Master Thesis

Effective Quantifier-Based Reasoning for Quantitative Deductive Verification

by
Emil Beothy-Elo

First Examiner:
Prof. Dr. Ir. Dr. h.c. Joost-Pieter Katoen

Second Examiner:
apl. Prof. Dr. Thomas Noll

Supervisors:
Philipp Schroer
Darion Haase

Communicated by Joost-Pieter Katoen

Abstract

Caesar is a deductive verifier for probabilistic programs. It builds on modern SMT solvers to automatically check if probabilistic programs conform to their specification. This high degree of automation sometimes comes at the cost of brittle verification. Seemingly unrelated changes in the input program can cause the verifier to hang and verification to fail. These instabilities are often caused by quantifiers that are used in axioms to describe the relevant theories for verification. A common problem here are matching loops – an ill-behaved set of quantifiers that can cause an infinite number of quantifier instantiations by themselves. A large contributor of matching loops are user-defined recursive functions.

A common approach taken by other verifiers is to encode such functions as *limited functions*, limiting the number of recursive instantiations and avoiding matching loops by construction. While they have been proven to be effective, there is little information available about them and they lacked a formal treatment. We present and formally define different limited function encodings used by other verifiers, and subsequently prove that these transformations are sound. Furthermore, we examine how one of the encodings can be modified to obtain finite and constructible counterexamples involving recursive functions.

The presented encodings are implemented in Caesar. We provide guidance on the subtleties that are required for the encodings to work well in practice. Our evaluation shows that the implemented encodings are very effective in eliminating brittleness for problematic programs in Caesar's test suite.

Contents

1. Introduction

Many systems, such as systems performing network communication, inherently include uncertainty. These can be modelled using probabilistic models. For example, the ZeroConf protocol [11] is a dynamic configuration protocol for assigning IP addresses. Among others, it has to deal with the possibility that an assigned IP address is already used in the network and that messages that are sent to perform address resolution might get lost during transmission. The protocol was well studied by the model checking community [4, 10, 25], including probabilistic models such as probabilistic timed automata [27], before its standardization.

ZeroConf can also be modelled as a *probabilistic programs*. These are programs that can additionally sample from probability distributions. Due to their similarity to classical programs, they are an interesting option to express probabilistic models. When analysing probabilistic programs, new questions arise like "what is the expected runtime?" or "what is the expected value of a variable after termination?" Answering and verifying the results to such questions requires different formalisms and tooling than for the classical setting.

The problem of verifying probabilistic programs is even more undecidable than in the classical setting [26]. So one cannot expect to always get an answer. Still, modern SMT solvers like Z3 [15] and cvc5 [5] can often find solutions using sophisticated heuristics. This sparked the development of a number of deductive verification infrastructures for classical programs (including Boogie [29], Viper [35], and why3 [20]) and prove oriented programming languages (including Dafny [28], and F* [43]). Caesar [39] aims to be the probabilistic extension.

Caesar is a quantitative deductive verifier for probabilistic programs. It can automatically formally verify probabilistic programs. Figure 1.1 shows a probabilistic program written in HeyVL, the input language for Caesar. ARP stands for Address Resolution Protocol [3] and is internally used as part of the ZeroConf protocol. For now, we focus on the `arp` procedure. The procedure is taken from a larger HeyVL program that is part of the Caesar test suite, which analyses the probability of ZeroConf successfully assigning a new IP address to a device in one attempt. We are interested in the expected probability of successfully sending an ARP message given a certain number of retries. Therefore, the procedure only contains the retry logic and sending a network message is modelled as sampling from a probability distribution. The procedure sends multiple messages until either a message was successfully transmitted or the number of retries are exhausted. The probability of a message loss is modelled as a constant (zero argument function). Whether a message is lost is determined by the probability distribution `flip(p)`. It returns `true` with probability `p` and `false` with probability `1-p`.

Caesar verifies upper and lower bounds of random variables of probabilistic programs. The random variable is specified in the `post` and the bound in the `pre`. In the case of `proc`, Caesar verifies a lower bound. So in the example, we verify a lower bound for the success probability. The Iverson notation in the post-expectation maps `success` to `0` or `1` depending on if `success` is `false` or `true`. Together with the pre-expectation, the specification therefore requires that `1 - exp(probMessageLost(), triesRemaining)` is a lower bound of the expected value of `success` after executing `arp`. This is indeed the case, since `exp(probMessageLost(), triesRemaining)` is the probability that all `triesRemaining` messages are lost, i.e. `success` being `false`. Hence, the converse probability is `success` being `true`. The power of deductive verification is that the claim is not established for a single or finite set of inputs or message loss probability, but for *all* infinite number of inputs and probabilities.

Figure 1.1 also shows another interesting feature of HeyVL. Similar to classical deductive verifiers [29, 35], users can specify custom theories. The underlying SMT solver (in our case Z3 [15]) does not support exponential functions [21]. Hence, the exponential function is not provided as a built-in function, but it must be defined as an uninterpreted function `exp` together with three axioms (`exp_base`, `exp_step`, `exp_bounded`). The first two axioms make up the standard recursive definition for exponentiation with integer exponents. The third axiom states that exponentiation with a base from the interval $[0, 1]$ always produces a value from the same interval. This is something that the underlying solver cannot derive from the definition itself, but it is required to verify the program.

Using the current Caesar release, the procedure verifies. But after only updating the Z3 solver (the underlying SMT solver used by Caesar) from version 4.12.1 to 4.12.5 the motivating program from Figure 1.1 goes from successfully verifying in well under 1 second to timing out after 5 minutes. The underlying issue here is a so-

```
domain Constants {
  func probMessageLost(): UReal
  axiom messageLostProb probMessageLost() <= 1
}

domain Arith {
  func exp(b: UReal, i: UInt): UReal
  axiom exp_base forall b: UReal. exp(b, 0) == 1
  axiom exp_step forall b: UReal, i: UInt. exp(b, i + 1) == b * exp(b, i)
  axiom exp_bounded forall b: UReal, i: UInt. (b <= 1) ==> (exp(b, i) <= 1)
}

proc arp(triesRemaining: UInt) -> (success: Bool)
  pre 1 - exp(probMessageLost(), triesRemaining)
  post [success]
{
  if triesRemaining == 0 {
    success = false
  } else {
    var messageLost: Bool = flip(probMessageLost())
    if messageLost {
      success = arp(triesRemaining - 1)
    } else {
      success = true
    }
  }
}
```

Figure 1.1: Modelling ARP sub procedure of ZeroConf in HeyVL. It verifies that the success probability is at least $1 - \texttt{probMessageLost}^{\texttt{triesRemaining}}$.

called matching loop. Instantiating the `exp_step` axiom to learn something about exp introduces a new exp-term, which can be again instantiated to obtain more information. Such loops can degrade solver performance to the extent that timeouts occur. The heuristics successfully circumvented the problem in the earlier Z3 version, but not in the later one.

Unexpectedly failing verifications after changing seemingly unrelated parts of the program, updating the version of a tool, or changing the seed of the SMT solver is known as *verification brittleness* [45] or the *butterfly effect* [31]. It is a major usability concern. Often quantifiers, like the ones used in the axiomatization of the exponential function in Figure 1.1, are responsible for the brittleness. There has been extensive research [2, 30, 31, 34] for developing encoding techniques that stabilize the behaviour of quantifiers for program verification, and tool support was developed to help with debugging quantifier related problems in SMT queries [8].

In theory, deductive verifiers like Caesar can provide counterexamples in the case that verification fails. For simpler programs, the SMT solver is complete on the logic fragment that Caesar encodes the problem into. Examples are programs that only contain linear expressions. In these cases, the solver can always either verify the program or provide a counterexample. Verifying more complicated programs requires quantifiers such as in Figure 1.1. This makes the problem in general undecidable. Using the wrong specification from Figure 1.2 Caesar times out, producing no counterexample or a hint of what the problem could be.

```
proc arp(triesRemaining: UInt) -> (success: Bool)
  pre exp(probMessageLost(), triesRemaining) // <- not converse probability
  post [success]
{
  // body ommited
}
```

Figure 1.2: Wrong specification for the `arp` procedure from Figure 1.1.

These are two examples of the currently suboptimal user experience of using quantifiers in Caesar.

- Quantifiers are required for encoding constructs that are not natively supported, but then verification often just hangs, failing to verify correct programs.
- If the program is actually wrong and contains quantifiers. Then the verification also often hangs and produces no counterexample.

Both of these cases look the same to the user. They are left guessing whether the program is actually correct and the SMT solver was just unable to construct a proof, or whether there is an actual error in the program.

These problems cannot be solved in general, since the underlying problem is undecidable. Still, improvements are possible in practice. For example, Dafny ultimately suffers the same problems but, from experience, they occur way less often. The goal of this thesis is to improve the experience of using quantifiers in Caesar by transferring approaches from classical deductive verifiers. Specifically, we focus on an encoding for recursive user-defined functions called *limited functions*. It was originally implemented in Dafny [2] but is also used in other tools like F* [1]. The idea is to guide the SMT solver during the proof search by limiting its options. To gain a better theoretical understanding, we also examine the semi-decision procedures to learn why they sometimes produce unsatisfactory results and the formal properties of the encoding.

We start by giving the required background in Chapter 2, stating how Caesar transforms the verification problem into an SMT query, introducing the logic used by SMT solvers and examining the heuristics used by SMT solvers to deal with quantifiers. Chapter 3 introduces the limited function encodings. We subsequently examine the formal properties of the encodings. Most importantly, we prove that they are sound. Furthermore, an extension of limited functions is introduced that allows for unbounded evaluation of certain safe applications. Counterexamples are examined in Chapter 4. Both the common approach of using potentially unsound `unknown`-models and a modified limited function encoding for obtaining finite models are explored. The limited function encodings were implemented in Caesar. We detail their implementation in Chapter 5 and evaluate their implementation in Chapter 6. We close out in Chapter 7 by mentioning related work and conclude this thesis in Chapter 8 while also exploring future work.

2. Background

This chapter gives an overview and introduction to the topics of interest. It starts at a high level in Section 2.1 with a crash course in deductive verification and Caesar in particular. Then, in Section 2.2, the first-order logic is introduced, into which Caesar encodes the verification problem. This is then passed to an SMT solver. Finally, Section 2.3 discusses the semi-decision procedures that SMT solvers use for quantifiers.

2.1 Deductive Verification and Caesar

We give a brief introduction to probabilistic programs (specifically the language HeyVL) and quantitative deductive verification. Formal definitions and details can be found in [39]. Ultimately, Caesar reduces the verification problem to determining the validity of a *first-order logic* (FOL) formula. This task is then passed on to an SMT solver. We outline the process of transforming it into this final formula. The subsequent work is then concerned with the problem on this lower SMT layer.

2.1.1 Probabilistic Programs and HeyVL

Probabilistic programs are a kind of stochastic model that are particularly interesting due to their relative familiarity to programmers. They are an extension of classical programs that can additionally sample from probability distributions. For example, decisions can be made based on the outcome of a coin flip or by uniformly choosing a value at random from a set of options. Therefore, when starting in an initial state, there is no single final state but a probability distribution of final states.

HeyVL is a programming language for expressing probabilistic programs and Caesar's *intermediate verification language* (IVL). We have already encountered the Bernoulli distribution, from which samples are taken in HeyVL using the `flip` expression. (Figure 1.1). Caesar also supports sampling from other distributions, like uniform distributions or hypergeometric distributions.[1] Since HeyVL is an IVL, it features additional constructs for verification like `assert` and `assume` statements, syntax to specify pre- / post-expectations of procedures and to specify loop invariants, and the ability to define custom theories. A HeyVL program consists of a number of `domain` declarations and procedures. The example program from Figure 1.1 consists of two domain declarations and one procedure. Here, each domain declaration, declares a new function and provides axiom(s) for that function. Axioms are logical statements that are assumed to always hold.

Similar to classical programs, the states of a probabilistic program are mappings from variables to values, i.e.

$$\textbf{States} \coloneqq \{\, \sigma : \textbf{Vars} \to \textbf{Vals} \,\} \ .$$

The `arp` procedure in Figure 1.1 attempts to send a message which may be lost during transmission. If this happens, we try again until all retries have been used up. We are now going to examine the distribution of final states of the `arp` procedure in more detail. Specifically, we are interested in the probability of terminating in a state σ with $\sigma(\texttt{success}) = \texttt{true}$. To this aim, we make an inductive argument using the hypothesis that the probability of `arp` resulting in a state with $\sigma(\texttt{success}) = \texttt{true}$ is $1 - \texttt{exp}(P_{\texttt{messageLost}}, \texttt{triesRemaining})$ (using $P_{\texttt{messageLost}}$ for `probMessageLost()`).

- If we start in an initial state σ_i with $\sigma_i(\texttt{triesRemaining}) = 0$, then the resulting final state σ_f has always $\sigma_f(\texttt{success}) = \texttt{false}$. Therefore, reaching a final state with $\sigma_f(\texttt{success}) = \texttt{true}$ has probability 0. The hypothesis also evaluates to 0 in this case.
- If we start in an initial state σ_i with $\sigma_i(\texttt{triesRemaining}) > 0$, then we lose the message with probability $P_{\texttt{messageLost}}$ when we send it.

[1] https://www.caesarverifier.org/docs/stdlib/distributions

- So, with probability $P_{\texttt{messageLost}}$ we reach the then case and with probability $1 - \texttt{exp}(P_{\texttt{messageLost}}, \texttt{triesRemaining} - 1)$,`true` is assigned to success. Here, we use our hypothesis with $\texttt{triesRemaining} - 1$ to reason about the recursive call of `arp`.
- Conversely, with probability $1 - P_{\texttt{messageLost}}$ the else case is reached and `true` is assigned to success.

The total probability of reaching a final state with $\sigma_f(\texttt{success}) = \texttt{true}$ is thus:

$$\begin{aligned} &\overbrace{P_{\texttt{messageLost}} \cdot (1 - \texttt{exp}(P_{\texttt{messageLost}}, \texttt{triesRemaining} - 1))}^{\text{probability of then case}} + \overbrace{(1 - P_{\texttt{messageLost}}) \cdot 1}^{\text{probability of else case}} \\ &= P_{\texttt{messageLost}} - P_{\texttt{messageLost}} \cdot \texttt{exp}(P_{\texttt{messageLost}}, \texttt{triesRemaining} - 1) + 1 - P_{\texttt{messageLost}} \\ &= 1 - \texttt{exp}(P_{\texttt{messageLost}}, \texttt{triesRemaining}) \end{aligned}$$

Therefore, starting in any initial state, the probability of ending in a final state with $\sigma_f(\texttt{success}) = \texttt{true}$ is `1 - exp(probMessageLost(), triesRemaining)`. This gives an idea how one can reason about expected values of probabilistic programs.

Such manual reasoning can quickly become tedious and error-prone. As a deductive verifier, Caesar aims to provide a solution with a high degree of automation. Caesar can prove the above statement automatically. The shown statement is almost the same as the one stated by the specification in Figure 1.1. Since Caesar always verifies bounds, Figure 1.1 states that `1 - exp(probMessageLost(), triesRemaining)` is a lower bound for the success probability. Caesar can also show that it is an upper bound by using the dual co-constructs. Together, establishing that it is precisely the success probability.

2.1.2 From HeyVL to SMT

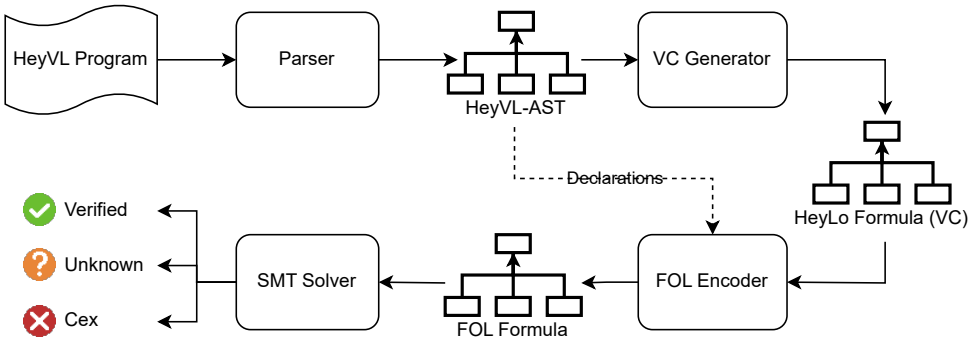


Figure 2.1: Caesar's verification pipeline.

Caesar's architecture resembles that of classical deductive verifiers, like Boogie [29] or Viper[35]. An overview of the relevant parts of the verification pipeline is given in Figure 2.1. First, the input program is parsed, resulting in an abstract syntax tree (AST) for HeyVL. Then a verification condition is generated for it. This verification condition is encoded into a first-order formula that is passed to an SMT solver. Depending on the answer of the SMT solver, Caesar then concludes that the program verified, produces a counterexample (cex), or that no judgment can be made.

HeyVL is the IVL of Caesar and acts as a common interface. Every verification problem is given to Caesar in the form of a HeyVL program, like the one we saw in Figure 1.1. The HeyVL input program is parsed into an abstract syntax tree (AST) before it is further analysed (type checked, etc.).

The main difference from classical deductive verifiers is the use of *expectations* instead of logical conditions for expressing the specification. Expectations map program states to extended unsigned reals $\mathbb{R}^{\infty}_{\geq 0} = \mathbb{R}_{\geq 0} \cup \{\infty\}$ instead of truth values. The complete lattice of expectations $(\mathbb{E}, \sqsubseteq)$ is given by

$$\mathbb{E} \coloneqq \left\{ X : \textbf{States} \to \mathbb{R}^{\infty}_{\geq 0} \right\} \quad \text{with} \quad X \sqsubseteq Y \text{ iff for all } \sigma \in \textbf{States} : X(\sigma) \leq Y(\sigma) \,.$$

The value 0 can be thought of as meaning entirely false and ∞ meaning entirely true. *HeyLo* is a quantitative logic for expressing expectations. In HeyVL, each procedure must be annotated with a quantitative `pre` and a `post` expectation written in HeyLo.

In the setting of classical deductive verification (using the Boolean lattice (**States** $\to \mathbb{B}, \Rightarrow$)), a procedure verifies if the pre-condition in the initial state implies ($\Rightarrow$) the post-condition in the final state. This perspective generalises to the quantitative setting, using the lattice of expectations $(\mathbb{E}, \sqsubseteq)$. A HeyVL procedure verifies if the pre-expectation evaluated in the initial states is a lower bound ($\sqsubseteq$) for the post-expectation evaluated in the final states.

Analogous to the weakest pre-condition in classical deductive verification, there exists the weakest pre-expectation. The weakest pre-expectation $\text{wp}[\![S]\!](\mathit{post})$ maps each initial state σ to the expected value of *post* after executing S on σ. In the case of a `proc`, the pre-expectation must then be a lower bound of the weakest pre-expectation for the procedure to verify. A technical note: Caesar does not directly use the weakest pre-expectation, but the *verification pre-expectation transformer* (vp) which is a computable approximation. The verification condition (VC) is therefore

$$\mathit{pre} \sqsubseteq \text{vp}[\![S]\!](\mathit{post}).$$

This inequality is generated by the VC generator for each procedure and subsequently encoded into a FOL formula by the FOL encoder.

We use the small program in Figure 2.2 to illustrate these two steps of the pipeline. The procedure `coin` takes a natural number and adds 1 to it if a fair coin flip succeeds. We want to verify that `in + 0.5` is a lower bound of the expected return value, i.e. the result is increased by at least 0.5 on average.

```
1  proc coin(in: UInt) -> (out: UInt)
2    pre in + 0.5
3    post out
4  {
5    var x: Bool = flip(0.5)
6    out = in + [x]
7  }
```

Figure 2.2: Adding 1 to a variable if a fair coin flip succeeds increases its expected value by 0.5.

The verification condition for this procedure is

$$\texttt{in} + 0.5 \sqsubseteq \text{vp}[\![\texttt{var x: Bool = flip(0.5); out = in + [x]}]\!](\texttt{out}).$$

The result of the verification pre-expectation transformer can be computed backwards through the sequence of HeyVL statements, starting with the post-expectation [39, Figure 14]. It is again a HeyLo formula. We sketch for the `coin` procedure from Figure 2.2 how the verification pre-expectation is computed. We start with the post-expectation `out`. The assignment in line 6 then replaces `out` with `in + [x]`. In line 5, `x` is assigned with a probability of 50% `true` and with a probability of 50% `false`. Thus, we get $0.5 \cdot (\texttt{in + [true]}) + 0.5 \cdot (\texttt{in + [false]})$. Inlining the conversion by the Iverson brackets, we end up with

$$0.5 \cdot (\texttt{in} + 1) + 0.5 \cdot (\texttt{in} + 0)\ .$$

Hence, the final computed verification condition is

$$\texttt{in} + 0.5 \sqsubseteq 0.5 \cdot (\texttt{in} + 1) + 0.5 \cdot (\texttt{in} + 0)$$

which is equivalent to the first-order formula

$$\texttt{in} + 0.5 \leq 0.5 \cdot (\texttt{in} + 1) + 0.5 \cdot (\texttt{in} + 0)$$

being valid. For the small example program from Figure 2.2, which does not include any other declarations, the VC is the only component of the final FOL formula. In general, the encoding also includes additional information from the declarations in the HeyVL program, like the `exp` function in Figure 1.1 together with the axioms.

For the validity check, Caesar uses the SMT solver Z3 [15]. SMT solvers take first-order logic formulas as input, such as the one we generated. However, they only check for satisfiability, i.e. whether there exists an assignment that satisfies the formula. We are interested in validity, i.e. whether the program is correct for all possible assignments (think states). Thus, the VC is first negated, and we check for unsatisfiability instead. So for the example procedure, it is checked that

$$\texttt{in} + 0.5 \not\leq 0.5 \cdot (\texttt{in} + 1) + 0.5 \cdot (\texttt{in} + 0)$$

is unsatisfiable. If the negation is unsatisfiable (the solver reports `unsat`), then there exists no state such that the program is wrong. Therefore, it must be correct. This approach also has the effect that a satisfying model (the solver reports `sat`) represents a counterexample to the verification problem that can be presented to the user. As discussed before, the satisfiability of the generated formulae is often undecidable. The result is that the solver can also report `unknown` due to the solver heuristics giving up or the exhaustion of resource limits. In this case, we cannot conclude anything. The program might be correct or wrong.

For our purposes, the key takeaway is that the whole verification problem is reduced in many steps to the (un)satisfiability checking of a FOL formula. The quantitative parts have been encoded into a Boolean logic. We will analyse the problem from this lower level.

Since the usual quantifier reasoning is very similar to the classical case, we will sometimes resort to non-probabilistic (Boolean) examples. Boolean formulas can be embedded into HeyLo with the embed expression `?(b)`. It maps `b` to 0 if b is `false` and to ∞ if b is `true`.[2]

2.2 Many-sorted FOL

As we have seen in the previous section, Caesar transforms the verification problem into a satisfiability problem of a classical Boolean first-order logic (FOL) formula. This satisfiability problem is dispatched to an underlying SMT solver. In the following, we will work with/transform these generated FOL formulas. Thus, the semantics of this logic are of key interest. The following section formally introduces the many-sorted first-order logic that underpins all considerations in the rest of the work. It is closely based on the SMT-LIB logic defined by the SMT-LIB Standard Version 2.6 [6]. The standard defines the input language for all major SMT solvers. Some simplifications have been applied to areas that are not necessary for the purposes of this thesis. These differences are highlighted at the end of the section.

To better illustrate the introduced constructs, we will use the formula

$$\xi \quad \coloneqq \quad \mathit{square}(a) - \mathit{square}(b) \approx 7 \land \forall x\colon \mathbf{Int}.\ \mathit{square}(x) \approx x * x$$

as a running example.

We assume a fixed but arbitrary finite set of variables denoted by $\mathbf{Vars} \coloneqq \{ x, y, z, \dots \}$ and $\mathbb{B} \coloneqq \{ \mathbf{true}, \mathbf{false} \}$ denotes the set of Boolean values. A many-sorted logic is not limited to Boolean terms but can have terms of many different sorts. A term can also contain function symbols. A *signature* defines the sorts and function symbols with their possible types that can be used in a term.

> **Definition 2.3. Signatures**
>
> A *signature* is a tuple $\Sigma = \left\langle \Sigma^{\mathrm{S}}, \Sigma^{\mathrm{F}}, \Sigma^{\mathrm{FS}} \right\rangle$ where
>
> - Σ^{S} is a set of sorts, with $\mathbf{Bool} \in \Sigma^{\mathrm{S}}$,
> - Σ^{F} is a set of function symbols, with $\neg, \land, \approx\ \in \Sigma^{\mathrm{F}}$,
> - $\Sigma^{\mathrm{FS}} \subseteq \Sigma^{\mathrm{F}} \times (\Sigma^{\mathrm{S}})^+$ is a left-total[a] relation that assigns a set of possible types to each function symbol such that
>   - $(\neg, \mathbf{Bool}\ \mathbf{Bool}) \in \Sigma^{\mathrm{FS}}$,
>   - $(\land, \mathbf{Bool}\ \mathbf{Bool}\ \mathbf{Bool}) \in \Sigma^{\mathrm{FS}}$, and
>   - $(\approx, \sigma\ \sigma\ \mathbf{Bool}) \in \Sigma^{\mathrm{FS}}$ for all $\sigma \in \Sigma^{\mathrm{S}}$.
>
> A *ranked function symbol* with arity n is a pair $(f, \sigma_1 \cdots \sigma_n\ \sigma) \in \Sigma^{\mathrm{F}} \times (\Sigma^{\mathrm{S}})^+$ written $f\colon \sigma_1 \cdots \sigma_n\ \sigma$ and a *sorted variable* is a pair $(x, \sigma) \in \mathbf{Vars} \times \Sigma^{\mathrm{S}}$ written $x\colon \sigma$. We write $f\colon \sigma_1 \cdots \sigma_n\ \sigma \in \Sigma$ for $f\colon \sigma_1 \cdots \sigma_n\ \sigma \in \Sigma^{\mathrm{FS}}$.

[a] A relation $R \subseteq S \times T$ is left-total iff for every $s \in S$ there exists a $t \in T$ such that $(s, t) \in R$.

For convenience, a ranked function symbol with arity 0 is called a *constant*. In this formalism, there is no distinction between functions and constants. Neither is there a distinction between terms and formulae. Each

[2] https://www.caesarverifier.org/docs/heyvl/procs/#embedding-boolean-specifications

signature always contains a special **Bool** sort and the set of functions

$$\{ \neg\colon \mathbf{Bool}\,\mathbf{Bool},\ \wedge\colon \mathbf{Bool}\,\mathbf{Bool}\,\mathbf{Bool} \}$$

(with fixed meaning) that make up a functionally complete set of logical connectives. For brevity, we will sometimes specify signatures only as $\Sigma = \langle \Sigma^{\mathrm{FS}} \rangle$. Σ^{S} and Σ^{F} then include all sorts/function symbols mentioned by Σ^{FS}.

Example 2.4. Signature of ξ

A possible signature of our running example is $\Sigma_{ex} = \langle \Sigma^{\mathrm{S}}_{ex}, \Sigma^{\mathrm{F}}_{ex}, \Sigma^{\mathrm{FS}}_{ex} \rangle$ with

$$\begin{aligned}
\Sigma^{\mathrm{S}}_{ex} &= \{\, \mathbf{Int}, \mathbf{Bool} \,\} \\
\Sigma^{\mathrm{F}}_{ex} &= \{\, \neg, \wedge, \approx, -, *, \mathit{square}, a, b \,\} \cup \mathbb{Z} \\
\Sigma^{\mathrm{FS}}_{ex} &= \{\, \neg\colon \mathbf{Bool}\,\mathbf{Bool},\ \wedge\colon \mathbf{Bool}\,\mathbf{Bool}\,\mathbf{Bool},\ \approx\colon \mathbf{Bool}\,\mathbf{Bool}\,\mathbf{Bool},\ \approx\colon \mathbf{Int}\,\mathbf{Int}\,\mathbf{Bool} \,\} \\
&\quad \cup \{\, -\colon \mathbf{Int}\,\mathbf{Int}\,\mathbf{Int},\ *\colon \mathbf{Int}\,\mathbf{Int}\,\mathbf{Int},\ \mathit{square}\colon \mathbf{Int}\,\mathbf{Int},\ a\colon \mathbf{Int},\ b\colon \mathbf{Int} \,\} \\
&\quad \cup \{\, z\colon \mathbf{Int} \mid z \in \mathbb{Z} \,\}\,.
\end{aligned}$$

Besides the functions required by Definition 2.3, it also includes integer subtraction, multiplication and a square function. All the integers $(0, 1, -1, \dots)$ are included as integer constants and a, b are also constants of sort **Int**.

Definition 2.5. FOL-terms

Given a signature Σ, the set of Σ-terms $\textsc{Term}_\Sigma$ is defined by the following grammar:

$$\begin{array}{rcll}
t & ::= & x & \text{(variable)} \\
 & \mid & f(t_1, \dots, t_n) & \text{(function application)} \\
 & \mid & \exists x_1\colon \sigma_1 \dots x_n\colon \sigma_n\ \alpha^*.\ t \qquad (n > 0) & \text{(existential quantification)} \\
 & \mid & \forall x_1\colon \sigma_1 \dots x_n\colon \sigma_n\ \alpha^*.\ t \qquad (n > 0) & \text{(universal quantification)} \\
\alpha & ::= & \{t_1, \dots, t_n\} \qquad (n > 0) & \text{(trigger annotation)}
\end{array}$$

where $x, x_1, \dots, x_n$ are variables from **Vars**, f is a function symbol from Σ^{F} and $\sigma_1, \dots \sigma_n$ are sorts from Σ^{S}. For clarity, the parenthesis for function applications with zero arguments are omitted, and some functions are written in infix notation (like, $\wedge, \approx$). Additional parenthesis are used to resolve resulting ambiguities.

We assume that all terms are *well-sorted*[3] and it is unambiguous from the context which version of a function symbol is meant.[4] With these restrictions, each well-sorted term t belongs to exactly one sort σ. We say the term t has/is of sort σ, also written $t\colon \sigma$. For further details, refer to [6, Section 5.2.2].

A term of sort **Bool** is also called *formula*. *Free variables* are variables that are not bound by any surrounding quantifier. A term without free variables is called *closed*. A closed formula is called *sentence*. The set of all Σ-sentences is denoted by $\textsc{Sentence}_\Sigma$. Looking at our running example, ξ is a well-sorted Σ_{ex}-term of sort **Bool**. Since ξ has no free variables, it is also a Σ_{ex}-sentence.

The trigger annotations that can be associated with quantifiers have no semantic meaning in this first-order logic. Instead, they help the SMT solver deal with quantification. These will be discussed in more detail in Section 2.3.1.

One operation we will frequently perform on terms is replacing a function application with another term.

Definition 2.6. Substituting function applications

Substituting function applications of the ranked functions symbol $f\colon \sigma_1 \cdots \sigma_n\ \sigma$ in a term t with another term $t'\colon \sigma$, written

$$t[f(x_1\colon \sigma_1, \dots, x_n\colon \sigma_n) \mapsto t'],$$

[3] Loosely speaking, function applications have the correct number of arguments of the correct sort as dictated by Σ^{FS}.

[4] Namely, we forbid that two ranked function symbols differ only in their result sort, i.e. both $f\colon \sigma_1 \cdots \sigma_n\ \rho, f\colon \sigma_1 \cdots \sigma_n\ \rho' \in \Sigma^{\mathrm{FS}}$ with $\rho \neq \rho'$ is not permitted.

is inductively defined by

$$
\begin{aligned}
x[f(x_1\colon \sigma_1, \ldots, x_n\colon \sigma_n) \mapsto t'] &= x \\
f(t_1, \ldots, t_n)[f(x_1\colon \sigma_1, \ldots, x_n\colon \sigma_n) \mapsto t'] &= && \text{(if } t_i \text{ of sort } \sigma_i \text{ for } i = 1, \ldots, n\text{)} \\
&\quad t'[x_1\colon \sigma_1 \mapsto t_1[f(x_1\colon \sigma_1, \ldots, x_n\colon \sigma_n) \mapsto t'], \ldots, x_n\colon \sigma_n \mapsto t_n[f(x_1\colon \sigma_1, \ldots, x_n\colon \sigma_n) \mapsto t']] \\
g(t_1, \ldots, t_k)[f(x_1\colon \sigma_1, \ldots, x_n\colon \sigma_n) \mapsto t'] &= \\
&\quad g(t_1[f(x_1\colon \sigma_1, \ldots, x_n\colon \sigma_n) \mapsto t'], \ldots, t_k[f(x_1\colon \sigma_1, \ldots, x_n\colon \sigma_n) \mapsto t']) \\
(Qy_1\colon \sigma'_1 \ldots y_k\colon \sigma'_k.\ t)[f(x_1\colon \sigma_1, \ldots, x_n\colon \sigma_n) \mapsto t'] &= && \text{(where } Q \in \{\forall, \exists\}\text{)} \\
&\quad Qy_1\colon \sigma'_1 \ldots y_k\colon \sigma'_k.\ t[f(x_1\colon \sigma_1, \ldots, x_n\colon \sigma_n) \mapsto t']
\end{aligned}
$$

where $t[x_1\colon \sigma_1 \mapsto t_1, \ldots, x_n\colon \sigma_n \mapsto t_n]$ denotes substituting the sorted variables $x_1\colon \sigma_1, \ldots x_n\colon \sigma_n$ with terms $t_1, \ldots, t_n$ in the term t in a capture avoiding manner and is defined as usual. Further, we assume that substituting function applications is also capture avoiding, meaning $x_1\colon \sigma_1, \ldots, x_n\colon \sigma_n$ and the free variables of t' are distinct from the variables $y_1, \ldots, y_k$ of the quantifiers.

Note that the matched arguments can be used in the substituted term t'. This enabled us to swap out the functions called in a term.

Example 2.7. Substituting + for $*$

With $[*(y\colon \mathbf{Int}, z\colon \mathbf{Int}) \mapsto +(y, z)]$, we can replace every application of $*\colon \mathbf{Int}\,\mathbf{Int}\,\mathbf{Int}$ with an application of $+\colon \mathbf{Int}\,\mathbf{Int}\,\mathbf{Int}$ ($*$ and $+$ written in prefix notation):

$$
\begin{aligned}
\xi' &= \xi[*(y\colon \mathbf{Int}, z\colon \mathbf{Int}) \mapsto +(y, z)] \\
&= \mathit{square}(a) - \mathit{square}(b) \approx 7 \wedge (\forall x\colon \mathbf{Int}.\ \mathit{square}(x) \approx *(x, x))[*(y\colon \mathbf{Int}, z\colon \mathbf{Int}) \mapsto +(y, z)] \\
&= \mathit{square}(a) - \mathit{square}(b) \approx 7 \wedge \forall x\colon \mathbf{Int}.\ \mathit{square}(x) \approx *(x, x)[*(y\colon \mathbf{Int}, z\colon \mathbf{Int}) \mapsto +(y, z)] \\
&= \mathit{square}(a) - \mathit{square}(b) \approx 7 \wedge \forall x\colon \mathbf{Int}.\ \mathit{square}(x) \approx +(y, z)[y\colon \mathbf{Int} \mapsto x, z\colon \mathbf{Int} \mapsto x] \\
&= \mathit{square}(a) - \mathit{square}(b) \approx 7 \wedge \forall x\colon \mathbf{Int}.\ \mathit{square}(x) \approx +(x, x)
\end{aligned}
$$

The resulting term ξ' is no longer a Σ_{ex}-term but a Σ'_{ex}-term where

$$
\Sigma'^{\mathrm{FS}}_{ex} = (\Sigma^{\mathrm{FS}}_{ex} \setminus \{ *\colon \mathbf{Int}\,\mathbf{Int}\,\mathbf{Int} \}) \cup \{ +\colon \mathbf{Int}\,\mathbf{Int}\,\mathbf{Int} \}\,.
$$

A *structure* attaches meaning to the sorts and functions symbols introduced by the signature. It defines the universe of possible values, which sorts map to which parts of the universe and gives an interpretation to the functions.

Definition 2.8. Structures

Let Σ be a signature. A Σ-structure is a pair $\mathfrak{A} = \left\langle \mathbf{A}, \square^{\mathfrak{A}} \right\rangle$ consisting of

- a set $\mathbf{A}$, called the universe of $\mathfrak{A}$, that contains $\mathbb{B}$, and
- a mapping $\square^{\mathfrak{A}}$ that interprets
  - each sort symbol $\sigma \in \Sigma^{\mathrm{S}}$ as a subset $\sigma^{\mathfrak{A}} \subseteq \mathbf{A}$, with $\mathbf{Bool}^{\mathfrak{A}} = \mathbb{B}$,
  - each constant $f\colon \sigma \in \Sigma$ as an element $(f\colon \sigma)^{\mathfrak{A}} \in \sigma^{\mathfrak{A}}$, and
  - each ranked function symbol $f\colon \sigma_1 \cdots \sigma_n\, \sigma \in \Sigma$ with $n > 0$ as a function

$$
(f\colon \sigma_1 \cdots \sigma_n\, \sigma)^{\mathfrak{A}} : \left(\sigma_1^{\mathfrak{A}} \times \cdots \times \sigma_n^{\mathfrak{A}}\right) \to \sigma^{\mathfrak{A}},
$$

  where $\neg, \wedge, \approx$ are assigned their canonical meaning.

$\textsc{Structure}_\Sigma$ denotes the set of all Σ-structures.

The extra side conditions ensure that we always have the required buildings blocks for a Boolean logic.

Example 2.9. Σ_{ex}-structure

A possible Σ_{ex}-structure $\mathfrak{A}_{ex} = \left\langle \mathbf{A}_{ex}, \boxed{}^{\mathfrak{A}_{ex}} \right\rangle$ is defined by

$$\begin{aligned} \mathbf{A}_{ex} &= \mathbb{Z} \cup \mathbb{B} \\ \mathbf{Int}^{\mathfrak{A}_{ex}} &= \mathbb{Z} & \mathbf{Bool}^{\mathfrak{A}_{ex}} &= \mathbb{B} \\ (a\colon \mathbf{Int})^{\mathfrak{A}_{ex}} &= 4 & (b\colon \mathbf{Int})^{\mathfrak{A}_{ex}} &= 3 \\ (z\colon \mathbf{Int})^{\mathfrak{A}_{ex}} &= z \quad \text{for } z \in \mathbb{Z} & (\mathit{square}\colon \mathbf{Int}\,\mathbf{Int})^{\mathfrak{A}_{ex}}(z) &= z *_{\mathbb{Z}} z \end{aligned}$$

with the remaining functions being given their canonical meaning. Elements of sort **Int** are mapped to elements of $\mathbb{Z}$. All the integer constants are mapped to themselves. $*_{\mathbb{Z}}$ denotes standard multiplication on $\mathbb{Z}$.

A structure alone is not yet enough to evaluate terms. It only covers the static parts of a term (constant/functions) but we must also be able to evaluate variables. Therefore, an *interpretation* additionally includes a function that tracks the values of variables.

Definition 2.10. Interpretations

Let Σ be a signature. A Σ-interpretation is a pair $\mathfrak{I} = \langle \mathfrak{A}, \mathfrak{v} \rangle$ consisting of

- a Σ-structure $\mathfrak{A}$ with universe $\mathbf{A}$, and
- a function $\mathfrak{v} : (\mathbf{Vars} \times \Sigma^{\mathrm{S}}) \to \mathbf{A}$ that assigns each sorted variable a value that is compatible with its sort, i.e. for all $x\colon \sigma \in \mathbf{Vars} \times \Sigma^{\mathrm{S}}$ it holds that $\mathfrak{v}(x\colon \sigma) \in \sigma^{\mathfrak{A}}$.

We write $f[d_1 \mapsto r_1, \ldots, d_n \mapsto r_n]$ for the function that maps d_i to r_i and is otherwise identical to f, i.e.

$$f[d_1 \mapsto r_1, \ldots, d_n \mapsto r_n](d) = \begin{cases} r_i, & \text{if } d = d_i \text{ for some } i = 1, \ldots, n \\ f(d), & \text{otherwise} \end{cases}$$

The notation $\mathfrak{I}[x_1\colon \sigma_1 \mapsto a_1, \ldots, x_n\colon \sigma_n \mapsto a_n]$ stands for $\langle \mathfrak{A}, \mathfrak{v}[x_1\colon \sigma_1 \mapsto a_1, \ldots, x_n\colon \sigma_n \mapsto a_n] \rangle$ with $a_i \in \sigma_i^{\mathfrak{A}}$.

Definition 2.11. Term evaluation

Let Σ be a signature and $\mathfrak{I} = \langle \mathfrak{A}, \mathfrak{v} \rangle$ a Σ-interpretation. The mapping $[\![\boxed{}]\!]^{\mathfrak{I}}$ maps each term t of sort σ into $\sigma^{\mathfrak{A}}$. $[\![t]\!]^{\mathfrak{I}}$ is inductively defined by

$$\begin{aligned} [\![x]\!]^{\mathfrak{I}} &= \mathfrak{v}(x\colon \sigma) \\ [\![f(t_1, \ldots, t_n)]\!]^{\mathfrak{I}} &= (f\colon \sigma_1 \cdots \sigma_n\, \sigma)^{\mathfrak{A}}([\![t_1]\!]^{\mathfrak{I}}, \ldots, [\![t_n]\!]^{\mathfrak{I}}) \quad \text{if } t_i \text{ has sort } \sigma_i \text{ for } i = 1, \ldots, n \\ [\![\forall x_1\colon \sigma_1, \ldots, x_n\colon \sigma_n.\ t]\!]^{\mathfrak{I}} &= \mathbf{true} \text{ iff} \\ [\![t]\!]^{\mathfrak{I}'} &= \mathbf{true} \text{ for all } (a_1, \ldots, a_n) \in \sigma_1^{\mathfrak{A}} \times \cdots \times \sigma_n^{\mathfrak{A}},\ \mathfrak{I}' = \mathfrak{I}[x_1\colon \sigma_1 \mapsto a_1, \ldots, x_n\colon \sigma_n \mapsto a_n] \\ [\![\exists x_1\colon \sigma_1, \ldots, x_n\colon \sigma_n.\ t]\!]^{\mathfrak{I}} &= \mathbf{true} \text{ iff} \\ [\![t]\!]^{\mathfrak{I}'} &= \mathbf{true} \text{ for some } (a_1, \ldots, a_n) \in \sigma_1^{\mathfrak{A}} \times \cdots \times \sigma_n^{\mathfrak{A}},\ \mathfrak{I}' = \mathfrak{I}[x_1\colon \sigma_1 \mapsto a_1, \ldots, x_n\colon \sigma_n \mapsto a_n] \end{aligned}$$

Since formulas are sentences of sort **Bool**, the evaluation of terms immediately gives us the satisfiability relation as well.

Definition 2.12. Satisfiability

The Σ-interpretation $\mathfrak{I} = \langle \mathfrak{A}, \mathfrak{v} \rangle$ *satisfies* the Σ-formula φ, written $\mathfrak{I} \models \varphi$, iff $[\![\varphi]\!]^{\mathfrak{I}} = \mathbf{true}$. Otherwise, $\mathfrak{I}$ *falsifies* φ, written $\mathfrak{I} \not\models \varphi$. If there exists no Σ-interpretation $\mathfrak{I}$ such that $\mathfrak{I} \models \varphi$, then φ is *unsatisfiable*.

In a Σ-sentence ψ, all variables are bound by quantifiers. When evaluating ψ, the value of a variable is always set by the evaluation rule for quantifiers before it is accessed. Hence, the satisfiability of ψ is independent of the variable assignment $\mathfrak{v}$ and only depends on the Σ-structure $\mathfrak{A}$. We therefore lift the notion of satisfiability from

interpretations to structures and directly write $\mathfrak{A} \models \psi$ or $\mathfrak{A} \not\models \psi$. The Σ-structure $\mathfrak{A}$ is called a *model* of ψ if $\mathfrak{A} \models \psi$.

Example 2.13. Satisfiability of ξ

$\mathfrak{A}_{ex}$ is a model of ξ. This can be validated by checking that evaluating ξ with $\mathfrak{I}_{ex} = \langle \mathfrak{A}_{ex}, \mathfrak{v} \rangle$ results in **true** for any $\mathfrak{v} : (\mathbf{Vars} \times \Sigma^{\mathrm{S}}_{ex}) \to \mathbf{A}_{ex}$:

$$
\begin{aligned}
&\llbracket square(a) - square(b) \approx 7 \land \forall x\colon \mathbf{Int}.\ square(x) \approx x * x \rrbracket^{\mathfrak{I}_{ex}} = \mathbf{true} \\
\text{iff } &\llbracket square(a) - square(b) \approx 7 \rrbracket^{\mathfrak{I}_{ex}} = \mathbf{true} \quad \text{and} \quad \llbracket \forall x\colon \mathbf{Int}.\ square(x) \approx x * x \rrbracket^{\mathfrak{I}_{ex}} = \mathbf{true} \\
\text{iff } &(square\colon \mathbf{Int}\,\mathbf{Int})^{\mathfrak{A}_{ex}}((a\colon \mathbf{Int})^{\mathfrak{A}_{ex}}) -_{\mathbb{Z}} (square\colon \mathbf{Int}\,\mathbf{Int})^{\mathfrak{A}_{ex}}((b\colon \mathbf{Int})^{\mathfrak{A}_{ex}}) = (7\colon \mathbf{Int})^{\mathfrak{A}_{ex}} \\
&\quad \text{and } \llbracket square(x) \approx x * x \rrbracket^{\mathfrak{I}_{ex}[x\colon \mathbf{Int} \mapsto z]} \text{ for all } z \in \mathbb{Z} \\
\text{iff } &(square\colon \mathbf{Int}\,\mathbf{Int})^{\mathfrak{A}_{ex}}(4) -_{\mathbb{Z}} (square\colon \mathbf{Int}\,\mathbf{Int})^{\mathfrak{A}_{ex}}(3) = 7 \\
&\quad \text{and } (square\colon \mathbf{Int}\,\mathbf{Int})^{\mathfrak{A}_{ex}}(\llbracket x \rrbracket^{\mathfrak{I}_{ex}[x\colon \mathbf{Int} \mapsto z]}) = \llbracket x \rrbracket^{\mathfrak{I}_{ex}[x\colon \mathbf{Int} \mapsto z]} *_{\mathbb{Z}} \llbracket x \rrbracket^{\mathfrak{I}_{ex}[x\colon \mathbf{Int} \mapsto z]} \text{ for all } z \in \mathbb{Z} \\
\text{iff } &4 *_{\mathbb{Z}} 4 -_{\mathbb{Z}} 3 *_{\mathbb{Z}} 3 = 7 \quad \text{and} \quad (square\colon \mathbf{Int}\,\mathbf{Int})^{\mathfrak{A}_{ex}}(z) = z *_{\mathbb{Z}} z \text{ for all } z \in \mathbb{Z} \\
\text{iff } &7 = 7 \quad \text{and} \quad z *_{\mathbb{Z}} z = z *_{\mathbb{Z}} z \text{ for all } z \in \mathbb{Z}
\end{aligned}
$$

In the following, we only consider satisfiability of sentences because every formula can be converted into an equivalent sentence by either binding any free variables by an outermost existential quantifier, or replacing free variables with constants. This restriction means that we do not need to separately specify the sort of variables in the formula. This approach is identical to that adopted by SMT-LIB and SMT solvers.

Somewhat surprisingly,

$$\xi' = square(a) - square(b) \approx 7 \land \forall x\colon \mathbf{Int}.\ square(x) \approx x + x$$

from Example 2.7 is also satisfiable. We can construct a model $\mathfrak{A}'_{ex} = \langle \mathbf{A}_{ex}, \cdot^{\mathfrak{A}'_{ex}} \rangle$ for ξ' by interpreting + as multiplication on $\mathbb{Z}$, i.e. $(+\colon \mathbf{Int}\,\mathbf{Int}\,\mathbf{Int})^{\mathfrak{A}'_{ex}}(x, y) = x *_{\mathbb{Z}} y$, and otherwise copying $\mathfrak{A}_{ex}$. This interpretation is counterintuitive, but nothing in the current formalism forces us to choose addition as the meaning of +. The constant $7\colon \mathbf{Int}$ could also be interpreted as 0, even the fact that terms of sort **Int** are evaluated to elements of $\mathbb{Z}$ is currently arbitrary.

This degree of freedom is usually undesirable. It makes it impossible to know what a formula is talking about. We therefore restrict the allowed models through the concept of *theories* to the subset that match our intended meaning.

Definition 2.14. Theory

Let Σ be a signature. A Σ-theory is a set of Σ structures. These structures are called a model of the theory.

Most theories consist of exactly one model, such as the theory of integers (+ is addition, $*$ is multiplication, the integer constants are themselves, ...). Another common possibility is that all the models must satisfy a set of axioms characterizing the theory. Two compatible theories $\mathcal{T}_1, \mathcal{T}_2$ can be combined. Each model of the combined theory must be isomorphic to a model of $\mathcal{T}_1/\mathcal{T}_2$ when projected to the signature of $\mathcal{T}_1/\mathcal{T}_2$. More details can be found in [6, Section 5.4].

Definition 2.15. Satisfiability modulo theory

A Σ-sentence φ is satisfiable modulo the Σ-theory $\mathcal{T}$ iff there exists a structure $\mathfrak{A} \in \mathcal{T}$ such that $\mathfrak{A} \models \varphi$.

We are only interested in satisfiability modulo our chosen background theories. Common theories include *equality with uninterpreted functions* (EUF), which we will cover next, and various arithmetic theories. For program verification, and Caesar in particular, EUF and *nonlinear integer/real arithmetic* (NIA, NRA) with quantifiers are relevant. The complexity of the satisfiability problem depends on the chosen theories. On its own, NRA is decidable [44] but NIA is undecidable [33]. Therefore, The decision problem using the combination of these theories with quantifiers is, in general, undecidable. However, modern SMT solvers are often able to find solutions using sophisticated heuristics.

Readers familiar with the SMT-LIB standard will miss some features and constructs. Parts that are not the focus of this work were simplified. Theses are mainly concepts related to sorts. Sorts with arity other than 0,

```
domain Pair {
  func pair(x: Int, y: Int): Pair

  func select0(p: Pair): Int
  axiom select0_def forall x: Int, y: Int. select0(pair(x, y)) == x

  func select1(p: Pair): Int
  axiom select1_def forall x: Int, y: Int. select1(pair(x, y)) == y
}
```

Figure 2.18: Axiomatization of a pair data structure with two associated operations `select0` and `select1`.

parametric sorts and algebraic data-types are omitted. The well-sortedness of terms is assumed, but not checked. The constructs `let` and `match` are also omitted.

2.2.1 Equality with Uninterpreted Function

The theory of *equality with uninterpreted functions* (EUF) allows for encoding custom theories, which makes it very powerful. In the following it is assumed, that EUF is always included as a base theory. It consists of a finite number of uninterpreted sorts and uninterpreted function symbols with arbitrary arity, that can be flexibly added. It also includes $\approx$ (equality) for each sort, as defined previously in Definition 2.3. Additionally, $\approx$ must be a *congruence relation*, meaning it behaves nicely with function application.

Definition 2.16. Congruence

An equivalence relation $\cong$ is called a *congruence relation* if it is preserved by function application, i.e. for all function symbols $f : \sigma_1 \cdots \sigma_n$ and terms $t_1, u_1 : \sigma_1, \ldots, t_n, u_n : \sigma_n$

$$t_i \cong u_i \text{ for all } 1 \leq i \leq n \quad \text{iff} \quad f(t_1, \ldots, t_n) \cong f(u_1, \ldots, u_n) \,.$$

Definition 2.17. Congruence closure

Let s, t be terms, f a function symbol, and $\cong$ an equivalence relation. The *congruence closure* $\cong^*$ is the smallest relation closed under congruence that contains $\cong$. Equivalently, $\cong^*$ is the smallest relation satisfying

1. if $s \cong t$, then also $s \cong^* t$,
2. if $s = f(s_1, \ldots, s_n)$ and $t = f(t_1, \ldots, t_n)$ with $s_i \cong^* t_i$ for all $1 \leq i \leq n$, then also $s \cong^* t$,
3. and if $f(s_1, \ldots, s_n) \cong^* f(t_1, \ldots, t_n)$, then also $s_i \cong^* t_i$ for all $1 \leq i \leq n$.

The congruence closure extends an equivalence relation to a congruence relation. We use the notion of the congruence closure $\approx^*$ to explicitly express when congruence or transitivity is used in an argument. That $\approx$ must be congruence relation is the only restriction given by EUF. Therefore, the sorts and functions from EUF are called uninterpreted. The theory does not restrict how they can be interpreted. In contrast, the sort **Int** and function $+$: **Int Int Int** from NIA are interpreted. The theory fixes their meaning.

The absence of any predefined concepts makes EUF very flexible. Many custom theories can be encoded into EUF with the help of axioms. We already saw this in our running example. The formula ξ contains the uninterpreted function *square*: **Int Int**, which is defined by the axiom $\forall x$: **Int**. $square(x) \approx x * x$ to be the square of integers.

Like other deductive verifiers (e.g. Viper [35] and [28]), Caesar exposes this functionality to its users, providing a flexible extension point that allows users to provide the axiomatisation of custom theories required for their use case.

Example 2.19. Encoding a pair ADT using EUF

Consider the HeyVL code in Figure 2.18 that is an encoding of a pair of integers. The code declares a new uninterpreted sort `Pair`, three uninterpreted functions (`pair`, `select0`, and `select1`) and two axioms. The `domain Pair` block introduces a new uninterpreted sort named `Pair`. This sort contains all values of our pair ADT. The constructor of the pair is represented by the function `pair`.

The operations are always defined in two steps. First, the function with its signature is declared. Second, an axiom is added, that states a fact that the function has to always fulfil. In Caesar, an axiom declaration has the form `axiom <name> <body>`. The name is only for documentation, and the body usually starts with a universal quantifier. Here, the axiom `select0_def` states that applying `select0` to the pair constructor is the same as the first element of the pair. Analogously, `select1_def` does this for `select1`. Note that, although these are uninterpreted functions, the given axioms fully define the operations in this example. There is always an axiom that specifies the result for each possible argument. This fact must not necessarily hold for other examples.

2.3 Quantifier Instantiation

This work focuses on quantifier reasoning. We have observed that quantifier handling is often the reason for non-termination in Caesar and we believe that it can be significantly improved. The following section gives a brief primer on how SMT solvers support universal quantifiers (in the following often just called quantifiers). The core satisfiability procedure only works with closed quantifier-free formulae, also called *ground terms*. At opportune moments, information from the quantified formulae is incorporated into the ground terms. This operation is called *quantifier instantiation*. We focus on two strategies for quantifier instantiation:

E-matching. *E-matching-based quantifier instantiation* is a syntax-guided heuristic that attempts to find relevant values for the quantified variables. It can only prove unsatisfiability.

MBQI. *Model-based quantifier instantiation* is a heuristic that attempts to extend partial models for the quantifier-free part to also include the quantifiers. It can be used to prove satisfiability as well as unsatisfiability.

By default, both are used in parallel by Z3. Many deductive verifiers disable MBQI[5] and rely solely on E-matching for quantifier reasoning. We investigate the role that MBQI plays for verification in Caesar in Section 5.2 and how to leverage it for counterexamples in Chapter 4.

2.3.1 E-matching

E-matching-based quantifier instantiation (often just called E-matching) is a procedure used by SMT solvers to check if a first-order formula is unsatisfiable. It gets a sentence in conjunctive normal form with only universal quantifiers as an input. Existential quantifiers can be eliminated by Skolemisation [16, Section 5.3]. It either determines that the sentence is unsatisfiable (returning `unsat`), gives up with no result (returning `unknown`), or does not terminate. It uses the sub procedure E-MATCH to find instantiations such that the newly added terms already existing mention ground terms. This increases the chance of producing a contradiction and therefore proving `unsat`.

Algorithm 1 is a high-level overview of E-matching-based quantifier instantiation. It is simplified to only expect sentences with a single conjunct and quantifier.

The conjuncts without a quantifier are the initial ground terms (line 1). If these ground terms are already unsatisfiable, the complete formula is unsatisfiable (lines 3-5). Otherwise, the ground terms might be unsatisfiable if we also consider the quantified formulae. The first step of incorporating universally quantified formulae is the observation that $\forall \bar{x} \colon \bar{\sigma}.\ \varphi$ can be thought of as the infinite conjunction $\bigwedge_{\bar{t} \colon \bar{\sigma}} \varphi[\bar{x} \colon \bar{\sigma} \mapsto \bar{t}]$. Here, φ denotes a quantifier-free formula and $\bar{t}$ ranges over all possible substitutions of $\bar{x}$. To reduce the conjunction to a finite one, only instances of $\varphi[\bar{x} \colon \bar{\sigma} \mapsto \bar{t}]$ are considered that are "relevant" to the current set of ground terms (line 6). The term $\varphi[\bar{x} \colon \bar{\sigma} \mapsto \bar{t}]$ is called an *instantiation* of the quantifier and is quantifier-free. It is added as a new conjunct to the ground terms (line 10) and the satisfiability procedure can again check if the new set of ground terms is unsatisfiable in the next loop iteration. If the heuristic is unable to determine a new instantiation, then it is unable to make any further progress and terminates with `unknown` (line 7-9).

During the proof search, the ground terms are stored in an E-graph. A data structure that exploits the congruence closure to efficiently store large numbers of terms and (dis)equalities between them [16, 46]. The E-MATCH procedure searches for a pattern p with variables $\bar{x} \colon \bar{\sigma}$ in the E-graph. A resulting match comes with a substitution $[\bar{x} \colon \bar{\sigma} \mapsto \bar{t}]$ that specifies how the variables need to be substituted, such that $p[\bar{x} \colon \bar{\sigma} \mapsto \bar{t}]$ appears in the E-graph (modulo equality) [14].

[5] See Boogie (Dafny backend) https://github.com/boogie-org/boogie/blob/6d8896fa476d0b623a08e887cbc11d3b56e9ec4d/Source/Provers/SMTLib/Z3.cs#L63 or Silicon (one of the Viper backends) https://github.com/viperproject/silicon/blob/19bbbb524c4d2555abdbde68870a52f5abec49b4/src/main/resources/z3config.smt2#L23

Algorithm 1: E-matching-based quantifier instantiation (sketch), based on [16, Section 5.1]

Input : A sentence of the form $\psi \wedge \forall \bar{x} : \bar{\sigma}.\ \varphi$, where ψ, φ are quantifier free
Output: Either `unsat` if the unsatisfiability of the sentence could be established or `unknown` otherwise

1 $G \leftarrow \{ \psi \}$;
2 **while** *true* **do**
3 **if** G *is unsatisfiable* **then**
4 **return** `unsat`;
5 **end**
6 $\bar{t} \leftarrow$ E-MATCH$(\forall \bar{x} : \bar{\sigma}.\ \varphi, G)$ such that $\varphi[\bar{x} : \bar{\sigma} \mapsto \bar{t}]$ is not in G;
7 **if** *no such terms* $\bar{t}$ *exists* **then**
8 **return** `unknown`;
9 **end**
10 $G \leftarrow G \cup \{ \varphi[\bar{x} : \bar{\sigma} \mapsto \bar{t}] \}$;
11 **end**

For E-MATCH to work, each quantifier is associated with a non-empty set of *triggers*. A trigger consists of one or more *patterns*. A trigger matches if all of its patterns match, and a quantifier can be initiated if one of its pattern matches. The terms used as patterns can be arbitrary, the only restrictions being that a pattern may not be just a variable and that the free variables of all patterns belonging to a trigger must be precisely the variables of the quantifier. A quantifier often has a single trigger with a single pattern. Hence, these terms are sometimes used interchangeably.

If no trigger is specified for the quantifier, the SMT solvers have heuristics to select triggers themselves, usually a sub-term of φ mentioning all quantified variables. It is generally advisable to annotate quantifiers with triggers to have more control over the quantifier instantiation and to stabilize behaviour [31]. In logic formulae, we denote triggers in curly braces after the quantified variables (compare Definition 2.5) like

$$\forall x : \mathbf{Int}\ y : \mathbf{Int}\ \{pair(x, y)\}.\ sel_0(pair(x, y)) \approx x$$

and in HeyVL they are specified with the `@trigger` annotation like

```
1  axiom select0_def forall x: Int, y: Int @trigger(pair(x, y)).
2    select0(pair(x, y)) == x
```

The introduction and application of a second formal semantic that models the effect of the triggers on the SMT solvers [23, 18] was outside the scope of this thesis.

Example 2.20. E-matching and congruence

We build on a variant of the program from Figure 2.18 that encodes a pair data type. In this example, we write sel_0 instead of `select0`. The signature we are considering is

$$\Sigma_P = \langle \{\ pair : \mathbf{Int}\ \mathbf{Int}\ \mathbf{Pair},\ sel_0 : \mathbf{Pair}\ \mathbf{Int},\ a : \mathbf{Int},\ b : \mathbf{Int},\ c : \mathbf{Int},\ d : \mathbf{Int}\ \} \rangle\ .$$

Our goal is, given the axiom

$$A_1 \quad := \quad \forall x : \mathbf{Int}\ y : \mathbf{Int}\ \{pair(x, y)\}.\ \underbrace{sel_0(pair(x, y)) \approx x}_{F}$$

to prove that

$$C_1 \quad := \quad pair(a, b) \approx pair(c, d) \implies a \approx c$$

is valid using E-matching. Proving that the implication is valid given the axiom is the same as proving that $A_1 \wedge \neg C_1$ is unsatisfiable.

- Since $\neg C_1$ is equivalent to $pair(a, b) \approx pair(c, d) \wedge a \not\approx c$ and there are no other quantifier-free formulae, the initial ground terms are $G_1 = \{\ pair(a, b) \approx pair(c, d), a \not\approx c\ \}$ (line 1).
- *Loop iteration 1:*
  - G_1 by itself is not yet unsatisfiable (lines 3-5).

- E-matching the trigger $pair(x, y)$ of A_1 against G_1 yields the possible match $pair(a, b)$ with the substitution $\theta_1 = [x\colon \mathbf{Int} \mapsto a, y\colon \mathbf{Int} \mapsto b]$ (line 6).
- Adding the found instantiations to the ground terms results in (line 10)

$$\begin{aligned} G_2 &= G_1 \cup \{\, F\theta_1 \,\} \\ &= \{\, pair(a,b) \approx pair(c,d),\ a \not\approx c,\ sel_0(pair(a,b)) \approx a,\ \} \end{aligned}$$

- *Loop iteration 2:*
  - G_1 by itself is not yet unsatisfiable (lines 3-5).
  - Matching the same trigger results in another match $\theta_2 = [x\colon \mathbf{Int} \mapsto c, y\colon \mathbf{Int} \mapsto d]$ (line 6).
  - Adding the instantiations to the ground terms results in

$$\begin{aligned} G_3' &= G_2 \cup \{\, F\theta_2 \,\} \\ &= \{\, pair(a,b) \approx pair(c,d),\ a \not\approx c,\ sel_0(pair(a,b)) \approx a,\ sel_0(pair(c,d)) \approx c \,\} \end{aligned}$$

    Since $\approx$ is a congruence relation, we can conclude additional equalities (line 10):

$$\begin{aligned} & pair(a,b) \approx pair(c,d) \\ \text{implies}\quad & sel_0(pair(a,b)) \approx^* sel_0(pair(c,d)) \\ \text{implies}\quad & a \approx^* c \qquad \text{(by } sel_0(pair(a,b)) \approx a \text{ and } sel_0(pair(c,d)) \approx c\text{)} \end{aligned}$$

    The ground terms are extended with these equalities, such that $\approx$ remains a congruence relation, resulting in

$$\begin{aligned} G_3 &= G_3' \cup \{\, sel_0(pair(a,b)) \approx sel_0(pair(c,d)),\ a \approx c \,\} \\ &= \{pair(a,b) \approx pair(c,d),\ a \not\approx c,\ sel_0(pair(a,b)) \approx a,\ sel_0(pair(c,d)) \approx c, \\ &\qquad sel_0(pair(a,b)) \approx sel_0(pair(c,d)),\ a \approx c\} \end{aligned}$$

- *Loop iteration 3:*
  - G_3 is now inconsistent, since it contains $a \not\approx c$ as well as $a \approx c$. Hence, `unsat` is returned (line 4).

Therefore, $A_1 \wedge \neg C_1$ is unsatisfiable and $A_1 \implies C_1$ is valid.

Matching loops

The chosen triggers play a crucial role in determining the effectiveness of E-matching based quantifier instantiation. For example, if the trigger of A_1 from Example 2.20 is replaced with $sel_0(pair(y, x))$ we get

$$\begin{aligned} & \forall x\colon \mathbf{Int}\ y\colon \mathbf{Int}\ \{sel_0(pair(y,x))\}.\ sel_0(pair(x,y)) \approx x \\ \text{implies}\quad & pair(a,b) \approx pair(c,d) \implies a \approx c \end{aligned}$$

The initial ground terms are still $G_1 = \{\, pair(a,b) \approx pair(c,d), a \not\approx c \,\}$ but the new trigger does not match any of these ground terms. Therefore, the algorithm terminates with `unknown` and the proof fails.

Besides proofs failing because the required instantiations do not happen due to overly restrictive triggers, they can also fail if the triggers are too liberal and too many instantiations are performed. First, if the solver is occupied with E-matching, it cannot perform other necessary tasks like theory solving. Second, each instantiation adds more ground terms, slowing down all operations. This can lead to poor verification performance and eventually timeouts.

A common problem in this regard are so-called *matching loops*. In its simplest form, a matching loop occurs when the new terms produced by an instantiation match the trigger of that quantifier, potentially resulting in another match and instantiation. These new terms also match again and so on. In general, the loop can consist of multiple instantiations of different quantifiers and use equalities between terms.

Example 2.21. Matching loop

Matching loops are easily introduced when encoding recursive structures. For example, the exponential function frequently occurs in the probabilistic setting but is currently not provided by Caesar. As we have already seen in Figure 1.1, it can be encoded as a user-defined function. The following code shows a slightly simpler encoding using the `ite` (if-then-else) expression such that only a single defining axiom is required.

```
1  func exp(b: UReal, x: UInt): UReal
2  axiom exp_def forall b: UReal, x: UInt @trigger(exp(b, x)).
3    exp(b, x) == ite(x == 0, 1, b * exp(b, x - 1))
```

The `ite` expression takes three expressions. If the first Boolean expression evaluates to `true`, then it yields the second expression; otherwise the third expression is yielded.

Using this axiomatisation of exp we want to show that

$$n > 0 \implies \exp(a, n) \approx \exp(a, n-1) * a$$

is valid, i.e. $n > 0 \wedge exp(a, n) \not\approx \exp(a, n-1) * a$ is unsatisfiable. The initial ground terms are

$$G_1 = \{\, n > 0,\ \exp(a, n) \not\approx \exp(a, n-1) * a \,\}\,.$$

From here, we examine two different scenarios.

Senario 1 (matching loop). The solver matches $\exp(a, n-1)$ with the pattern and thus increases the ground terms to

$$G_2^1 = G_1 \cup \{\, \exp(a, n-1) \approx ite(n-1-1 \approx 0, 1, a * \exp(a, n-1-1)) \,\}\,.$$

Such an instantiation always adds a new exp-term which the solver can match in the next iteration. Repeatedly applying this creates the sets of ground terms:

$$G_3^1 = G_2^1 \cup \{\, \exp(a, n-1-1) \approx ite(n-1-1-1 \approx 0, 1, a * \exp(a, n-1-1-1)) \,\}$$
$$G_4^1 = G_3^1 \cup \{\, \exp(a, n-1-1-1) \approx ite(n-1-1-1-1 \approx 0, 1, a * \exp(a, n-1-1-1-1)) \,\}$$
$$\vdots$$

This chain can continue indefinitely, resulting in non-termination. The proof fails.

Senario 2 (successful proof). The solver matches the term $\exp(a, n)$ and adds the resulting instantiation to the ground terms, resulting in

$$G_2^2 = \{\, n > 0,\ \exp(a, n) \not\approx \exp(a, n-1) * a,\ \exp(a, n) \approx ite(n \approx 0, 1, a * \exp(a, n-1)) \,\}\,.$$

Using theory solving, the solver concludes that $\exp(a, n) \approx a * \exp(a, n-1)$ must hold, which makes the ground terms unsatisfiable. The proof succeeds.

The presence of a matching loop alone is not enough to cause non-termination. The solver heuristics must choose this "bad" path of instantiations. Solver developers are aware of the problem and solvers have heuristics that try to avoid the problem [16, Section 5] but from the perspective of a general purpose SAT solver, deep instantiations chains might be necessary so they cannot be ruled out. We already saw that in the motivating example (Figure 1.1) which has a matching loop slumbering in the exp axiomatisation. The previous Z3 version did not run into the loop, but after updating with slightly changed heuristics, the newer Z3 version does.

As a result, matching loops can easily go undetected in a program until the program is slightly changed, affecting how the heuristics work, or a different version of the SMT solver is used. This makes them difficult to debug. Matching loops are a common reason for verification bitterness [31].

SMTscope

The SMTscope[6] tool (based on the Axiom Profiler [8]) was developed to detect matching loops and generally help to debug poor solver performance. It works by analysing traces generated by Z3 during the proof search. The GUI presents different information regarding the solver run, including in what order quantifiers are initiated

[6] https://github.com/viperproject/smt-scope

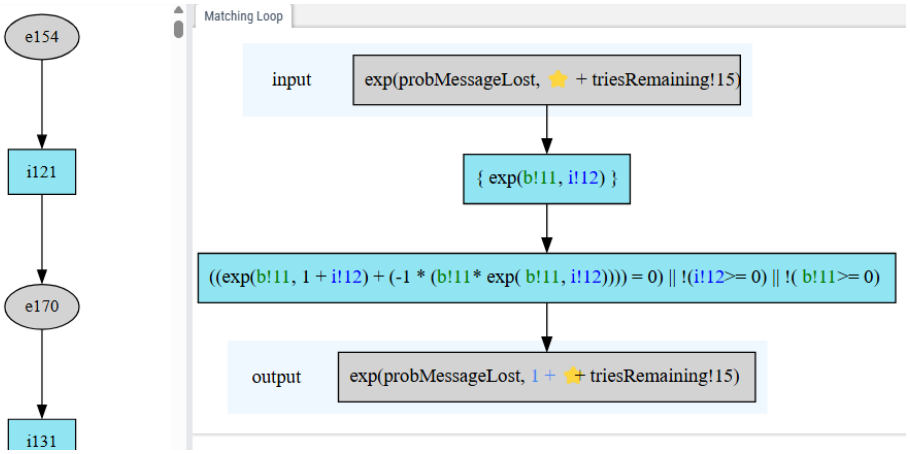


Figure 2.22: Screenshot showing a part of the SMTscope GUI. On the left is part of the instantiation graph and on the right the generalisation of a matching loop.

and the instantiation graph. The screenshot in Figure 2.22 shows part of the tools GUI. There, a trace from a non-terminating verification run of the program in Figure 1.1 was loaded. On the left-hand side, part of the instantiation graph is shown. It is a graph where an edge goes from a quantifier (box) to a term (grey ellipse) if the term was produced by the instantiation of the quantifier, and an edge goes from a term (grey ellipse) to a quantifier (box) if it was matched by the quantifier trigger, causing an initiation. Every quantifier gets a different colour for easy identification. From the small section shown, we can already spot repeating behaviour. The blue quantifier is instantiated by a term that was previously created by instantiating the blue quantifier. This graphical representation helps to identify matching loops.

The tool also comes with functionality to automatically find and generalize matching loops. This can be seen on the right-hand side of the screenshot (Figure 2.22). A term that has the shape shown in the input box causes an instantiation, which creates a new term that has the shape shown in the output box. The shape of the output term also matches the input shape. Thus, we have a matching loop.

SMTscope is useful for determining if a matching loop might be the reason for non-termination. In such a case, it also helps to determine which quantifiers and triggers are to blame. It was invaluable for analysing and tracking down quantifier-related problems while working on this thesis.

2.3.2 MBQI

Model-based quantifier instantiation (MBQI) is a procedure used by SMT solvers to find models for FOL formulae containing quantifiers or to show that they are unsatisfiable. It gets a sentence in conjunctive normal form with only universal quantifiers as an input. MBQI either determines that the sentence is unsatisfiable (returning `unsat`), finds a model that satisfies the sentence including the quantifiers (returning `sat` with the model), or does not terminate. It works by checking if a model for the ground terms (ground term model) also satisfies the quantifier. This is done by checking that it does not permit a counter example. If a counter example is found, it is added to the ground terms, such that a new ground term model is found on the next iteration.

Algorithm 2 is a high-level overview of model-based quantifier instantiation. It is simplified to only expect sentences with a single conjunct and quantifier.

The conjuncts without a quantifier are the initial ground terms (line 1). If these ground terms are already unsatisfiable, the complete formula is unsatisfiable (lines 3-5). Otherwise, there must exist a model that satisfies the current ground terms (line 6). This model of the ground terms need not be a model of the whole input formula. It is only guaranteed to satisfy the ground terms and might violate the quantifier $\forall \bar{x} : \bar{\sigma}.\ \varphi$. To check if the model also satisfies the quantifier, the body of the quantifier φ is partially evaluated with the model (everything gets fixed by $\mathfrak{A}$, except the variables $\bar{x} : \bar{\sigma}$), here written $\varphi^{\mathfrak{A}}$. If the negation of the resulting formula is unsatisfiable, then it must hold for all $\bar{x} : \bar{\sigma}$, i.e. $\mathfrak{A}$ also satisfies the quantifier. Therefore, the complete formula is satisfiable with $\mathfrak{A}$ and the algorithm returns `sat` (lines 7-9). If $\neg\varphi^{\mathfrak{A}}$ is not unsatisfiable, then there exists again a model that satisfies it (line 10). It is a counterexample that specifies for which $\bar{x}$ the ground term model $\mathfrak{A}$ violated the quantifier. By instantiating the quantifier with these values and adding it to the ground terms, it is guaranteed that in the next loop iteration the ground term model will not violate the quantifier for the same $\bar{x}$ (lines 11-12).

Algorithm 2: Model-based quantifier instantiation (sketch) [9, Section 6.1.4]

Input : A sentence of the form $\psi \wedge \forall \bar{x} \colon \bar{\sigma}.\ \varphi$, where ψ, φ are quantifier free
Output: Either `sat` with a model satisfiying the sentence or `unsat` if the sentence is unsatisfiable

1 $G \leftarrow \{ \psi \}$;
2 **while** *true* **do**
3 **if** G *is unsatisfiable* **then**
4 **return** `unsat`;
5 **end**
6 $\mathfrak{A} \leftarrow$ model satisfying G, i.e. $\mathfrak{A} \models G$;
7 **if** $\neg\varphi^{\mathfrak{A}}$ *is unsatisfiable* **then**
8 **return** `sat` *with model* $\mathfrak{A}$;
9 **end**
10 $\mathfrak{A}' \leftarrow$ model satisfying $\neg\varphi^{\mathfrak{A}}$;
11 $\bar{t} \leftarrow$ terms with value of $\bar{x} \colon \bar{\sigma}$ in $\mathfrak{A}'$;
12 $G \leftarrow G \cup \{ \varphi[\bar{x} \colon \bar{\sigma} \mapsto \bar{t}] \}$;
13 **end**

The algorithm loops and tries again with the new set of ground terms.

Example 2.23. MBQI

Considering again the Σ_{ex}-sentence that was used as a running example throughout Section 2.2:

$$\xi \quad \coloneqq \quad square(a) - square(b) \approx 7 \wedge \forall x \colon \mathbf{Int}.\ \underbrace{square(x) \approx x * x}_{F}$$

MBQI might proceed as follows to construct a model for ξ modulo NIA and EUF.

- The initial set of ground terms is $G_1 = \{ square(a) - square(b) \approx 7 \}$ (line 1).
- *Loop iteration 1:*
  - G_1 is not unsatisfiable (line 3)
  - A possible model for the ground terms is $\mathfrak{A}_1$ with

    $$(a \colon \mathbf{Int})^{\mathfrak{A}_1} = 7 \qquad (b \colon \mathbf{Int})^{\mathfrak{A}_1} = 0 \qquad (square \colon \mathbf{Int}\,\mathbf{Int})^{\mathfrak{A}_1}(z) = z$$

    and the remaining functions and sorts being interpreted as required by the theories (line 6).
  - Using this model to partially evaluate the quantifier body yields the term $F^{\mathfrak{A}_1} = (\lambda z.\ z)(x) \approx x *_{\mathbb{Z}} x$. The negation $\neg F^{\mathfrak{A}_1} = x \not\approx x *_{\mathbb{Z}} x$ is satisfiable with $x = 2$ (lines 7-11).
  - Therefore, the term $F[x \colon \mathbf{Int} \mapsto 2]$ is added to the ground terms resulting in the new set $G_2 = \{ square(a) - square(b) \approx 7, square(2) \approx 4 \}$ (line 12).
- *Loop iteration 2:*
  - G_2 is also not unsatisfiable (line 3)
  - Assume the core satisfiability procedure returns the following model $\mathfrak{A}_2$ of G_2 (line 6):

    $$(a \colon \mathbf{Int})^{\mathfrak{A}_2} = 4 \qquad (b \colon \mathbf{Int})^{\mathfrak{A}_2} = 3 \qquad (square \colon \mathbf{Int}\,\mathbf{Int})^{\mathfrak{A}_2}(z) = z *_{\mathbb{Z}} z.$$

  - Partially evaluating and negating the quantifier body results in

    $$\begin{aligned} \neg F^{\mathfrak{A}_2} &= \neg((\lambda z.\ z *_{\mathbb{Z}} z)(x) \approx x *_{\mathbb{Z}} x) \\ &= x *_{\mathbb{Z}} x \not\approx x *_{\mathbb{Z}} x \end{aligned}$$

    which is unsatisfiable (line 7).
  - Hence, $\mathfrak{A}_2 = \mathfrak{A}_{ex}$ is a model for ξ and the algorithm returns `sat` (line 4).

MBQI is not guaranteed to terminate. If the quantifiers range over infinite domains, then MBQI can never list all counterexamples and add them to the ground terms. Thus, the model for the ground terms can always violate the quantifiers. The set of ground terms may also never become unsatisfiable, as is the case if the input sentence if satisfiable.

For Caesar, MBQI is relevant because it theoretically provides the ability to construct models in the presence of quantifiers. These models are required for displaying verification counterexamples to the user. Counterexamples are discussed in Chapter 4. We also observed one case where MBQI was required to verify a program, i.e. proof `unsat` (Section 5.2).

3. Limited Functions

Frequently, user-defined functions are used in procedure specifications. The definitions of user-defined functions always require quantifiers. Many relevant functions, such as for harmonic numbers, general sums, and exponential functions (see Figure 1.1), are recursive. Here we face a problem. By the nature of recursive functions, the recursive function symbol appears on both sides of the equality sign. Therefore, the default encoding of recursive functions introduces a matching loop. We already saw that in Example 2.21 with the exponential function. We want to give another example with the factorial function, that is slightly simpler since it has only a single argument. The function is defined in Figure 3.1.

```
1  func fac(n: UInt): UInt
2  axiom fac_def forall n: UInt @trigger(fac(n)).
3    fac(n) == ite(n == 0, 1, n * fac(n - 1))
```

Figure 3.1: HeyVL code defining the factorial function $n!$.

Again, the trigger allows the solver to instantiate the axioms if it sees a term of the form `fac(n')`. Doing so, it learns that this term is equal to `ite(n' == 0, 1, n' * fac(n' - 1))`. Effectively unfolding the definition of `fac` once. We again have a matching loop since the instantiation produced the new term `fac(n' - 1)` which the solver can unfold once more.

These deep instantiations are necessary if we want to correctly and completely model the recursive function. In the context of verification, very few recursive unfoldings (often just one) are usually sufficient to verify a program. This is due to the structure of the proof rules for loops and procedure calls, which resemble a proof by induction. In this case, a single unfolding is usually sufficient to allow the application of the induction hypothesis.

This sparks the idea to artificially limit the number of recursive unfoldings, introducing incompleteness on purpose to guide the solver during the proof search. We want to further motivate the idea that this is not a big restriction in practice and better than the current incompleteness caused by non-termination.

Example 3.2. Inductive proof rules

Consider the following `factorial` procedure.

```
1  proc factorial(n: UInt) -> (res: UInt) post ?(fac(n) == res) {
2    res = 1
3    var i: UInt = 0
4
5    @invariant(?(res == fac(i)))
6    while i != n {
7      i = i + 1
8      res = res * i
9    }
10 }
```

It uses a `while` loop to compute the n-th factorial. The loop is annotated with an invariant. The invariant is used by Caesar when computing the verification pre-expectation to approximate the loop semantics. Looking at the while rule in Hoare logic

$$\frac{\{\!\{ I \wedge \mathsf{b} \}\!\}\ \mathsf{C}\ \{\!\{ I \}\!\}}{\{\!\{ I \}\!\}\ \texttt{while b \{ C \}}\ \{\!\{ I \wedge \neg \mathsf{b} \}\!\}}$$

we can see that the invariant I must hold initially and must be preserved by the loop body. This idea is also

present in the (Boolified) verification condition (VC) generated by Caesar:

$$
\begin{array}{lll}
1 \approx \texttt{fac}(0) \wedge & & \} \text{ induction base} \\
(res \approx \texttt{fac}(i) \implies & \text{(assume invariant - I.H.)} & \left.\begin{array}{l}\\\\\\\end{array}\right\} \text{ induction step} \\
\quad ite(i \not\approx n, & \text{(switch on loop condition)} & \\
\quad\quad res * (i+1) \approx \texttt{fac}(i+1), & \text{(invariant must be preseverd)} & \\
\quad\quad res \approx \texttt{fac}(n))) & \text{(post-condition must hold)} &
\end{array}
$$

We quickly examine how many unfoldings of `fac` are required to prove the VC. The induction base follows immediately from unfolding $\texttt{fac}(0)$ once and the base case of `fac`:

$$
\begin{array}{lll}
 & 1 \approx \texttt{fac}(0) & \\
\text{iff} & 1 \approx ite(0 \approx 0, 1, \texttt{fac}(0-1)) & \text{(unfold } \texttt{fac}(0)\text{)} \\
\text{iff} & 1 \approx 1 & \text{(simplify)}
\end{array}
$$

For showing that the invariant is preserved in the induction step, we can assume the invariant holds for i, i.e. $res \approx \texttt{fac}(i)$:

$$
\begin{array}{lll}
 & res * (i+1) \approx \texttt{fac}(i+1) & \\
\text{iff} & res * (i+1) \approx ite(i+1 \approx 0, 1, (i+1) * \texttt{fac}(i)) & \text{(unfold } \texttt{fac}(i+1)\text{)} \\
\text{iff} & res * (i+1) \approx (i+1) * \texttt{fac}(i) & (i+1 \text{ cannot be } 0) \\
\text{iff} & \texttt{fac}(i) * (i+1) \approx (i+1) * \texttt{fac}(i) & (res \approx \texttt{fac}(i))
\end{array}
$$

When the loop condition is false ($i \approx n$) the post condition must hold. We can additionally use that the loop invariant also holds after the loop execution. The post-condition $res \approx \texttt{fac}(n)$ follows directly from $res \approx \texttt{fac}(i)$ and $i \approx n$.

Proving the VC required two unfoldings, but no recursive unfoldings. Something similar can be observed for procedure calls.

The SMT solver, being a general purpose solver, cannot rule out that deep unfoldings are not required and must explore them. But we know how the usual structure of a program correctness proof looks, and can limit the number of possible recursive instantiations by adjusting the encoding of the functions.

The technique of artificially limiting the number of instantiations was used by Leino and Monahan to axiomatize comprehensions[1] in Spec# [30]. Later, Amin et al. developed an encoding for user-defined functions based on the above idea [2] and implemented it in Dafny [28]. We call functions encoded in such a way *limited functions*. Other proof oriented programming languages like F* also adopted limited functions [1]. The verification-condition-generation-based verifier for Viper [35] (Carbon) also uses limited functions.[2] Limited functions are also very relevant for Caesar, due to the aforementioned matching loop problem. There is nothing inherently probabilistic or quantitative about functions in Caesar. This justifies using the same encoding as classical deductive verifiers.

We formally define the encodings in the next section before giving a soundness proof in Section 3.2.

3.1 The Encodings

Limited functions are (recursive) functions encoded in such a way that the number of possible (recursive) instantiations (with E-matching) are limited (usually to once or twice). This prevents matching loops by construction. We will examine two different encodings that systematically create limited functions. Both encodings should be sound:

1. Wrong programs should not verify
2. and no spurious counterexamples should be produced.

[1] Comprehensions are a family of expressions that are reduced using an operator. Examples are $\texttt{sum}\{ a[i] \mid 0 \le i < a.\text{length} \}$ and $\texttt{count}\{ a[i] \mod 2 = 0 \mid 0 \le i < a.\text{length} \}$.

[2] https://github.com/viperproject/carbon/blob/e6d2393f85b5d8b53639b0f22e59c84004379ee7/src/main/scala/viper/carbon/modules/impls/DefaultFuncPredModule.scala

Additionally, they should ensure termination. These properties are proven in the next section. Regarding completeness, we have to expect to be able to prove strictly fewer programs. We argued above that the usually generated verification conditions are still provable, but we are still trading completeness for efficiency. In practice, this can actually mean that we can verify more programs since we do not run into timeouts. An evaluation is done in Chapter 6.

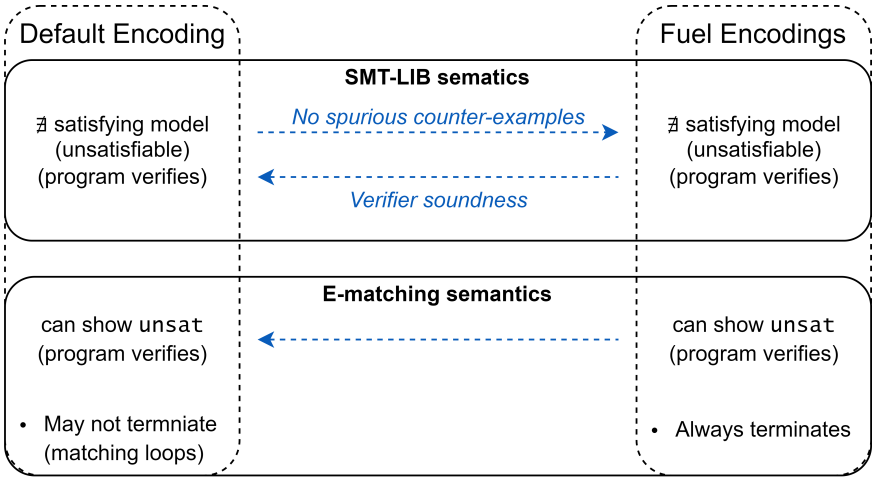


Figure 3.3: Overview of the relation between the default and the fuel encodings. They are equivalent under SMT-LIB semantics. Under E-matching semantics, the fuel encodings are less complete than the Default encoding but they do not introduce matching loops.

Figure 3.3 summarizes these points. On the left is the Default encoding (the canoncial reference encoding) and on the right are the fuel encodings which create the limited functions. Under the previously described FO semantics defined by SMT-LIB (cf. Section 2.2), the encodings are equisatisfiable. Either they are all satisfiable or all unsatisfiable. If the fuel encodings are unsatisfiable, this means that the Default encoding is also unsatisfiable. This guarantees us that the verifier remains sound. The reverse direction (after negating both sides) ensures that if the encoding with the limited functions is satisfiable, the Default encoding is too. This ensures that we do not produce any spurious counterexamples. When considering a more operational semantics in the form of E-matching, then the fuel encodings are less complete. A proof being constructible for the fuel encodings only implies that the same can be done for the Default encoding. The reverse is not true. Instead, we get the guarantee that the user-defined functions, encoded as limited functions, did not introduce any matching loops.

The functions that we are encoding are defined by a body that is a single expression. Figure 3.4 demonstrates the associated HeyVL syntax by defining the previously seen factorial function. This function definition is used as an example throughout the next section.

```
1  func fac(n: UInt): UInt = ite(n == 0, 1, n * fac(n - 1))
```

Figure 3.4: Defining the `fac` function directly with a body.

The encodings will introduce both new function/sort symbols and alter the existing formulae. Therefore, each encoding consists of two functions. The first function $\mathcal{E}_{\text{sig}}(\Sigma, g\colon \bar{D}\,R)$ performs alterations to the signature Σ that are necessary for encoding the function $g\colon \bar{D}\,R \in \Sigma$. The second function $\mathcal{E}_{\text{term}}(\varphi, \text{def}_g)$ transforms the Σ-term φ (usually referencing g) into a $\mathcal{E}_{\text{sig}}(\Sigma, g\colon \bar{D}\,R)$-term, also fixing the meaning of $g\colon \bar{D}\,R$ by incorporating the defining body def_g. Here, def_g is a Σ-term of sort R with only the free variables $\bar{d}\colon \bar{D}$.

For the function definition in Figure 3.4 the function is `fac`$\colon$ **UInt UInt** and the defining body $\text{def}_{\texttt{fac}}$ is the right side of the definition $ite(n \approx 0, 1, n * \texttt{fac}(n-1))$.

3.1.1 Default Encoding

Before coming to the limited function encodings, we need a base case that we can compare them to. The Default encoding directly corresponds to the definition of the `define-fun` and `define-fun-rec` commands from the SMT-LIB standard [6]. We will use it as our canonical reference encoding.

The Default encoding adds a single definitional axiom that states that for all arguments, the function is equal to its definition. For our `fac` example, the result can be seen in Figure 3.1.

Definition 3.5. Default encoding

The default encoding is given by the encoding functions:

$$\mathcal{E}_{\text{sig}}(\Sigma, g\colon \bar{D}\, R) = \Sigma$$
$$\mathcal{E}_{\text{term}}(\varphi, \text{def}_g) = \delta_g \wedge \varphi$$

where the definitional axiom δ_g is

$$\delta_g \quad \coloneqq \quad \forall \bar{d}\colon \bar{D}\ \{g(\bar{d})\}.\ g(\bar{d}) \approx \text{def}_g \qquad \text{(def-axiom)}$$

3.1.2 Fixed Fuel Encoding

As previously stated, the idea is to limit the number of times the universal quantifier of the defining axiom is instantiated with E-matching. There are several ways of encoding this idea, but they all introduce the notion of *fuel*. A fuel value, representing a natural number $n \in \mathbb{N}_0$, is associated to every function application. Each instantiation of the defining axiom burns one fuel (n is decremented). If the fuel reaches 0, no further instantiations are possible. One parameter common to all fuel encodings is the number $mf \in \mathbb{N}_{>0}$ of allowed instantiations. This parameter is usually set to 1 or 2.

In the Fixed fuel encoding $mf + 1$ function symbols are introduced, one for each possible fuel value. For our example with $mf = 1$, this would mean the two functions `fac0` and `fac1` (compare Figure 3.6). `fac1` should be unfoldable once and `fac0` should not be unfoldable. This is achieved by providing a definitional axiom for `fac1` that uses `fac0` in the body and providing no axiom for `fac0` (lines 4-5). Note that the trigger enables us to read the axiom from left to right. There is no matching loop this time, since unfolding `fac1` only introduces a new `fac0` term, which cannot be unfolded further. We split the original function into two functions, but these are not independent of each other. They are different versions of the same function. This fact is captured by the synonym axiom that states that `fac1` and `fac0` always have the same result. Again, the trigger only allows the fuel to be reduced. The existence of `fac0`-term cannot be used to instantiate the axiom and create a `fac1`-term.

```
1  func fac1(n: UInt): UInt
2  func fac0(n: UInt): UInt
3  axiom fac_syn1 forall n: UInt @trigger(fac1(n)). fac1(n) == fac0(n)
4  axiom fac_def1 forall n: UInt @trigger(fac1(n)).
5    fac1(n) == ite(n == 0, 1, n * fac0(n - 1))
```

Figure 3.6: `fac` from Figure 3.4 encoded as a limited function with fixed fuel and $mf = 1$.

Definition 3.7. Fixed fuel encoding

The fixed fuel encoding is defined by the encoding functions:

$$\mathcal{E}^{ff}_{\text{sig}}(\Sigma, g\colon \bar{D}\, R) = \Big\langle \overbrace{\Sigma^{\text{S}}}^{\Sigma^{\text{S}}_{ff}},\ \overbrace{(\Sigma^{\text{F}} \setminus \{g\}) \dot\cup \{g_0, \ldots, g_{mf}\}}^{\Sigma^{\text{F}}_{ff}},\ \overbrace{(\Sigma^{\text{FS}} \setminus \{g\colon \bar{D}\, R\}) \dot\cup \{g_0, \ldots, g_{mf}\colon \bar{D}\, R\}}^{\Sigma^{\text{FS}}_{ff}} \Big\rangle$$
$$\mathcal{E}^{ff}_{\text{term}}(\varphi, \text{def}_g) = \delta^{ff}_{g_1} \wedge \cdots \wedge \delta^{ff}_{g_{mf}} \wedge \sigma^{ff}_{g_1} \wedge \cdots \wedge \sigma^{ff}_{g_{mf}} \wedge \varphi[g(\bar{d'}\colon \bar{D}) \mapsto g_{mf}(\bar{d'})]$$

with the axioms

$$\delta^{ff}_{g_i} \quad \coloneqq \quad \forall \bar{d}\colon \bar{D}\ \{g_i(\bar{d})\}.\ g_i(\bar{d}) \approx \text{def}_g[g(\bar{d'}\colon \bar{D}) \mapsto g_{i-1}(\bar{d'})] \qquad \text{for } i = 1, \ldots, mf \qquad \text{(ff-def-axiom)}$$
$$\sigma^{ff}_{g_i} \quad \coloneqq \quad \forall \bar{d}\colon \bar{D}\ \{g_i(\bar{d})\}.\ g_i(\bar{d}) \approx g_{i-1}(\bar{d}) \qquad \text{for } i = 1, \ldots, mf \qquad \text{(ff-syn-axiom)}$$

The original function $g\colon \bar{D}\, R$ is replaced with $mf + 1$ fuelled versions. A ff-def-axiom and ff-syn-axiom is added for each g_i except g_0. The Σ-term def_g is transformed into a $\mathcal{E}^{ff}_{\text{sig}}(\Sigma, g\colon \bar{D}\, R)$-term by replacing recursive calls to g with calls to the function that has one less fuel. In the remaining formula φ, the function g_{mf} is used instead of the

original function g. The Fixed fuel encoding makes matching loops impossible, as instantiating the quantifiers only introduces function symbols with a smaller fuel value, creating a chain $g_{mf} > \cdots > g_1 > g_0$. The SMT solver is still able to learn something about the structure of the recursive function (to the depth mf), while the rest is hidden behind the uninterpreted and unconstrained symbol g_0.

3.1.3 Variable Fuel Encoding

The Variable fuel encoding was originally developed in [2] for use in Dafny [28]. Rather than introducing a new function symbol and axioms for each fuel value, only a single function is introduced, with the fuel value added as an additional parameter of type `Fuel`. This drastically reduces the number of declarations and axioms required for higher maximal fuel values. The extra fuel parameter keeps track of the number of remaining instantiations. Since the `fuel` parameter must be available during E-matching, it must be entirely syntactic and cannot contain interpreted symbols such as `0` or `1`. Therefore, values of sort `Fuel` are represented as Peano numbers by using two auxiliary ranked function symbols $Z\colon$ `Fuel` (representing zero) and $S\colon$ `Fuel Fuel` (representing the successor of another fuel value).[3]

```
1  domain Fuel {
2    func Z(): Fuel
3    func S(prev: Fuel): Fuel
4  }
5  domain Factorial {
6    func fac(fuel: Fuel, n: UInt): UInt
7    axiom fac_syn forall fuel: Fuel, n: UInt @trigger(fac(S(fuel), n)).
8      fac(S(fuel), n) == fac(fuel, n)
9    axiom fac_def forall fuel: Fuel, n: UInt @trigger(fac(S(fuel), n)).
10     fac(S(fuel), n) == ite(n == 0, 1, n * fac(fuel, n - 1))
11 }
```

Figure 3.8: Dynamic fuel encoding for `fac` from Figure 3.4. The enclosing domain declarations are also shown, as they are required to define the `Fuel` sort.

Applying this to our running example (Figure 3.8) means that we need to declare the `Fuel` sort (line 1-4) and add another parameter to the declaration of `fac`. A synonym axiom states again that the result of `fac` does not depend on the extra `fuel` parameter, i.e. it is the same for all fuel parameter values. Since the triggers for the definitional and synonym axiom require a non-zero fuel value (`S(fuel)`) the fuel value can once again only be reduced. In the remaining formula φ, `fac` would be used with a fixed fuel value of mf (encoded as a Peano number).

Definition 3.9. Variable fuel encoding

The variable fuel encoding is defined by the encoding functions:

$$\mathcal{E}^{vf}_{\text{sig}}(\Sigma, g\colon \bar{D}\,R) = \Big\langle \overbrace{\Sigma^{\mathbf{S}} \dot\cup \{\,\texttt{Fuel}\,\}}^{\Sigma^{\mathbf{S}}_{vf}},\quad \overbrace{(\Sigma^{\mathbf{F}} \setminus \{\,g\,\}) \dot\cup \{\,g_{\texttt{fuel}}, Z, S\,\}}^{\Sigma^{\mathbf{F}}_{vf}},\quad \underbrace{(\Sigma^{\mathbf{FS}} \setminus \{\,g\colon \bar{D}\,R\,\}) \dot\cup \{\,Z\colon \texttt{Fuel}, S\colon \texttt{Fuel Fuel}, g_{\texttt{fuel}}\colon \texttt{Fuel}\,\bar{D}\,R\,\}}_{\Sigma^{\mathbf{FS}}_{vf}} \Big\rangle$$

$$\mathcal{E}^{vf}_{\text{term}}(\varphi, \text{def}_g) = \delta^{vf}_g \wedge \sigma^{vf}_g \wedge \varphi[g(\bar{d}'\colon \bar{D}) \mapsto g_{\texttt{fuel}}(mf, \bar{d}')]$$

with the axioms

$$\delta^{vf}_g \;\coloneqq\; \forall \texttt{fuel}\colon \texttt{Fuel}, \bar{d}\colon \bar{D}\ \{g_{\texttt{fuel}}(S(\texttt{fuel}), \bar{d})\}.\ g_{\texttt{fuel}}(S(\texttt{fuel}), \bar{d}) \approx \text{def}_g[g(\bar{d}'\colon \bar{D}) \mapsto g_{\texttt{fuel}}(\texttt{fuel}, \bar{d}')] \qquad \text{(vf-def-axiom)}$$

$$\sigma^{vf}_g \;\coloneqq\; \forall \texttt{fuel}\colon \texttt{Fuel}, \bar{d}\colon \bar{D}\ \{g_{\texttt{fuel}}(S(\texttt{fuel}), \bar{d})\}.\ g_{\texttt{fuel}}(S(\texttt{fuel}), \bar{d}) \approx g_{\texttt{fuel}}(\texttt{fuel}, \bar{d}) \qquad \text{(vf-syn-axiom)}$$

[3] For brevity, we assume that values of sort `Fuel` and $\mathbb{N}_0$ are automatically converted into each other, as required by the context.

3.1.4 First Comparison and Analysis

The two fuel encodings are quite similar. The extra `fuel` parameter is a compact way of introducing multiple function symbols plus axioms, with $g_i(\bar{d})$ being equivalent to $g_{\texttt{fuel}}(i, \bar{d})$. The synonym axioms are vital for verification. Intuitively, their effect is to bundle the different functions so that they all describe the same single function.

Example 3.10. Importance of the synonym axiom

Consider the following program. It is the Fixed fuel encoding of the factorial function from Figure 3.4 but without the (ff-syn-axiom).

```
1  domain Fac {
2    func fac1(n: UInt): UInt
3    func fac0(n: UInt): UInt
4
5    axiom fac_def forall n: UInt @trigger(fac1(n)).
6      fac1(n) == ite(n == 0, 1, n * fac0(n - 1))
7  }
8
9  proc factorial(n: UInt) -> (res: UInt) post ?(res == fac1(n)) {
10   if n == 0 {
11     res = 1
12   } else {
13     var temp: UInt = factorial(n - 1)
14     res = temp * n
15   }
16 }
```

The `factorial` procedure should verify. It is a direct implementation of the factorial function as defined by Figure 3.4. But the presented program produces a counterexample:

$$\texttt{fac1}(x) = \begin{cases} 0, & \text{if } x = 2 \quad (2 \cdot \texttt{fac0}(1) = 2 \cdot 0) \\ 2, & \text{if } x = 1 \quad (1 \cdot \texttt{fac0}(0) = 1 \cdot 2) \\ 1, & \text{if } x = 0 \quad (\textit{base-case}) \\ x \cdot \texttt{fac0}(x-1), & \text{else} \end{cases}$$

$$\texttt{fac0}(x) = \begin{cases} 2, & \text{if } x = 0 \\ 0, & \text{else} \end{cases}$$

$$\texttt{n} = 2$$
$$\texttt{temp} = 2$$
$$\texttt{res} = 4$$

The counterexample is spurious, since the produced model for `fac1` does not actually represent the factorial function except for the base case. But it is correct in the sense that all axioms are fulfilled and that the `factorial` procedure does not implement the chosen version of `fac1`. Adding the synonym axiom makes the counterexample invalid (e.g. $\texttt{fac1}(0) = 1 \neq 2 = \texttt{fac0}(0)$) and is enough to successfully verify the example.

Another closely related approach was used in [30] to encode comprehensions. Comprehensions are a family of expressions that are reduced using an operator. Examples are $\texttt{sum}\{\, a[i] \mid 0 \leq i < a.\text{length} \,\}$ and $\texttt{count}\{\, a[i] \mod 2 = 0 \mid 0 \leq i < a.\text{length} \,\}$. They used a synonym function (a new function that is equivalent to the first one), that is not mentioned by the body of any axiom, in the triggers to control the instantiations. The result of applying this technique to the `fac` function is shown in Figure 3.11. As usual, `fac1` would be used in the remaining program.

The `fac_def` axiom is formulated only in terms of `fac0` but the trigger only mentions `fac1`. The matching loop is broken because the body only generates new `fac0`-terms that do not match the trigger. It relies on the pattern in the trigger to prevent the solver from instantiating the axiom again. The SMT solver can learn something about `fac1` by using its existence to insatiate the definitional axiom, learning something about `fac0`, and then concluding with the synonym axiom that the same must hold for `fac1`.

```
1  func fac1(n: UInt): UInt
2  func fac0(n: UInt): UInt
3  axiom fac_syn forall n: UInt @trigger(fac1(n)). fac1(n) == fac0(n)
4  axiom fac_def forall n: UInt @trigger(fac1(n)).
5    fac0(n) == ite(n == 0, 1, n * fac0(n - 1))
```

Figure 3.11: Encoding `fac` as a limited function using the synonym pattern encoding.

This synonym pattern encoding is a variant of the Fixed fuel encoding with $mf = 1$. The (ff-def-axiom) is changed to (also using g_0 on the left-hand side):

$$\forall \bar{d} : \bar{D}\ \{g_1(\bar{d})\}.\ g_0(\bar{d}) = \text{def}_g[g_0(d' \dot{:} \bar{D}) \mapsto g(\bar{d'})] \qquad \text{(synp-ff-def-axiom)}$$

This encoding can be more flexible. When encoding other constructs that require multiple axioms (such as comprehensions), it is possible to specify on a per-pattern basis whether the instantiation should be limited (by using g_1) or unlimited (by using g_0).

The same idea can also be applied to the Variable fuel encoding, by only requiring the non-zero `fuel` value in the trigger:

$$\forall \texttt{fuel} : \texttt{Fuel}, \bar{d} : \bar{D}\ \{g_{fuel}(S(\texttt{fuel}), \bar{d})\}.\ g_{\texttt{fuel}}(\texttt{fuel}, \bar{d}) = \text{def}_g[g(d' \dot{:} \bar{D}) \mapsto g_{\texttt{fuel}}(\texttt{fuel}, \bar{d'})] \qquad \text{(synp-vf-def-axiom)}$$

3.2 Soundness

We now formally motivate the previously presented encodings. Most importantly, we show that Caesar does not become unsound by implementing any of the fuel encodings.

The statements made in Figure 3.3 are proven in this section. In Section 3.2.1 we prove that all the presented encodings are equisatisfiable under the SMT-LIB semantics introduced in Section 2.2 which ignores triggers. This means if there does not exist a satisfying model for the Default encoding then there also exist no satisfying model for the fuel encodings and vice versa. The unsatisfiability of a fuel encoding implies the unsatisfiability of the Default encoding, which ensures the verifier's soundness, i.e. that no wrong programmes are verified. The opposite direction ensures that we do not get spurious counter examples. More details can be found in Section 3.2.2.

Since the fuel encodings were designed specifically with E-matching in mind, we also examine how the encodings behave in practice when E-matching used for quantifier reasoning, i.e. under E-matching semantics, in Section 3.2.4 and Section 3.2.3. If E-matching can be used to show that a fuel encoding is unsatisfiable, the same can be done for the Default encoding. The reverse is not necessarily the case due to the more restrictive triggers. Therefore, when using a fuel encoding, it is theoretically expected to get `unknown` more often. On the other hand, the fuel encodings are guaranteed to terminate (or at least not to introduce new matching loops), which is again relevant in practice where E-matching with the Default encoding can run into matching loops and timeout instead of proving `unsat`.

3.2.1 Equisatisfiability under SMT-LIB Semantics

First, we show that both the Fixed fuel encoding and Variable fuel encoding are equisatisfiable to the Default encoding. In this section, we focus on the SMT-LIB semantics. Under the standard first-order (FO) semantics, the triggers are ignored. While they provide additional information for the SMT solver's heuristics, which is very relevant in practice, they play no role when determining if a model theoretical exists. Due to the role heuristics play, the fuel encodings are generally not equivalent to the Default encoding in practice. We discuss the practical implications afterwards in Section 3.2.2 and following.

Theorem 3.12. Equisatisfiability of encodings

Let Σ by a signature, $g\colon \bar{D}\,R \in \Sigma$ a ranked function symbol from Σ, def_g a definitional body for $g\colon \bar{D}\,R$, $mf \in \mathbb{N}_{>0}$, and φ a Σ-sentence. The Default encoding, Fixed fuel encoding, and Variable fuel encoding (with above parameters) are logically equivalent. Formally:

$$\begin{aligned}&\text{there exists a } \mathcal{E}_{\mathrm{sig}}(\Sigma, g\colon \bar{D}\,R)\text{-structure } \mathfrak{A} \text{ such that } \mathfrak{A} \models \mathcal{E}_{\mathrm{term}}(\varphi, \mathrm{def}_g)\\ \text{iff }&\text{there exists a } \mathcal{E}^{ff}_{\mathrm{sig}}(\Sigma, g\colon \bar{D}\,R)\text{-structure } \mathfrak{A}_{ff} \text{ such that } \mathfrak{A}_{ff} \models \mathcal{E}^{ff}_{\mathrm{term}}(\varphi, \mathrm{def}_g)\\ \text{iff }&\text{there exists a } \mathcal{E}^{vf}_{\mathrm{sig}}(\Sigma, g\colon \bar{D}\,R)\text{-structure } \mathfrak{A}_{vf} \text{ such that } \mathfrak{A}_{vf} \models \mathcal{E}^{vf}_{\mathrm{term}}(\varphi, \mathrm{def}_g).\end{aligned}$$

We prove the equivalence by showing three implications. From the Default encoding to the Fixed fuel encoding, from the Fixed fuel encoding to the Variable fuel encoding, and back from the Variable fuel encoding to the Default encoding. In each step, the existence of the model is shown by transforming the given model into a model for the other encoding. To improve readability, we define the following abbreviations:

$\Sigma = \mathcal{E}_{\mathrm{sig}}(\Sigma, g\colon \bar{D}\,R)$	(default signature)	$\psi \coloneqq \mathcal{E}_{\mathrm{term}}(\varphi, \mathrm{def}_g)$	(default term)
$\Sigma_{ff} \coloneqq \mathcal{E}^{ff}_{\mathrm{sig}}(\Sigma, g\colon \bar{D}\,R)$	(fixed fuel signature)	$\psi^{ff} \coloneqq \mathcal{E}^{ff}_{\mathrm{term}}(\varphi, \mathrm{def}_g)$	(fixed fuel term)
$\Sigma_{vf} \coloneqq \mathcal{E}^{vf}_{\mathrm{sig}}(\Sigma, g\colon \bar{D}\,R)$	(variable fuel signature)	$\psi^{vf} \coloneqq \mathcal{E}^{vf}_{\mathrm{term}}(\varphi, \mathrm{def}_g)$	(variable fuel term)

The convention is that constructs associated with the Default encoding have no additional subscript/superscript. The equivalent constructs for the Fixed fuel encoding have a ff subscript/superscript and the equivalent constructs for the Variable fuel encoding have a vf subscript/superscript.

Default encoding to Fixed fuel encoding

Assuming there exists a Σ-structure $\mathfrak{A}$ with $\mathfrak{A} \models \psi$, we show there exists a Σ_{ff}-model of ψ^{ff} by explicitly constructing it from the Σ-model.

Definition 3.13. Fixed fuel structure

Let $\mathfrak{A} = \langle \mathbf{A}, \cdot^{\mathfrak{A}} \rangle$ be a Σ-structure such that $\mathfrak{A} \models \psi$. The corresponding Σ_{ff}-structure $\mathfrak{A}_{ff} = \langle \mathbf{A}, \cdot^{\mathfrak{A}_{ff}} \rangle$ is defined by (notice that $\Sigma^{\mathrm{S}} = \Sigma^{\mathrm{S}}_{ff}$):

$$\sigma^{\mathfrak{A}_{ff}} = \sigma^{\mathfrak{A}} \qquad \text{for } \sigma \in \Sigma^{\mathrm{S}}_{ff}$$

$$(f\colon \bar{\sigma}\,\rho)^{\mathfrak{A}_{ff}} = \begin{cases} (g\colon \bar{D}\,\bar{R})^{\mathfrak{A}}, & \text{if } f\colon \bar{\sigma}\,\rho = g_i\colon \bar{D}\,R \text{ for some } i = 1, \ldots, mf \\ (f\colon \bar{\sigma}\,\rho)^{\mathfrak{A}}, & \text{otherwise} \end{cases} \qquad \text{for } f\colon \bar{\sigma}\,\rho \in \Sigma_{ff}$$

It remains to show that $\mathfrak{A}_{ff}$ is a model of ψ^{ff}. The central observation is that the two structures assign the same value to all the different g-functions $(g, g_0, \ldots, g_{mf})$, so that we can replace applications of g with applications of g_i without changing the value of a term. This fact is shown in the following lemma. Since we consider arbitrary terms, it does not suffice to discuss structures, but we must consider interpretations. The structures are extended to interpretations by attaching identical variable assignments to them.

Lemma 3.14. Preservation of value (fixed fuel)

Given two interpretations $\mathfrak{I} = \langle \mathfrak{A}, \mathfrak{v} \rangle, \mathfrak{I}_{ff} = \langle \mathfrak{A}_{ff}, \mathfrak{v} \rangle$, and $i = 0, \ldots, mf$, then for every Σ-term t, and Σ_{ff}-term $t^{ff} \coloneqq t[g(\bar{d}'\colon \bar{D}) \mapsto g_i(\bar{d}')]$, we have

$$[\![t]\!]^{\mathfrak{I}} = [\![t^{ff}]\!]^{\mathfrak{I}_{ff}}.$$

In particular, for a Σ-sentence ϕ it holds that, $\mathfrak{A} \models \phi$ if and only if $\mathfrak{A}_{ff} \models \phi[g(\bar{d}'\colon \bar{D}) \mapsto g_i(\bar{d}')]$.

Proof. By structural induction on t.

Case $t = x = t^{ff}$. The claim directly follows from $[\![x]\!]^{\mathfrak{I}} = \mathfrak{v}(x) = [\![x]\!]^{\mathfrak{I}_{ff}}$.

Case $t = \forall \bar{x} \colon \bar{\sigma}.\ t'$ *and* $t^{ff} = \forall \bar{x} \colon \bar{\sigma}.\ t'[g(\bar{d}' \colon \bar{D}) \mapsto g_i(\bar{d}')]$.

$$
\begin{aligned}
[\![t]\!]^{\mathfrak{I}} &= [\![\forall \bar{x} \colon \bar{\sigma}.\ t']\!]^{\mathfrak{I}} = \mathbf{true} \\
&\quad \text{iff } [\![t']\!]^{\mathfrak{I}[\bar{x} \colon \bar{\sigma} \mapsto \bar{a}]} = \mathbf{true} \text{ for all } \bar{a} \in \bar{\sigma}^{\mathfrak{A}} && \text{(by def. of } [\![\cdot]\!]^{\mathfrak{I}}) \\
&\quad \text{iff } [\![t'[g(\bar{d}' \colon \bar{D}) \mapsto g_i(\bar{d}')]]\!]^{\mathfrak{I}_{ff}[\bar{x} \colon \bar{\sigma} \mapsto \bar{a}]} = \mathbf{true} \text{ for all } \bar{a} \in \bar{\sigma}^{\mathfrak{A}} && \text{(induction hypothesis)} \\
&\quad \text{iff } [\![t^{ff}]\!]^{\mathfrak{I}_{ff}} = \mathbf{true} && \text{(by def. of } [\![\cdot]\!]^{\mathfrak{I}_{ff}})
\end{aligned}
$$

Case $t = \exists \bar{x} \colon \bar{\sigma}.\ t'$ *and* $t^{ff} = \exists \bar{x} \colon \bar{\sigma}.\ t'[g(\bar{d}' \colon \bar{D}) \mapsto g_i(\bar{d}')]$. *analogous*

Case $t = g(\bar{t}')$ *with* $\bar{t}' \colon \bar{D}$ *and* $t^{ff} = g_i(\bar{t}'[g(\bar{d}' \colon \bar{D}) \mapsto g_i(\bar{d}')])$.

$$
\begin{aligned}
[\![t]\!]^{\mathfrak{I}} &= [\![g(\bar{t}')]\!]^{\mathfrak{I}} \\
&= (g \colon \bar{D}\, R)^{\mathfrak{A}}([\![\bar{t}']\!]^{\mathfrak{I}}) && \text{(by def. of } [\![\cdot]\!]^{\mathfrak{I}}) \\
&= (g \colon \bar{D}\, R)^{\mathfrak{A}}([\![\bar{t}'[g(\bar{d}' \colon \bar{D}) \mapsto g_i(\bar{d}')]]\!]^{\mathfrak{I}_{ff}}) && \text{(induction hypothesis)} \\
&= (g_i \colon \bar{D}\, R)^{\mathfrak{A}_{ff}}([\![\bar{t}'[g(\bar{d}' \colon \bar{D}) \mapsto g_i(\bar{d}')]]\!]^{\mathfrak{I}_{ff}}) && \text{(by def. of } \mathfrak{A}^{ff}) \\
&= [\![t^{ff}]\!]^{\mathfrak{I}_{ff}} && \text{(by def. of } [\![\cdot]\!]^{\mathfrak{I}_{ff}})
\end{aligned}
$$

Case $t = f(\bar{t}')$ *with* $\bar{t}' \colon \bar{\sigma}$, $f \neq g$ *or* $\bar{\sigma} \neq \bar{D}$ *and* $t^{ff} = f(\bar{t}'[g(\bar{d}' \colon \bar{D}) \mapsto g_i(\bar{d}')])$.

$$
\begin{aligned}
[\![t]\!]^{\mathfrak{I}} &= [\![f(\bar{t}')]\!]^{\mathfrak{I}} \\
&= (f \colon \bar{\sigma}\, \rho)^{\mathfrak{A}}([\![\bar{t}']\!]^{\mathfrak{I}}) && \text{(by def. of } [\![\cdot]\!]^{\mathfrak{I}}) \\
&= (f \colon \bar{\sigma}\, \rho)^{\mathfrak{A}}([\![\bar{t}'[g(\bar{d}' \colon \bar{D}) \mapsto g_i(\bar{d}')]]\!]^{\mathfrak{I}_{ff}}) && \text{(induction hypothesis)} \\
&= (f \colon \bar{\sigma}\, \rho)^{\mathfrak{A}_{ff}}([\![\bar{t}'[g(\bar{d}' \colon \bar{D}) \mapsto g_i(\bar{d}')]]\!]^{\mathfrak{I}_{ff}}) && \text{(by def. of } \mathfrak{A}^{ff}) \\
&= [\![t^{ff}]\!]^{\mathfrak{I}_{ff}} && \text{(by def. of } [\![\cdot]\!]^{\mathfrak{I}_{ff}})
\end{aligned}
$$

□

Building on the previous point, we can also interchange any g_i with any other g_j without affecting the value.

> Lemma 3.15. Fuel value is irrelevant (fixed fuel)
> For all $i, j = 0, \ldots, mf$, Σ_{ff}-term t, and $\mathfrak{I}_{ff} = \langle \mathfrak{A}_{ff}, \mathfrak{v} \rangle$
>
> $$[\![t]\!]^{\mathfrak{I}_{ff}} = [\![t[g_i(\bar{d} \colon \bar{D}) \mapsto g_j(\bar{d})]]\!]^{\mathfrak{I}_{ff}}$$

Proof. By structural induction on t, similar to the proof of Lemma 3.14. The statement follows directly from the definition of $\mathfrak{A}_{ff}$ (Definition 3.13) – *Omitted.* □

With these preliminaries done, we now show that we have successfully constructed a model for ψ^{ff}.

> Lemma 3.16. Fixed fuel model
> The constructed Σ_{ff}-structure $\mathfrak{A}_{ff}$ is a model of ψ^{ff}, i.e. $\mathfrak{A}_{ff} \models \psi^{ff}$.

Proof. We show that $\mathfrak{A}_{ff}$ satisfies ψ^{ff} by showing that $\mathfrak{A}_{ff}$ satisfies each conjunct of ψ^{ff}. Let $\mathfrak{I} = \langle \mathfrak{A}, \mathfrak{v} \rangle$ and $\mathfrak{I}_{vf} = \langle \mathfrak{A}_{vf}, \mathfrak{v} \rangle$ denote some interpretations with structures $\mathfrak{A}/\mathfrak{A}_{vf}$ and identical variable assignments $\mathfrak{v}$. Let $i = 1, \ldots, mf$.

$\mathfrak{A}_{ff} \models \delta^{ff}_{g_i} = \forall \bar{d}\colon \bar{D}.\ g_i(\bar{d}) \approx def_g[g(\bar{d'}\colon \bar{D}) \mapsto g_{i-1}(\bar{d'})]$. The definitional axioms are satisfied, since all g_j are interpreted like g.

$$
\begin{aligned}
&[\![\delta^{ff}_{g_i}]\!]^{\mathfrak{I}_{ff}} \\
&= [\![\delta^{ff}_{g_i}[g_{i-1}(\bar{d'}\colon \bar{D}) \mapsto g_i(\bar{d'})]]\!]^{\mathfrak{I}_{ff}} && \text{(by Lemma 3.15)} \\
&= [\![\delta_g[g(\bar{d'}\colon \bar{D}) \mapsto g_i(\bar{d'})]]\!]^{\mathfrak{I}_{ff}} && \text{(term equality)} \\
&= [\![\delta_g]\!]^{\mathfrak{I}} && \text{(by Lemma 3.14)} \\
&= \textbf{true} && (\mathfrak{A} \text{ is model of } \delta_g)
\end{aligned}
$$

$\mathfrak{A}_{ff} \models \sigma^{ff}_{g_i} = \forall \bar{d}\colon \bar{D}.\ g_i(\bar{d}) \approx g_{i-1}(\bar{d})$. The synonym axioms are satisfied, since $\mathfrak{A}_{ff}$ assigns all g_j the same value.

$$
\begin{aligned}
&[\![\sigma^{ff}_{g_i}]\!]^{\mathfrak{I}_{ff}} = \textbf{true} \\
&\text{iff } [\![\sigma^{ff}_{g_i}[g_{i-1}(\bar{d'}\colon \bar{D}) \mapsto g_i(\bar{d'})]]\!]^{\mathfrak{I}_{ff}} = \textbf{true} && \text{(by Lemma 3.15)} \\
&\text{iff } [\![g_i(\bar{d}) \approx g_i(\bar{d})]\!]^{\mathfrak{I}_{ff}[\bar{d}\colon \bar{D} \mapsto \bar{a}]} = \textbf{true} \text{ for all } \bar{a} \in \bar{D}^{\mathfrak{A}_{ff}} && \text{(by def. of } [\![\square]\!]^{\mathfrak{I}}) \\
&\text{iff } \textbf{true} = \textbf{true} && \text{(by def. of } \approx)
\end{aligned}
$$

$\mathfrak{A}_{ff} \models \varphi[g(\bar{d'}\colon \bar{D}) \mapsto g_{mf}(\bar{d'})]$. Follows from $\mathfrak{A} \models \varphi$ using Lemma 3.14. □

Fixed fuel encoding to Variable fuel encoding

Assuming there exists a Σ_{ff}-structure $\mathfrak{A}_{ff}$ with $\mathfrak{A}_{ff} \models \psi^{ff}$, we show there exists a Σ_{vf}-model of ψ^{vf} by explicitly constructing it from the Σ_{ff}-model. The central idea is again to ensure that all g-functions $(g_{\texttt{fuel}}, g_0, \dots)$ are assigned the same value by the two structures. We will see that the synonym axioms ensure that $g_0, \dots, g_{mf}$ are interpreted the same. This is important as otherwise Theorem 3.12 does not hold, and we can get spurious counterexamples (cf. Example 3.10).

Definition 3.17. Variable fuel structure

Let $\mathfrak{A}_{ff} = \langle \mathbf{A}, \square^{\mathfrak{A}_{ff}} \rangle$ be a Σ_{ff}-structure such that $\mathfrak{A}_{ff} \models \psi^{ff}$. The corresponding Σ_{vf}-structure $\mathfrak{A}_{vf} = \langle \mathbf{A}^{vf}, \square^{\mathfrak{A}_{vf}} \rangle$ is defined by:

$$
\begin{aligned}
\mathbf{A}^{vf} &= \mathfrak{A} \cup \mathbb{N}_0 \\
\sigma^{\mathfrak{A}_{vf}} &= \begin{cases} \mathbb{N}_0, & \text{if } \sigma = \texttt{Fuel} \\ \sigma^{\mathfrak{A}_{ff}}, & \text{if } \sigma \in \Sigma^{\mathsf{S}}_{ff} \end{cases} && \text{for } \sigma \in \Sigma^{\mathsf{S}}_{vf} \\
(f\colon \bar{\sigma}\ \rho)^{\mathfrak{A}_{vf}} &= \begin{cases} \lambda \texttt{fuel}, \bar{d}.\ (g_0\colon \bar{D}\ \bar{R})^{\mathfrak{A}_{ff}}(\bar{d}), & \text{if } f\colon \bar{\sigma}\ \rho = g_{\texttt{fuel}}\colon \texttt{Fuel}\ \bar{D}\ R \\ 0, & \text{if } f\colon \bar{\sigma}\ \rho = Z\colon \texttt{Fuel} \\ \lambda \texttt{fuel}.\ \texttt{fuel} + 1, & \text{if } f\colon \bar{\sigma}\ \rho = S\colon \texttt{Fuel}\ \texttt{Fuel} \\ (f\colon \bar{\sigma}\ \rho)^{\mathfrak{A}_{vf}}, & \text{else} \end{cases} && \text{for } f\colon \bar{\sigma}\ \rho \in \Sigma_{vf}
\end{aligned}
$$

The Variable fuel encoding introduces the new sort `Fuel`. Consistent with the intuition given when defining the encodings, it is interpreted as the set of natural numbers. The additional function symbols Z and S are interpreted as 0 and incrementing by 1 respectively. $g_{\texttt{fuel}}$ is mapped to g_0 ignoring the `fuel` parameter. The next lemma shows that we could have chosen any g_i, since $\mathfrak{A}_{ff}$ has to satisfy the synonyms axioms and thus all g_i are interpreted the same.

Lemma 3.18. Fuel functions coincide (fixed fuel)

If $\mathfrak{A}_{ff} \models \psi^{ff}$, then $g^* \coloneqq (g_0\colon \bar{D}\ R)^{\mathfrak{A}_{ff}} = (g_i\colon \bar{D}\ R)^{\mathfrak{A}_{ff}}$ for all $i = 0, \dots, mf$.

Proof. By induction on i. For $i = 0$, the claim holds by definition. Assuming that the claim holds for $i < mf$, we show that it also holds for $i + 1$. To this end, assume that the claim is false, i.e. $(g_{i+1}\colon \bar{D}\ R)^{\mathfrak{A}_{ff}} \neq g^* = (g_i\colon \bar{D}\ R)^{\mathfrak{A}_{ff}}$. But then clearly, $\mathfrak{A}_{ff} \not\models \sigma^{ff}_{i+1}$ and hence $\mathfrak{A}_{ff} \not\models \psi^{ff}$. We arrive at a contradiction. Therefore, $(g_{i+1}\colon \bar{D}\ R)^{\mathfrak{A}_{ff}} = g^*$ must hold. □

It immediately follows that, for any $i, j \in \{0, \dots, mf\}$, Σ_{ff}-term t, model $\mathfrak{A}_{ff}$ of ψ^{ff}, and $\mathfrak{I}_{ff} = \langle \mathfrak{A}_{ff}, \mathfrak{v} \rangle$

$$[\![t]\!]^{\mathfrak{I}_{ff}} = [\![t[g_i(\bar{d} : \bar{D}) \mapsto g_j(\bar{d})]]\!]^{\mathfrak{I}_{ff}}$$

holds. With our definition of $\mathfrak{A}_{vf}$, the same holds for Σ_{vf}-terms. Meaning, we can change the fuel parameter without affecting the term's value.

Lemma 3.19. Fuel value is irrelevant (variable fuel)

Let t be a Σ_{vf}-term and $\mathfrak{I}_{vf} = \langle \mathfrak{A}_{vf}, \mathfrak{v} \rangle$ an interpretation with structure $\mathfrak{A}_{vf}$. For any term $t_{\texttt{fuel}}$ of sort `Fuel`, it holds that

$$[\![t]\!]^{\mathfrak{I}_{vf}} = [\![t[g_{\texttt{fuel}}(\_ : \texttt{Fuel}, \bar{d'} : \bar{D}) \mapsto g_{\texttt{fuel}}(t_{\texttt{fuel}}, \bar{d'})]]\!]^{\mathfrak{I}_{vf}}.$$

Proof. By structural induction on t. The claim directly follows from the definition of $\mathfrak{A}_{vf}$ – *Omitted* □

The main step towards showing that $\mathfrak{A}_{vf}$ is a model of ψ^{vf} is realizing that $g_0, \dots, g_{mf}$ and $g_{\texttt{fuel}}$ can be interchanged and the two structures (packaged into interpretations) assign the same values to the resulting terms.

Lemma 3.20. Preservation of value (variable fuel)

Given two interpretations $\mathfrak{I}_{ff} = \langle \mathfrak{A}_{ff}, \mathfrak{v} \rangle, \mathfrak{I}_{vf} = \langle \mathfrak{A}_{vf}, \mathfrak{v} \rangle$, $i \in \{0, \dots, mf\}$ and some term $t_{\texttt{fuel}}$ of sort `Fuel`, then for every Σ-term t, Σ_{ff}-term $t^{ff} \coloneqq t[g(\bar{d'} : \bar{D}) \mapsto g_i(\bar{d'})]$ and Σ_{vf}-term $t^{vf} \coloneqq t[g(\bar{d'} : \bar{D}) \mapsto g_{\texttt{fuel}}(t_{\texttt{fuel}}, \bar{d'})]$, it holds that

$$[\![t^{ff}]\!]^{\mathfrak{I}_{ff}} = [\![t^{vf}]\!]^{\mathfrak{I}_{vf}}.$$

In particular, if these are sentences it holds that, $\mathfrak{A}_{ff} \models t^{ff}$ if and only if $\mathfrak{A}_{vf} \models t^{vf}$.

Proof. By structural induction on t.

Case $t^{ff} = x = t^{vf}$. The claim directly follows from $[\![x]\!]^{\mathfrak{I}_{ff}} = \mathfrak{v}(x) = [\![x]\!]^{\mathfrak{I}_{vf}}$.

Case $t^{ff} = \forall \bar{x} : \bar{\sigma}.\ t'[g(\bar{d'} : \bar{D}) \mapsto g_i(\bar{d'})]$ *and* $t^{vf} = \forall \bar{x} : \bar{\sigma}.\ t'[g(\bar{d'} : \bar{D}) \mapsto g_{\texttt{fuel}}(t_{\texttt{fuel}}, \bar{d'})]$.

$$\begin{aligned}
[\![t^{ff}]\!]^{\mathfrak{I}_{ff}} &= [\![\forall \bar{x} : \bar{\sigma}.\ t'[g(\bar{d'} : \bar{D}) \mapsto g_i(\bar{d'})]]\!]^{\mathfrak{I}_{ff}} = \textbf{true} \\
&\text{iff } [\![t'[g(\bar{d'} : \bar{D}) \mapsto g_i(\bar{d'})]]\!]^{\mathfrak{I}_{ff}[\bar{x} : \bar{\sigma} \mapsto \bar{a}]} = \textbf{true} \text{ for all } \bar{a} \in \bar{\sigma}^{\mathfrak{A}_{ff}} && \text{(by def. of } [\![\square]\!]^{\mathfrak{I}_{ff}}) \\
&\text{iff } [\![t'[g(\bar{d'} : \bar{D}) \mapsto g_{\texttt{fuel}}(t_{\texttt{fuel}}, \bar{d'})]]\!]^{\mathfrak{I}_{vf}[\bar{x} : \bar{\sigma} \mapsto \bar{a}]} = \textbf{true} \text{ for all } \bar{a} \in \bar{\sigma}^{\mathfrak{A}_{ff}} && \text{(induction hypothesis)} \\
&\text{iff } [\![t'[g(\bar{d'} : \bar{D}) \mapsto g_{\texttt{fuel}}(t_{\texttt{fuel}}, \bar{d'})]]\!]^{\mathfrak{I}_{vf}[\bar{x} : \bar{\sigma} \mapsto \bar{a}]} = \textbf{true} \text{ for all } \bar{a} \in \bar{\sigma}^{\mathfrak{A}_{vf}} && \text{(none of the } \bar{\sigma} \text{ can be } \texttt{Fuel}) \\
&\text{iff } [\![t^{vf}]\!]^{\mathfrak{I}_{vf}} = \textbf{true} && \text{(by def. of } [\![\square]\!]^{\mathfrak{I}_{vf}})
\end{aligned}$$

Case $t^{ff} = \exists \bar{x} : \bar{\sigma}.\ t'[g(\bar{d'} : \bar{D}) \mapsto g_i(\bar{d'})]$ *and* $t^{vf} = \exists \bar{x} : \bar{\sigma}.\ t'[g(\bar{d'} : \bar{D}) \mapsto g_{\texttt{fuel}}(t_{\texttt{fuel}}, \bar{d'})]$. *analogous*

Case $t^{ff} = g_i(\bar{t'}[g(\bar{d'} : \bar{D}) \mapsto g_i(\bar{d'})])$ *with* $\bar{t'} : \bar{D}$ *and* $t^{vf} = g_{\texttt{fuel}}(t_{\texttt{fuel}}, \bar{t'}[g(\bar{d'} : \bar{D}) \mapsto g_{\texttt{fuel}}(t_{\texttt{fuel}}, \bar{d'})])$.

$$\begin{aligned}
[\![t^{ff}]\!]^{\mathfrak{I}_{ff}} &= [\![g_i(\bar{t'}[g(\bar{d'} : \bar{D}) \mapsto g_i(\bar{d'})])]\!]^{\mathfrak{I}_{ff}} \\
&= (g_i : \bar{D}\ R)^{\mathfrak{A}_{ff}}([\![\bar{t'}[g(\bar{d'} : \bar{D}) \mapsto g_i(\bar{d'})]]\!]^{\mathfrak{I}_{ff}}) && \text{(by def. of } [\![\square]\!]^{\mathfrak{I}_{ff}}) \\
&= (g_i : \bar{D}\ R)^{\mathfrak{A}_{ff}}([\![\bar{t'}[g(\bar{d'} : \bar{D}) \mapsto g_{\texttt{fuel}}(t_{\texttt{fuel}}, \bar{d'})]]\!]^{\mathfrak{I}_{vf}}) && \text{(induction hypothesis)} \\
&= (g_0 : \bar{D}\ R)^{\mathfrak{A}_{ff}}([\![\bar{t'}[g(\bar{d'} : \bar{D}) \mapsto g_{\texttt{fuel}}(t_{\texttt{fuel}}, \bar{d'})]]\!]^{\mathfrak{I}_{vf}}) && \text{(by Lemma 3.18)} \\
&= (g_{\texttt{fuel}} : \texttt{Fuel}\ \bar{D}\ R)^{\mathfrak{A}_{vf}}([\![t_{\texttt{fuel}}]\!]^{\mathfrak{I}_{vf}}, [\![\bar{t'}[g(\bar{d'} : \bar{D}) \mapsto g_{\texttt{fuel}}(t_{\texttt{fuel}}, \bar{d'})]]\!]^{\mathfrak{I}_{vf}}) && \text{(by def. of } \mathfrak{A}_{vf}) \\
&= [\![t^{vf}]\!]^{\mathfrak{I}} && \text{(by def. of } [\![\square]\!]^{\mathfrak{I}_{vf}})
\end{aligned}$$

Case $t^{ff} = f(\bar{t}'[g(\bar{d}' : \bar{D}) \mapsto g_i(\bar{d}')])$ *with* $\bar{t}' : \bar{\sigma}$, $f \neq g_i$ *or* $\bar{\sigma} \neq \bar{D}$ *and* $t^{vf} = f(\bar{t}'[g(\bar{d}' : \bar{D}) \mapsto g_{\mathtt{fuel}}(t_{\mathtt{fuel}}, \bar{d}')])$.

$$
\begin{aligned}
[\![t^{ff}]\!]^{\mathfrak{I}_{ff}} &= [\![f(\bar{t}'[g(\bar{d}' : \bar{D}) \mapsto g_i(\bar{d}')])]\!]^{\mathfrak{I}_{ff}} \\
&= (f : \bar{\sigma}\ \rho)^{\mathfrak{A}_{ff}}([\![\bar{t}'[g(\bar{d}' : \bar{D}) \mapsto g_i(\bar{d}')]]\!]^{\mathfrak{I}_{ff}}) && \text{(by def. of } [\![\boxdot]\!]^{\mathfrak{I}_{ff}}) \\
&= (f : \bar{\sigma}\ \rho)^{\mathfrak{A}_{ff}}([\![\bar{t}'[g(\bar{d}' : \bar{D}) \mapsto g_{\mathtt{fuel}}(t_{\mathtt{fuel}}, \bar{d}')]]\!]^{\mathfrak{I}_{vf}}) && \text{(induction hypothesis)} \\
&= (f : \bar{\sigma}\ \rho)^{\mathfrak{A}_{vf}}([\![\bar{t}'[g(\bar{d}' : \bar{D}) \mapsto g_{\mathtt{fuel}}(t_{\mathtt{fuel}}, \bar{d}')]]\!]^{\mathfrak{I}_{vf}}) && \text{(by def. of } \mathfrak{A}_{vf}) \\
&= [\![t^{vf}]\!]^{\mathfrak{I}_{vf}} && \text{(by def. of } [\![\boxdot]\!]^{\mathfrak{I}_{vf}})
\end{aligned}
$$

Note that by the construction of the terms f is neither Z nor S. □

We can now show that we have actually constructed a model for ψ^{vf}.

Lemma 3.21. Variable fuel model

The constructed Σ_{vf}-structure $\mathfrak{A}_{vf}$ is a model of ψ^{vf}, i.e. $\mathfrak{A}_{vf} \models \psi^{vf}$.

Proof. We show that $\mathfrak{A}_{vf}$ satisfies ψ^{vf}, by showing that $\mathfrak{A}_{vf}$ satisfies each conjunct of ψ^{vf}. Let $\mathfrak{I}_{ff} = \langle \mathfrak{A}_{ff}, \mathfrak{v} \rangle$ and $\mathfrak{I}_{vf} = \langle \mathfrak{A}_{vf}, \mathfrak{v} \rangle$ denote some interpretations with structures $\mathfrak{A}_{ff}/\mathfrak{A}_{vf}$ and identical variable assignments $\mathfrak{v}$.

$\mathfrak{A}_{vf} \models \delta_g^{vf} = \forall \mathtt{fuel} : \mathtt{Fuel}, \bar{d} : \bar{D}.\ g_{\mathtt{fuel}}(S(\mathtt{fuel}), \bar{d}) \approx def_g[g(\bar{d}' : \bar{D}) \mapsto g_{\mathtt{fuel}}(\mathtt{fuel}, \bar{d}')]$. All g-functions are assigned the same values. Since,

$$\mathfrak{A}_{ff} \models \delta_{g_1}^{vf} \quad \text{and} \quad \mathfrak{A}_{ff} \models \sigma_{g_1}^{vf}$$

we also have

$$\mathfrak{A}_{ff} \models \forall \bar{d} : \bar{D}.\ g_1(\bar{d}) \approx \mathrm{def}_g[g(\bar{d}' : \bar{D}) \mapsto g_1(\bar{d})] \qquad (*)$$

and can conclude

$$
\begin{aligned}
&[\![\delta_g^{vf}]\!]^{\mathfrak{I}_{vf}} \\
&= [\![\forall \mathtt{fuel} : \mathtt{Fuel}, \bar{d} : \bar{D}.\ g_{\mathtt{fuel}}(Z, \bar{d}) \approx \mathrm{def}_g[g(\bar{d}' : \bar{D}) \mapsto g_{\mathtt{fuel}}(Z, \bar{d}')]]\!]^{\mathfrak{I}_{vf}} && \text{(by Lemma 3.19)} \\
&= [\![\forall \bar{d} : \bar{D}.\ g_{\mathtt{fuel}}(Z, \bar{d}) \approx \mathrm{def}_g[g(\bar{d}' : \bar{D}) \mapsto g_{\mathtt{fuel}}(Z, \bar{d}')]]\!]^{\mathfrak{I}_{vf}} && (\mathtt{fuel} \text{ does not occur}) \\
&= [\![\delta_g[g(\bar{d}' : \bar{D}) \mapsto g_{\mathtt{fuel}}(Z, \bar{d}')]]\!]^{\mathfrak{I}_{vf}} && \text{(by term equality)} \\
&= [\![\delta_g[g(\bar{d}' : \bar{D}) \mapsto g_1(\bar{d})]]\!]^{\mathfrak{I}_{ff}} && \text{(by Lemma 3.20)} \\
&= \mathbf{true} && (*)
\end{aligned}
$$

$\mathfrak{A}_{ff} \models \sigma_g^{vf} = \forall \mathtt{fuel} : \mathtt{Fuel}, \bar{d} : \bar{D}.\ g_{\mathtt{fuel}}(S(\mathtt{fuel}), \bar{d}) \approx g_{\mathtt{fuel}}(\mathtt{fuel}, \bar{d})$. The synonym axiom is satisfied, since $\mathfrak{A}_{vf}$ ignores the fuel parameter when interpreting $g_{\mathtt{fuel}}$.

$$
\begin{aligned}
&[\![\sigma_g^{vf}]\!]^{\mathfrak{I}_{vf}} = \mathbf{true} \\
&\text{iff } [\![\sigma_g^{vf}[g_{\mathtt{fuel}}(\_ : \mathtt{Fuel}, \bar{d}' : \bar{D}) \mapsto g_{\mathtt{fuel}}(\mathtt{fuel}, \bar{d}')]]\!]^{\mathfrak{I}_{vf}} = \mathbf{true} && \text{(by Lemma 3.19)} \\
&\text{iff } [\![g_{\mathtt{fuel}}(\mathtt{fuel}, \bar{d}) \approx g_{\mathtt{fuel}}(\mathtt{fuel}, \bar{d})]\!]^{\mathfrak{I}_{vf}[\mathtt{fuel} \mapsto f, \bar{d} : \bar{D} \mapsto \bar{a}]} = \mathbf{true} \\
&\qquad \text{for all } f \in \mathtt{Fuel}^{\mathfrak{A}_{vf}}, \bar{a} \in \bar{D}^{\mathfrak{A}_{vf}} && \text{(by def. of } [\![\boxdot]\!]^{\mathfrak{I}_{vf}}) \\
&\text{iff } \mathbf{true} = \mathbf{true} && \text{(by def. of } \approx)
\end{aligned}
$$

$\mathfrak{A}_{vf} \models \varphi[g(\bar{d}' : \bar{D}) \mapsto g_{\mathtt{fuel}}(mf, \bar{d}')]$. Directly follows from $\mathfrak{A}_{ff} \models \varphi[g(\bar{d}' : \bar{D}) \mapsto g_{mf}(\bar{d}')]$ using Lemma 3.20. □

Variable fuel encoding to Default encoding

Assuming there exists a Σ_{vf}-structure $\mathfrak{A}_{vf}$ with $\mathfrak{A}_{vf} \models \psi^{vf}$, we show there exists a Σ-model of ψ by explicitly constructing it from the Σ_{vf}-model. As previously, the construction relies on the fact that all g-functions are interpreted the same.

Definition 3.22. Default structure

Let $\mathfrak{A}_{vf} = \langle \mathbf{A}, \cdot^{\mathfrak{A}_{vf}} \rangle$ be a Σ_{vf}-structure such that $\mathfrak{A}_{vf} \models \psi^{vf}$. The corresponding Σ-structure $\mathfrak{A} = \langle \mathbf{A}, \cdot^{\mathfrak{A}} \rangle$ is defined by (notice $\Sigma^{\mathrm{S}} \subseteq \Sigma^{\mathrm{S}}_{vf}$):

$$\sigma^{\mathfrak{A}} = \sigma^{\mathfrak{A}_{vf}} \qquad \text{for } \sigma \in \Sigma^{\mathrm{S}}$$

$$(f \colon \bar{\sigma}\, \rho)^{\mathfrak{A}} = \begin{cases} \lambda \bar{d}.\ (g_{\mathtt{fuel}} \colon \mathtt{Fuel}\ \bar{D}\, R)^{\mathfrak{A}_{vf}}(\llbracket Z \rrbracket^{\mathfrak{I}_{vf}}, \bar{d}), & \text{if } f \colon \bar{\sigma}\, \rho = g \colon \bar{D}\, R \\ (f \colon \bar{\sigma}\, \rho)^{\mathfrak{A}_{vf}}, & \text{otherwise} \end{cases} \qquad \text{for } f \colon \bar{\sigma}\, \rho \in \Sigma^{\mathrm{F}}$$

The fact that the universe $\mathbf{A}$ contains values of sort `Fuel`, which is not present in Σ^{S}, is not a problem. These parts simply remain unused by $\mathfrak{A}$. The next lemma is based on the observation that, at least for `fuel` values that are only constructed from Z and S, the `fuel` parameter is irrelevant for the result of $g_{\mathtt{fuel}} \colon \mathtt{Fuel}\ \bar{D}\, R$.

Lemma 3.23. Fuel functions coincide (variable fuel)

Let $\mathfrak{I}_{vf} = \langle \mathfrak{A}_{vf}, \mathfrak{v} \rangle$ be an interpretation that satisfies ψ^{vf} ($\llbracket \psi^{vf} \rrbracket^{\mathfrak{I}_{vf}} = \mathbf{true}$), then for a closed term $t_{\mathtt{fuel}}$ of sort `Fuel` and a tuple of argument values $\bar{a} \in \bar{D}^{\mathfrak{A}_{vf}}$

$$(g_{\mathtt{fuel}} \colon \mathtt{Fuel}\ \bar{D}\, R)^{\mathfrak{A}_{vf}}(\llbracket Z \rrbracket^{\mathfrak{I}_{vf}}, \bar{a}) = (g_{\mathtt{fuel}} \colon \mathtt{Fuel}\ \bar{D}\, R)^{\mathfrak{A}_{vf}}(\llbracket t_{\mathtt{fuel}} \rrbracket^{\mathfrak{I}_{vf}}, \bar{a}).$$

Proof. By induction on the closed fuel term $t_{\mathtt{fuel}}$. Notice that it must have either the form Z or $S(t'_{\mathtt{fuel}})$, for some other closed term $t'_{\mathtt{fuel}}$ of sort `Fuel`.

Case $t_{\mathtt{fuel}} = Z$. Holds by term equality.

Case $t_{\mathtt{fuel}} = S(t'_{\mathtt{fuel}})$. Assuming that the claim is false, i.e.

$$\begin{aligned} & (g_{\mathtt{fuel}} \colon \mathtt{Fuel}\ \bar{D}\, R)^{\mathfrak{A}_{vf}}(\llbracket Z \rrbracket^{\mathfrak{I}_{vf}}, \bar{a}) \\ = {} & (g_{\mathtt{fuel}} \colon \mathtt{Fuel}\ \bar{D}\, R)^{\mathfrak{A}_{vf}}(\llbracket t'_{\mathtt{fuel}} \rrbracket^{\mathfrak{I}_{vf}}, \bar{a}) && \text{(induction hypothesis)} \\ \neq {} & (g_{\mathtt{fuel}} \colon \mathtt{Fuel}\ \bar{D}\, R)^{\mathfrak{A}_{vf}}(\llbracket t_{\mathtt{fuel}} \rrbracket^{\mathfrak{I}_{vf}}, \bar{a}) && \text{(assuming the claim is false)} \\ = {} & (g_{\mathtt{fuel}} \colon \mathtt{Fuel}\ \bar{D}\, R)^{\mathfrak{A}_{vf}}(\llbracket S(t'_{\mathtt{fuel}}) \rrbracket^{\mathfrak{I}_{vf}}, \bar{a}) && t_{\mathtt{fuel}} = S(t'_{\mathtt{fuel}}), \end{aligned}$$

we arrive at a contradiction. The statement clearly violates σ_g^{vf} but $\llbracket \sigma_g^{vf} \rrbracket^{\mathfrak{I}_{vf}} = \mathbf{true}$. □

Similar to before, the two structures (lifted to interpretations with the same variable assignment) assign the same value to terms when interchanging $g \colon \bar{D}\, R$ and $g_{\mathtt{fuel}} \colon \mathtt{Fuel}\ \bar{D}\, R$.

Lemma 3.24. Preservation of value

Given two interpretations $\mathfrak{I}_{vf} = \langle \mathfrak{A}_{vf}, \mathfrak{v} \rangle, \mathfrak{I} = \langle \mathfrak{A}, \mathfrak{v} \rangle$ and some closed term $t_{\mathtt{fuel}}$ of sort `Fuel`, then for every Σ-term t, and Σ_{ff}-term $t^{vf} \coloneqq t[g(\bar{d}' \colon \bar{D}) \mapsto g_{\mathtt{fuel}}(t_{\mathtt{fuel}}, \bar{d}')]$, it holds that

$$\llbracket t^{vf} \rrbracket^{\mathfrak{I}_{vf}} = \llbracket t \rrbracket^{\mathfrak{I}}.$$

In particular, if t is a sentence it holds that, $\mathfrak{A}_{vf} \models t^{vf}$ if and only if $\mathfrak{A} \models t$.

Proof. By structural induction on t.

Case $t^{vf} = x = t$. The claim directly follows from $\llbracket x \rrbracket^{\mathfrak{I}_{vf}} = \mathfrak{v}(x) = \llbracket x \rrbracket^{\mathfrak{I}}$.

Case $t^{vf} = \forall \bar{x} \colon \bar{\sigma}.\ t'[g(\bar{d}' \colon \bar{D}) \mapsto g_{\mathtt{fuel}}(t_{\mathtt{fuel}}, \bar{d}')]$ *and* $t = \forall \bar{x} \colon \bar{\sigma}.\ t'$.

$$\begin{aligned} \llbracket t^{vf} \rrbracket^{\mathfrak{I}_{vf}} = \llbracket \forall \bar{x} \colon \bar{\sigma}.\ t'[g(\bar{d}' \colon \bar{D}) \mapsto g_{\mathtt{fuel}}(t_{\mathtt{fuel}}, \bar{d}')] \rrbracket^{\mathfrak{I}_{vf}} = \mathbf{true} & \\ \text{iff } \llbracket t'[g(\bar{d}' \colon \bar{D}) \mapsto g_{\mathtt{fuel}}(t_{\mathtt{fuel}}, \bar{d}')] \rrbracket^{\mathfrak{I}_{vf}[\bar{x} \colon \bar{\sigma} \mapsto \bar{a}]} = \mathbf{true} \text{ for all } \bar{a} \in \bar{\sigma}^{\mathfrak{A}_{vf}} & \quad \text{(by def. of } \llbracket \cdot \rrbracket^{\mathfrak{I}_{vf}}) \\ \text{iff } \llbracket t' \rrbracket^{\mathfrak{I}[\bar{x} \colon \bar{\sigma} \mapsto \bar{a}]} = \mathbf{true} \text{ for all } \bar{a} \in \bar{\sigma}^{\mathfrak{A}_{vf}} & \quad \text{(by induction hypothesis)} \\ \text{iff } \llbracket t' \rrbracket^{\mathfrak{I}[\bar{x} \colon \bar{\sigma} \mapsto \bar{a}]} = \mathbf{true} \text{ for all } \bar{a} \in \bar{\sigma}^{\mathfrak{A}} & \quad \text{(none of the } \bar{\sigma} \text{ can be } \mathtt{Fuel}) \\ \text{iff } \llbracket t \rrbracket^{\mathfrak{I}} = \mathbf{true} & \quad \text{(by def. of } \llbracket \cdot \rrbracket^{\mathfrak{I}}) \end{aligned}$$

Case $t^{vf} = \exists \bar{x}\colon \bar{\sigma}.\ t'[g(\bar{d'}\colon \bar{D}) \mapsto g_{\texttt{fuel}}(t_{\texttt{fuel}}, \bar{d'})]$ *and* $t = \exists \bar{x}\colon \bar{\sigma}.\ t'$. *analogous*

Case $t^{vf} = g_{\texttt{fuel}}(t_{\texttt{fuel}}, \bar{t'}[g(\bar{d'}\colon \bar{D}) \mapsto g_{\texttt{fuel}}(t_{\texttt{fuel}}, \bar{d'})])$ *with* $\bar{t'}\colon \bar{D}$ *and* $t = g(\bar{t'})$.

$$
\begin{aligned}
[\![t^{vf}]\!]^{\mathfrak{I}_{vf}} &= [\![g_{\texttt{fuel}}(t_{\texttt{fuel}}, \bar{t'}[g(\bar{d'}\colon \bar{D}) \mapsto g_{\texttt{fuel}}(t_{\texttt{fuel}}, \bar{d'})])]\!]^{\mathfrak{I}_{vf}} \\
&= (g_{\texttt{fuel}}\colon \texttt{Fuel}\ \bar{D}\ R)^{\mathfrak{A}_{vf}}([\![t_{\texttt{fuel}}]\!]^{\mathfrak{I}_{vf}}, [\![\bar{t'}[g(\bar{d'}\colon \bar{D}) \mapsto g_{\texttt{fuel}}(t_{\texttt{fuel}}, \bar{d'})]]\!]^{\mathfrak{I}_{vf}}) && \text{(by def. of } [\![\boxdot]\!]^{\mathfrak{I}_{vf}}) \\
&= (g_{\texttt{fuel}}\colon \texttt{Fuel}\ \bar{D}\ R)^{\mathfrak{A}_{vf}}([\![t_{\texttt{fuel}}]\!]^{\mathfrak{I}_{vf}}, [\![\bar{t'}]\!]^{\mathfrak{I}}) && \text{(by induction hypothesis)} \\
&= (g_{\texttt{fuel}}\colon \texttt{Fuel}\ \bar{D}\ R)^{\mathfrak{A}_{vf}}([\![Z]\!]^{\mathfrak{I}_{vf}}, [\![\bar{t'}]\!]^{\mathfrak{I}}) && \text{(by Lemma 3.23)} \\
&= (g\colon \bar{D}\ R)^{\mathfrak{A}}([\![\bar{t'}]\!]^{\mathfrak{I}}) && \text{(by def. of } \mathfrak{A}) \\
&= [\![t]\!]^{\mathfrak{I}} && \text{(by def. of } [\![\boxdot]\!]^{\mathfrak{I}})
\end{aligned}
$$

Case $t^{vf} = f(\bar{t'}[g(\bar{d'}\colon \bar{D}) \mapsto g_{\texttt{fuel}}(t_{\texttt{fuel}}, \bar{d'})])$ *with* $\bar{t'}\colon \bar{\sigma}$, $f \neq g_{\texttt{fuel}}$ *or* $\bar{\sigma} \neq \bar{D}$ *and* $t = f(\bar{t'})$.

$$
\begin{aligned}
[\![t^{vf}]\!]^{\mathfrak{I}_{vf}} &= [\![f(\bar{t'}[g(\bar{d'}\colon \bar{D}) \mapsto g_{\texttt{fuel}}(t_{\texttt{fuel}}, \bar{d'})])]\!]^{\mathfrak{I}_{vf}} \\
&= (f\colon \bar{\sigma}\ \rho)^{\mathfrak{A}_{vf}}([\![\bar{t'}[g(\bar{d'}\colon \bar{D}) \mapsto g_{\texttt{fuel}}(t_{\texttt{fuel}}, \bar{d'})]]\!]^{\mathfrak{I}_{vf}}) && \text{(by def. of } [\![\boxdot]\!]^{\mathfrak{I}_{vf}}) \\
&= (f\colon \bar{\sigma}\ \rho)^{\mathfrak{A}_{vf}}([\![\bar{t'}]\!]^{\mathfrak{I}}) && \text{(by induction hypothesis)} \\
&= (f\colon \bar{\sigma}\ \rho)^{\mathfrak{A}}([\![\bar{t'}]\!]^{\mathfrak{I}}) && \text{(by def. of } \mathfrak{A}^{ff}) \\
&= [\![t]\!]^{\mathfrak{I}} && \text{(by def. of } [\![\boxdot]\!]^{\mathfrak{I}})
\end{aligned}
$$

□

We finish, by showing that we have successfully constructed a model for ψ.

> Lemma 3.25. Default model
> The constructed Σ-structure $\mathfrak{A}$ is a model of ψ, i.e. $\mathfrak{A} \models \psi$.

Proof. We show that $\mathfrak{A}$ satisfies ψ, by showing that $\mathfrak{A}$ satisfies each conjunct of ψ.

$\mathfrak{A} \models \delta_g = \forall \bar{d}\colon \bar{D}.\ g(\bar{d}) \approx def_g$. Since, $\mathfrak{A}_{vf} \models \delta_g^{vf}$ and $\mathfrak{A}_{vf} \models \sigma_g^{vf}$ we also have

$$\mathfrak{A}_{vf} \models \forall \texttt{fuel}\colon \texttt{Fuel}, \bar{d}\colon \bar{D}.\ g_{\texttt{fuel}}(\texttt{fuel}, \bar{d}) \approx \text{def}[g(\bar{d'}\colon \bar{D}) \mapsto g_{\texttt{fuel}}(\texttt{fuel}, \bar{d'})].$$

In particular,

$$\mathfrak{A}_{vf} \models \forall \bar{d}\colon \bar{D}.\ g_{\texttt{fuel}}(Z, \bar{d}) \approx \text{def}_g[g(\bar{d'}\colon \bar{D}) \mapsto g_{\texttt{fuel}}(Z, \bar{d'})] \qquad (*)$$

Let $\mathfrak{I} = \langle \mathfrak{A}, \mathfrak{v} \rangle$ and $\mathfrak{I}_{vf} = \langle \mathfrak{A}_{vf}, \mathfrak{v} \rangle$ denote some interpretations with structures $\mathfrak{A}/\mathfrak{A}_{vf}$ and identical variable assignments $\mathfrak{v}$.

$$
\begin{aligned}
&[\![\delta_g]\!]^{\mathfrak{I}} \\
&= [\![\forall \bar{d}\colon \bar{D}.\ g(\bar{d})[g(\bar{d'}\colon \bar{D}) \mapsto g_{\texttt{fuel}}(Z, \bar{d'})] \approx \text{def}_g[g(\bar{d'}\colon \bar{D}) \mapsto g_{\texttt{fuel}}(Z, \bar{d'})]]\!]^{\mathfrak{I}_{vf}} && \text{(Lemma 3.24)} \\
&= \textbf{true} && (*)
\end{aligned}
$$

$\mathfrak{A} \models \varphi$. Directly follows from $\mathfrak{A}_{vf} \models \varphi[g(\bar{d'}\colon \bar{D}) \mapsto g_{\texttt{fuel}}(mf, \bar{d'})]$ using Lemma 3.24. □

3.2.2 High-level Soundness

Theorem 3.12 (Equisatisfiability of encodings) might suggest that one can arbitrarily choose any encoding (preferably the Default encoding as it is the simplest) since they are all logically equivalent. Although true in theory, this is not the case in practice, where we rely on semi decision procedures. Remember that we initialy motivated the fuel encodings as a strategy for preventing matching loops — a problem related to E-matching. E-matching is the primary heuristics used by SMT solvers to show unsatisfiable in the presence of quantifiers. The fuel encodings are designed to work well with E-matching. It is therefore also required to turn off other quantifier instantiation heuristics, namely MBQI. The triggers used by the encodings were ignored in the proof. This is because they

```
1  func g(d̄:D̄): R = def_g
2  proc P pre pre post post S
```

Figure 3.28: Procedure with a function declaration.

carry no logical meaning in the reference SMT-LIB semantics, meaning they are irrelevant for determining the theoretical existence of a model. But in practise we rely on heuristics to find models, or in our case to show the absence of a model. For E-matching, good triggers are vital for guiding the search.

Theorem 3.12 is interesting because it lets us directly prove, that Caesar remains sound when using a fuel encoding for user-defined functions. This is done in the remainder of this section. Since Caesar heavily relies on SMT solvers, we first define more precisely what an SMT solver does.

Definition 3.26. SMT solver

An SMT solver implements the following families of functions,

$$SMT_\Sigma : \textsc{Sentence}_\Sigma \times \textsc{Theory}_\Sigma \to \{\,\texttt{sat}\,\} \times \textsc{Structure}_\Sigma \cup \{\,\texttt{unknown}, \texttt{unsat}\,\},$$

i.e. given a Σ-sentence and a Σ-compatible theory it either returns `sat` together with a model, `unknown`, or `unsat`. Situations where the solver stops due to exceeded resource limits are grouped with the `unknown` case.

Assumtion 3.27. Soundness of SMT solvers

An SMT solver is sound. Meaning for a signature Σ, a Σ-sentence φ and a Σ-compatible theory $\mathcal{T}$

$SMT_\Sigma(\varphi, \mathcal{T}) = (\texttt{sat}, \mathfrak{A})$	implies	$\mathfrak{A} \in \mathcal{T}$ and $\mathfrak{A} \models \varphi$
$SMT_\Sigma(\varphi, \mathcal{T}) = \texttt{unsat}$	implies	there exists no $\mathfrak{A} \in \mathcal{T}$ such that $\mathfrak{A} \models \varphi$
$SMT_\Sigma(\varphi, \mathcal{T}) = \texttt{unknown}$	implies	*nothing in perticular*

If the SMT solver returns `sat`, then the returned model should satisfy the formula modulo $\mathcal{T}$. If the solver returns `unsat`, then the formula should not be satisfiable modulo $\mathcal{T}$. Given the result, `unknown` we did not learn any new information. It was neither possible to prove satisfiability nor unsatisfiability.

We also assume that Caesar was previously sound using the Default encoding.

Assumtion 3.29. Soundness of the Default encoding

Given a HeyVL program with the procedure P and the user-defined function $g\colon \bar{D}\, R$ in the form of Figure 3.28. If the user-defined function $g\colon \bar{D}\, R$ is internally encoded using the Default encoding, then

1. if Caesar verifies P, it holds that $pre \sqsubseteq \mathrm{vp}[\![S]\!](post)$, and
2. if Caesar finds a counterexample $\mathfrak{C}$ for P, then $pre(\mathfrak{C}) > \mathrm{vp}[\![S]\!](post)(\mathfrak{C})$.

With these preliminaries, we show that Caesar remains sound when using one of the fuel encodings. The soundness follows directly from the Equisatisfiability of encodings theorem (Theorem 3.12), stating that the results for the fuel encodings are always the same as for the default encoding.

Theorem 3.30. Soundness of Caesar

Given a HeyVL program with the procedure P and the user-defined function $g\colon \bar{D}\, R$ in the form of Figure 3.28. If the user-defined function $g\colon \bar{D}\, R$ is internally encoded using a fuel encoding, then

1. if Caesar verifies P, it holds that $pre \sqsubseteq \mathrm{vp}[\![S]\!](post)$, and
2. if Caesar finds a counterexample $\mathfrak{C}$ for P, then $pre(\mathfrak{C}) > \mathrm{vp}[\![S]\!](post)(\mathfrak{C})$.

Proof. We will only consider the Fixed fuel encoding for the proof, but the exact same argument works for the Variable fuel encoding as well. Previously, Caesar used the Default encoding where the final formula that is SAT checked is

$$\rho \quad \coloneqq \quad \mathcal{E}_{\text{term}}(pre \not\leq \mathrm{vp}[\![S]\!](post), \mathrm{def}_g).$$

This encoding is assumed to be sound (cf. Assumtion 3.29). The final formula produced by Caesar with the Fixed fuel encoding is

$$\rho^{ff} \quad \coloneqq \quad \mathcal{E}^{ff}_{\text{term}}(pre \not\leq \text{vp}[\![S]\!](post), \text{def}_g).$$

This is dispatched to an STM solver.

Case 1. If Caesar verified P, then the SMT solver returned `unsat`, i.e. $SMT_{\Sigma_{ff}}(\rho^{ff}, \mathcal{T}) = \texttt{unsat}$ for a suitable background theory $\mathcal{T}$ (EUF + NRA + NIA). By the assumption that the SMT solver is sound (Assumption Assumtion 3.27) this means that there exists no $\mathfrak{A} \in \mathcal{T}$ such that $\mathfrak{A} \models \rho^{ff}$. According to Theorem 3.12, this is equivalent to the non-existence of a model for ρ. Since the Default encoding is assumed to be sound $pre \sqsubseteq \text{vp}[\![S]\!](post)$ must be valid.

Case 2. If Caesar produced a counterexample for P, then the SMT solver returned `sat` with a model $\mathfrak{C}^{ff}$, i.e. $SMT_{\Sigma_{ff}}(\rho^{ff}, \mathcal{T}) = (\texttt{sat}, \mathfrak{C}^{ff})$. By Assumtion 3.27, this means that $\mathfrak{C}^{ff} \models \rho^{ff}$. According to Theorem 3.12, this means there also exists a model $\mathfrak{C}$ for ρ. The proof of Theorem 3.12 also gives us a way to construct $\mathfrak{C}$ from $\mathfrak{C}^{ff}$. By assumption, the model $\mathfrak{C}$ for ρ is a correct counterexample, such that $pre(\mathfrak{C}) > \text{vp}[\![S]\!](post)(\mathfrak{C})$ holds.

□

The previous theorem gives us the requested soundness guarantees. Namely, the first point ensures that wrong programs do not verify and the second point ensures that a reported counterexample is not spurious.

3.2.3 Incompleteness under E-matching Semantics

The fuel encodings will not have the intended effect when other quantifier instantiation techniques besides E-matching are used. Therefore, MBQI must be disabled. Otherwise, MBQI instantiations can introduce new ground terms that have a higher fuel value, which defeats the point of the fuel encodings. When only E-matching is used and MBQI is disabled, one will never get a `sat` response from the solver if the formula contains quantifiers. E-matching can never return `sat`, see Algorithm 1. We ignore this shortcoming that is related to the heuristic selection for now and return to it in Chapter 4 when discussing how we could obtain counterexamples.

In the context of a deductive verifier, showing `unsat` is generally more interesting, since this means that the program fulfils its specification. When it comes to showing unsatisfiability with E-matching, the Fixed fuel encoding and Variable fuel encoding are less complete than the Default encoding, since they allow for fewer instantiations. This leads to situations where the solver can theoretically prove `unsat` for the Default encoding, with only E-matching, but not for the other two (resulting in `unknown`).

Example 3.31. Fuel encodings cannot perform computation

The fuel encodings were specifically designed to only allow for mf instantiations. We can exploit this fact by constructing an assertion that requires $mf + 1$ instantiations.

Consider the Fixed fuel encoding of the factorial function with $mf = 1$ given in Figure 3.6. Proving the assertion `assert ?(fac1(1) == 1)` fails. The initial set of ground terms is $G^{ff}_1 = \{\,\texttt{fac1(1) != 1}\,\}$. The term `fac1(1)` matches both the trigger of the `fac_syn1` and the `fac_def1` axiom. Instantiating both axioms yields the new set of ground terms

$$\begin{aligned} G^{ff}_2 &= G_1 \cup \{\,\texttt{fac1(1) == fac0(1), fac1(1) == 1 * fac0(0)}\,\} \\ &= \{\,\texttt{fac1(1) != 1, fac1(1) == fac0(1), fac1(1) == 1 * fac0(0)}\,\}. \end{aligned}$$

The set of ground terms is not inconsistent, but the patterns do not match any of the new terms. Therefore, no new instantiations that can be performed. Pending a result, but also with no way to proceed, E-matching has to give up with `unknown`.

This failure is only due to the triggers preventing the required instantiations. Increasing the maximum fuel value mf to 2 would make the above assertion provable but would fail again for `fac2(2)`. In general, $fac_{mf}(mf) \approx mf!$ is never provable. Everything above also applies to the Variable fuel encoding.

Example 3.32. Default encoding can perform computation

Using Default encoding (Figure 3.1) to prove `assert ?(fac(1) == 1)` similarly works without problems. The initial set of ground term is $G_1 = \{\,$`fac(1) != 1`$\,\}$. After instantiating the `fac_def` axiom with `fac(1)` and then again with the resulting `fac(0)` term we get the sets

$$G_2 = \{\,\texttt{fac(1) != 1, fac(1) == 1 * fac(0)}\,\}$$
$$G_3 = \{\,\texttt{fac(1) != 1, fac(1) == 1 * fac(0), fac(0) == 1}\,\}.$$

This simplifies to

$$G_3' = \{\,\texttt{fac(1) != 1, fac(1) == 1, fac(0) == 1}\,\}.$$

Which is clearly inconsistent. Thus, E-matching returns with `unsat` and we have shown that `fac(1) == 1` must be true.

Generally, if a quantifier instantiation is possible in the fuel encodings, it can be matched in the Default encoding to obtain the same information. An instantiation of (ff-def-axiom)/(vf-def-axiom) can always be matched with an instantiation of the (def-axiom) due to the more liberal trigger, and a synonym axiom does not need to be matched since the information is already present. The reverse is not the case, as shown by the previous examples. None of this is in conflict with Theorem 3.12 but a result of the undecidability of the problem and the inherent incompleteness of E-matching.

3.2.4 Termination under E-matching Semantics

The initial motivation for introducing the fuel encodings was to prevent matching loops when encoding user-defined functions. We argue in this section that we have achieved this goal by giving an upper bound on the number of possible instantiations. Naturally, we assume that only E-matching is used for quantifier reasoning in this section.

When counting the instantiations caused by the term $\texttt{fac}_2(n)$, it is not enough to only examine the direct instantiations. The instantiations of the synonym and definitional axiom produce new $\texttt{fac}_1$-terms and so on. We therefore must also examine the transitive instantiations. For $\texttt{fac}_2(n)$ these are shown in an instantiation graph in Figure 3.33.

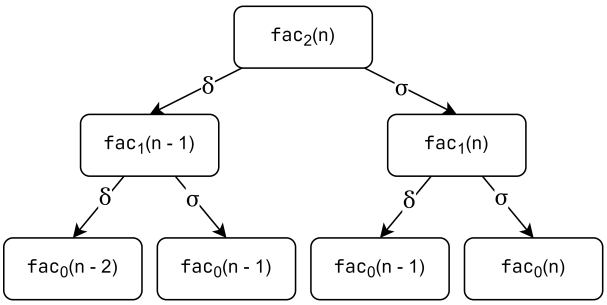


Figure 3.33: All (transitive) instantiations caused by the term $\texttt{fac}_2(n)$. A δ-arrow denotes an instantiation of the corresponding (ff-def-axiom) and a σ-arrow of the corresponding (ff-syn-axiom).

We call a term that is a function application of g with k fuel, i.e. a term of the form $g_k(\bar{d})$ or $g_{\texttt{fuel}}(k, \bar{d})$ (depending on the encoding), a g_k-term.

Intuitively, at some point no more instantiations are possible since the fuel value of the new function symbols decreases each time an axiom is instantiated. Once the fuel value reached 0, no new instantiations are possible. As can be seen in Figure 3.33, a $\texttt{fac}_0$-term (e.g. $\texttt{fac}_0(n-2)$ or $\texttt{fac}_0(3)$) causes 0 instantiations, a $\texttt{fac}_1$-term causes 2 instantiations, and a $\texttt{fac}_2$-term 6 instantiations.

Another fact that can be seen in Figure 3.33 is that each of the axioms is only instantiated once for each term. We formulate these as two observations as they are relevant for the proof.

Observation 1. A g_0-term can neither be used to instantiate the synonym axiom nor the definitional axiom. This is ensured by the chosen triggers.

Observation 2. Once a quantifier was instantiated using a g_k-term, then the same g_k-term cannot be used to

instantiate the same quantifier again. The second instantiation is prevented since it would not add any new ground terms. All produced ground terms were already added by the first instantiation.

We now consider the general case, of encoding the function $g\colon \bar{D}\, R$ with a fuel encoding. When encoding a formula φ, it can contain arbitrary additional quantifiers. We therefore assume that the only quantifiers producing g_k-terms are the ones introduced by the Fixed fuel encoding/Variable fuel encoding. Instantiating the synonym axiom results in exactly one (potentially) new g_k-term. The number of new g_k-term produced by the definitional axiom depends on the number of occurrences (recursive calls) of g in def_g, in the following denoted by $\#_g(\mathrm{def}_g)$.

Lemma 3.34
Given a signature Σ, a function $g\colon \bar{D}\, R \in \Sigma$, a Σ-sentence φ, and def_g, then during the proof search of the fuel encoded formula φ ($\mathcal{E}^{ff}_{\mathrm{term}}(\varphi, \mathrm{def}_g)/\mathcal{E}^{vf}_{\mathrm{term}}(\varphi, \mathrm{def}_g)$), each g_k ground term can cause at most

$$\sum_{i=0}^{k-1} 2 \cdot (\#_g(\mathrm{def}_g) + 1)^i$$

E-matching instantiations (direct and transitively).

Proof. By induction on k.

Induction base ($k = 0$). By Observation 1, a g_0-term can be instantiated 0 times, which is bounded by

$$\sum_{i=0}^{0-1} 2 \cdot (\#_g(\mathrm{def}_g) + 1)^i = 0.$$

Induction step ($k \rightsquigarrow k+1$). Assume now that a g_k-term can cause at most $\sum_{i=0}^{k-1} 2 \cdot (\#_g(\mathrm{def}_g) + 1)^i$ instantiations. A g_{k+1}-term can trigger the corresponding synonym and definitional axiom each at most once (see Observation 2).

- The instantiation of the synonym axiom produces a g_k-term. By induction hypothesis, this term can cause at most $\sum_{i=0}^{k-1} 2 \cdot (\#_g(\mathrm{def}_g) + 1)^i$ instantiations.
- The instantiation of the definitional axiom produces $\#_g(\mathrm{def}_g)$ g_k-terms. By induction hypothesis, these terms can cause at most $\sum_{i=0}^{k-1} 2 \cdot (\#_g(\mathrm{def}_g) + 1)^i$ instantiations each.

In total, that makes at most

$$\begin{aligned}
&2 + (\#_g(\mathrm{def}_g) + 1) \cdot \sum_{i=0}^{k-1} 2 \cdot (\#_g(\mathrm{def}_g) + 1)^i \\
&= 2 + \sum_{i=0}^{k-1} 2 \cdot (\#_g(\mathrm{def}_g) + 1)^{i+1} \\
&= 2 + \sum_{i=1}^{k} 2 \cdot (\#_g(\mathrm{def}_g) + 1)^i \\
&= \sum_{i=0}^{k} 2 \cdot (\#_g(\mathrm{def}_g) + 1)^i
\end{aligned}$$

instantiations. This is precisely our claimed upper bound.

□

Since we have a fixed maximal fuel value mf and the number of initial g_{mf} ground terms is finite ($N_g \coloneqq \#_{g_{mf}}(\mathcal{E}^{ff}_{\mathrm{term}}(\varphi, \mathrm{def}_g)) = \#_{g_{\mathsf{fuel}}}(\mathcal{E}^{vf}_{\mathrm{term}}(\varphi, \mathrm{def}_g))$), the maximum number of quantifier instantiations caused by the encodings is

$$N_g \cdot \sum_{i=0}^{mf-1} 2 \cdot (\#_g(\mathrm{def}_g) + 1)^i.$$

Thus, the fuel encodings are guaranteed to terminate.

Note again that this only true under the assumption that φ does not contain quantifiers that produce g-terms and that only E-matching is used for quantifier instantiation. The latter can be ensured by configuring the SMT solver accordingly. The former is generally not guaranteed. The formula φ might also already contain matching loops by itself.

For practical purposes, the above considerations guarantee that adding a user-defined function encoded with a fuel encoding does not introduce a matching loop on its own.

A more rigorous termination proof based on the operational small-step semantics for E-matching developed by Ge et al. in [23] was outside the scope of this thesis. The core ideas would be the same as in the above argument. The general limitation that we can only make statements about quantifiers that were created by the encodings is expected and remains.

3.3 Enabling Unbounded Computations

At the start of the chapter, we motivated that limiting the number of unfoldings is reasonable and the completeness impact regarding program proofs is acceptable. But in one situation, limited functions have a clear drawback. Trying to compute the actual value of a (now limited) function often fails, since it usually requires more unfoldings than admitted by the encoding. We saw in Example 3.31 that the solver is unable to prove `fac(1) == 1` with the fuel encodings and a maximal fuel of $mf = 1$. This limitation is not easily fixable by increasing the maximum available fuel. On the one hand, for most recursive functions and a fixed number of instantiations, there always exist an argument that requires more instantiations. So the fix would never be complete. On the other hand, using a large fuel value is contrary to the initial idea of guiding the solver during the proof search. The fuel encodings effectively degenerate to the Default encoding if the maximal fuel is too large. The possibility of increasing the maximum fuel value iteratively is briefly discussed in Section 6.5. In this section, we explore an alternative solution that allows for unbounded unfoldings in certain "safe" cases.

3.3.1 Literal Terms

One solution to this problem is to allow unfolding of the limited function without consuming fuel if all arguments are known to be literal. Intuitively, literal terms are terms that have a known value, such as literals like 0 and 1 and operations on them like 1 + 3. When a limited function is applied to only literal terms, it can be safely allowed to be unfolded an unbounded number of times without running into an endless matching loop. This is the case since all the arguments are known values, such that evaluating the function eventually terminates with a concrete value (assuming of course that the function terminates). This solution was presented by Amin et al. together with the Variable fuel encoding in [2].

To our knowledge, there exists no formal definition of literal values/terms in this context. Based on the idea that a literal term always has the same value, regardless of the interpretation, we propose the following definition:

> Definition 3.35. Literal term
> Let Σ be a signature, **AX** a set of Σ-sentences (axioms), and $\mathcal{T}$ a Σ-theory. A Σ-term φ is called *literal* iff
>
> $$|\{[\![\varphi]\!]^{\mathfrak{I}} \mid \Sigma\text{-interpretation } \mathfrak{I} = \langle \mathfrak{A}, \mathfrak{v} \rangle \text{ with } \mathfrak{A} \in \mathcal{T} \text{ and } \mathfrak{A} \models \mathbf{AX}\}| \leq 1$$

We discuss problems with the definition later in Section 5.4. By this definition, interpreted symbols from theories such as NRA or NIA are literal by the fact that these theories only have a single model, i.e. `1`: **Int** is always the number $1 \in \mathbb{Z}$ and `+`: **Real Real Real** is always standard addition on $\mathbb{R}$. The additional set of axioms is required to fix the meaning of uninterpreted functions from EUF. This includes user-defined functions. The axioms from the encoding are part of the set **AX**. Therefore, the term `fac(2)` is also literal, since the possible interpretation of `fac`: **UInt UInt** is uniquely determined by the definitional axiom in **AX**. This definition also requires that the recursive definition given in **AX** for an uninterpreted function f is terminating when an application $f(\bar{d})$ is considered literal. If the definition of f does not terminate for the input $\bar{d}$, then f is not uniquely defined for $\bar{d}$ and thus not all possible interpretations assign the same value to $f(\bar{d})$.

We use "literal" to denote this concept to differentiate it from the concept of constants. Take the term `probMessageLost()` from the program in Figure 1.1. It is a constant in the sense that, given an interpretation, it

always has the same value, regardless in which position it is evaluated. But it is not literal, since it can have any value from the interval $[0, 1]$.

Which terms can be considered literal is statically approximated by the verifier and then passed to the SMT solver. Since this information must be available during E-matching, this is done through `Lit`-marker functions. A `Lit`-marker is an identity function and marks its argument as literal. For example, `Lit(fac(2))` is logically equivalent to `fac(2)` and additionally communicates that `fac(2)` is literal. The `Lit`-marker can be used in the trigger of quantifiers such that a computation axiom can only be instantiated if all function arguments are known to be literal. The computational axiom for the `fac` function using the Variable fuel encoding is shown in Figure 3.36.

```
1  axiom fac_comp forall fuel: Fuel, n: UInt @trigger(fac(fuel, Lit(n))).
2    fac(fuel, Lit(n)) == ite(n == 0, 1, n * fac(fuel, Lit(n - 1)))
```

Figure 3.36: Computation axiom for `fac` from Figure 3.1. The `Lit`-markers are reduced to the minimum necessary.

It is very similar to the (vf-def-axiom) with the key difference that it does not require a non-zero fuel value in the trigger and the fuel is not decremented in the body. Additionally, the `Lit`-marker in the trigger ensure that it can only be used if the arguments of `fac` are literal. The `Lit`-marker are also propagated through the body, to enable nested computation. Note that `n - 1` is considered literal since the parameter `n` is ensured to be literal by the trigger.

3.3.2 Fuel Encodings with Computation

Both the Variable fuel encoding and Fixed fuel encoding encoding can be extended with computation. We use the function $tagLit_{\bar{d}\colon \bar{D}}(\varphi)$ to wrap all literal sub terms of φ with `Lit`-marker. For that, all axioms of the program are collected in **AX** and the variables $\bar{d}\colon \bar{D}$ are also assumed to be literal. The definition of literal terms (Definition 3.35) does not lend itself to an implementation. The heuristic used by Caesar for determining literal terms is discussed in the implementation chapter (Section 5.4). For the sake of readability and consistency with the implementation, not all literal terms are wrapped in the following examples, only a necessary subset.

Definition 3.37. Variable fuel encoding with computation

The variable fuel encoding with computation is an extension of the Variable fuel encoding and defined by the encoding functions

$$\mathcal{E}^{vfc}_{\text{sig}}(\Sigma, g\colon \bar{D}\, R) = \left\langle \Sigma^{\text{S}}_{vf},\quad \Sigma^{\text{F}}_{vf} \dot\cup \{\,\texttt{Lit}\,\},\qquad \Sigma^{\text{FS}}_{vf} \dot\cup \left\{\, \texttt{Lit}\colon \sigma\, \sigma \mid \sigma \in \Sigma^{\text{S}} \,\right\} \right\rangle$$
$$\mathcal{E}^{vfc}_{\text{term}}(\varphi, \text{def}_g) = \delta_{\texttt{Lit}} \wedge \chi^{vfc}_g \wedge \delta^{vf}_g \wedge \sigma^{vf}_g \wedge tagLit(\varphi[g(\bar{d'}\colon \bar{D}) \mapsto g_{\texttt{fuel}}(mf, \bar{d'})])$$

with the computation axiom

$$\chi^{vfc}_g \quad \coloneqq \quad \forall \texttt{fuel}\colon \texttt{Fuel}, \bar{d}\colon \bar{D}\ \{g_{\texttt{fuel}}(\texttt{fuel}, \texttt{Lit}(\bar{d}))\}. \qquad \text{(vfc-comp-axiom)}$$
$$g_{\texttt{fuel}}(\texttt{fuel}, \texttt{Lit}(\bar{d})) \approx tagLit_{\bar{d}\colon \bar{D}}(\text{def}_g[g(\bar{d'}\colon \bar{D}) \mapsto g_{\texttt{fuel}}(\texttt{fuel}, \bar{d'})])$$

and the definitional axioms for the `Lit`-markers

$$\delta_{\texttt{Lit}} \quad \coloneqq \quad \bigwedge_{\sigma \in \Sigma^{\text{S}}} \forall x\colon \sigma\ \{\texttt{Lit}(x)\}.\, \texttt{Lit}(x) \approx x$$

The signature of the Variable fuel encoding is extended with a `Lit`-marker function for each sort, a computation axiom is added to the final sentence, and all the literal sub-terms in the original formula φ are tagged with `Lit`-markers. Additionally, all `Lit`-makers are fixed to be identity functions by $\delta_{\texttt{Lit}}$. The quires all arguments to be literal and does not decrease the fuel value. In the body, the arguments of the function are assumed to be literal and marked accordingly.

The same happens when using the Fixed fuel encoding as a base. Here, an individual computation axiom is added for each of the individual functions g_i.

Definition 3.38. Fixed fuel encoding with computation

The fixed fuel encoding with computation is an extension of the Fixed fuel encoding and defined by the encoding functions:

$$\mathcal{E}^{ffc}_{\text{sig}}(\Sigma, g\colon \bar{D}\, R) = \left\langle \Sigma^{\text{S}}_{ff},\quad \Sigma^{\text{F}}_{ff} \dot\cup \{\, \texttt{Lit} \,\},\quad \Sigma^{\text{FS}}_{ff} \dot\cup \left\{\, \texttt{Lit}\colon \sigma\, \sigma \mid \sigma \in \Sigma^{\text{S}} \,\right\} \right\rangle$$
$$\mathcal{E}^{ffc}_{\text{term}}(\varphi, \text{def}_g) = \delta_{\texttt{Lit}} \wedge \chi^{ffc}_{g_0} \wedge \cdots \wedge \chi^{ffc}_{g_{mf}} \wedge \delta^{ff}_{g_1} \wedge \cdots \wedge \delta^{ff}_{g_{mf}} \wedge \sigma^{ff}_{g_1} \wedge \cdots \wedge \sigma^{ff}_{g_{mf}} \wedge tagLit(\varphi[g(\bar{d'}\colon \bar{D}) \mapsto g_{mf}(\bar{d'})])$$

with the computation axioms

$$\chi^{ffc}_{g_i} \quad \coloneqq \quad \forall \bar{d}\colon \bar{D}\ \{g_i(\texttt{Lit}(\bar{d}))\}.\ g_i(\texttt{Lit}(\bar{d})) \approx tagLit_{\bar{d}\colon \bar{D}}(\text{def}_g[g(\bar{d'}\colon \bar{D}) \mapsto g_i(\bar{d'})]) \qquad \text{for } i = 0, \ldots, mf \quad \text{(ffc-comp-axiom)}$$

and the same definitional axioms for the `Lit`-markers as above.

Example 3.39. Variable fuel encoding with computation

Encoding the factorial function from Figure 3.1 with the Variable fuel encoding with computation we additionally get the computation axiom for `fac`, shown in Figure 3.36. This lets us prove $\texttt{fac}(2) \approx 2$, even with a maximal fuel of $mf = 1$. The formula is negated and transformed by the encoding to

$$\texttt{Lit}(\texttt{fac}(S(Z), \texttt{Lit}(2)) \not\approx 2).$$

As before, we show that the negated formula is unsatisfiable under the axioms using E-matching. The initial set of ground terms is

$$G_1 = \{\, \texttt{Lit}(\texttt{fac}(S(Z), \texttt{Lit}(2)) \not\approx 2) \,\}.$$

Repeatedly instantiating the computation axiom we get the following sets of ground terms: [a]

$$G_2 = G_1 \cup \{\, \texttt{fac}(S(Z), \texttt{Lit}(2)) \approx 2 * \texttt{fac}(S(Z), \texttt{Lit}(1)) \,\}$$
$$G_3 = G_2 \cup \{\, \texttt{fac}(S(Z), \texttt{Lit}(1)) \approx 1 * \texttt{fac}(S(Z), \texttt{Lit}(0)) \,\}$$
$$G_4 = G_3 \cup \{\, \texttt{fac}(S(Z), \texttt{Lit}(0)) \approx 1 \,\}$$

Following all the equalities, we can conclude that

$$\texttt{fac}(S(Z), \texttt{Lit}(2)) \approx 2 * 1 * 1$$

which simplifies to the contradiction

$$\texttt{fac}(S(Z), \texttt{Lit}(2)) \approx 2.$$

Thus, $\texttt{Lit}(\texttt{fac}(S(Z), \texttt{Lit}(2)) \not\approx 2)$ is unsatisfiable and $\texttt{fac}(2) \approx 2$ must hold.

Notice that the application of `fac` produced by the computation axiom had their argument again marked as literal, allowing for repeated instantiation of the computation axiom until the base case was reached.

[a] To keep the example short, we apply some simplifications immediately to the instantiated terms.

3.3.3 Soundness and Termination

We briefly argue (without proving) that the fuel encodings with computation are also sound and do not impose any new non-termination.

That the computation encodings are logically equivalent to the other encodings can be seen by the fact that the computation axioms do not add any new constraints. After the `Lit`-markers are removed (since they are identity functions), the remaining axioms are precisely the original (def-axiom) but for the fuelled functions ($g_{\texttt{fuel}}/g_0, \ldots, g_{vf}$ respectively). Thus, any model for the Fixed fuel encoding/Variable fuel encoding is also a model for the Fixed fuel encoding with computation/Variable fuel encoding with computation (adding the `Lit`-marker as identity functions). The reverse also holds.

Regarding termination, the story is more complicated. The computational axiom theoretically allows an unbounded number of instantiations when the arguments are literal. Here, it is important to only add the com-

putation axiom for terminating functions. Then the whole application is literal, and these instantiations must eventually result in a concrete value. A formal proof is complicated by the facts that there is no formal definition of literal terms that completely captures the intuitive idea (we discuss shortcomings of our definition related to uninterpreted sorts in Section 5.4) and that termination does not only rely on E-matching but also on theory solving. The latter is theoretically supported by the formal E-matching semantics from [23], but was outside the scope of this thesis.

4. Counterexamples

The fuel encodings discussed in the previous chapter require that only E-matching is enabled for quantifier instantiation to have their intended effect. E-matching can only be used to create a contradiction, not a model satisfying the quantifier. This suffices when we are only interested in proving that a program is correct. Remember that the verification condition is negated. Therefore, unsatisfiability means that there exists no counterexample, i.e. the verification condition is valid. But in cases where the program is wrong, it is desirable to get an actual concrete counterexample to facilitate debugging, not just unknown. Counterexamples are also important when it comes to IDE integration. They are required to pinpoint the user to the relevant assertions upon verification failure. Otherwise, a localisation of the error is not possible and the user can only be told "could not verify this procedure" without being able to provide a reason. Of course, due to the undecidability of the problem, one cannot expect to always get a counterexample or successfully verify the program, but must expect unknown as an answer as well.

We start by examining what other verifiers currently do: using potentially unsound models that are produced during E-matching. Afterwards, we explore how the Fixed fuel encoding can be modified to perform a bounded search for a counterexample.

4.1 Using unknown-models

When only relying on E-matching for quantifier reasoning, the SMT solver can no longer return `sat` in the presence of quantifier. This is the case since the solver cannot prove that the ground term model also satisfies the quantifier. Thus, the solver can only return unknown after it failed to prove `unsat`.

The interface defined by the SMT-LIB to obtain a model is as follows: After adding the required definitions and formulae to the solver, it is instructed with the `check-sat` command (or similar) to perform the SAT check. Upon finishing, the command returns with the result of the SAT check (`sat`/`unsat`/unknown). If the result was `sat`, the `get-model` command can then be used to obtain the satisfying model. Interestingly, if the SAT check resulted in unknown the solver will sometimes also respond with a model when queried with `get-model`. We call such models, that are produced after an unknown response, unknown-models.

At least Boogie/Dafny[1] and Viper [41] use unknown-models as a basis for generating counterexamples. The SMT solver Z3 (primarily used by Boogie, Dafny, Viper) has a dedicated option (`smt.candidate_models`) that forces the creation of a model even when quantifier or theory reasoning is incomplete [17].[2]

It is unclear what, if any, guarantees are given by unknown-models. One theory is that an unknown-model satisfies all the ground terms at the time when E-matching cannot find a new instantiation, returning unknown (Algorithm 1 line 8), i.e. the model that was previously constructed in the same loop iteration to show that the ground terms are not unsatisfiable (line 3). This would mean that an unknown-model is only guaranteed to satisfy the instantiations of quantifiers that were made by E-matching and generally not the quantifiers themselves. Validating this theory or investigating generally unknown-models was outside the scope of this thesis and remains future work.

Dafny always requests a model upon verification failure and uses it for error localization. Additionally, a counterexample can be requested via the command line interface. Then Dafny uses the same model to construct a counterexample and prints it out. It issues a warning that the counterexample may be inconsistent or invalid.[3]

[1] https://github.com/boogie-org/boogie/issues/1008

[2] The option was added in response to https://github.com/Z3Prover/z3/issues/4924 where a Boogie/Dafny developer mentioned that newer versions of Z3 sometimes do not produce a model after responding with unknown.

[3] See http://dafny.org/dafny/DafnyRef/DafnyRef#sec-counterexamples for more clarification.

```
1  function exp(b: real, n: nat): real requires n >= 0 {
2    if n == 0 then 1.0 else b * exp(b, n - 1)
3  }
4
5  method badExp(b: real, n: nat) returns (res: real)
6    requires n >= 0
7    ensures res == exp(b, n)
8  {
9    if n == 0 {
10     res := 1.0;
11   } else {
12     var temp := badExp(b, n - 1);
13     res := b + temp; // <- + instead of *
14   }
15 }
```

Figure 4.1: Dafny program that recursively defines the exponential function and then wrongly implements it with addition instead of multiplication.

Example 4.2. Wrong Dafny counterexample

This warning is justified, as we will show for the Dafny program in Figure 4.1. The method `badExp` wrongly implements the exponential function. Thus, we expect verification to fail and to get a counterexample. Running the program through the Dafny verifier version 4.9.0.0 with the `-extractCounterexample` option results in the error that the post-condition could not be proven and the following counterexample:

```
exp.dfy(8,0): initial state:
assume 0.0 == b && 8101 == n;
exp.dfy(12,29):
assume 0.0 == b && 0.0 == temp && 8101 == n;
exp.dfy(13,19):
assume 0.0 == b && 0.0 == res && 0.0 == temp && 8101 == n;
```

Dafny prints the counterexample as a series of assumptions that can be inserted at the specified program locations to construct the situation where the failing assertion might be violated. The given counterexample is wrong. It says that for `b = 0` and `n = 8101` the method violates its post-condition. The recursive call `badExp(0, 8101 - 1)` evaluates (by specification) to $\texttt{exp}(0, 8100) = 0$. Therefore, the result variable `temp` is 0. The output variable `res` is then assigned $0 + 0 = 0$, but that actually means that the method satisfies the post-condition for this input pair ($\texttt{exp}(0, 8101) = 0$).

The model for uninterpreted functions that are part of the counterexample are not shown by Dafny. We can gain access to them by saving the prover log with the `-proverLog:<file>` option and running it through Z3 ourselves. The model printed by Z3 for `exp` (called `exp'` in the following) is

$$\begin{array}{lll}
\texttt{exp}'(S(S(Z())), 0, 8101) = -1 & \texttt{exp}'(S(Z()), 0, 8101) = -1 & \texttt{exp}'(Z(), 0, 8101) = -1 \\
\texttt{exp}'(S(S(Z())), 0, 8100) = 0 & \texttt{exp}'(S(Z()), 0, 8100) = 0 & \texttt{exp}'(Z(), 0, 8100) = 0 \\
 & \texttt{exp}'(S(Z()), 0, 8199) = 0 & \texttt{exp}'(Z(), 0, 8199) = 0 \\
\textit{otherwise: } \texttt{exp}'(_, _, _) = 0 & & \texttt{exp}'(Z(), 0, 8198) = 0
\end{array}$$

First, notice that Dafny uses the Variable fuel encoding for user-defined functions. Hence, we get a third parameter containing the fuel value. Second, the given model `exp'` is not an exponential function. This explains why Dafny claimed that the post-condition is violated for `b = 0` and `n = 8101`. Using this model, the expected return value of `badExp` is $\texttt{exp}'(0, 8101) = -1$ not 0.

We do not have evidence that suggests that the error localisation based on unknown-models is incorrect, which does happen by default and does not issue a warning.

Even though the use of unknown-models is generally not sound, it appears to be pretty successful in practice. We believe that the reason is threefold.

1. They are quick to produce, since they come from the verification query. Here, there has often been put

extensive work into the encoding striving for fast results and termination (e.g. limited functions).

2. They are always available when the solver terminates. Which comes back to the first point, that there has been done work in this direction anyway.
3. In practice, the unknown-models seem to be good enough for identifying problematic assertions and therefore localising verification errors.

So upon termination, there are then two possible results. Either the program (provably) verified or a (potentially wrong) counterexamples is given.

We believe that wrong counterexamples can be confusing and that they should have explainable guarantees. That is why we look into another approaches next.

4.2 Using a Fixed Depth Encoding

E-matching fundamentally cannot be used to construct models that satisfy quantifiers, let alone recursive functions. MBQI can construct models that provably satisfy quantifiers, but there are problems with constructing models for recursive function in practice. If the recursive function has an infinite domain, the solver (in our case Z3) is unable to construct a model with MBQI. Such a model would generally be infinite and require an inductive argument to prove its correctness, which most solvers do not support. This is usually also the case if the recursive function is not defined by an uninterpreted function + axiom but with `define-funs-rec`, a command introduced by the SMT-LIB standard to define a set of mutually recursive functions. Here, Z3 also resorts to incrementally unfolding the definition without any inductive reasoning.[4] We found that in some simple cases, Z3 can find counterexamples involving recursive functions if the recursive function was declared using `define-funs-rec`.

If recursive functions are the problem, we can possibly circumvent the problem by encoding them in a non-recursive manner. Essentially weakening the constraints to make it easier to construct a model. This is almost what happens when the function is transformed to a limited function. The definitional axioms are not recursive by themselves.[5] But we have seen in Theorem 3.12 that the encodings are logically equivalent to the recursive one. This means we cannot get spurious counterexamples (Theorem 3.30) but also that a model (counterexample) must satisfy the full recursive definition of the recursive function, resulting in no counterexamples in practise.

The synonym axioms are the key here. They bundle the multiple introduced function symbols such that the whole construct becomes recursive again. We have also seen that if the synonym axioms are omitted, we can get spurious counterexamples (Example 3.10). When encoding a recursive function $g\colon \bar{D}\,R$ with the Fixed fuel encoding but omitting the synonym axioms (ff-syn-axiom), then in a model each function g_k is guaranteed to be correct for argument tuples $\bar{d}$ that require less than k unfoldings of g to compute the value $g(\bar{d})$. So g_0 can be arbitrary, since it is unconstrained, g_1 is correct for arguments that require no recursive call, and g_2 is correct for arguments that need zero or one recursive call to compute their result. In total, such a model is correct for all arguments that require less than the maximum fuel (mf) unfoldings. This gives us at least some correctness guarantees for the resulting models. We call the Fixed fuel encoding with only the definitional axioms and without the synonym axioms the *fixed depth encoding*.

In summary, when dealing with recursive uninterpreted functions, with the fuel encodings we have encodings that can only yield `unsat` or `unknown`. If we omit the synonym axioms, then the function is no longer recursive, and we could get counterexamples. These observations lead to the following approach: Run two separate queries in parallel against the SAT solver. One optimized for showing `unsat` (proving correctness) and one for showing `sat` (constructing a counterexample, short cex).

verification query. Using a complete fuel encoding as previously described (definitional axiom, synonym axiom, possibly including computation axiom and lit-wrapping), enabling only E-matching. This query tries to prove the correctness of the program.

cex query. Using the fixed depth encoding (Fixed fuel encoding with only the definitional axioms) and only enabling MBQI. This query tries to find a counterexample (cex) for the program.

When the two queries return conflicting results, the verification query has priority. A complete overview is given in Table 4.1.

[4] https://microsoft.github.io/z3guide/docs/logic/Recursive%20Functions/

[5] Here, $g_{\texttt{fuel}}$ with a different fuel values is considered to be a different function.

cex query \ verification query	sat	unsat	unknown
sat	Use cex from verification query	Program verified	Use cex from cex query
unsat	——— " ———	——— " ———	unknown
unknown	——— " ———	——— " ———	unknown

Table 4.1: Which result to use, depending on the results of the two queries

A big problem with this approach is that the constraints are too weak. The resulting counterexamples often have only the minimal required structure. This means the function model is correct for arguments that require at most *mf* instantiations, and wrong for arguments that require more than *mf* instantiations. The counterexample then consists of a state that needs more than *mf* instantiations, see Example 3.10 or the following.

```
1  domain Math {
2    func exp0(base: Real, exponent: UInt): Real
3    func exp1(base: Real, exponent: UInt): Real
4    func exp2(base: Real, exponent: UInt): Real
5
6    axiom exp_def1 forall b: Real, e: UInt @trigger(exp1(b, e)).
7      exp1(b, e) == ite(e == 0, 1, b * exp0(b, e - 1))
8
9    axiom exp_def2 forall b: Real, e: UInt @trigger(exp2(b, e)).
10     exp2(b, e) == ite(e == 0, 1, b * exp1(b, e - 1))
11 }
12
13 proc badExp(base: Real, exponent: UInt) -> (res: Real)
14   post ?(res == exp2(base, exponent))
15 {
16   if exponent == 0 {
17     res = 1
18   } else {
19     var temp: Real = badExp(base, exponent - 1)
20     res = base + temp // <-- + instead of *
21   }
22 }
```

Figure 4.3: HeyVL program where the exponential is function encoded with the fixed depth encoding ($mf = 2$) and that wrongly implements the exponential function with addition instead of multiplication.

Example 4.4

Consider the program in Figure 4.3, the HeyVL counterpart to the earlier Dafny program in Figure 4.1. It has the fixed depth encoding applied, and the procedure `badExp` wrongly implements the exponential function. The counterexample produced for the program is:

$$\texttt{exp2}(1,3) = 0 \quad \texttt{exp2}(1,2) = 0 \qquad \textit{otherwise} \quad \texttt{exp2}(b,e) = \begin{cases} 1, & \text{if } e = 0 \\ b \cdot \texttt{exp1}(b, e-1), & \text{else} \end{cases}$$

$$\texttt{exp1}(1,2) = 0 \quad \texttt{exp1}(1,1) = 0 \qquad \textit{otherwise} \quad \texttt{exp1}(b,e) = \begin{cases} 1, & \text{if } e = 0 \\ b \cdot \texttt{exp0}(b, e-1), & \text{else} \end{cases}$$

$$\texttt{exp0}(\_,\_) = 0 \qquad \texttt{base} = 1 \qquad \texttt{exponent} = 3 \qquad \texttt{temp} = 0 \qquad \texttt{res} = 1$$

Since `exp2` is unconstrained for the exponents 3 and 2 (their computation requires more than two unfoldings), the solver can assign arbitrary values to $\texttt{exp2}(1,3)$ and $\texttt{exp2}(1,2)$. The model for the exponential function bears little resemblance to the actual exponential function. A "counterexample" is then created by choosing values such that the procedure implementation does not produce the same values. The task for the solver degenerates from finding a state where the procedure violates the specification to finding a new specification (and then trivially a state) that the procedure does not satisfy.

The resulting counterexamples are usually wrong and offer little insight for debugging a program. Perhaps there are other use cases where they are useful. We will improve this approach in Section 4.3 by restricting the counterexample to the defined parts of the function.

4.2.1 Why not to use the Variable fuel encoding?

First, we want to address another question: Why is the Fixed fuel encoding used as a basis for the limited depth encoding and not the Variable fuel encoding? This has the practical reason that the solver is unable to find a model for recursive functions encoded with Variable fuel encoding even without the synonym axiom. To understand that, we have to examine how the uninterpreted `Fuel` sort can be interpreted by a model. The sort must contain at least one element — the result of $Z()$. Let us name the elements of the `Fuel` sort $\texttt{vfuel}_n$, with $Z() = \texttt{vfuel}_0$.

```
1  domain Fuel {
2    func Z(): Fuel
3    func S(fuel: Fuel): Fuel
4  }
5
6  domain Math {
7    func exp(fuel: Fuel, base: Real, exponent: UInt): Real
8
9    axiom exp_def forall fu: Fuel, b: Real, e: UInt @trigger(exp(S(fu), b, e)).
10     exp(S(fu), b, e) == ite(e == 0, 1, b * exp(fu, b, e - 1))
11 }
```

Figure 4.5: Exponential function encoded with Variable fuel encoding but without the (vf-syn-axiom). Model construction for this encoding fails.

If $\texttt{vfuel}_0$ is the only element, i.e. $\texttt{Fuel}^{\mathfrak{A}} = \{ \texttt{vfuel}_0 \}$, then $S(\texttt{vfuel}_0) = \texttt{vfuel}_0$ must hold. This fact turns the (vf-def-axiom) into the original recursive (def-axiom). For example, the definitional axiom of the exponential function in Figure 4.5 becomes:

```
axiom exp_def forall b: Real, e: UInt @trigger(exp(vfuel₀, b, e)).
  exp(vfuel₀, b, e) == ite(e == 0, 1, b * exp(vfuel₀, b, e - 1))
```

The axiom is again recursive. Therefore, it is clear that MBQI fails to construct a model for the same reasons as previously stated.

If the `Fuel` sort consists of more than one element and is finite, i.e. $\texttt{Fuel}^{\mathfrak{A}} = \{ \texttt{vfuel}_0, \texttt{vfuel}_1, \ldots, \texttt{vfuel}_k \}$, then there must exist a successor loop (named based on the intuition that S: `Fuel` `Fuel` is the successor function). Meaning that there exists a $\texttt{vfuel} \in \texttt{Fuel}^{\mathfrak{A}}$ and a number of applications $n \in \mathbb{N}_0$ such that the n-fold application of S to `vfuel` is again `vfuel`, i.e. $S^n(\texttt{vfuel}) = \texttt{vfuel}$. Otherwise, all elements produced by $S^n(\texttt{vfuel})$ for $n \in \mathbb{N}_0$ must be pairwise distinct, which contradicts the finiteness of $\texttt{Fuel}^{\mathfrak{A}}$. The existence of the loop means that the definitional axiom is again recursive, and that model creation generally fails.

> Example 4.6. Three fuel values
>
> For 3 distinct fuel values with
>
> $$S(\texttt{vfuel}_0) = \texttt{vfuel}_1 \qquad S(\texttt{vfuel}_1) = \texttt{vfuel}_2 \qquad S(\texttt{vfuel}_2) = \texttt{vfuel}_1$$
>
> we get three axioms for the encoding in Figure 4.5
>
> ```
> axiom exp_def1 forall b: Real, e: UInt @trigger(exp(vfuel₁, b, e)).
>   exp(vfuel₁, b, e) == ite(e == 0, 1, b * exp(vfuel₀, b, e - 1))
> axiom exp_def2 forall b: Real, e: UInt @trigger(exp(vfuel₂, b, e)).
>   exp(vfuel₂, b, e) == ite(e == 0, 1, b * exp(vfuel₁, b, e - 1))
> axiom exp_def3 forall b: Real, e: UInt @trigger(exp(vfuel₁, b, e)).
>   exp(vfuel₁, b, e) == ite(e == 0, 1, b * exp(vfuel₂, b, e - 1))
> ```
>
> where the last two axioms are mutually recursive.

If the `Fuel` sort consists of an infinite number of elements, i.e. $\texttt{Fuel}^{\mathfrak{A}} = \{ \texttt{vfuel}_0, \texttt{vfuel}_1, \texttt{vfuel}_2, \ldots, \}$, then a successor loop may not exist. In such a case, the subset $\{ (S^n(Z))^{\mathfrak{A}} \mid n \in \mathbb{N} \}$ of $\texttt{Fuel}^{\mathfrak{A}}$ is isomorph to the natural numbers. The definitional axiom is then not recursive, but the existence of arbitrarily large fuel values means that there must exist versions of the encoded function that are correct to an arbitrary large depth. Apart from the infiniteness of the `Fuel` sort, this makes a model unconstruable.

4.3 Using a Fixed Depth Encoding and Bounded Inputs

The problem that we previously encountered in Example 4.4 was due to the fact that the counterexamples produced by the fixed depth encoding used parts of the function that were not defined by the encoding. If we restrict the counterexamples to only use parts of the function that are defined by encoding, we would guarantee correct counterexamples. Remember that this means restricting the argument values to those that require at most mf instantiations to compute their value.

```
1  // fixed depth encoding of exp - omitted ...
2
3  proc badExp(base: Real, exponent: UInt) -> (res: Real)
4    pre ?(exponent < 2) // <-- argument restricted to defined fragment of exp2
5    post ?(res == exp2(base, exponent))
6  {
7    if exponent == 0 {
8      res = 1
9    } else {
10     var temp: Real = badExp(base, exponent - 1)
11     res = base + temp // <-- + instead of *
12   }
13 }
```

Figure 4.7: Adjusted version of the program from Figure 4.3 with the input bounded such that `exp2` is defined for it.

Example 4.8

In the case of the exponential function, the number of required unfoldings directly corresponds to the exponent plus one. Thus, if we restrict the input `exponent` to be less than 2 (for $mf = 2$) we do not use undefined parts of `exp2`. The resulting program is shown in Figure 4.7. The found counterexample is:

$$\texttt{exp2}(0,1) = 0 \quad \texttt{exp2}(0,0) = 1 \qquad \textit{otherwise} \quad \texttt{exp2}(b,e) = \begin{cases} 1, & \text{if } e = 0 \\ b \cdot \texttt{exp1}(b, e-1), & \text{else} \end{cases}$$

$$\texttt{exp1}(0,0) = 1 \qquad\qquad\qquad \textit{otherwise} \quad \texttt{exp1}(b,e) = \begin{cases} 1, & \text{if } e = 0 \\ b \cdot \texttt{exp0}(b, e-1), & \text{else} \end{cases}$$

$$\texttt{exp0}(\_,\_) = 0 \qquad \texttt{base} = 0 \qquad \texttt{exponent} = 1 \qquad \texttt{temp} = 1 \qquad \texttt{res} = 1$$

Finally, `exp2` resembles an exponential function and the counterexample is correct. For the inputs `base` $= 0$ and `exponent` $= 1$ the recursive call `badExp(0, 1)` returns 1 such that the procedure terminates with `res` $= 1$. This is in violation of the specification that says that `res` should be $\texttt{exp2}(0,1) = 0$. For this simple procedure, Z3 also finds a counterexample when declaring `exp` with `define-funs-rec`.

Some caution is required when interpreting the results obtained by the above encoding. A resulting counterexample (solver returning `sat`) is also a counterexample for the original program. But if it verifies (solver returning `unsat`) that does not mean that the original program is correct. Since the input was bounded, it only means that the program is correct for this restricted set of inputs and could be wrong for other inputs. For example, if the bound in Figure 4.7 is tightened to `exponent < 1`, then the program spuriously verifies. A closer inspection reveals that `badExp` is actually correct if `exponent` is 0, but otherwise wrong. Only `unsat` from the verification query can be used to conclude that the program is correct. If no conclusive result is obtained, the depth (mf)

```
domain Constants {
  func probMessageLost(): UReal
  axiom messageLostProb probMessageLost() <= 1
}

domain Arith {
  func exp0(base: UReal, exponent: UInt): UReal
  func exp1(base: UReal, exponent: UInt): UReal
  func exp2(base: UReal, exponent: UInt): UReal

  axiom exp_def1 forall b: UReal, e: UInt @trigger(exp1(b, e)).
    exp1(b, e) == ite(e == 0, 1, b * exp0(b, e - 1))
  axiom exp_def2 forall b: UReal, e: UInt @trigger(exp2(b, e)).
    exp2(b, e) == ite(e == 0, 1, b * exp1(b, e - 1))

  axiom exp_bounded forall b: UReal, e: UInt. (b <= 1) ==> (exp2(b, e) <= 1)
}

proc arp(triesRemaining: UInt) -> (success: Bool)
  pre ?(triesRemaining < 2) // <-- restricted to defined fragment of exp2
  pre exp2(probMessageLost(), triesRemaining) // <-- not converse probability
  post [success]
{
  if triesRemaining == 0 {
    success = false
  } else {
    var messageLost: Bool = flip(probMessageLost())
    if messageLost {
      success = arp(triesRemaining - 1)
    } else {
      success = true
    }
  }
}
```

Figure 4.9: Version of the program from Figure 1.1 with the wrong specification from Figure Figure 1.2. The exponential function is encoded with the Fixed fuel encoding ($mf = 2$). The exponent (`triesRemaining`) is bounded to be less than 2, such that a counterexample only uses defined parts of `exp2`.

of the fixed depth encoding could be iteratively increased together with corresponding bounds to search for counterexamples that require more unfoldings.

How it can be ensured, in general, that only defined parts of the encoding are used by the counterexample remains an open question.

4.3.1 Case Study: Counterexample for `arp`

We close out the section with a small case study on the motivating example from Figure 1.1 with the wrong specification from Figure 1.2 that reveals some more challenges. The complete resulting program is shown in Figure 4.9. It uses the fixed depth encoding for `exp` and restricts the exponent to the defined fragment.

In our experiments, Z3 was unable to create a model (counterexample) for the program in Figure 4.9 and timed out. Moreover, simplifying the problem by fixing `probMessageLost()` to a concrete probability did not help either. Investigating the performed MBQI instantiations with SMTscope revealed that, even after fixing `probMessageLost()`, Z3 performed many instantiations of `exp`-terms with widely different bases. Only a single base is relevant for a counterexample, thus Z3 made no real progress.

Based on this observation, a slightly different encoding was created (see Figure 4.10). The `base` parameter was removed from the signature of `exp`. `probMessageLost()` remains an arbitrary probability, but is hard-coded as the base in the definitional axioms. This reduced degree of freedom (`probMessageLost()` being always the base) is enough guidance for Z3 to instantly (less than a second) find a model.

```
1  domain Constants {
2    func probMessageLost(): UReal
3    axiom messageLostProb probMessageLost() <= 1
4  }
5
6  domain Arith {
7    func exp0(exponent: UInt): UReal
8    func exp1(exponent: UInt): UReal
9    func exp2(exponent: UInt): UReal
10
11   axiom definitional1 forall e: UInt @trigger(exp1(e)).
12     exp1(e) == ite(e == 0, 1, probMessageLost() * exp0(e - 1))
13   axiom definitional2 forall e: UInt @trigger(exp2(e)).
14     exp2e == ite(e == 0, 1, probMessageLost() * exp1(e - 1))
15
16   axiom exp_bounded forall b: UReal, i: UInt. exp2(i) <= 1
17 }
18
19 proc arp(triesRemaining: UInt) -> (success: Bool)
20   pre ?(triesRemaining < 2) // <-- restricted to defined fragment of exp2
21   pre exp2(triesRemaining) // <-- not converse probability
22   post [success]
23 {
24   // body - omitted ...
25 }
```

Figure 4.10: Adjusted version of the program from Figure 4.9. The defined exponential function has a fixed base of `probMessageLost()`.

The produced counterexample for the program in Figure 4.10 is:

$$\texttt{exp2}(0) = 1 \qquad \textit{otherwise} \quad \texttt{exp2}(e) = \frac{1}{2} \cdot \texttt{exp1}(e-1)$$

$$\texttt{exp1}(0) = 1 \qquad \textit{otherwise} \quad \texttt{exp1}(e) = \frac{1}{2} \cdot \texttt{exp0}(e-1)$$

$$\texttt{exp0}(\_) = 4 \qquad \texttt{probMessageLost()} = \frac{1}{2} \qquad \texttt{triesRemaining} = 0 \qquad \texttt{success} = \texttt{false}$$

The counterexample is correct. `exp2` resembles an exponential function and the post-expectation is violated. The pre-expectation `exp2(triesRemaining)` evaluates to 1, but the post-expectation is 0 (pre is not a lower bound of the post).

The initial problems of finding a model are again a reminder that the underlying procedure is incomplete, and it is not always obvious what the underlying problem is. Tooling such as SMTscope allowed us to analyse the problem and improve the encoding. Finding a standard encoding that generally works out of the box for user-provided definitions is probably unrealistic. Here, another approach is to identify commonly required functions/constructs and provide well tuned built-in encodings/procedures. For exponentials, a dedicated counter example guided abstraction refinement (CEGAR) loop presented in [21] looks promising. We relied entirely on MBQI, a very general CEGAR loop on the level of arbitrary terms. A dedicated CEGAR loop for exponentials can exploit additional facts, e.g. monotonicity, and possibly perform better.

For this more complicated procedure, Z3 fails to find a counterexample when declaring `exp` with `define-funs-rec`. This is the case for both versions (Figure 4.9/Figure 4.10).

5. Implementation

As part of this thesis, the presented encodings were implemented in Caesar and evaluated. This chapter details their general implementation as well as the design decisions and challenges specific to working with the SMT solver Z3 [15].

5.1 General

The function encodings are applied to function declarations with an associated body. Figure 5.1 shows the syntax for this HeyVL construct. This syntax was already previously used in Figure 3.4 to define the factorial function. Function declarations without a body are continued to be directly mapped to an uninterpreted function.

```
func g(d̄:D̄): R = def_g
```

Figure 5.1: HeyVL syntax for defining a function where g is the name of the function, $\bar{d} : \bar{D}$ is a possibly empty list of typed parameters, R is the return type and def_g a HeyVL expression of type R (assuming the varaibles $\bar{d}$ have types $\bar{D}$).

The encoding of user defined-functions is the responsibility of the FOL Encoder (see Figure 2.1). It takes the generated verification condition and the declarations in the HeyVL program and produces the input for the SMT solver. The different function encodings are implemented using the strategy pattern [22] as shown in Figure 5.2.

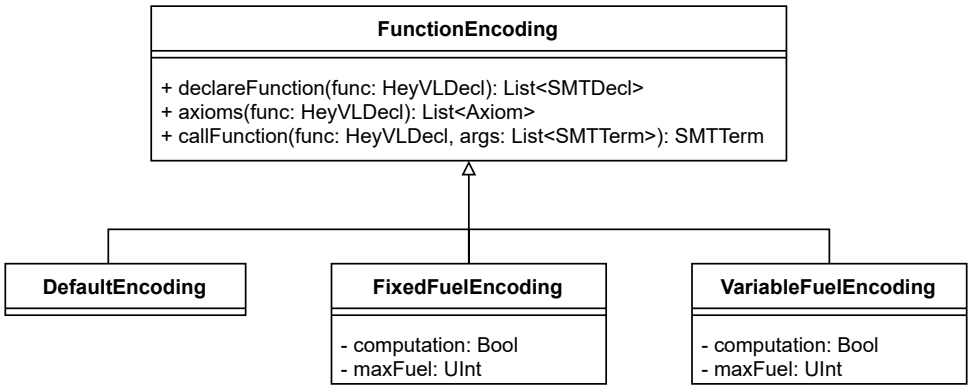


Figure 5.2: The strategy pattern is used to implement the different encodings. They each the `FunctionEncoding` interface.

The structure of the implementation is close to how the encodings were introduced in Section 3.1. The fuel encodings have additional parameters, determining the maximum fuel and if computation axioms should be generated. Each encoding must implement three operations. The first operation (`declareFunction`) generates the necessary declarations for the SMT solver. This is one function declaration in the case of the Default encoding and Variable fuel encoding and multiple functions in the case of the Fixed fuel encoding. It is the implementation of the $\mathcal{E}_{\text{sig}}$ function from the definitions, which performs the necessary signature modifications. The second operation (`axioms`) generates all the required (and applicable) axioms for a function (definitional, synonym and computational if enabled). The third operation (`callFunction`) generates the appropriated call to the function with the given arguments. Together, the second and third operations implement the $\mathcal{E}_{\text{term}}$ function from the definitions, which adds the axioms and substitutes in the correct function calls in the remaining formula.

5.2 Disabling MBQI

The fuel encodings presented in Chapter 3 are designed with only E-matching in mind. Thus, MBQI must be disabled. Both E-matching and MBQI are enabled in Z3 by default.

Disabling MBQI the standard way. The standard way of disabling MBQI is to first disable automatic configuration mode by setting the parameter `smt.auto_config` to false and then disabling MBQI with the parameter `smt.mbqi` [9, 17].

Problems with quantitative quantifiers. Disabling MBQI generally like that has the effect that some examples that rely on quantitative quantifiers no longer verify. In the quantitative setting, the infimum (greatest lower bound) and supremum (least upper bound) are analogues to the universal and existential quantifier. Most of the time, quantitative quantifiers can be eliminated by Caesar. In cases where they cannot be eliminated, they are currently encoded using the textbook definition, i.e. inf is the infimum of $R \subseteq \mathbb{R}^{\infty}_{\geq 0}$ iff

$$\forall r \in R.\ inf \leq r \qquad (inf \text{ is lower bound})$$
$$\forall lb \in \mathbb{R}^{\infty}_{\geq 0}.\ (\forall r \in R.\ lb \leq r) \implies lb \leq inf \qquad (inf \text{ is greater than every other lower bound})$$

and dually for the supremum. The exact reason why E-matching alone does not suffice was not determined. A possible reason could be that the encoding is too naive and that bad patterns are inferred such that E-matching is rendered useless. Improvements to the quantitative quantifier elimination and the encoding of quantitative quantifiers are important next steps.

Alternative way of disabling MBQI. With the observation that MBQI can be useful/is required for proving `unsat`, we decided not to disable it globally but only on a per-quantifier basis. Z3 has the option `smt.mbqi.id` that when specified has the effect that model-based quantifier instantiation only happens for quantifiers with an ID that starts with the specified prefix [17]. Quantifier IDs are additional metadata that can be associated with quantifiers (similar to triggers), primarily for debugging purposes. The idea is to set this option to `mbqi_` and add that as a prefix to the three quantifiers of the infimum/supremum encoding. This disables MBQI for all other quantifiers that do not have the prefix, such as those produced by the fuel encodings. However, it keeps MBQI enabled for the infimum/supremum encoding. This means that examples with quantitative quantifiers still verify.

Z3 bug. This approach faced the practical problem that the `smt.mbqi.id` did not work as the documentation suggested. Setting the option caused Z3 to frequently return `sat` together with unsound models. We opened an issue[1] with our findings and a bug fix was subsequently added by Nikolaj Bjørner. We think that MBQI previously only ignored quantifier not matching the prefix when checking if a model is valid. Thus, returning unsound models. This is similar to our approach in Section 4.2, where we removed the synonym axioms to make model construction possible. Perhaps that was the original intention of the option and the documentation was misleading.

The confusion reveals a problem when working with a large and complex tool with vast configuration options such as Z3. It is not always clear what the intended use for the option is and how they interact with each other and the different solver components. For example, it is not obvious from the documentation that only disabling `smt.mbqi` is not enough to turn off MBQI but that `smt.auto_config` must be disabled as well. We found that it is often best to just try the different options and observe how the solver behaviour changes. For less frequently used options, there exists the additional complication that one cannot be sure if the observed behaviour is intended or if the option has a bug (as with `smt.mbqi.id`).

5.3 `Lit`-marker

The literal values for which the unfolding limit is suspended are marked with `Lit`-marker. They are identity functions and relevant for E-matching, since their existence is required to instantiate the computation axiom. Hence, they need to be part of the E-graph and are declared as an uninterpreted function plus a definitional axiom stating they are equal to their argument. For illustrative purposes, the equivalent HeyVL code is shown in Figure 5.3.

[1] https://github.com/Z3Prover/z3/issues/7510

```
1  func LitInt(x: Int): Int
2  axiom lit_int_def forall x: Int @trigger(LitInt(x)). LitInt(x) == x
```

Figure 5.3: HeyVL code that demonstrates how the `Lit`-markers are encoded, here for the sort **Int**.

A different function and axiom is generated for each SMT-LIB. Some HeyVL types are mapped to the same SMT-LIB sort, but with additional constraints. For example, `UInt`-terms are translated to terms of sort **Int** with a non-negativity constraint (same for `UReal` and `Real`). This means that `UInt`-terms get marked with the same functions as `Int`-terms. The default encoding of the `EUReal` type uses two different terms. One term of sort **Bool**, which keeps track of whether the value is infinite or not, and one term of sort **Real** with non-negativity constraint, which keeps track of the numeric value. Here, each of the two terms is marked separately.

The marker functions are not implemented with `define-fun` or `define-fun-rec`, dedicated commands provided by the SMT-LIB standard [6] for defining functions, since they are subject to preprocessing in Z3. For both commands, Z3 first inlines all calls to Lit-markers. Replacing them with their argument. This is logically sound, but has the effect of eliminating all Lit-marker. Thus, no Lit-marker are present in the E-graph and the computation axiom never triggers. Interestingly, the `define-fun` command is defined to be equivalent to declaring an uninterpreted function and adding a definitional axiom [6, Section 4.2.3] like we ended up doing (see Figure 5.3). But as we have seen several times, this theoretical equivalence does not imply an equivalence in practice.

5.4 Determining Literal Terms

Our definition of literal terms (Definition 3.35) characterizes them by the fact that they only ever have a single value assigned to them, regardless of the interpretation. This definition cannot be directly implemented, since it would require checking all (potentially infinite) possible interpretations. Instead, Caesar uses a bottom up heuristic that approximates the literal terms of an expression. At a high level, it operates on the principle that interpreted constants are literal, such as 1 and 2, and that interpreted operations involving only literal terms are also literal. For example, $1 + 2$ is literal. For a user-defined function $f : D\, R$, the fact that the term $t : D$ is literal does not always imply that the term $f(t)$ is also literal. Here, the user-defined function f must be computable in a finite number of steps. If the function f was declared with a body

```
func f(x: D): R = def
```

then Caesar assumes that the definition terminates and $f(t)$ is also considered literal. Caesar does not prove that the function terminates. Automatically proving termination of recursive functions is not part of this thesis. We rely on the user giving sensible function definitions. The user can also annotate functions that they know are computable with `@computable`.

```
@computable func f(x: D): R
```

Again, this claim is not checked and the responsibility of the user to ensure that it is correct. In this second case, $f(t)$ is also considered to be literal. It is not considered literal in all other cases.

The complete algorithm that determines if the term t is literal under the assumption that the variables $\bar{x} : \bar{\sigma}$ are literal is presented in Algorithm 3. If the term t is one of the variables that are assumed to be literal, then it is literal. All other variables are not considered literal. If the term is a function application, it is first recursively checked if all the arguments are literal. If this is not the case, then the function application is not considered literal. Otherwise, if all arguments are literal, the applied function is examined next. As previously discussed, if the function is an interpreted function from a background theory or a user-defined function that was declared with a body or is annotated as computable, then the application is considered literal. In all other cases, t is not considered literal. Remember that constants are functions with zero arguments. All of their arguments are vacuously literal. Therefore, interpreted constants like $0, 1, \ldots$ are always considered literal by line 11.

> Example 5.4. The need for `@computable`
>
> Caesar currently has not special support for custom data types. But they can still be encoded with uninterpreted functions. The HeyVL program in Figure 5.5 creates a Peano number data type called `Nat` with two variants `Z()` (zero) and `S(n)` (successor). The code also declares a tester function `isZ` that checks if the variant that was used to construct the Peano number was `Z()`, and an accessor `getPrev` to obtain the previous value of

Algorithm 3: IsLit$(t, \bar{x} : \bar{\sigma})$

Input : A term $t : \sigma$ and a set of variables that are $\bar{x} : \bar{\sigma}$ assumed to be literal
Output: Either `true` if t is determined to be literal or `false` otherwise

switch *on the structure of* t **do**
  case t *is a varaible* y **do**
    if $y : \sigma$ *is in* $\bar{x} : \bar{\sigma}$ **then**
      return `true`;
    else
      return `false`;
    end
  end
  case t *is a function application* $f(\bar{t}')$ **do**
    if *for all* t' *in* $\bar{t}'$: IsLit$(t', \bar{x} : \bar{\sigma})$ **then**
      if f *is an interpreted function* **then** `/* includes interpreted constants like 0 */`
        return `true`;
      else if f *is a user-defined function with body or* `@computable` *annotation* **then**
        return `true`;
      else
        return `false`;
      end
    else
      return `false`;
    end
  end
  otherwise do `/* t is a quantifier */`
    return `false`;
  end
end

a Peano number created with `S()`. The `plus` function uses these helper functions to implements the addition of two Peano numbers.

The functions `isZ` and `getPrev` currently cannot be defined with a single expression body. They need to match on the `Nat` value to determine how it was built. HeyVL has no match expression, instead the matching must be encoded with axioms as done in the code. The two constructor functions have no definition, since their value is just themselves.

If the other functions were not marked `@computable` the generated computation axiom for `plus` would be

```
forall f: Fuel, n: Nat, m: Nat @trigger(plus(f, Lit(n), Lit(m))).
  plus(f, Lit(n), Lit(m)) == ite(isZ(Lit(n)), Lit(m),
    S(plus(f, getPrev(Lit(n)), Lit(m))))
```

which cannot be applied repeatedly since the first argument (`getPrev(Lit(n))`) of the recursive `plus` call is not marked as literal. Without an annotation, Caesar has no way of telling that `n` being literal also implies `getPrev(n)` being literal. Therefore, proving `plus(S(S(Z())), Z()) == S(S(Z()))` with a maximum fuel of $mf = 2$ is not possible. With the annotations, the computation axiom becomes

```
forall f: Fuel, n: Nat, m: Nat @trigger(plus(f, Lit(n), Lit(m))).
  plus(f, Lit(n), Lit(m)) == ite(isZ(Lit(n)), m,
    S(plus(f, Lit(getPrev(Lit(n))), Lit(m))))
```

which can be repeatedly applied. The annotations also have the effect that `S(S(Z()))` is considered literal, which is required to kick-start the computation. Note that *ite* and its arguments are now considered literal too, but are not marked due to the optimization described in Section 5.4.1.

The `@computable` annotation offers a flexible solution for user to specify which functions are computable. In the future, if HeyVL adds extra syntax for data types and the ability to match on them, then these constructs would provide the algorithm with additional information and guarantees, such that it could infer more things to be literal. This would reduce the user's responsibility in this regard.

```
domain Nat {
  @computable func Z(): Nat
  @computable func S(n: Nat): Nat

  @computable func isZ(n: Nat): Bool
  axiom isZ_Z isZ(Z()) == true
  axiom isZ_S forall n: Nat @trigger(isZ(S(n))). isZ(S(n)) == false

  @computable func getPrev(n: Nat): Nat
  axiom getPrev_def forall n: Nat @trigger(getPrev(S(n))). getPrev(S(n)) == n

  func plus(n: Nat, m: Nat): Nat = ite(isZ(n), m, S(plus(getPrev(n), m)))
}
```

Figure 5.5: Computation with custom data types requires `@computable` annotation.

Our use of `@computable` for `Z` and `S` is technically in conflict with our previously given definition of literal terms (Definition 3.35). Since the sort **Nat** is uninterpreted, there are many possible interpretations for it like $\mathbf{Nat}^{\mathfrak{A}} = \mathbb{N}_0$ or even $\mathbf{Nat}^{\mathfrak{A}} = \{ a \}$, since it is not required to be infinite. Thus `Z()`, `S(Z())` and so on, can have many possible values. $\mathbf{Nat}^{\mathfrak{A}}$ can also contain values that are not constructible by a combination of `Z` and `S` making the definitions of all the other functions partial. These hypotheticals are not relevant from a practical perspective. Only the constructible fragment of $\mathbf{Nat}^{\mathfrak{A}}$ is ever relevant, and our intuition that terms like `S(Z())` are custom literals holds. If used as an argument, they can be used to compute the concrete value of the other defined functions.

The function `getPrev` is also only defined partially for numbers built with `S`. It can be made into a total function, such that it is considered literal by Definition 3.35, by retuning a fixed value for all other numbers. The extension is not necessary for the program in Figure 5.5 since all its applications are guarded by a negative `isZ` check.

The proposed definition does not work for uninterpreted sorts, and it is currently not clear how it could be extended to include them.

5.4.1 Which Terms to `Lit`-mark

In an implementation, it is not adventitious to wrap all literal terms in `Lit`-marker. Too many `Lit`-marker can slow down the solver. They each require an instantiation and introduce a new equality that must be tracked. Looking at Algorithm 3 we see that if a term t is considered literal also all it (immediate/transitive) child terms are considered literal. Most of this literal information is not relevant and is only required during the algorithm run. In fact, the only case where the information is directly relevant is for arguments of user-defined functions. Therefore, a literal term is only marked with a `Lit`-marker if its parent is a function application of a user-defined function or if its parent itself is not literal. The latter prevents the information from getting lost completely.

In [2], Amin et al. briefly mentioned that it is important in practice to use *ite* (if-then-else) as a stopper for propagating literal information, i.e. considering *ite* itself not literal and every term containing it. We can confirm this claim, but are unsure if it is related to *ite*. It could also be that it should be avoided to mark the result of the computation axiom as literal. It is difficult to distinguish the two due to the usual structure of recursive function. They contain a top-level case distinction (*ite*) to separate the base case from the recursive case. As a result, the code produced by both rules is almost identical. Still, we decided to go with the latter as it is more localized to the area we saw problems.

In summary, in our implementation a term is wrapped in a `Lit`-marker iff

- Algorithm 3 determines that it is literal,
- and its parent is a function application of a user-defined function or the parent is not literal,
- and it is not the result of a computation axiom.

Our evaluation in Section 6.4 confirms that these optimisations are important for obtaining a working implementation.

We have no explanation as to why it is important to not mark *ite*/the result of the computation axiom as literal. We can only describe the observed behaviour if it is not done. In some runs, it causes Z3 to hang and to time

out. When examining the produced traces for such a run with SMTscope we see no matching-loops (which would have been surprising), but numerous equalities and gigantic terms filling multiple screens. For illustrative proposes, we will discuss two equalities taken from a verification run for a program working with different definitions of the factorial function. The first suspicious equality is

```
ite(n - LitInt(1) >= 0, n - LitInt(1), 0) == LitInt(1)
```

which says that `n` (from the verification condition that we do not know anything of) minus 1 but bounded at 0 is equal to `LitInt(1)`. The reason for this equality is not given. Subsequent equalities then write `LitInt(1)` in more and more elaborate ways. For example, an early one is:

```
ite(n - LitInt(1) >= 0, n - LitInt(1), 0) ==
  LitInt(ite(Factorial(S(S(Z)), LitInt(0)) == 0,
    1,
    Factorial(S(S(Z)), LitInt(0)) * Factorial(S(S(Z)),
      LitInt(ite(Factorial(S(S(Z)), LitInt(0)) >= 1,
        -1 + Factorial(S(S(Z)), LitInt(0)),
        0
      ))
    )
  ))
```

later equalities fill whole screens.

Our two hypotheses are that there are so many equalities due to the `Lit`-terms that Z3 can get stuck propagating equalities and/or case splits introduce equalities with `Lit`-terms which interact poorly with the computation axiom.

5.5 Quantifier IDs and Weights

Quantifier IDs and quantifier weight are additional metadata that can be associated with quantifiers (similar to triggers). We use the Quantifier IDs together with the `smt.mbqi.id` option to disable MBQI for most of the quantifiers. Quantifier IDs are also useful when debugging verification performance. For example, if present, SMTscope uses the quantifier ID to refer to a quantifier in the GUI. This enables quick cross-referencing to the original program, provided that the IDs are sensible. The quantifier weight is an unsigned integer, where a higher weight deprioritizes the quantifier for instantiation. It is important to give the computation axioms a higher weight such that the solver first tries to use the general axioms to obtain a proof before restoring to computing a value [2]. Like Dafny, we went with a weight of 3 for the computation axioms (the default weight is 1).

Both ID and weight can be set using the Z3 API, but the rust bindings[2] used by Caesar were missing this feature. The necessary bindings were implemented in a PR[3] and accepted.

[2] Caesar uses z3.rs for Z3 rust bindings: https://github.com/prove-rs/z3.rs.
[3] https://github.com/prove-rs/z3.rs/pull/326

6. Evaluation

To evaluate the quality of the implementation and to compare quantifier handling with/without the encodings, we ran the integration tests on different versions and configurations of Caesar. For each configuration, we set out to answer three questions:

1. How brittle is the verification? How many procedures sometimes switch between verified and `unknown` or counterexample and `unknown`.
2. How complete is the verification? How often do we get verified or a counterexample, not just `unknown`.
3. How fast is the verification?

That the result for a procedure changes from verified/counterexample to `unknown`, or the other way around, for different verifier configurations or seeds, is not desirable but expected due to the inherent incompleteness. Changes between verified and counterexample should never occur and would mean that Caesar is unsound. Further, questions are

- What are the required manual changes such that the encodings can be applied?
- Are the encodings with computation beneficial? Do the `Lit`-marker introduce new problems?
- What is a good maximum fuel value?

First, we present how we measured brittleness, characterize the integration tests and describe the applied modifications to the integration tests. Next, in Section 6.2, we test the brittleness of Caesar without any of the mentioned changes and show that verification is highly sensitive with respect to the Z3 version. The five presented encodings for user-defined recursive functions are compared in Section 6.3. Thereafter, we highlight and quantify problems introduced by the `Lit`-marker (Section 6.4), before discussing what a good maximum fuel values is, and possible alternatives to the encodings with computation in Section 6.5.

6.1 Methodology and Benchmark Set

6.1.1 Measuring Brittleness

To measure brittleness, we run Caesar ten times for each procedure. Each time, we supply a different random seed to Z3 and record the results. This is in line with earlier work by Leino and Pit-Claudel [31]. The `random-seed` option is defined by the SMT-LIB standard [6] and instructs the solver to use the provided seed for initializing its internal random number generators. Consecutive solver runs with the same `random-seed` (that is not the special default seed 0) are guaranteed to behave the same. We always used the seeds 1-10.

Caesar does not have a built-in command for measuring brittleness that also modifies the SMT solver input, for example by renaming variables (an example is the `measure-complexity` command from Dafny [12, Section 13.7.6.1]). But it is also not necessary for the evaluation, as different random seeds are enough of a change to provoke widely different solver behaviour.

Unless otherwise noted, all runs used Z3 4.14.1, an individual timeout of 90 seconds and a memory limit of 4 GiB for each procedure. Everything was run on a system with an Intel Core i7-8550U with 16 GB of RAM.

6.1.2 The Integration Tests

Before this thesis, the Caesar integration tests suite was composed of 34 HeyVL programs, totalling 62 procedures. They include probabilistic as well as Boolean examples and cover the various features and proof rules of

Caesar. Only 11 of the test programs contain recursive user-defined functions. There are not that many examples with recursive functions. This is partly because they previously did not work particularly well. During the implementation, 6 new examples with recursive functions were added, primarily Dafny examples ported from [2] and the sum example from [30]. The final test suite used for evaluation consists of 40 files with 75 different procedures. 17 of these files contain recursive user-defined functions, comprising a total of 33 procedures. Examples of recurring recursive functions include exponential functions, array sums, harmonic numbers and triangle numbers.

6.1.3 Required Modifications to Programs

For the fuel encodings to be applied, the function must be defined with a body, see Figure 5.1. This syntax in not new, but previously the integration tests did not use it. Thus, the programs were updated to use it. For example,

```
func exp(b: UReal, i: UInt): UReal
axiom exp_base forall b: UReal. exp(b, 0) == 1
axiom exp_step forall b: UReal, i: UInt. exp(b, i + 1) == b * exp(b, i)
```

from Figure 1.1 is rewritten to

```
func exp(b: UReal, i: UInt): UReal = ite(i == 0, 1, b * exp(b, i))
```

Similarly for other functions. Computable functions that cannot be defined with this syntax were annotated with `@computable`, see Example 5.4. Some verification problems require axillary axioms that provide additional facts about the functions. An example is

```
axiom exp_bounded forall b: UReal, i: UInt. (b <= 1) ==> (exp(b, i) <= 1)
```

from Figure 1.1. These previously all lacked trigger annotations, leaving it up to the SMT solver to select triggers. Suitable trigger annotations were added to these axioms, such as

```
axiom exp_bounded forall b: UReal, i: UInt @trigger(exp(b, i)).
  (b <= 1) ==> (exp(b, i) <= 1)
```

The overall changes were minor and could be applied mechanical. One can argue that declaring the functions with a body makes the programs easier to read.

6.2 Previous Brittleness

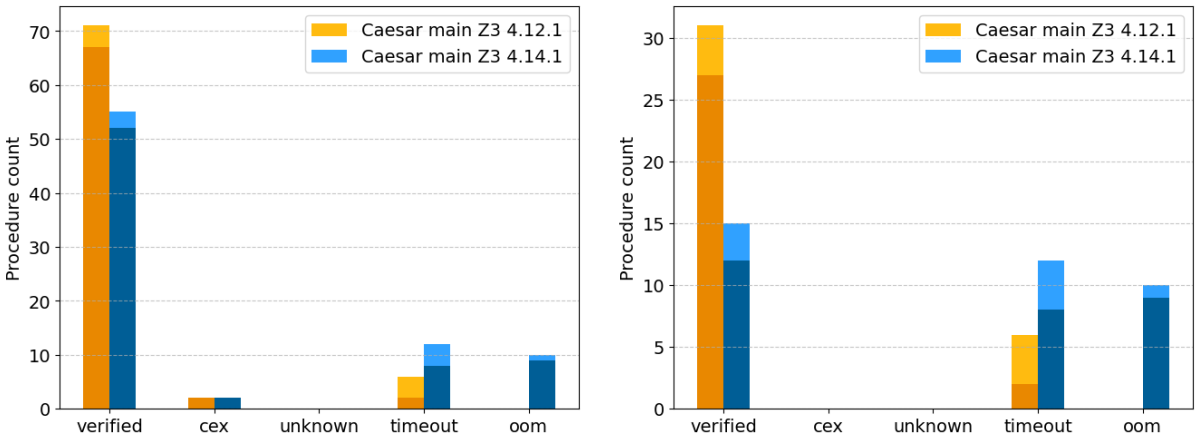
To get a baseline for the previous brittleness, we ran all the integration tests with the current Caesar main branch[1] (without the new changes merged in) in two different configurations. For a fair and accurate view of the current version, we used the same Z3 version (Z3 4.12.1) that is currently bundled with Caesar, the seeds 0-9 (for Z3 the seed 0 is also stable), and included any extra command line flags specified in the test files. For better comparison with all the following configurations, we also ran Caesar main with the same setup as used in the other runs. That is, Z3 4.14.1, seeds 1-10, and without any extra command line flags specified in the files. For both configurations, the previously existing integration tests were taken from that same revision (without the changes described in Section 6.1.3). Any newly added tests were back ported to that version of Caesar, if necessary. Caesar was run with default arguments, meaning user-defined functions and axioms are directly mapped to SMT-LIB, and both E-matching and MBQI are used for quantifier instantiation. A summary of the results is shown in Table 6.1 and Figure 6.1.

All the unknown responses come from procedures that contain user-defined recursive functions. Both configurations of Caesar main show brittleness, with the newer Z3 version performing significantly worse. For 18 procedures unknown is always returned due to exceeded resource limits. This is more than half of the 33 procedures that contain user-defined recursive functions. Both the timeouts and abortions due to memory limits, together with the fact that all problematic procedures included recursive functions, are indications of matching loops. Either quickly resulting in so many quantifier instantiations and new terms that the 4 GiB RAM limit is exceeded, or at least producing too many terms that the remaining solver operations become so slow that the 90-second timeout is exceeded.

[1] https://github.com/moves-rwth/caesar/tree/6a66db5e67d4d346baaf822ecbbc97a8ce3e075f

	Caesar main			
	all integration tests		only with recursive functions	
	Z3 4.12.1	Z3 4.14.1	Z3 4.12.1	Z3 4.14.1
Verified	67 (71)	52 (55)	27 (31)	12 (15)
Counterexample	2 (2)	2 (2)	0 (0)	0 (0)
Unknown	0 (0)	0 (0)	0 (0)	0 (0)
Unknown timeout/OOM	2/0 (6/0)	8/9 (12/10)	2/0 (6/0)	8/9 (12/10)
Total procedures	75	75	33	33
Total execution time	3063 s	13210 s	2632 s	12527 s
Avg. time no timeouts	875 ms	5483 ms	666 ms	12711 ms

Table 6.1: The number of procedures with each possible verification result across all ten seeds is given. The number of procedures in which this result occurred at least once is given in parentheses. The results are given for both Caesar main configurations, both across all integration tests and across the fragment containing user-defined recursive functions. All brittle procedures are part of this fragment.



(a) All procedures

(b) Only procedures with user-defined recursive functions

Figure 6.1: Bar charts with the results from Table 6.1. The darker colours represent the number of procedures that produced this result for all ten seeds, while the lighter colours represent the number of procedures that produced this result at least once. A difference between the two (visible lighter colour) indicates brittleness.

The newer Z3 version is slower in every aspect. The procedures without user-defined functions take longer, as well as the procedures with user-defined functions. This is partially due to the higher number of timeouts, increasing the overall execution time. But also the average verification time for each procedure (excluding timeouts) increased multiple times over.

We think the stark difference between the two Z3 versions is partially explained by the fact that all the tests were developed with Z3 4.12.1. Thus, implicitly formulations were chosen that work well with that Z3 version and not with others. Furthermore, the integration tests are run as part of CI on every commit, as described for the first configuration, but only with seed 0. So at least one successful run is expected for each old procedure with the older Z3 version. This is indeed the case. The 2 procedures that failed completely were newly added. Two old files specified an additional `--no-simplify` option, which turns off an optimization that uses syntactic rewrite rules to simplify the SMT solver input. This option was necessary to verify the files; otherwise, they timed out. The problem here is not with the optimization but that the files contain matching loops and with the different solver input, the solver avoids them (at least for seed 0). These are all different ways how the test files were (subconsciously) designed to work with the older Z3 version. With a different Z3 version, the underlying problems are then highlighted.

In summary, the results are as follows: Both configurations of the Caesar main exhibit brittleness. However, the configuration with the currently bundled Z3 version performs significantly better. Regarding completeness, several procedures are never verified, even when trying with ten different seeds. The overall execution time is relatively long due to the timeouts (Caesar is primarily used interactively within an IDE). Even the better configuration took more than 40 minutes for only the procedures with recursive functions.

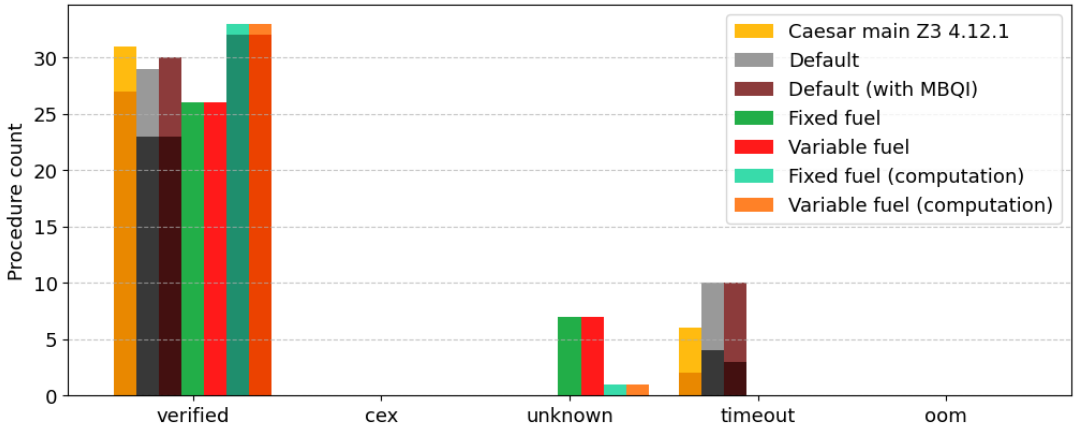


Figure 6.2: Bar charts with the results from Table 6.2. The darker colours represent the number of procedures that produced this result for all ten seeds, while the lighter colours represent the number of procedures that produced this result at least once. A difference between the two (visible lighter colour) indicates brittleness.

6.3 Comparing the Encodings

Next, we compare the five presented encodings with each other. For that, we ran each procedure ten times with each of the encodings (Default encoding, Fixed fuel encoding, Variable fuel encoding, Fixed fuel encoding with computation, and Variable fuel encoding with computation), only enabling E-matching and with a maximum fuel $mf = 2$. This time, we restricted the test to the 33 files that contained user-defined recursive functions. Since the encodings only differ in how they handle those. To judge the effect of disabling MBQI, the Default encoding was run twice, once with only E-matching and once with both E-matching and MBQI. For the fuel encodings this was not done since it defeats their purpose. As a baseline, Caesar main with the bundled Z3 version from the previous section is also included. A summary of the results is shown in Table 6.2 and Figure 6.2.

	Z3 4.12.1	Z3 4.14.1					
			(+ MBQI)			(computation)	
	main	**Default**	**Default**	**Fixed**	**Variable**	**Fixed**	**Variable**
Verified	27(31)	23 (29)	23 (30)	26 (26)	26 (26)	32 (33)	32 (33)
Counterexample	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
Unknown	0 (0)	0 (0)	0 (0)	7 (7)	7 (7)	0 (1)	0 (1)
Unknown timeout/OOM	2/0 (6/0)	4/0 (10/0)	3/0 (10/0)	0/0 (0/0)	0/0 (0/0)	0/0 (0/0)	0/0 (0/0)
Total procedures	33	33	33	33	33	33	33
Total execution time	2632 s	6337 s	6260 s	26.5 s	24 s	33 s	33.5 s
Avg. time no timeouts	666 ms	1503 ms	1881 ms	81 ms	73 ms	100 ms	102 ms

Table 6.2: Comparison of the five different encodings on the files containing user-defined recursive functions. A maximum fuel of $mf = 2$ and only E-matching was enabled for the function encodings. The Default encoding was run twice, once with only E-matching and once also with MBQI. Caesar main is included in the table as a baseline. For each possible result, the table shows the number of procedures that yielded that resulted for every seed, along with the number of procedures in which this result occurred at least once in parentheses.

The difference between the run with Caesar main and the Default encoding is that the latter specifies triggers, either manually added (see Section 6.1.3) or by the (def-axiom), that MBQI was disabled, and the usage of a newer Z3 version. The Default encoding performs worse than Caesar main with Z3 4.12.1, regarding completeness and brittleness, but significantly better than Caesar main with the same newer Z3 version. The Default encoding also still has numerous timeouts, since it does not address the matching loops, present in all the run programs. It performs slightly better when additionally enabling MBQI with one additional procedure verifying some of the time. On average, enabling MBQI results in slower verification times for procedures, excluding timeouts.

The two different ways (fixed vs. variable) of encoding fuel behave very similar. As we expect, the non computation encodings cannot verify all the procedures. They both fail at 7 procedures, all of them containing assertions that check a computed value. They are strictly less complete than the other encodings, but completely stable

on the integration tests. The encodings with computation can verify all examples. Subsequently, they take a bit longer than the non computation encodings, since the 7 computation examples do not immediately result in `unknown`. The fuel endings with computation are less stable. There exists a procedure (same for both fixed and variable) that for some seeds verifies and for others not. Overall, the similar results between the two fuel variants justifies the focus on the Variable fuel encoding/Variable fuel encoding with computation in this chapter.

Regarding counterexamples, the need to disable MBQI for the fuel encodings (with and without computation) to work as desired means that generating a counterexample in the presence of quantifiers is impossible. The integration tests do not include an example that contains user-defined recursive functions and results in a counterexample. If it would, MBQI also would not help, since it cannot construct the infinite model corresponding to the recursive function. Details were discussed in Chapter 4. Even when MBQI is disabled, the two procedures in the test suite that result in counterexamples still do so, since they do not contain quantifiers.

In conclusion, the Default encoding is not an improvement in any regard over Caesar main. The fuel encodings without computation are extremely stable and fast, but have to make compromises in terms of completeness. With computation, the fuel encodings are still fast and can verify all procedures, but are sightly less stable. We explore this fact next.

6.4 Impact of `Lit`-Marking

Next, we examine the impact of the `Lit`-markers, that are used by the computation encodings to mark literal terms. We modified the implementation of the Variable fuel encoding with computation in such a way that we do not restrict the `Lit`-marker to a handful of terms, as described in Section 5.4.1. Instead, every literal term, as determined by Algorithm 3, is wrapped in a `Lit` identity function. We also disabled the check that skips everything related to literal terms if the program does not contain user-defined functions. In such cases, `Lit`-marking terms has no benefit. We called this version aggressive `Lit`-marking. The results are obtained in the usual way (ten runs per procedure with only E-matching, a maximum fuel value $mf = 2$, a 90-second timeout, and 4 GiB memory limit).

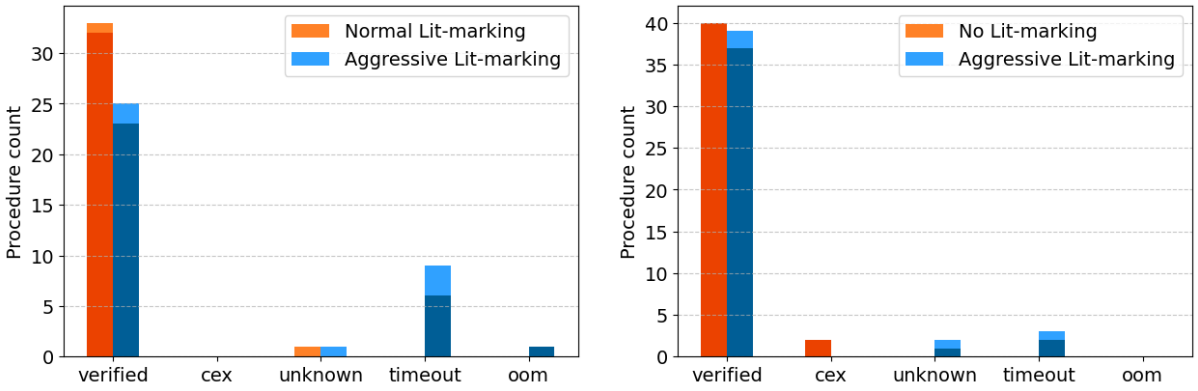
The results are listed separately for procedures with and without recursive functions in Table 6.3 and Figure 6.3. The results for the Variable fuel encoding with computation with normal `Lit`-wrapping (as previously described) is included for comparison.

	Variable fuel (computation)			
	with recursive functions		without recursive functions	
	normal `Lit`	aggressive `Lit`	no `Lit`	aggressive `Lit`
Verified	32 (33)	23 (25)	40 (40)	37 (39)
Counterexample	0 (0)	0 (0)	2 (2)	0 (0)
Unknown	0 (1)	0 (1)	0 (0)	1 (2)
Unknown timeout/OOM	0/0 (0/0)	6/1 (9/1)	0/0 (0/0)	2/0 (3/0)
Total procedures	33	33	42	42
Total execution time	34 s	7127 s	830 s	2604 s
Avg. time no timeouts	104 ms	2845 ms	1989 ms	443 ms

Table 6.3: Results for a modified Variable fuel encoding with computation implementation that always `Lit`-marks terms that are literal according to Algorithm 3. Each procedure was run ten times with different seeds. The number of procedures yielding the result for all seeds is shown, with the number of procedures yielding the result at least once shown in parentheses.

For the procedures with recursive functions, the results are worse than for the Default encoding (compare Table 6.2). As we have seen before with the computation encodings, for one procedure the solver heuristics give up, resulting in `unknown`. But this time, also many runs exceed the timeout and some exceed the memory limit. It is unclear to us why more `Lit`-functions degrade the performance so much. Perhaps this is due to the potentially large number of additional equalities. We also cannot rule out a bug in our implementation, but prior work also performs similar `Lit`-marker optimizations [2].

For the procedures without recursive functions, the (unnecessary) `Lit`-markers are detrimental to the solver performance. The additional identity functions are enough to slow down the solver enough for timeouts to occur. We guess that this is due to the additional equalities and terms. The average verification time per procedure,



(a) Procedures *with* recursive user-defined functions

(b) Procedures *without* recursive user-defined functions

Figure 6.3: Bar charts with the results from Table 6.3. The darker colours represent the number of procedures that produced this result for all ten seeds, while the lighter colours represent the number of procedures that produced this result at least once. A difference between the two (visible lighter colour) indicates brittleness.

excluding timeouts, is actually decreasing, as the more difficult procedures are slowed down so much that they result in timeouts.

With the normal `Lit`-marking, and without MBQI, counterexamples can still be produced for the two wrong procedures in the test suite. This is since they both do not contain quantifiers, and the encoding also does not add any. Therefore, MBQI is not required to construct a model in this case. Always adding `Lit` markers introduces universal quantifiers[2]. Since MBQI is disabled, no more counterexamples can be produced. For wrong procedures, we can then only expect `unknown` as a result.

In conclusion, the additional `Lit`-marker introduced by the computation encodings are bad for the solver performance and should be reduced to the minimum necessary. For procedures without user-defined functions, this means completely omitting them and otherwise applying the optimizations described in Section 5.4.1.

6.4.1 Brittleness Introduced by `Lit`-Marker

As can be observed in Table 6.2 the encodings with computation are also not entirely stable with the mentioned `Lit` optimizations. The one brittle procedure is `arp` from Figure 1.1, with an adjusted definition of `exp`, as described in Section 6.1.3. Proving the correctness of `arp` never requires computing a large value of `exp`. The additional `Lit`-markers are only a hindrance in this case.

We also believe that calculating the exact value of a function for *one* specific input is an unusual use case. Typically, deductive verification is used to obtain a correctness result for *all* inputs. This is supported by the fact that all the examples in the test suite that require computation were added solely to test the computation capability. This raises the question of how beneficial the encodings with computation are over the fuel encodings without computation. An alternative is discussed in the next section.

6.5 Optimal Fuel Value, Fuel Ramping and Hybrid Approaches

To obtain the minimal maximal fuel value mf required for each procedure, we ran each procedure multiple times, starting with $mf = 1$ and increasing it by one if the solver result was `unknown`. mf was iteratively increased until the procedure was verified. This was again done for all procedures containing user-defined recursive functions for ten different seeds. Only E-matching was enabled and the usual resource limits were used. A summary of the results can be found in Table 6.4 and the distribution of the maximum fuel values that were used in the successful runs is shown in Figure 6.4.

By iteratively increasing mf, all procedures can be verified for all seeds. Theoretically, the minimum required mf for a procedure should always be the same. In practice, this is not the case, with one procedure verifying for

[2]The quantifiers come from the implementation of the `Lit`-marker, see Section 5.3.

	Variable fuel (ramping max fuel)
Verified	33 (33)
Counterexample	0 (0)
Unknown	0 (10)
Unknown timeout/OOM	0/0 (0/0)
Total procedures	33
Total runs	878
Total execution time	41.5 s

Table 6.4: Result summeries of verifying each procedure ten times with different seeds using the Variable fuel encoding. Starting at 1, *mf* was iterativly increased by one util verification was succesfull.

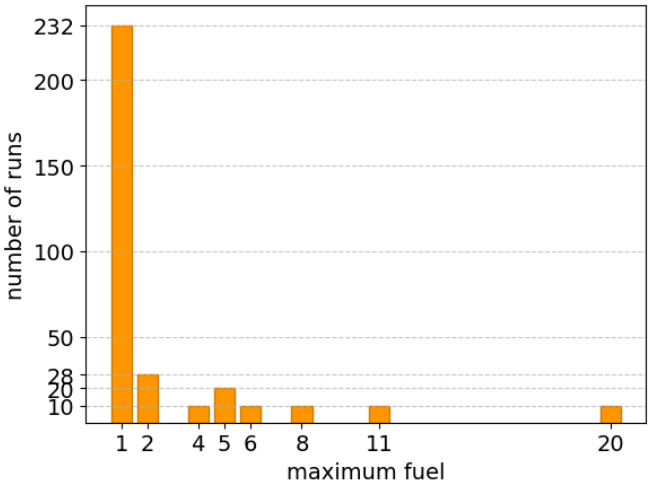


Figure 6.4: Cut of points of the maximum fuel value. When ramping the fuel, these fuel values where required to verify the prodedure.

two seeds with $mf = 1$ but requiring $mf = 2$ for the remaining 8 seeds. Since the fuel encoding is built in such a way that the solver quickly returns `unknown` if a proof is not possible, the total execution time is only about 25% (8 seconds) longer than when using a fuel encoding with computation (compare Table 6.2). This is notable because the total number of runs is 166% more[3], suggesting that the fuel encodings without computation are more efficient than the encodings with computation. The number of required runs could also be decreased by increasing the maximum fuel by more than one upon failure, e.g. doubeling.

Looking at the distribution of *mf* in Figure 6.4 supports that $mf = 2$ is a good default, covering all but the 7 procedures that explicitly compute the value of a function in an assertion.

Iteratively increasing the maximum fuel upon an `unknown` response was inspired by F* [42, Section 46.2.6]. Here, the user can provide an initial and upper maximum fuel value. These are then tried until verification is successful or the upper limit is reached. The ramping of the maximum fuel value seems like a promising alternative to the somewhat unstable computation encodings with their `Lit`-markers. Another possibility is a hybrid approach, where after a certain maximum fuel value we switch to an encoding with computation. The fact that the fuel encodings result in quick `unknown` responses instead of timeouts when verification fails allows us to quickly try multiple different queries, while preserving quick verifier responses, enabling interactive usages.

6.6 Case Study: Coupon Collector

As a final evaluation, we demonstrate with an example how the implemented features help in modelling and successfully verifying probabilistic programs.

Coupon Collector's Problem. We will use the `coupon-collector.heyvl`, the most complex of the programs in the Caesar Test Suite, as the example. The Coupon Collector's Problem is a well-known problem in probability theory. It asks the question: if each box of breakfast cereals contains one of N different coupons, what is the expected number of boxes you need to buy to collect all N different coupons and win the prize (assuming all coupons are equally likely) [13]. The expected number of required boxes turns out to be $N \cdot H_N$, where H_N is the N-th harmonic number. The aforementioned HeyVL program models the Coupon Collector's Problem as a probabilistic programs and verifies that $N \cdot H_N$ is an upper bound on the expected number of required boxes.

Previous. The complete program is relatively involved and not shown here.[4] We focus on the simplifications and improvements enabled by the new features. The program contains two recursive functions, shown in Figure 6.5. The first is a helper function for summing the elements of a list, and the second one generates the harmonic

[3] For the other configuration, the total number of runs is $33 \cdot 10 = 330$.

[4] Interested readers can find it here: https://github.com/moves-rwth/caesar/blob/9840dd86d73068a1085d458f876e00c0a4dc415b/tests/coupon-collector.heyvl.

numbers. Since the verification of the program was very brittle, `harmonic` was encoded as a limited function by hand in an attempt to make it more stable. The (ff-syn-axiom) is missing from the hand-rolled encoding. For the CI, it also used the `--no-simplify` "trick" (cf. Section 6.2).

```
1  func sum(elements: []UReal, start: UInt, len: UInt): UReal =
2    ite(len>0, select(elements, start+len-1) + sum(elements, start, len-1), 0)
3
4  func harmonic(n: UInt): UReal
5  func harmonic0(n: UInt): UReal
6  axiom harmonic_def forall n: UInt.
7    (harmonic(n) == ite(n==0, 0, (1/n) + harmonic0(n-1)))
```

Figure 6.5: The two recursive functions from `coupon-collector.heyvl` as previously defined. The first implements the sum $\sum_{i=\texttt{start}}^{\texttt{start+len}-1} \texttt{elements}[i]$ and the second function generates the harmonic numbers. In an attempt to fight brittleness, a partial limited function encoding was done by hand.

Problems. Since the Equation (ff-syn-axiom) is missing from the hand-rolled encoding, it had to manually be included using (co)assume stamens in the relevant places such that verification could succeed. In general, the lack of triggers and the fact that MBQI is not disabled renders the hand-rolled encoding for `harmonic` not very effective in reducing the brittleness of the program. The recursive `sum` function was not addressed at all. Even though it is defined with a body, in previous Caesar versions these were just translated using a definitional axiom without triggers. When run with the Caesar main (Table 6.1), this caused the complete program (all 7 procedures) to not be verified for one seed with Z3 4.12.1 and to never be verified with Z3 4.14.1. Ignoring timeouts, the average time needed for verification was over 6 seconds.

Improvements. By defining `harmonic` with a body and using any of the fuel encodings, we correctly encode both recursive functions as limited functions (all axioms and triggers). The manually assumed synonym axiom can be removed. By disabling MBQI the limited functions ensure that the solver terminates. For any of the 4 fuel encodings, the modified programs always stably verify. The average verification time also goes down significantly to well below a second for all encodings. Using a built-in fuel encoding also has the advantage that the HeyVL program is not cluttered with SMT encoding details. Users can write standard code, in the knowledge that the verifier will take care of the encoding. This approach is also more flexible. For example, it is easily possible to test the program with a higher maximum fuel.

These positive results are not limited to this example. First tests with other complex examples using quantifiers also show promising results. Examples that never worked previously, now verify and are stable with the newly introduced fuel encodings.

6.7 Conclusion

We saw that there were a number of instabilities caused by matching loops on the Caesar integration tests. This problem was especially highlighted after updating the Z3 version. Due to the frequent timeouts, overall execution time was relatively slow. By using one of the fuel encodings and only enabling E-matching, the matching loops are eliminated and thus also the timeouts. The procedure either quickly verifies or the solver quickly returns `unknown`. The two different fuel variants (fixed/variable) are very comparable. The Default encoding does not perform very well, since it does not address the matching loops. Procedures that compute the concrete value of a user-defined recursive function as part of an assertion cannot be verified by the fuel encodings without computation, at least for a small maximal fuel value. The fuel encodings with computation were able to verify all examples. So the fuel encodings with computation are, in practice, more complete than the current Caesar main and also faster. When only considering successful runs, that did not time out, the average execution time went down by a factor of more than 6.

The computation encodings are not without their problems. The introduced `Lit`-functions are not good for the solver performance and can cause brittleness. They should be reduced to the minimum amount necessary.

The most promising strategy, mainly regarding stability, seems to be fuel ramping. Here, a fuel encoding without computation is run multiple times, increasing the maximum fuel value, until the procedure is verified. It uses

the fact that when using a restrictive encoding, E-Matching terminates with unknown if a proof is not possible. Allowing for another try with a little less restrictive encoding.

7. Related Work

7.1 Related Work

Limited functions. The fuel encodings are borrowed from other deductive verifiers. The Variable fuel encoding with computation was initially designed in [2] for Dafny [28]. Inspired by this, F* [43] uses the Variable fuel encoding for recursive functions [1]. F* can also incrementally try higher fuel values by setting an initial fuel n and maximum fuel m. The fuel is initially n and then increased whenever the SMT solver returns unknown up to m [42, Section 46.2.6]. F* does not include computation axioms or determine literal terms. Instead, proofs that can be performed using computation leverage symbolic execution [42, Section 1.4]. Finally, close similarities between the fuel encodings and the comprehension encoding in [30] were discussed in detail in Section 3.1.4.

Formal semantics of E-matching. Recently, Ge et al. developed small-step operational semantics for E-matching, which can be used for proving the termination of axiomatizations when relying on E-matching for quantifier reasoning [23]. In our setting, it is the operational counterpart to the more denotational semantics defined by SMT-LIB standard [6]. In an earlier formalization of E-matching, Dross et al. extended FOL with triggers and witnesses [18].

Counterexamples. To enable the solver to find models involving recursive functions, one has to deal with the infinite models. We bounded the depth of the recursive function and searched for a model within this defined fragment, in Section 4.3. By bounding the depth of the recursive function, it is only defined for the relevant parts of the domain and the model is finite. A similar approach was taken in [37]. For each recursive function f, a new uninterpreted sort is introduced, representing the abstract domain of f. Additionally, for each argument of f a concretization function from the abstract domain to the argument sort is introduced. The definitional axiom for f only quantifies over the abstract domain, and concretization functions are inserted as necessary. Since, quantification is over the abstract domain, which is an uninterpreted sort, finite model finding techniques implemented by SMT solvers are applicable. The generated formula also falls into the *essentially uninterpreted* fragment, for which a complete instantiation procedure is implemented in Z3 [24]. The produced model f is correctly defined for all points that are relevant to the counterexample. In theory, this makes it a more flexible version of the bounded approach discussed in Section 4.3 that does not require giving a prior bound, but where this bound is determined by the SMT solver during the SAT check.

Another option is to find a finite representation for infinite models. Elad et al. used symbolic representation to represent certain infinite counter-models occurring in verification problems. For this class of infinite models, the model-checking problem can be reduced to the satisfiability-checking problem for LIA formulae. The search for a symbolic representation can also be encoded, based on templates, as a satisfiability problem of a LIA formula. The approach is complete for Effectively Propositional formulae extended by one sort that is allowed to have cyclic functions [19].

A third option is to externalize the definition of the recursive functions from the solver input. Frohn and Giesl presented a technique for incorporating exponential integer arithmetic into SMT solvers [21]. It is based on a CEGAR (Counter Example Guided Abstraction Refinement) loop. The exponential function exp is abstracted to be an uninterpreted function. Whenever a model is produced, it is checked if exp matches the semantics of an exponential function. If it does, a correct model was found. Otherwise, suitable lemmas are added to rule out the current model and the next iteration of the loop starts. By not including the definition of exponentials in the input, but iteratively adding what is necessary for a correct model, the model stays constructible for the solver.

Unfolding definitions. The instantiation of the definitional axiom for $f(\bar{x})$, when $f(\bar{t})$ is present in the ground terms, is distinctly similar to a heuristic known in the literature as "unfold-and-match" [32, 36]. Here, recursive functions are first unfolded. In the subsequent satisfiability check, they are then considered to be uninterpreted. In practice, the recursive definitions can not be unfolded completely, but the unfolding depth is iteratively increased. The presented UQFR (Unfolding and Quantifier-Free Reasoning) algorithm in [36, Fig. 1] reads very

similar to E-matching (Algorithm 1). UQFR differs from E-matching in the aspect that it completely unfolds (think instantiate) all definitions once on each iteration. This yields a complete procedure for a FOL fragment called FLUID (First-Order Logic of Universal properties under Inductive Definitions) [36]. Using a similar approach, a completeness result is given in [32] for a different FOL fragment called *safe*. These results shed some light on why E-matching is so effective for program verification. They also give two reasons why some proofs fail. One reason is that there exists a nonstandard model that satisfies the formula (remember that we show validity by the unsatisfiability of the negation). Another reason is that, one can only model fixpoint semantics in FOL, not *least*-fixpoint semantics. And thus the formula might be satisfiable when choosing another fixpoint than the least. Additional induction lemmas must be added that exclude the unwanted models [32, 36].

Brittleness and verification performance. Brittleness and unpredicable verification performance are problems that all deductive verifier have to deal with. The literature includes examples of general guidance on trigger design [34, 31] (manual and automated). The SMTscope[1] tool (formerly known as Axiom Profiler [8]), is specifically being developed to be able to better understand and debug the behaviour of SMT solvers (for more details, see Section 2.3.1). Dafny has separate guides for stabilising verification [45] and addressing poor verification performance[2]. The F* handbook also contains a section detailing techniques to address performance problems [42, Section 46.4].

[1] https://github.com/viperproject/smt-scope
[2] https://dafny.org/dafny/VerificationOptimization/VerificationOptimization

8. Conclusion

In this thesis, we greatly reduced the verification brittleness in Caesar. The instabilities were mainly caused by quantifiers. More specifically, matching loops – a problem related to E-matching-based quantifier instantiation. When naively encoding user-defined recursive functions, a matching loop is always introduced. Therefore, we looked at limited functions, an encoding technique used by other verifies to prevent matching loops, by limiting the number of recursive quantifier instantiations.

We formally defined two different variants of a fuel encoding, which generate limited functions. The main theoretical contribution of this thesis is a formal proof that the limited function encodings are equisatisfiable to the Default encoding. Building on this result, we showed that Caesar remains sound when implementing the fuel encodings. We also showed that, when using E-matching, the number of quantifier instantiations caused by the fuel encodings is finite, and thus the solver terminates.

We also examined possibilities for obtaining models (counterexamples) for recursive functions. Since a model of a recursive function is usually infinite, it cannot be constructed by current solvers. We explored how one of the fuel encodings can be modified to make the resulting model finite. A bounded search on a finite fragment of the recursive function can then result in guaranteed sound and constructible counterexamples.

The fuel encodings were all implemented in Caesar. An evaluation on the integration test shows that brittleness is nearly eliminated. Subsequently, the verification performance is greatly increased for programs with recursive functions. On the one hand, this is due to no longer running into timeouts and on the other hand, by better guiding the SMT solver. Excluding previous timeouts, verification of programs with recursive functions became faster by more than a factor of 6. First tests suggest that the increased stability enables modelling and verifying new kinds of programs that were previously impossible.

Having to largely disable MBQI for the fuel encodings means that counterexamples are no longer be produced if the input contains quantifiers. The approach also only covers functions that can be defined with a single expression. There are still many possibilities for better quantifier handling and reducing brittleness in Caesar.

8.1 Future Work

Synthesizing triggers. In this work, we were only concerned with finding good encodings (that include triggers) for user-defined functions. But the user can include quantifiers in any condition (axiom, specification, assume, …). Here, the user can manually specify triggers with the `@trigger` annotation, but when they are omitted, Caesar leaves it up to the SMT solver to come up with suitable triggers. For Dafny, it has been shown to be beneficial to synthesize the triggers on the level of the verifier for more predicatable results [31]. Viper also synthesizes the trigger itself [40, Section 2.6]. In the future, it should be evaluated whether it would be beneficial for Caesar to determine the triggers instead of the SMT solver, particularly for quantitative quantifiers.

Quantitative quantifiers. Caesar, using a quantitative logic, also has quantitative quantifiers (infimum/supremum). For certain fragments, they can be eliminated [38, Section 4.2]. We only briefly discussed their current (non-optimal) encoding in Section 5.2 for when they cannot be eliminated. Very recently, a quantitative quantifier elimination technique for piecewise linear quantities was presented in [7]. If it turns out to be feasible in practise, it would allow circumventing some problems related to quantitative quantifiers by eliminating more of them. Regardless, important next improvements are that suitable triggers should be specified when quantitative quantifiers are introduced by Caesar. The user should also be able to specify triggers when declaring a quantitative quantifier themselves. Here it is not immediately obvious how this would be done since (at least currently) a single quantitative quantifier is encoded with multiple different Boolean quantifiers. This might also be an opportunity where Caesar can infer better triggers than the SMT solver (see the previous point).

Testing for brittlenss. For evaluating the brittleness, we used external scripts. But this information is also in-

teresting for intermediate and advanced users of the tool when developing and debugging their programs. For example, Dafny has an extra `measure-complexity` command to measure the brittleness of proofs [12, Section 13.7.6.1]. Similar functionality would also be valuable for Caesar. Helping users and developers of Caesar to quantify proof brittleness keep it under control.

Built-in support. Constructs like exponentials, harmonic numbers or general sums frequently occur in probabilistic programs. Rather than the user having to come up with a suitable encoding each time, Caesar could provide a high-quality, well-tuned encoding itself. For encoding sums, there exists prior work by Leino et al. [30] that is closely related to the fuel encodings (cf. Section 3.1.4). We already briefly discussed the advantages of a dedicated CEGAR loop for exponentials [21] over a generic CEGAR loop (MBQI) in Section 4.3.1.

Implementation. The different present encodings for user-defined functions were implemented as part of this thesis. Still, while testing and evaluating, other encodings and strategies emerged. It would be interesting to implement and test the strategies mentioned in Section 6.5. There is fuel ramping inspired by F* [42, Section 46.2.6] where the maximum fuel is iteratively increased when it was not enough and the solver returned `unknown`. There is also the hybrid strategy of first trying a fuel encoding without computation and then on an `unknown` response switching to a fuel encoding with computation. Maybe they could also both be combined. For obtaining counterexamples, the fixed depth encdoing (Section 4.2 and onwards) and similar approaches in the literature [37] should be explored further. The general idea of running multiple different queries in parallel that are tuned for different results seems promising. For example, one query with only E-matching for verification and another with MBQI for finding a counterexample.

Bibliography

[1] Alejandro Aguirre. *Towards a provably correct encoding from F* to SMT*. Tech. rep. INRIA, 2016.

[2] Nada Amin, K Rustan M Leino, and Tiark Rompf. "Computing with an SMT solver". In: *Tests and Proofs: 8th International Conference, TAP 2014, Held as Part of STAF 2014, York, UK, July 24-25, 2014. Proceedings 8*. Springer. 2014, pp. 20–35.

[3] *An Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware*. RFC 826. Nov. 1982. DOI: 10.17487/RFC0826. URL: https://www.rfc-editor.org/info/rfc826.

[4] Suzana Andova, Holger Hermanns, and Joost-Pieter Katoen. "Discrete-time rewards model-checked". In: *Formal Modeling and Analysis of Timed Systems: First International Workshop, FORMATS 2003, Marseille, France, September 6-7, 2003. Revised Papers 1*. Springer. 2004, pp. 88–104.

[5] Haniel Barbosa et al. "cvc5: A versatile and industrial-strength SMT solver". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2022, pp. 415–442.

[6] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.6*. Tech. rep. Available at www.SMT-LIB.org. Department of Computer Science, The University of Iowa, 2017.

[7] Kevin Batz, Joost-Pieter Katoen, and Nora Orhan. "Quantifier Elimination and Craig Interpolation: The Quantitative Way". In: *Foundations of Software Science and Computation Structures*. Ed. by Parosh Aziz Abdulla and Delia Kesner. Cham: Springer Nature Switzerland, 2025, pp. 176–197. ISBN: 978-3-031-90897-2.

[8] Nils Becker, Peter Müller, and Alexander J Summers. "The axiom profiler: Understanding and debugging smt quantifier instantiations". In: *Tools and Algorithms for the Construction and Analysis of Systems: 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings, Part I 25*. Springer. 2019, pp. 99–116.

[9] Nikolaj Bjørner et al. "Programming Z3". In: *Engineering Trustworthy Software Systems: 4th International School, SETSS 2018, Chongqing, China, April 7–12, 2018, Tutorial Lectures 4* (2019), pp. 148–201.

[10] Henrik Bohnenkamp et al. "Cost-optimisation of the IPv4 zeroconf protocol". In: *International Computer Performance and Dependability Symposium (IPDS'03)*. IEEE Computer Society Press, 2003, pp. 531–540.

[11] Stuart Cheshire, Dr. Bernard D. Aboba, and Erik Guttman. *Dynamic Configuration of IPv4 Link-Local Addresses*. RFC 3927. May 2005. DOI: 10.17487/RFC3927. URL: https://www.rfc-editor.org/info/rfc3927.

[12] The dafny-lang community. *Dafny Reference Manual*. URL: https://dafny.org/dafny/DafnyRef/DafnyRef (visited on 05/05/2025).

[13] *Coupon collector's problem – Wikipedia*. URL: https://en.wikipedia.org/wiki/Coupon_collector%27s_problem (visited on 05/15/2025).

[14] Leonardo De Moura and Nikolaj Bjørner. "Efficient E-matching for SMT solvers". In: *Automated Deduction–CADE-21: 21st International Conference on Automated Deduction Bremen, Germany, July 17-20, 2007 Proceedings 21*. Springer. 2007, pp. 183–198.

[15] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver". In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[16] David Detlefs, Greg Nelson, and James B Saxe. "Simplify: a theorem prover for program checking". In: *Journal of the ACM (JACM)* 52.3 (2005), pp. 365–473.

[17] The Z3 developers. *Parameters | Online Z3 Guide*. URL: https://microsoft.github.io/z3guide/programming/Parameters (visited on 04/15/2025).

[18] Claire Dross et al. “Adding decision procedures to SMT solvers using axioms with triggers”. In: *Journal of Automated Reasoning* 56 (2016), pp. 387–457.

[19] Neta Elad, Oded Padon, and Sharon Shoham. “An infinite needle in a finite haystack: Finding infinite counter-models in deductive verification”. In: *Proceedings of the ACM on Programming Languages* 8.POPL (2024), pp. 970–1000.

[20] Jean-Christophe Filliâtre and Andrei Paskevich. “Why3—where programs meet provers”. In: *European symposium on programming*. Springer. 2013, pp. 125–128.

[21] Florian Frohn and Jürgen Giesl. “Satisfiability Modulo Exponential Integer Arithmetic”. In: *International Joint Conference on Automated Reasoning*. Springer. 2024, pp. 344–365.

[22] Erich Gamma et al. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1994. ISBN: 0-201-63361-2.

[23] Rui Ge, Ronald Garcia, and Alexander J Summers. “A Formal Model to Prove Instantiation Termination for E-matching-Based Axiomatisations”. In: *International Joint Conference on Automated Reasoning*. Springer. 2024, pp. 419–438.

[24] Yeting Ge and Leonardo De Moura. “Complete instantiation for quantified formulas in satisfiabiliby modulo theories”. In: *Computer Aided Verification: 21st International Conference, CAV 2009, Grenoble, France, June 26-July 2, 2009. Proceedings 21*. Springer. 2009, pp. 306–320.

[25] Biniam Gebremichael, Frits Vaandrager, and Miaomiao Zhang. “Analysis of a protocol for dynamic configuration of IPv4 link local addresses using Uppaal”. In: *ICIS, Radboud University Nijmegen, Tech. Rep. ICIS-R06xxx* (2006).

[26] Benjamin Lucien Kaminski and Joost-Pieter Katoen. “On the hardness of almost–sure termination”. In: *International Symposium on Mathematical Foundations of Computer Science*. Springer. 2015, pp. 307–318.

[27] Marta Kwiatkowska et al. “Performance analysis of probabilistic timed automata using digital clocks”. In: *Formal Methods in System Design* 29.1 (2006), pp. 33–78.

[28] K Rustan M Leino. “Dafny: An automatic program verifier for functional correctness”. In: *International conference on logic for programming artificial intelligence and reasoning*. Springer. 2010, pp. 348–370.

[29] K Rustan M Leino. “This is boogie 2”. In: *manuscript KRML* 178.131 (2008), p. 9.

[30] K Rustan M Leino and Rosemary Monahan. “Reasoning about comprehensions with first-order SMT solvers”. In: *Proceedings of the 2009 ACM symposium on Applied Computing*. 2009, pp. 615–622.

[31] K Rustan M Leino and Clément Pit-Claudel. “Trigger selection strategies to stabilize program verifiers”. In: *International Conference on Computer Aided Verification*. Springer. 2016, pp. 361–381.

[32] Christof Löding, P Madhusudan, and Lucas Peña. “Foundations for natural proofs and quantifier instantiation”. In: *Proceedings of the ACM on Programming Languages* 2.POPL (2017), pp. 1–30.

[33] IUrii V Matiyasevich. *Hilbert’s tenth problem*. MIT press, 1993.

[34] Michał Moskal. “Programming with triggers”. In: *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*. 2009, pp. 20–29.

[35] Peter Müller, Malte Schwerhoff, and Alexander J Summers. “Viper: A verification infrastructure for permission-based reasoning”. In: *Verification, Model Checking, and Abstract Interpretation: 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings 17*. Springer. 2016, pp. 41–62.

[36] Adithya Murali et al. “Complete first-order reasoning for properties of functional programs”. In: *Proceedings of the ACM on Programming Languages* 7.OOPSLA2 (2023), pp. 1063–1092.

[37] Andrew Reynolds et al. “Model finding for recursive functions in SMT”. In: *Automated Reasoning: 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27–July 2, 2016, Proceedings 8*. Springer. 2016, pp. 133–151.

[38] Philipp Schroer. “A Deductive Verifier for Probabilistic Programs”. Available at https://publications.rwth-aachen.de/record/998370/files/998370.pdf. Master’s thesis. RWTH Aachen, Apr. 2023.

[39] Philipp Schröer et al. “A Deductive Verification Infrastructure for Probabilistic Programs”. In: *Proceedings of the ACM on Programming Languages* 7.OOPSLA2 (2023), pp. 2052–2082.

[40] Malte H Schwerhoff. “Advancing automated, permission-based program verification using symbolic execution”. PhD thesis. ETH Zurich, 2016.

[41] Cédric Stoll. “SMT Models for Verification Debugging”. Available at https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Theses/Cedric_Stoll_MA_report.pdf. Master’s thesis. ETH Zürich, May 2019.

[42] Nikhil Swamy, Guido Martinez, and Aseem Rastogi. *Proof-Oriented Programming in F**. Available at https://fstar-lang.org/tutorial/proof-oriented-programming-in-fstar.pdf. Apr. 2025.

[43] Nikhil Swamy et al. “Dependent types and multi-monadic effects in F”. In: *Proceedings of the 43rd annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2016, pp. 256–270.

[44] Alfred Tarski. “A Decision Method for Elementary Algebra and Geometry”. In: *Journal of Symbolic Logic* 17.3 (1952).

[45] Aaron Tomb and Jean-Baptiste Tristan. *Avoiding verification brittleness in Dafny*. URL: https://dafny.org/blog/2023/12/01/avoiding-verification-brittleness/ (visited on 04/02/2025).

[46] Max Willsey et al. “Egg: Fast and extensible equality saturation”. In: *Proceedings of the ACM on Programming Languages* 5.POPL (2021), pp. 1–29.