(E_n)) \\ C(\tau_1, \ldots, \tau_n) : \tau \in \mathsf{Constructor} \\ \Phi = \bigcup_{i=1}^{n} \Phi_i \end{cases}$$

$$\mathsf{encLExp}_{\hat{T}}(\mathsf{get}C_i(E)) = (\mathsf{get}C_i\ t,\ \ \sigma,\ \ \Phi \cup \{\mathsf{is}C\ t\}) \quad \text{iff} \begin{cases} 1 \le i \le n \\ C(\tau_1,\ \ldots,\ \tau_n) : \tau \in \mathsf{Constructor} \\ (t, \sigma, \Phi) = \mathsf{toDatatype}_\tau(\mathsf{encLExp}_{\hat{T}}(E)) \\ \sigma = \mathsf{encTys}(\tau_i) \end{cases}$$

$$\mathsf{encLExp}_{\hat{T}}(\mathsf{is}C(E)) = (\mathsf{is}C\ t,\ \ \sigma_{\mathsf{Bool}},\ \ \Phi) \quad \text{iff} \begin{cases} C(\tau_1,\ \ldots,\ \tau_n) : \tau \in \mathsf{Constructor} \\ (t', \sigma', \Phi') = \mathsf{encLExp}_{\hat{T}}(E) \\ \sigma' \in \{\sigma_\tau, \sigma_{\mathsf{LVal}}\} \\ (t, \sigma, \Phi) = \mathsf{toDatatype}_\tau(t', \sigma', \Phi') \end{cases}$$

$$\mathsf{encLExp}_{\hat{T}}(\mathsf{is}C(E)) = (\mathsf{false},\ \ \sigma_{\mathsf{Bool}},\ \ \emptyset) \quad \text{iff} \begin{cases} C(\tau_1,\ \ldots,\ \tau_n) : \tau \in \mathsf{Constructor} \\ (t, \sigma, \Phi) = \mathsf{encLExp}_{\hat{T}}(E) \\ \sigma \notin \{\sigma_\tau, \sigma_{\mathsf{LVal}}\} \end{cases}$$

$$\mathsf{encLExp}_{\hat{T}}(f(E_1,\ \ldots,\ E_n)) = (t,\ \ \sigma_{\mathsf{LVal}},\ \ \Phi) \quad \text{iff} \begin{cases} (t_1, \sigma_1, \Phi_1) = \mathsf{toSort}_{\tau_1}(\mathsf{encLExp}_{\hat{T}}(E_1)) \\ \vdots \\ (t_n, \sigma_n, \Phi_n) = \mathsf{toSort}_{\tau_n}(\mathsf{encLExp}_{\hat{T}}(E_n)) \\ f(x_1 : \tau_1,\ \ldots,\ x_n : \tau_n)\{E\} \in \mathsf{Func} \\ t = f\ t_1\ \ldots\ t_n \\ \Phi = \left(\bigcup_{i=1}^n \Phi_i\right) \cup \{(\neq\ \ t\ \lightning)\} \end{cases}$$

$$\mathsf{encLExp}_{\hat{T}}(E\ ?\ E_1\ :\ E_2) = (\mathsf{ite}\ t\ t_1\ t_2,\ \ \sigma_{\mathsf{LVal}},\ \ \Phi) \quad \text{iff} \begin{cases} (t, \sigma, \Phi') = \mathsf{toBool}(\mathsf{encLExp}_{\hat{T}}(E)) \\ (t_1, \sigma_1, \Phi_1) = \mathsf{toVal}(\mathsf{encLExp}_{\hat{T}}(E_1)) \\ (t_2, \sigma_2, \Phi_2) = \mathsf{toVal}(\mathsf{encLExp}_{\hat{T}}(E_2)) \\ \Phi = \Phi' \cup \Phi_1 \cup \Phi_2 \end{cases}$$

$$\mathsf{encLExp}_{\hat{T}}(E \in \tau) = (\mathsf{true},\ \ \sigma_{\mathsf{Bool}},\ \ \Phi) \quad \text{iff} \begin{cases} (t, \sigma, \Phi) = \mathsf{encLExp}_{\hat{T}}(E) \\ \sigma = \mathsf{encTys}(\tau) \end{cases}$$

$$\mathsf{encLExp}_{\hat{T}}(E \in \tau) = (\mathsf{false},\ \ \sigma_{\mathsf{Bool}},\ \ \emptyset) \quad \text{iff} \begin{cases} (t, \sigma, \Phi) = \mathsf{encLExp}_{\hat{T}}(E) \\ \sigma \neq \mathsf{encTys}(\tau) \\ \sigma \neq \sigma_{\mathsf{LVal}} \\ \tau \neq \mathsf{LVal} \end{cases}$$

$$\mathsf{encLExp}_{\hat{T}}(E \in \tau) = \mathsf{isSort}_\tau(t, \sigma, \Phi) \quad \text{iff} \begin{cases} (t, \sigma, \Phi) = \mathsf{encLExp}_{\hat{T}}(E) \\ \sigma \neq \mathsf{encTys}(\tau) \\ \sigma = \sigma_{\mathsf{LVal}} \\ \tau \neq \mathsf{LVal} \end{cases}$$

with auxilliary functions defined as follows:

$$\mathsf{toInt}(t, \sigma, \Phi) = \begin{cases} (t,\ \ \sigma_{\mathsf{Int}},\ \ \Phi) & \text{iff } \sigma = \sigma_{\mathsf{Int}} \\ (\mathsf{getInt}\ t,\ \ \sigma_{\mathsf{Int}},\ \ \Phi \cup \{(\mathsf{isInt}\ t)\}) & \text{iff } \sigma = \sigma_{\mathsf{LVal}} \end{cases}$$

$$\mathsf{toBool}(t, \sigma) = \begin{cases} (t,\ \ \sigma_{\mathsf{Bool}},\ \ \Phi) & \text{iff } \sigma = \sigma_{\mathsf{Bool}} \\ (\mathsf{getBool}\ t,\ \ \sigma_{\mathsf{Bool}},\ \ \Phi \cup \{(\mathsf{isBool}\ t)\}) & \text{iff } \sigma = \sigma_{\mathsf{LVal}} \end{cases}$$

$$\mathsf{toSeq}(t,\sigma) = \begin{cases} (t,\ \ (\sigma_{\mathsf{Seq}}\sigma_{\mathsf{LVal}}),\ \ \Phi) & \text{iff } \sigma = (\sigma_{\mathsf{Seq}}\sigma_{\mathsf{LVal}}) \\ (\mathsf{getSeq}\ t,\ \ (\sigma_{\mathsf{Seq}}\sigma_{\mathsf{LVal}}),\ \ \Phi \cup \{(\mathsf{isSeq}\ t)\}) & \text{iff } \sigma = \sigma_{\mathsf{LVal}} \end{cases}$$

$$\mathsf{toDatatype}_\tau(t,\sigma) = \begin{cases} (t,\ \ \sigma_\tau,\ \ \Phi) & \text{iff } \sigma = (\sigma_{\mathsf{Seq}}\sigma_{\mathsf{LVal}}) \\ (\mathsf{getDatatype}_\tau\ t,\ \ \sigma_\tau,\ \ \Phi \cup \{(\mathsf{isDatatype}_\tau\ t)\}) & \text{iff } \sigma = \sigma_{\mathsf{LVal}} \end{cases}$$

$$\mathsf{toVal}(t,\sigma,\Phi) = \begin{cases} (t,\ \ \sigma_{\mathsf{LVal}},\ \ \Phi) & \text{iff } \sigma = \sigma_{\mathsf{LVal}} \\ (\mathsf{bool}\ t,\ \ ,\sigma_{\mathsf{LVal}},\ \ \Phi) & \text{iff } \sigma = \sigma_{\mathsf{Bool}} \\ (\mathsf{int}\ t,\ \ ,\sigma_{\mathsf{LVal}},\ \ \Phi) & \text{iff } \sigma = \sigma_{\mathsf{Int}} \\ (\mathsf{seq}\ t,\ \ ,\sigma_{\mathsf{LVal}},\ \ \Phi) & \text{iff } \sigma = (\sigma_{\mathsf{Seq}}\sigma_{\mathsf{LVal}}) \\ (\mathsf{datatype}_\tau\ t,\ \ \sigma_{\mathsf{LVal}},\ \ \Phi) & \text{iff } \sigma = \sigma_\tau \end{cases}$$

$$\mathsf{toSort}_\tau(t,\sigma,\Phi) = \begin{cases} \mathsf{toInt}(t,\sigma,\Phi) & \text{iff } \tau = \mathsf{Int} \\ \mathsf{toBool}(t,\sigma,\Phi) & \text{iff } \tau = \mathsf{Bool} \\ \mathsf{toSeq}(t,\sigma,\Phi) & \text{iff } \tau = \mathsf{List} \\ \mathsf{toDatatype}_\tau(t,\sigma,\Phi) & \text{iff } \tau \in \mathsf{Datatype} \\ \mathsf{toVal}(t,\sigma,\Phi) & \text{iff } \tau = \mathsf{LVal} \end{cases}$$

$$\mathsf{isSort}_\tau(t,\sigma,\Phi) = \begin{cases} (\mathsf{isBool}\ t,\ \ \sigma_{\mathsf{Bool}},\ \ \Phi) & \text{iff } \tau = \mathsf{Bool} \\ (\mathsf{isInt}\ t,\ \ \sigma_{\mathsf{Bool}},\ \ \Phi) & \text{iff } \tau = \mathsf{Int} \\ (\mathsf{isSeq}\ t,\ \ \sigma_{\mathsf{Bool}},\ \ \Phi) & \text{iff } \tau = \mathsf{List} \\ (\mathsf{isDatatype}_\tau\ t,\ \ \sigma_{\mathsf{Bool}},\ \ \Phi) & \text{iff } \tau \in \mathsf{Datatype} \\ (\mathsf{true},\ \ \sigma_{\mathsf{Bool}},\ \ \Phi) & \text{iff } \tau = \mathsf{LVal} \end{cases}$$

which simply unwrap any terms which are wrapped in the σ_{LVal} ADT, or vice versa.

C.4. Proof of Lemma 7

We prove Lemma 7, which states that if $\hat{T} \vdash E$, then $\mathsf{encLExp}_{\hat{T}}(E)$ is defined. To prove this, we first show the following lemma to hold:

Lemma 17. *For all $E \in \mathsf{LExp}$, for all typing environments $\hat{T}$ and $\tau \in \mathsf{Tys}$:*

$$\hat{T} \vdash E : \tau \implies \exists t, \sigma, \Phi.\ \big(\mathsf{encLExp}_{\hat{T}}(E) = (t,\sigma,\Phi) \land \sigma \in \{\mathsf{encTys}(\tau),\ \sigma_{\mathsf{LVal}}\}\big)$$

Proof. We show Lemma 17 by structural induction over LExp. We illustrate one interesting case (function application) of the inductive proof. Take arbitrary $E_1, \ldots, E_n \in \mathsf{LExp}$.

Inductive Hypothesis. For $i = 1, \ldots, n$, for all $\hat{T}$ and $\tau \in \mathsf{Tys}$:

$$\textbf{(IH)}\quad \hat{T} \vdash E_i : \tau \implies \exists t, \sigma, \Phi.\ \big(\mathsf{encLExp}_{\hat{T}}(E_i) = (t,\sigma,\Phi) \land \sigma \in \{\mathsf{encTys}(\tau),\ \sigma_{\mathsf{LVal}}\}\big)$$

To Show. For all $\hat{T}$ and $\tau \in \mathsf{Tys}$:

$$\hat{T} \vdash f(E_1, \ldots, E_n) : \tau \implies \exists t, \sigma, \Phi.\ \big(\mathsf{encLExp}_{\hat{T}}(f(E_1, \ldots, E_n)) = (t, \sigma, \Phi) \land \sigma \in \{\mathsf{encTys}(\tau), \sigma_{\mathsf{LVal}}\}\big)$$

Take arbitrary $\hat{T}$ and $\tau \in \mathsf{Tys}$. Suppose that **(A1)** $\hat{T} \vdash f(E_1, \ldots, E_n) : \tau$. Then:

(1) $\neg\hat{T} \nvdash f(E_1, \ldots, E_n) : \tau$ (def. $\vdash$) (A1)
(2) $f(x_1 : \tau_1, \ldots, x_n : \tau_n)\{E\} \in \mathsf{Func}$ (def. $\nvdash$) (1)
(3) $\hat{T} \vdash E_i : \tau_i$ for all $i \in \{1, \ldots, n\}$ (def. $\nvdash$) (1)
(4) $\mathsf{encLExp}_{\hat{T}}(E_i) = (t_i, \sigma_i, \Phi_i) \land \sigma_i \in \{\mathsf{encTys}(\tau_i), \sigma_{\mathsf{LVal}}\}$ for $i = 1, \ldots, n$ (3) (IH)
(5) $\exists t_i, \sigma_i, \Phi_i.\ \mathsf{toSort}_\tau(\mathsf{encLExp}_{\hat{T}}(E_i)) = (t_i, \sigma_i, \Phi_i)$ for $i = 1, \ldots, n$ (4) (Lemma 23)

Therefore, from (5) and the definition of $\mathsf{encLExp}_{\hat{T}}$ we have that

$$\exists t, \sigma, \Phi.\ \big(\mathsf{encLExp}_{\hat{T}}(f(E_1, \ldots, E_n)) = (t, \sigma, \Phi) \land \sigma \in \{\mathsf{encTys}(\tau), \sigma_{\mathsf{LVal}}\}\big)$$

as required. □

Lemma 7 follows from Lemma 17:

Proof. Take arbitrary $E \in \mathsf{LExp}$, and $\hat{T}$. Suppose that $\hat{T} \vdash E$. It then follows that $\neg\hat{T} \nvdash E$, and so $\neg\forall\tau \in \mathsf{Tys}.\ \hat{T} \nvdash E : \tau$. But then $\exists\tau \in \mathsf{Tys}.\ \hat{T} \vdash E : \tau$. By applying Lemma 17, we have that $\exists t, \sigma, \Phi.\ \mathsf{encLExp}_{\hat{T}}(E) = (t, \sigma, \Phi)$, as required. □

D. Bridging CSE and SMT-LIB

D.1. $\Sigma_{\hat{T}}$-structures and LVal

We detail here the total mapping $[\![\cdot]\!]_{\mathbf{A}}$ introduced in Section 4.4.2. Let Σ be the background signature introduced in Section 4.3.1. Let Σ' be an arbitrary variant of Σ. We have that every $\Sigma_{\hat{T}}$-structure $\mathbf{A}$ has a universe A defined by:

$$A = \sigma_{\mathsf{Int}}^{\mathbf{A}} \cup \sigma_{\mathsf{Bool}}^{\mathbf{A}} \cup \sigma_{\mathsf{LVal}}^{\mathbf{A}} \cup \bigcup\{\sigma_{\tau}^{\mathbf{A}} \mid \tau \in \mathsf{Datatype}\} \cup \bigcup\{(\sigma_{\mathsf{Seq}}\sigma)^{\mathbf{A}} \mid \sigma \in \mathsf{Sort}(\Sigma)\}$$

where $\Sigma^S = \{\sigma_{\mathsf{Int}}, \sigma_{\mathsf{Bool}}, \sigma_{\mathsf{LVal}}, \sigma_{\mathsf{Seq}}\} \cup \{\sigma_\tau \mid \tau \in \mathsf{Datatype}\}$. We also note that $\mathsf{Int} = \sigma_{\mathsf{Int}}^{\mathbf{A}}$, $\mathsf{Bool} = \sigma_{\mathsf{Bool}}^{\mathbf{A}}$, and $\tau = \sigma_\tau^{\mathbf{A}}$ for $\tau \in \mathsf{Datatype}$. We thus define $[\![\cdot]\!]_{\mathbf{A}} : A \rightarrow \mathsf{LVal}$ as follows:

$$\begin{array}{ll}
[\![v]\!]_{\mathbf{A}} = n & \text{iff } \exists n \in \sigma_{\mathsf{Int}}^{\mathbf{A}}.\ \mathsf{int}^{\mathbf{A}}(n) = v \\
[\![v]\!]_{\mathbf{A}} = b & \text{iff } \exists b \in \sigma_{\mathsf{Bool}}^{\mathbf{A}}.\ \mathsf{bool}^{\mathbf{A}}(b) = v \\
[\![v]\!]_{\mathbf{A}} = d & \text{iff } \exists d \in \sigma_{\tau}^{\mathbf{A}}.\ \mathsf{datatype}_{\tau}^{\mathbf{A}}(d) = v \\
[\![v]\!]_{\mathbf{A}} = [\![\vec{v}]\!]_{\mathbf{A}} & \text{iff } \exists \vec{v} \in (\sigma_{\mathsf{Seq}}\sigma_{\mathsf{LVal}})^{\mathbf{A}}.\ \mathsf{seq}^{\mathbf{A}}(\vec{v}) = v \\
[\![\vec{v}]\!]_{\mathbf{A}} = [\![u]\!]_{\mathbf{A}} : [\![\vec{u}]\!]_{\mathbf{A}} & \text{iff } \begin{array}{c}\exists \sigma \in \mathsf{Sort}(\Sigma).\, \exists u \in \sigma^{\mathbf{A}},\ \vec{u} \in (\sigma_{\mathsf{Seq}}\sigma)^{\mathbf{A}}. \\ \vec{v} = \mathsf{seq}.\!+\!\!+^{\mathbf{A}}(\mathsf{seq.unit}^{\mathbf{A}}(u),\ \vec{u})\end{array} \\
[\![v]\!]_{\mathbf{A}} = v & \text{otherwise}
\end{array}$$

We observe that this is a well-defined function. Clearly it is defined for all $v \in A$. Secondly, more than one case of the definition cannot apply, since the ranges of constructors are disjoint, and the penultimate case only applies for v with sequence sorts. Lastly, it can be shown inductively that the codomain of the function is indeed LVal.

D.2. Logical Variable Substitutions and Valuations

Suppose $\Sigma_{\hat{T}}$ is an aribtrary variant of Σ, and $\mathbf{A}$ is a $\Sigma_{\hat{T}}$-structure with universe A. We detail here the partial mapping $[\![\cdot]\!]_{\Sigma_{\hat{T}},\mathbf{A}}$ introduced in Section 4.4.3. Given a $\theta : \mathsf{LVar} \rightarrow \mathsf{LVal}$, we define $[\![\theta]\!]_{\Sigma_{\hat{T}},\mathbf{A}} = \theta'$ where $\theta' : \mathsf{LVar} \rightarrow A$ and:

$$\theta'(x) = \begin{cases}
\mathsf{wrap}(\theta(x)) & \text{if } x : \sigma_{\mathsf{LVal}} \in \Sigma_{\hat{T}} \\
\theta(x) & \text{if } \begin{array}{c}\theta(x) \in \tau \wedge \tau \in \{\mathsf{Int}, \mathsf{Bool}\} \cup \mathsf{Datatype} \wedge \\ \mathsf{encTys}(\tau) = \sigma \wedge x : \sigma \in \Sigma_{\hat{T}}\end{array} \\
\vec{\mathsf{wrap}}(\theta(x)) & \text{if } \theta(x) \in \mathsf{List} \wedge x : (\sigma_{\mathsf{Seq}}\sigma_{\mathsf{LVal}}) \in \Sigma_{\hat{T}} \\
v & \text{otherwise, where } x : \sigma \in \Sigma_{\hat{T}} \wedge v \in \sigma^{\mathbf{A}}
\end{cases}$$

We define an auxilliary function wrap taking a primitive value and wrapping it in the σ_{LVal} ADT:

$$\mathsf{wrap}(v) = \begin{cases} \mathsf{bool}^{\mathbf{A}}(v) & \text{iff } v \in \mathsf{Bool} \\ \mathsf{int}^{\mathbf{A}}(v) & \text{iff } v \in \mathsf{Int} \\ \mathsf{datatype}^{\mathbf{A}}_{\tau}(v) & \text{iff } v \in \tau \land \tau \in \mathsf{Datatype} \\ \mathsf{seq}^{\mathbf{A}}(\vec{\mathsf{wrap}}(v)) & \text{iff } v \in \mathsf{List} \end{cases}$$

We similarly define $\vec{\mathsf{wrap}}$, taking a primitive list and converting it to a value with sort $(\sigma_{\mathsf{Seq}}\sigma_{\mathsf{LVal}})$:

$$\vec{\mathsf{wrap}}(v) = \begin{cases} \mathsf{seq}.\mathbin{+\!\!+}^{\mathbf{A}}(\mathsf{seq.unit}^{\mathbf{A}}(\mathsf{wrap}(u)),\ \vec{\mathsf{wrap}}(\vec{u})) & \text{iff } v \in \mathsf{List} \land v = u : \vec{u} \\ \mathsf{seq.empty}^{\mathbf{A}} & \text{iff } v \in \mathsf{List} \land v = [\,] \end{cases}$$

We note one useful property about the relationship between $[\![\cdot]\!]_{\Sigma_{\hat{T}},\mathbf{A}}$ and $[\![\cdot]\!]_{\mathbf{A}}$:

Lemma 18. *Suppose that*

$$\theta(x) = v$$
$$[\![v']\!]_{\mathbf{A}} = v$$
$$v' \in \sigma^{\mathbf{A}}$$
$$x : \sigma \in \Sigma_{\hat{T}}$$
$$\theta' = [\![\theta]\!]_{\Sigma_{\hat{T}},\mathbf{A}}$$

then:

$$\theta'(x) = v'$$

Proof. This can be shown trivially by considering each case of σ. □

E. Correctness

E.1. Proof of Lemma 11

Proof. We prove Lemma 11 by structural induction over LExp. We detail here the cases of encoding logical variables, and function application.

Base Case – $x \in \mathsf{LVar}$

Take arbitrary $x \in \mathsf{LVar}$.

To Show. If $(t, \sigma, \Phi) = \mathsf{encLExp}_{\hat{T}}(x)$, then:

$$\Sigma_{\hat{T}} \vdash t : \sigma \ \wedge \ \Sigma_{\hat{T}} \vdash \Phi : \sigma_{\mathsf{Bool}}$$

Assume that **(A1)** $\mathsf{encLExp}_{\hat{T}}(x) = (t, \sigma, \Phi)$. We have two cases:

(C1) $x : \tau \in \hat{T}$

(1)	$t = x$	(A1) (C1) (def. $\mathsf{encLExp}_{\hat{T}}$)
(2)	$\sigma = \mathsf{encTys}(\tau)$	(A1) (C1) (def. $\mathsf{encLExp}_{\hat{T}}$)
(3)	$\Phi = \emptyset$	(A1) (C1) (def. $\mathsf{encLExp}_{\hat{T}}$)
(4)	$\Sigma_{\hat{T}} \vdash \Phi : \sigma_{\mathsf{Bool}}$	(3) (def. $\vdash$)
(5)	$x : \mathsf{encTys}(\tau) \in \Sigma_{\hat{T}}$	(C1) (def. $\Sigma_{\hat{T}}$)
(6)	$\Sigma_{\hat{T}} \vdash t : \sigma$	(1) (2) (5) (def. $\vdash$)
(7)	$\Sigma_{\hat{T}} \vdash t : \sigma \wedge \Sigma_{\hat{T}} \vdash \Phi : \sigma_{\mathsf{Bool}}$	(4) (6)

(C2) $x \notin \mathsf{dom}(\hat{T})$

(1)	$t = x$	(A1) (C2) (def. $\mathsf{encLExp}_{\hat{T}}$)
(2)	$\sigma = \sigma_{\mathsf{LVal}}$	(A1) (C2) (def. $\mathsf{encLExp}_{\hat{T}}$)
(3)	$\Phi = \emptyset$	(A1) (C2) (def. $\mathsf{encLExp}_{\hat{T}}$)
(4)	$\Sigma_{\hat{T}} \vdash \Phi : \sigma_{\mathsf{Bool}}$	(3) (def. $\vdash$)
(5)	$x : \sigma_{\mathsf{LVal}} \in \Sigma_{\hat{T}}$	(C2) (def. $\Sigma_{\hat{T}}$)
(6)	$\Sigma_{\hat{T}} \vdash t : \sigma$	(1) (2) (5) (def. $\vdash$)
(7)	$\Sigma_{\hat{T}} \vdash t : \sigma \wedge \Sigma_{\hat{T}} \vdash \Phi : \sigma_{\mathsf{Bool}}$	(4) (6)

Inductive Case – $f(\vec{E})$

Take arbitrary $E_i \in \mathsf{LExp}$ for $i = 1, \ldots, n$.

Inductive Hypothesis. For $i = 1, \ldots, n$:

(IH) $\mathsf{encLExp}_{\hat{T}}(E_i) = (t_i, \sigma_i, \Phi_i) \implies \Sigma_{\hat{T}} \vdash t_i : \sigma_i \ \wedge \ \Sigma_{\hat{T}} \vdash \Phi_i : \sigma_{\mathsf{Bool}}$

To Show. If $\mathsf{encLExp}_{\hat{T}}(f(E_1, \ldots, E_n)) = (t, \sigma, \Phi)$, then:

$$\Sigma_{\hat{T}} \vdash t : \sigma \ \wedge \ \Sigma_{\hat{T}} \vdash \Phi : \sigma_{\mathsf{Bool}}$$

Suppose that **(A1)** $\mathsf{encLExp}_{\hat{T}}(f(E_1, \ldots, E_n)) = (t, \sigma, \Phi)$. Let $i \in \{1, \ldots, n\}$ be arbitrary. Then:

(1)	$f(x_1 : \tau_1, \ldots, x_n : \tau_n)\{E\} \in \mathsf{Func}$	(A1) (def. $\mathsf{encLExp}_{\hat{T}}$)
(2)	$(t_i, \sigma_i, \Phi_i) = \mathsf{toSort}_{\tau_i}(\mathsf{encLExp}_{\hat{T}}(E_i))$ for some (t_i, σ_i, Φ_i)	(A1) (def. $\mathsf{encLExp}_{\hat{T}}$)
(3)	$(t'_i, \sigma'_i, \Phi'_i) = \mathsf{encLExp}_{\hat{T}}(E_i)$ for some (t_i, σ_i, Φ_i)	(2) (def. toSort_{τ_i})
(4)	$\Sigma_{\hat{T}} \vdash t'_i : \sigma'_i$	(3) (IH)
(5)	$\Sigma_{\hat{T}} \vdash \Phi'_i : \sigma_{\mathsf{Bool}}$	(3) (IH)
(6)	$\sigma'_i \in \{\mathsf{encTys}(\tau_i), \sigma_{\mathsf{LVal}}\}$	(2) (3) (Lemma 23)
(7)	$\Sigma_{\hat{T}} \vdash t_i : \sigma_i$	(2) (3) (4) (5) (6) (Lemma 20)
(8)	$\sigma_i = \mathsf{encTys}(\tau_i)$	(2) (3) (4) (5) (6) (Lemma 20)
(9)	$\Sigma_{\hat{T}} \vdash \Phi_i : \sigma_{\mathsf{Bool}}$	(2) (3) (4) (5) (6) (Lemma 20)

Since i was arbitrary, **(10)** $\Sigma_{\hat{T}} \vdash t_i : \sigma_i$ and $\sigma_i = \mathsf{encTys}(\tau_i)$ and $\Sigma_{\hat{T}} \vdash \Phi_i : \sigma_{\mathsf{Bool}}$ where $(t_i, \sigma_i, \Phi_i) = \mathsf{toSort}_{\tau_i}(\mathsf{encLExp}_{\hat{T}}(E_i))$ for all $i \in \{1, \ldots, n\}$. Therefore:

(11)	$f : \sigma_1 \ldots \sigma_n \sigma_{\mathsf{LVal}} \in \Sigma_{\hat{T}}$ where $\sigma_i = \mathsf{encTys}(\tau_i)$ for $i = 1, \ldots, n$	(1) (def. $\Sigma_{\hat{T}}$)
(12)	$\Sigma_{\hat{T}} \vdash (f \ t_1 \ \ldots \ t_n) : \sigma_{\mathsf{LVal}}$	(10) (11)
(13)	$\Sigma_{\hat{T}} \vdash \bigcup_{i=1}^{n} \Phi_i : \sigma_{\mathsf{Bool}}$	(10)

It follows from (10), (12), (13), (A1) and the definition of $\mathsf{encLExp}_{\hat{T}}$ that

$$\Sigma_{\hat{T}} \vdash t : \sigma \wedge \Sigma_{\hat{T}} \vdash \Phi : \sigma_{\mathsf{Bool}}$$

□

E.2. Proof of Lemma 14

Proof. We proceed by functional induction over the definition of logical expression evaluation. Functional induction allows us to prove statements of the form:

$$\forall E, \theta, v. \ [\![E]\!]_\theta = v \implies P(E, \theta, v)$$

To prove Lemma 14, we define $P(E, \theta, v)$ to mean the following:

Assume:

$$(t, \sigma, \Phi) = \mathsf{encLExp}_{\hat{T}}(E)$$
$$\mathbf{A} \in \mathcal{M}(\mathcal{T}^{\Sigma_{\hat{T}}})$$
$$\mathbf{A} \models_{\texttt{SMT}} \Psi$$
$$[\![\theta]\!]_{\Sigma_{\hat{T}},\mathbf{A}} = \theta'$$

Then:

$$v \neq \lightning \;\wedge\; \theta \models_{\mathsf{Type}} \hat{T}$$
$$\Longrightarrow$$
$$\theta', \mathbf{A} \models_{\texttt{SMT}} \Phi \;\wedge\; \exists v'.\; [\![t]\!]_{\theta',\mathbf{A}} = v' \;\wedge\; [\![v']\!]_{\mathbf{A}} = v$$

and conversely:

$$\theta', \mathbf{A} \models_{\texttt{SMT}} \Phi \;\wedge\; [\![t]\!]_{\theta',\mathbf{A}} = v'$$
$$\Longrightarrow$$
$$[\![v']\!]_{\mathbf{A}} = v \;\wedge\; v \neq \lightning \;\wedge\; \theta \models_{\mathsf{Type}} \hat{T}$$

We detail one interesting case of the inductive proof, namely that of evaluating function application.

Inductive Hypothesis. Suppose that

(IH1) $f(x_1 : \tau_1, \ldots, x_n : \tau_n)\{E\} \in \mathsf{Func}$

(IH2) For $i = 1, \ldots, n$:

(IH2a) $[\![E_i]\!]_\theta = v_i$ **(IH2b)** $P(E_i, \theta, v)$ **(IH2c)** $v_i \in \tau_i$

(IH3) $[\![E]\!]_{\theta_f} = v$

(IH4) $P(E, \theta_f, v)$

(IH5) $\theta_f = \theta[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$

To Show. $P(f(E_1, \ldots, E_n), \theta, v)$

First assume that:

(A1) $(t, \sigma, \Phi) = \mathsf{encLExp}_{\hat{T}}(f(E_1, \ldots, E_n))$ **(A2)** $\mathbf{A} \in \mathcal{M}(\mathcal{T}^{\Sigma_{\hat{T}}})$

(A3) $\mathbf{A} \models_{\texttt{SMT}} \Psi$ **(A4)** $[\![\theta]\!]_{\Sigma_{\hat{T}},\mathbf{A}} = \theta'$

Assume also that **(A5)** $v \neq \lightning$ and **(A6)** $\theta \models_{\mathsf{Type}} \hat{T}$. We aim to show that

$$\theta', \mathbf{A} \models_{\texttt{SMT}} \Phi \;\wedge\; \exists v'.\; [\![t]\!]_{\theta',\mathbf{A}} = v' \;\wedge\; [\![v']\!]_{\mathbf{A}} = v$$

Take arbitrary $i \in \{1, \ldots, n\}$. Then:

(1)	$(t_i, \sigma_i, \Phi_i) = \mathsf{toSort}_{\tau_i}(\mathsf{encLExp}_{\hat{T}}(E_i))$ for some (t_i, σ_i, Φ_i)	(A1) (def. $\mathsf{encLExp}_{\hat{T}}$)
(2)	$(t'_i, \sigma'_i, \Phi'_i) = \mathsf{encLExp}_{\hat{T}}(E_i)$ for some $(t'_i, \sigma'_i, \Phi'_i)$	(1) (def. toSort_{τ_i})
(3)	$v_i \neq \lightning$	(IH2c)

Applying our inductive hypothesis, we get that for some v'_i:

(4) $\theta', \mathbf{A} \models_{\texttt{SMT}} \Phi'_i$ (IH2b) (2) (A2) (A3) (A4) (3) (A6)
(5) $[\![t'_i]\!]_{\theta',\mathbf{A}} = v'_i$ (IH2b) (2) (A2) (A3) (A4) (3) (A6)
(6) $[\![v'_i]\!]_{\mathbf{A}} = v_i$ (IH2b) (2) (A2) (A3) (A4) (3) (A6)
(7) $\Sigma_{\hat{T}} \vdash t'_i : \sigma'_i$ and $\Sigma_{\hat{T}} \vdash \Phi'_i : \sigma_{\mathsf{Bool}}$ (2) (Lemma 11)
(8) $[\![t'_i]\!]_{\theta',\mathbf{A}} \in \sigma_i'^{\mathbf{A}}$ (7) (A2) (A4) (Lemma 10) (Lemma 5)
(9) $[\![t_i]\!]_{\theta',\mathbf{A}} = v''_i$ and $[\![v''_i]\!]_{\theta',\mathbf{A}} = v_i$ (A2) (1) (2) (5) (6) (IH2c) (8) (4) (Lemma 21)
(10) $\theta', \mathbf{A} \models_{\texttt{SMT}} \Phi_i$ (A2) (1) (2) (5) (6) (IH2c) (8) (4) (Lemma 21)

We now turn our attention towards the function body. Define $\hat{T}_f$ to be such that $x_i : \tau_i \in \hat{T}_f$ for $i = 1, \ldots, n$. Also define $[\![\theta_f]\!]_{\Sigma^{\mathcal{T}_f}} = \theta'_f$.

(11) $(t'', \sigma'', \Phi'') = \mathsf{encLExp}_{\hat{T}_f}(E)$ for some (t'', σ'', Φ'') (IH1) (Property 2)
(12) $(t', \sigma', \Phi') = \mathsf{toVal}(\mathsf{encLExp}_{\hat{T}_f}(E))$ (11) (def. toVal)
(13) $\theta_f \models_{\mathsf{Type}} \hat{T}_f$ (IH5) (IH2c) (def. $\hat{T}_f$) (def. $\models_{\mathsf{Type}}$)

We can now apply the inductive hypothesis to the function body.

(14) $\theta'_f, \mathbf{A} \models_{\texttt{SMT}} \Phi''$ (11) (A2) (A3) (def. θ'_f) (A5) (13) (IH4)
(15) $[\![t'']\!]_{\theta'_f,\mathbf{A}} = v'$ (11) (A2) (A3) (def. θ'_f) (A5) (13) (IH4)
(16) $[\![v']\!]_{\mathbf{A}} = v$ (11) (A2) (A3) (def. θ'_f) (A5) (13) (IH4)
(17) $(t', \sigma', \Phi') = \mathsf{toSort}_{\mathsf{LVal}}(\mathsf{encLExp}_{\hat{T}_f}(E))$ (12) (def. $\mathsf{toSort}_{\mathsf{LVal}}$)
(18) $\Sigma_{\hat{T}_f} \vdash t'' : \sigma''$ (11) (Lemma 11)
(19) $[\![t'']\!]_{\theta'_f,\mathbf{A}} \in \sigma''^{\mathbf{A}}$ (17) (18) (A2) (def. θ'_f) (Lemma 10) (Lemma 5)
(20) $[\![t']\!]_{\theta'_f,\mathbf{A}} = v''$ and $[\![v'']\!]_{\theta'_f,\mathbf{A}} = v$ (A2) (16) (11) (15) (16) (IH2c) (19) (14) (Lemma 21)
(21) $\theta'_f, \mathbf{A} \models_{\texttt{SMT}} \Phi'$ (A2) (16) (11) (15) (16) (IH2c) (19) (14) (Lemma 21)

This allows us to use the background assertions to prove our desired result. We begin by proving some facts about θ'_f

(22) $\sigma'_i \in \{\mathsf{encTys}(\tau_i), \sigma_{\mathsf{LVal}}\}$ (1) (2) (Lemma 23)
(23) $\Sigma_{\hat{T}} \vdash t_i : \sigma_i$ and $\sigma_i = \mathsf{encTys}(\tau_i)$ (7) (22) (Lemma 20)
(24) $[\![t_i]\!]_{\theta',\mathbf{A}} \in \sigma_i^{\mathbf{A}}$ (23) (A2) (A4) (Lemma 5)
(25) $v''_i \in \sigma_i^{\mathbf{A}}$ (24) (9)
(26) $x_i : \tau_i \in \hat{T}_f$ (def. $\hat{T}_f$)
(27) $x_i : \sigma_i \in \Sigma_{\hat{T}_f}$ (26) (def. $\Sigma_{\hat{T}_f}$) (23)
(28) $\theta'_f(x_i) = v''_i$ for $i = 1, \ldots, n$ (IH5) (9) (25) (27) (Lemma 18)
(29) $\theta'_f, \mathbf{A} \models_{\texttt{SMT}} \Psi$ (A3)
(30) $[\![\forall x_1 : \sigma_1 \ldots x_n : \sigma_n . (f\ x_1\ \ldots\ x_n) = (\mathsf{ite}\ (\bigwedge \Phi')\ t'\ \lightning)]\!]_{\theta'_f,\mathbf{A}} = \mathsf{true}$ (29) (IH1) (def. Ψ)

(31) $[\![f\ x_1\ \dots\ x_n]\!]_{\theta'_f,\mathbf{A}} = [\![(\mathsf{ite}\ (\bigwedge \Phi')\ t'\ \lightning)]\!]_{\theta'_f,\mathbf{A}}$ — (30) (28) (25) (def. $[\![\cdot]\!]_{\theta'_f,\mathbf{A}}$) (A2)

(32) $[\![\bigwedge \Phi']\!]_{\theta'_f,\mathbf{A}} = \mathsf{true}$ — (21)

(33) $[\![(\mathsf{ite}\ (\bigwedge \Phi')\ t'\ \lightning)]\!]_{\theta'_f,\mathbf{A}} = [\![t']\!]_{\theta'_f,\mathbf{A}}$ — (def. $[\![\cdot]\!]_{\theta'_f,\mathbf{A}}$) (A2)

(34) $[\![(\mathsf{ite}\ (\bigwedge \Phi')\ t'\ \lightning)]\!]_{\theta'_f,\mathbf{A}} = v''$ — (33) (20)

(35) $[\![f\ x_1\ \dots\ x_n]\!]_{\theta'_f,\mathbf{A}} = v''$ — (34) (31)

(36) $f^{\mathbf{A}}([\![x_1]\!]_{\theta'_f,\mathbf{A}}, \dots, [\![x_n]\!]_{\theta'_f,\mathbf{A}}) = v''$ — (35) (def. $[\![\cdot]\!]_{\theta,\mathbf{A}}$)

(37) $f^{\mathbf{A}}(v''_i, \dots, v''_i) = v''$ — (36) (28)

(38) $f^{\mathbf{A}}([\![t_1]\!]_{\theta',\mathbf{A}}, \dots, [\![t_n]\!]_{\theta',\mathbf{A}}) = v''$ — (37) (9)

(39) $[\![f\ t_1\ \dots\ t_n]\!]_{\theta',\mathbf{A}} = v''$ — (38) (def. $[\![\cdot]\!]_{\theta',\mathbf{A}}$)

(40) $[\![t]\!]_{\theta',\mathbf{A}} = v''$ — (A1) (def. $\mathsf{encLExp}_{\hat{T}}$)

(41) $[\![v'']\!]_{\mathbf{A}} = v$ — (20)

(42) $\theta', \mathbf{A} \models_{\mathtt{SMT}} \bigcup_{i=1}^{n} \Phi_i$ — (10)

(43) $[\![(\neq\ t\ \lightning)]\!]_{\theta',\mathbf{A}} = \mathsf{true}$ — (40) (41) (def. $[\![\cdot]\!]_{\mathbf{A}}$) (def. $[\![\cdot]\!]_{\theta',\mathbf{A}}$)

(44) $\theta', \mathbf{A} \models_{\mathtt{SMT}} \Phi$ — (42) (43) (def. $\mathsf{encLExp}_{\hat{T}}$)

We have shown what was required. Now we show the converse to be true. Assume that (A7) $[\![t]\!]_{\theta',\mathbf{A}} = v'$ and (A8) $\theta', \mathbf{A} \models_{\mathtt{SMT}} \Phi$. Then, for each $i \in \{1, \dots, n\}$:

(45) $(t_i, \sigma_i, \Phi_i) = \mathsf{toSort}_{\tau_i}(\mathsf{encLExp}_{\hat{T}}(E_i))$ — (A1) (def. $\mathsf{encLExp}_{\hat{T}}$)

(46) $(t'_i, \sigma'_i, \Phi'_i) = \mathsf{encLExp}_{\hat{T}}(E_i)$ for some $(t'_i, \sigma'_i, \Phi'_i)$ — (45) (def. toSort_{τ_i})

(47) $\Phi_i \subseteq \Phi$ — (45) (def. $\mathsf{encLExp}_{\hat{T}}$)

(48) $\theta', \mathbf{A} \models_{\mathtt{SMT}} \Phi_i$ — (47) (A8) (def. $\models_{\mathtt{SMT}}$)

(49) $\Phi'_i \subseteq \Phi_i$ — (45) (46) (Lemma 22)

(50) $\theta', \mathbf{A} \models_{\mathtt{SMT}} \Phi'_i$ — (48) (49) (def. $\models_{\mathtt{SMT}}$)

(51) $\Sigma_{\hat{T}} \vdash t'_i : \sigma_i$ and $\Sigma_{\hat{T}} \vdash \Phi'_i : \sigma_{\mathsf{Bool}}$ — (46) (Lemma 11)

(52) $[\![t'_i]\!]_{\theta',\mathbf{A}} = v''_i$ for some v''_i — (51) (A4) (A2) (def. $[\![\cdot]\!]_{\theta',\mathbf{A}}$)

(53) $v_i \neq \lightning$ — (IH2b) (46) (A2) (A3) (A4) (50) (52)

(54) $\theta \models_{\mathsf{Type}} \hat{T}$ — (IH2b) (46) (A2) (A3) (A4) (50) (52)

(55) $[\![v''_i]\!]_{\theta',\mathbf{A}} = v$ — (IH2b) (46) (A2) (A3) (A4) (50) (52)

(56) $[\![t_i]\!]_{\theta',\mathbf{A}} = v'_i$ for some v'_i — (A2) (A4) (45) (46) (52) (55)

(57) $[\![v'_i]\!]_{\mathbf{A}} = v_i$ — (A2) (A4) (45) (46) (52) (55)

Turning our attention towards the function body, we define $\hat{T}_f$ to be such that $x_i : \tau_i \in \hat{T}_f$ for $i = 1, \dots, n$. Define also $\theta'_f = [\![\theta_f]\!]_{\Sigma_{\hat{T}_f}}$. Then:

(58) $(t'', \sigma'', \Phi'') = \mathsf{encLExp}_{\hat{T}_f}(E)$ for some (t'', σ'', Φ'') — (IH1) (Property 2)

(59) $(t', \sigma', \Phi') = \mathsf{toVal}(\mathsf{encLExp}_{\hat{T}_f}(E))$ for some (t', σ', Φ') — (58) (def. toVal)

(60) $(t', \sigma', \Phi') = \mathsf{toSort}_{\mathsf{LVal}}(\mathsf{encLExp}_{\hat{T}_f}(E))$ for some (t', σ', Φ') — (59) (def. $\mathsf{toSort}_{\mathsf{LVal}}$)

(61) $\theta'_f, \mathbf{A} \models_{\mathtt{SMT}} \Psi$ — (A3)

(62) $[\![\forall x_1 : \sigma_1 \dots x_n : \sigma_n.\ (f\ x_1\ \dots\ x_n) = (\mathsf{ite}\ (\bigwedge \Phi')\ t'\ \lightning)]\!]_{\theta'_f,\mathbf{A}} = \mathsf{true}$ — (61) (IH1) (def. Ψ)

(63) $\sigma'_i \in \{\mathsf{encTys}(\tau_i), \sigma_{\mathsf{LVal}}\}$ (45) (46) (Lemma 23)
(64) $\Sigma_{\hat{T}} \vdash t_i : \sigma_i$ and $\sigma_i = \mathsf{encTys}(\tau_i)$ (51) (63) (Lemma 20)
(65) $[\![t_i]\!]_{\theta',\mathbf{A}} \in \sigma_i^{\mathbf{A}}$ (64) (A2) (A4) (Lemma 5)
(66) $v'_i \in \sigma_i^{\mathbf{A}}$ (65) (9)
(67) $x_i : \tau_i \in \hat{T}_f$ (def. $\hat{T}_f$)
(68) $x_i : \sigma_i \in \Sigma_{\hat{T}_f}$ (67) (def. $\Sigma_{\hat{T}_f}$) (23)
(69) $\theta'_f(x_i) = v'_i$ for $i = 1, \ldots, n$ (IH5) (57) (66) (68) (Lemma 18)
(70) $[\![f\ x_1\ \ldots\ x_n]\!]_{\theta'_f,\mathbf{A}} = [\![(\mathsf{ite}\ (\bigwedge \Phi')\ t'\ \lightning)]\!]_{\theta'_f,\mathbf{A}}$ (62) (66) (69) (def. $[\![\cdot]\!]_{\theta'_f,\mathbf{A}}$) (A2)
(71) $t = f\ t_1\ \ldots\ t_n$ (45) (IH1) (A1) (def. $\mathsf{encLExp}_{\hat{T}}$)
(72) $[\![t]\!]_{\theta',\mathbf{A}} = f^{\mathbf{A}}([\![t_1]\!]_{\theta',\mathbf{A}}, \ldots, [\![t_n]\!]_{\theta',\mathbf{A}})$ (71) (def. $[\![\cdot]\!]_{\theta',\mathbf{A}}$)
(73) $v' = f^{\mathbf{A}}(v'_1, \ldots, v'_n)$ (72) (A7) (56)
(74) $(\neq\ t\ \lightning) \in \Phi$ (A1) (def. $\mathsf{encLExp}_{\hat{T}}$)
(75) $[\![\neq\ t\ \lightning]\!]_{\theta',\mathbf{A}} = \mathsf{true}$ (74) (A8)
(76) $[\![t]\!]_{\theta',\mathbf{A}} \neq \lightning^{\mathbf{A}}$ (75) (def. $[\![\cdot]\!]_{\theta',\mathbf{A}}$) (A2)
(77) $v' \neq \lightning^{\mathbf{A}}$ (76) (A7)
(78) $f^{\mathbf{A}}(v'_1, \ldots, v'_n) = [\![(\mathsf{ite}\ (\bigwedge \Phi')\ t'\ \lightning)]\!]_{\theta',\mathbf{A}}$ (69) (70) (def. $[\![\cdot]\!]_{\theta',\mathbf{A}}$)
(79) $[\![(\mathsf{ite}\ (\bigwedge \Phi')\ t'\ \lightning)]\!]_{\theta',\mathbf{A}} = v'$ (78) (73)
(80) $[\![t']\!]_{\theta',\mathbf{A}} = v'$ (79) (77) (A2) (def. $[\![\cdot]\!]_{\theta',\mathbf{A}}$)
(81) $[\![(\bigwedge \Phi')]\!]_{\theta',\mathbf{A}} = \mathsf{true}$ (79) (77) (A2) (def. $[\![\cdot]\!]_{\theta',\mathbf{A}}$)
(82) $\theta', \mathbf{A} \models_{\mathtt{SMT}} \Phi'$ (81) (A2) (def. $[\![\cdot]\!]_{\theta',\mathbf{A}}$)
(83) $\Phi'' \subseteq \Phi''$ (58) (60) (Lemma 22)
(84) $\theta', \mathbf{A} \models_{\mathtt{SMT}} \Phi''$ (83) (82) (def. $\models_{\mathtt{SMT}}$)
(85) $[\![t'']\!]_{\theta',\mathbf{A}} = v''$ for some v'' (def. $[\![\cdot]\!]_{\theta',\mathbf{A}}$)
(86) $[\![v'']\!]_{\mathbf{A}} = v$ (58) (A2) (A3) (def. θ'_f) (84) (85) (IH4)
(87) $v \neq \lightning$ (58) (A2) (A3) (def. θ'_f) (84) (85) (IH4)
(88) $[\![v']\!]_{\mathbf{A}} = v$ (A2) (A4) (60) (85) (86) (84) (Lemma 21)

We have shown what was required: (88), (87) and (54). □

E.3. Proof of Lemma 15

Proof. Assume that:

(A1) $\Phi = \mathsf{encPC}_{\hat{T}}(\hat{\pi})$ (A2) $\mathbf{A} \in \mathcal{M}(\mathcal{T}^{\Sigma_{\hat{T}}})$ (A3) $\mathbf{A} \models_{\mathtt{SMT}} \Psi$ (A4) $[\![\theta]\!]_{\Sigma_{\hat{T}},\mathbf{A}} = \theta'$

Then:

(1) $(\varphi', \sigma', \Phi') = \mathsf{toBool}(\mathsf{encLExp}_{\hat{T}}(\hat{\pi}))$ for some φ', σ', Φ' (A1)
(2) $\Phi = \Phi' \cup \{\varphi'\}$ (A1) (1)
(3) $(t'', \sigma'', \Phi'') = \mathsf{encLExp}_{\hat{T}}(\hat{\pi})$ for some t'', σ'', Φ'' (1) (def. toBool)
(4) $[\![\hat{\pi}]\!]_\theta = v$ for some v (Property 1)

First assume that (A5) $[\![\hat{\pi}]\!]_\theta = \mathsf{true}$ and (A6) $\theta \models_{\mathsf{Type}} \hat{T}$. We aim to show that $\theta', \mathbf{A} \models_{\mathtt{SMT}} \Phi$. We proceed by applying Lemma 14:

(5)	$v = \mathsf{true}$	(4) (A5)
(6)	$v \neq \lightning$	(5)
(7)	$\theta', \mathbf{A} \models_{\mathtt{SMT}} \Phi''$	(3) (4) (A2) (A3) (A4) (6) (A6) (Lemma 14)
(8)	$[\![t'']\!]_{\theta',\mathbf{A}} = v''$ for some v''	(3) (4) (A2) (A3) (A4) (6) (A6) (Lemma 14)
(9)	$[\![v'']\!]_{\mathbf{A}} = v$	(3) (4) (A2) (A3) (A4) (6) (A6) (Lemma 14)
(10)	$v \in \mathsf{Bool}$	(5)
(11)	$\Sigma_{\hat{T}} \vdash t'' : \sigma''$	(3) (Lemma 11)
(12)	$[\![t'']\!]_{\theta',\mathbf{A}} \in \sigma''^{\mathbf{A}}$	(11) (Lemma 5)
(13)	$v'' \in \sigma''^{\mathbf{A}}$	(12) (8)
(14)	$(\varphi', \sigma', \Phi') = \mathsf{toSort}_{\mathsf{Bool}}(\mathsf{encLExp}_{\hat{T}}(\hat{\pi}))$	(1) (def. $\mathsf{toSort}_{\mathsf{Bool}}$)

(15)	$[\![\varphi']\!]_{\mathbf{A}} = v$	(A2) (14) (8) (9) (10) (13) (7) (Lemma 21)
(16)	$\theta', \mathbf{A} \models_{\mathtt{SMT}} \Phi'$	(A2) (14) (8) (9) (10) (13) (7) (Lemma 21)
(17)	$[\![\varphi']\!]_{\mathbf{A}} = \mathsf{true}$	(15) (5)
(18)	$\theta', \mathbf{A} \models_{\mathtt{SMT}} \Phi$	(16) (17) (2)

We have that $\theta', \mathbf{A} \models_{\mathtt{SMT}} \Phi$ as required.

Now suppose that (A7) $\theta', \mathbf{A} \models_{\mathtt{SMT}} \Phi$. It follows that:

(19)	$\theta', \mathbf{A} \models_{\mathtt{SMT}} \Phi'$	(A7) (2)
(20)	$[\![\varphi']\!]_{\theta',\mathbf{A}} = \mathsf{true}$	(A7) (2)
(21)	$\Phi'' \subseteq \Phi'$	(1) (3) (Lemma 22)
(22)	$\theta', \mathbf{A} \models_{\mathtt{SMT}} \Phi''$	(19) (21) (def. $\models_{\mathtt{SMT}}$)
(23)	$\varphi' = t''$ or $\varphi' = \mathsf{getBool}\ t''$	(1) (3) (def. toBool)

We have two cases:

(C1) $\varphi' = t''$

(24)	$[\![t'']\!]_{\theta',\mathbf{A}} = \mathsf{true}$	(C1) (20)
(25)	$[\![\mathsf{true}]\!]_{\mathbf{A}} = \mathsf{true}$	(def $[\![\cdot]\!]_{\mathbf{A}}$)

(C2) $\varphi' = \mathsf{getBool}\ t''$

(26)	$[\![\mathsf{getBool}\ t'']\!]_{\theta',\mathbf{A}} = \mathsf{true}$	(C2) (20)
(27)	$[\![t'']\!]_{\theta',\mathbf{A}} = \mathsf{bool\ true}$	(26) (def. $[\![\cdot]\!]_{\theta',\mathbf{A}}$) (A2)
(28)	$[\![\mathsf{bool\ true}]\!]_{\mathbf{A}} = \mathsf{true}$	(def $[\![\cdot]\!]_{\mathbf{A}}$)

In both cases, we have that (29) $[\![t'']\!]_{\theta',\mathbf{A}} = v''$ for some v'', where (30) $[\![v'']\!]_{\mathbf{A}} = \mathsf{true}$. It therefore follows that:

(31)	$[\![v'']\!]_{\mathbf{A}} = v$	(3) (4) (A2) (A3) (A4) (29) (22) (Lemma 14)
(32)	$v \neq \lightning$	(3) (4) (A2) (A3) (A4) (29) (22) (Lemma 14)
(33)	$\theta \models_{\mathsf{Type}} \hat{T}$	(3) (4) (A2) (A3) (A4) (29) (22) (Lemma 14)
(34)	$[\![\hat{\pi}]\!]_{\theta} = \mathsf{true}$	(31) (30) (4)

□

E.4. Proof of Lemma 16

Proof. Assume that:

(A1) $\Phi = \mathsf{encSto}_{\hat{T}}(\hat{s})$ (A2) $\mathbf{A} \in \mathcal{M}(\mathcal{T}^{\Sigma_{\hat{T}}})$ (A3) $\mathbf{A} \models_{\texttt{SMT}} \Psi$ (A4) $[\![\theta]\!]_{\Sigma_{\hat{T}},\mathbf{A}} = \theta'$

Assume first that (A5) $\theta, s \models_{\mathsf{Sto}} \hat{s}$ for some s, and that (A6) $\theta \models_{\mathsf{Type}} \hat{T}$. We aim to show that $\theta', \mathbf{A} \models_{\texttt{SMT}} \Phi$. Then, for all $x \in \mathsf{dom}(\hat{s})$:

(1) $\hat{s}(x) = E$ (A5)
(2) $[\![E]\!]_\theta = v$ (A5)
(3) $v = s(x)$ (A5)
(4) $v \in \mathsf{Val}$ (3) (def. s)
(5) $v \neq \lightning$ (4)

We have two cases:

(C1) $\hat{T} \not\vdash \hat{s}$
(6) $\neg\exists s.\, \theta, s \models_{\mathsf{Sto}} \hat{s}$ (C1) (A6) (Lemma 3)
(7) $\bot$ (6) (A5)

(C2) $\hat{T} \vdash \hat{s}$
(8) $(t', \sigma', \Phi') = \mathsf{encLExp}_{\hat{T}}(E)$ for $\hat{s}(x) = E$ (C2) (A1) (def. $\mathsf{encSto}_{\hat{T}}$)
(9) $\theta', \mathbf{A} \models_{\texttt{SMT}} \Phi'$ (8) (A2) (A3) (A4) (2) (5) (A6) (Lemma 14)
(10) $\theta', \mathbf{A} \models_{\texttt{SMT}} \Phi$ (9) (def. $\mathsf{encSto}_{\hat{T}}$)

In both cases, we conclude that $\theta', \mathbf{A} \models_{\texttt{SMT}} \Phi$ as required.

Now we show the converse to be true. Assume that (A7) $\theta', \mathbf{A} \models_{\texttt{SMT}} \Phi$. Then, for arbitrary $x \in \mathsf{dom}\hat{s}$ with $E = \hat{s}(x)$

(11) $[\![E]\!]_\theta = v$ (Property 1)
(12) $\Phi \neq \{\mathsf{false}\}$ (A7)
(13) $(t', \sigma', \Phi') = \mathsf{encLExp}_{\hat{T}}(E)$ (12) (A1) (def. $\mathsf{encSto}_{\hat{T}}$)
(14) $[\![t']\!]_{\theta',\mathbf{A}} = v'$ for some v' (def. $[\![\cdot]\!]_{\theta',\mathbf{A}}$)
(15) $[\![v']\!]_{\mathbf{A}} = v$ (13) (A2) (A3) (A4) (11) (A7) (14) (Lemma 14)
(16) $v \neq \lightning$ (13) (A2) (A3) (A4) (11) (A7) (14) (Lemma 14)
(17) $\theta \models_{\mathsf{Type}} \hat{T}$ (13) (A2) (A3) (A4) (11) (A7) (14) (Lemma 14)
(18) $v \in \mathsf{Val}$ (16)

Since this holds for all such x and E, it follows from (18) that we can define a s such that $s(x) = [\![\hat{s}(x)]\!]_\theta$. Therefore, we conclude that $\exists s.\, \theta, s \models_{\mathsf{Sto}} \hat{s}$. We also have from (17) that $\theta \models_{\mathsf{Type}} \hat{T}$. □

E.5. Proof of Lemma 12

Proof. Take an arbitrary symbolic state $\hat{\sigma} = (\hat{s}, \hat{\mu}, \hat{\mathcal{P}}, \hat{\pi}, \hat{T})$. Let $\mathsf{encSAT}^{\mathsf{OX}}(\hat{\sigma}) = \Phi$. Take arbitrary θ, μ, s. Suppose that these satisfy the symbolic state: **(A1)** $\theta, (\mu, s) \models \hat{\sigma}$. In particular, we have:

(1)	$\theta, s \models_{\mathsf{Sto}} \hat{s}$	(A1)
(2)	$[\![\hat{\pi}]\!]_\theta = \mathsf{true}$	(A1)
(3)	$\theta \models_{\mathsf{Type}} \hat{T}$	(A1)
(4)	$\mu = \mu_1 \cdot \mu_2$	(A1)
(5)	$\theta, \mu_1 \models_{\mathsf{Mem}} \hat{\mu}$	(A1)
(6)	$\theta, \mu_2 \models_{\mathsf{Pred}} \hat{\mathcal{P}}$	(A1)

Since our symbolic state is satisfiable (A1), by Lemma 4 we have that **(7)** $\vdash \hat{\sigma}$. Then:

(8)	$\Phi = \Phi_{\mathsf{Sto}} \cup \Phi_{\mathsf{PC}} \cup \Phi_{\mathsf{SMem}}$	(def. $\mathsf{encSAT}^{\mathsf{OX}}$) (7)

where

(9)	$\Phi_{\mathsf{Sto}} = \mathsf{encSto}_{\hat{T}}(\hat{s})$
(10)	$\Phi_{\mathsf{PC}} = \mathsf{encPC}_{\hat{T}}(\hat{\pi})$
(11)	$\Phi_{\mathsf{SMem}} = \mathsf{encSMem}_{\hat{T}}(\hat{\mu})$

From Lemma 19, we know there is some **(12)** $\mathbf{A} \in \mathcal{M}(\mathcal{T}^{\Sigma_{\hat{T}}})$ such that **(13)** $\mathbf{A} \models_{\mathtt{SMT}} \Psi$. Let **(14)** $\theta' = [\![\theta]\!]_{\Sigma_{\hat{T}}, \mathbf{A}}$. It then follows that:

(15)	$\theta', \mathbf{A} \models_{\mathtt{SMT}} \Phi_{\mathsf{PC}}$	(9) (12) (13) (14) (Lemma 15) (2) (3)
(16)	$\theta', \mathbf{A} \models_{\mathtt{SMT}} \Phi_{\mathsf{Sto}}$	(9) (12) (13) (14) (Lemma 16) (1) (3)
(17)	$\theta', \mathbf{A} \models_{\mathtt{SMT}} \Phi_{\mathsf{Mem}}$	(9) (12) (13) (14) (Instance Property 4) (5) (3)
(18)	$\theta', \mathbf{A} \models_{\mathtt{SMT}} \Phi$	(15) (16) (17)

Thus we conclude that $\theta \models_{\Sigma_{\hat{T}}} \Phi$. □

E.6. Proof of Lemma 13

Proof. Take an arbitrary symbolic state $\hat{\sigma} = (\hat{s}, \hat{\mu}, \hat{\mathcal{P}}, \hat{\pi}, \hat{T})$. Let $\Phi = \mathsf{encMem}^{\mathsf{UX}}_{\hat{T}}(\hat{\sigma})$. Take an arbitrary θ, and assume **(A1)** $\theta \models_{\Sigma_{\hat{T}}} (\Phi \cup \Psi)$. Let **(A2)** $\theta' = [\![\theta]\!]_{\Sigma_{\hat{T}}, \mathbf{A}}$ for some $\mathbf{A} \in \mathcal{M}(\mathcal{T}^{\Sigma_{\hat{T}}})$. By definition of $\models_{\Sigma_{\hat{T}}}$, we know:

(1)	$\theta', \mathbf{A} \models_{\mathtt{SMT}} \Phi$	(A1) (A2)
(2)	$\theta', \mathbf{A} \models_{\mathtt{SMT}} \Psi$	(A1) (A2)

Since $\mathtt{fv}(\Psi) = \emptyset$, we conclude: **(3)** $\mathbf{A} \models_{\mathtt{SMT}} \Psi$ (2). Since $\theta', \mathbf{A} \models_{\mathtt{SMT}} \Phi$, we must have: **(4)** $\Phi \neq \{\mathsf{false}\}$.

By definition of $\mathsf{encMem}^{\mathsf{UX}}_{\hat{T}}$, we know:

(5) $\hat{\mathcal{P}} = \emptyset$ (def. $\mathsf{encMem}^{\mathsf{UX}}_{\hat{T}}$) (4)
(6) $\vdash \hat{\sigma}$ (def. $\mathsf{encMem}^{\mathsf{UX}}_{\hat{T}}$) (4)
(7) $\Phi = \Phi_{\mathsf{Sto}} \cup \Phi_{\mathsf{PC}} \cup \Phi_{\mathsf{SMem}}$ (def. $\mathsf{encMem}^{\mathsf{UX}}_{\hat{T}}$) (4)

where:

(8) $\Phi_{\mathsf{Sto}} = \mathsf{encSto}_{\hat{T}}(\hat{s})$
(9) $\Phi_{\mathsf{PC}} = \mathsf{encPC}_{\hat{T}}(\hat{\pi})$
(10) $\Phi_{\mathsf{SMem}} = \mathsf{encSMem}_{\hat{T}}(\hat{\mu})$

From Lemma 16, using (1), (8), and (7), we obtain:

(11) $\exists s.\ \theta, s \models_{\mathsf{Sto}} \hat{s}$
(12) $\hat{T} \models_{\mathsf{Type}} \theta$

From Lemma 15, (1), (9), and (7) give:

(13) $[\![\hat{\pi}]\!]_{\theta} = \mathsf{true}$

From Instance Property 4, (1), (10), and (7) imply:

(14) $\exists \mu.\ \mu \models_{\mathsf{Mem}} \hat{\mu}$

Since $\hat{\mathcal{P}} = \emptyset$, we have:

(15) $\mu_{\emptyset} \models_{\mathsf{Pred}} \hat{\mathcal{P}}$ (5)

Since (16) $\mu = \mu \cdot \mu_{\emptyset}$, we get:

(17) $\exists \mu, \mu_1, \mu_2.\ \Big(\mu = \mu_1 \cdot \mu_2 \ \wedge\ \mu_1 \models_{\mathsf{Mem}} \hat{\mu} \ \wedge\ \mu_2 \models_{\mathsf{Pred}} \hat{\mathcal{P}}\Big)$ (14) (15) (16)

Finally, by the definition of symbolic state satisfaction:

(18) $\exists s, \mu.\ \theta, (s, \mu) \models \hat{\sigma}$ (11) (12) (13) (16)

□

E.7. Proof of Theorem 8

Proof. We proceed by induction over the definition of the unfold judgement. Theorem 8 holds trivially for base case of $\mathsf{unfold}(\hat{\sigma}) \rightsquigarrow \hat{\sigma}$. For the inductive case, take arbitrary $\hat{\sigma} = (\hat{s}, \hat{\mu}, \hat{\mathcal{P}}, \hat{\pi}, \hat{T})$.

Inductive Hypothesis. Assume that:

(IH1) $\mathsf{unfold}(\hat{\sigma}) \rightsquigarrow \hat{\sigma}''$ (IH2) $\mathsf{SAT}(\hat{\sigma}'') \implies \mathsf{SAT}(\hat{\sigma})$ (IH3) $\hat{\mathcal{P}} = \hat{\sigma}''.\mathsf{pred}$
(IH4) $p(\vec{E}_1; \vec{E}_2) \in \hat{\mathcal{P}}$ (IH5) $\hat{\mathcal{P}}' = \hat{\mathcal{P}} \setminus \{p(\vec{E}_1; \vec{E}_2)\}$ (IH6) $\hat{\sigma}''' = \hat{\sigma}''[\mathsf{pred} := \hat{\mathcal{P}}']$
(IH7) $p(\vec{x}_1; \vec{x}_2)\{A\} \in \mathsf{Preds}$ (IH8) $\hat{\theta} = \{\vec{x}_1 \mapsto \vec{E}_1,\ \vec{x}_2 \mapsto \vec{E}_2\}$ (IH9) $\mathsf{produce}(A, \hat{\theta}, \hat{\sigma}''') \rightsquigarrow \hat{\sigma}'$

To Show. $\mathsf{SAT}(\hat{\sigma}') \implies \mathsf{SAT}(\hat{\sigma})$.

Assume that (A1) $\mathsf{SAT}(\hat{\sigma}')$.

(1)	$\theta, s, \mu \models \hat{\sigma}'$	(A1)
(2)	$\mu = \mu_A \cdot \mu_f$	(1) (IH9) (Property 4)
(3)	$\theta, \mu_A \models \hat{\theta}(A)$	(1) (IH9) (Property 4)
(4)	$\theta, s, \mu_f \models \hat{\sigma}'''$	(1) (IH9) (Property 4)
(5)	$\theta, \mu_A \models p(\vec{E}_1; \vec{E}_2)$	(3) (IH8) (IH7)
(6)	$\theta, \mu_{f1} \models \hat{\mathcal{P}}'$	(IH6) (4)
(7)	$\theta, \mu_{f2} \models \hat{\sigma}'''.\mathsf{mem}$	(IH6) (4)
(8)	$\mu_f = \mu_{f1} \cdot \mu_{f2}$	(IH6) (4)
(9)	$\mu = \mu_A \cdot \mu_{f1} \cdot \mu_{f2}$	(8) (2)
(10)	$\mu_1 = \mu_A \cdot \mu_{f1}$	(9)
(11)	$\mu = \mu_1 \cdot \mu_{f2}$	(9) (10)
(12)	$\theta, \mu_1 \models \hat{\mathcal{P}}$	(IH5) (6) (5)
(13)	$\theta, s, \mu \models \hat{\sigma}''$	(IH6) (4) (7-12)
(14)	$\mathsf{SAT}(\Sigma'')$	(13)
(15)	$\mathsf{SAT}(\Sigma)$	(14) (IH2)

□

E.8. Proof of Theorem 9

Proof. Let (1) $\Phi = \mathsf{encImpl}(\hat{\sigma}, E)$, where $\hat{\sigma}.\mathsf{te} = \hat{T}$. Assume that (A1) $\mathsf{UNSAT}_{\Sigma_{\hat{T}}}(\Phi \cup \Psi)$. Then:

(2) $\theta, \mathbf{A} \not\models_{\mathsf{SMT}} \Phi \cup \Psi$ for all valuations θ into $\mathbf{A} \in \mathcal{M}(\mathcal{T}^{\Sigma_{\hat{T}}})$ (A1)

Now assume that (A2) $\mathsf{SAT}(\hat{\sigma})$. Let (3) $\Phi_{\hat{\sigma}} = \mathsf{encSAT}^{\mathsf{OX}}(\hat{\sigma})$.

(4)	$\theta, \sigma \models \hat{\sigma}$	(A2) (def. SAT)
(5)	$\theta', \mathbf{A} \models_{\mathsf{SMT}} \Phi_{\hat{\sigma}} \cup \Psi$	(4) (3) (Lemma 12)
(6)	$\mathbf{A} \in \mathcal{M}(\mathcal{T}^{\Sigma_{\hat{T}}})$ and $\theta' = [\![\theta]\!]_{\Sigma_{\hat{T}}, \mathbf{A}}$	(4) (3) (Lemma 12)

We have two cases:

(C1) $\hat{T} \vdash E : \mathsf{Bool}$

(7)	$\Phi = \Phi_{\hat{\sigma}} \cup \{\neg(\varphi_E \land \bigwedge \Phi_E)\}$ where $(\varphi_E, \sigma_E, \Phi_E) = \mathsf{toBool}(\mathsf{encLExp}_{\hat{T}}(E))$	(C1) (3) (1) (def. encImpl)
(8)	$[\![\neg(\varphi_E \land \bigwedge \Phi_E)]\!]_{\theta', \mathbf{A}} = \mathsf{false}$	(2) (5) (6) (7) (def. $\models_{\mathsf{SMT}}$)
(9)	$\mathbf{A} \models_{\mathsf{SMT}} \Psi$	(5) ($\mathtt{fv}(\Psi) = \emptyset$)
(10)	$[\![(\varphi_E \land \bigwedge \Phi_E)]\!]_{\theta', \mathbf{A}} = \mathsf{true}$	(8) (def. $[\![\cdot]\!]_{\theta', \mathbf{A}}$) (6)
(11)	$[\![\varphi_E]\!]_{\theta', \mathbf{A}} = \mathsf{true}$	(10) (def. $[\![\cdot]\!]_{\theta', \mathbf{A}}$) (6)

(12) $\theta', \mathbf{A} \models_{\mathtt{SMT}} \Phi_E$ (10) (def. $[\![\cdot]\!]_{\theta',\mathbf{A}}$) (4) (def. $\models_{\mathtt{SMT}}$)
(13) $[\![E]\!]_\theta = v$ (Property 1)
(14) $(t'_E, \sigma'_E, \Phi'_E) = \mathsf{encLExp}_{\hat{T}}(E)$ (7) (def. toBool)
(15) $[\![t'_E]\!]_{\theta',\mathbf{A}} = v'$
(16) $[\![v']\!]_{\mathbf{A}} = \mathsf{true}$ (15) (11) (7) (def. toBool)
(17) $[\![v']\!]_{\mathbf{A}} = v$ (14) (6) (9) (13) (Lemma 14) (12) (15)
(18) $[\![E]\!]_\theta = \mathsf{true}$ (13) (16) (17)

(C1) $\hat{T} \not\vdash E : \mathsf{Bool}$
(5) $\Phi_{\hat{\sigma}} = \Phi$ (C1) (3) (1) (def. encImpl)
(6) $\theta, \mathbf{A} \not\models_{\mathtt{SMT}} \Phi \cup \Psi$

We have reached a contradiction.

We conclude that in both cases, $[\![E]\!]_\theta = \mathsf{true}$. We have shown that, $\hat{\sigma} \models E$, as required. □

E.9. Additional Lemmas and Proofs

Lemma 19. *For all variants $\Sigma_{\hat{T}}$ of the background signature Σ, there exists a $\Sigma_{\hat{T}}$-structure under which the background assertions are valid:*

$$\exists \mathbf{A} \in \mathcal{M}(\mathcal{T}^{\Sigma_{\hat{T}}}).\ \mathbf{A} \models_{\mathtt{SMT}} \Psi$$

Proof. Consider the $\Sigma_{\hat{T}}$-structure $\mathbf{A}$, which interprets user-defined function symbols as follows:

$$f^{\mathbf{A}}(v_1, \ldots, v_n) = [\![t]\!]_{\theta,\mathbf{A}}$$

where $\theta(x_i) = v_i$ for $i = 1, \ldots, n$, and $(\forall x_1 : \sigma_1, \ldots, x_n : \sigma_n.\ f\ x_1\ \ldots\ x_n = t) \in \Psi$.

Now take an arbitrary $\psi \in \Psi$.

(1) $\psi = (\forall x_1 : \sigma_1, \ldots, x_n : \sigma_n.\ f\ x_1\ \ldots\ x_n = t)$ (def. Ψ)

Let θ be an arbitrary substitution. Define $\theta' = \theta[x_i \mapsto v_i]_{i=1}^n$.
Then:

(2) $[\![f\ x_1\ \ldots\ x_n]\!]_{\theta',\mathbf{A}} = f^{\mathbf{A}}(\theta'(x_1), \ldots, \theta'(x_n))$ (def. $[\![\cdot]\!]_{\theta',\mathbf{A}}$)
(3) $[\![f\ x_1\ \ldots\ x_n]\!]_{\theta',\mathbf{A}} = f^{\mathbf{A}}(v_1, \ldots, v_n)$ (def. θ') (2)
(4) $[\![f\ x_1\ \ldots\ x_n]\!]_{\theta',\mathbf{A}} = [\![t]\!]_{\theta,\mathbf{A}}$ (def. $\mathbf{A}$) (3)
(5) $[\![f\ x_1\ \ldots\ x_n = t]\!]_{\theta',\mathbf{A}} = \mathsf{true}$ (4)

Since θ' was arbitrary, it follows from (5) that:

$$[\![\forall x_1 : \sigma_1, \ldots, x_n : \sigma_n.\ f\ x_1\ \ldots\ x_n = t]\!]_{\theta,\mathbf{A}} = \mathsf{true}$$

Since ψ was arbitrary, we conclude that $\theta, \mathbf{A} \models_{\texttt{SMT}} \Psi$. Finally, since θ was arbitrary, it follows that:

$$\mathbf{A} \models_{\texttt{SMT}} \Psi.$$

□

Lemma 20. *Suppose* $\mathsf{toSort}_\tau(t', \sigma', \Phi') = (t, \sigma, \Phi)$*. Then:*

$$\sigma' \in \{\mathsf{encTys}(\tau), \sigma_{\mathsf{LVal}}\} \land \Sigma_{\hat{\mathcal{T}}} \vdash t' : \sigma' \land \Sigma_{\hat{\mathcal{T}}} \vdash \Phi' : \sigma_{\mathsf{Bool}}$$
$$\Longrightarrow$$
$$\Sigma_{\hat{\mathcal{T}}} \vdash t : \sigma \land \sigma = \mathsf{encTys}(\tau) \land \Sigma_{\hat{\mathcal{T}}} \vdash \Phi : \sigma_{\mathsf{Bool}}$$

where $\Sigma_{\hat{\mathcal{T}}}$ *is some variant of the background signature.*

Proof. The proof of this lemma is trivial, and can be shown by considering each case of τ individually. □

Lemma 21. *Let* $\mathbf{A} \in \mathcal{M}(\mathcal{T}^{\Sigma_{\hat{\mathcal{T}}}})$ *be some* $\Sigma_{\hat{\mathcal{T}}}$*-structure, for a variant* $\Sigma_{\hat{\mathcal{T}}}$ *of the background signature* Σ*. Let* θ *be some valuation into* $\mathbf{A}$*. If* $(t, \sigma, \Phi) = \mathsf{toSort}_\tau(t', \sigma', \Phi')$ *and* $[\![t']\!]_{\theta,\mathbf{A}} = v'$ *and* $[\![v']\!]_{\mathbf{A}} = v$ *and* $v \in \tau$ *and* $v' \in \sigma^{\mathbf{A}}$ *and* $\theta, \mathbf{A} \models_{\texttt{SMT}} \Phi'$*, then there is some* v'' *such that:*

$$[\![t]\!]_{\theta,\mathbf{A}} = v'' \land [\![v'']\!]_{\mathbf{A}} = v \land \theta, \mathbf{A} \models_{\texttt{SMT}} \Phi$$

Proof. Lemma 21 can be shown by considering each case of τ individually. We detail an arguement for this in the case of (C1) $\tau = \mathsf{Int}$. Assume that:

(A1) $\mathbf{A} \in \mathcal{M}(\mathcal{T}^{\Sigma_{\hat{\mathcal{T}}}})$ (A2) $(t, \sigma, \Phi) = \mathsf{toSort}_\tau(t', \sigma', \Phi')$ (A3) $[\![t']\!]_{\theta,\mathbf{A}} = v'$
(A4) $[\![v']\!]_{\mathbf{A}} = v$ (A5) $v \in \tau$ (A6) $v' \in \sigma'^{\mathbf{A}}$
(A7) $\theta, \mathbf{A} \models_{\texttt{SMT}} \Phi'$

Then:

(1) $v' = v \lor v' = \mathsf{int}^{\mathbf{A}}(v)$ — (A4) (A5) (C1) (def. $[\![\cdot]\!]_{\mathbf{A}}$)
(2) $\sigma' \in \{\sigma_{\mathsf{Int}}, \sigma_{\mathsf{LVal}}\}$ — (A2) (Lemma 23) (C1)

We now have two cases

(C1a) $\sigma' = \sigma_{\mathsf{Int}}$

(3) $t = t'$ — (C1) (A2) (C1a) (def. toSort_τ)
(4) $[\![t]\!]_{\theta,\mathbf{A}} = v'$ — (A3) (3)
(5) $[\![v']\!]_{\mathbf{A}} = v$ — (A4)
(6) $\Phi = \Phi'$ — (C1) (A2) (C1a) (def. toSort_τ)
(7) $\theta, \mathbf{A} \models_{\texttt{SMT}} \Phi$ — (7) (A7)

(C1b) $\sigma' = \sigma_{\mathsf{LVal}}$

(8)	$v' = \mathsf{int}^{\mathbf{A}}(v)$	(1) (C1b) (A6)
(9)	$t = \mathsf{getInt}\ t'$	(C1) (C1b) (A2) (def. toSort_τ)
(10)	$[\![t]\!]_{\theta,\mathbf{A}} = \mathsf{getInt}^{\mathbf{A}}([\![t']\!]_{\theta,\mathbf{A}})$	(9) (def. $[\![\cdot]\!]_{\theta,\mathbf{A}}$)
(11)	$[\![t]\!]_{\theta,\mathbf{A}} = \mathsf{getInt}^{\mathbf{A}}(v')$	(10) (A3)
(12)	$[\![t]\!]_{\theta,\mathbf{A}} = \mathsf{getInt}^{\mathbf{A}}(\mathsf{int}^{\mathbf{A}}(v))$	(11) (8)
(13)	$[\![t]\!]_{\theta,\mathbf{A}} = v$	(12) (A1)
(14)	$[\![v]\!]_{\mathbf{A}} = v$	(A5) (C1) (def. $[\![\cdot]\!]_{\mathbf{A}}$)
(15)	$\Phi = \Phi' \cup \{(\mathsf{isInt}\ t')\}$	(C1) (C1b) (A2) (def. toSort_τ)
(16)	$[\![\mathsf{isInt}\ t']\!]_{\theta,\mathbf{A}} = \mathsf{isInt}^{\mathbf{A}}([\![t']\!]_{\theta,\mathbf{A}})$	(def. $[\![\cdot]\!]_{\theta,\mathbf{A}}$)
(17)	$[\![\mathsf{isInt}\ t']\!]_{\theta,\mathbf{A}} = \mathsf{isInt}^{\mathbf{A}}(\mathsf{int}^{\mathbf{A}}(v))$	(16) (A3) (8)
(18)	$[\![\mathsf{isInt}\ t']\!]_{\theta,\mathbf{A}} = \mathsf{true}$	(18) (9)
(19)	$\theta, \mathbf{A} \models_{\mathtt{SMT}} \Phi$	(15) (18) (A7)

In both cases, we conclude that for some v'', $[\![t]\!]_{\theta,\mathbf{A}} = v''$ and $[\![v'']\!]_{\mathbf{A}} = v$ and $\theta, \mathbf{A} \models_{\mathtt{SMT}} \Phi$. □

Lemma 22. *If* $(t, \sigma, \Phi) = \mathsf{toSort}_\tau(t', \sigma', \Phi')$*, then:*

$$\Phi' \subseteq \Phi$$

Proof. Again, this can be shown trivially by considering each case of τ. □

Lemma 23. *For all* t*,* σ*,* Φ *and* τ*:*

$$\sigma \in \{\mathsf{encTys}(\tau),\ \sigma_{\mathsf{LVal}}\} \iff \exists t', \sigma', \Phi'.\ \mathsf{toSort}_\tau(t, \sigma, \Phi) = (t', \sigma', \Phi')$$

Proof. Lemma 23 can be shown trivially by considering each case of τ. □