Summing Up Smart Transitions

CAV2021

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

[1] Tel Aviv University
[2] TU Vienna
[3] UMass Amherst

July 20-23, 2021

Introduction

Introduction

- Smart Contracts: our initial motivation

Introduction

- Smart Contracts: our initial motivation
- Example: Transferring money

a_1, a_2: Address, n: Nat
transferFrom(a_1,a_2,n)
Post-conditions assert new-bal(a_1)=old-bal(a_1)-n and (i) new-bal(a_1)=old-bal(a_1)+n assert for each Address a≠a_1, a≠a_2: (ii) assert new-bal(a)=old-bal(a) assert new-sum()=old-sum() (iii)

Introduction

- Smart Contracts: our initial motivation
- Example: Transferring money

a_1, a_2: Address, n: Nat
transferFrom(a_1, a_2, n)
Post-conditions assert new-bal(a_1)=old-bal(a_1)-n and (i) new-bal(a_1)=old-bal(a_1)+n assert for each Address a≠a_1, a≠a_2: (ii) assert new-bal(a)=old-bal(a) assert new-sum()=old-sum() (iii)

- Challenging for automated reasoning

Introduction

- Smart Contracts: our initial motivation
- Example: Transferring money

a_1, a_2: Address, n: Nat
transferFrom(a_1,a_2,n)
Post-conditions assert new-bal(a_1)=old-bal(a_1)-n and (i) new-bal(a_1)=old-bal(a_1)+n assert for each Address a≠a_1, a≠a_2: (ii) assert new-bal(a)=old-bal(a) assert new-sum()=old-sum() (iii)

- Challenging for automated reasoning
- Abstract this *specific* setting away

Contributions

Contributions

- Theory:

Contributions

- Theory:
  - Introduce *Sum Logic*

Contributions

- Theory:
  - Introduce *Sum Logic*
  - Decidability results

Contributions

- Theory:
  - Introduce *Sum Logic*
  - Decidability results
- Practice:

Contributions

- Theory:
  - Introduce *Sum Logic*
  - Decidability results
- Practice:
  - Encodings of *Sum Logic* in first-order logic

Contributions

- Theory:
  - Introduce *Sum Logic*
  - Decidability results
- Practice:
  - Encodings of *Sum Logic* in first-order logic
  - Automated verification of 31 benchmarks

Sum Logic

Sum Logic

- PA + `Addr` sort + `bal` functions + semantic condition

Sum Logic

- PA + `Addr` sort + `bal` functions + semantic condition
- $$\left(\begin{array}{ll} & \mathtt{bal}'(a_1) \simeq \mathtt{bal}(a_1) - n \\ \wedge & \mathtt{bal}'(a_2) \simeq \mathtt{bal}(a_2) + n \\ \wedge & \forall a \not\simeq a_1, a_2 . \mathtt{bal}'(a) \simeq \mathtt{bal}(a) \end{array}\right) \rightarrow s' \simeq s$$

Sum Logic

- PA + `Addr` sort + `bal` functions + semantic condition
- $$\left(\begin{array}{ll} & \mathtt{bal}'(a_1) \simeq \mathtt{bal}(a_1) - n \\ \wedge & \mathtt{bal}'(a_2) \simeq \mathtt{bal}(a_2) + n \\ \wedge & \forall a \not\simeq a_1, a_2 . \mathtt{bal}'(a) \simeq \mathtt{bal}(a) \end{array}\right) \rightarrow s' \simeq s$$
- Theoretic results:

Sum Logic

- PA + `Addr` sort + `bal` functions + semantic condition
- $$\left(\begin{array}{ll} & \mathtt{bal}'(a_1) \simeq \mathtt{bal}(a_1) - n \\ \wedge & \mathtt{bal}'(a_2) \simeq \mathtt{bal}(a_2) + n \\ \wedge & \forall a \not\simeq a_1, a_2 . \mathtt{bal}'(a) \simeq \mathtt{bal}(a) \end{array}\right) \rightarrow s' \simeq s$$
- Theoretic results:
  - A small, one-`bal`-function decidable fragment

Sum Logic

- PA + `Addr` sort + `bal` functions + semantic condition
- $$\left(\begin{array}{ll} & \mathtt{bal}'(a_1) \simeq \mathtt{bal}(a_1) - n \\ \wedge & \mathtt{bal}'(a_2) \simeq \mathtt{bal}(a_2) + n \\ \wedge & \forall a \not\simeq a_1, a_2 . \mathtt{bal}'(a) \simeq \mathtt{bal}(a) \end{array}\right) \rightarrow s' \simeq s$$
- Theoretic results:
  - A small, one-`bal`-function decidable fragment
  - Undecidable already with 3 `bal` functions

Undecidability Sketch

Undecidability Sketch

We reduce the halting problem of a 2-counter machine to a satisfiability check of a Sum Logic formula:

Undecidability Sketch

We reduce the halting problem of a 2-counter machine to a satisfiability check of a Sum Logic formula:

- We use `Addr` elements to represent the time-steps of the machine.

Undecidability Sketch

We reduce the halting problem of a 2-counter machine to a satisfiability check of a Sum Logic formula:

- We use `Addr` elements to represent the time-steps of the machine.
- Different `bal` functions represent the content of the different registers.

Undecidability Sketch

We reduce the halting problem of a 2-counter machine to a satisfiability check of a Sum Logic formula:

- We use `Addr` elements to represent the time-steps of the machine.
- Different `bal` functions represent the content of the different registers.
- We use the sum as a way to count the number of time-steps.

Theoretical Gap

Our theoretical results have a gap — what about 2 `bal` functions?

Sums in First-Order Logic

Sums in First-Order Logic

- An additional `Coin` sort

Sums in First-Order Logic

- An additional `Coin` sort
- We relate coins to addresses

Sums in First-Order Logic

- An additional `Coin` sort
- We relate coins to addresses
- We count coins

Sums in First-Order Logic

- An additional `Coin` sort
- We relate coins to addresses
- We count coins

Foundational Property
Every "existing" coin is related to precisely one address.

Sums in First-Order Logic

- An additional `Coin` sort
- We relate coins to addresses
- We count coins

Foundational Property

Every "existing" coin is related to precisely one address.

Soundness of the Encoding

Summation constant s satisfies the semantic property of SL

$$\sum_{a} \texttt{bal}(a) = s$$

Counting Coins

Two counting mechanisms.

- $\texttt{count} : \texttt{Coin} \to \mathbb{N}^+$ bijective

Counting Coins

Two counting mechanisms.

- $\texttt{count} : \texttt{Coin} \to \mathbb{N}^+$ bijective
- $\texttt{idx} : \texttt{Addr} \times \texttt{Coin} \to \mathbb{N}^+$, where $\texttt{idx}(a, .)$ bijective for every a

Counting Coins

Two counting mechanisms.

- $\texttt{count} : \texttt{Coin} \to \mathbb{N}^+$ bijective
- $\texttt{idx} : \texttt{Addr} \times \texttt{Coin} \to \mathbb{N}^+$, where $\texttt{idx}(a, .)$ bijective for every a

Existing Coins. A coin c exists iff $\texttt{count}(c) \leq s$.

Counting Coins

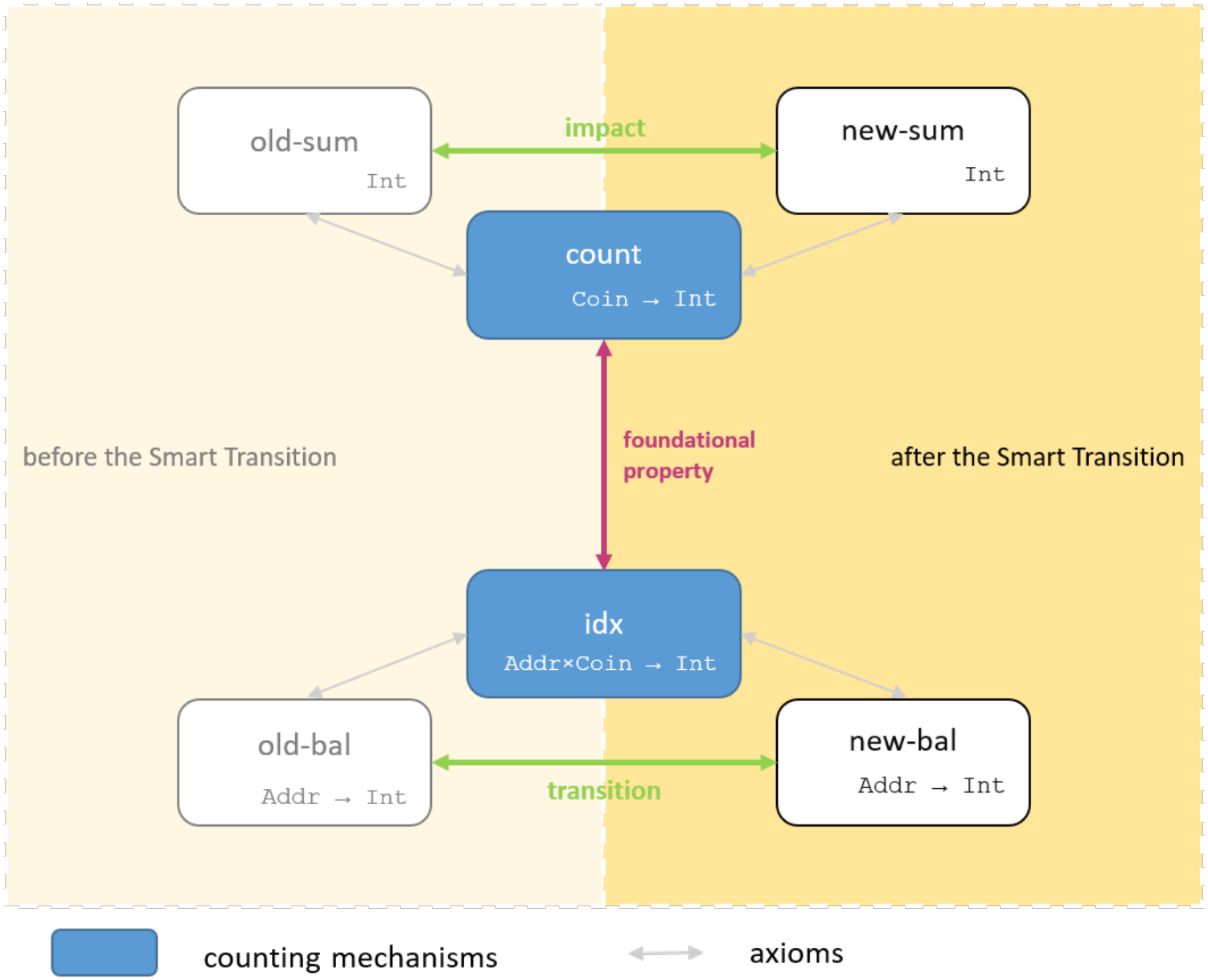
Two counting mechanisms.

- $\texttt{count} : \texttt{Coin} \to \mathbb{N}^+$ bijective
- $\texttt{idx} : \texttt{Addr} \times \texttt{Coin} \to \mathbb{N}^+$, where $\texttt{idx}(a, .)$ bijective for every a

Existing Coins. A coin c exists iff $\texttt{count}(c) \leq s$.

Related Addresses. A coin c is related to an address a iff $\texttt{idx}(a, c) \leq \texttt{bal}(a)$.

High-Level Overview



Benchmarks

Benchmarks

- Automated reasoning for the transfer verification task — transfer_n

Benchmarks

- Automated reasoning for the transfer verification task — transfer_n
- As well as additional *linear* balance updates such as minting

Benchmarks

- Automated reasoning for the transfer verification task — transfer_n
- As well as additional *linear* balance updates such as minting
- And other simple arithmetic problems

Benchmarks

- Automated reasoning for the transfer verification task — transfer_n
- As well as additional *linear* balance updates such as minting
- And other simple arithmetic problems
- For the $\text{transfer}_n, \text{mint}_n$ problems we provide a simplifying assumption

Encodings

For each of those problems, we have different concrete encodings:

Encodings

For each of those problems, we have different concrete encodings:

- Axiomatic naturals (`nat`)

Encodings

For each of those problems, we have different concrete encodings:

- Axiomatic naturals (`nat`)
- UFLIA (`int`)

Encodings

For each of those problems, we have different concrete encodings:

- Axiomatic naturals (`nat`)
- UFLIA (`int`)
- Inductive data-types (`id`)

Encodings

For each of those problems, we have different concrete encodings:

- Axiomatic naturals (`nat`)
- UFLIA (`int`)
- Inductive data-types (`id`)

Experimental Setting.

Encodings

For each of those problems, we have different concrete encodings:

- Axiomatic naturals (`nat`)
- UFLIA (`int`)
- Inductive data-types (`id`)

Experimental Setting.

- SMT-solvers Z3 and CVC4, first-order prover Vampire

Encodings

For each of those problems, we have different concrete encodings:

- Axiomatic naturals (`nat`)
- UFLIA (`int`)
- Inductive data-types (`id`)

Experimental Setting.

- SMT-solvers Z3 and CVC4, first-order prover Vampire
- run on a standard machine

Encodings

For each of those problems, we have different concrete encodings:

- Axiomatic naturals (`nat`)
- UFLIA (`int`)
- Inductive data-types (`id`)

Experimental Setting.

- SMT-solvers Z3 and CVC4, first-order prover Vampire
- run on a standard machine
- time limit of 300s

Experiments

Version	Task							
	$\texttt{mint}_1$		$\texttt{transfer}_1$		$\texttt{mint}_n$		$\texttt{transfer}_n$	
`nat`	Z3:	0.02	Z3:	×	Z3:	×	Z3:	×
	CVC4:	×	CVC4:	×	CVC4:	×	CVC4:	×
	Vampire:	0.92	Vampire:	15.3	Vampire:	$23.2^{\dagger}$	Vampire:	$228^{\dagger}$
`int`	Z3:	0.02	Z3:	0.02	Z3:	0.03	Z3:	0.11
	CVC4:	0.03	CVC4:	0.07	CVC4:	0.05	CVC4:	0.35
	Vampire:	×	Vampire:	×	Vampire:	×	Vampire:	×
`id`	Z3:	×	Z3:	×	Z3:	×	Z3:	×
	CVC4:	×	CVC4:	×	CVC4:	×	CVC4:	×
	Vampire:	$7.36^{\ddagger}$	Vampire:	$17.1^{\ddagger}$	Vampire:	$23.5^{\ddagger}$	Vampire:	×

Table 1: Smart Transitions using different SL encodings.

Results

- Every problem solved by every solver in at least one version (in under 300 seconds)

Results

- Every problem solved by every solver in at least one version (in under 300 seconds)

  $\rightarrow$ encodings suitable for automated reasoners

Results

- Every problem solved by every solver in at least one version (in under 300 seconds)

  $\rightarrow$ encodings suitable for automated reasoners
- For harder tasks splitting proof using *lemmas*

Results

- Every problem solved by every solver in at least one version (in under 300 seconds)

  $\rightarrow$ encodings suitable for automated reasoners
- For harder tasks splitting proof using *lemmas*
- No changes to internals of solvers

Results

- Every problem solved by every solver in at least one version (in under 300 seconds)

  $\rightarrow$ encodings suitable for automated reasoners
- For harder tasks splitting proof using *lemmas*
- No changes to internals of solvers

- For the arithmetic benchmarks, Z3 works very well

Conclusion

Conclusion

- Novel method to automate unbounded sum reasoning about uninterpreted functions

Conclusion

- Novel method to automate unbounded sum reasoning about uninterpreted functions
- Sum Logic appropriate

Conclusion

- Novel method to automate unbounded sum reasoning about uninterpreted functions
- Sum Logic appropriate
- No need to manually update "ghost state"

Conclusion

- Novel method to automate unbounded sum reasoning about uninterpreted functions
- Sum Logic appropriate
- No need to manually update "ghost state"
- Complementary to other smart contract verification approaches

Conclusion

- Novel method to automate unbounded sum reasoning about uninterpreted functions
- Sum Logic appropriate
- No need to manually update "ghost state"
- Complementary to other smart contract verification approaches
- Future work:

Conclusion

- Novel method to automate unbounded sum reasoning about uninterpreted functions
- Sum Logic appropriate
- No need to manually update "ghost state"
- Complementary to other smart contract verification approaches
- Future work:
  - stricter decidability results

Conclusion

- Novel method to automate unbounded sum reasoning about uninterpreted functions
- Sum Logic appropriate
- No need to manually update "ghost state"
- Complementary to other smart contract verification approaches
- Future work:
  - stricter decidability results
  - other aggregates

Conclusion

- Novel method to automate unbounded sum reasoning about uninterpreted functions
- Sum Logic appropriate
- No need to manually update "ghost state"
- Complementary to other smart contract verification approaches
- Future work:
  - stricter decidability results
  - other aggregates
  - multidimensional arrays