Summing Up Smart Transitions

Neta Elad[1], Sophie Rain[2], Neil Immerman[3], Laura Kovács[2], and Mooly Sagiv[1]

[1] Tel Aviv University, Israel
[2] TU Wien, Austria
[3] UMass Amherst, USA

Abstract. Some of the most significant high-level properties of currencies are the sums of certain account balances. Properties of such sums can ensure the integrity of currencies and transactions. For example, the sum of balances should not be changed by a transfer operation. Currencies manipulated by code present a verification challenge to mathematically prove their integrity by reasoning about computer programs that operate over them, e.g., in Solidity. The ability to reason about sums is essential: even the simplest ERC-20 token standard of the Ethereum community provides a way to access the total supply of balances.

Unfortunately, reasoning about code written against this interface is non-trivial: the number of addresses is unbounded, and establishing global invariants like the preservation of the sum of the balances by operations like transfer requires higher-order reasoning. In particular, automated reasoners do not provide ways to specify summations of arbitrary length. In this paper, we present a generalization of first-order logic which can express the unbounded sum of balances. We prove the decidablity of one of our extensions and the undecidability of a slightly richer one. We introduce first-order encodings to automate reasoning over software transitions with summations. We demonstrate the applicability of our results by using SMT solvers and first-order provers for validating the correctness of common transitions in smart contracts.

1 Introduction

A basic challenge in smart contract verification is how to express the functional correctness of transactions, such as currency minting or transferring between accounts. Typically, the correctness of such a transaction can be verified by proving that the transaction leaves the sum of certain account balances unchanged.

Consider for example the task of minting an unbounded number of tokens in the simplified ERC-20 token standard of the Ethereum community [?], as illustrated in Figure 1[4]. This example deposits the minted amount (`n`) into the receiver's address (`a`) and we need to ensure that the `mint` operation *only* changed the balance of the receiver. To do so, in addition to (i) proving that the balance of the receiver has been increased by `n`, we also need to verify that (ii) the

[4] The `old-` prefix denotes the value of a function before the `mint` transition, and the `new-` prefix denotes the value afterwards.

```
a: Address
n: Nat
---------------------------------------------------------------
mint(a,n)
---------------------------------------------------------------
# Post-conditions
assert new-bal(a) = old-bal(a) + n                          #(i)
for each Address a′ ≠ a:                                    #(ii)
    assert new-bal(a′) = old-bal(a′)
assert new-sum() = old-sum() + n                            #(iii)
```

Fig. 1: Minting n Tokens in ERC-20.

account balance of every user address a′ different than a has not been changed during the mint operation and that (iii) the sum of all balances changed exactly by the amount that was minted. The validity of these three requirements (i)-(iii), formulated as the post-conditions of Figure 1, imply its functional correctness.

Surprisingly, proving formulas similar to the post-conditions of Figure 1 is challenging for state-of-the-art automated reasoners, such as SMT solvers [?,?,?] and first-order provers [?,?,?]: it requires reasoning that links local changes of the receiver (a) with a global state capturing the sum of all balances, as well as constructing that global state as an aggregate of an unbounded but finite number of Address balances. Moreover, our encoding of the problem uses discrete coins that are minted and deposited, whose number is unbounded but finite as well.

In this paper we address verification challenges of software transactions with aggregate properties, such as preservation of sums by transitions that manipulate low-level, individual entities. Such properties are best expressed in higher-order logic, hindering the use of existing automated reasoners for proving them. To overcome such a reasoning limitation, we introduce *Sum Logic* (SL) as a generalization of first-order logic, in particular of Presburger arithmetic. Previous works [?,?,?] have also introduced extensions of first-order logic with aggregates by counting quantifiers or generalized quantifiers. In Sum Logic (SL) we only consider the special case of integer sums over uninterpreted functions, allowing us to formalize SL properties with and about unbounded sums, in particular sums of account balances, without higher-order operations (Section 3). We prove the decidability of one of our SL extensions and the undecidability of a slightly richer one (Section 4). Given previous results [?], our undecidability result is not surprising. In contrast, what may be unexpected is our decidability result and the fact that we can use our first-order fragment for a convenient and practical new way to verify the correctness of smart contracts.

We further introduce first-order encodings which enable automated reasoning over software transactions with summations in SL (Section 5). Unlike [?], where SMT-specific extensions supporting higher-order reasoning have been introduced, the logical encodings we propose allow one to use existing reasoners without any modification. We are not restricted to SMT reasoning, but can also leverage generic automated reasoners, such as first-order theorem provers, supporting first-order logic. We believe our results ease applying automated reasoning to smart contract verification even for non-experts.

We demonstrate the practical applicability of our results by using SMT solvers and first-order provers for validating the correctness of common financial transitions appearing in *smart contracts* (Section 6). We refer to these transitions as *smart transitions*. We encode SL into pure first-order logic by adding another sort that represents the tokens of the crypto-currency themselves (which we dub “coins”).

Although the encodings of Section 5 do not translate to our decidable SL fragment from Section 4, our experimental results show that automated reasoning engines can handle them consistently and fast. The decidability results of Section 5 set the boundaries for what one can expect to achieve, while our experiments from Section 5 demonstrate that the unknown middle-ground can still be automated.

While our work is mainly motivated by smart contract verification, our results can be used for arbitrary software transactions implementing sum/aggregate properties. Further, when compared to the smart contract verification framework of [**?**], we note that we are not restricted to proving the correctness of smart contracts as finite-state machines, but can deal with semantic properties expressing financial transactions in smart contracts, such as currency minting/-transfers.

While ghost variable approaches [**?**] can reason about changes to the global state (the sum), our approach allows the verifier to specify only the local changes and automatically prove the impact on the global state.

Contributions. In summary, this paper makes the following contributions:

- We present a generalization to Presburger arithmetic (SL, in Section 3) that allows expressing properties about summations. We show how we can formalize verification problems of smart contracts in SL.
- We discuss the decidability problem of checking validity of SL formulas (Section 4): we prove that it is undecidable in the general case, but also that there exists a small decidable fragment.
- We show different encodings of SL to first-order logic (Section 5). To this end, we consider theory-specific reasoning and variations of SL, for example by replacing non-negative integer reasoning with term algebra properties.
- We evaluate our results with SMT solvers and first-order theorem provers, by using 31 new benchmarks encoding smart transitions and their properties (Section 6). Our experiments demonstrate the applicability of our results within automated reasoning, in a fully automated manner, without any user guidance.

2 Preliminaries

We consider many-sorted first-order logic (FOL) with equality, defined in the standard way. The equality symbol is denoted by $\approx$.

We denote by STRUCT $[\Sigma]$ the *set of all structures* for the vocabulary Σ. A structure $\mathcal{A} \in$ STRUCT $[\Sigma]$ is a pair $(\mathcal{D}, \mathcal{I})$, where for each sort $\boldsymbol{s}$, its domain in $\mathcal{A}$ is $\mathcal{D}(\boldsymbol{s})$, and for each symbol S, its interpretation in $\mathcal{A}$ is $\mathcal{I}(S)$. Note that *models* of a formula φ over a vocabulary Σ are structures $\mathcal{A} \in$ STRUCT $[\Sigma]$.

A *first-order theory* is a set of first-order formulas closed under logical consequence. We will consider, the first-order theory of the natural numbers with addition. This is Presburger arithmetic (PA) which is of course decidable [**?**]. We write $\mathbb{N}$ to denote the set of natural numbers. We consider $0 \in \mathbb{N}$ and write $\mathbb{N}^+$ to explicitly exclude 0 from $\mathbb{N}$. The vocabulary of PA is $\Sigma_{\text{Presburger}} = \left(0, 1, c_1, \ldots, c_l, +^2\right)$, with all constants $0, 1, c_i$ of sort `Nat`. A structure $\mathcal{A} = (\mathcal{D}, \mathcal{I}) \in$ STRUCT $[\Sigma_{\text{Presburger}}]$ is called a *Standard Model of Arithmetic* when $\mathcal{D}(\texttt{Nat}) = \mathbb{N}$ and $+^2$ is interpreted as the standard binary addition + function over the naturals. The vocabulary $\Sigma_{\text{Presburger}}$ can be extended with a total order relation, yielding $\Sigma^*_{\text{Presburger}} = \left(0, 1, +^2, \leq^2\right)$, where $\leq^2$ is interpreted as the binary relation $\leq$ in Standard Models of Arithmetic.

3 Sum Logic (SL)

We now define *Sum Logic (* SL*)* as a generalization of Presburger arithmetic, extending Presburger arithmetic with unbounded sums. SL is motivated by applications of financial transactions over cryptocurrencies in smart contracts. Smart contracts are decentralized computer programs executed on a blockchain-based system, as explained in [**?**]. Among other tasks, they automate financial transactions such as transferring and minting money. We refer to these transactions as *smart transitions*. The aim of this paper and SL in particular is to express and reason about the post-conditions of smart transitions similar to Figure 1.

SL expresses smart transition relations among sums of accounts of various kinds, e.g., at different banks, times, etc. Each such kind, j, is modeled by an uninterpreted function symbol, b_j, where $b_j(a)$ denotes the balance of a's account of kind j, and a constant symbol s_j, which denotes the sum of all outputs of b_j. As such, our SL generalizes Presburger arithmetic with (i) a sort `Address` corresponding to the (unbounded) set of account *addresses*; (ii) *balance* functions b_j mapping account addresses from `Address` to account values of sort `Nat`; and (iii) *sum constants* s_j of sort `Nat` capturing the total sum of all account balances represented by b_j. Formally, the vocabulary of SL is defined as follows.

Definition 1 (SL Vocabulary). *Let*

$$\Sigma^{l,m,d}_{+,\leq} = \left(a_1, \ldots, a_l, b^1_1, \ldots, b^1_m, c_1, \ldots, c_d, s_1, \ldots, s_m, 0, 1, +^2, \leq^2\right)$$

be a sorted first-order vocabulary of SL *over sorts* $\{\texttt{Address}, \texttt{Nat}\}$*, where*

- *(Addresses) The constants* $a_1, \ldots, a_l$ *are of sort* `Address`*;*
- *(Balance functions)* $b^1_1, \ldots, b^1_m$ *are unary function symbols from* `Address` *to* `Nat`*;*
- *(Constants and Sums) The constants* $c_1, \ldots, c_d, s_1, \ldots, s_m$ *and* $0, 1$ *are of sort* `Nat`*;*

Function	**Encoding in SL**	**Reference in ERC-20**
`sum`	s or s'	totalSupply
`bal`(a)	$b(a)$ or $b'(a)$	balanceOf
`mint`(a, v)	$b'(a) \approx b(a) + v$	transfer
`transferFrom`(f, t, v)	$b'(t) \approx b(t) + v \wedge b(f) \approx b'(f) + v$	transferFrom

Table 1: ERC-20 Token Standard

- $+^2$ *is a binary function* $\texttt{Nat} \times \texttt{Nat} \to \texttt{Nat}$*;*
- $\leq^2$ *is a binary relation over* $\texttt{Nat} \times \texttt{Nat}$*.*

In what follows, when the cardinalities in an SL vocabulary are clear from context, we simply write Σ instead of $\Sigma^{l,m,d}_{+,\leq}$. Further, by $\Sigma^{l,m,d}_{\cancel{+},\cancel{\leq}}$ we denote the sub-vocabulary where the crossed-out symbols are not available. Note that even when addition is not available, we still allow writing numerals larger than 1.

We restrict ourselves to *universal sentences* over an SL vocabulary, with quantification only over the `Address` sort.

We now extend the Tarskian semantics of first-order logic to ensure that the sum constants of an SL vocabulary $(s_1, \ldots, s_m)$ are equal to the sum of outputs of their associated balance functions (b_j for each s_j) over the respective entire domains of sort `Address`.

Let Σ be an SL vocabulary. An SL structure $\mathcal{A} = (\mathcal{D}, \mathcal{I}) \in \mathrm{STRUCT}[\Sigma]$ representing a model for an SL formula φ is called an SL *model* iff

$$\mathcal{I}(s_j) = \sum_{a \in \mathcal{D}(\texttt{Address})} [\mathcal{I}(b_j)](a), \quad \text{for each } 1 \leq j \leq m. \qquad \text{(Sum Property)}$$

We write $\mathcal{A} \vDash_{\mathrm{SL}} \varphi$ to mean that $\mathcal{A}$ is an SL model of φ. When it is clear from context, we simply write $\mathcal{A} \vDash \varphi$.

Example 1 (Encoding ERC-20 in SL*).* As a use case of SL, we showcase the encoding of the ERC-20 token standard of the Ethereum community [**?**] in SL. To this end, we consider an SL vocabulary $\Sigma^{l,2,d}$. We respectively denote the balance functions and their associated sums as b, b', s, s' in the SL structure over $\Sigma^{l,2,d}$. The resulting instance of SL can then be used to encode ERC-20 operations/smart transitions as SL formulas, as shown in Table 1. Using this encoding, the post-condition of Figure 1 is expressed as the SL formula

$$b'(a) \approx b(a) + n \;\wedge\; \forall a' \not\approx a. b'(a') \approx b(a') \;\wedge\; s' \approx s + n \qquad (1)$$

formalizing the correctness of the smart transition of minting n tokens in Figure 1. In the applied verification examples in Section 6, rather than verifying the low-level implementation of built-in functions such as $\texttt{mint}_n$, we assume their correctness by including suitable axioms.

4 Decidability of SL

We consider the decidability problem of verifying formulas in SL. We show that when there are several function symbols b_j to sum over, the satisfiability problem for SL becomes undecidable[5]. We first present, however, a useful decidable fragment of SL.

4.1 A Decidable Fragment of SL

We prove decidability for a fragment of SL, which we call the $(l, 1, d)$-FRAG fragment of SL (Theorem 4). For doing so, we reduce the fragment to Presburger arithmetic, by using regular Presburger constructs to encode SL extensions, that is the uninterpreted functions and sum constants of SL.

The first step of our reduction proof is to consider distinct models, which are models where the `Address` constants a_i represent distinct elements in the domain $\mathcal{D}(\texttt{Address})$. While this restriction is somewhat unnatural, we show that for each vocabulary and formula that has a model, there exists an equisatisfiable formula over a different vocabulary that has a *distinct* model (Theorem 1). The crux of our decidability proof is then proving that $(l, 1, d)$-FRAG has *small* `Address` *space*: given a formula φ, if it is satisfiable, then there exists a model where $|\mathcal{D}(\texttt{Address})| \leq \kappa(|\varphi|)$, $|\varphi|$ is the length of φ, and $\kappa(.)$ is some computable function (Theorem 3)[6].

Distinct Models An SL structure $\mathcal{A}$ is considered *distinct* when the l `Address` constants represent l distinct elements in $\mathcal{D}(\texttt{Address})$. I.e.,

$$|\{\mathcal{I}(a_1), \ldots, \mathcal{I}(a_l)\}| = l\,.$$

Since each SL model induces an equivalence relation over the `Address` constants, we consider partitions P over $\{a_1, \ldots, a_l\}$. For each possible partition P we define a transformation of terms and formulas $\mathcal{T}_P$ that substitutes equivalent `Address` constants with a single `Address` constant. The resulting formulas are defined over a vocabulary that has $|P|$ `Address` constants. We show that given an SL formula φ, if φ has a model, we can always find a partition P such that each of its classes corresponds to an equivalence class induced by that model.

Theorem 1 (Distinct Models). *Let φ be an* SL *formula over Σ, then φ has a model iff there exists a partition P of $\{a_1, \ldots, a_l\}$ such that $\mathcal{T}_P(\varphi)$ has a* distinct *model.* □

Small `Address` Space In order to construct a reduction to Presburger arithmetic, we bound the size of the `Address` sort. For a fragment of SL to be decidable, we therefore need a way to bound its models upfront. We formalize this requirement as follows.

Definition 2 (Small `Address` Space). *Let* FRAG *be some fragment of SL over vocabulary $\Sigma = \Sigma^{l,m,d}_{+,\leq}$.* FRAG *is said to have* small `Address` space *if there exists a computable function $\kappa_\Sigma(.)$, such that for any* SL *formula $\varphi \in$* FRAG*, φ has a distinct model iff φ has a distinct model $\mathcal{A} = (\mathcal{D}, \mathcal{I})$ with* small `Address` space*, where $|\mathcal{D}(\texttt{Address})| \leq \kappa_\Sigma(|\varphi|)$.*

We call $\kappa_\Sigma(.)$ the bound function *of* FRAG*; when the vocabulary is clear from context we simply write $\kappa(.)$.*

[5] Proofs of our results are given in the appendix of [**?**].

[6] The function $\kappa(.)$ is defined per decidable fragment of SL, and not per formula.

One instance of a fragment (or rather, family of fragments) that satisfies this property is the $(l, 1, d)$-FRAG fragment: the simple case of a *single* uninterpreted "balance" function (and its associated sum constant), further restricted by removing the binary function $+$ and the binary relation $\leq$. Therefore, we derive the following theorem:

Theorem 2 (Small Address Space of $(l, 1, d)$-FRAG).
For any l, d, it holds $(l, 1, d)$-FRAG, the fragment of SL *formulas over the* SL *vocabulary*

$$\Sigma^{l,1,d}_{\not+,\not\leq} = \left(a_1, \ldots, a_l, b^1, c_1, \ldots, c_d, s, 0, 1\right) ,$$

has small Address space *with bound function $\kappa(x) = l + x + 1$.* □

An attempt to trivially extend Theorem 2 for a fragment of SL with two balance functions falls apart in a few places, but most importantly when comparing balances to the sum of a different balance function. In Section 4.2 we show that these comparisons are essential for proving our undecidability result in SL.

Presburger Reduction For showing decidability of some FRAG fragment of SL, we describe a Turing reduction to pure Presburger arithmetic. We introduce a transformation $\tau(.)$ of formulas in SL into formulas in Presburger arithmetic. It maps universal quantifiers to disjunctions, and sums to explicit addition of all balances. In addition, we define an auxiliary formula $\eta(\varphi)$, which ensures only valid addresses are considered, and that invalid addresses have zero balances. The formal definitions of $\tau(.)$ and $\eta(\varphi)$ can be found in [**?**].

By relying on the properties of *distinctness* and *small* Address *space* we get the following results.

Theorem 3 (Presburger Reduction). *An* SL *formula φ has a* distinct*,* SL *model with small* Address *space iff $\tau(\varphi) \wedge \eta(\varphi)$ has a Standard Model of Arithmetic.* □

Theorem 4 (SL Decidability). *Let* FRAG *be a fragment of* SL *that has* small Address space*, as defined in Definition 2. Then,* FRAG *is decidable.*

Proof (Theorem 4). Let φ be a formula in FRAG. Then φ has an SL model iff for some partition P of $\{a_1, \ldots, a_l\}$, $\mathcal{T}_P(\varphi)$ has a *distinct* SL model. For any P, the formula $\mathcal{T}_P(\varphi)$ is in FRAG, therefore $\mathcal{T}_P(\varphi)$ has a *distinct* SL model iff it has a *distinct* SL model with *small* Address *space*.

From Theorem 3, we get that for any P, $\varphi_P \triangleq \mathcal{T}_P(\varphi)$ has a *distinct* SL model iff $\tau(\varphi_P) \wedge \eta(\varphi_P)$ has a Standard Model of Arithmetic. By using the PA decision procedure as an oracle, we obtain the following *decision procedure for a* FRAG *formula φ*:

- For each possible partition P of $\{a_1, \ldots, a_l\}$, let $\varphi_P = \mathcal{T}_P(\varphi)$;
- Using a PA decision procedure, check whether $\tau(\varphi_P) \wedge \eta(\varphi_P)$ has a model, for each P;

	`Address`	l(`Address`)	c(`Address`)	g(`Address`)
Time-step #0		0	1	0
		1	1	c_1 at #0
		2	1	c_2 at #0
	a_0	3	1	PC at $\#0 = 1$
	$\vdots$	$\vdots$	$\vdots$	$\vdots$
Time-step $\#i$	x_1	$4i$	1	0
	x_2	$4i+1$	1	c_1 at $\#i$
	x_3	$4i+2$	1	c_2 at $\#i$
	x_4	$4i+3$	1	PC at $\#i$
Time-step $\#(i+1)$	x_5	$4i+4$	1	0
	x_6	$4i+5$	1	c_1 at $\#(i+1)$
	x_7	$4i+6$	1	c_2 at $\#(i+1)$
	x_8	$4i+7$	1	PC at $\#(i+1)$
	$\vdots$	$\vdots$	$\vdots$	$\vdots$
Time-step $\#n = \frac{s_c}{4} - 1$		$s_c - 4$	1	0
		$s_c - 3$	1	c_1 at $\#n$
		$s_c - 2$	1	c_2 at $\#n$
	a_1	$s_c - 1$	1	PC at $\#n = H$

Table 2: Transition System of a 2-Counter Machine, Array View.

- If a model for some partition P was found, the formula φ_P has a *distinct* SL model, and therefore φ has SL model;
- Otherwise, there is no *distinct* SL model for any partition P, and therefore there is no SL model for φ.

Remark 1. Our decision procedure for Theorem 4 requires B_l Presburger queries, where B_l is Bell's number for all possible partitions of a set of size l.

Using Theorem 4 and Theorem 2, we then obtain the following result.

Corollary 1. $(l, 1, d)$-FRAG *is decidable.* □

4.2 SL Undecidability

We now show that simple extensions of our decidable $(l, 1, d)$-FRAG fragment lose its decidability (Theorem 5). For doing so, we encode the halting problem of a two-counter machine using SL with 3 balance functions, thereby proving that the resulting SL fragment is undecidable.

Consider a two-counter machine, whose transitions are encoded by the Presburger formula $\pi(c_1, c_2, p, c'_1, c'_2, p')$ with 6 free variables: 2 for each of the three registers, one of which being the program counter (PC). We assume w.l.o.g. that all three registers are within $\mathbb{N}^+$, allowing us to use addresses with a zero balance as a special "separator". In addition, we assume that the program counter is 1 at the start of the execution, and that there exists a single halting statement at line H. That is, the two-counter machine halts iff the PC is equal to H.

Reduction Setting We have 4 `Address` elements for each time-step, 3 of them hold one register each, and one is used to separate between each group of `Address` elements (see Table 2). We have 3 uninterpreted functions from `Address` to `Nat` ("balances"). For readability we denote these functions as c, l, g (instead of b_1, b_2, b_3) and their respective sums as s_c, s_l, s_g:

1. Function c: Cardinality function, used to force size constraints. We set its value for all addresses to be 1, and therefore the number of addresses is s_c.
2. Function l: Labeling function, to order the time-steps. We choose one element to have a maximal value of $s_c - 1$ and ensure that l is injective. This means that the values of l are distinctly $[0, s_c - 1]$.
3. Function g: General purpose function, which holds either one of the registers or 0 to mark the `Address` element as a separating one.

Each group representing a time-step is a 4 `Address` element, ordered as follows:

1. First, a separating `Address` element x (where $g(x)$ is 0).
2. Then, the two general-purpose counters.
3. Lastly, the program counter.

In addition we have 2 `Address` constants, a_0 and a_1 which represent the PC value at the start and at the end of the execution. The element a_1 also holds the maximal value of l, that is, $l(a_1) + 1 \approx s_c$. Further, a_0 holds the fourth-minimal value, since its the last element of the first group, and each group has four elements.

Formalization Using a Two-Counter Machine We now formalize our reduction, proving undecidability of SL.
(i) We impose an injective labeling

$$\varphi_1 = \forall x, y. \, (l(x) \approx l(y)) \rightarrow (x \approx y)$$

(ii) We next formalize properties over the program counter PC. The `Address` constant that represents the program counter PC value of the last time-step is set to have the maximal labeling, that is

$$\varphi_2 = \forall x. l(x) \leq l(a_1)$$

Further, the `Address` constant that represents the PC value of the first time-step has the fourth labeling, hence

$$\varphi_3 = l(a_0) \approx 3$$

Finally, the first and last values of the program counter are respectively 1 and H, that is

$$\varphi_4 = g(a_0) \approx 1 \wedge g(a_1) \approx H$$

(iii) We express *cardinality constraints* ensuring that there are as many `Address` elements as the labeling of the last `Address` constant $(a_1) + 1$. We assert

$$\varphi_5 = (s_c \approx l(a_1) + 1) \wedge \forall x. \, (c(x) \approx 1)$$

(iv) We encode the transitions of the two-counter machine, as follows. For every 8 `Address` elements, if they represent two sequential time-steps, then the formula for the transitions of the two-counter machine is valid for the registers it holds. As such, we have

$$\begin{aligned}\varphi_6 = \forall x_1, \dots, x_8.\, &(F1 \land F2 \land F3) \\ &\to \pi\left(g(x_2), g(x_3), g(x_4), g(x_6), g(x_7), g(x_8)\right)\end{aligned}$$

where the conjunction $F1 \land F2 \land F3$ expresses that $x_1, \dots, x_8$ are two sequential time-steps, with $F1$, $F2$ and $F3$ defined as below. In particular, $F1$, $F2$ and $F3$ formalize that $x_1, \dots, x_8$ have sequential labeling, starting with one zero-valued `Address` element ("separator") and continuing with 3 non-zero elements, as follows:

- Sequential: $$l(x_2) \approx l(x_1) + 1 \land \dots \land l(x_8) \approx l(x_7) + 1 \tag{F1}$$
- Time-steps: $$g(x_1) \approx 0 \land g(x_2) > 0 \land g(x_3) > 0 \land g(x_4) > 0 \;, \tag{F2}$$ $$g(x_5) \approx 0 \land g(x_6) > 0 \land g(x_7) > 0 \land g(x_8) > 0 \tag{F3}$$

Based on the above formalization, the formula $\varphi = \varphi_1 \land \dots \land \varphi_6$ is satisfiable iff the two-counter machine halts within a finite amount of time-steps (and the exact amount would be given by $\frac{s_c}{4}$). Since the halting problem for two-counter machines is undecidable, our SL, already with 3 uninterpreted functions and their associated sums, is also undecidable.

Theorem 5. *For any $l \geq 2, m \geq 3$ and d, any fragment of* SL *over $\Sigma^{l,m,d}_{+,\leq}$ is undecidable.* □

Remark 2. Note that in the above formalization the only use of associated sums comes from expressing the size of the set of `Address` elements. As for our uninterpreted function $c(.)$ we have $\forall x.c(x) \approx 1$, its sum s_c is thus the amount of addresses. Hence, we can encode the halting problem for two-counter machines in an almost identical way to the encoding presented here, using a generalization of PA with two uninterpreted functions for $l(.)$ and $g(.)$, and a *size operation* replacing $c(.)$ and its associated sum.

5 SL Encodings of Smart Transitions

The definition of SL models in Sections 3 and 4 ensured that the summation constants s_j were respectively equal to the actual summation of all balances $b_j(.)$. In this section, we address the challenge to formalize relations between s_j and $b_j(.)$ in a way that the resulting encodings can be expressed in the logical frameworks of automated reasoners, in particular of SMT solvers and first-order theorem provers.

In what follows, we consider a single transaction or one time-step of multiple transactions over $s_j, b_j(.)$. We refer to such transitions as *smart transitions*. Smart transitions are common in smart contracts, expressing for example the minting and/or transferring of some coins, as evidenced in Figure 1 and discussed later.

Based on Section 3, our smart transitions are encoded in the $\Sigma^{l,2,d}$ fragment of SL. Note however, that neither decidability nor undecidability of this fragment is implied by Theorem 4, nor Theorem 5. In this section, we show that our SL encoding of smart transitions is expressible in first-order logic. We first introduce a sound, *implicit* SL *encoding*, by "hiding" away sum semantics and using invariant relations over smart transitions (Section 5.1). This encoding does not allow us to directly assert the values of any balance or sum, but we can prove that this implicit encoding is complete, relative to a translation function (Section 5.2).

By further restricting our implicit SL encoding to this relative complete setting, we consider counting properties to explicitly reason with balances and directly express verification conditions with unbounded sums on s_j and $b_j(.)$. This is shown in Section 5.3, and we evaluate different variants of the *explicit* SL *encoding* in Section 6, showcasing their practical use and relevance within automated reasoning.

To directly present our SL encodings and results in the smart contract domain, in what follows we rely on the notation of Table 1. As such, we respectively denote b, b' by `old-bal`, `new-bal` and write `old-sum`, `new-sum` for s, s'. As already discussed in Figure 1, the prefixes `old-` and `new-` refer to the entire state expressed in the encoding before and after the smart transition. We explicitly indicate this state using `old-world`, `new-world` respectively. The non-prefixed versions `bal` and `sum` are stand-ins for *both* the `old-` and `new-` versions — Figure 2 illustrates our setting for the smart transition of minting one coin.

With this SL notation at hand, we are thus interested in finding first-order formulas that verify smart transition relations between `old-sum` and `new-sum`, given the relation between `old-bal` and `new-bal`. In this paper, we mainly focus on the smart transitions of minting and transferring money, yet our results could be used in the context of other financial transactions/software transitions over unbounded sums.

Example 2. In the case of minting n coins in Figure 1, we require formulas that (a) describe the state before the transition (the `old-world`, thus pre-condition), (b) formalize the transition (the relation between `old-bal` and `new-bal`; (i)-(ii) in Figure 1) and (c) imply the consequences for the `new-world` ((iii) in Figure 1). These formulas verify that minting and depositing n coins into some address result in an increase of the sum by n, that is $\texttt{new-sum} = \texttt{old-sum} + n$, as expressed in the functional correctness formula (1) of Figure 1.

5.1 SL Encoding using Implicit Balances and Sums

The first encoding we present is a set of first-order formulas with equality over sorts {`Coin`, `Address`}. No additional theories are considered. The `Coin` sort represents money, where one coin is one unit of money. The `Address` sort represents the account addresses as before. As a consequence, balance functions and sum constants only exist implicitly in this encoding. As such, the property $\texttt{sum} = \sum_{a \in \texttt{Address}} \texttt{bal}(a)$ *cannot be directly expressed in this encoding.* Instead,

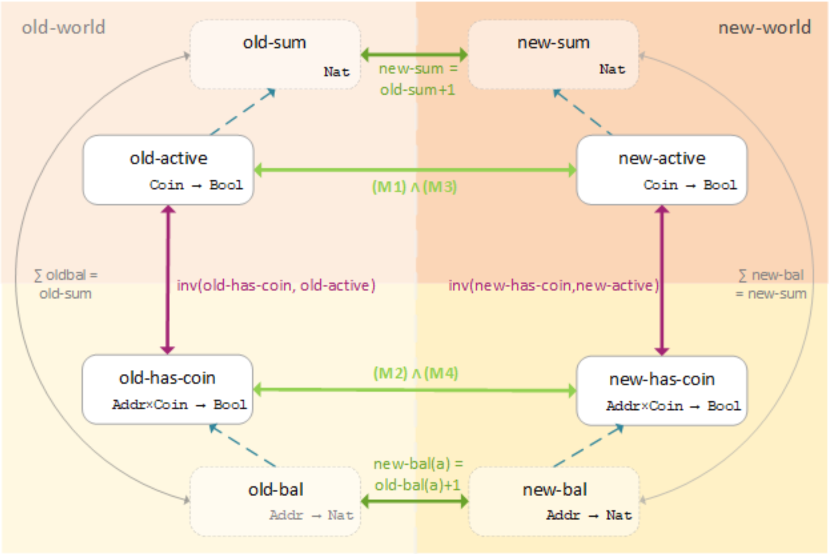

Fig. 2: Implicit SL Encoding of $\texttt{mint}_1$, where `Addr` is short for `Address`.

we formalize this property by using so-called *smart invariant* relations between two predicates `has-coin` and `active` over coins $c \in \texttt{Coin}$ and $a \in \texttt{Address}$, as follows.

Definition 3 (Smart Invariants). *Let* $\texttt{has-coin} \subseteq \texttt{Address} \times \texttt{Coin}$ *and consider* $\texttt{active} \subseteq \texttt{Coin}$*. A* smart invariant *of the pair* $(\texttt{has-coin}, \texttt{active})$ *is the conjunction of the following three formulas*

1. *Only active coins c can be owned by an address a:*

$$\forall c : \texttt{Coin}.\ \exists a : \texttt{Address}.\ \texttt{has-coin}(a, c) \to \texttt{active}(c)\ . \tag{I1}$$

2. *Every active coin c belongs to some address a:*

$$\forall c : \texttt{Coin}.\ \texttt{active}(c) \to \exists a : \texttt{Address}.\ \texttt{has-coin}(a, c)\ . \tag{I2}$$

3. *Every coin c belongs to at most one address a:*

$$\begin{aligned}&\forall c : \texttt{Coin}.\forall a, a' : \texttt{Address}.\\ &\qquad (\texttt{has-coin}(a, c) \land \texttt{has-coin}(a', c) \to a \approx a')\ .\end{aligned} \tag{I3}$$

We write $\texttt{inv}(\texttt{has-coin}, \texttt{active})$ *to denote the smart invariant (I1)∧(I2)∧(I3) of* $(\texttt{has-coin}, \texttt{active})$ *.*

Intuitively, our *smart invariants* ensure that a coin c is *active* iff it is *owned* by precisely one address a. Our smart invariants imply the soundness of our implicit SL encoding, as follows.

Theorem 6 (Soundness of SL Encoding). *Given that* $\texttt{sum} = |\texttt{active}|$ *and for every* $a \in \texttt{Address}$ *it holds* $\texttt{bal}(a) = |\{c \in \texttt{Coin} \mid (a,c) \in \texttt{has-coin}\}|$*, then* $\texttt{inv}(\texttt{has-coin}, \texttt{active}) \implies \texttt{sum} = \sum_{a \in \texttt{Address}} \texttt{bal}(a)$. □

We say that a *smart transition preserves smart invariants*, when

$$\begin{aligned}&\texttt{inv}(\texttt{old-has-coin}, \texttt{old-active})\\ &\qquad \iff \texttt{inv}(\texttt{new-has-coin}, \texttt{new-active}),\end{aligned}$$

where $\texttt{old-has-coin}, \texttt{old-active}$ and $\texttt{new-has-coin}, \texttt{new-active}$ respectively denote the functions $\texttt{has-coin}, \texttt{active}$ in the states before and after the smart transition. Based on the soundness of our implicit SL encoding, we formalize smart transitions preserving smart invariants as first-order formulas. We only discuss smart transitions implementing minting n coins here, but other transitions, such as transferring coins, can be handled in a similar manner. We first focus on miniting a single coin, as follows.

Definition 4 (Transition $\texttt{mint}_1(a,c)$). *Let there be* $c \in \texttt{Coin}, a \in \texttt{Address}$*. The transition* $\texttt{mint}_1(a,c)$ *activates coin* c *and deposits it into address* a*.*

1. *The coin* c *was inactive before and is active now:*
$$\neg\texttt{old-active}(c) \land \texttt{new-active}(c)\ . \tag{M1}$$
2. *The address* a *owns the new coin* c*:*
$$\texttt{new-has-coin}(a,c) \land \forall a' : \texttt{Address}.\ \neg\texttt{old-has-coin}(a',c)\ . \tag{M2}$$
3. *Everything else stays the same:*
$$\forall c' : \texttt{Coin}.\ c' \not\approx c \to (\texttt{new-active}(c') \leftrightarrow \texttt{old-active}(c'))\ , \tag{M3}$$
$$\begin{aligned}&\forall c' : \texttt{Coin}.\ \forall a' : \texttt{Address}.\ (c' \not\approx c \lor a' \not\approx a) \to\\ &\qquad (\texttt{new-has-coin}(a',c') \leftrightarrow \texttt{old-has-coin}(a',c'))\ .\end{aligned} \tag{M4}$$

The transition $\texttt{mint}_1(a,c)$ *is defined as (M1)* $\land$ *(M2)* $\land$ *(M3)* $\land$ *(M4).*

By minting one coin, the balance of precisely one address, that is of the receiver's address, increases by one, whereas all other balances remain unchanged. Thus, the expected impact on the sum of account balances is also increased by one, as illustrated in Figure 2. The following theorem proves that the definition of $\texttt{mint}_1$ is *sound*. That is, $\texttt{mint}_1$ affects the implicit balances and sums as expected and hence $\texttt{mint}_1$ preserves smart invariants.

Theorem 7 (Soundness of $\texttt{mint}_1(a,c)$). *Let* $c \in \texttt{Coin}$*,* $a \in \texttt{Address}$ *such that* $\texttt{mint}_1(a,c)$*. Consider balance functions* $\texttt{old-bal}, \texttt{new-bal} : \texttt{Address} \to \mathbb{N}$*, non-negative integer constants* $\texttt{old-sum}, \texttt{new-sum}$*, unary predicates* $\texttt{old-active}, \texttt{new-active} \subseteq \texttt{Coin}$ *and binary predicates* $\texttt{old-has-coin}, \texttt{new-has-coin} \subseteq \texttt{Address} \times \texttt{Coin}$ *such that*

$$|\texttt{old-active}| = \texttt{old-sum}\ ,\ |\texttt{new-active}| = \texttt{new-sum},$$

and for every address a', we have

$$\begin{aligned}\texttt{old-bal}(a') &= |\{c' \in \texttt{Coin} \mid (a', c') \in \texttt{old-has-coin}\}| \ ,\\ \texttt{new-bal}(a') &= |\{c' \in \texttt{Coin} \mid (a', c') \in \texttt{new-has-coin}\}| \ .\end{aligned}$$

Then, $\texttt{new-sum} = \texttt{old-sum} + 1$*,* $\texttt{new-bal}(a) = \texttt{old-bal}(a) + 1$*. Moreover, for all other addresses $a' \neq a$, it holds* $\texttt{new-bal}(a') = \texttt{old-bal}(a')$*.* □

Smart transitions minting an arbitrary number of n coins, as in our Figure 1, is then realized by repeating the $\texttt{mint}_1$ transition n times. Based on the soundness of $\texttt{mint}_1$, ensuring that $\texttt{mint}_1$ preserves smart invariants, we conclude by induction that n repetitions of $\texttt{mint}_1$, that is *minting n coins, also preserves smart invariants.* The precise definition of $\texttt{mint}_n$ together with the soundness result is stated in [?].

5.2 Completeness Relative to a Translation Function

Smart invariants provide sufficient conditions for ensuring soundness of our SL encodings (Theorem 6). We next show that, under additional constraints, smart invariants are also necessary conditions, establishing thus *(relative) completeness of our encodings.*

A straightforward extension of Theorem 6 however does not hold. Namely, only under the assumptions of Theorem 6, the following formula is not valid:

$$\texttt{sum} = \sum_{a \in \texttt{Address}} \texttt{bal}(a) \quad \Longleftrightarrow \quad \texttt{inv}(\texttt{has-coin}, \ \texttt{active}) \, .$$

As a counterexample, assume (i) $\texttt{sum} = |\texttt{active}|$, (ii) for every $a \in \texttt{Address}$ it holds that $\texttt{bal}(a) = |\{c \in \texttt{Coin} \mid (a, c) \in \texttt{has-coin}\}|$, that is the assumptions of Theorem 6. Further, let (iii) the smart invariants $\texttt{inv}(\texttt{has-coin}, \texttt{active})$ hold for all but the coins $c_1, c_2 \in \texttt{Coin}$ and all but the addresses $a_1, a_2 \in \texttt{Address}$. We also assume that (iv) c_1 is active but not owned by any address and (v) c_2 is active and owned by the two distinct addresses a_1, a_2. We thus have $\texttt{sum} = \sum_{a \in \texttt{Address}} \texttt{bal}(a)$, yet $\texttt{inv}(\texttt{has-coin}, \ \texttt{active})$ does not hold.

To ensure completeness of our encodings, we therefore introduce a translation function f that restricts the set $\mathcal{F} \triangleq 2^{\texttt{Address} \times \texttt{Coin}} \times 2^{\texttt{Coin}}$ of $(\texttt{has-coin}, \texttt{active})$ pairs, as follows. We exclude from $\mathcal{F}$ those pairs $(\texttt{has-coin}, \texttt{active})$ that violate smart invariants by both (i) not satisfying (I2), as (I2) ensures that there are not too many active coins, and by (ii) not satisfying at least one of (I1) and (I3), as (I1) and (I3) ensure that there are not too few active coins. The required translation function f (as in [?]) now assigns every pair $(\texttt{bal}, \texttt{sum})$ the set of all $(\texttt{has-coin}, \texttt{active}) \in \mathcal{F}$ that satisfy $\texttt{sum} = |\texttt{active}|$, $\texttt{bal}(a) = |\{c \in \texttt{Coin} \mid \texttt{has-coin}(a, c)\}|$ for every address a and have not been excluded.

Theorem 8 (Relative Completeness of SL Encoding). *Let* $(\texttt{bal}, \texttt{sum}) \in \mathbb{N}^{\texttt{Address}} \times \mathbb{N}$ *and let* $(\texttt{has-coin}, \texttt{active}) \in f(\texttt{bal}, \texttt{sum})$ *be arbitrary. Then,*

$$\texttt{sum} = \sum_{a \in \texttt{Address}} \texttt{bal}(a) \quad \Longleftrightarrow \quad \texttt{inv}\,(\texttt{has-coin}, \ \texttt{active}) \, . \qquad \square$$

5.3 SL Encodings using Explicit Balances and Sums

We now restrict our SL encoding from Section 5.1 to explicitly reason with balance functions during smart transitions. We do so by expressing our translation function f from Section 5.2 in first-order logic. We now use the summation constant $\texttt{sum} \in \mathbb{N}$ and the balance function $\texttt{bal} : \texttt{Address} \to \mathbb{N}$ in our SL encoding. In particular, we use our smart invariants $\texttt{inv}(\texttt{has-coin}, \texttt{active})$ in this explicit SL encoding together with two additional axioms (Ax1, Ax2), ensuring that $\texttt{sum} = |\texttt{active}|$ and $\texttt{bal}(a) = |\{c \in \texttt{Coin} \mid \texttt{has-coin}(a, c)\}|$ for all $a \in \texttt{Address}$.

To formalize the additional properties, we introduce two counting mechanisms in our SL encoding. The first one is a bijective function $\texttt{count} : \texttt{Coin} \to \mathbb{N}^+$ and the second one is a function $\texttt{idx} : \texttt{Address} \times \texttt{Coin} \to \mathbb{N}^+$, where $\texttt{idx}(a, .) : \texttt{Coin} \to \mathbb{N}^+$ is bijective for every $a \in \texttt{Address}$. To ensure that $\texttt{count}$ and $\texttt{idx}(a, .)$ count coins, we impose the following two properties:

$$\forall c : \texttt{Coin}.\, \texttt{active}(c) \iff \texttt{count}(c) \leq \texttt{sum}\ , \tag{Ax1}$$

$$\forall c : \texttt{Coin}.\, \forall a : \texttt{Address}.\, \texttt{has-coin}(a, c) \iff \texttt{idx}(a, c) \leq \texttt{bal}(a)\ . \tag{Ax2}$$

Figure 3 illustrates our revised SL encoding for our smart transition $\texttt{mint}_1$. We next ensure soundness of our resulting explicit encoding for summation, as follows.

Theorem 9 (Soundness of Explicit SL Encodings). *Let there be a pair* $(\texttt{bal}, \texttt{sum}) \in \mathbb{N}^{\texttt{Address}} \times \mathbb{N}$*, a pair* $(\texttt{has-coin}, \texttt{active}) \in \mathcal{F}$*, and functions* $\texttt{count} : \texttt{Coin} \to \mathbb{N}^+$ *and* $\texttt{idx} : \texttt{Address} \times \texttt{Coin} \to \mathbb{N}^+$*.*

Given that $\texttt{count}$ *is bijective,* $\texttt{idx}(a, .) : \texttt{Coin} \to \mathbb{N}^+$ *is bijective for every* $a \in \texttt{Address}$*, and that (Ax1), (Ax2) and* $\texttt{inv}\,(\texttt{has-coin}, \texttt{active})$ *hold, then,* $\texttt{sum} = |\texttt{active}|$ *and* $\texttt{bal}(a) = |\{c \in \texttt{Coin} : \texttt{has-coin}(a, c)\}|$*, for every* $a \in \texttt{Address}$*.*

In particular, we have $\texttt{sum} = \sum_{a \in \texttt{Address}} \texttt{bal}(a)$*.* □

When compared to Section 5.1, our explicit SL encoding introduced above uses our smart invariants as axioms of our encoding, together with (Ax1) and (Ax2). In our explicit SL encoding, the post-conditions asserting functional correctness of smart transitions express thus relations among $\texttt{old-sum}$ to $\texttt{new-sum}$. For example, for $\texttt{mint}_n$ we are interested in ensuring

$$\texttt{mint}_n \Rightarrow \texttt{new-sum} = \texttt{old-sum} + n\ . \tag{2}$$

By using two new constants $\texttt{old-total}, \texttt{new-total} \in \mathbb{N}$, we can use $\texttt{sum} = \texttt{total}$ as smart invariant for $\texttt{mint}_n$. As a result, the property to be ensured is then

$$\begin{aligned}&(\texttt{old-sum} = \texttt{old-total} \wedge \texttt{new-total} = \texttt{old-total} + n \wedge \texttt{mint}_n)\\ &\Rightarrow (\texttt{new-sum} = \texttt{new-total})\ .\end{aligned} \tag{3}$$

It is easy to see that the negations of (2) and (3) are equisatisfiable. We note however that the additional constants $\texttt{old-total}$, $\texttt{new-total}$ used in (3) lead to unstable results within automated reasoners, as discussed in Section 6.

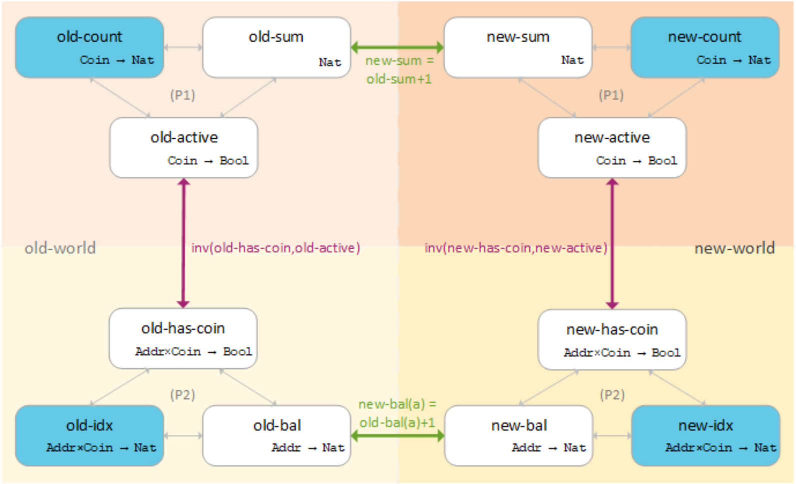

Fig. 3: Explicit SL Encoding of $\texttt{mint}_1$, where `Addr` is short for `Address`.

6 Experiments

From Theory to Practice. To make our explicit SL encodings handier for automated reasoners, we improved the setting illustrated in Figure 3 by applying the following restrictions without losing any generality.
(i) The predicates `has-coin` and `active` were removed from the explicit SL encodings, by replacing them by their equivalent expressions (Ax1)-(Ax2).
(ii) The surjectivity assertions of `count` and `idx` were restricted to the relevant intervals $[1, \texttt{sum}]$, $[1, \texttt{bal}(a)]$ respectively.
(iii) Compared to Figure 3, only one mutual `count` and one mutual `idx` functions were used. We however conclude that we do not lose expressivity of our resulting SL encoding, as shown in [**?**].
(iv) When our SL encoding contains expressions such as $\forall c : \texttt{Coin}.\ \texttt{idx}(a_0, c) \in [l_0, u_0] \iff \texttt{idx}(a_1, c) \in [l_1, u_1]$, with a_0, a_1 being distinct addresses such that either $u_i \leq \texttt{bal}(a_i)$ or $l_i > \texttt{bal}(a_i)$, $i \in \{0, 1\}$, then it can be assumed that the coins in those intervals are in the same order for both functions [**?**].

Based on the above, we derive three different explicit SL encodings to be used in automated reasoning about smart transitions. We respectively denote these explicit SL encodings by `int`, `nat` and `id`, and describe them next.
Benchmarks. In our experiments, we consider four smart transitions $\texttt{mint}_1$, $\texttt{mint}_n$, $\texttt{transferFrom}_1$ and $\texttt{transferFrom}_n$, respectively denoting minting and transferring one and n coins. These transitions capture the main operations of linear integer arithmetic. In particular, $\texttt{mint}_n$ implements the smart transition of our running example from Figure 1.

For each of the four smart transitions, we implement four SL encodings: the implicit SL encoding `uf` from Section 5.1 using only uninterpreted functions and three explicit encodings `int`, `nat` and `id` as variants of Section 5.3. We

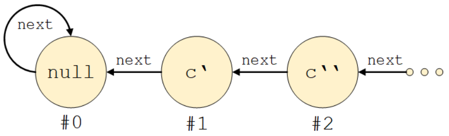

Fig. 4: Linked Lists in id.

also consider three additional arithmetic benchmarks using int, which are not directly motivated by smart contracts. Together with variants of int and nat presented in the sequel, our benchmark set contains 31 examples altogether, with each example being formalized in the SMT-LIB input syntax [?]. In addition to our encodings, we also proved consistency of the axioms used in our encodings.

SL Encodings and Relaxations. Our explicit SL encoding int uses linear integer arithmetic, whereas nat and id are based on natural numbers. As naturals are not a built-in theory in SMT-LIB, we assert the axioms of Presburger arithmetic directly in the encodings of nat and id.

In our id encodings, inductive datatypes are additionally used to order coins. There exists one linked list of all coins for count and one for each $\texttt{idx}(a, .)$, $a \in \texttt{Address}$. Additionally, there exists a "null" coin, which is the first element of every list and is not owned by any address. As shown in Figure 4, the numbering of each coin is defined by its position in the respective list. This way surjectivity for count and idx can respectively be asserted by the formulas $\exists c : \texttt{Coin}.\ \texttt{count}(c) \approx \texttt{sum}$ and $\forall a : \texttt{Address}.\ \exists c : \texttt{Coin}.\ \texttt{idx}(a, c) \approx \texttt{bal}(a)$. However, asserting surjectivity for int and nat cannot be achieved without quantifying over $\mathbb{N}^+$. Such quantification would drastically effect the performance of automated reasoners in (fragments of) first-order logics. As a remedy, within the default encodings of int and nat, we only consider relevant instances of surjectivity.

Further, we consider variations of int and nat by asserting proper surjectivity to the relevant intervals of idx and count (denoted as *surj*) and/or adding the total constants mentioned in Section 5.3 (denoted as *with* total, *no* total) . These variations of int and nat are implemented for $\texttt{mint}_1$ and $\texttt{transferFrom}_1$.

Experimental Setting. We evaluated our benchmark set of 31 examples using SMT solvers Z3 [?] and CVC4 [?], as well as the first-order theorem prover Vampire [?]. Our experiments were run on a standard machine with an Intel Core i5-6200U CPU (2.30GHz, 2.40GHz) and 8 GB RAM. The time is given in seconds and we ran all experiments with a time limit of 300s. Time out is indicated by the symbol ×. The default parameters were used for each solver, unless stated otherwise in the corresponding tables[7].

Experimental Analysis. We first report on our experiments using different variations of int and nat. Table 3 shows that asserting complete surjectivity for int and nat is computationally hard and indeed significantly effects the performance of automated reasoners. Thus, for the following experiments only relevant

[7] The precise calls and encodings are available at github.com/SoRaTu/SmartSums.

$\texttt{mint}_1$				$\texttt{transferFrom}_1$			
no `total`	**Z3**	**CVC4**	**Vampire**	no `total`	**Z3**	**CVC4**	**Vampire**
`nat`	0.02	×	0.92	`nat`	×	×	15.35
`nat` surj.	×	×	×	`nat` surj.	100.03	×	×
`int`	0.02	0.03	×	`int`	0.02	0.07	×
`int` surj.	×	5.96	×	`int` surj.	1.02	×	×
with `total`	**Z3**	**CVC4**	**Vampire**	with `total`	**Z3**	**CVC4**	**Vampire**
`nat`	0.03	×	2.92	`nat`	0.28	×	22.54
`nat` surj.	0.11	×	×	`nat` surj.	38.24	×	×
`int`	0.02	0.03	×	`int`	0.02	0.10	×
`int` surj.	3.81	5.95	×	`int` surj.	×	6.56	×

Table 3: Results of $\texttt{mint}_1$ and $\texttt{transferFrom}_1$ using `nat` and `int`, with/without the `total` Constants and with/without Surjectivity.

Encoding	Task			
	$\texttt{mint}_1$	$\texttt{transferFrom}_1$	$\texttt{mint}_n$	$\texttt{transferFrom}_n$
`uf`	Z3: 0.01 CVC4: 0.02 Vampire: 0.18	Z3: 0.02 CVC4: 0.03 Vampire: 0.19	Z3: × CVC4: × Vampire: 0.35*	Z3: × CVC4: × Vampire: 0.44*
`nat`	Z3: 0.02 CVC4: × Vampire: 0.92	Z3: × CVC4: × Vampire: 15.35	Z3: × CVC4: × Vampire: 23.23†	Z3: × CVC4: × Vampire: 228.22†
`int`	Z3: 0.02 CVC4: 0.03 Vampire: ×	Z3: 0.02 CVC4: 0.07 Vampire: ×	Z3: 0.03 CVC4: 0.05 Vampire: ×	Z3: 0.11 CVC4: 0.35 Vampire: ×
`id`	Z3: × CVC4: × Vampire: 7.36‡	Z3: × CVC4: × Vampire: 17.16‡	Z3: × CVC4: × Vampire: 23.52‡	Z3: × CVC4: × Vampire: ×

Table 4: Smart Transitions using Implicit/Explicit SL Encodings.

instances of surjectivity, such as $\exists c : \texttt{Coin.count}(c) = \texttt{sum}$ were asserted in `int` and `nat`. Table 3 also illustrates the instability of using the `total` constant. Some tasks seem to be easier even though their reasoning difficulty increased strictly by adding additional constants.

Our most important experimental findings are shown in Table 4, demonstrating that *our* SL *encodings are suitable for automated reasoners. Thanks to our explicit* SL *encodings, each solver can certify every smart transition in at least one encoding.* Our explicit SL encodings are more relevant than the implicit encoding `uf` as we can express and compare any two non-negative integer sums, whereas for `uf` handling arbitrary values n can only be done by iterating over the $\texttt{mint}_1$ (or $\texttt{transferFrom}_1$) transition. This iteration requires inductive reasoning, which currently only Vampire could do [**?**], as indicated by the superscript $*$. Nevertheless, the transactions $\texttt{mint}_1$, $\texttt{transferFrom}_1$, which involve only one coin in `uf`, require no inductive reasoning as the actual sum is not considered; each of our solvers can certify these examples.

We note that the tasks $\texttt{mint}_n$ and $\texttt{transferFrom}_n$ from Table 4 yield a huge search space when using their explicit SL encodings within automated reasoners.

Task		Time
Transition	Impact	
$\texttt{new-bal}(a_0) = \texttt{old-bal}(a_0) + 3$ $\texttt{new-bal}(a_1) = \texttt{old-bal}(a_1) - 3$	$\texttt{new-sum} = \texttt{old-sum}$	Z3: 0.20 CVC4: 1.28 Vampire: ×
$\texttt{new-bal}(a_0) = \texttt{old-bal}(a_0) + 4$ $\texttt{new-bal}(a_1) = \texttt{old-bal}(a_1) - 2$	$\texttt{new-sum} = \texttt{old-sum} + 2$	Z3: 0.58 CVC4: 7.14 Vampire: ×
$\texttt{new-bal}(a_0) = \texttt{old-bal}(a_0) + 5$ $\texttt{new-bal}(a_1) = \texttt{old-bal}(a_1) - 3$ $\texttt{new-bal}(a_2) = \texttt{old-bal}(a_2) - 1$	$\texttt{new-sum} = \texttt{old-sum} + 1$	Z3: 1.52 CVC4: 155.20 Vampire: ×

Table 5: Arithmetic Reasoning in the Explicit SL Encoding `int`.

We split these tasks into proving intermediate lemmas and proved each of these lemmas independently, by the respective solver. In particular, we used one lemma for $\texttt{mint}_n$ and four lemmas for $\texttt{transferFrom}_n$. In our experiments, we only used the recent theory reasoning framework of Vampire with split queues [?] and indicate our results in by superscript †.

We further remark that our explicit SL encoding `id` using inductive datatypes also requires inductive reasoning about smart transitions and beyond. The need of induction explains why SMT solvers failed proving our `id` benchmarks, as shown in Table 4. We note that Vampire found a proof using built-in induction [?] and theory-specific reasoning [?], as indicated by superscript ‡.

We conclude by showing the generality of our approach beyond smart transitions. It in fact enables fully automated reasoning about any two summations $\sum_{i \in I} g(i)$, $\sum_{i \in I} h(i)$ of non-negative integer values $g(i)$, $h(i)$ $(i \in I)$ over a mutual finite set I. The examples of Table 5 affirm this claim.

7 Related work

Smart Contract Safety. Formal verification of smart contracts is an emerging hot topic because of the value of the assets stored in smart contracts, e.g. the DeFi software [?]. Due to the nature of the blockchain, bugs in smart contracts are irreversible and thus the demand for provably bug-free smart contracts is high.

The K interactive framework has been used to verify safety of a smart contract, e.g. in [?]. Isabelle [?] was also shown to be useful in manual, interactive verification of smart contracts [?]. We, however, focus on automated approaches.

There are also efforts to perform deductive verification of smart contracts both on the source level in languages such as Solidity [?,?,?] and Move [?], as well as on the the Ethereum virtual machine (EVM) level [?,?]. This paper improves the effectiveness of these approaches by developing techniques for automatically reasoning about unbounded sums. This way, we believe we support a more semantic-based verification of smart contracts.

Our approach differs from works using ghost variables [?], since we do not manually update the “ghost state”. Instead, the verifier needs only to reason about the local changes, and the aggregate state is maintained by the axioms.

That means other approaches assume (a) the local changes and (b) the impact on ghost variables (sum), whereas we only assume (a) and automatically prove $a \Rightarrow b$. This way, we reduce the user-guidance in providing and proving (b).

Our work complements approaches that verify smart contracts as finite state machines [**?**] and methods, like ZEUS [**?**], using symbolic model checking and abstract interpretation to verify generic safety properties for smart contracts.

The work in [**?**] provides an extensive evaluation of ERC-20 and ERC-721 tokens. ERC-721 extends ERC-20 with ownership functions, one of which being "approve". It enables transactions on another party's behalf. This is independent of our ability to express sums in first-order logic, as the transaction's initiator is irrelevant to its effect.

Reasoning about Financial Applications. Recently, the Imandra prover introduced an automated reasoning framework for financial applications [**?**,**?**,**?**]. Similarly to our approach, these works use SMT procedures to verify and/or generate counter-examples to safety properties of low- and high-level algorithms. In particular, results of [**?**,**?**,**?**] include examples of verifying ranking orders in matching logics of exchanges, proving high-level properties such as transitivity and anti-symmetry of such orders. In contrast, we focus on verifying properties relating local changes in balances to changes of the global state (the sum). Moreover, our encodings enable automated reasoning both in SMT solving and first-order theorem proving.

Automated Aggregate Reasoning. The theory of first-order logic with aggregate operators has been thoroughly studied in [**?**,**?**]. Though proven to be strictly more expressive than first-order logic, both in the case of general aggregates as well as simple counting logics, in this paper we present a practical way to encode a weakened version of aggregates (specifically sums) in first-order logic. Our encoding (as in Section 5) works by expressing particular sums of interest, harnessing domain knowledge to avoid the need of general aggregate operators.

Previous works [**?**,**?**] in the field of higher-order reasoning do not directly discuss aggregates. The work of [**?**] extends Presburger arithmetic with Boolean algebra for finite, unbounded sets of uninterpreted elements. This includes a way to express the set cardinalities and to compare them against integer variables, but does not support uninterpreted functions, such as the balance functions we use throughout our approach.

The SMT-based framework of [**?**] takes a different, white-box approach, modifying the inner workings of SMT solvers to support higher-order logic. We on the other hand treat theorem provers and SMT solvers as black-boxes, constructing first-order formulas that are tailored to their capabilities. This allows us to use any off-the-shelf SMT solver.

In [**?**], an SMT module for the theory of FO(Agg) is presented, which can be used in all DPLL-based SAT, SMT and ASP solvers. However, FO(Agg) only provides a way to express functions that have sets or similar constructs as inputs, but not to verify their semantic behavior.

8 Conclusions

We present a methodology for reasoning about unbounded sums in the context of *smart transitions*, that is transitions that occur in smart contracts modeling transactions. Our sum logic SL and its usage of sum constants, instead of fully-fledged sum operators, turns out to be most appropriate for the setting of smart contracts. We show that SL has decidable fragments (Section 4.1), as well as undecidable ones (Section 4.2). Using two phases to first implicitly encode SL in first-order logic (Section 5.1), and then explicitly encode it (Section 5.3), allows us to use off-the-shelf automated reasoners in new ways, and automatically verify the semantic correctness of smart transitions.

Showing the (un)decidability of the SL fragment with two sets of uninterpreted functions and sums is an interesting step for further work, as this fragment supports encoding smart transition systems. Another interesting direction of future work is to apply our approach to different aggregates, such as minimum and maximum and to reason about under which conditions these values stay above/below certain thresholds. A slightly modified setting of our SL axioms can already handle min/max aggregates in a basic way, namely by using $\geq$ and $\leq$ instead of equality and dropping the injectivity/surjectivity (respectively) axioms of the counting mechanisms.

Summing upon multidimensional arrays in various ways is yet another direction of future research. Our approach supports the summation over all values in all dimensions by adding the required number of parameters to the predicate `idx` and by adapting the axioms accordingly.

Acknowledgements

We thank Petra Hozzová for fruitful discussions on our encodings and Sharon Shoham-Buchbinder for her insights and contributions to this paper. This work was partially funded by the ERC CoG ARTIST 101002685, the ERC StG SYMCAR 639270, the United States-Israel Binational Science Foundation (BSF) grant No. 2016260, Grant No. 1810/18 from the Israeli Science Foundation, Len Blavatnik and the Blavatnik Family foundation, the FWF grant LogiCS W1255-N23, the TU Wien DK SecInt and the Amazon ARA 2020 award FOREST.