Informatics

Eine Entscheidungsprozedur für Separation-Logic mit Daten

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik (066 938)

eingereicht von

Stefan Krulj, BSc
Matrikelnummer 00826564

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dipl.-Ing. Georg Weissenbacher, D.Phil.
Mitwirkung: Univ.Ass. Jens Pagel, MSc

Wien, 1. Oktober 2020

Stefan Krulj

Georg Weissenbacher

Informatics

A Decision Procedure for a Separation Logic with Data

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering (066 938)

by

Stefan Krulj, BSc
Registration Number 00826564

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dipl.-Ing. Georg Weissenbacher, D.Phil.
Assistance: Univ.Ass. Jens Pagel, MSc

Vienna, 1st October, 2020

Stefan Krulj

Georg Weissenbacher

Erklärung zur Verfassung der Arbeit

Stefan Krulj, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. Oktober 2020

Stefan Krulj

Acknowledgements

First and foremost, I want to thank my advisor Georg Weissenbacher and associate advisor Jens Pagel for giving me the chance to work on this thesis. Their input has helped me greatly and widened my knowlege in the fields of logic, formal verification and model checking. In particular, without their precedent work on SL^*_{data} and Separation Logic in general this work would not have been possible.

During my studies, I got to know many other students. Some of them briefly, some of them well and some of them even became close friends. I want to thank all of them for the time we spent learning, for their help and different views that often opened my mind, and for their reminders of deadlines. Above all, I want to thank Matthias Fassl and my now partner, Veronika Strobl, for proofreading the thesis and their feedback on scientific writing.

Last but not least, I want to thank my family, in particular my parents and my little sister, for their support, for believing in me and talking me out of my doubts when things were hard.

Kurzfassung

Separation Logic (zu Deutsch etwa "Trennungslogik") ist eine Erweiterung des Hoare-Kalküls, welche logische Schlussfolgerungen über imperative Computer-Programme, die dynamische Heap-Speicher verwenden, erlaubt. Es ist ein beliebter Formalismus für Programme, welche dynamische Ressourcen manipulieren und wird bereits industriell eingesetzt. Die große Ausdruckskraft der *Separation Logic* erlaubt es modular komplizierte Konzepte, wie dynamisch allokierte Arrays, uneingeschränkte Zeigerarithmetik und rekursive Prozeduren abzubilden.

SL^*_{data} ist ein Fragment der *Separation Logic*, die auf einen Kompromiss zwischen Ausdruckskraft und Komplexität abzielt und ermöglicht Aussagen über im Heap-Speicher gespeicherte Daten zu treffen. Dabei liegen das Erfüllbarkeits-Problem in NP und das Gültigkeits-Problem von Implikationen in coNP. SL^*_{data} kann auf Theorien reduziert werden, welche in gängigen SMT-Solvern verfügbar sind, und zur Darstellung von Daten eine beliebige Theorie verwenden.

Mit Separation Logic and Theories (SLOTH) existiert bereits eine Implementierung der Entscheidungsprozedur, diese ist jedoch noch eingeschränkt. SL^*_{data} erlaubt das Auswählen einer Adress- sowie Datentheorie, was jedoch von SLOTH nicht unterstützt wird. Zusätzlich wurde SLOTH noch nicht auf Performance optimiert und die Entscheidungen für einige Probleminstanzen dauern sehr lange.

Wir schlagen Optimierungen für die SL^*_{data}-Entscheidungsprozedur vor. Zusätzlich haben wir die Entscheidungsprozedur sowie die Optimierungen implementiert und einige Einschränkungen von SLOTH überwunden. Die Prozedur wurde in Z3, einem gängigen SMT-Solver integriert. Wir zeigen, die Effektivität der Optimierungen mit einer empirischen Studie.

Abstract

Separation Logic is an extension of Hoare Logic that allows reasoning about imperative programs that use dynamic data structures. It is a popular formalism for programs that manipulate the heap and on top of that has been proven on an industrial scale. Its expressive power permits modular definitions of complex concepts like dynamically allocated arrays, unrestricted pointer arithmetic and recursive procedures and data structures.

SL^*_{data} is a Separation Logic fragment with emphasis on data constraints and decidability. It aims to strike a balance between expressibility and complexity, with satisfiability being decidable in NP and entailment in coNP. SL^*_{data} can be encoded into theories supported by off-the-shelf SMT-solvers and can be combined with arbitrary data theories.

While with Sloth a decision procedure implementation exists, it is still limited. The theory allows for configurable location and data sorts, but is not supported by Sloth. Moreover, Sloth has not yet been optimised for performance and the decision of some specific instances takes a long time.

We propose optimisations for the decision procedure of SL^*_{data}. In addition, have implemented the decision procedure that overcomes some of the current limits of Sloth and integrated it within Z3, a state-of-the-art SMT solver. We show the effectiveness of the proposed optimisations with empirical benchmarks.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Reasoning about imperative programs with dynamic data structures and about the data that such programs process is an important task of modern theoretical computer science. It has potential uses in verification of hardware and software relevant in practice.

Separation Logic is an extension of Hoare Logic that allows reasoning about imperative programs that use dynamic data structures [19]. It is a popular formalism for programs that manipulate the heap and has been proven on an industrial scale [11]. Most notably there are Facebook's static analyzer INFER, which is used in production to prove memory safety upon check-ins within a huge code base consisting of millions of lines of code [6], or Microsoft's SLAYER, which is capable of automatically discovering memory defects and issues in C programs [5].

While Separation Logic is undecidable in general, several decidable fragments have been proposed, which will be briefly introduced in Section 2.2.2. Still, for most of their decision procedures computational complexity is unsatisfactory (EXPTIME or PSPACE).

The priority of this work lies in improving upon an existing decision procedure – SLOTH [12] – that allows reasoning of some relevant verification problems in practice and aims to overcome present shortcomings which are outlined in the next section.

1.2 Problem Statement

SL^*_{data} is a Separation Logic fragment with emphasis on data constraints (i.e. constraints about the memory content instead of just its shape) and decidability. It aims to find a balance between expressibility and complexity, with satisfiability being decidable in NP and entailment in coNP [11]. The details on these results will be explained in Section 3.4.

SL^*_{data} can be encoded into a format for off-the-shelf SMT-solvers. SMT-Solvers are solvers of satisfiability problems with respect to theories. Those theories are expressed in First-Order Logic (FOL) with equality. They consist of additional axioms and interpretations for functions and predicates, and feature efficient decision procedures [10]. The satisfiability problem of SL^*_{data} can be reduced to existing theories and can be combined with arbitrary data theories (i.e. it can use an arbitrary theory supported by the SMT-Solver for its data constrains) [11]. We offer an introduction to SMT and supported theories in Section 2.3.

A decision procedure for SL^*_{data} has been proposed by encoding the formula in SMT [11]. The encoding is described in Chapter 4. However, Sloth [12], the only implementation of the proposed decision procedure, has several limitations:

1. SL^*_{data} is parametric in sorts (also known as *types* in other contexts) and theories representing the memory locations and stored data. In contrast, Sloth uses a single fixed theory [12].

2. For some large and important instance classes Sloth has performance issues. In particular, some instances could be efficiently solved without a complete interpretation of SL^*_{data} semantics.

3. In addition, benchmarks also show that for many of the instances the interfacing between the encoder and SMT solver has significant impact on the run-time. Sloth is not tightly integrated within an SMT solver, which has proved detrimental, as not all features are exposed as public interfaces.

1.3 Contribution and Objectives

The goal of this thesis was to extend the state-of-the-art SMT solver Z3 by a robust implementation of a decision procedure for SL^*_{data}. As mentioned in the previous section, there is an existing implementation of the decision procedure called Sloth [12]. With our implementation overcome some of the limitations mentioned before:

1. We extended upon available SMT syntax to allow specifying theory and sort combinations for the location and the data.

2. We identified poorly performing instances empirically, researched possible optimisations, and evaluated them. Encoding the formula step-wise and reducing the size of formulas by a semantically aware equality propagation could potentially lead to significantly lower run-times.

3. We embedded our implementation of the decision procedure directly into the SMT solver source code, rather than using its public API. This allowed better optimisations and eliminated interfacing overhead.

1.4 Methodology

After an initial literature research, we implemented the decision procedure and integrated it within Z3. The research revealed Separation Logic formulas which served as a baseline for a run-time comparison. In addition to those, we identified low-performing formulas, we proposed and implemented optimisations that overcome problems of the decision procedure. A benchmark against the baseline was used to empirically show the effectiveness of the optimisations. After several iterations, we used the best performing combination of optimisations for a comparison against the existing implementation of the decision procedure Sloth. Our procedure is illustrated in Figure 1.1.

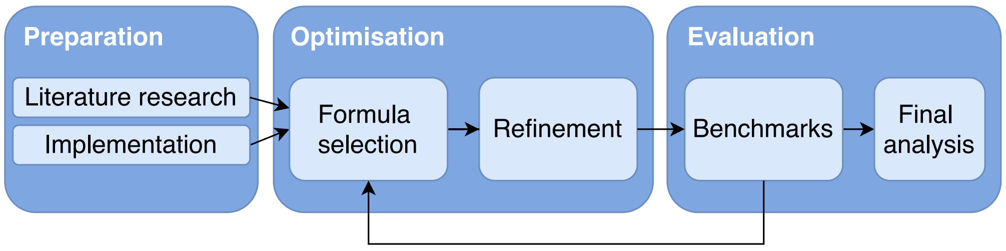


Figure 1.1: Illustration of our working procedure

1.5 Thesis Structure

State of the art

In Section 2 “Related Work”, we will introduce Separation Logic in general and SMT solvers in the next section. Using that as the cornerstone, we will describe SL^*_{data} and how to encode it to SMT in the following sections 3 “Introduction to SL^*_{data}” and “Encoding SL^*_{data} to SMT)”.

Optimisations

We will propose possible optimisations for the existing decision procedure in Section 5 “Optimisations for the SL^*_{data} Decision Procedure”. We will briefly describe the optimisations and the reasoning behind their effectiveness on an algorithmic level.

Implementation

To test the proposed optimisations, we will implement the decision procedure as well as the optimisations. The implementation will be integrated into the existing SMT-Solver “Z3”. The details will be described in Section 6 “Implementation”.

Evaluation and results

We will test the performance of the decision procedure using benchmarks. In Section 7 “Results”, we will compare our implementation to the existing decision procedure as well as evaluate the optimisations and their impact on the run-time.

CHAPTER 2

Related Work

2.1 A Motivating Example

Consider the simple algorithm in Listing 2.1. It computes the maximum value of a linked list. If we wanted to prove its correctness using Hoare triplets, we would need to express pre- and post-conditions according to our specifications: "The argument **a** is a valid list". "No element within **a** is greater than **m**". With traditional propositional logic, we would need to introduce additional constraints to model the memory structure as well as constraints that handle aliases (i.e. values that point to the same memory). Such constraints would not be concise and for more complex programs might quickly become unintelligible. As we will later see Separation Logic and SL^*_{data} in particular allows reasoning to express these kinds of assertions in a concise and clear manner.

```
function max(a: list<int>)
  b = a
  m = head(a)
  while $b \neq \textbf{null}$ do
    if $head(b) > m$ then
      m = head(b)
    fi
    b = next(b)
  od
  return m
end
```

Listing 2.1: max-function for computing the maximum of a linked list

Another problem that occurs when proving correctness using Hoare triplets is composition. Consider we have two sub-routines *foo* and *bar* and have proven facts about them:

$$\{F\}\, foo()\, \{G\} \text{ and } \{F'\}\, bar()\, \{G'\}$$

Consider further that both routines do not share memory locations and *bar* has no effects on the precondition of *foo* and vice versa. When working with programs one is often encountering compositions of sub-routines. For example:

$$\{F \land F'\}\, foo(); bar()\, \{G \land G'\}$$

Traditional Hoare calculus only offers the *sequence rule* for composition:

$$\frac{\{\phi\}\, c_1\, \{\phi'\} \qquad \{\phi'\}\, c_2\, \{\psi\}}{\{\phi\}\, c_1; c_2\, \{\psi\}}$$

However, it can only be applied if the pre- and post-conditions match exactly. To be able to apply our rules about foo and *bar*, we need to apply the *consequence rule*:

$$\frac{\phi \implies \phi' \qquad \{\phi'\}\, c\, \{\psi'\} \qquad \psi' \implies \psi}{\{\phi\}\, c\, \{\psi\}}$$

$F \land F' \implies F$ so $\{F \land F'\}\, foo()\, \{G\}$ is sound. Unfortunately, by applying the rule and strenghten the precondition we lose information about F'. At this point we no longer have a generally valid rule to apply. We need to prove that foo does not alter any fact about F' (i.e. $\{F \land F'\}\ foo()\{G \land F'\}$ is sound). For that we need to revisit our proof for foo in this new context. A solution to this problem would be a rule of the following form:

$$\frac{\{\phi\}\ c\ \{\psi\}}{\{\phi \land \varphi\}\ c\ \{\psi \land \varphi\}}$$

However, the rule is not admissible in general for heap manipulating programs. Consider the following example:

$$\frac{\{x \rightarrow v\}\ (*x) = w\ \{x \rightarrow w\}}{\{x \rightarrow v \land y \rightarrow v\}\ (*x) = w\ \{x \rightarrow w \land y \rightarrow v\}}$$

If x and y point to the same memory address, after executing the assignment (where $(*x)$ is a pointer dereference i.e. access to the memory where x is pointing to) both $x \rightarrow w$ and $y \rightarrow w$ must be true, however, using the rule above we can derive $x \rightarrow w$ and $y \rightarrow v$. Since semantics of pointer arithmetic are so powerful, seemingly local actions can have an effect on global constraints. In the next section, we will take a closer look at Separation Logic, which can help overcome this problem as well.

2.2 Separation Logic

2.2.1 Introduction to Separation Logic

Separation Logic was created in order to counter problems that arise in analysis of dynamic memory when using FOL as assertion language. In FOL, modelling the heap involves introducing global invariants to forbid situations that are not possible or considered bugs in practice (e.g. double allocation, access of unallocated memory and similar problems). Those invariants tend to become unnecessarily long and even simple examples can be difficult to comprehend. This is bad for several reasons: It is hard to "naturally" express assertions about the code, and the tools used for analysing them will have larger inputs and potentially a large computational overhead [19].

Separation Logic circumvents those problems by introducing a memory *separating conjunction*. It is denoted by the asterisk symbol $\ast$. The formula $F * G$ (pronounced F *and separately* G) asserts similarly to the traditional conjunction that both F and G are true, but also asserts that F and G do not share any memory locations. In other words, operands of the separating conjunction are guaranteed to describe disjoint portions of the heap [19].

This fact lets us formulate the so-called *frame rule*:

$$\frac{\{F\}\ c\ \{G\}}{\{F * \phi\}\ c\ \{G * \phi\}}$$

Going back to the example from the previous section, the expression with the facts

$$\{F\}\, foo()\, \{G\} \text{ and } \{F'\}\, bar()\, \{G'\}$$

we can alter our claim to:

$$\{F * F'\}\, foo(); bar()\, \{G * G'\}$$

We can then proceed and apply the new frame rule to prove $\{F * F'\}\, foo()\, \{G * F'\}$ and in a further step $\{G * F'\}\, bar()\, \{G * G'\}$. Note that our previous counter-example

$$\frac{\{x \to v\}\ (*x) = w\ \{x \to w\}}{\{x \to v * y \to v\}\ (*x) = w\ \{x \to w * y \to v\}}$$

is still sound when using the new rule. $x \to v * y \to v$ is unsatisfiable if x and y are aliases to the same memory location.

This shows that it is sufficient to reason about the local heap (i.e. the part of the heap that is modified by a code segment) without side effects involving other parts of the heap. This allows compositional reasoning about different parts of the program [19].

In turn, this leads to expressive power that permits definitions of complex concepts like dynamically allocated arrays, unrestricted pointer arithmetic, recursive procedures, and data structures [19].

Since its introduction, Separation Logic has proven to be an effective tool for reasoning about dynamic data structures in imperative programming languages. It is popular in academical studies regarding deductive verification [16, 17] and symbolic execution [4]. In addition, it has already been adopted by industry (for instance Facebook's INFER [6]). or Microsoft's SLAYER [5]).

2.2.2 Decidable Fragments of Separation Logic

One of the drawbacks of Separation Logic is that validity of formulas is undecidable in general [8]. Therefore, the use of Separation Logic is often restricted to decidable fragments. Many ways were proposed to obtain such a decidable fragment. However, the obtainment of decidability comes at a cost: the fragments restrict the heap to lists [7, 3], only support constraints on data structures (i.e. the shape of the heap) but not on the data itself [7, 3] or the other way around [18]. Also, the fragments often have undesirable run-times for their decision procedures [14] (EXPTIME or PSPACE). In contrast SL^*_{data} supports lists, trees and data. On top of that, its satisfiability problem lies in NP, while entailment lies in coNP [11]. We will concentrate on comparing similar fragments with inductive predicates (i.e. recursively defined predicates) for lists [16] and trees [17], that also feature encodings in SMT.

2.2.3 First Order Logic and SMT Encodings of Separation Logic

Several encodings for Separation Logic fragments were proposed. Many only solve parts of problems that are dealt with in SL^*_{data} (e.g. reachability theory [13] or particular data structures [15]). A complete quantifier-free fragment of SL, including the separating conjunction and separating implication (*magic wand operator*) has been proposed but is lacking an encoding for inductive predicates [18]. Two of the fragments mentioned in the previous subsection feature encodings for lists [16] and trees [17]. However, they require inclusion of theories, that are not available in SMT-LIB [1]. In contrast, the proposed encoding for SL^*_{data} only relies on SMT-LIB defined theories [11], making it readily applicable in a number of off-the-shelf SMT-solvers.

2.3 Satisfiability Modulo Theories SMT

When dealing with FOL, one is often concerned with satisfiability of formulas. While there has been progress for general FOL solvers, many applications only require satisfiability with respect to certain interpretations of predicates. For example if we look at applications based on integer arithmetic and consider the formula $x \cdot 2 \geq y + z$, we are only concerned with usual interpretations of $\geq, +, \cdot$ and 2. *Satisfiability Modulo Theories*, is a technique dedicated to analysing satisfiability of first-order formulas with respect to some underlying theory $\mathcal{T}$. Often even two or more theories are involved. In that case it is necessary that the theories do not have common symbols, which are interpreted differently (otherwise a renaming must be considered) [2]. Throughout this thesis, we will use the symbol $\oplus$ for a combination of disjoint theories (e.g. $\mathcal{T}_2 \oplus \mathcal{T}_2$ denotes a combined theory which consists of all axioms and symbols from both theories).

2.3.1 Common SMT theories

Equality with uninterpreted functions (EUF)

In the most general case, all interpretations for predicates and functions are allowed except for the interpretation of the equality predicate and functional consistency (the same function returns the same value given equal arguments). A formula consisting of conjunctions of ground EUF-formulas is decidable in polynomial time using a procedure called *congruence closure* [2].

Linear integer arithmetic (LIA)

Linear integer arithmetic includes EUF and introduces binary predicates $<, >, \leq$ and $\geq$ as well as binary functions $+$ and $-$ and symbols from $\mathbb{Z}$ (e.g. 0, 1, -5 etc.) using their usual interpretations. The satisfiability of LIA-formulas is NP-complete. It is noteworthy that adding multiplication to linear arithmetic makes it undecidable in general [2].

Arrays

There have been several SMT array theories. For the encoding in Chapter 4, a theory with $store(\cdot)$ and $read(\cdot)$ operations will be necessary. In addition, it requires a constant combinator $\mathbf{K}$ and a map_f combinator. They have the following properties $\forall i \mathbf{K}(c)[i] = c$ respectively $\forall i map_f(a_1, ..., a_n)[i] = f(a_1[i], ..., a_n[i])$. A NP-complete decision procedure for such a theory has been proposed by L. de Moura and N. Bjørner [9].

Fixed width bit-vectors

Several theories for bit-vectors have been proposed. Typically bit-vectors are represented by constant symbols with a bit-width associated with each constant. The functions and predicates include bit-wise boolean and arithmetic operations. For non-trivial theories of bit-vectors decidability is NP-complete, as it can be easily reduced to SAT [2].

2.3.2 SMT-LIB and SMT

SMT-LIB is an initiative for research and development in SMT. The aims of the initiative include standard descriptions of theories, a common input (and output) language for SMT-solvers and to connect researchers, developers and users of SMT. They provide the

SMT-LIB language specification and organise frequent competitions (SMT-COMP) to drive the development and improvements of SMT-solvers [1].

SMT-LIB language

The SMT-LIB language describes the common input and output format used by most SMT-solvers. Each well-formed expression has a unique *sort* (known as a *type* in some other contexts). The only pre-defined sort is `Bool`, other sorts may be defined by specific theories [1].

All conforming SMT-solvers use S-expressions for their input and output. S-expressions are easily parsable as they only consist of simple tokens, as shown in Listing 2.2, and use prefix notation. Comments may be used and are preceded by the `;` character. Listing 2.3 shows some examples of S-expressions [1].

Custom function symbols are reserved keywords in SMT-LIB and may only be used in specific contexts. They have special semantics for the SMT solvers. In the following, we will describe selected function symbols, which will be used in the examples in the following chapters:

set-logic
: may be used as the first statement for specifying a logic family. E.g. `(set-logic QF_LIA)` is used to limit all further input to linear integer arithmetic symbols. The logic's classes available in SMT-LIB are shown in Table 2.1 while a full illustration of all standardised logics is depicted in Figure 2.1.

declare-fun
: is used for declaring a new function symbol or constant (a function without arguments). The following `(declare-fun x (Int Int) Bool)` declares a function `x` of sort `Bool` that takes two Int arguments.

define-fun
: similar to `declare-fun` but also provides a definition for the function. E.g. `(define-fun a ((x Int)) Bool (ite (= x 3) true false) )` defines a function `a` that takes an `Int` argument `x` and returns `true` if $x = 3$ or `false` otherwise.

assert
: is used to assert a top-level conjunct of the formula. E.g. `(assert (= x 3))` asserts $x = 3$.

check-sat
: is used to check satisfiability of asserted conjuncts. It prints `sat` if the formula is satisfiable, `unsat` if the formula is not satisfiable or `unknown` otherwise (the latter may occur with certain theories if quantifiers are used).

get-model
is used to obtain a satisfying model for the asserted conjuncts. The output uses the same syntax as the input. A possible output for the formula described in Listing 2.4 is shown in Listing 2.5.

```
true false                ; booleans
255 #xFF #xb11111111      ; numeral, hex-numeral and binary numeral
"hello world"             ; string
12.34 3.14156             ; decimals
x y f_n QF_LIA            ; symbols
```

Listing 2.2: Examples for tokens

```
( )                           ; an empty S-expression
123                           ; a single token
( 123 "hello" "world" )       ; an S-expression with 3 tokens
( + x ( - y z ) )             ; a nested S-expression
```

Listing 2.3: Examples for S-expressions

```
(set-logic QF_LIA)                ; specify logic family

(declare-fun x () Int)            ; declare constants of sort Int
(declare-fun y () Int)
(declare-fun z () Int)

(assert (> (- 2 x) (+ y z)))      ; assert top level conjuncts
(assert (= z 15))

(check-sat)                       ; check for satisfiability
(get-model)                       ; print the satisfying model
```

Listing 2.4: A possible SMT input

```
sat
(model
     (define-fun x () Int 0)
     (define-fun y () Int (- 14))
     (define-fun z () Int 15)
)
```

Listing 2.5: A possible SMT output

Table 2.1: Categories of theories available in SMT-LIB. They can be combined e.g. QF_UFBVA is the quantifier-free theory of bit-vector arithmetic with uninterpreted functions and arrays

Linear real arithmetic	LRA
Linear integer arithmetic	LIA
Mixed linear arithmetic	LIRA
Uninterpreted functions	UF
Array theories	A/AX
Bit-vector arithmetic	BV
Nonlinear real arithmetic	NRA
Nonlinear integer arithmetic	NIA
Mixed nonlinear arithmetic	NIRA
Quantifier-free	QF_

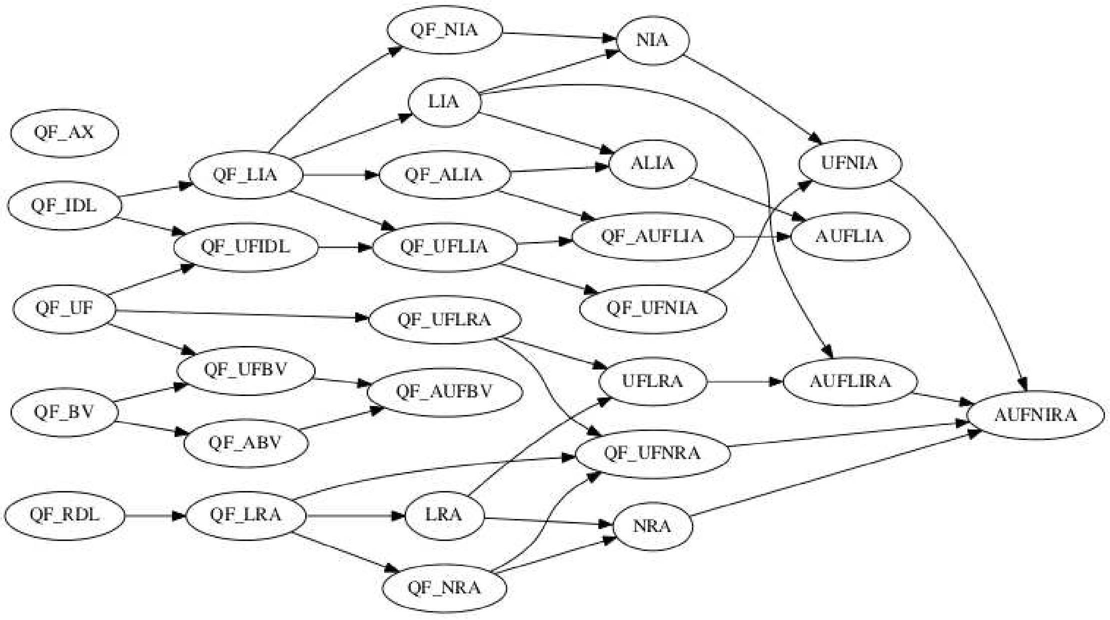


Figure 2.1: Standard SMT-LIB logics (image source SMT-LIB [1])

CHAPTER 3

Introduction to SL^*_{data}

3.1 Introduction to SL^*_{data}

This chapter is based on the paper which first proposed SL^*_{data} [11] by J. Katelaan et al., which we advise on reading for a complete and formal definition of the logic. We describe the syntax and semantics of SL^*_{data} in a more informal and intuitive manner. After that, we apply it to prove the example from the previous chapter (Listing 2.1).

SL^*_{data} is a Separation Logic with data. It aims to strike a balance between decidability and expressiveness. While many NP-hard fragments of Separation Logic only allow reasoning about the memory structure, SL^*_{data} also allows assertions about interesting data properties within the memory. For example, it is possible to model sorted lists or binary search trees, all while maintaining decidability in NP.

Other properties of SL^*_{data} include:

1. Boolean closure (with restrictions as this would otherwise lead to PSpace-hardness)
2. Support for list-segments and partial trees for defining heaps that locally violate shape or data properties
3. Per-field allocation to allow easy extension of SL^*_{data}

3.2 Syntax of SL^*_{data}

SL^*_{data} relies upon two background theories: a location theory $\mathcal{T}_{loc}$ and a data theory $\mathcal{T}_{dat}$. They are used for the location (i.e. *"memory address"*) and the data domain, respectively. Figure 3.1 shows the syntax of the core logic, where $\mathcal{F}_{loc}$ and $\mathcal{F}_{dat}$ denote quantifier free formulas from the theories $\mathcal{T}_{loc}$ and $\mathcal{T}_{dat}$, respectively.

$$\begin{array}{rcll}
t & := & \mathbf{null}_{\mathsf{tree}} \mid x \in \mathcal{X}_{tree} & \\
l & := & \mathbf{null}_{\mathsf{list}} \mid x \in \mathcal{X}_{list} & \\
d & := & x \in \mathcal{X}_{data} & \\
A_{Spatial} & ::= & t \rightarrow_l t \mid t \rightarrow_r t \mid t \rightarrow_d d \mid l \rightarrow_n l \mid l \rightarrow_d d & \text{Spatial atoms} \\
 & & \mid \mathsf{list}(l, \vec{s}) \mid \mathsf{tree}(t, \vec{s}) \mid \mathcal{F}_{\mathsf{loc}} \mid \mathcal{F}_{\mathsf{data}} & \\
F_{Spatial} & ::= & A_{Spatial} \mid F_{Spatial} * F_{Spatial} & \text{Spatial formulas} \\
F & ::= & F_{Spatial} \mid \neg F \mid F \vee F \mid F \wedge F & \mathrm{SL}^*_{data} \text{ formulas}
\end{array}$$

Figure 3.1: Syntax of the core Separation Logic SL^*_{data} with lists, trees, and data [11]

SL^*_{data} requires two sorts $\mathcal{S} = \{\mathsf{loc}, \mathsf{data}\}$ [11]. As we will later see, interpretations for locations of trees and lists need to be disjunct (except for **null**). For this reason, the loc sort is separated into $\mathsf{loc}_{\mathsf{list}}$ and $\mathsf{loc}_{\mathsf{tree}}$ and thus differs from the definition in [11]. It can be considered *syntactic sugar* to disallow some UNSAT-formulas on syntactic level. Ultimately, during the encoding we ensure the two refer to the same sort.

$\mathcal{X}_{list}$ and $\mathcal{X}_{tree}$ are countable sets of sort $\mathsf{loc}_{\mathsf{list}}$ and $\mathsf{loc}_{\mathsf{tree}}$ respectively, while $\mathcal{X}_{data}$ is a countable set of sort data. $\vec{s}$ denotes a vector $\langle s_1, ..., s_2 \rangle$ of variables from $\mathcal{X}_{list}$ or $\mathcal{X}_{tree}$ and $\vec{s_1} \cdot \vec{s_2}$ is used for the concatenation of two vectors.

Following the definition, it is apparent that boolean structure is not allowed in spatial formulas and negation can only occur on the top level of a spatial formula, but not at its atoms if those are connected by the separating conjunction $*$ (i.e. formulas of the form $F * \neg G$ are not allowed).

3.2.1 Spatial atoms

The spatial atoms include points-to predicates ($\rightarrow$) as well as two inductive predicates list and tree. The predicates have the following signatures:

$$\begin{array}{ll}
\rightarrow_n : \mathsf{loc}_{\mathsf{list}} \times \mathsf{loc}_{\mathsf{list}} \mapsto \mathsf{Bool} & \rightarrow_d : \mathsf{loc} \times \mathsf{data} \mapsto \mathsf{Bool} \\
\rightarrow_l : \mathsf{loc}_{\mathsf{tree}} \times \mathsf{loc}_{\mathsf{tree}} \mapsto \mathsf{Bool} & \rightarrow_r : \mathsf{loc}_{\mathsf{tree}} \times \mathsf{loc}_{\mathsf{tree}} \mapsto \mathsf{Bool} \\
\mathsf{list} : \mathsf{loc}_{\mathsf{list}} \times \mathsf{loc}^*_{\mathsf{list}} \mapsto \mathsf{Bool} & \mathsf{tree} : \mathsf{loc}_{\mathsf{tree}} \times \mathsf{loc}^*_{\mathsf{tree}} \mapsto \mathsf{Bool}
\end{array}$$

For abbreviation we use $x \rightarrow_{f_1,...,f_n} (y_1, ..., y_n)$ instead of $x \rightarrow_{f_1} y_1 * ... * x \rightarrow_{f_n} y_n$ (e.g. $x \rightarrow_{n,d} (y, x_{data})$ instead of $x \rightarrow_n y * x \rightarrow_d x_{data}$), $\mathsf{list}(x)$ and $\mathsf{tree}(x)$ instead of $\mathsf{list}(x, \mathbf{null})$ and $\mathsf{tree}(x, \mathbf{null})$, respectively.

3.2.2 Data predicates

What is not shown in Figure 3.1 is the fact that the inductive predicates can be parameterised by so called *data predicates*. There are two types of data predicates: unary and binary predicates. Unary data predicates are formulas of the form $P(\alpha)$ and the binary data predicates are of the form $(f, P(\alpha, \beta))$ where $f \in \{\mathsf{n}, \mathsf{l}, \mathsf{r}\}$. Both can contain other variables from $\mathcal{X}_{loc}$ (for $f = \mathsf{n}$) respectively $\mathcal{X}_{tree}$ (for $f \in \{\mathsf{l}, \mathsf{r}\}$). Both list and tree can include data predicates as arguments.

Table 3.1 shows examples of valid formulas.

3.3 Semantics of SL^*_{data}

To be able to interpret SL^*_{data} formulas we must first define a heap interpretation. A heap interpretation is a sorted set $\mathcal{X}$ of of location values (of sort loc) including a constant **null**. Each location can relate to one or multiple other locations, which defines the heap's structure. In addition, every location (except **null**) can point to a value of the data-sort. List locations (or nodes) relate to other list locations via the point-to predicate *next* ($\rightarrow_n$). Tree locations (or nodes) relate to other tree locations via the point-to predicates *left* and *right* ($\rightarrow_l, \rightarrow_r$). Both can relate to data via the point-to predicate *data* ($\rightarrow_d$), but no location can relate to both a tree and a list location at the same time. Note that nodes are not always fully allocated. It is possible for a heap interpretation to be incomplete, i.e a node can only have a *next* successor without a *data* successor, also a node can only have a *left* successor without a *right* successor. This way we can have a model that violates list and tree properties locally. Figure 3.2 shows a simple example of such a heap.

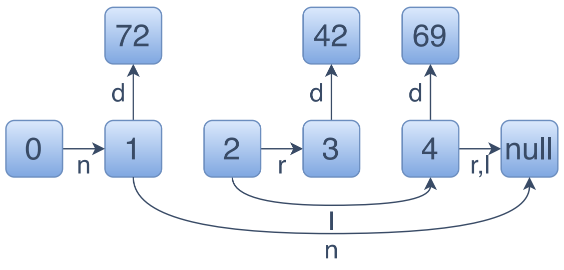


Figure 3.2: Example of a heap

SL^*_{data} formulas fulfill the following semantical rules:

location and data formulas: F_{loc}, F_{dat}
Sub-formulas from the two theories $\mathcal{T}_{loc}$ or $\mathcal{T}_{dat}$ are interpreted as usual in those theories, with the additional constraint that they describe an empty heap.

Conjunction: $F \wedge G$
A heap interpretation $\mathcal{M} \models F \wedge G$ iff $\mathcal{M} \models F$ and $\mathcal{M} \models G$.

Disjunction: $F \vee G$
A heap interpretation $\mathcal{M} \models F \vee G$ iff $\mathcal{M} \models F$ or $\mathcal{M} \models G$.

Negation: $\neg F$

A heap interpretation $\mathcal{M} \models \neg F$ iff $\mathcal{M} \not\models F$.

separating conjunction: $F * G$

The separating conjunction is very similar to a usual conjunction. If we find a heap interpretation $\mathcal{M} \models F * G$, then (like with conjunction) both $\mathcal{M} \models F$ and $\mathcal{M} \models G$ are true as well. However, it *separates* the heap into two disjoint heaps such that $\mathcal{M} = \mathcal{M}_f \oplus \mathcal{M}_g$ while $\mathcal{M}_f \models F$ and $\mathcal{M}_g \models G$. By $\mathcal{M}_f \oplus \mathcal{M}_g$ we denote a (disjoint) combination model combination, which consists of all locations and data from both models.

locations

Locations are any possible value of sort loc interpreted by the means of $\mathcal{T}_{loc}$. In the heap, one can think of them as a *memory address* that has the potential to be a tree or list node and can contain data (with the exception of **null**).

null ***location***

null is a dedicated constant of sort loc that is both a tree and a list node. However, **null** itself must not point to any other location nor data (**null** $\rightarrow_f x$ is unsatisfiable).

data

Data is any possible value of sort data interpreted by the means of $\mathcal{T}_{dat}$.

points-to-next: $x \rightarrow_n y$

For a location x, the points-to-next predicate indicates that x is a list-location and the next location of that list is y. x can not point to two different list nodes i.e. $x \rightarrow_n y \wedge x \rightarrow_n z \implies y = z$. In addition, x can not point to a tree node via $\rightarrow_l$ or $\rightarrow_r$.

points-to-left and points-to-right: $x \rightarrow_l y$**,** $x \rightarrow_r z$

For a location x, the points-to-left (or points-to-right) predicate indicates that x is a tree node and the sub-tree to the left is y (respectively to the right is z). x can not point to two different tree nodes via the same predicate i.e. $x \rightarrow_l y \wedge x \rightarrow_l z \implies y = z$ and $x \rightarrow_r y \wedge x \rightarrow_r z \implies y = z$. In addition, x can not point to a list node via $\rightarrow_n$.

points-to-data: $x \rightarrow_d d$

A tree- or list-location x can point to data d but can not point to two different data points i.e. $x \rightarrow_d a \wedge x \rightarrow_d b \implies a = b$.

list: $\mathsf{list}(x, y)$

$\mathsf{list}(x_0, y)$ expresses that: (1) x_0 is a list node starting at x_0, (2) there is a cycle free, possibly empty, path $\{x_0, ..., x_i, y\}$ following $\rightarrow_n$ predicates, and (3) each location along the way also has an allocated data point i.e. for each of the x_i, $x_i \rightarrow_d d$ is true for some data point d. Note that there are no claims about y itself in particular y can be equal to **null**.

Table 3.1: Examples for SL^*_{data} formulas

Formula	Informal description	Status	Model
$list(x, z, \{\mathsf{n}, \alpha \leq \beta\})$	The heap consists of a sorted list segment from x to z	SAT	$\mathcal{M}_1$
$x \to_{l,r,d} (y, z, d) * (d > 0)$	The heap consists of a tree node x that points to two other nodes y and z, and x points to a data value d that is greater than 0	SAT	$\mathcal{M}_2$
$tree(x, \{(\mathsf{l}, \alpha \leq \beta), (\mathsf{r}, \alpha > \beta), \})$	The heap consists of a binary search tree	SAT	$\mathcal{M}_3$
$tree(x) \land true$	The heap consists of a tree with root node x and at the same time it is empty	UNSAT	-

tree: $\mathsf{tree}(x, \vec{s})$

$\mathsf{tree}(x_0, y)$ denotes that: (1) x_0 is a tree node starting at x_0, (2) there is a cycle free, possibly empty, path $\{x_0, ..., x_i, s\}$ to each of the stop points (i.e. leaf nodes) $s \in \vec{s}$ following $\to_l$ and pto_r predicates, and (3) each location along the way also has an allocated data point i.e. for each of the x_i, $x_i \to_d d$ is true for some data point d. Again, there are no claims about the leaf nodes themselves.

unary data predicates: $\mathsf{tree}(x, \vec{s}, \{\mathcal{P}(\alpha)\})$ **resp.** $\mathsf{list}(x, \vec{s}, \{\mathcal{P}(\alpha)\})$

Inductive predicates with unary data predicates are interpreted like ones without, but they constrain the data according to the predicate $\mathcal{P}(\alpha)$. For each data point d along the path, $\mathcal{P}(d)$ must hold true, while $\mathcal{P}$ is interpreted by the means of the theories $\mathcal{T}_{loc} \oplus \mathcal{T}_{dat}$. If there is more than one predicate, all of them must hold true.

binary data predicates: $\mathsf{tree}(x, \vec{s}, \{f, \mathcal{P}(\alpha, \beta)\})$ **resp.** $\mathsf{list}(x, \vec{s}, \{\mathcal{P}(\alpha, \beta)\})$

Binary data predicates are interpreted similarly to unary data predicates. However, they include a *pointer selector* f. For each two nodes along the path $x \to_f y$ where $x \to_d d_x$ and $y \to_d d_y$, the predicate $\mathcal{P}(d)$ must hold true. Like for unary predicates, $\mathcal{P}$ is interpreted by the means of the theories $\mathcal{T}_{loc} \oplus \mathcal{T}_{dat}$ and if there are more than one predicate all of them must hold true.

Table 3.1 shows some examples and in Figure 3.3 heap interpretations that satisfy them, while Figure 3.4 shows some invalid heaps.

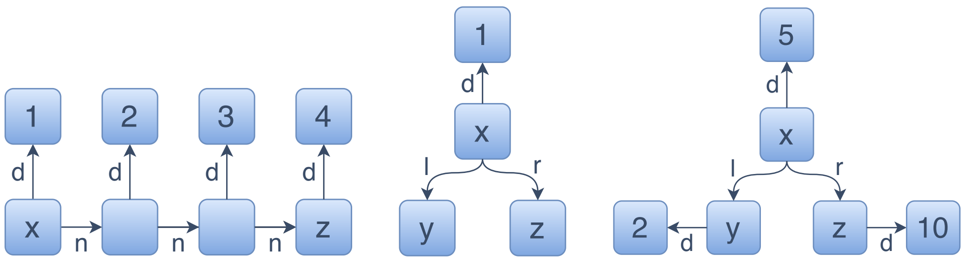
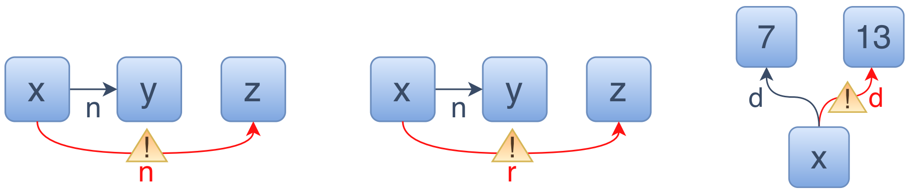


Figure 3.3: Heap interpretations



(a) two descendants (b) tree and list descendants (c) double allocation

Figure 3.4: Invalid heap interpretations

3.4 Complexity Results for SL$^*_{data}$

As mentioned previously, general Separation Logic is undecidable in general, so it remains to back up the statement about SL$^*_{data}$'s complexity. Recall that we claimed that the satisfiability problem of SL$^*_{data}$ is NP-hard. In the next chapter (Chapter 4), we will introduce a reduction to SMT using the theories $\mathcal{T}_{array} \oplus \mathcal{T}_{loc} \oplus \mathcal{T}_{dat}$. Since $\mathcal{T}_{array}$ is decidable in NP [9], we can choose the data and location theory carefully such that $\mathcal{T}_{loc} \oplus \mathcal{T}_{dat}$ is also decidable in NP. This is the case for *linear arithmetic* (e.g. $\mathcal{T}_{loc} = \mathcal{T}_{dat} = LIA$).

It remains to show that the reduction is linear in size. This is also the case, because of the small model property of SL$^*_{data}$ [11]. Assume there is a heap interpretation that models a satisfiable SL$^*_{data}$ formula. Furthermore assume that formula has n_{list} list variables, n_{tree} tree variables, m_{list} list predicates with data constraints, m_{tree} tree predicates with data constraints and a maximum of $k \geq 1$ leaf nodes per tree predicate. The small model property tells us, there is a satisfying heap interpretation $\mathcal{M}$ that is linear in size i.e. $|\mathcal{M}| \leq max(4, 2n_{list} + (3 + k)n_{tree} + 2m_{list} + m_{tree})$.

For the entailment in SL$^*_{data}$ consider the following:

$$\begin{aligned}
&F \models G && \iff \\
&\mathit{true} \models F \implies G && \iff \\
&F \implies G \quad \mathit{valid} && \iff \\
&\neg F \lor G \quad \mathit{valid} && \iff \\
&F \land \neg G \quad \mathit{unsat} &&
\end{aligned}$$

$F \land \neg G$ is a permissible SL^*_{data} formula and we can encode it in SMT using the encoding from the next chapter. Since we need to negate the answer, the entailment problem lies in coNP.

3.5 Symbolic Execution with SL^*_{data}

Now that we know the semantics and syntax of SL^*_{data} we can illustrate a practical use of SL^*_{data} by proving that the algorithm from 2.1 is partially correct and indeed computes the maximum of a linked list.

First, we now know how to express "The argument **a** is a valid list", since we introduced a predicate for it: it is simply $\mathsf{list}(a)$. Exploiting data predicates, we can express "No element within **a** is greater than **m**" as well: $\mathsf{list}(a, \{\alpha \leq m\})$.

Next, we establish a few Hoare triplet rules. We already know the frame rule:

$$\frac{\{F\}\ c\ \{G\}}{\{F * \phi\}\ c\ \{G * \phi\}}$$

Note the traditional Hoare Logic rules for while and if:

$$\frac{\{I \land e\}\ p\ \{I\}}{\{I\}\ \mathsf{while}\ e\ \mathsf{do}\ p\ \mathsf{od}\ \{I \land \neg e\}}\ (\mathrm{wh})$$

$$\frac{\{F \land e\}\ p\ \{G\} \qquad F \land \neg e \implies G}{\{F\}\ \mathsf{if}\ e\ \mathsf{then}\ p\ \mathsf{fi}\ \{G\}}\ (\mathrm{if})$$

If we disallow heap manipulations within evaluation of the boolean expression e, we can simply replace the boolean conjunction with the separating conjunction to obtain the Separation Logic counterparts of those rules:

$$\frac{\{I * e\}\ p\ \{I\}}{\{I\}\ \mathsf{while}\ e\ \mathsf{do}\ p\ \mathsf{od}\ \{I * \neg e\}}\ (\mathrm{wh}^*)$$

$$\frac{\{F * e\}\, p\, \{G\} \qquad F * \neg e \implies G}{\{F\}\ \mathsf{if}\ e\ \mathsf{then}\ p\ \mathsf{fi}\ \{G\}}\ (\text{if}^*)$$

Similarly, we can say about the forward assignment rule:

$$\frac{}{\{F\}\ x = e\ \{\exists x'.F[x/x'] \land x = e[x/x']\}}\ (\text{as}\downarrow)$$

The rule is sound as long as the assignment x' does not occur in F and e. If we wanted to replace the conjunction with the separating conjunction, by the semantics of $*$ we only need to ensure that $F[x/x']$ and $e[x/x']$ does not have an alias to x. Since we replaced every occurrence of x with x', this can only occur for pointer arithmetics i.e. the statement is of the form $x = f(x)$. Therefore the rule

$$\frac{}{\{F\}\ x = e\ \{\exists x'.F[x/x'] * x = e[x/x']\}}\ (\text{as}^*\downarrow)$$

is sound as well, but only if the right hand expression does not contain the memory location x that is being assigned.

Now it remains to formalise the semantics of the heap manipulating sub-routines that are used within the algorithm.

head(x)

If x is an allocated list location, head will return the data that list location is pointing to. If x is **null** or not a list location it will act like an abort statement. Formally, we can express that as the Hoare triplet

$$\frac{}{\{list(x) * F\}\ d = \mathsf{head}(x)\ \{\exists d'.F[d/d'] * x \rightarrow_{n,d} (x_{tail}, d) * \mathsf{list}(x_{tail})\}}\ (\text{head}\downarrow)$$

for partial correctness. For total correctness, we need to ensure x is a valid list location and $x \neq \textbf{null}$ as well:

$$\frac{}{\{\mathsf{list}(x) * x \neq \textbf{null} * F\}\ d = \mathsf{head}'(x)\ \{\exists d'.F[d/d'] * x \rightarrow_{n,d} (x_{tail}, d) * \mathsf{list}(x_{tail})\}}\ (\text{head}\downarrow)$$

Note that every data predicate P, $P[d/d']$ needs to be preserved in the $\mathsf{list}(x_{tail})$ call as well. This is omitted for brevity. Further, it is also sound that for every P, $P(d)[d/d']$ needs to be true. However, omitting it is admissible, since that only weakens the post condition and is sound by the rule of logical consequence. Binary predicates are not used in the proof, but similar constraints might be required. For brevity we also use the predicate head(x) in assertions. It is simply a placeholder for the value that head(x) evaluated to during the execution. Note that it is a pure function i.e. it will always evaluate to the same value for a fixed heap for the same argument.

next(x)

If x is an allocated list location, next will return the next list location x is pointing to. If x is **null** or not a list location, it will act like an abort statement. Formally, we can express that as the Hoare triplet

$$\frac{}{\{\mathsf{list}(x) * F\}\ v = \mathsf{next}(x)\ \{\exists v', d.F[v/v'] * x \rightarrow_{n,d} (v, d) * \mathsf{list}(v)\}}\ (\text{next}\downarrow)$$

for partial correctness and for some x_{dat} that does not occur in F. For total correctness, we need to ensure that x is a valid list location and $x \neq \mathbf{null}$:

$$\frac{}{x \neq \mathbf{null} * \{\mathsf{list}(x) * F\}\ v = \mathsf{next}(x)\ \{\exists v', d.F[v/v'] * x \rightarrow_{n,d} (v, d) * \mathsf{list}(v)\}}\ (\text{next}\downarrow)$$

Again, we need to take care of data predicates and every data predicate P needs to be preserved as $P[v/v']$ in the $\mathsf{list}(x_{tail})$ call as well. Omitting the validity for skipped locations is admissible as well, since it again only weakens the post condition and is sound by the rule of logical consequence.

Finally, we have the means for proving the *max* sub-routine. We can annotate the program using *annotation calculus* according to the Hoare rules we defined:

```
function max(a)
   $\{\,\mathsf{list}(a)\,\}$
   b = a
   $\{\,\exists b'\ .\ \mathsf{list}(a) * a = b\,\}$     (as*)↓
   $\{\,\mathsf{list}(a) * a = b\,\}$     (lc)
   m = head(a)
   $\{\,\exists m'\ .\ \mathsf{list}(a_{tail}) * a \rightarrow_{n,d} (a_{tail}, m) * a = b\,\}$     (head)↓
   $\{\,1\colon \mathsf{list}(a_{tail}) * a \rightarrow_{n,d} (a_{tail}, m) * a = b\,\}$     (lc)
   $\{\,2\colon \mathsf{list}(a, b, \{\alpha \leq m\})\,\}$     (wh*)
   while $b \neq \mathbf{null}$ do
      $\{\,\mathsf{list}(a, b, \{\alpha \leq m\}) * b \neq \mathbf{null}\,\}$     (wh*)
      b = next(b)
      $\{\,3\colon \exists b', d\ .\ \mathsf{list}(a, b', \{\alpha \leq m\}) * b' \neq \mathbf{null} * b' \rightarrow_{n,d} (b, d))\,\}$     (next)↓
      if $head(b) > m$ then
         $\{\,\exists b', d\ .\ \mathsf{list}(a, b', \{\alpha \leq m\}) * b \neq \mathbf{null} * b' \rightarrow_{n,d} (b, d) * \mathsf{head}(b) > m\,\}$     (if*)
         m = head(b)
         $\{\,4\colon \exists b', d, m'\ .\ \mathsf{list}(a, b', \{\alpha \leq m'\}) * b \neq \mathbf{null} *$
            $b' \rightarrow_{n,d} (b, d) * \mathsf{head}(b) > m' * m = \mathsf{head}(b)\}$     (as*)↓
         $\{\,5\colon \mathsf{list}(a, b, \{\alpha \leq m\})\,\}$     (fi*)↑
      fi
      $\{\,5\colon \mathsf{list}(a, b, \{\alpha \leq m\})\,\}$     (wh*)
   od
   $\{\,6\colon \mathsf{list}(a, b, \{\alpha \leq m\}) * b = \mathbf{null}\,\}$     (wh*)
   $\{\,7\colon \mathsf{list}(a, \{\alpha \leq m\})\,\}$
   return m
end
```

It remains to show the entailments of several assertion pairs.

$1 \implies 2$

The validity of this implication seems to be true intuitively. However, it requires the formal semantics of the list predicate to be proven. We omit the proof here but use our decision procedure to show its correctness. It is part of the benchmarks in the Results Results. We used the input $1 \wedge \neg 2$ which was proven UNSAT by our tool and thus proves the validity of the entailment. The formula is named *entailment-max* and can be found in the Appendix in Section 8.2.

$3 * \mathbf{head}(b) \leq m \implies 5$ **(premise for if*-rule)**

$$\exists b', d\ .\ \mathsf{list}(a, b', \{\alpha \leq m\}) * b' \neq \mathbf{null} * b' \rightarrow_{n,d} (b, d)) * \mathsf{head}(b) \leq m \implies$$
$$\exists b', d\ .\ \mathsf{list}(a, b', \{\alpha \leq m\}) * b' \rightarrow_{n,d} (b, d)) \implies$$
$$\mathsf{list}(a, b, \{\alpha \leq m\})$$

$4 \implies 5$

$$\begin{aligned}
&\exists b', d, m' \;.\; \mathsf{list}(a, b', \{\alpha \leq m'\}) * b \neq \mathbf{null} \\
&\qquad * b' \rightarrow_{n,d} (b, d) * \mathsf{head}(b) > m' * m = \mathsf{head}(b) \implies \\
&\exists b', d, m' \;.\; \mathsf{list}(a, b', \{\alpha \leq m'\}) * b \neq \mathbf{null} * b' \rightarrow_{n,d} (b, d) * m > m' \implies \\
&\exists b', d \;.\; \mathsf{list}(a, b', \{\alpha \leq m\}) * b \neq \mathbf{null} * b' \rightarrow_{n,d} (b, d) \implies \\
&\exists b', d \;.\; \mathsf{list}(a, b, \{\alpha \leq m\}) \implies \\
&\mathsf{list}(a, b, \{\alpha \leq m\})
\end{aligned}$$

$6 \implies 7$

$$\begin{aligned}
&\mathsf{list}(a, b, \{\alpha \leq m\}) * b = \mathbf{null} \implies \\
&\mathsf{list}(a, \mathbf{null}, \{\alpha \leq m\}) \implies \\
&\mathsf{list}(a, \{\alpha \leq m\})
\end{aligned}$$

This proves partial correctness of the algorithm. The proof for total correctness is very similar, but requires a weaker precondition $\mathsf{list}(a) * a \neq \mathbf{null}$ because of the assignment $m = \mathsf{head}(a)$ without a **null**-check. Also, we would need to prove the termination of the while loop. This is simple, since we know that the list must be cycle free (by semantics of list) and in each body execution the list segment is growing by one, so at some point, it will reach the end of the list ($\mathsf{list}(a, b) * b = \mathbf{null}$) and since $b \neq \mathbf{null}$ is the loop condition, the loop will terminate.

CHAPTER 4

Encoding SL^*_{data} to SMT

4.1 Basics

This chapter is based on the encoding from the SL^*_{data} paper [11]. Again, it describes the SMT encoding in an more informal and intuitive manner.

The basic idea of encoding SL^*_{data} to SMT is to define partial functions that map the relations between the locations. Since all functions in SMT solvers are total functions, we need to encode the partial domains of the functions. This is where the aforementioned array theory $\mathcal{T}_{array}$ comes into play. Along with the usual $store(\cdot)$ and $read(\cdot)$ operations, we require a constant combinator $\mathbf{K}$ and a map_f combinator for which $\mathbf{K}(c)[i] = c$ respectively $\mathsf{map}_f(a_1, ..., a_n)[i] = f(a_1[i], ..., a_n[i])$ holds.

Using those operations, we can encode a set (and thus also a domain of a function). A set is then simply an infinite array of boolean values whose indices are of sort loc. We can define set operations as follows:

Empty set:
An empty set is simply the constant $false$ array i.e. $\mathbf{K}(false)$. We can define an $\mathsf{isempty}(X)$ predicate for an array X as $X = \mathbf{K}(false)$.

Adding elements:
Adding elements to an array can be done via the $store$ operation. $X \cup \{x\} := store(X, x, true)$ (i.e. "set $X[x]$ to $true$"). In particular, a single element set can be defined as $\{x\} := store(\mathbf{K}(false), x, true)$.

Membership check:
We can easily check if $x \in X$ by evaluating the array at that point $read(X, x)$ ($X[x]$ for short).

Set conjunction and disjunction:
By using the map operator, we can define both conjunction and disjunction, namely $X \cup Y := \mathsf{map}_\vee(X, Y)$ and $X \cap Y := \mathsf{map}_\wedge(X, Y)$.

Subset relations:
We can also check easily if a set is a subset by mapping implication and checking if the result is the constant *true* array: $X \subseteq Y := \mathsf{map}_\Rightarrow(X, Y) = \mathbf{K}(true)$.

In the following, we will use the set notations as a brief notation for the above encodings. In addition, a big-letter variable X will represent a set, an arrow over the variable $\vec{x}$ will represent a vector (a big letter vector $\vec{X}$ is a vector of sets). We will use predicates with vectors that are defined for scalars in a point-wise manner e.g.

$$\mathsf{isempty}(\vec{X}) \iff \bigwedge_{X \in \vec{X}} \mathsf{isempty}(X)$$

and

$$\vec{X} = \vec{Y} \cup \vec{Z} \iff \bigwedge_{i=1..|\vec{X}|} X_i = Y_i \cup Z_i$$

(for binary and n-ary predicates all vectors are expected to have equal dimensions).

For a heap interpretation of size N (e.g. obtained by the small model theorem), we encode the N possible locations as $x_1, ..., x_N$ of sort loc. For each of the possible heap relations ($\mathsf{n}, \mathsf{l}, \mathsf{r}$ and d), we define a partial function i.e. f_n, f_l, f_r and f_d with associated domains X_n, X_l, X_r and X_d. We also define a set of locations X such that $X = X_n \cup X_l \cup X_r \cup X_d$ and $X \subseteq x_1, ..., x_N$.

We can now encode the heap interpretation by the following formula:

$$\Delta_{SL}^N := (X = X_n \cup X_l \cup X_r \cup X_d) \wedge X \subseteq \{x_1, ..., x_N\} \wedge \mathbf{null} \not\in X \wedge \mathsf{isempty}(X_n \cap (X_l \cup X_r))$$

Δ_{SL}^N makes sure that:

1. the heap has at most size N: $X \subseteq \{x_1, ..., x_N\}$
2. **null** is not allocated: $\mathbf{null} \not\in X$
3. no location is a tree and a list node at the same time: $\mathsf{isempty}(X_n \cap (X_l \cup X_r))$

Figure 4.1 shows an example for a model of Δ_{SL}^N and how it relates to a heap interpretation.

$$X = \{0,1,2,3,4\} \quad X_n = \{0\} \quad X_l = \{2\} \quad X_r = \{2,4\} \quad X_d = \{1,3,4\}$$

$$f_n(x) = \begin{cases} 1 & x = 0 \\ 5 & x = 1 \\ undef. & else \end{cases} \qquad f_l(x) = \begin{cases} 4 & x = 2 \\ 5 & x = 4 \\ undef. & else \end{cases}$$

$$f_r(x) = \begin{cases} 3 & x = 2 \\ 5 & x = 4 \\ undef. & else \end{cases} \qquad f_d(x) = \begin{cases} 72 & x = 1 \\ 42 & x = 3 \\ 69 & x = 4 \\ undef. & else \end{cases}$$

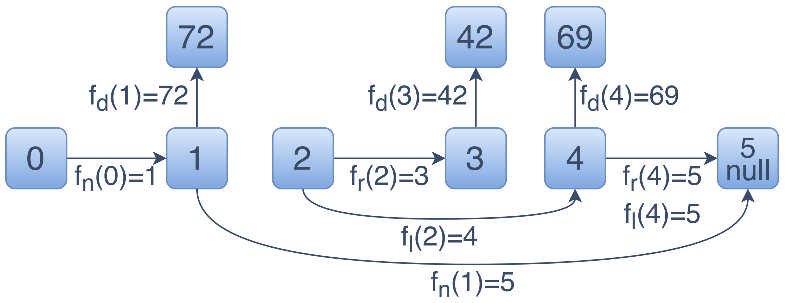


Figure 4.1: A graphical representation of an SMT encoding of the heap depicted in Figure 3.2

4.2 Encoding of Simple Formulas

In the following, we will use $\vec{X} = \langle X_n, X_l, X_r, X_d \rangle$ and will refer to it as the *footprint* of a formula.

The encoding of a spatial formula F is defined by a translation function T_N that yields a formula pair $\langle A, B \rangle$. A captures the structure of F while B uses the set operations we defined in the previous section to keep track of the domains of the points-to relatioins, i.e the footprint of the sub-formula. T_N is defined recursively on the structure of F. It relies on two helper functions: T^s_N for spatial and T^b_N for boolean sub-formula. The result of the encoding is then defined as $F_{SMT} := A \wedge B \wedge \Delta^N_{SL}$ where $\langle A, B \rangle = T_N(F)$. F_{SMT} is satisfiable if and only if the SL$^*_{data}$ formula F is satisfiable.

Location and data subformula

For a location formula F_{loc} or a data formula F_{dat}, the semantics of SL$^*_{data}$ dictate an empty heap:

$$T_N^s(F_{loc}, \vec{Y}) = \langle F_{loc}, \mathsf{isempty}(\vec{Y}) \rangle$$
$$T_N^s(F_{dat}, \vec{Y}) = \langle F_{dat}, \mathsf{isempty}(\vec{Y}) \rangle$$

Point-to predicates

For the point-to predicates, we know the result of their partial function, as well as their domain. The domain consists of a single element for that particular function, while all the others are empty.

$$T_N^s(x \rightarrow_n y, \vec{Y}) = \langle f_n(x) = y, Y_n = \{x\} \wedge \mathsf{isempty}(\vec{Y} \setminus \{Y_n\}) \rangle$$
$$T_N^s(x \rightarrow_l y, \vec{Y}) = \langle f_l(x) = y, Y_l = \{x\} \wedge \mathsf{isempty}(\vec{Y} \setminus \{Y_l\}) \rangle$$
$$T_N^s(x \rightarrow_r y, \vec{Y}) = \langle f_r(x) = y, Y_r = \{x\} \wedge \mathsf{isempty}(\vec{Y} \setminus \{Y_r\}) \rangle$$
$$T_N^s(x \rightarrow_d y, \vec{Y}) = \langle f_d(x) = y, Y_d = \{x\} \wedge \mathsf{isempty}(\vec{Y} \setminus \{Y_d\}) \rangle$$

Separating conjunction

For two sub-formulas connected by a speparating conjunct $F_1 * F_2$, we know that both of their constraints A_1 and A_2 must still hold. In addition, their combined footprint is the union of the sub-footprints and by the semantics of the separator, they are disjoint. Thus, for two fresh footprint variables $\vec{Y_1}$ and $\vec{Y_2}$:

$$T_N^s(F_1 * F_2, \vec{Y}) = \langle A_1 \wedge A_2 \wedge \mathsf{isempty}(\vec{Y_1} \cap \vec{Y_2}), B_1 \wedge B_2 \wedge \vec{Y} = \vec{Y_1} \cup \vec{Y_2} \rangle$$
$$\text{where } \langle A_1, B_1 \rangle = T_N^s(F_1, \vec{Y_1}) \text{ and } \langle A_2, B_2 \rangle = T_N^s(F_2, \vec{Y_2})$$

Boolean conjunction

Boolean operands do not change anything about the footprint. Only the structure constrains are preserved:

$$T_N^b(F_1 \wedge F_2) = \langle A_1 \wedge A_2, B_1 \wedge B_2 \rangle$$
$$\text{where } \langle A_1, B_1 \rangle = T_N^b(F_1) \text{ and } \langle A_2, B_2 \rangle = T_N^b(F_2)$$

Boolean disjunction

Boolean operands do not change anything about the footprint. Either one of the structure constraints can be true:

$$T_N^b(F_1 \wedge F_2) = \langle A_1 \vee A_2, B_1 \wedge B_2 \rangle$$
$$\text{where } \langle A_1, B_1 \rangle = T_N^b(F_1) \text{ and } \langle A_2, B_2 \rangle = T_N^b(F_2)$$

Negation

Boolean operands do not change anything about the footprint. Just the structure constraints must be false:

$$T^b_N(\neg F) = \langle \neg A, B\rangle \text{ where } \langle A, B\rangle = T^b_N(F)$$

Top level conjuncts

For a fresh footprint variable $\vec{Y}$, we can constrain that its sub-footprint is equal to the footprint of the entire heap:

$$T^b_N(F) = \langle A \wedge \vec{X} = \vec{Y}, B\rangle \text{ where } \langle A, B\rangle = T^s_N(F, \vec{Y})$$

4.2.1 Encoding a simple formula by example

We will now encode the simple formula $F_{SL} = x \to_n y * y \to_n z$ using $\mathcal{T}_{loc} = \mathcal{T}_{dat} = LIA$. An obvious upper bound is 3, since there are no inductive predicates and 3 location variables used in the formula. Thus, $\Delta^N_{SL} := (X = X_n \cup X_l \cup X_r \cup X_d) \wedge X \subseteq \{x_1, x_2, x_3\} \wedge$ **null** $\notin X \wedge \mathsf{isempty}(X_n \cap (X_l \cup X_r))$

current sub-formula	encoding result	next sub-formulas
$T_N(x \to_n y * y \to_n z)$	$F_{SMT} := A_1 \wedge B_1 \wedge \Delta^N_{SL} \wedge \vec{Y_1} = \vec{X}$	$\langle A_1, B_1\rangle =$ $T^s_N(x \to_n y * y \to_n z)$
$T^s_N(x \to_n y * y \to_n z)$	$A_1 := A_2 \wedge A_3 \wedge \mathsf{isempty}(\vec{Y_2} \cap \vec{Y_3})$ $B_1 := B_2 \wedge B_3 \wedge \vec{Y_1} = \vec{Y_2} \cup \vec{Y_3}$	$\langle A_2, B_2\rangle = T^s_N(x \to_n y)$ $\langle A_3, B_3\rangle = T^s_N(y \to_n z)$
$T^s_N(x \to_n y)$	$A_2 := f_n(x) = y,$ $B_2 := Y_{2n} = \{x\} \wedge \vec{Y_2} \setminus \{Y_{2n}\}$	-
$T^s_N(y \to_n z)$	$A_3 := f_n(y) = z,$ $B_3 := Y_{3n} = \{y\} \wedge \vec{Y_3} \setminus \{Y_{3n}\}$	-

Since the formula only contains $\to_n$ predicates, one can shorten the formula. Since all the other footprint sets are empty, one could shorten it by assuming $X = X_n$ and $Y_i = Y_{in}$. This example expressed in the SMT-LIB file format as well as an output by the SMT-Solver Z3 can be found in the Appendix in subSection 8.2.

4.3 Encoding of Inductive Predicates

To encode the list and tree predicates, we first need to define helper predicates. These will help use ensure the semantics of both predicates are correctly simulated in the resulting formula. We will consider the predicates $\mathsf{tree}(x, \vec{s}, \mathcal{P})$ and $\mathsf{list}(x, \vec{s}, \mathcal{P})$. We also assume a fresh footprint $\vec{Y}$. $\vec{s}$ are the leaf nodes and $\mathcal{P}$ the unary and binary data predicates. Moreover, we will introduce a reachable location set Z as the set of all locations reachable from x.

The following is true for a node x if that node is a leaf node. Note that **null** is an implicit leaf node for all valid lists and trees. The affected nodes are depicted in Figure 4.2.

$$\mathsf{isleaf}(x) := x = \mathbf{null} \vee x = s_1 \vee ...x = s_n$$

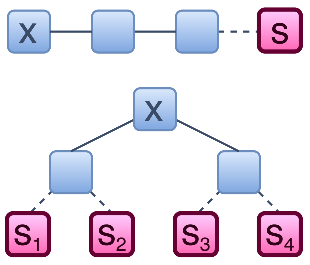


Figure 4.2: Leaf nodes as selected by $\mathsf{isleaf}(x)$ marked in bold red

The successor predicate is true if y is a direct successor of x. There are two variants, one for trees and one for lists.

$$S_{list}(x, y) := f_n(x) = y$$

$$S_{tree}(x, y) := f_l(x) = y \vee f_r(x) = y$$

The following will ensure that every location within the reachable set Z is fully allocated, i.e. every reachable tree location is allocated with respect to $\to_l, \to_r$ and $\to_d$ and every reachable list location is allocated with respect to $\to_n$ and $\to_d$.

$$\mathsf{defineY}_{list} := Y_n = Z \wedge Y_d = Z \wedge Y_r = \varnothing \wedge Y_l = \varnothing$$

$$\mathsf{defineY}_{tree} := Y_l = Z \wedge Y_r = Z \wedge Y_d = Z \wedge Y_r = \varnothing$$

The following helper predicate will define fresh reachability constraints $r_1(x_i, x_j)$ which are true for each direct successor pair x_i and x_j, where x_i is within the reachable set Z and x_j is not a leaf node of the inductive predicate.

$$R_1 := \bigwedge_{i,j \in [1..N]} r_1(x_i, x_j) \iff (x_i \in Z \wedge \neg\mathsf{isleaf}(x_j) \wedge S(x_i, x_j))$$

The R_K predicates will define fresh reachability constraints $r_K(x_i, y_j)$ which are true for a path starting in x_i and ending in x_j in K steps or less, where x_i is within the reachable set Z and x_j is not a leaf node of the inductive predicate.

$$R_K := \bigwedge_{i,j \in [1..N]} r_K(x_i, x_j) \iff (r_{K-1}(x_i, x_j) \vee \bigvee_{k \in [1..N]} (r_{K-1}(x_i, x_k) \wedge r_1(x_k, x_j))$$

It consists of two parts: either the reachability constraint is already true with one step less $(r_{K-1}(x_i, x_j))$ or there exists a x_k that is reachable from x_i by $K - 1$ steps and

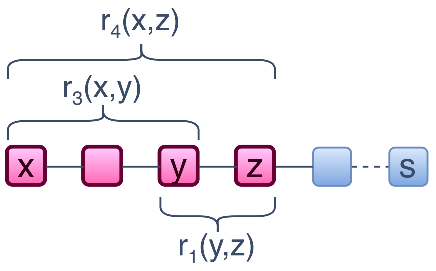


Figure 4.3: Depiction of a single reachability constraint $r_4(x,\cdot)$, selected nodes marked in bold red

x_j is reachable in the next step $(r_{K-1}(x_i, x_k) \wedge r_1(x_k, x_j))$. The existence quantifier is eliminated by applying a logical *or* on all possibilities. An example is depicted in Figure 4.3: $r_4(x, z)$ is true because $r_3(x, y)$ and $r_1(y, z)$ are true, while $r_4(x, y)$ is true because $r_3(x, y)$ is true.

The following predicate defines the reachability constraints used in later predicates.

$$\mathsf{reachability} := R_1 \wedge R_2 \wedge ... \wedge R_M$$

Note that in the original decision procedure [11], $M = N$ but this is not always necessary. It is possible to choose a bound per predicate, i.e. an upper bound N_{list} for list predicates and an N_{tree} for tree predicates. Depending on the formula they can differ from the upper bound N. The details for that are described in chapter Optimisations for the SL^*_{data} Decision Procedure in Section 5.1.

emptyZ is true when the footprint should be empty, this is the case when x itself is a leaf or not element of our heap.

$$\mathsf{emptyZ} := \mathsf{isleaf}(x) \vee x \notin \{x_1, ..., x_N\}$$

$\mathsf{footprint}$ ensures that Z is the set of locations reachable from x

$$\begin{aligned}\mathsf{footprint} :=\ & Z \subseteq x_1, ..., x_N \\ & \wedge (\mathsf{emptyZ} \implies Z = \varnothing) \\ & \wedge (\neg\mathsf{emptyZ} \implies \textstyle\bigwedge_{i\in[1..N]}(x_i \in Z) \iff (x_i = x \vee r_M(x, x_i)))\end{aligned}$$

$\mathsf{nobranch}$ dictates that a if node has two equal successors, then all of them must be **null**. This predicate only makes sense for structures with a branching structure, i.e. trees in SL^*_{data}.

$$\mathsf{nobranch} := \bigwedge_{i\in[1..N]} x_i \in Z \implies (f_l(x_i) = f_r(x_i) \implies f_l(x_i) = \textbf{null})$$

successor dictates that if two different nodes have the same successor, then it must be **null** as well. There are two variants, one for list and one for tree predicates:

$$\mathsf{successor}_{list} := \bigwedge_{i,j\in[1..N]} (x_i, x_j \in Z \wedge x_i \neq x_j) \implies (f_n(x_i) = f_n(x_j) \implies f_n(x_i) = \mathbf{null})$$

$$\begin{aligned}\mathsf{successor}_{tree} := \textstyle\bigwedge_{i,j\in[1..N]}(x_i, x_j \in Z \wedge x_i \neq x_j) \implies \quad & ((f_l(x_i) = f_l(x_j) \implies f_l(x_i) = \mathbf{null}) \\ & \wedge (f_l(x_i) = f_r(x_j) \implies f_l(x_i) = \mathbf{null}) \\ & \wedge (f_r(x_i) = f_l(x_j) \implies f_r(x_i) = \mathbf{null}) \\ & \wedge (f_r(x_i) = f_r(x_j) \implies f_r(x_i) = \mathbf{null}))\end{aligned}$$

Next, we ensure that the elements are not part of a cycle, and that no double allocations occur:

$$\mathsf{structure}_{list} := (\neg\mathsf{isleaf}(x) \implies x \in Z) \wedge \mathsf{successor}_{list} \wedge \neg r_M(x, x)$$

$$\mathsf{structure}_{tree} := (\neg\mathsf{isleaf}(x) \implies x \in Z) \wedge \mathsf{nobranch} \wedge \mathsf{successor}_{tree} \wedge \neg r_M(x, x)$$

For leaf nodes, we need to ensure that they are pairwise different, occur exactly once and are the only leaf nodes of the structure. Furthermore for trees, they are ordered as defined in the vector $\vec{s} = \langle s_1, ..., s_k \rangle$.

To ensure pairwise difference and if the root x is a leaf, then all leafs must be equal to the root we use:

$$\mathsf{leafseq} := \left(\mathsf{isleaf}(x) \implies \bigwedge_{s\in\vec{s}} x = s\right) \wedge \bigwedge_{1\leq i<j\leq k} s_i \neq s_j$$

To ensure leafs occur exactly once (i.e. no two elements of the leaf-vector $\vec{s}$ are equal) we use:

$$\mathsf{leafsoccur} := \neg\mathsf{isleaf}(x) \implies \bigwedge_{s\in\vec{s}} \bigvee_{i\in[1..N]} x_i \in Z \wedge S(x_i, s)$$

To ensure the leafs are the only leafs we use:

$$\mathsf{stopleaves}_{list} := (x_i \in Z \wedge f_n(x_i) \notin Z) \implies \mathsf{isleaf}(f_n(x_i))$$

$$\begin{aligned}\mathsf{stopleaves}_{tree} := \quad & (x_i \in Z \wedge f_l(x_i) \notin Z) \implies \mathsf{isleaf}(f_l(x_i)) \\ & \wedge\ (x_i \in Z \wedge f_r(x_i) \notin Z) \implies \mathsf{isleaf}(f_r(x_i))\end{aligned}$$

To assure the leafs are ordered, we define two helpers. fstop_l assures for a given x_i and leaf node s that s is either the direct left successor of x_i ($f_l(x_i) = s$) or it can be reached by going left first $\bigvee_{j\in[1..N]} r_M^Z(\mathsf{l}, x_i, x_j) \wedge x_j \in Z \wedge S(x_j, s)$. Similarly, fstop_r assures s is the direct right successor or can be reached by going right first. In other words, they assure s is in the left or right sub-tree of x_i.

$$r^Z_M(p, x, y) := (f_p(x) = y \vee (f_p(x) \in Z \wedge r_M(f_p(x), y)))$$

$$\mathsf{fstop}_l(x_i, s) := f_l(x_i) = s \vee \bigvee_{j \in [1..N]} r^Z_M(\mathsf{l}, x_i, x_j) \wedge x_j \in Z \wedge S(x_j, s)$$

$$\mathsf{fstop}_r(x_i, s) := f_r(x_i) = s \vee \bigvee_{j \in [1..N]} r^Z_M(\mathsf{r}, x_i, x_j) \wedge x_j \in Z \wedge S(x_j, s)$$

Now for each two subsequent leaf nodes s_i and s_{i+1}, we know there must exist a node where s_i is in the left and s_j is in the right sub-tree, namely their first common ancestor. We eliminate the existence by a disjunction over all possibilities and end up with:

$$\mathsf{leafsordered} := \bigwedge_{i \in [1..k-1]} \bigvee_{j \in [1..N]} x_i \in Z \wedge \mathsf{fstop}_l(x_j, s_i) \wedge \mathsf{fstop}_r(x_j, s_{i+1})$$

In Figure 4.4, we can see that x is the common ancestor of s_1 and s_2 and y is the common ancestor of s_2 and s_3. Indeed, s_1 is in the left sub-tree of x while s_2 is in the right one. The same is true for y, s_2 and s_3. However, we cannot find any node for which s_2 is in the left sub-tree, while s_1 is on the right one.

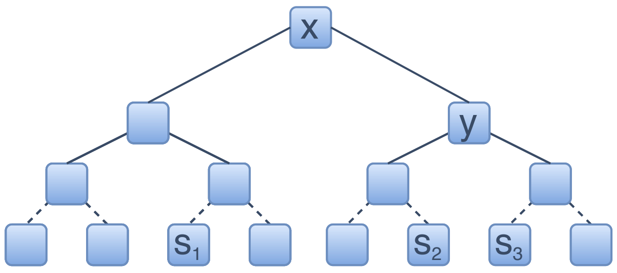


Figure 4.4: An illustration of a tree structure with three specified leaf nodes s_1, s_2 and s_3

The leaf constraints are combined in the leafs predicates: $\mathsf{leafs}_{list} := \mathsf{leafseq} \wedge \mathsf{leafsoccur} \wedge \mathsf{stopleaves}_{list}$ and $\mathsf{leafs}_{tree} := \mathsf{leafseq} \wedge \mathsf{leafsoccur} \wedge \mathsf{stopleaves}_{tree} \wedge \mathsf{leafsordered}$.

Next, we need to encode the data constraints as dictated by the data predicates $\mathcal{P}$.

For a single unary predicate $P \in \mathcal{P}$ we have:

$$\mathsf{udata}(P) := \bigwedge_{i \in [1..N]} x_i \in Z \implies P(f_d(x_i))$$

This can be shortened to two maps and a comparison if the partial function f_d is converted to an array.

$$\mathsf{udata}(P) := \mathsf{map}_{\Rightarrow}(Z, \mathsf{map}_P(f_d)) = \mathbf{K}(true)$$

For *next* constrained binary predicates, we have two elements x_i and x_j of the footprint Z. If x_j is a direct successor of x_i ($f_n(x_i) = x_j$) or is reachable by a direct successor ($r_M(f_n(x_i), x_j)$), then this implies that the predicate is true for the data behind those two elements $P(f_d(x_i), f_d(x_j))$.

$$\mathsf{bdata_n}(P) := \bigwedge_{i,j \in [1..N]} (x_i, x_j \in Z \wedge r_M^Z(n, x_i, x_j) \implies P(f_d(x_i), f_d(x_j))$$

Similarly, for *left* and *right* constrained predicates we have:

$$\mathsf{bdata_l}(P) := \bigwedge_{i,j \in [1..N]} (x_i, x_j \in Z \wedge r_M^Z(l, x_i, x_j) \implies P(f_d(x_i), f_d(x_j))$$
$$\mathsf{bdata_r}(P) := \bigwedge_{i,j \in [1..N]} (x_i, x_j \in Z \wedge r_M^Z(r, x_i, x_j) \implies P(f_d(x_i), f_d(x_j))$$

Combining the data predicate encodings we end up with:

$$\mathsf{data}_{list} := \bigwedge_{P \in \mathcal{P}_u} udata(P) \wedge \bigwedge_{\mathsf{n}P_n \in \mathcal{P}_n} \mathsf{bdata_n}(P_n)$$

$$\mathsf{data}_{tree} := \bigwedge_{P \in \mathcal{P}_u} udata(P) \wedge \bigwedge_{\mathsf{l}P_l \in \mathcal{P}_l} \mathsf{bdata_l}(P_l) \wedge \bigwedge_{\mathsf{r}P_r \in \mathcal{P}_r} \mathsf{bdata_r}(P_r)$$

Finally, we can put all the helper predicates together to define the translation of the predicates for a list:

$$\begin{aligned} T_N^s(\mathsf{list}(x, \vec{s}, \mathcal{P}), \vec{Y}) = \;& \langle A, B \rangle \text{ where} \\ & A = \mathsf{structure}_{list} \wedge \mathsf{leafs}_{list} \wedge \mathsf{data}_{list} \\ & B = \mathsf{reachability} \wedge \mathsf{footprint} \wedge \mathsf{defineY}_{list} \end{aligned}$$

and a tree:

$$\begin{aligned} T_N^s(\mathsf{list}(x, \vec{s}, \mathcal{P}), \vec{Y}) = \;& \langle A, B \rangle \text{ where} \\ & A = \mathsf{structure}_{tree} \wedge \mathsf{leafs}_{tree} \wedge \mathsf{data}_{tree} \\ & B = \mathsf{reachability} \wedge \mathsf{footprint} \wedge \mathsf{defineY}_{tree} \end{aligned}$$

Note that for both, the reachability constraints need to be fresh definitions, as they only make sense in the footprint Z we defined. Outside Z they, can be interpreted arbitrarily.

CHAPTER 5

Optimisations for the SL^*_{data} Decision Procedure

5.1 Encoding Size Reduction

Deciding the satisfiability of an SMT formula $F \in \mathcal{T}_{array} \oplus \mathcal{T}_{loc} \oplus \mathcal{T}_{dat}$ is NP hard. This means, that in the worst case, the run-time for the solver is exponential in time, with respect to the formula size. In other words, reducing the formula size a little bit can lead to a big impact on the overall run-time. Our central optimisations therefore aim to minimise the input size for the SMT solver. We have found several ways to do so:

1. One of the advantages of using Directed Acyclic Graph (DAG)s for storing the Abstract Syntax Tree (AST) is that all syntactically equivalent sub-formulas are represented by the same node. Technically, this means that when encoding an SL^*_{data} formula, we will encounter the same references for syntactically equivalent sub-parts of it. This allows several optimisations. If we cache the encoding per node, we can reuse the encoding of the identical sub parts. This reduces the number of footprint symbols in the formula.

2. If a formula does not contain any list and $\to_n$ predicates, this means that we do not need to keep track of the domain of f_n in the encoding. This means we can omit that part of the footprints (i.e. X_n and all helper Y_n). Analogously, if a formula lacks tree, $\to_l$ and $\to_r$ predicates, we can omit the domains of f_l and f_r.

3. The reachability constraints are a big portion of the encoding as each inductive predicate needs fresh constraints. If we split the bound calculation into one bound per data structure, the reachability constraints can have separate bounds too. For formulas describing both trees and lists, this can lead to significantly smaller formulas.

5.2 Spatial Equality Propagation

Modern SMT solvers keep track of equalities and implied equalities of formulas. Doing so helps reduce the size of the search space. Since we introduced a new spatial conjunction operator, equality propagation is limited as the solver has no deeper insight into the semantics of the operator. We therefore need to reintroduce the semantics of equalities within a spatial formula. In order to do so, the spatial equality propagation optimisation keeps track of equalities as an early step. It does so by creating bins of variables related by an equality. It then replaces the occurrences of variables by a single representative from the bins. As every unique location in the formula increases the formula footprint by one, every unique equality we find prevents this.

As an example, take a look at the following formula: $x \rightarrow_n y * \mathsf{list}(x) * x = y$. It is easy to see that, since both list locations are equal, the formula is in fact unsatisfiable because the separating conjuction implies disjunct heaps and in particular $x \neq y$. If we apply the above strategy to the formula we end up with $\mathsf{list}(x) * \mathsf{list}(x) * x = x$ and after a simplification step $\mathsf{list}(x) * \mathsf{list}(x)$.

Another example where equality propagation could help is $\mathsf{list}(x) \wedge \mathsf{list}(y) * x = y$. We end up with $\mathsf{list}(x) \wedge \mathsf{list}(x) * x = x$ and after simplification, we get $\mathsf{list}(x)$.

5.3 Stepwise Encoding

In many cases, it is possible a formula is unsatisfiable even without the axioms of a theory. E.g. if the boolean skeleton of the formula is unsatisfiable, then the whole formula cannot be satisfiable. In such cases, handling the specifics of a theory often only make the formula bigger and potentially increases run-time. This is even more true for encodings that can make the formula much bigger such as SL^*_{data}.

As an example, take a look at the following formula: $\mathsf{list}(x) * x = y \wedge \neg\mathsf{list}(y)$. It is easy to see that in order to hold true, both $\mathsf{list}(x)$ and $\neg\mathsf{list}(x)$ must be true and thus, the formula is unsatisfiable.

The fact is of course implied by the encoding. This, however, is not immediately apparent to the solver. Therefore, the entire encoding must be performed for it to find a conflict.

The idea behind stepwise encoding is to not perform the encoding in its entirety but leave out the most expensive parts of the encoding, namely the *list* and *tree* predicates, as uninterpreted function calls.

Since the reachability constraints required by those two predicates are no longer required, the resulting size of the encoded formula is much smaller and a conflict can be found much faster.

For the example above, in the first step our encoder would only produce an encoding of the separating conjunction and leave out the encodings for $\neg\mathsf{list}(y)$ and $\mathsf{list}(x)$. Because the encoding of the separating conjunction produces, among others, the top level conjuncts $\neg\mathsf{list}(y)$, $\mathsf{list}(x)$ and $x = y$ this step is already enough to prove the formula are unsatisfiable.

Following the same idea, can go further than that and try to encode the list and tree predicates iteratively: Instead of using the calculated upper bound for the footprint, we start at one and try to find a counterexample. If we cannot find one, we increase the footprint by one and try again until we find a counterexample or reach the upper bound for the footprint (because only then, we can guarantee there will be no conflict by increasing the bound further).

For example, take the formula $\mathsf{list}(x) * \mathsf{list}(y) * \mathsf{list}(z)$. The first step would only encode the separating conjunction. This time that does not help us to prove that the formula is unsatisfiable. Next we would try to encode the list predicates, but assume a heap of size 1. This produces a significantly shorter formula for the SMT-solver than a heap size according to the calculated bound. It is, however sufficient to find a solution with a heap size of 1: $x = y = z = \mathbf{null}$.

CHAPTER 6

Implementation

6.1 Integrating SL^*_{data} Within Z3 SMT-Solver

Our decision procedure was integrated within the state-of-the-art SMT-solver Z3. Because of that, we utilise both internal and external interfaces, allowing us to implement optimisations, which would not be possible by just using the public Application Programming Interface (API). An overview of the implementation is depicted in Figure 6.1. Due to the architecture of Z3, the implementation can be separated into three logical components:

Lexical analysis and syntax parsing:
In Z3, the entire parsing is done in advance. The parser builds up an AST and stores the parsed code in a DAG. Since we are introducing new symbols, we need to inform the parser of these new symbols. This is done by implementing a so-called *declaration plugin.*

Decision procedure:
After the input has been parsed, Z3 decides which decision procedure to instantiate. Decision procedures are loosely called *tactic* within the Z3 source code. This is because a *tactic* might not be a full decision procedure but might also define intermediate steps (such as formula simplification, normal form conversion etc.) that are used by multiple decision procedures. Typically, a decision procedure consists of multiple interacting tactics. This is decided based on the logic set by the SET-LOGIC expression. If no such expression is provided, Z3 will decide which decision procedure is best, based on the sorts and assertions used in the specified formula. Our decision procedure, however, needs to be explicitly specified.

Model conversion:
By utilising different tactics to find a solution, Z3 comes up with a model that does not necessarily represents a model as defined by the theory that was requested.

For example, Z3 might find a solution for a bit-vector formula just by utilising a SAT-solver. To produce a model that is correct in terms of the specified theory *model converters* are used. Coming back to the previous example, the result of a SAT-solver would be value assignments for individual bits instead of vectors, the bit-vector *model converter* would keep track of the separated bits and construct them back into a vector. Since our decision procedure entirely encodes the input into a regular SMT formula, we rely heavily on a *model converter* to produce a heap interpretation for the input SL^*_{data} formula.

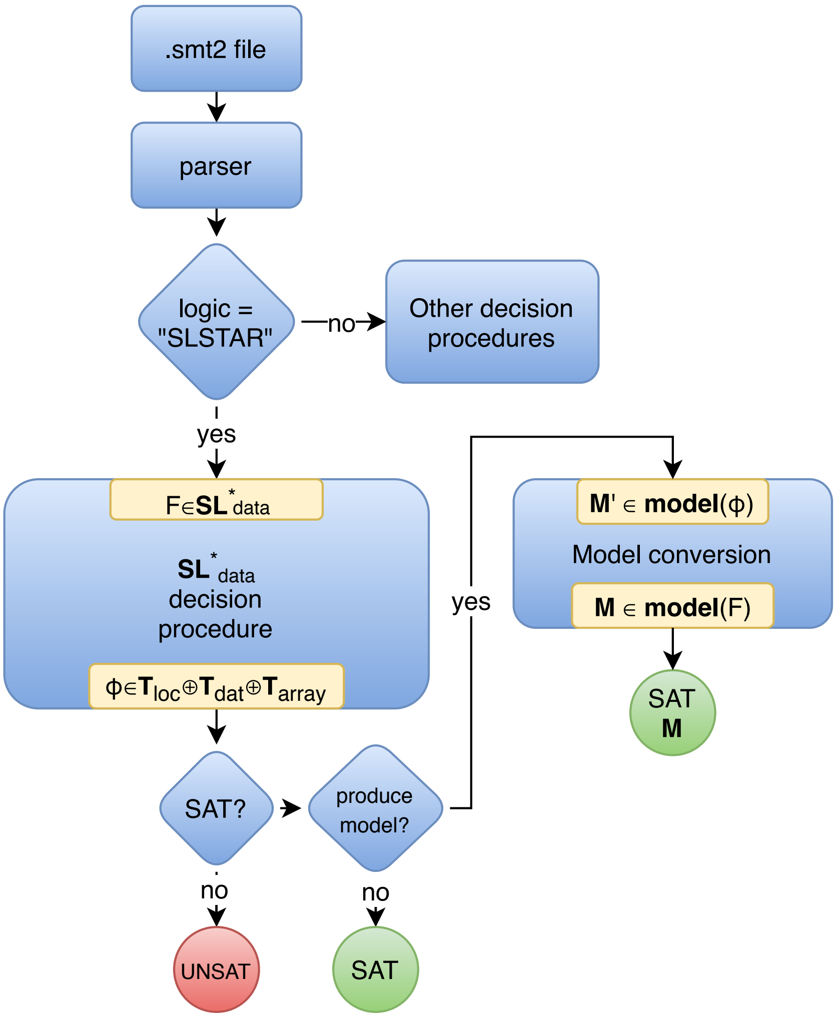


Figure 6.1: Implementation overview

6.1.1 Syntax Implementation

Z3 relies on so-called *declaration plugins* for introducing sorts and predicates. Our syntax introduces the following sorts:

`ListLoc`

The list location sort is parametric in sorts and expects up to two parameters. The first parameter specifies the sort which will be used to represent the location during the later encoding. The second parameter specifies the sort that will be used to represent data in the encoding. Both sorts default to `Int` and can be omitted.

Ultimately, those two sorts determine the parametric theories $\mathcal{T}_{loc}$ and $\mathcal{T}_{dat}$ of SL^*_{data}. If we, for example, specify both to `Int`, then LIA will be used as both theories (as long as the data and location formulas do not contain quantifiers or multiplication); if we specify them to `(_ BitVec 64)` (i.e. a 64-bit vector), then BV theory will be used.

Since the encoding depends upon all locations and data to be of the same sort, specifying two variables with two different `ListLoc` variants results in a syntax error.

`TreeLoc`

The tree location sort is similar to the list location sort. It, too, takes the same two parameters to guide the encoding. The only difference is the context in which both may be used. `ListLoc` can only be used in predicates describing lists, while `TreeLoc` in predicates describing trees. Even though they are different on syntax level, the encoding uses the same sorts for representing data and locations. Specifying a `TreeLoc` and a `ListLoc` variable with different data and location sorts results in a syntax error.

`Dpred`

`Dpred` is an internal helper sort and cannot be used for variables. It is used to identify data predicates during parsing.

`NullSort`

`NullSort` is an internal helper sort and cannot be used for variables. It is used to represent **null** and must be cast into a specific location sort when used in a formula.

Apart from the sorts, the syntax defines the following predicates:

sep represents the *separating conjunction.* $F * G \leftrightarrow$ `(sep F G)`
It has variable arity with at least two arguments. More than two arguments can be used to abbreviate a formula e.g. `(sep F G H)` is semantically the same as `(sep F (sep G H))`.

ptol represents the $\rightarrow_l$ predicate. $x \rightarrow_l y \leftrightarrow$ `(ptol x y)`
It has arity of two and expects two `TreeLoc` arguments.

ptor represents the $\rightarrow_l$ predicate. $x \rightarrow_r y \leftrightarrow$ `(ptor x y)`
It has arity of two and expects two `TreeLoc` arguments.

ptolr represents the $\rightarrow_{l,r}$ predicate. $x \rightarrow_{l,r} (y, z) \leftrightarrow$ `(ptolr x y z)`
It has arity of three and expects three `TreeLoc` arguments.

pton represents the $\rightarrow_n$ predicate. $x \rightarrow_n y) \leftrightarrow$ `(pton x y)`
It has arity of two and expects two `ListLoc` arguments.

ptod represents the $\rightarrow_n$ predicate. $x \rightarrow_d y) \leftrightarrow$ `(pton x y)`
It has arity of two and expects either a `(ListLoc TLoc TDat)` or `(TreeLoc TLoc TDat)` argument as the first and the generic TDat argument.

tree represents the tree predicate.
$\mathsf{tree}(x, \vec{s}, \mathcal{P}) \leftrightarrow$ `(tree P1 P2 ... P_j x s1 s2 ... s_i)`
It has variable arity with at least one `TreeLoc` argument. It expects a variable number of data predicates, followed by a `TreeLoc` argument, last it expects a variable number of `TreeLoc` arguments for the leaf nodes.

list represents the list predicate.
$\mathsf{list}(x, \vec{s}, \mathcal{P}) \leftrightarrow$ `(list P1 P2 ... P_j x s1 s2 ... s_i)`
It has variable arity with at least one `ListLoc` argument. It expects a variable number of data predicates, followed by a `TreeLoc` argument, last it expects a variable number of `ListLoc` arguments for the leaf nodes (semantically, only one makes sense).

null represents the **null** location. **null** $\leftrightarrow$ `null` It is a constant and thus has no arguments. It has sort `NullSort` and needs to be cast to a specific location sort (e.g. `(as null ListLoc)`)

unary used to mark an expression as a unary data predicate.
$\mathsf{list}(x, \{(\alpha = 0)\}) \leftrightarrow$ `(list x (unary (= alpha 0)))`

left used to mark an expression as a left-constrained binary data predicate.
$\mathsf{tree}(x, \{(\mathsf{l}, \alpha < \beta)\}) \leftrightarrow$ `(tree (left (< alpha beta)) x )`

right used to mark an expression as a right-constrained binary data predicate.
$\mathsf{tree}(x, \{(\mathsf{r}, \alpha \geq \beta)\}) \leftrightarrow$ `(tree (right (>= alpha beta)) x)`

next used to mark an expression as a next-constrained binary data predicate.
$\mathsf{tree}(x, \{(\mathsf{n}, \alpha < \beta)\}) \leftrightarrow$ `(list (next (< alpha beta)) x)`

alpha used as an argument for unary and binary data predicates. $\alpha \leftrightarrow$ `alpha`
If a data sort different from Int is used, it needs to be cast.

beta used as an argument for binary data predicates. β $\leftrightarrow$ **beta**
If a data sort different from INT is used, it needs to be cast.

Examples

To illustrate the syntax of the input format, we show the input files for a few example formulas.

$x \rightarrow_{l,r,d} (y, z, d) * y \rightarrow_l y_l * y_l \neq \mathbf{null} * (d > 0)$:

```
(set-logic SLSTAR)

(declare-const x TreeLoc)
(declare-const y TreeLoc)
(declare-const z TreeLoc)
(declare-const yl TreeLoc)
(declare-const d Int)

(assert
    (sep
        (ptolr x y z)
        (ptod x d)
        (ptol y yl)
        (not (= yl (as null TreeLoc)))
        (> d 0)
    )
)

(check-sat)
(get-model)
```

$\mathsf{list}(x, \{(\mathsf{n}, \alpha < \beta)\}) \wedge x \rightarrow_{n,d} (y, a) * y \rightarrow_{n,d} (z, b)$:

The following example demonstrates redefining the location and data sorts, as well as data predicates. Here we used `(_ BitVector 3)` i.e. a 3-bit bit-vector for the location, resulting in $\mathcal{T}_{loc} = \mathsf{BV}$ and `Real` for data, resulting in $\mathcal{T}_{dat} = \mathsf{LRA}$.

```
(set-logic SLSTAR)

(define-sort BV () (_ BitVec 3))
```

```
(declare-const x (ListLoc BV Real))
(declare-const y (ListLoc BV Real))
(declare-const z (ListLoc BV Real))
(declare-const a Real)
(declare-const b Real)

(assert
    (list (next (< (as alpha Real) (as beta Real) ))
          (unary (> (as alpha Real) 15.0))
         x)
)

(assert
    (sep
        (pton x y)
        (ptod x a)
        (pton y z)
        (ptod y b)
    )
)

(check-sat)
(get-model)
```

tree$(x, \{(\mathsf{l}, \alpha < \beta), (\mathsf{r}, \alpha \geq \beta)\})$:

The following example demonstrates usage of a location theory formula.

```
(set-logic SLSTAR)

(declare-const x TreeLoc)
(assert
    (sep
        (tree (left (< alpha beta))
              (right (>= alpha beta))
              x)
        (> (as x Int) 15)
    )
)
(check-sat)
```

Technical Details

As mentioned previously, Z3 uses a so-called *declaration plugin* to define new symbols. This is simply done by extending the abstract class `decl_plugin` and implementing two methods: `get_sort_names`, which returns the names of new sorts, and `get_op_names`, which returns the names of constants, functions and predicates. During parsing, the parser calls the methods `mk_sort` and `mk_func_decl`. Those are responsible for checking arity and types of arguments as well as actually creating node objects for the AST and DAG. A simplified class diagramm of our *declaration plugin* is depicted in Figure 6.2.

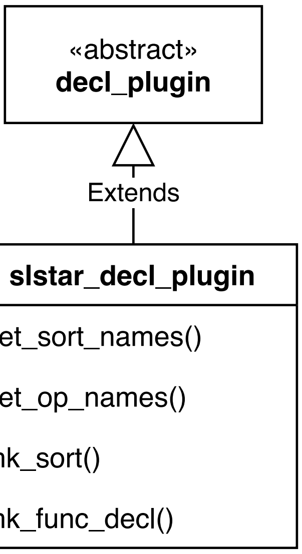


Figure 6.2: Declaration plugin class

6.1.2 Implementing the Decision Procedure

The implementation of the decision procedure consists of several parts:

1. Simplification (before encoding)
2. Bound calculation
3. Encoding to SMT formula ϕ
4. Simplification (after encoding)
5. Decision procedure for ϕ

An overview is depicted in Figure 6.3. First, a simplification is performed. This consists of flattening the formula by eliminating *and*-operands, removing redundancies and rewriting the formula to Conjunctive Normal Form (CNF). This is done by exploiting Z3 built-in functionality (denoted as “Simplify” and “Propagate values” in the overview). We end up with a list of top level conjuncts, each of which is either already a spatial formula or spatial formula connected by logical *or*-operands. Since the built-in tactics are not aware of the semantics of the spatial atoms, we then perform a spatial equality propagation. The details of this steps are described later in this chapter in Section 6.2.2.

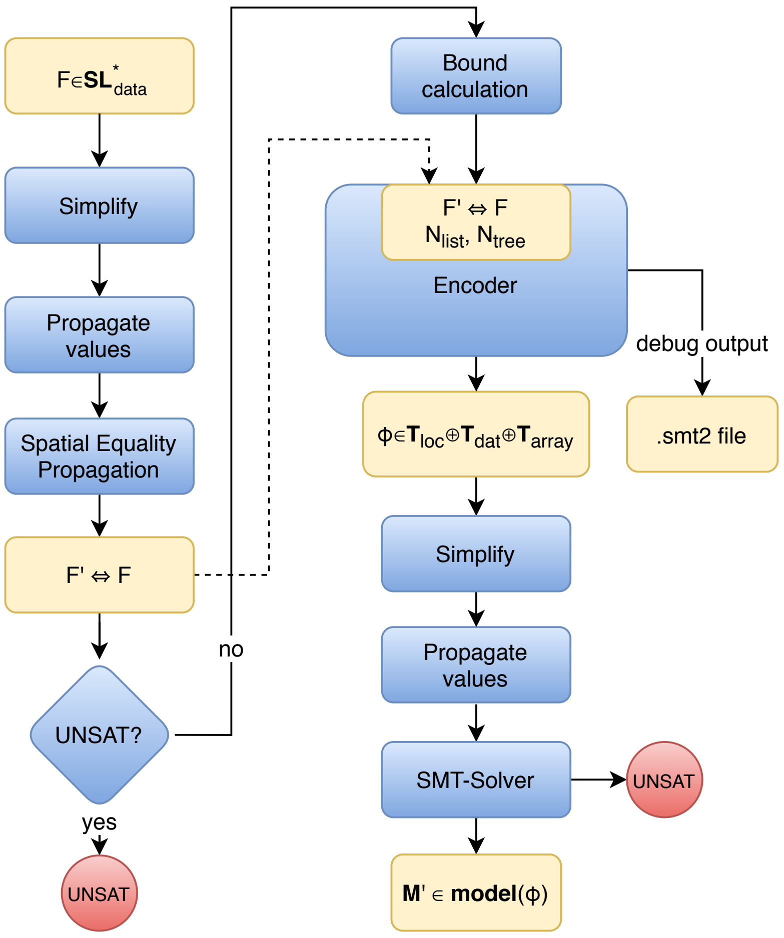


Figure 6.3: SL^*_{data} decision procedure

After simplification, Z3 might already be aware of a conflict. If this is the case, we can report that the formula is not satisfiable. If there is no conflict, we proceed by calculating an upper bound for the heap size. The bound calculation is ispired by the implementation in Sloth [12]. There are several rules used to determine the bounds:

1. The bound N is separated in $N = N_{list} + N_{tree}$

2. If there is a top level conjunct that is not a negation or a disjunction, without an inductive predicate (list or tree), this gives us a hard bound: N_{list} is equal to the number of unique list locations on the left side of $\rightarrow_n$ and *ptod* predicates, while N_{tree} is equal to the number of unique tree locations on the left side of $\rightarrow_l, \rightarrow_r$ and *ptod* predicates. In particular, a formula $F \in \mathcal{T}_{loc}$ or $F \in \mathcal{T}_{dat}$ dictates an empty heap and thus a bound of 0.

3. If there is no such conjunct (i.e. every conjunct is a negation ($\neg F$), a disjunction ($F \vee G$) or contains an inductive predicate), the bound is the maximum of each conjunct's bound.

4. A bound of a negation is the bound of the non-negated formula but it has impact on data predicates within, so we need to keep track of negations.

5. A bound of a disjunction is the maximum of each disjunct.

6. N_{list} of a top level spatial conjunct with inductive predicates is equal to the number of unique list locations plus the number of inductive list predicates. If the conjunct is negated, 1 is added for each unary data predicate and 2 for each binary data predicate.
   Similarly, N_{tree} is equal to the number of unique tree locations plus the number of inductive tree predicates plus the number of leaf nodes. If the conjunct is negated, 1 is added for each unary data predicate and 2 for each binary data predicate.

In Table 6.1 are some examples for bound calculation:

Table 6.1: Examples for bound calculation

$\underbrace{y}_{+1} \rightarrow_r \underbrace{x}_{+1} * \underbrace{\mathsf{tree}}_{+1}(\underbrace{x}_{+0,\text{ not unique}}, \underbrace{\langle s_1, s_2, s_3\rangle}_{+3\cdot 2}, \underbrace{\{(\alpha \geq 0), (\mathsf{r}, \alpha > \beta)\}}_{+0,\text{ not negated}})$	$\rightarrow N_{tree} = 9$
$\neg(\overbrace{\mathsf{list}}^{+1}(\overbrace{x}^{+1}, \{\overbrace{(\alpha = 0)}^{+1}\}) * \underbrace{\mathsf{tree}}_{+1}(\underbrace{x}_{+1}, \underbrace{\langle s_1, s_2, s_3\rangle)}_{+3\cdot 2}, \underbrace{\{(\mathsf{r}, \alpha > \beta)\}}_{+2})$	$\rightarrow N_{list} = 3, N_{tree} = 10$
$\underbrace{x \rightarrow_r y}_{+1,\text{ no }\mathsf{list}\text{ or }\mathsf{tree}} \wedge \underbrace{\neg(\mathsf{tree}(x, \langle s_1, s2\rangle)}_{+0})$	$\rightarrow N_{tree} = 1$

Following the bound calculation, the SMT encoding is performed. The implementation closely complies with the encoding described in Chapter 4. There are two noteworthy differences:

1. The bound used for defining the reachability constraints, differ. The original encoding always uses the bound $N = N_{list} + N_{tree}$ to define the reachability constraints, when encoding list and tree predicates. In contrast, our encoding uses

N_{list} as the upper bound for the reachability constraints in list predicates and N_{tree} as the upper bound for the reachability constraints in the tree predicates.

2. If one of the two bounds N_{list} or N_{tree} is equal to zero, the footprint parts used to keep track of the corresponding predicates are eliminated. Specifically, if $N_{list} = 0$ then $X_n \notin \vec{X}$ and if $N_{tree} = 0$ then X_l and $X_r \notin \vec{X}$. This is also true for all the footprint helper variables ($\vec{Y}$).

Technical Details

Decision procedures are called *tactic* in Z3. Tactics do not always describe a full decision procedure but might also define intermediate steps. Our decision procedure is implemented in the `slstar_tactic`. It consists of multiple other tactics: `slstar_spatial_eq_propagation_tactic` `slstar_reduce_tactic`, as well as the Z3 built-in `simplify_tactic` and `propagate_values_tactic`. In addition to providing boilerplate and glue code, it also implements the calculation of the bounds.

The `slstar_spatial_eq_propagation_tactic` implements the spatially aware equality propagation optimisation, described in Section 5.2 and its implementation in Section 6.2.2. The `slstar_reduce_tactic` implements the encoding. The name "reduce" is following Z3 naming conventions as the encoding is a reduction to some other theory. The encoding itself is implemented in the `slstar_encoder` class. For extensibility reasons, the encoding of the predicates is separated into separate classes: `list_encoder` and `tree_encoder`, while the common functionality is implemented in `pred_encoder`.

A class diagram of the implementation is depicted in Figure 6.4.

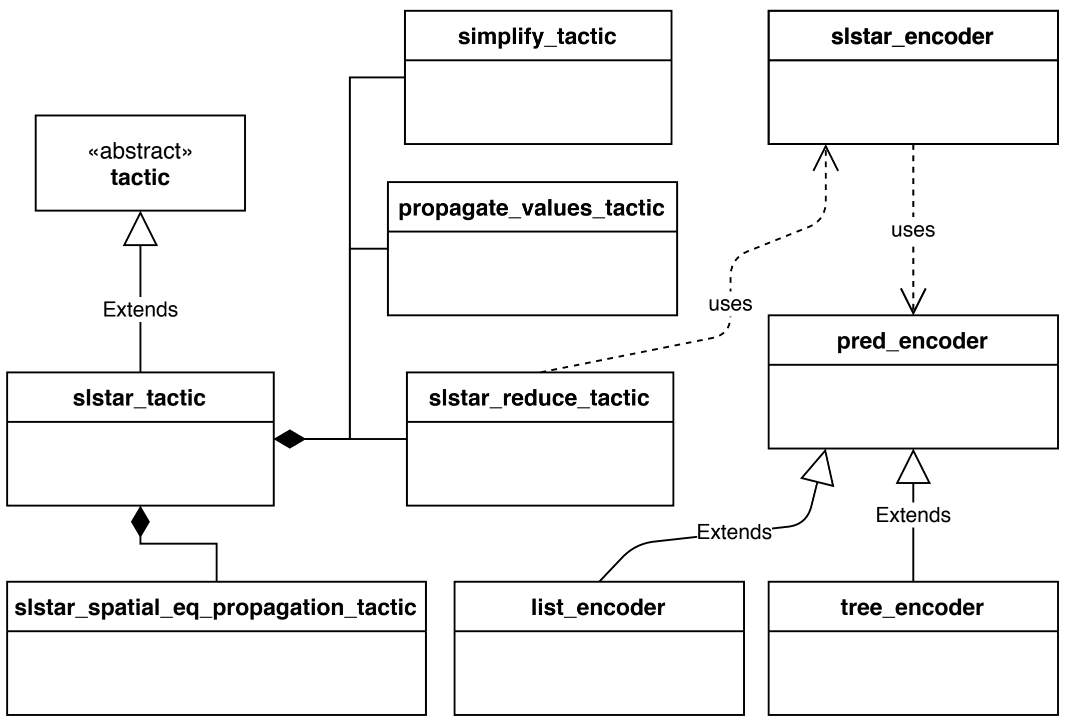


Figure 6.4: Encoding tactic class diagramm

6.1.3 Converting Z3 Models to SL^*_{data} Models

Z3 SMT solving algorithm produces a model for the encoded formula. This model not only contains interpretations that are used to describe the heap, but also consists of definitions for every single helper predicate that we used to simulate the semantics of SL^*_{data}. This is bad for several reasons: 1. the number of helper predicates can be quite big. 2. the interpretations are always total functions and might contain values we do not expect (e.g. f_n might be defined for values that are not a valid location, i.e. not one of $x_1, ..., x_N$ or **null**). 3. since we use arrays to simulate the footprint sets, the interpretations will be arrays as well.

In order to address this and similar issues, Z3 offers model converters that can be attached to decision procedures. Our model converter performs the following tasks:

1. It keeps track of all helper predicates and removes them from the interpretation but caches the values for *null* and the other locations $x_1, ..., x_N$.
2. It evaluates each of the footprint arrays $X_n, X_l, X_r and X_d$ for the locations $x_1, ..., x_N$ in order to create concise lists of elements rather than array interpretations.
3. It evaluates each of the predicate functions $f_n, f_l, f_r and f_d$ and cleans them up to not contain any unnecessary definitions. They have the form of an *if-then-else*

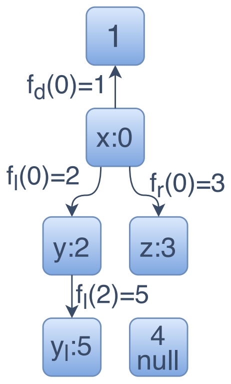


Figure 6.5: The graphical representation of the model from Listing 6.1

chain that only contains equalities with location variables. The resulting model resembles something similar to the following example:

$$f_r(x) = \begin{cases} 3 & x = 2 \\ 5 & x = 4 \\ 0 & else \end{cases}$$

Note that because SMT functions always are total functions, it has a fallback value. We just make sure that the equalities are always of the form $x = y$ where $x \in \{x_1, ..., x_N\}$

Listing 6.1 shows an example of a processed model in the SMT-LIB format. It describes a heap interpretation, its graphical representation is depicted in Figure 6.5. The heap interpretation is indeed a model for the formula $x \rightarrow_{l,r,d} (y, z, d) * y \rightarrow_l y_l * y_l \neq \textbf{null} * (d > 0)$. It includes:

1. The resolved locations for the variables $x = 0, y = 2, z = 3$ and $y_l = 5$
2. A null value $\textbf{null} = 4$
3. The function $\mathsf{f\_left}$ and a domain $\mathsf{Xl} = \{0, 2\}$, $\mathsf{f\_left}(0) = 2, \mathsf{f\_left}(2) = 5$, representing the partial function f_l
4. The function $\mathsf{f\_right}$ and a domain $\mathsf{Xr} = \{0\}$, $\mathsf{f\_right}(0) = 3$, representing the partial function f_r
5. The function $\mathsf{f\_dat\_int}$ and a domain $\mathsf{Xd} = \{0\}$, $\mathsf{f\_dat\_int}(0) = 1$, representing the partial function f_d

```
sat
(model
  (define-fun Xl () Array ( 0 2))
  (define-fun x () Int 0)
  (define-fun d () Int 1)
  (define-fun null () Int 4)
  (define-fun y () Int 2)
  (define-fun Xr () Array ( 0))
  (define-fun Xd () Array ( 0))
  (define-fun yl () Int 5)
  (define-fun z () Int 3)
  (define-fun f_left ((x!0 Int)) Int
    (ite (= x!0 0) 2
    (ite (= x!0 2) 5
      2)))
  (define-fun f_right ((x!0 Int)) Int
    (ite (= x!0 0) 3
      3))
  (define-fun f_dat_int ((x!0 Int)) Int
    (ite (= x!0 0) 1
      1))
)
```

.

Listing 6.1: The output of our decision procedure for the input file from page 41

6.2 Implementing the Proposed Optimisations

6.2.1 Encoding Size Reduction

Since the reduction of the size has been achieved by modifying the encoding, these optimisations were not a distinct part of the implementation. Rather, they were guidelines for the realisation of the encoding:

1. The encoding was implemented recursively. This means the encoding is performed bottom up, joining encodings of the sub-formula according to the rules from Chapter 4. Since Z3 is already providing a DAG for us, we just need to use a dictionary to cache already encoded sub-formula. We can therefore prevent unnecessary encodings and the introduction of redundant symbols.

2. At all times during the encoding, we have the information about the bound N, N_{list} and N_{tree}. If one of the bounds is zero, we omit specifications about lists or

trees, depending on which one is unnecessary. This includes all partial functions f_n, f_lorf_r and their domains X_n, X_lorX_r and sub domains Y_n, Y_lorY_r.

3. Lastly, when encoding the inductive predicate, we encode the reachability only up to N_{list} or N_{tree}, respectively, not up to N.

6.2.2 Equality Propagation

The equality propagation was implemented using a list of equality bins. The bins themselves were implemented using hash-sets. A better solution would be a union-find data structure, however no readily available or reusable implementation was present within the z3 source code. We did not find our implementation limiting, however union-find would perform better for formulas with lots of equalities.

First, the equality bins are created using the algorithm depicted in Figure 6.6. Then for each of the bins, a single representative is chosen, let it be x. For every other symbol y in the bin, we create a *rewrite rule* "$x \leftrightarrow y$". The rewrite rules cause Z3 to substitute the symbols accordingly.

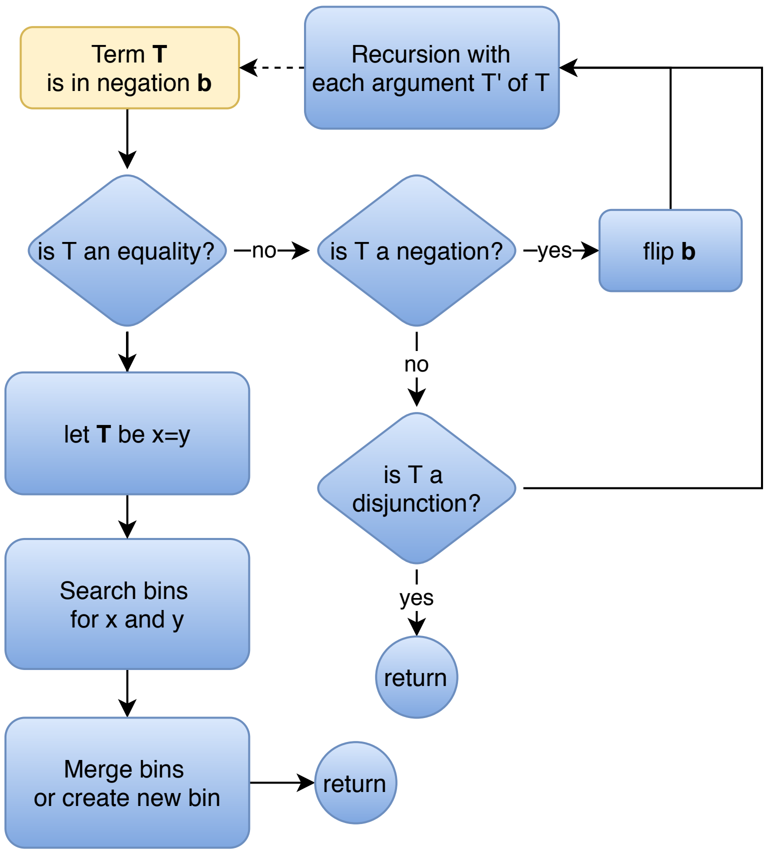


Figure 6.6: Finding equality bins

6.2.3 Stepwise Encoding

We implemented two variants of the *Stepwise Encoding*. The first variant encodes in two steps: (1) encoding everything except the list and tree then (2) encoding the remaining list and tree, predicates. This is depicted in Figure 6.7.

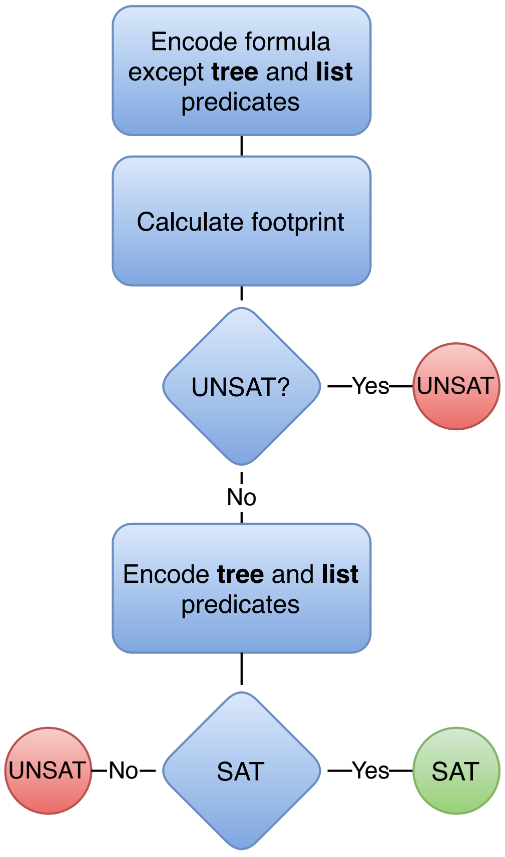

Encode formula except **tree** and **list** predicates
Calculate footprint
UNSAT?
Yes
UNSAT
No
Encode **tree** and **list** predicates
UNSAT
No
SAT
Yes
SAT

Figure 6.7: Modified part of the decision procedure for SL^*_{data}using two-step stepwise encoding

The second variant performs the encoding like the first variant but instead of using the full footprint, it starts with a footprint equal to 1 and increases it with each step. This is not done separately for N_{list} and N_{tree}, which means there are at most $N_{list} \cdot N_{tree}$ steps. This variant is depicted in Figure 6.8.

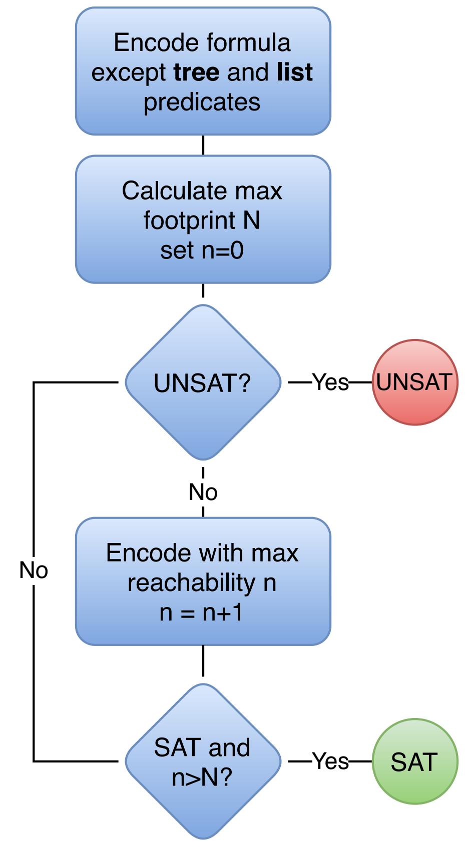

Encode formula except **tree** and **list** predicates
Calculate max footprint N set n=0
UNSAT?
Yes
UNSAT
No
Encode with max reachability n n = n+1
No
SAT and n>N?
Yes
SAT

Figure 6.8: Modified part of the decision procedure for SL$^*_{data}$using iterative encoding

CHAPTER 7

Results

The benchmarks were separated in two parts: an evaluation of the encoding run-time and an evaluation of the decision procedure run-time.

Figure 7.1 shows the final results of the encoding run-time. Our encoding is faster by several orders of magnitude, but likely because different programming languages used to implement the encoders.

Figure 7.2 shows the total run-time comparison of our best performing variant of the decision procedure, while Figure 7.3 depicts the comparisons for the run-times of the SMT-decision. The total run-times show significant improvement over SLOTH. The SMT run-times show the optimisations cause a small overhead for small formulas, but a significant improvement for most other formulas.

The following two sections describe the details of the evaluations.

7.1 Benchmark: Encoding

The benchmark for the encoding consisted of using a family of SL^*_{data} formulas $\underline{\mathsf{list}_N}$:

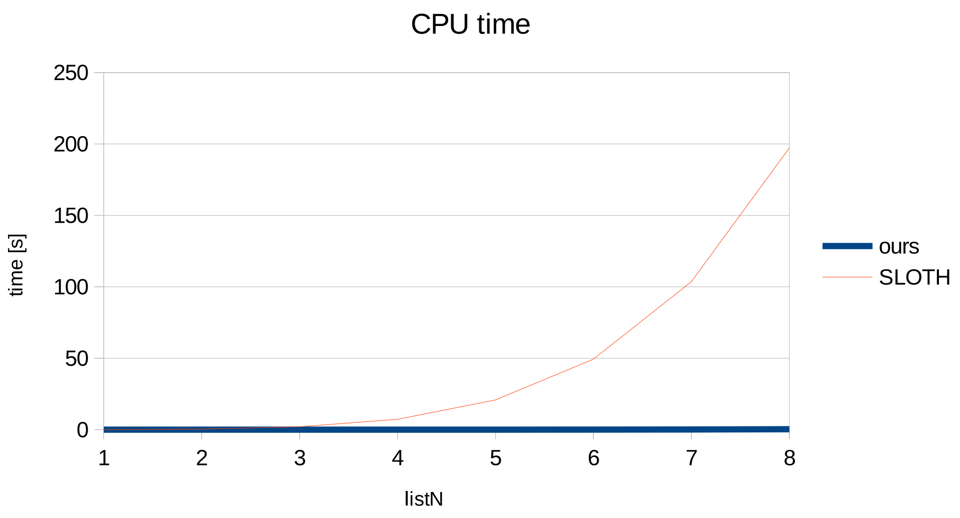
$$\mathsf{list}_N = \mathop{\ast}_{n=1}^{N} \mathsf{list}(x_n) * \mathsf{distinct}(x_1 ... x_n)$$

The formulas grow linearly in size and consist of N list-predicates and N heap locations that are all distinct. The bound for the formula is therefore $N \cdot 2$. By exploiting the linear growth and the fact all formulas are similar, we can easily compare the relation between the formula size to the encoding run-time.

All run-time measurements were performed on an Intel Core i5-3230M clocked at 2.6GHz. The timings were done three times, the resulting data is the arithmetic average of those three runs. As a representative for our solution, we chose our decision procedure with the optimisation variant SRed+Eq+UF. The optimisations are listed in Table 7.1. That particular variant was chosen because it performed best.

In Figure 7.1, you can see the comparison between our encoder and the encoding part of Sloth. In the logarithmic diagram, we can see that both are parallel and flatten in the same way. This shows that our encoding is not better on an algorithmic level. Both flatten as the formulas grow, which shows that as the footprint grows, the run-time of the encoding becomes more and more insignificant.

Although our encoding is faster by several orders of magnitude, it appears to be a constant factor, which is easily explained by the different programming languages used to implement the encoders. While Sloth was implemented using the Python programming language, our encoder is implemented in C++. The former is an interpreted language, while the latter is compiled to a bare metal executable.



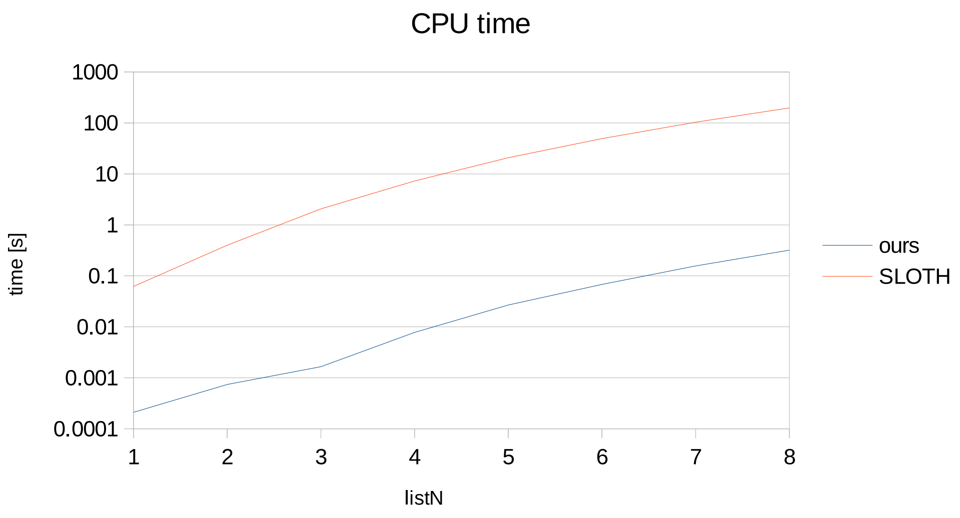


Figure 7.1: Comparison of encoder-run-times

7.2 Benchmarks: Decision Procedure

For the second benchmark, we compared the run-times of the satisfiability of the decision procedure. We are comparing the run-times for the satisfiability decision of the encoded SMT formula as well as the footprints used by the encoder. The benchmark consists of 16 representative formulas. The following is the list of the formulas we used, while the input files we used are listed in the Appendix in Section 8.2.

binary-search-tree

This formula describes a binary search tree with two sub-trees to enforce a minimal size.

$$\begin{aligned}&\mathsf{tree}(x, \{(\mathsf{l}, \alpha < \beta), (\mathsf{r}, \alpha > \beta)\})\\ &\quad\wedge (x \rightarrow_{l,r} l, r) * (l \rightarrow_{l,r} ll, lr)\\ &\quad * (r \rightarrow_{l,r} rl, rr) * (x \rightarrow_d x_{data})\\ &\quad * (l \rightarrow_d l_{data}) * (r \rightarrow_d r_{data})\end{aligned}$$

different-lists

This formula describes two lists: $\{x, b, \mathbf{null}\}$ and $\{y, d, \mathbf{null}\}$. The data is constrained such that all elements must be equal. The data values for the list elements staring in y are constrained not to be equal to A, while at the location x we have the constraint, that it is not a list where all elements are equal to A, which forces all elements to point to A.

$$\begin{aligned}&\mathsf{list}(x, \{(\mathsf{n}, \alpha = \alpha)\}) * \mathsf{list}(y, \{(\mathsf{n}, \alpha \neq A)\})\\ &\quad\wedge \neg\mathsf{list}(x, \{(\mathsf{n}, \alpha \neq A)\}) * \mathsf{list}(y, \{(\mathsf{n}, \alpha = \alpha)\})\\ &\quad\wedge A = 9001 * (x \rightarrow_n b) * (b \rightarrow_n \mathbf{null}) * (y \rightarrow_n d) * (d \rightarrow_n \mathbf{null})\\ &\quad * (x \rightarrow_d x_{data}) * (b \rightarrow_d b_{data}) * (y \rightarrow_d y_{data}) * (d \rightarrow_d d_{data})\end{aligned}$$

different-lists-no-size-bound

Similar to the previous formula, this describes two different constant lists, but without the size constrains, i.e. without in-between elements.

$$\begin{aligned}&\mathsf{list}(x, \{(\mathsf{n}, \alpha = \alpha)\}) * \mathsf{list}(y, \{(\mathsf{n}, \alpha \neq A)\})\\ &\quad\wedge \neg\mathsf{list}(x, \{(\mathsf{n}, \alpha \neq A)\}) * \mathsf{list}(y, \{(\mathsf{n}, \alpha = \alpha)\})\end{aligned}$$

list-all-distinct

This formula describes a list with at least four elements. The data values behind all the elements are constrained to be different.

$$\begin{aligned}&\mathsf{list}(a, \{(\mathsf{n}, \alpha \neq \beta)\}) \wedge (a \rightarrow_n b) * (b \rightarrow_n c) * (c \rightarrow_n d) * (d \rightarrow_n e)\\ &\quad * (a \rightarrow_d a_{data}) * (b \rightarrow_d b_{data}) * (c \rightarrow_d c_{data}) * (d \rightarrow_d d_{data})\end{aligned}$$

list-equal-zero

This formula describes a list with at least four elements. The data values are constrained to be zero.

$$\begin{aligned}&\mathsf{list}(a, \{(\alpha = 0)\}) \wedge (a \rightarrow_n b) * (b \rightarrow_n c) * (c \rightarrow_n d) * (d \rightarrow_n e)\\ &\quad * (a \rightarrow_d a_{data}) * (b \rightarrow_d b_{data}) * (c \rightarrow_d c_{data}) * (d \rightarrow_d d_{data})\end{aligned}$$

list-increasing

This formula describes a list with at least four elements. The list is sorted and does not contain equal elements.

$$\begin{aligned}&\mathsf{list}(a, \{(\mathsf{n}, \alpha < \beta)\}) \wedge (a \rightarrow_n b) * (b \rightarrow_n c) * (c \rightarrow_n d) * (d \rightarrow_n e)\\&\quad * (a \rightarrow_d a_{data}) * (b \rightarrow_d b_{data}) * (c \rightarrow_d c_{data}) * (d \rightarrow_d d_{data})\end{aligned}$$

max-heap

This formula describes a tree that is a maximum heap. The root node has two sub-trees to enforce a minimal size.

$$\begin{aligned}&\mathsf{tree}(x, \{(\mathsf{l}, \alpha > \beta), (\mathsf{r}, \alpha < \beta)\}) \wedge (x \rightarrow_{l,r} l, r) * (l \rightarrow_{l,r} ll, lr)\\&\quad * (r \rightarrow_{l,r} rl, rr) * (x \rightarrow_d x_{data}) * (l \rightarrow_d l_{data}) * (r \rightarrow_d r_{data})\end{aligned}$$

list-pivot

This formula describes a pivot element M that separates a list into two: one with elements that point to data that is smaller than M, the other with elements that are greater than M. In-between elements are used to enforce a minimal size.

$$\begin{aligned}&\mathsf{list}(xm, \{(\alpha < M)\}) * \mathsf{list}(y, \{(\alpha > M)\}) * (m \rightarrow_n y) * (m \rightarrow_d M)\\&\quad \wedge (x \rightarrow_n a) * (a \rightarrow_n b) * (b \rightarrow_n m) * (m \rightarrow_n y)\\&\quad * (y \rightarrow_n d) * (d \rightarrow_n e) * (e \rightarrow_n f) * (f \rightarrow_n g)\\&\quad * (x \rightarrow_d x_{data}) * (a \rightarrow_d a_{data}) * (b \rightarrow_d b_{data}) * (m \rightarrow_d m_{data})\\&\quad * (y \rightarrow_d y_{data}) * (d \rightarrow_d d_{data}) * (e \rightarrow_d e_{data}) * (f \rightarrow_d f_{data})\end{aligned}$$

list-pivot-distinct

Similar to the previous formula with the additional constraint that all elements must be distinct.

$$\begin{aligned}&\mathsf{list}(x, m, \{(\alpha < M)\}) * \mathsf{list}(y, \{(\alpha > M)\}) * (m \rightarrow_n y) * (m \rightarrow_d M)\\&\quad \wedge \mathsf{list}(x, \{(\mathsf{n}, \alpha \neq \beta)\}) \wedge\\&\quad \wedge (x \rightarrow_n a) * (a \rightarrow_n b) * (b \rightarrow_n m) * (m \rightarrow_n y)\\&\quad * (y \rightarrow_n d) * (d \rightarrow_n e) * (e \rightarrow_n f) * (f \rightarrow_n g)\\&\quad * (x \rightarrow_d x_{data}) * (a \rightarrow_d a_{data}) * (b \rightarrow_d b_{data}) * (m \rightarrow_d m_{data})\\&\quad * (y \rightarrow_d y_{data}) * (d \rightarrow_d d_{data}) * (e \rightarrow_d e_{data}) * (f \rightarrow_d f_{data})\end{aligned}$$

double-alloc

The formula describes a double allocated list location.

$$(y \rightarrow_n x) * (y \rightarrow_n \mathbf{null})$$

entailment
The formula describes three locations that are trees and at the same locations they are not. The result is UNSAT which indicates that first part of the formula implies the second.

$$\begin{aligned}&\mathsf{tree}(a) * \mathsf{tree}(b) * \mathsf{tree}(c)\\&\quad \wedge \neg(\mathsf{tree}(a) * \mathsf{tree}(b) * \mathsf{tree}(c))\end{aligned}$$

entailment-eq
Similar to the previous formula, but with an equality that needs to be propagated.

$$\begin{aligned}&\mathsf{tree}(a) * \mathsf{tree}(b) * \mathsf{tree}(d)\\&\quad \wedge \neg(\mathsf{tree}(a) * \mathsf{tree}(b) * \mathsf{tree}(c)) \wedge c = d\end{aligned}$$

equal-lists
This formula describes three lists, that are at the same location. By the semantics of the separating conjunction this is UNSAT. Equality propagation can help decide it faster.

$$\mathsf{list}(x) * \mathsf{list}(y) * \mathsf{list}(z) * x = y * y = z$$

four-lists
A simple formula that describes four unconstrained lists.

$$\mathsf{list}(x_1) * \mathsf{list}(x_2) * \mathsf{list}(x_3) * \mathsf{list}(x_4)$$

four-list-cycle
This formula that describes a four element cycle.

$$(x_1 \to_n x_2) * (x_2 \to_n x_3) * (x_3 \to_n x_4) * (x_4 \to_n x_1)$$

entailment-max This formula comes from the proof in Section 3.5. We use it to prove an implication that needs to be proven as part of the proof.

$$\mathsf{list}(a_{tail}) * a \to_{n,d} (a_{tail}, m) * a = b \wedge \neg\mathsf{list}(a, b, \{\alpha \leq m\})$$

All run-time measurements were performed on an Intel Core i5-3230M clocked at 2.6GHz. The timings were done three times, the resulting data is the arithmetic average of those three runs. In addition to the final solution with all optimisations, we measured the impact of the optimisations themselves by only enabling one optimisation at a time.

Table 7.1 shows the abbreviations we used for labelling the combinations we used in the benchmarks. Figures 7.4, 7.5, 7.6 and 7.7 illustrate individual comparisons between tested variants, while Figures 7.2 and 7.3 show comparisons between our best performing variant and SLOTH. Each axis represents the timing of one of the variants; the diagonal line illustrates points of equal timing ($x = y$) and thus separates the measurements which perform better in one or the other variant. The footprint sizes are shown in Figures 7.8, 7.9,7.10 and 7.11.

In a direct comparison of total run-time with SLOTH, our best variant performs significantly better, as is evident in Figure 7.2. If we only compare the run-times of the decision procedure for the SMT formula, as shown in Figure 7.3, the results are less significant but still show that our procedure is favorable. Most of the formulas perform similar to SLOTH. There are some outliers, especially for the very low run-time formulas. This can be easily explained by the blow up produced by the CNF conversion. On the other hand, we have significant performance improvements for edge cases, due to our proposed optimisations.

Our other observations include:

- SRed performs good on decisions for entailments
- SRed performs better than SLOTH for “mixed” formulas that contain both tree and list predicates.
- EQ is able to lower the footprint required by the encoding in serveral cases and brings a small performance gain for those formulas
- When there is a lower bound than our bound calculation predicts, the Step variant brings a overwhelming improvement for the run-time. On the other hand, if no lower bound exists or the formula is UNSAT, it lowers the performance significantly. This indicates that a better bound calculation heuristics could be a better choice for an optimisation.
- UF has a small negative impact on performance but also brings significant improvements for edge cases.

Table 7.1: Abbreviations of optimisation variants

Abbreviation	Optimisation
SRed	Encoding Size Reduction
EQ	Spatial Equality Propagation
UF	Stepwise Encoding (two steps)
Step	Stepwise Encoding (iterative)

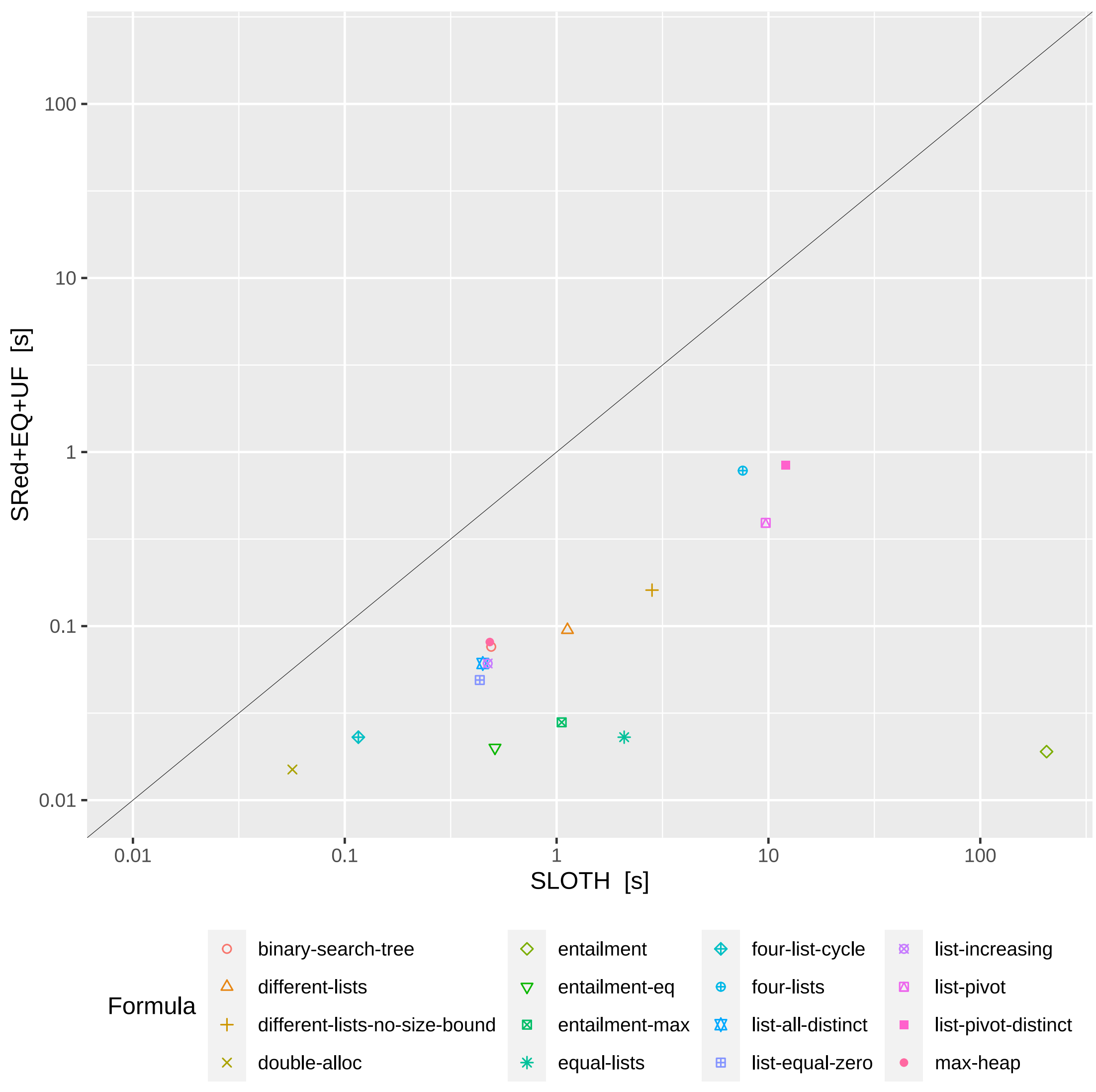


Figure 7.2: Comparison of total decision procedure run-times between SLOTH and SRed+EQ+UF

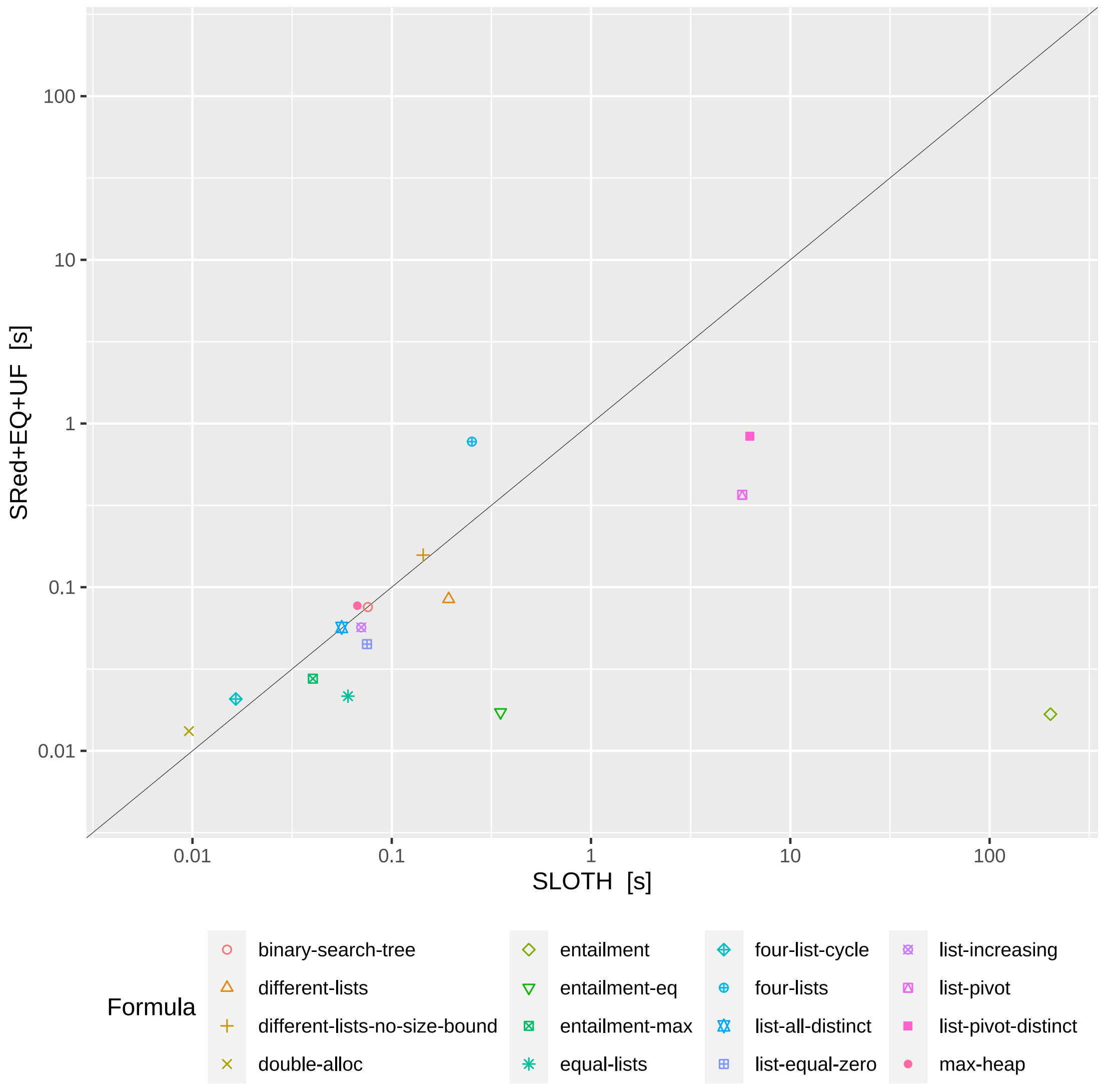


Figure 7.3: Comparison of SMT run-times between SLOTH and SRed+EQ+UF

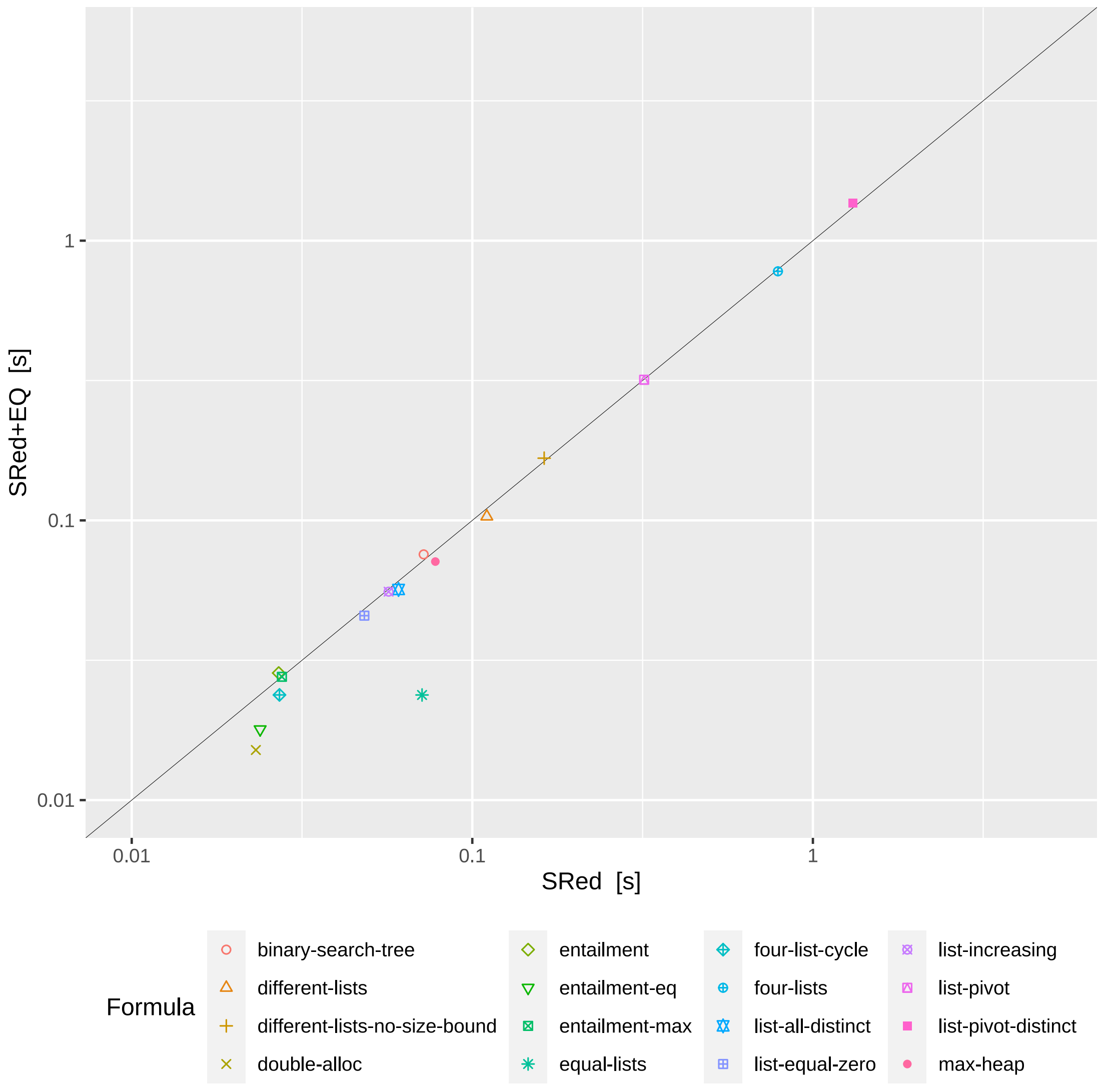


Figure 7.4: Comparison of SMT run-times between SRed and SRed+EQ

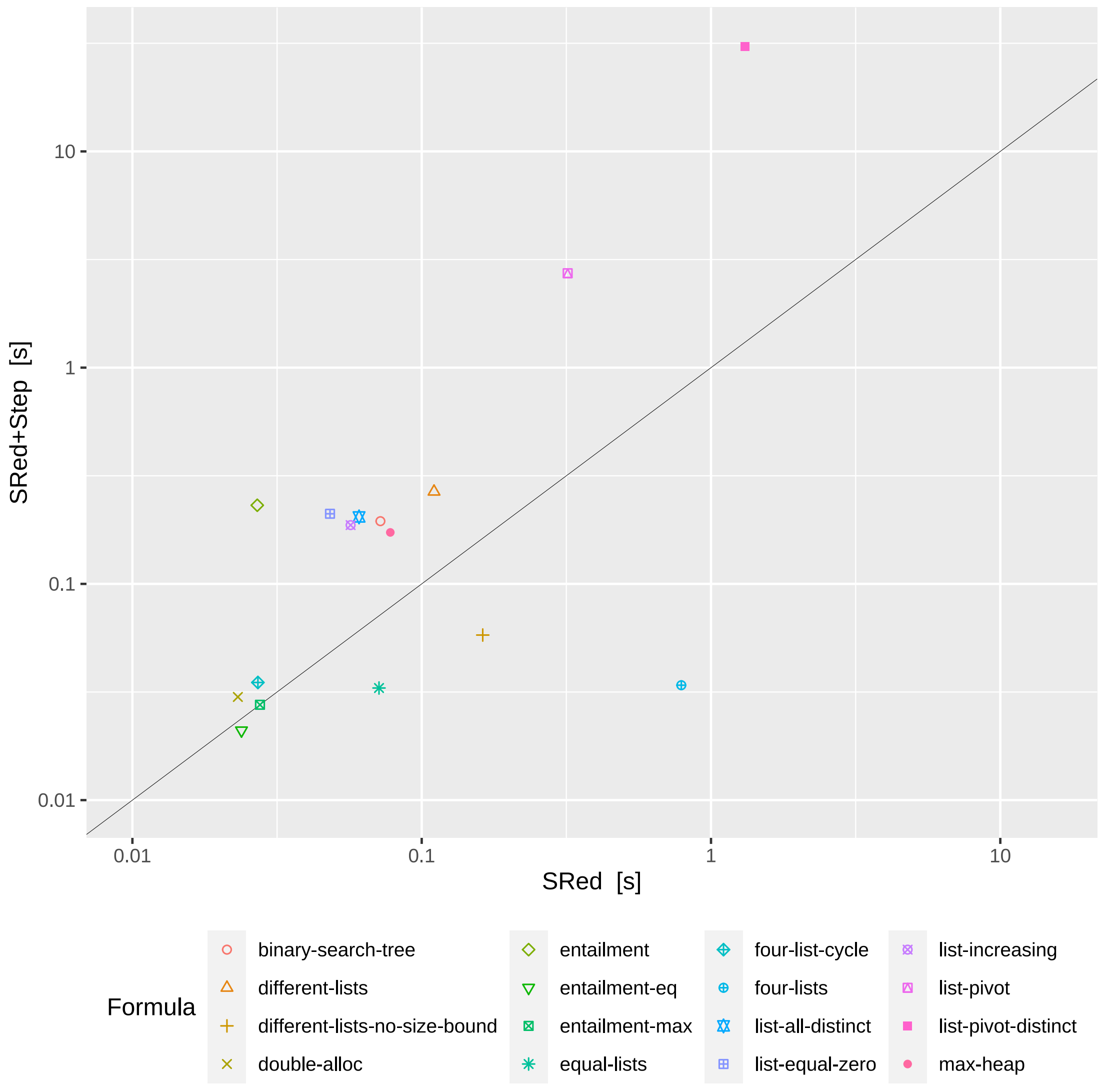


Figure 7.5: Comparison of SMT run-times between SRed and SRed+Step

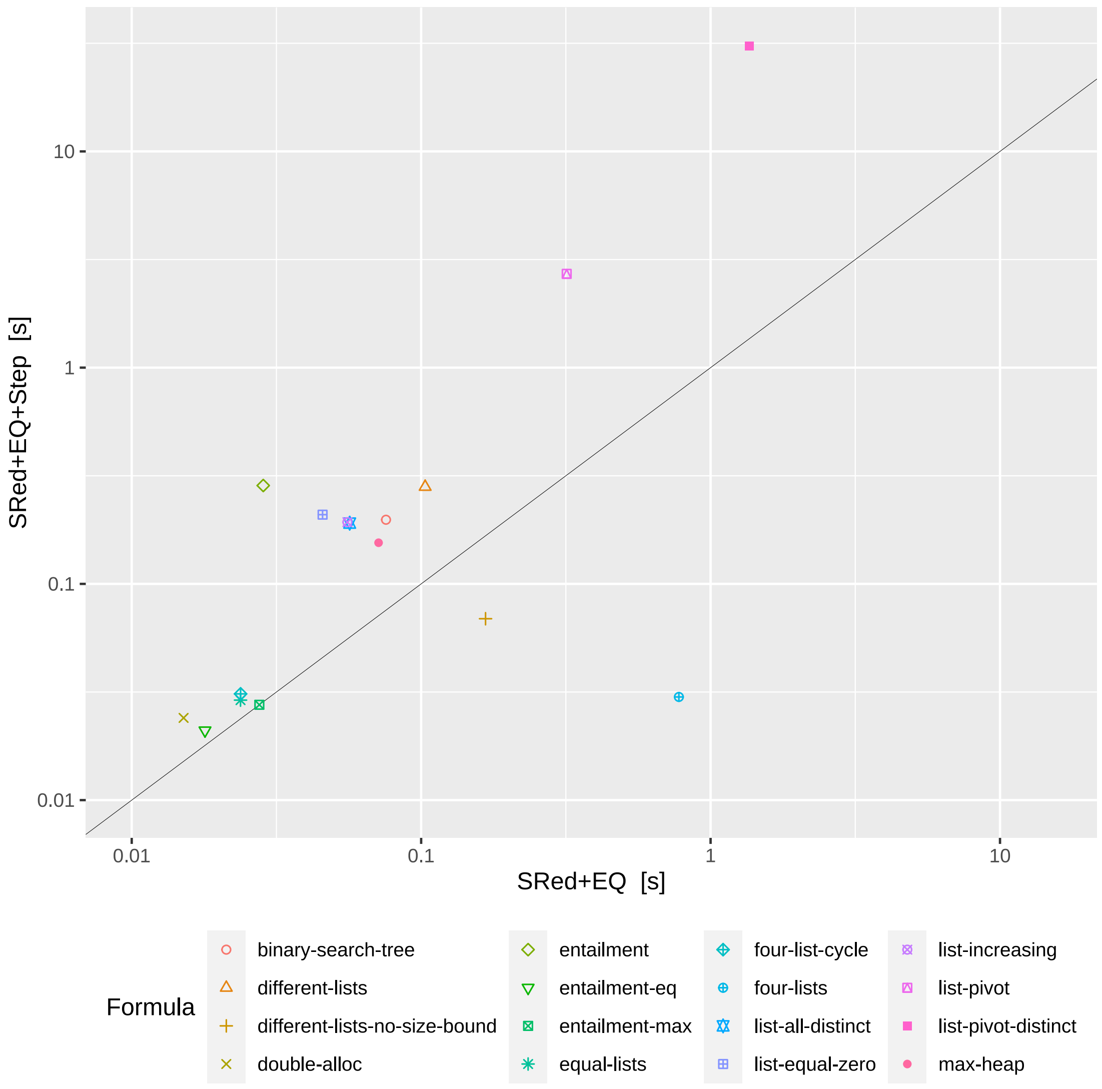


Figure 7.6: Comparison of SMT run-times between SRed+EQ and SRed+EQ+Step

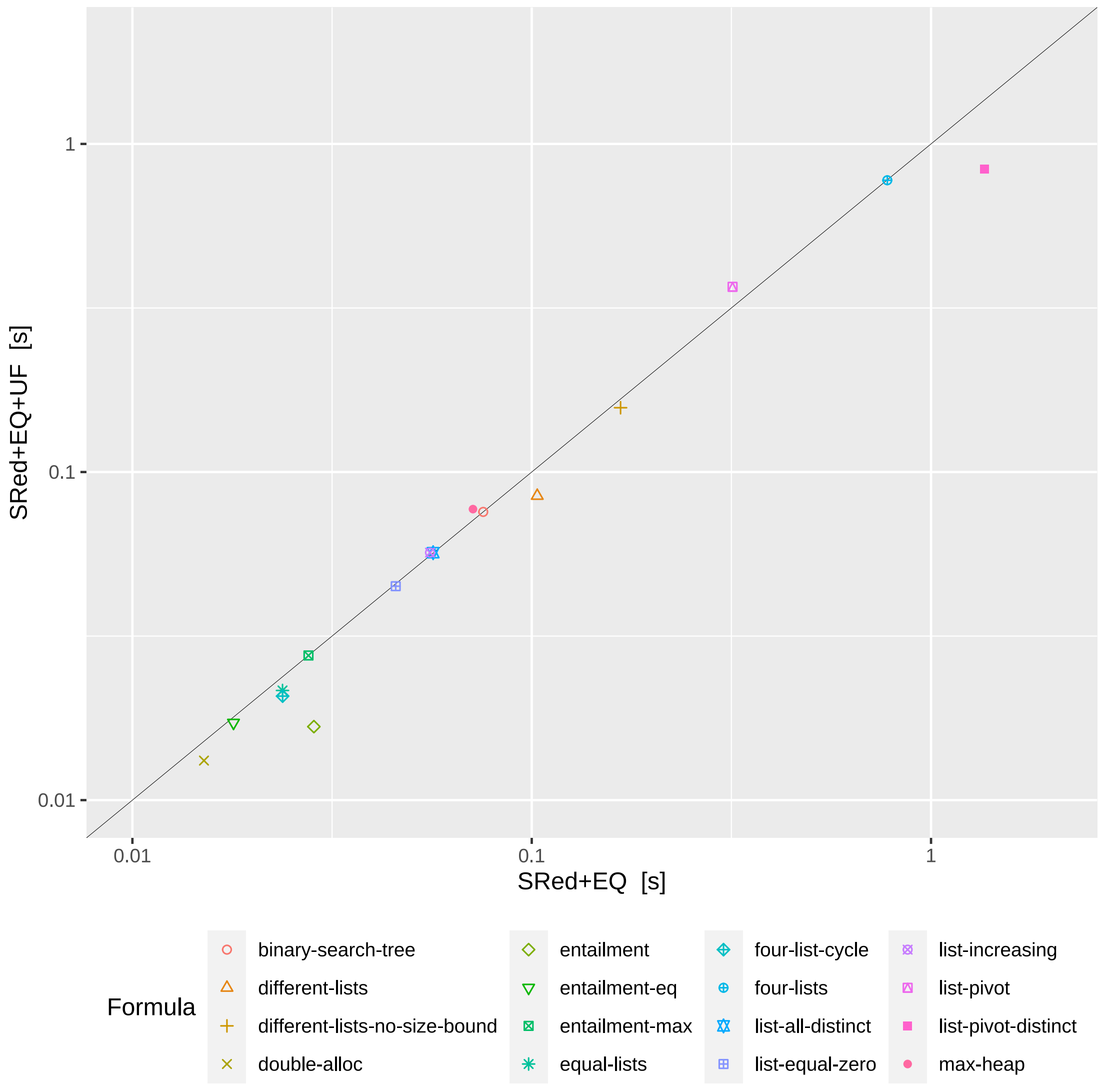


Figure 7.7: Comparison of SMT run-times between SRed+EQ and SRed+EQ+UF

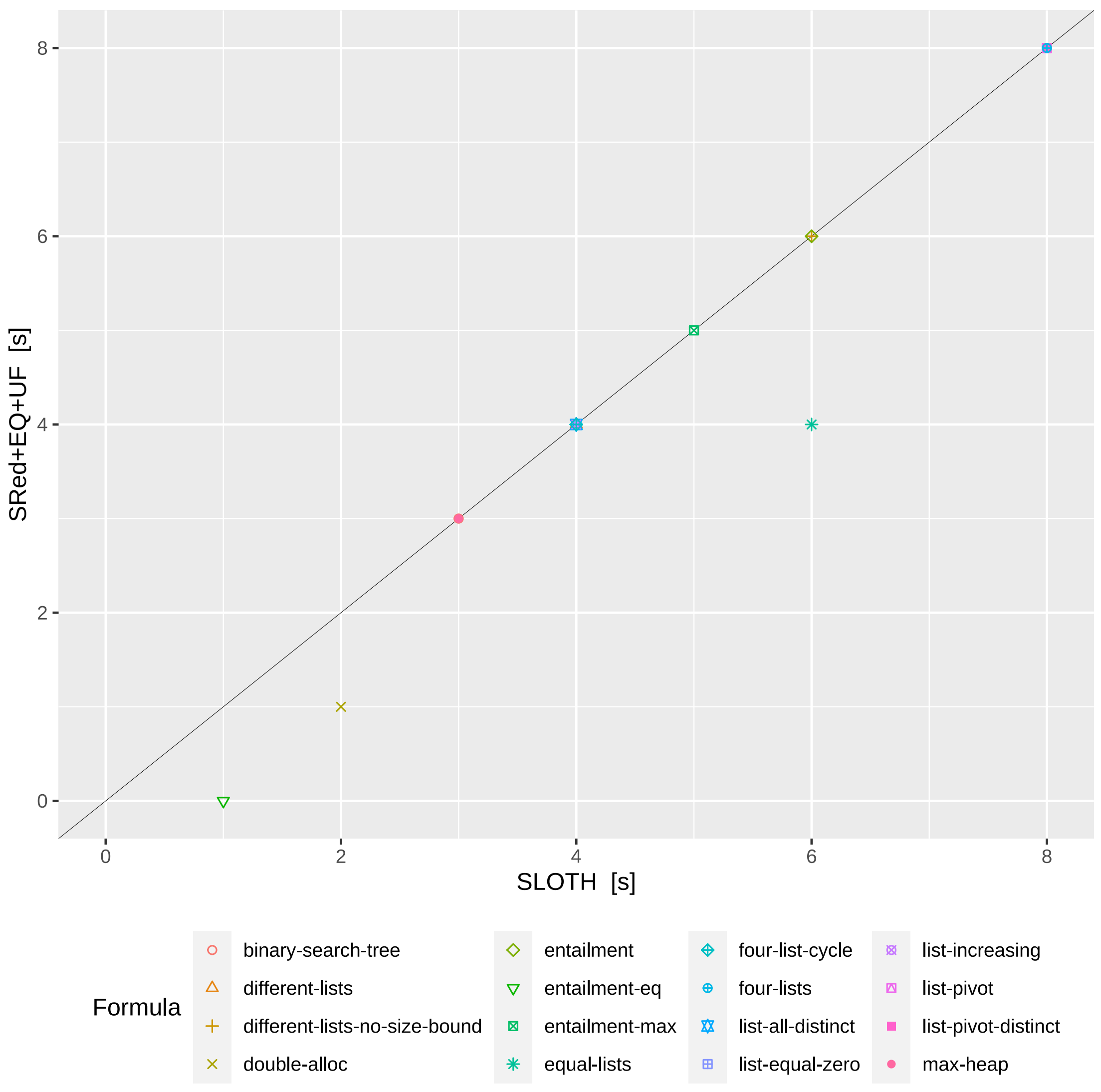


Figure 7.8: Comparison of footprint sizes between SLOTH and SRed+EQ+UF

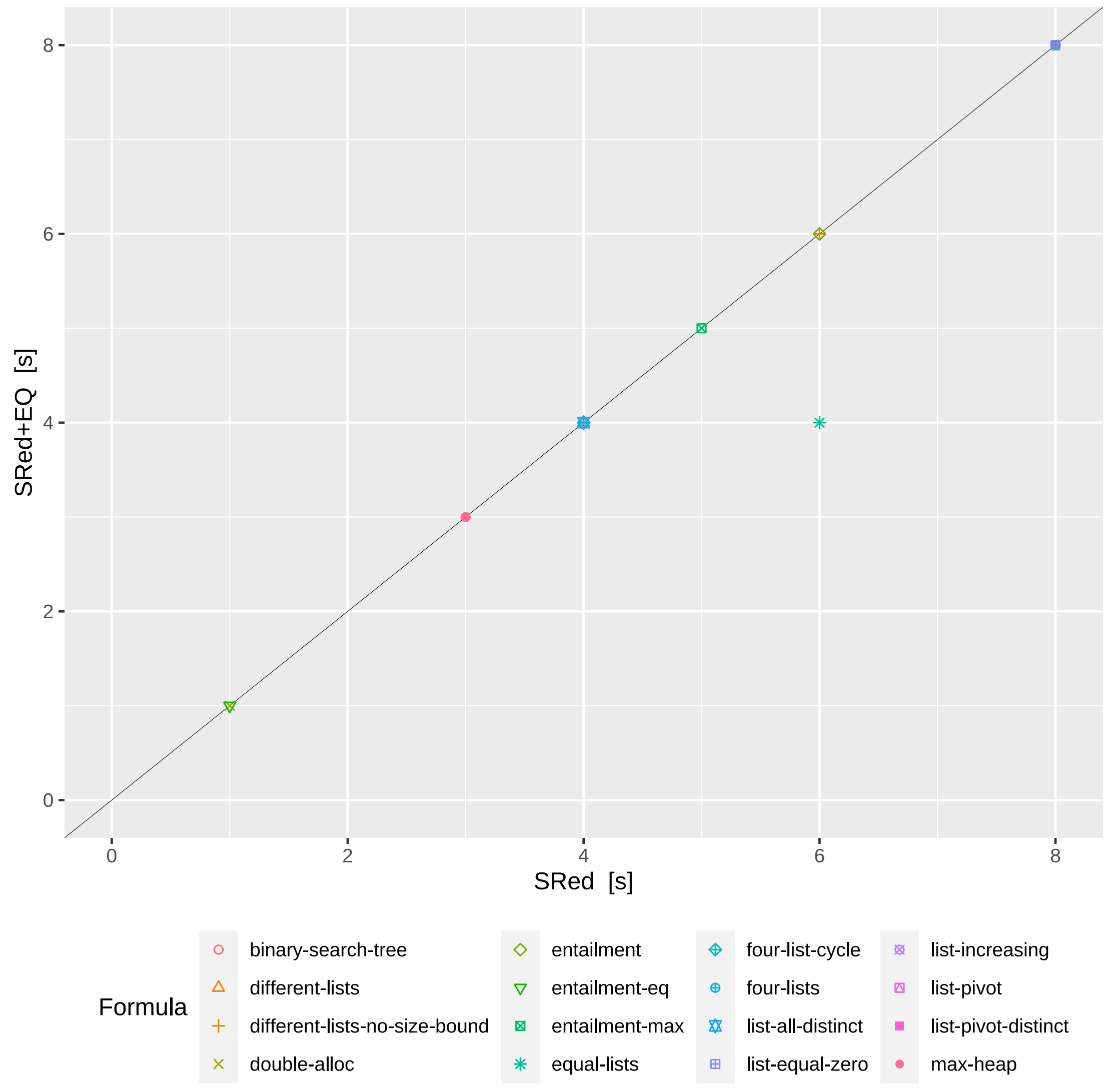


Figure 7.9: Comparison of footprint sizes between SRed and SRed+EQ

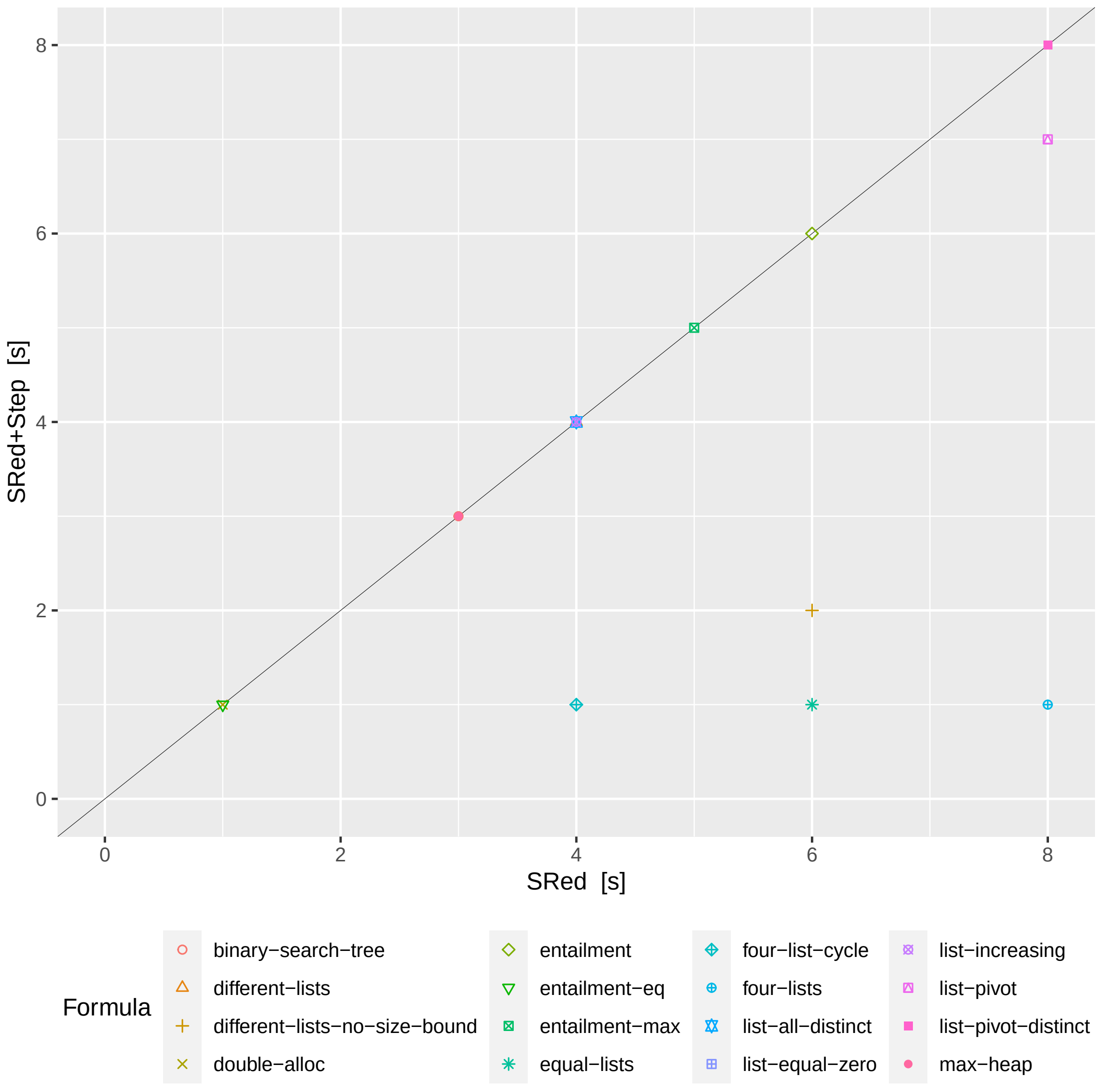


Figure 7.10: Comparison of footprint sizes between SRed and SRed+Step

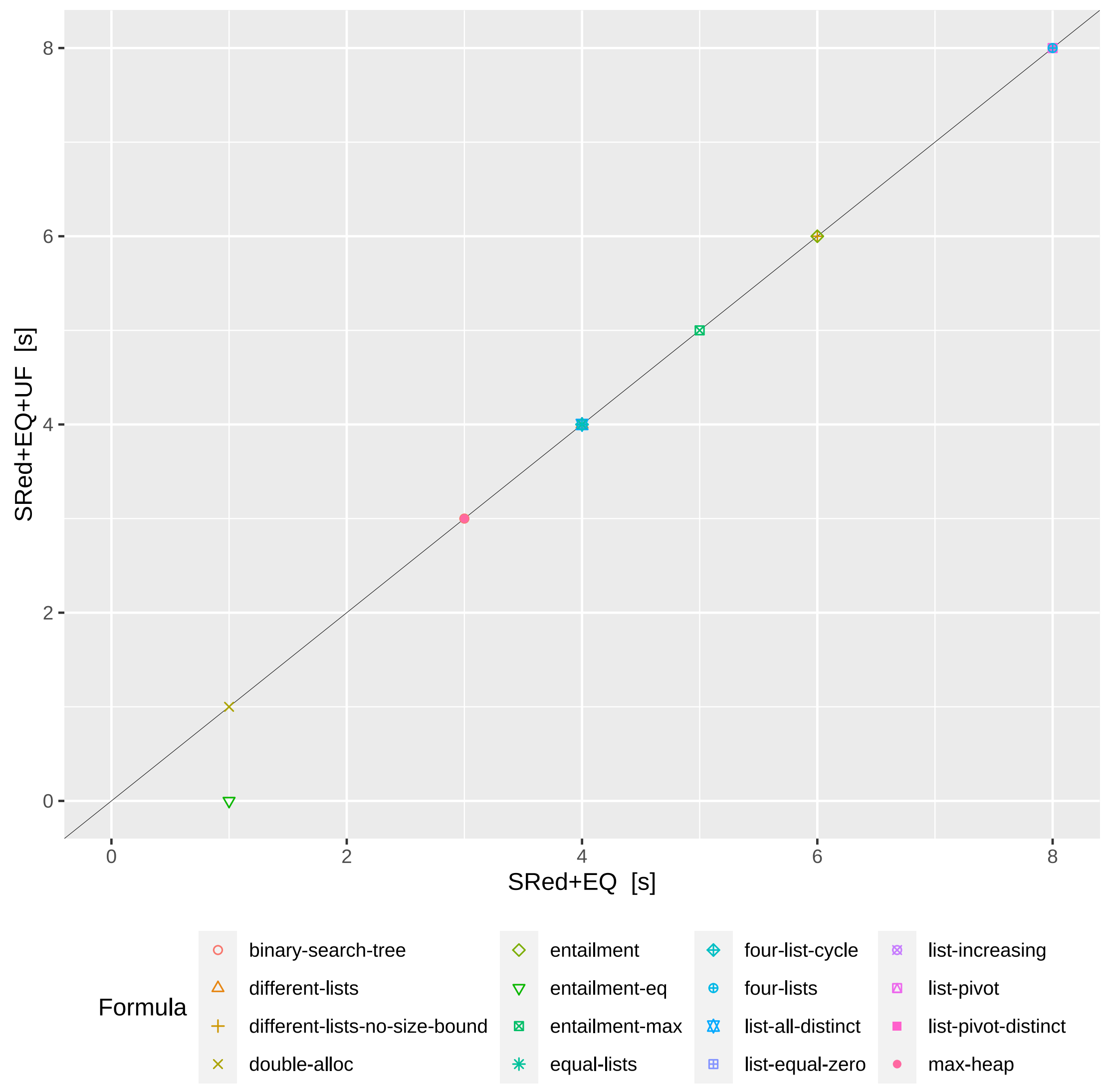


Figure 7.11: Comparison of footprint sizes between SRed+EQ and SRed+EQ+UF

CHAPTER 8

Conclusion

8.1 Summary

Separation Logic is an extension of Hoare Logic that allows modular reasoning about imperative programs that use dynamic data structures. On the other hand Separation Logic is not decidable in general. SL^*_{data} is a fragment that aims for decidability and expressiblity. A decision procedure for SL^*_{data}, that reduces it to SMT was proposed and implemented. We tackled some of the downsides of the original implementation: (1) We integrated it within the state-of-the-art SMT solver Z3 and succeeded implementing a faster encoding, which performs better by orders of magnitude. (2) We extended the implementation by allowing choosing different location and data theories. (3) We proposed optimisations for the decision procedure by looking at low performing edge cases of the original implementation.

The resulting implementation can be used as an assistant for formal proofs or verification of algorithms. Consequently it can act as a backend for an automatic and semi-automatic verification tool. Moreover it might be useful as a platform for further development and research on Separation Logic fragments.

8.2 Further Research

Our implementation is created to be extensible, the obvious choice would be implementing other data structures beside trees and lists. An other possiblity would be adding other predicates that allow to extend expressiblity of SL^*_{data}. Useful predicates could include expressing properties of the data structures (e.g, the size) or relations between them (e.g. list a contains all the elements from list b). Care has to be taken however, as such predicates could easily cost decidability.

Our best performing optimisations all have one in common: they helped decide the formula before encoding or lowering the footprint size required for the encoding. Better heuristics for calculating the upper bounds or finding undecidable cores before encoding to SMT could improve performance even further.

Last but not least a possible connection point is using SL^*_{data} and proving Hoare triplets for common operations on lists and trees such as adding or removing elements, sorting and similar.

Appendix

Example encoding

The following is the formula $x \rightarrow_n y * y \rightarrow_n z$ encoded to SMT in the SMT-LIB file format. The encoded formula can be found in the chapter Encoding SL^*_{data} to SMT in Section 4.2.1.

```
; define a Set to be an Array from Int -> Bool
(define-sort Set () (Array Int Bool))

; locations
(declare-const null Int)
(declare-const x Int)
(declare-const y Int)
(declare-const z Int)

(declare-const x1 Int)
(declare-const x2 Int)
(declare-const x3 Int)

; encoding formulas
(declare-const A1 Bool)
(declare-const A2 Bool)
(declare-const A3 Bool)

(declare-const B1 Bool)
(declare-const B2 Bool)
(declare-const B3 Bool)

; encoding footprints
(declare-const X Set)
(declare-const Y1 Set)
(declare-const Y2 Set)
(declare-const Y3 Set)
```

```

; encoding point-to-functions
(declare-fun f_n (Int) Int)

; encoding helpers
(define-const K_false Set
  ((as const Set) false) )
(define-const K_true Set
  ((as const Set) true) )

(define-const xi Set
  (store
    (store
      (store K_false
        x1 true)
      x2 true)
    x3 true))

(define-fun isElement ((e Int) (X Set)) Bool
  (select X e) )
(define-fun isEmpty ((X Set )) Bool
  (= X K_false) )
(define-fun _union ((X Set ) (Y Set )) Set
  ( (_ map (or (Int Bool) Bool)) X Y) )
(define-fun _intersect ((X Set ) (Y Set )) Set
  ( (_ map (and (Int Bool) Bool)) X Y) )
(define-fun _subset ((X Set ) (Y Set )) Set
  ( (_ map (implies (Int Bool) Bool)) X Y) )
(define-fun isSubset ((X Set ) (Y Set )) Bool
  (= K_true (_subset X Y) ) )

; Delta_SL
(declare-const delta_SL Bool)
(assert (= delta_SL
  (and
    (isSubset X xi))
    (not (isElement null X))
  )
)


; F_smt
```

```
(assert (and A1 B1 delta_SL (= Y1 X) ))
; A1
(assert (= A1
  (and
    A2 A3
    (isEmpty (_intersect Y2 Y3))
  )
))
; B1
(assert (= B1
  (and
    B2 B3
    (= Y1 (_union Y2 Y3))
  )
))
; A2
(assert (= A2
  (= (f_n x) y)
))
; B2
(assert (= B2
  (= Y2 (store K_false x true))
))
; A3
(assert (= A3
  (= (f_n y) z)
))
; B3
(assert (= B3
  (= Y3 (store K_false y true))
))

(check-sat)
(get-model)
```

The output of the z3 SMT solver:

```
sat
(model
  (define-fun y () Int
    3)
  (define-fun A1 () Bool
    true)
  (define-fun x2 () Int
    2)
  (define-fun x1 () Int
    3)
  (define-fun null () Int
    1)
  (define-fun A2 () Bool
    true)
  (define-fun Y1 () (Array Int Bool)
    (_ as-array k!0))
  (define-fun B1 () Bool
    true)
  (define-fun X () (Array Int Bool)
    (_ as-array k!0))
  (define-fun Y3 () (Array Int Bool)
    (_ as-array k!6))
  (define-fun x3 () Int
    0)
  (define-fun delta_SL () Bool
    true)
  (define-fun B3 () Bool
    true)
  (define-fun B2 () Bool
    true)
  (define-fun Y2 () (Array Int Bool)
    (_ as-array k!5))
  (define-fun x () Int
    2)
  (define-fun z () Int
    4)
  (define-fun A3 () Bool
    true)
  (define-fun k!4 ((x!0 Int)) Bool
    (ite (= x!0 3) true
    (ite (= x!0 1) true
    (ite (= x!0 2) true
    (ite (= x!0 0) true
      true)))))
  (define-fun k!1 ((x!0 Int)) Bool
    (ite (= x!0 0) false
      (ite (= x!0 1) false
      (ite (= x!0 2) false
      (ite (= x!0 3) false
        false)))))
    (define-fun f_n ((x!0 Int)) Int
      (ite (= x!0 2) 3
      (ite (= x!0 3) 4
        3)))
    (define-fun k!6 ((x!0 Int)) Bool
      (ite (= x!0 0) false
      (ite (= x!0 1) false
      (ite (= x!0 3) true
      (ite (= x!0 2) false
        false)))))
    (define-fun k!3 ((x!0 Int)) Bool
      (ite (= x!0 3) true
      (ite (= x!0 1) false
      (ite (= x!0 2) true
      (ite (= x!0 0) true
        false)))))
    (define-fun k!0 ((x!0 Int)) Bool
      (ite (= x!0 3) true
      (ite (= x!0 1) false
      (ite (= x!0 2) true
      (ite (= x!0 0) false
        false)))))
    (define-fun k!5 ((x!0 Int)) Bool
      (ite (= x!0 0) false
      (ite (= x!0 1) false
      (ite (= x!0 2) true
      (ite (= x!0 3) false
        false)))))
    (define-fun k!2 ((x!0 Int)) Bool
      (ite (= x!0 0) false
      (ite (= x!0 1) false
      (ite (= x!0 2) true
      (ite (= x!0 3) true
        false)))))
)
```

Formula Input Files

binary-search-tree

$$\begin{aligned}&\mathsf{tree}(x, \{(\mathsf{l}, \alpha < \beta), (\mathsf{r}, \alpha > \beta)\})\\ &\quad \wedge (x \rightarrow_{l,r} l, r) * (l \rightarrow_{l,r} ll, lr)\\ &\quad * (r \rightarrow_{l,r} rl, rr) * (x \rightarrow_d x_{data})\\ &\quad * (l \rightarrow_d l_{data}) * (r \rightarrow_d r_{data})\end{aligned}$$

```
(set-logic SLSTAR)

(declare-const x TreeLoc)
(declare-const l TreeLoc)
(declare-const r TreeLoc)
(declare-const ll TreeLoc)
(declare-const lr TreeLoc)
(declare-const rl TreeLoc)
(declare-const rr TreeLoc)
(declare-const xdata Int)
(declare-const ldata Int)
(declare-const rdata Int)
(assert
      (tree
            (left (< alpha beta))
            (right (> alpha beta))
      x))

(assert
      (sep
            (ptolr x l r)
            (ptolr l ll lr)
            (ptolr r rl rr)
            (ptod x xdata)
            (ptod r rdata)
            (ptod l ldata) ))

(check-sat)
```

different-lists

$$\begin{aligned}
&\mathsf{list}(x, \{(\mathsf{n}, \alpha = \alpha)\}) * \mathsf{list}(y, \{(\mathsf{n}, \alpha \neq A)\}) \\
&\qquad \wedge \neg \mathsf{list}(x, \{(\mathsf{n}, \alpha \neq A)\}) * \mathsf{list}(y, \{(\mathsf{n}, \alpha = \alpha)\}) \\
&\qquad \wedge A = 9001 * (x \to_n b) * (b \to_n \mathbf{null}_{\mathsf{list}}) * (y \to_n d) * (d \to_n \mathbf{null}_{\mathsf{list}}) \\
&\qquad * (x \to_d x_{data}) * (b \to_d b_{data}) * (y \to_d y_{data}) * (d \to_d d_{data})
\end{aligned}$$

```
(set-logic SLSTAR)

(declare-const x ListLoc)
(declare-const y ListLoc)
(declare-const b ListLoc)
(declare-const d ListLoc)
(declare-const xdata Int)
(declare-const ydata Int)
(declare-const bdata Int)
(declare-const ddata Int)
(declare-const A Int)

(define-fun notEqA ( (x Int) ) Bool (distinct x A) )
(assert
    (sep
        (list (unary (= alpha alpha)) x)
        (list (unary (notEqA alpha)) y) ))

(assert (not (sep
            (list (unary (notEqA alpha)) x)
            (list (unary (= alpha alpha)) y) )))
(assert (sep
            (= A 9001)
            (pton x b) (pton b (as null ListLoc))
            (pton y d) (pton d (as null ListLoc))
            (ptod x xdata)
            (ptod b bdata)
            (ptod y ydata)
            (ptod d ddata) ))

(check-sat)
```

different-lists-no-size-bound

$$\begin{aligned}&\mathsf{list}(x,\{(\mathsf{n},\alpha=\alpha)\}) * \mathsf{list}(y,\{(\mathsf{n},\alpha\neq A)\})\\ &\quad\wedge\neg\mathsf{list}(x,\{(\mathsf{n},\alpha\neq A)\}) * \mathsf{list}(y,\{(\mathsf{n},\alpha=\alpha)\})\end{aligned}$$

```
(set-logic SLSTAR)

(declare-const x ListLoc)
(declare-const y ListLoc)
(declare-const b ListLoc)
(declare-const d ListLoc)
(declare-const xdata Int)
(declare-const ydata Int)
(declare-const bdata Int)
(declare-const ddata Int)
(declare-const A Int)

(assert
    (sep (= A 9001)
        (list (unary (= alpha alpha)) x)
        (list (unary (distinct alpha A)) y) ))
(assert
    (not (sep
        (list (unary (distinct alpha A)) x)
        (list (unary (= alpha alpha)) y))))

(check-sat)
```

list-all-distinct

$$\begin{aligned}&\mathsf{list}(a,\{(\mathsf{n},\alpha\neq\beta)\}) \wedge (a\to_n b) * (b\to_n c) * (c\to_n d) * (d\to_n e)\\ &\quad * (a\to_d a_{data}) * (b\to_d b_{data}) * (c\to_d c_{data}) * (d\to_d d_{data})\end{aligned}$$

```
(set-logic SLSTAR)

(declare-const a ListLoc)
```

```
(declare-const b ListLoc)
(declare-const c ListLoc)
(declare-const d ListLoc)
(declare-const e ListLoc)
(declare-const adata Int)
(declare-const bdata Int)
(declare-const cdata Int)
(declare-const ddata Int)

(define-fun not-eq ( (x Int) (y Int) ) Bool
    (not (= x y))
)

(assert (list (next (not-eq alpha beta)) a))

(assert
    (sep
        (pton a b) (pton b c) (pton c d) (pton d e)
        (ptod a adata)
        (ptod b bdata)
        (ptod c cdata)
        (ptod d ddata) ))

(check-sat)
```

list-equal-zero

$$\mathsf{list}(a, \{(\alpha = 0)\}) \wedge (a \to_n b) * (b \to_n c) * (c \to_n d) * (d \to_n e)$$
$$* (a \to_d a_{data}) * (b \to_d b_{data}) * (c \to_d c_{data}) * (d \to_d d_{data})$$

```
(set-logic SLSTAR)

(declare-const a ListLoc)
(declare-const b ListLoc)
(declare-const c ListLoc)
(declare-const d ListLoc)
(declare-const e ListLoc)
(declare-const adata Int)
```

```
(declare-const bdata Int)
(declare-const cdata Int)
(declare-const ddata Int)

(assert (list (unary (= alpha 0)) a))

(assert
    (sep
        (pton a b) (pton b c) (pton c d) (pton d e)
        (ptod a adata)
        (ptod b bdata)
        (ptod c cdata)
        (ptod d ddata) ))

(check-sat)
```

list-increasing

$$\mathsf{list}(a, \{(\mathsf{n}, \alpha < \beta)\}) \wedge (a \to_n b) * (b \to_n c) * (c \to_n d) * (d \to_n e)$$
$$* (a \to_d a_{data}) * (b \to_d b_{data}) * (c \to_d c_{data}) * (d \to_d d_{data})$$

```
(set-logic SLSTAR)

(declare-const a ListLoc)
(declare-const b ListLoc)
(declare-const c ListLoc)
(declare-const d ListLoc)
(declare-const e ListLoc)
(declare-const adata Int)
(declare-const bdata Int)
(declare-const cdata Int)
(declare-const ddata Int)

(assert (list (next (< alpha beta)) a))
(assert
    (sep
        (pton a b) (pton b c) (pton c d) (pton d e)
        (ptod a adata)
```

```
          (ptod b bdata)
          (ptod c cdata)
          (ptod d ddata) ))

(check-sat)
```

max-heap

$$\mathsf{tree}(x, \{(\mathsf{l}, \alpha > \beta), (\mathsf{r}, \alpha < \beta)\}) \wedge (x \rightarrow_{l,r} l, r) * (l \rightarrow_{l,r} ll, lr) \\ * (r \rightarrow_{l,r} rl, rr) * (x \rightarrow_d x_{data}) * (l \rightarrow_d l_{data}) * (r \rightarrow_d r_{data})$$

```
(set-logic SLSTAR)

(declare-const x TreeLoc)
(declare-const l TreeLoc)
(declare-const r TreeLoc)
(declare-const ll TreeLoc)
(declare-const lr TreeLoc)
(declare-const rl TreeLoc)
(declare-const rr TreeLoc)
(declare-const xdata Int)
(declare-const ldata Int)
(declare-const rdata Int)
(assert
      (tree
            (left (> alpha beta))
            (right (> alpha beta))
            x))

(assert
      (sep
            (ptolr x l r)
            (ptolr l ll lr)
            (ptolr r rl rr)
            (ptod x xdata)
            (ptod r rdata)
            (ptod l ldata) ))
(check-sat)
```

list-pivot

$$\begin{aligned}&\mathsf{list}(xm, \{(\alpha < M)\}) * \mathsf{list}(y, \{(\alpha > M)\}) * (m \to_n y) * (m \to_d M)\\ &\quad\wedge (x \to_n a) * (a \to_n b) * (b \to_n m) * (m \to_n y)\\ &\quad * (y \to_n d) * (d \to_n e) * (e \to_n f) * (f \to_n g)\\ &\quad * (x \to_d x_{data}) * (a \to_d a_{data}) * (b \to_d b_{data}) * (m \to_d m_{data})\\ &\quad * (y \to_d y_{data}) * (d \to_d d_{data}) * (e \to_d e_{data}) * (f \to_d f_{data})\end{aligned}$$

```
(set-logic SLSTAR)

(declare-const x ListLoc) ;; head of first list
(declare-const m ListLoc) ;; pivot element
(declare-const y ListLoc) ;; head of second list
(declare-const a ListLoc)
(declare-const b ListLoc)
(declare-const d ListLoc)
(declare-const e ListLoc)
(declare-const f ListLoc)
(declare-const g ListLoc)
(declare-const xdata Int)
(declare-const mdata Int)
(declare-const ydata Int)
(declare-const adata Int)
(declare-const bdata Int)
(declare-const ddata Int)
(declare-const edata Int)
(declare-const fdata Int)
(declare-const M Int) ;; the pivot data

(assert (sep
            (list (unary (< alpha M)) x m)
            (list (unary (> alpha M)) y)
            (pton m y)
            (ptod m M) ))
;; Assert a few pointers as a classical conjunction
;; to force length
(assert
   (sep
    (pton x a) (pton a b) (pton b m) (pton m y)
    (pton y d) (pton d e) (pton e f) (pton f g)
    (ptod x xdata)
    (ptod a adata)
    (ptod b bdata)
    (ptod m mdata)
    (ptod y ydata)
    (ptod d ddata)
    (ptod e edata)
    (ptod f fdata) ))

(check-sat)
```

list-pivot-distinct

$$\begin{aligned}
&\mathsf{list}(x, m, \{(\alpha < M)\}) * \mathsf{list}(y, \{(\alpha > M)\}) * (m \to_n y) * (m \to_d M) \\
&\quad \wedge \mathsf{list}(x, \{(\mathsf{n}, \alpha \neq \beta)\}) \wedge \\
&\quad \wedge (x \to_n a) * (a \to_n b) * (b \to_n m) * (m \to_n y) \\
&\quad * (y \to_n d) * (d \to_n e) * (e \to_n f) * (f \to_n g) \\
&\quad * (x \to_d x_{data}) * (a \to_d a_{data}) * (b \to_d b_{data}) * (m \to_d m_{data}) \\
&\quad * (y \to_d y_{data}) * (d \to_d d_{data}) * (e \to_d e_{data}) * (f \to_d f_{data})
\end{aligned}$$

```
(set-logic SLSTAR)

(declare-const x ListLoc) ;; head of first list
(declare-const m ListLoc) ;; pivot element
(declare-const y ListLoc) ;; head of second list
(declare-const a ListLoc)
(declare-const b ListLoc)
(declare-const d ListLoc)
(declare-const e ListLoc)
(declare-const f ListLoc)
(declare-const g ListLoc)
(declare-const xdata Int)
(declare-const mdata Int)
(declare-const ydata Int)
(declare-const adata Int)
```

```
(declare-const bdata Int)
(declare-const ddata Int)
(declare-const edata Int)
(declare-const fdata Int)
(declare-const M Int) ;; the pivot data

(assert (sep
             (list (unary (< alpha M)) x m)
             (list (unary (> alpha M)) y)
             (pton m y)
             (ptod m M) ))
;; Additionally assert that all elements are distinct
(assert (list (next (distinct alpha beta)) x))
;; Assert a few pointers as a
;; classical conjunction to force length
(assert
   (sep
     (pton x a) (pton a b) (pton b m) (pton m y)
     (pton y d) (pton d e) (pton e f) (pton f g)
     (ptod x xdata)
     (ptod a adata)
     (ptod b bdata)
     (ptod m mdata)
     (ptod y ydata)
     (ptod d ddata)
     (ptod e edata)
     (ptod f fdata) ))

(check-sat)
```

double-alloc

$$(y \to_n x) * (y \to_n \mathbf{null}_{\mathsf{list}})$$

```
(set-logic SLSTAR)
(declare-const y ListLoc)
(declare-const z ListLoc)

(assert
    (sep (pton y z) (pton y (as null ListLoc)))
)
(check-sat)
```

entailment

$$\begin{aligned}&\mathsf{tree}(a) * \mathsf{tree}(b) * \mathsf{tree}(c)\\ &\qquad \wedge \neg(\mathsf{tree}(a) * \mathsf{tree}(b) * \mathsf{tree}(c))\end{aligned}$$

```
(set-logic SLSTAR)

(declare-const a ListLoc)
(declare-const b ListLoc)
(declare-const c ListLoc)

(assert (sep
        (list a)
        (list b)
        (list c)))

(assert (not(sep
        (list a)
        (list b)
        (list c))))

(check-sat)
```

entailment-eq

$$\begin{aligned}&\mathsf{tree}(a) * \mathsf{tree}(b) * \mathsf{tree}(d)\\ &\qquad \wedge \neg(\mathsf{tree}(a) * \mathsf{tree}(b) * \mathsf{tree}(c)) \wedge c = d\end{aligned}$$

```
(set-logic SLSTAR)

(declare-const a TreeLoc)
(declare-const b TreeLoc)
(declare-const c TreeLoc)
(declare-const d TreeLoc)

(assert (sep
          (tree a)
          (tree b)
          (tree d)
        ))

(assert (= c d))

(assert (not(sep
          (tree a)
          (tree b)
          (tree c)
        )))

(check-sat)
```

equal-lists

$$\mathsf{list}(x) * \mathsf{list}(y) * \mathsf{list}(z) * x = y * y = z$$

```
(set-logic SLSTAR)

(declare-const x (ListLoc))
(declare-const y (ListLoc))
(declare-const z (ListLoc))

(assert (sep
      (list x)
      (list y)
      (list z)
     (= x y)
     (= y z)
```

```
    ;(not (= x (as null ListLoc)))
    ))

(check-sat)
```

four-lists

$$\mathsf{list}(x_1) * \mathsf{list}(x_2) * \mathsf{list}(x_3) * \mathsf{list}(x_4)$$

```
(set-logic SLSTAR)

(declare-const x1 (ListLoc))
(declare-const x2 (ListLoc))
(declare-const x3 (ListLoc))
(declare-const x4 (ListLoc))

(assert
    (sep
        (list x1)
        (list x2)
        (list x3)
        (list x4)
    )
)

(check-sat)
```

four-list-cycle

$$(x_1 \to_n x_2) * (x_2 \to_n x_3) * (x_3 \to_n x_4) * (x_4 \to_n x_1)$$

```
(set-logic SLSTAR)

(declare-const x1 (ListLoc))
(declare-const x2 (ListLoc))
(declare-const x3 (ListLoc))
```

```
(declare-const x4 (ListLoc))

(assert
      (sep
            (pton x1 x2)
            (pton x2 x3)
            (pton x3 x4)
            (pton x4 x1)
      )
)

(check-sat)
```

list-not-list

$$\mathsf{list}(x) \land \neg\mathsf{list}(x)$$

```
(set-logic SLSTAR)

(declare-const a ListLoc)
(declare-const b ListLoc)
(declare-const c ListLoc)

(assert ((list a))

(assert (not(list a))

(check-sat)
```

entailment-max

$$\mathsf{list}(a_{tail}) * a \to_{n,d} (a_{tail}, m) * a = b \land \neg\mathsf{list}(a, b, \{\alpha \leq m\})$$

```
(set-logic SLSTAR)

(declare-const a (ListLoc))
(declare-const atail (ListLoc))
```

```
(declare-const m (Int))

(declare-const b (ListLoc))

(assert
    (sep
        (ptod a m)
        (pton a atail)
        (list atail)
        (= a b)
    )
)

(assert (not
    (list (unary (<= alpha m)) a b)
))

(check-sat)
```

List of Figures

List of Tables

Acronyms

Sloth	Separation Logic and Theories. ix, xi, 1–3, 44, 53, 54, 59–61, 66, 91
API	Application Programming Interface. 37
AST	Abstract Syntax Tree. 34, 37, 43
CNF	Conjunctive Normal Form. 44, 59
DAG	Directed Acyclic Graph. 34, 37, 43, 49
FOL	First-Order Logic. 2, 6, 7
SMT	Satisfiability Modulo Theories. xiii, 2, 7–9, 11, 34, 36, 38, 53

Bibliography

[1] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2016.

[2] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories, pages 825–885. Number 1 in Frontiers in Artificial Intelligence and Applications. IOS Press, 1 edition, 2009.

[3] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. A decidable fragment of separation logic. In Kamal Lodaya and Meena Mahajan, editors, FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science, pages 97–109, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[4] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In Kwangkeun Yi, editor, Programming Languages and Systems, pages 52–68, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[5] Josh Berdine, Byron Cook, and Samin Ishtiaq. Slayer: Memory safety for systems-level code. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, Computer Aided Verification, pages 178–183, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[6] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In Klaus Havelund, Gerard Holzmann, and Rajeev Joshi, editors, NASA Formal Methods, pages 3–11, Cham, 2015. Springer International Publishing.

[7] Cristiano Calcagno, Philippa Gardner, and Matthew Hague. From separation logic to first-order logic. In Vladimiro Sassone, editor, Foundations of Software Science and Computational Structures, pages 395–409, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[8] Cristiano Calcagno, Hongseok Yang, and Peter W. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In Proceedings of the 21st Conference on Foundations of Software Technology and Theoretical Computer Science, FST TCS '01, pages 108–119, Berlin, Heidelberg, 2001. Springer-Verlag.

[9] L. de Moura and N. Bjørner. Generalized, efficient array decision procedures. In 2009 Formal Methods in Computer-Aided Design, pages 45–52, 2009.

[10] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, Tools and Algorithms for the Construction and Analysis of Systems, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[11] Jens Katelaan, Dejan Jovanović, and Georg Weissenbacher. A separation logic with data: Small models and automation. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, Automated Reasoning, pages 455–471, Cham, 2018. Springer International Publishing.

[12] Jens Katelaan, Dejan Jovanović, and Georg Weissenbacher. Sloth: Separation logic and theories via small models. In Informal proceedings of the First Workshop on Automated Deduction for Separation Logics (ADSL)., 2018.

[13] Shuvendu Lahiri and Shaz Qadeer. Back to the future: Revisiting precise program verification using smt solvers. In Principles of Programming Languages (POPL '08), page 16. Association for Computing Machinery, Inc., January 2008.

[14] Quang Loc Le, Makoto Tatsuta, Jun Sun, and Wei-Ngan Chin. A decidable fragment in separation logic with inductive predicates and arithmetic. In Rupak Majumdar and Viktor Kunčak, editors, Computer Aided Verification, pages 495–517, Cham, 2017. Springer International Publishing.

[15] Juan Antonio Navarro Pérez and Andrey Rybalchenko. Separation logic modulo theories. In Chung-chieh Shan, editor, Programming Languages and Systems, pages 90–106, Cham, 2013. Springer International Publishing.

[16] Ruzica Piskac, Thomas Wies, and Damien Zufferey. Automating separation logic using smt. In Natasha Sharygina and Helmut Veith, editors, Computer Aided Verification, pages 773–789, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[17] Ruzica Piskac, Thomas Wies, and Damien Zufferey. Automating separation logic with trees and data. In Armin Biere and Roderick Bloem, editors, Computer Aided Verification, pages 711–728, Cham, 2014. Springer International Publishing.

[18] Andrew Reynolds, Radu Iosif, Cristina Serban, and Tim King. A decision procedure for separation logic in smt. In Cyrille Artho, Axel Legay, and Doron Peled, editors, Automated Technology for Verification and Analysis, pages 244–261, Cham, 2016. Springer International Publishing.

[19] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In Proceedings 17th Annual IEEE Symposium on Logic in Computer Science, pages 55–74, July 2002.