Secure Smart Contracts with Isabelle/Solidity[1]

on 2024-05-21

Diego Marmsoler
Department of Computer Science
University of Exeter

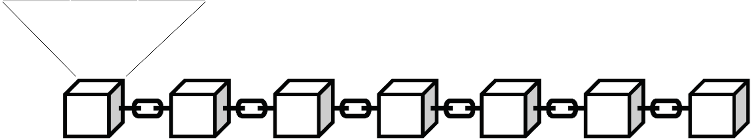
✉ d.marmsoler@exeter.ac.uk
💻 www.marmsoler.com
🐦 @DiegoMarmsoler

Joint work with Asad Ahmed, Achim D. Brucker, Naipeng Dong, Horacio Mijail, Billy Thornton, and Mark Utting

[1] Supported by the Engineering and Physical Sciences Research Council [EP/X027619/1]

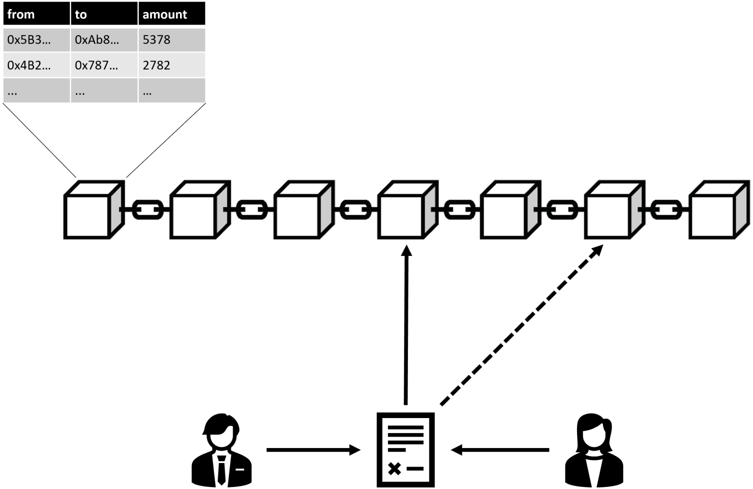
Smart Contracts

from	to	amount
0x5B3...	0xAb8...	5378
0x4B2...	0x787...	2782
...	...	...



Engineering and Physical Sciences Research Council

Smart Contracts



from	to	amount
0x5B3...	0xAb8...	5378
0x4B2...	0x787...	2782
...	...	...

Solidity

Solidity

```
contract Bank {
    mapping(address => uint256) balances;

    function deposit() public payable {
        balances[msg.sender] = balances[msg.sender] + msg.value;
    }

    function withdraw() public {
        uint256 bal = balances[msg.sender];
        balances[msg.sender] = 0;
        msg.sender.transfer(bal);
    }
}
```

Fallback Methods

Solidity

```
contract Customer {
  Bank bank;

  constructor(Bank b) public {
    bank = b;
  }
  function deposit(uint v) public {
    bank.deposit.value(v)();
  }
  function withdraw() public {
    bank.withdraw();
  }
  function() external payable {
    //received some funds
  }
}
```

Solidity

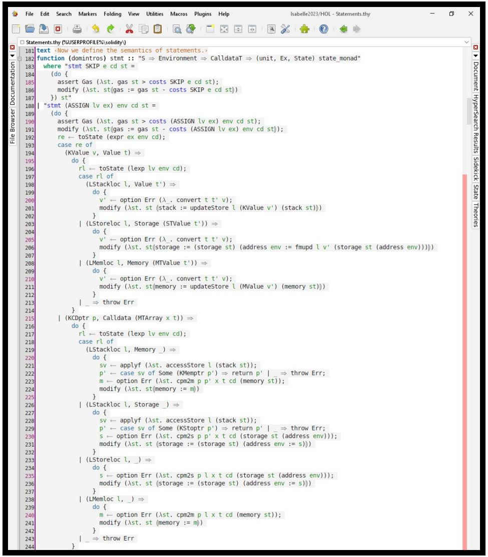
```
contract Bank {
  mapping(address => uint256) balances;

  function deposit() public payable {
    balances[msg.sender] =
    balances[msg.sender] + msg.value;
  }

  function withdraw() public {
    uint256 bal = balances[msg.sender];
    balances[msg.sender] = 0;
    msg.sender.transfer(bal);
  }
}
```

Problems with Smart Contracts

It is estimated that since 2019,
more than $5B was stolen
due to vulnerabilities in smart contracts

Isabelle/Solidity

Isabelle/Solidity is a deep empedding of Solidity (v0.5.16) in Isabelle/HOL



```
text ‹Now we define the semantics of statements.›
function (domintros) stmt :: "S ⇒ Environment ⇒ CalldataT ⇒ (unit, Ex, State) state_monad"
  where "stmt SKIP e cd st =
    (do {
      assert Gas (λst. gas st > costs SKIP e cd st);
      modify (λst. st⦇gas := gas st - costs SKIP e cd st⦈)
    }) st"
| "stmt (ASSIGN lv ex) env cd st =
    (do {
      assert Gas (λst. gas st > costs (ASSIGN lv ex) env cd st);
      modify (λst. st⦇gas := gas st - costs (ASSIGN lv ex) env cd st⦈);
      re ← toState (expr ex env cd);
      case re of
        (KValue v, Value t) ⇒
          do {
            rl ← toState (lexp lv env cd);
            case rl of
              (LStackloc l, Value t') ⇒
                do {
                  v' ← option Err (λ_. convert t t' v);
                  modify (λst. st ⦇stack := updateStore l (KValue v') (stack st)⦈)
                }
            | (LStoreloc l, Storage (STValue t')) ⇒
                do {
                  v' ← option Err (λ_. convert t t' v);
                  modify (λst. st⦇storage := (storage st) (address env := fmupd l v' (storage st (address env)))⦈)
                }
            | (LMemloc l, Memory (MTValue t')) ⇒
                do {
                  v' ← option Err (λ_. convert t t' v);
                  modify (λst. st⦇memory := updateStore l (MValue v') (memory st)⦈)
                }
            | _ ⇒ throw Err
          }
      | (KCDptr p, Calldata (MTArray x t)) ⇒
          do {
            rl ← toState (lexp lv env cd);
            case rl of
              (LStackloc l, Memory _) ⇒
                do {
                  sv ← applyf (λst. accessStore l (stack st));
                  p' ← case sv of Some (KMemptr p') ⇒ return p' | _ ⇒ throw Err;
                  m ← option Err (λst. cpm2m p p' x t cd (memory st));
                  modify (λst. st⦇memory := m⦈)
                }
            | (LStackloc l, Storage _) ⇒
                do {
                  sv ← applyf (λst. accessStore l (stack st));
                  p' ← case sv of Some (KStoptr p') ⇒ return p' | _ ⇒ throw Err;
                  s ← option Err (λst. cpm2s p p' x t cd (storage st (address env)));
                  modify (λst. st ⦇storage := (storage st) (address env := s)⦈)
                }
            | (LStoreloc l, _) ⇒
                do {
                  s ← option Err (λst. cpm2s p l x t cd (storage st (address env)));
                  modify (λst. st ⦇storage := (storage st) (address env := s)⦈)
                }
            | (LMemloc l, _) ⇒
                do {
                  m ← option Err (λst. cpm2m p l x t cd (memory st));
                  modify (λst. st ⦇memory := m⦈)
                }
            | _ ⇒ throw Err
          }
```

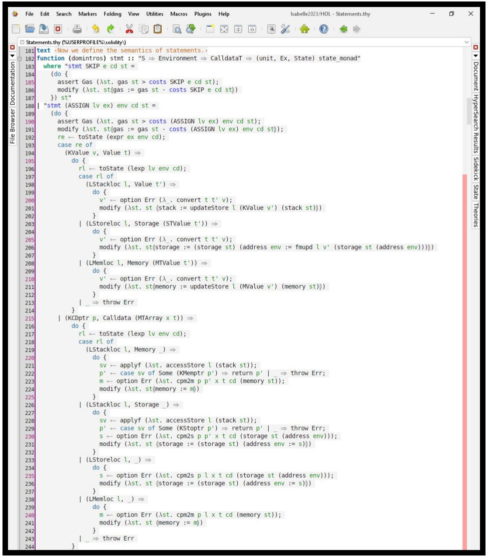
DM and A.D. Brucker. Isabelle/Solidity. AFP 2022.

Engineering and Physical Sciences Research Council

Isabelle/Solidity

Isabelle/Solidity is a deep empedding of Solidity (v0.5.16) in Isabelle/HOL

- *Fixed-size integer types* with and without overflow.



```
text ‹Now we define the semantics of statements.›
function (domintros) stmt :: "S ⇒ Environment ⇒ CalldataT ⇒ (unit, Ex, State) state_monad"
  where "stmt SKIP e cd st =
    (do {
      assert Gas (λst. gas st > costs SKIP e cd st);
      modify (λst. st⦇gas := gas st - costs SKIP e cd st⦈)
    }) st"
| "stmt (ASSIGN lv ex) env cd st =
    (do {
      assert Gas (λst. gas st > costs (ASSIGN lv ex) env cd st);
      modify (λst. st⦇gas := gas st - costs (ASSIGN lv ex) env cd st⦈);
      re ← toState (expr ex env cd);
      case re of
        (KValue v, Value t) ⇒
          do {
            rl ← toState (lexp lv env cd);
            case rl of
              (LStackloc l, Value t') ⇒
                do {
                  v' ← option Err (λ_. convert t t' v);
                  modify (λst. st ⦇stack := updateStore l (KValue v') (stack st)⦈)
                }
            | (LStoreloc l, Storage (STValue t')) ⇒
                do {
                  v' ← option Err (λ_. convert t t' v);
                  modify (λst. st⦇storage := (storage st) (address env := fmupd l v' (storage st (address env)))⦈)
                }
            | (LMemloc l, Memory (MTValue t')) ⇒
                do {
                  v' ← option Err (λ_. convert t t' v);
                  modify (λst. st⦇memory := updateStore l (MValue v') (memory st)⦈)
                }
            | _ ⇒ throw Err
          }
      | (KCDptr p, Calldata (MTArray x t)) ⇒
          do {
            rl ← toState (lexp lv env cd);
            case rl of
              (LStackloc l, Memory _) ⇒
                do {
                  sv ← applyf (λst. accessStore l (stack st));
                  p' ← case sv of Some (KMemptr p') ⇒ return p' | _ ⇒ throw Err;
                  m ← option Err (λst. cpm2m p p' x t cd (memory st));
                  modify (λst. st⦇memory := m⦈)
                }
            | (LStackloc l, Storage _) ⇒
                do {
                  sv ← applyf (λst. accessStore l (stack st));
                  p' ← case sv of Some (KStoptr p') ⇒ return p' | _ ⇒ throw Err;
                  s ← option Err (λst. cpm2s p p' x t cd (storage st (address env)));
                  modify (λst. st ⦇storage := (storage st) (address env := s)⦈)
                }
            | (LStoreloc l, _) ⇒
                do {
                  s ← option Err (λst. cpm2s p l x t cd (storage st (address env)));
                  modify (λst. st ⦇storage := (storage st) (address env := s)⦈)
                }
            | (LMemloc l, _) ⇒
                do {
                  m ← option Err (λst. cpm2m p l x t cd (memory st));
                  modify (λst. st ⦇memory := m⦈)
                }
            | _ ⇒ throw Err
          }
```

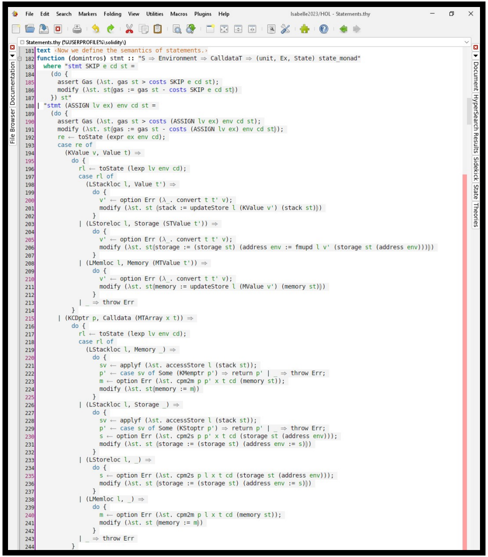
DM and A.D. Brucker. Isabelle/Solidity. AFP 2022.

Engineering and Physical Sciences Research Council

Isabelle/Solidity

Isabelle/Solidity is a deep empedding of Solidity (v0.5.16) in Isabelle/HOL

- *Fixed-size integer types* with and without overflow.
- *Domain-specific primitives*, such as transfer or balance.



```
text ‹Now we define the semantics of statements.›
function (domintros) stmt :: "S ⇒ Environment ⇒ CalldataT ⇒ (unit, Ex, State) state_monad"
  where "stmt SKIP e cd st =
    (do {
      assert Gas (λst. gas st > costs SKIP e cd st);
      modify (λst. st⦇gas := gas st - costs SKIP e cd st⦈)
    }) st"
| "stmt (ASSIGN lv ex) env cd st =
    (do {
      assert Gas (λst. gas st > costs (ASSIGN lv ex) env cd st);
      modify (λst. st⦇gas := gas st - costs (ASSIGN lv ex) env cd st⦈);
      re ← toState (expr ex env cd);
      case re of
        (KValue v, Value t) ⇒
          do {
            rl ← toState (lexp lv env cd);
            case rl of
              (LStackloc l, Value t') ⇒
                do {
                  v' ← option Err (λ_. convert t t' v);
                  modify (λst. st ⦇stack := updateStore l (KValue v') (stack st)⦈)
                }
            | (LStoreloc l, Storage (STValue t')) ⇒
                do {
                  v' ← option Err (λ_. convert t t' v);
                  modify (λst. st⦇storage := (storage st) (address env := fmupd l v' (storage st (address env)))⦈)
                }
            | (LMemloc l, Memory (MTValue t')) ⇒
                do {
                  v' ← option Err (λ_. convert t t' v);
                  modify (λst. st⦇memory := updateStore l (MValue v') (memory st)⦈)
                }
            | _ ⇒ throw Err
          }
      | (KCDptr p, Calldata (MTArray x t)) ⇒
          do {
            rl ← toState (lexp lv env cd);
            case rl of
              (LStackloc l, Memory _) ⇒
                do {
                  sv ← applyf (λst. accessStore l (stack st));
                  p' ← case sv of Some (KMemptr p') ⇒ return p' | _ ⇒ throw Err;
                  m ← option Err (λst. cpm2m p p' x t cd (memory st));
                  modify (λst. st⦇memory := m⦈)
                }
            | (LStackloc l, Storage _) ⇒
                do {
                  sv ← applyf (λst. accessStore l (stack st));
                  p' ← case sv of Some (KStoptr p') ⇒ return p' | _ ⇒ throw Err;
                  s ← option Err (λst. cpm2s p p' x t cd (storage st (address env)));
                  modify (λst. st ⦇storage := (storage st) (address env := s)⦈)
                }
            | (LStoreloc l, _) ⇒
                do {
                  s ← option Err (λst. cpm2s p l x t cd (storage st (address env)));
                  modify (λst. st ⦇storage := (storage st) (address env := s)⦈)
                }
            | (LMemloc l, _) ⇒
                do {
                  m ← option Err (λst. cpm2m p l x t cd (memory st));
                  modify (λst. st ⦇memory := m⦈)
                }
            | _ ⇒ throw Err
          }
```

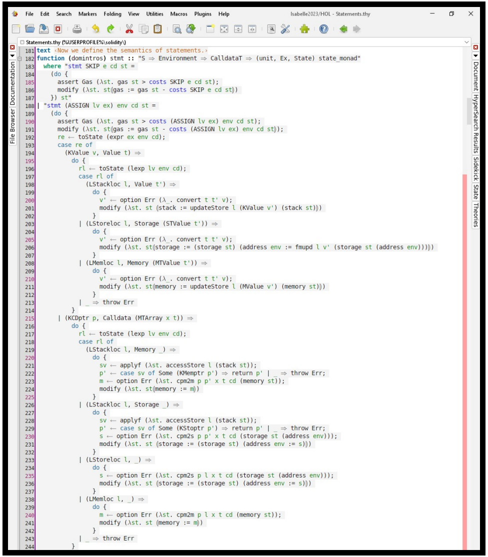
DM and A.D. Brucker. Isabelle/Solidity. AFP 2022.

Engineering and Physical Sciences Research Council

Isabelle/Solidity

Isabelle/Solidity is a deep empedding of Solidity (v0.5.16) in Isabelle/HOL

- *Fixed-size integer types* with and without overflow.
- *Domain-specific primitives*, such as transfer or balance.
- *Fallback methods* which are executed with monetary transfers.



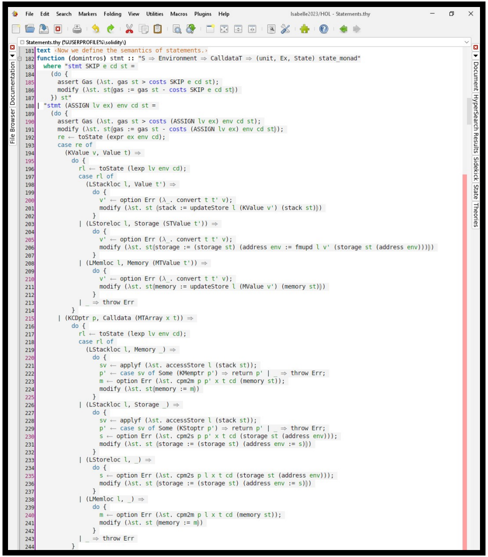
```
text ‹Now we define the semantics of statements.›
function (domintros) stmt :: "S ⇒ Environment ⇒ CalldataT ⇒ (unit, Ex, State) state_monad"
  where "stmt SKIP e cd st =
    (do {
      assert Gas (λst. gas st > costs SKIP e cd st);
      modify (λst. st⦇gas := gas st - costs SKIP e cd st⦈)
    }) st"
| "stmt (ASSIGN lv ex) env cd st =
    (do {
      assert Gas (λst. gas st > costs (ASSIGN lv ex) env cd st);
      modify (λst. st⦇gas := gas st - costs (ASSIGN lv ex) env cd st⦈);
      re ← toState (expr ex env cd);
      case re of
        (KValue v, Value t) ⇒
          do {
            rl ← toState (lexp lv env cd);
            case rl of
              (LStackloc l, Value t') ⇒
                do {
                  v' ← option Err (λ_. convert t t' v);
                  modify (λst. st ⦇stack := updateStore l (KValue v') (stack st)⦈)
                }
            | (LStoreloc l, Storage (STValue t')) ⇒
                do {
                  v' ← option Err (λ_. convert t t' v);
                  modify (λst. st⦇storage := (storage st) (address env := fmupd l v' (storage st (address env)))⦈)
                }
            | (LMemloc l, Memory (MTValue t')) ⇒
                do {
                  v' ← option Err (λ_. convert t t' v);
                  modify (λst. st⦇memory := updateStore l (MValue v') (memory st)⦈)
                }
            | _ ⇒ throw Err
          }
      | (KCDptr p, Calldata (MTArray x t)) ⇒
          do {
            rl ← toState (lexp lv env cd);
            case rl of
              (LStackloc l, Memory _) ⇒
                do {
                  sv ← applyf (λst. accessStore l (stack st));
                  p' ← case sv of Some (KMemptr p') ⇒ return p' | _ ⇒ throw Err;
                  m ← option Err (λst. cpm2m p p' x t cd (memory st));
                  modify (λst. st⦇memory := m⦈)
                }
            | (LStackloc l, Storage _) ⇒
                do {
                  sv ← applyf (λst. accessStore l (stack st));
                  p' ← case sv of Some (KStoptr p') ⇒ return p' | _ ⇒ throw Err;
                  s ← option Err (λst. cpm2s p p' x t cd (storage st (address env)));
                  modify (λst. st ⦇storage := (storage st) (address env := s)⦈)
                }
            | (LStoreloc l, _) ⇒
                do {
                  s ← option Err (λst. cpm2s p l x t cd (storage st (address env)));
                  modify (λst. st ⦇storage := (storage st) (address env := s)⦈)
                }
            | (LMemloc l, _) ⇒
                do {
                  m ← option Err (λst. cpm2m p l x t cd (memory st));
                  modify (λst. st ⦇memory := m⦈)
                }
            | _ ⇒ throw Err
          }
```

DM and A.D. Brucker. Isabelle/Solidity. AFP 2022.

Isabelle/Solidity

Isabelle/Solidity is a deep empedding of Solidity (v0.5.16) in Isabelle/HOL

- *Fixed-size integer types* with and without overflow.
- *Domain-specific primitives*, such as transfer or balance.
- *Fallback methods* which are executed with monetary transfers.
- *Different types of stores*, such as storage, memory, calldata, stack.



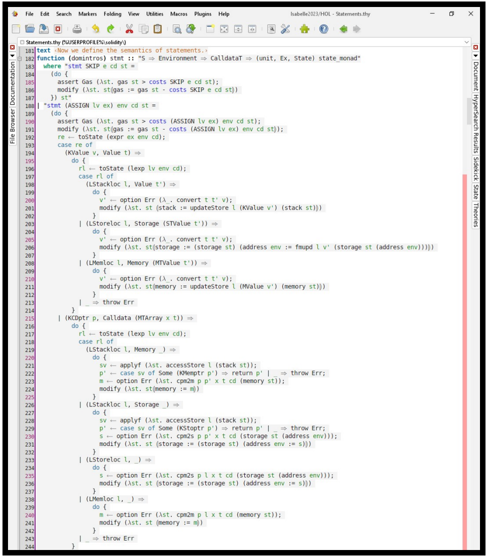
```
text ‹Now we define the semantics of statements.›
function (domintros) stmt :: "S ⇒ Environment ⇒ CalldataT ⇒ (unit, Ex, State) state_monad"
  where "stmt SKIP e cd st =
    (do {
      assert Gas (λst. gas st > costs SKIP e cd st);
      modify (λst. st⦇gas := gas st - costs SKIP e cd st⦈)
    }) st"
| "stmt (ASSIGN lv ex) env cd st =
    (do {
      assert Gas (λst. gas st > costs (ASSIGN lv ex) env cd st);
      modify (λst. st⦇gas := gas st - costs (ASSIGN lv ex) env cd st⦈);
      re ← toState (expr ex env cd);
      case re of
        (KValue v, Value t) ⇒
          do {
            rl ← toState (lexp lv env cd);
            case rl of
              (LStackloc l, Value t') ⇒
                do {
                  v' ← option Err (λ_. convert t t' v);
                  modify (λst. st ⦇stack := updateStore l (KValue v') (stack st)⦈)
                }
            | (LStoreloc l, Storage (STValue t')) ⇒
                do {
                  v' ← option Err (λ_. convert t t' v);
                  modify (λst. st⦇storage := (storage st) (address env := fmupd l v' (storage st (address env)))⦈)
                }
            | (LMemloc l, Memory (MTValue t')) ⇒
                do {
                  v' ← option Err (λ_. convert t t' v);
                  modify (λst. st⦇memory := updateStore l (MValue v') (memory st)⦈)
                }
            | _ ⇒ throw Err
          }
      | (KCDptr p, Calldata (MTArray x t)) ⇒
          do {
            rl ← toState (lexp lv env cd);
            case rl of
              (LStackloc l, Memory _) ⇒
                do {
                  sv ← applyf (λst. accessStore l (stack st));
                  p' ← case sv of Some (KMemptr p') ⇒ return p' | _ ⇒ throw Err;
                  m ← option Err (λst. cpm2m p p' x t cd (memory st));
                  modify (λst. st⦇memory := m⦈)
                }
            | (LStackloc l, Storage _) ⇒
                do {
                  sv ← applyf (λst. accessStore l (stack st));
                  p' ← case sv of Some (KStoptr p') ⇒ return p' | _ ⇒ throw Err;
                  s ← option Err (λst. cpm2s p p' x t cd (storage st (address env)));
                  modify (λst. st ⦇storage := (storage st) (address env := s)⦈)
                }
            | (LStoreloc l, _) ⇒
                do {
                  s ← option Err (λst. cpm2s p l x t cd (storage st (address env)));
                  modify (λst. st ⦇storage := (storage st) (address env := s)⦈)
                }
            | (LMemloc l, _) ⇒
                do {
                  m ← option Err (λst. cpm2m p l x t cd (memory st));
                  modify (λst. st ⦇memory := m⦈)
                }
            | _ ⇒ throw Err
          }
```

DM and A.D. Brucker. Isabelle/Solidity. AFP 2022.

Isabelle/Solidity

Isabelle/Solidity is a deep empedding of Solidity (v0.5.16) in Isabelle/HOL

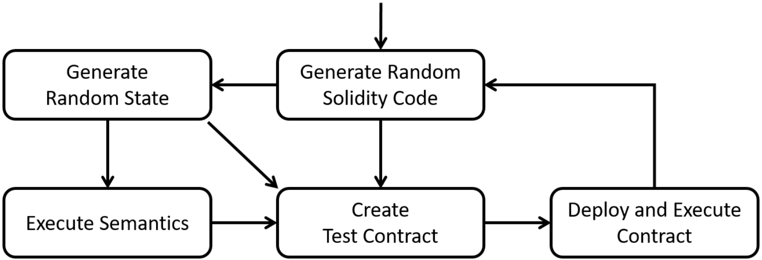
- *Fixed-size integer types* with and without overflow.
- *Domain-specific primitives*, such as transfer or balance.
- *Fallback methods* which are executed with monetary transfers.
- *Different types of stores*, such as storage, memory, calldata, stack.
- *Extendable Gas model* to model computational costs.



```
text ‹Now we define the semantics of statements.›
function (domintros) stmt :: "S ⇒ Environment ⇒ CalldataT ⇒ (unit, Ex, State) state_monad"
  where "stmt SKIP e cd st =
    (do {
      assert Gas (λst. gas st > costs SKIP e cd st);
      modify (λst. st⦇gas := gas st - costs SKIP e cd st⦈)
    }) st"
| "stmt (ASSIGN lv ex) env cd st =
    (do {
      assert Gas (λst. gas st > costs (ASSIGN lv ex) env cd st);
      modify (λst. st⦇gas := gas st - costs (ASSIGN lv ex) env cd st⦈);
      re ← toState (expr ex env cd);
      case re of
        (KValue v, Value t) ⇒
          do {
            rl ← toState (lexp lv env cd);
            case rl of
              (LStackloc l, Value t') ⇒
                do {
                  v' ← option Err (λ_. convert t t' v);
                  modify (λst. st ⦇stack := updateStore l (KValue v') (stack st)⦈)
                }
            | (LStoreloc l, Storage (STValue t')) ⇒
                do {
                  v' ← option Err (λ_. convert t t' v);
                  modify (λst. st⦇storage := (storage st) (address env := fmupd l v' (storage st (address env)))⦈)
                }
            | (LMemloc l, Memory (MTValue t')) ⇒
                do {
                  v' ← option Err (λ_. convert t t' v);
                  modify (λst. st⦇memory := updateStore l (MValue v') (memory st)⦈)
                }
            | _ ⇒ throw Err
          }
      | (KCDptr p, Calldata (MTArray x t)) ⇒
          do {
            rl ← toState (lexp lv env cd);
            case rl of
              (LStackloc l, Memory _) ⇒
                do {
                  sv ← applyf (λst. accessStore l (stack st));
                  p' ← case sv of Some (KMemptr p') ⇒ return p' | _ ⇒ throw Err;
                  m ← option Err (λst. cpm2m p p' x t cd (memory st));
                  modify (λst. st⦇memory := m⦈)
                }
            | (LStackloc l, Storage _) ⇒
                do {
                  sv ← applyf (λst. accessStore l (stack st));
                  p' ← case sv of Some (KStoptr p') ⇒ return p' | _ ⇒ throw Err;
                  s ← option Err (λst. cpm2s p p' x t cd (storage st (address env)));
                  modify (λst. st ⦇storage := (storage st) (address env := s)⦈)
                }
            | (LStoreloc l, _) ⇒
                do {
                  s ← option Err (λst. cpm2s p l x t cd (storage st (address env)));
                  modify (λst. st ⦇storage := (storage st) (address env := s)⦈)
                }
            | (LMemloc l, _) ⇒
                do {
                  m ← option Err (λst. cpm2m p l x t cd (memory st));
                  modify (λst. st ⦇memory := m⦈)
                }
            | _ ⇒ throw Err
          }
```

DM and A.D. Brucker. Isabelle/Solidity. AFP 2022.

How to ensure compliance of the semantics



Generate Random State

Generate Random Solidity Code

Execute Semantics

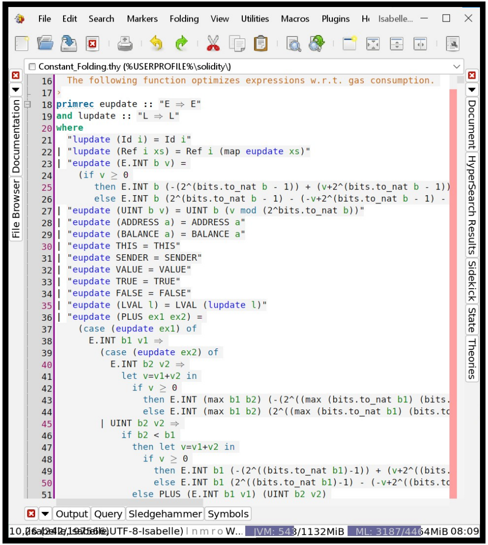
Create Test Contract

Deploy and Execute Contract

DM and A.D. Brucker.
Conformance Testing of Formal Semantics using Grammar-based Fuzzing. TAP 2022.

Applications

- Verified Constant Solving

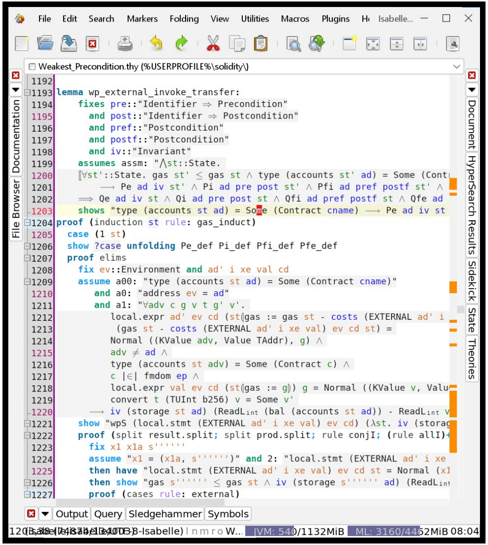


```
  The following function optimizes expressions w.r.t. gas consumption.
›
primrec eupdate :: "E ⇒ E"
and lupdate :: "L ⇒ L"
where
  "lupdate (Id i) = Id i"
| "lupdate (Ref i xs) = Ref i (map eupdate xs)"
| "eupdate (E.INT b v) =
    (if v ≥ 0
       then E.INT b (-(2^(bits.to_nat b - 1)) + (v+2^(bits.to_nat b - 1))
       else E.INT b (2^(bits.to_nat b - 1) - (-v+2^(bits.to_nat b - 1) -
| "eupdate (UINT b v) = UINT b (v mod (2^bits.to_nat b))"
| "eupdate (ADDRESS a) = ADDRESS a"
| "eupdate (BALANCE a) = BALANCE a"
| "eupdate THIS = THIS"
| "eupdate SENDER = SENDER"
| "eupdate VALUE = VALUE"
| "eupdate TRUE = TRUE"
| "eupdate FALSE = FALSE"
| "eupdate (LVAL l) = LVAL (lupdate l)"
| "eupdate (PLUS ex1 ex2) =
    (case (eupdate ex1) of
      E.INT b1 v1 ⇒
        (case (eupdate ex2) of
          E.INT b2 v2 ⇒
            let v=v1+v2 in
              if v ≥ 0
                then E.INT (max b1 b2) (-(2^((max (bits.to_nat b1) (bits.
                else E.INT (max b1 b2) (2^((max (bits.to_nat b1) (bits.to
        | UINT b2 v2 ⇒
            if b2 < b1
              then let v=v1+v2 in
                if v ≥ 0
                  then E.INT b1 (-(2^((bits.to_nat b1)-1)) + (v+2^((bits.
                  else E.INT b1 (2^((bits.to_nat b1)-1) - (-v+2^((bits.to
              else PLUS (E.INT b1 v1) (UINT b2 v2)
```

DM and A. Brucker. A Denotational Semantics of Solidity in Isabelle/HOL. SEFM 2021.

Applications

- Verified Constant Solving
- Soundness of SSCalc



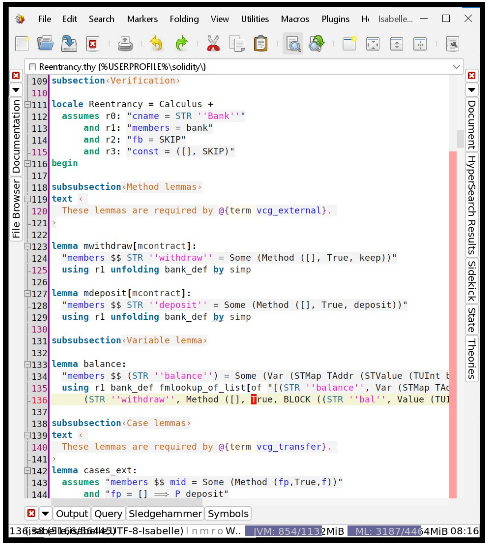
```
lemma wp_external_invoke_transfer:
    fixes pre::"Identifier ⇒ Precondition"
      and post::"Identifier ⇒ Postcondition"
      and pref::"Postcondition"
      and postf::"Postcondition"
      and iv::"Invariant"
    assumes assm: "⋀st::State.
    ⟦∀st'::State. gas st' ≤ gas st ∧ type (accounts st' ad) = Some (Cont
         ⟶ Pe ad iv st' ∧ Pi ad pre post st' ∧ Pfi ad pref postf st' ∧
    ⟹ Qe ad iv st ∧ Qi ad pre post st ∧ Qfi ad pref postf st ∧ Qfe ad
    shows "type (accounts st ad) = Some (Contract cname) ⟶ Pe ad iv st
proof (induction st rule: gas_induct)
  case (1 st)
  show ?case unfolding Pe_def Pi_def Pfi_def Pfe_def
  proof elims
    fix ev::Environment and ad' i xe val cd
    assume a00: "type (accounts st ad) = Some (Contract cname)"
       and a0: "address ev = ad"
       and a1: "∀adv c g v t g' v'.
          local.expr ad' ev cd (st⦇gas := gas st - costs (EXTERNAL ad' i
           (gas st - costs (EXTERNAL ad' i xe val) ev cd st) =
          Normal ((KValue adv, Value TAddr), g) ∧
          adv ≠ ad ∧
          type (accounts st adv) = Some (Contract c) ∧
          c |∈| fmdom ep ∧
          local.expr val ev cd (st⦇gas := g⦈) g = Normal ((KValue v, Valu
          convert t (TUInt b256) v = Some v'
       ⟶ iv (storage st ad) (ReadL_int (bal (accounts st ad)) - ReadL_int v
    show "wpS (local.stmt (EXTERNAL ad' i xe val) ev cd) (λst. iv (storag
    proof (split result.split; split prod.split; rule conjI; (rule allI)+
      fix x1 x1a s''''''
      assume "x1 = (x1a, s'''''')" and 2: "local.stmt (EXTERNAL ad' i xe
      then have "local.stmt (EXTERNAL ad' i xe val) ev cd st = Normal (x1
      then show "gas s'''''' ≤ gas st ∧ iv (storage s'''''' ad) (ReadL_int
      proof (cases rule: external)
```

DM and B. Thornton. SSCalc: A calculus for Solidity smart contracts. SEFM 2023.

UKRI Engineering and Physical Sciences Research Council

Applications

- Verified Constant Solving
- Soundness of SSCalc
- Verified Banking



```
subsection‹Verification›

locale Reentrancy = Calculus +
  assumes r0: "cname = STR ''Bank''"
      and r1: "members = bank"
      and r2: "fb = SKIP"
      and r3: "const = ([], SKIP)"
begin

subsubsection‹Method lemmas›
text ‹
  These lemmas are required by @{term vcg_external}.
›

lemma mwithdraw[mcontract]:
  "members $$ STR ''withdraw'' = Some (Method ([], True, keep))"
  using r1 unfolding bank_def by simp

lemma mdeposit[mcontract]:
  "members $$ STR ''deposit'' = Some (Method ([], True, deposit))"
  using r1 unfolding bank_def by simp

subsubsection‹Variable lemma›

lemma balance:
  "members $$ (STR ''balance'') = Some (Var (STMap TAddr (STValue (TUInt b
  using r1 bank_def fmlookup_of_list[of "[(STR ''balance'', Var (STMap TAd
      (STR ''withdraw'', Method ([], True, BLOCK ((STR ''bal'', Value (TUI

subsubsection‹Case lemmas›
text ‹
  These lemmas are required by @{term vcg_transfer}.
›
lemma cases_ext:
  assumes "members $$ mid = Some (Method (fp,True,f))"
      and "fp = [] ⟹ P deposit"
```

DM and A. Brucker. Isabelle/Solidity: A deep embedding of Solidity in Isabelle/HOL. TBP.

Summary

What is achieved so far

- Formalisation of a subset of Solidity in Isabelle/HOL
    - *Conservative extension* guarantees semantic consistency
    - *Deep embedding* allows to reason about the language itself

Summary

What is achieved so far

- Formalisation of a subset of Solidity in Isabelle/HOL
  - *Conservative extension* guarantees semantic consistency
  - *Deep embedding* allows to reason about the language itself
- Used in several case studies to verify ...
  - Gas-optimizer
  - soundness of Solidity calculus
  - concrete Solidity contracts

Summary

What is achieved so far

- Formalisation of a subset of Solidity in Isabelle/HOL
  - *Conservative extension* guarantees semantic consistency
  - *Deep embedding* allows to reason about the language itself
- Used in several case studies to verify ...
  - Gas-optimizer
  - soundness of Solidity calculus
  - concrete Solidity contracts

What are we currently working on

- *Shallow embedding* to improve automation for the verification of contracts

Summary

What is achieved so far

- Formalisation of a subset of Solidity in Isabelle/HOL
  - *Conservative extension* guarantees semantic consistency
  - *Deep embedding* allows to reason about the language itself
- Used in several case studies to verify ...
  - Gas-optimizer
  - soundness of Solidity calculus
  - concrete Solidity contracts

What are we currently working on

- *Shallow embedding* to improve automation for the verification of contracts
- First results are promising!

References I

- Diego Marmsoler and Achim D. Brucker.
  A Denotational Semantics of Solidity in Isabelle/HOL.
  In Radu Calinescu and Corina S. Păsăreanu, editors, *Software Engineering and Formal Methods*, pages 403–422, Cham, 2021. Springer International Publishing.

- Diego Marmsoler and Achim D. Brucker.
  Conformance testing of formal semantics using grammar-based fuzzing.
  In Laura Kovács and Karl Meinke, editors, *Tests and Proofs*, pages 106–125, Cham, 2022. Springer International Publishing.

- Diego Marmsoler and Achim D. Brucker.
  Isabelle/solidity: A deep embedding of solidity in isabelle/hol.
  Archive of Formal Proofs, July 2022.
  https://isa-afp.org/entries/Solidity.html, Formal proof development.

References II

Diego Marmsoler and Billy Thornton.
SSCalc: A Calculus for Solidity Smart Contracts.
In Carla Ferreira and Tim A. C. Willemse, editors, *Software Engineering and Formal Methods*, pages 184–204, Cham, 2023. Springer Nature Switzerland.

❹ Language Features

Fixed-size Integer Types

Domain-specific Primitives

Gas Model

Method Calls

Complex Data Types

Assignments with Different Semantics

❺ Testing

❻ Example Applications

Verified Constant Solving

SSCalc

Banking Contract

Fixed-size Integer Types

- Signed and unsigned integers from 8 . . . 256 bits (with steps of 8 bits)
- Signed integer types are only compatible with unsigned types of smaller size
- If a value is too large for a size a silent overflow will occur

UKRI Engineering and Physical Sciences Research Council

Fixed-size Integer Types

- Signed and unsigned integers from 8 . . . 256 bits (with steps of 8 bits)
- Signed integer types are only compatible with unsigned types of smaller size
- If a value is too large for a size a silent overflow will occur

Solidity

```
assert(int8(200) == int8(-56));
```

Fixed-size Integer Types

- Signed and unsigned integers from 8 . . . 256 bits (with steps of 8 bits)
- Signed integer types are only compatible with unsigned types of smaller size
- If a value is too large for a size a silent overflow will occur

Solidity

```
assert(int8(200) == int8(-56));  //true
```

Engineering and Physical Sciences Research Council

Fixed-size Integer Types

- Signed and unsigned integers from 8 . . . 256 bits (with steps of 8 bits)
- Signed integer types are only compatible with unsigned types of smaller size
- If a value is too large for a size a silent overflow will occur

Solidity

```
assert(int8(200) == int8(-56));  //true

assert(uint8(200) == uint8(-56));
```

Fixed-size Integer Types

- Signed and unsigned integers from 8 . . . 256 bits (with steps of 8 bits)
- Signed integer types are only compatible with unsigned types of smaller size
- If a value is too large for a size a silent overflow will occur

Solidity

```
assert(int8(200) == int8(-56));  //true

assert(uint8(200) == uint8(-56));  //true
```

UKRI Engineering and Physical Sciences Research Council

Fixed-size Integer Types

- Signed and unsigned integers from 8 . . . 256 bits (with steps of 8 bits)
- Signed integer types are only compatible with unsigned types of smaller size
- If a value is too large for a size a silent overflow will occur

Solidity

```
assert(int8(200) == int8(-56));  //true

assert(uint8(200) == uint8(-56));  //true

assert(uint8(200) + int16(32600) == int16(-32736));
```

Fixed-size Integer Types

- Signed and unsigned integers from 8 . . . 256 bits (with steps of 8 bits)
- Signed integer types are only compatible with unsigned types of smaller size
- If a value is too large for a size a silent overflow will occur

Solidity

```
assert(int8(200) == int8(-56));  //true

assert(uint8(200) == uint8(-56));  //true

assert(uint8(200) + int16(32600) == int16(-32736));  //true
```

Fixed-size Integer Types

- Signed and unsigned integers from 8 . . . 256 bits (with steps of 8 bits)
- Signed integer types are only compatible with unsigned types of smaller size
- If a value is too large for a size a silent overflow will occur

Solidity

```
assert(int8(200) == int8(-56));  //true

assert(uint8(200) == uint8(-56));  //true

assert(uint8(200) + int16(32600) == int16(-32736));  //true

assert(uint16(100) + int16(32700));
```

Fixed-size Integer Types

- Signed and unsigned integers from 8 . . . 256 bits (with steps of 8 bits)
- Signed integer types are only compatible with unsigned types of smaller size
- If a value is too large for a size a silent overflow will occur

Solidity

```
assert(int8(200) == int8(-56));  //true

assert(uint8(200) == uint8(-56));  //true

assert(uint8(200) + int16(32600) == int16(-32736));  //true

assert(uint16(100) + int16(32700));  //compiler error
```

Domain-specific Primitives

- External vs. contract accounts
- Query account balances
- Transfer money

Domain-specific Primitives

- External vs. contract accounts
- Query account balances
- Transfer money

Solidity

```
uint256 x = 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2.balance;
uint256 y = address(this).balance;
```

Engineering and Physical Sciences Research Council

Domain-specific Primitives

- External vs. contract accounts
- Query account balances
- Transfer money

Solidity

```
uint256 x = 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2.balance;
uint256 y = address(this).balance;

0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2.transfer(1000);
```

Domain-specific Primitives

- External vs. contract accounts
- Query account balances
- Transfer money

Solidity

```
uint256 x = 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2.balance;
uint256 y = address(this).balance;

0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2.transfer(1000);

assert(0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2.balance == x+1000);
//true
```

Domain-specific Primitives

- External vs. contract accounts
- Query account balances
- Transfer money

Solidity

```
uint256 x = 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2.balance;
uint256 y = address(this).balance;

0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2.transfer(1000);

assert(0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2.balance == x+1000);
//true

assert(address(this).balance == y-1000); //true
```

Gas Model

- Execution costs Gas
- Programs are guaranteed to terminate
- No specification for Gas costs at Solidity level

Gas Model

- Execution costs Gas
- Programs are guaranteed to terminate
- No specification for Gas costs at Solidity level

Solidity

```
while (true) {}
//terminates with an out of gas exception
```

Method Calls

Recently we added support for *method calls*

- Internal vs. external
- Send money with external calls
- Money transfer triggers fallback

UKRI Engineering and Physical Sciences Research Council

Method Calls

Recently we added support for *method calls*

- Internal vs. external
- Send money with external calls
- Money transfer triggers fallback

Solidity

```
contract R {
  mapping(address => uint256) map;

  function rcv() external payable {
    map[msg.sender] = msg.value;
  }
}
```

Solidity

```
contract S {
  R rec;

  constructor(R r) public payable {
    rec = r;
  }

  function snd(uint256 v) public {
    rec.rcv.value(v)();
  }
}
```

UKRI Engineering and Physical Sciences Research Council

Complex Data Types

- Three types of stores: storage, memory, calldata
- Mappings can only be kept in storage
- Arrays can be kept in all types of stores

Complex Data Types

- Three types of stores: storage, memory, calldata
- Mappings can only be kept in storage
- Arrays can be kept in all types of stores

Solidity

```
contract Example {
  mapping(address => uint256) myMapping; //storage map

  uint8[2][3] myStorageArray; //storage array

  //calldata array
  function example(uint8[2] calldata myCDArray) external {
    uint8[2] storage myPointer = myStorageArray[1]; //storage pointer

    uint8[2] memory myMemoryArray; //memory array
  }
}
```

Assignments with Different Semantics

- Assignment between memory moves pointer
- Assignment between storage copies (except for pointers)
- Assignment between memory and storage copies

Assignments with Different Semantics

- Assignment between memory moves pointer
- Assignment between storage copies (except for pointers)
- Assignment between memory and storage copies

Solidity

```
//initialized with 0
int[2] memory x;
int[2] memory y;

x=y;
x[1]=1;

assert(y[1] == 1); //true
```

Solidity

```
int[2][2] memory x;
int[2][2] memory y;

x[1]=y[1];
x[0][0]=1;
x[1][1]=1;

assert(y[0][0] == 1); //false
assert(y[1][1] == 1); //true
```

Assignments with Different Semantics

- Assignment between memory moves pointer
- Assignment between storage copies (except for pointers)
- Assignment between memory and storage copies

Solidity

```
contract Example {
  //initialized with 0
  int[2] storage y;

  function example() public {
    int[2] storage x=y;

    x[1]=1;

    assert(y[1]==1); //true
  }
}
```

Solidity

```
contract Example {
  //initialized with 0
  int[2] storage x;
  int[2] storage y;

  function example() public {
    x = y;
    x[1]=1;

    assert(y[1]==1); //false
  }
}
```

UKRI Engineering and Physical Sciences Research Council

Assignments with Different Semantics

- Assignment between memory moves pointer
- Assignment between storage copies (except for pointers)
- Assignment between memory and storage copies

Solidity

```
contract Example {
  //initialized with 0
  int[2] storage y;

  function example() public {
    int[2] memory x = y;

    x[1]=1;

    assert(y[1] == 1); //false
  }
}
```

Solidity

```
contract Example {
  //initialized with 0
  int[2] storage y;

  function example() public {
    int[2] memory x = y;
    x[1]=y[1];
    x[1][1]=1;

    assert(y[1][1]==1); //false
  }
}
```

Example

```
contract TestContract0 {
    uint8 v_u8_s8;
    mapping(uint16 => uint8) v_m_u16_u8_9;
    bool [1][2]  a_b_12_s5;
    ...
    function test() public {
        uint104 v_u104_m2;
        uint104 [1][1]  memory a_u104_11_m2;
        ...
        v_u104_m2=14622709355569675963178665339646;
        v_m_u16_u8_9[59381]=79;
        ...
        int8 counter1=int8(0);
        while((v_m_u224_s240_1[uint224(444)]==(v_u216_s1−v_u104_m2)) && counter1<int8(10)){
            0xf7218C33533a3F22e3296F8b1DC0074B399355Eb.transfer(v_m_u16_u8_9[uint16(0)]);
            counter1=counter1+int8(1);
        }
        ...
        Assert.equal(v_m_u16_u8_9[59381]==79, true);
        Assert.equal(a_u104_11_m2[0][0]==8130097819054169632795960896007, true);
        Assert.equal(0xf7218C33533a3F22e3296F8b1DC0074B399355Eb
            .balance==1000000000000000000, true);
        ...
    }
}
```

Extracted storage variables

Extracted memory/stack variables

Generated input state

Generated program

Computed result state

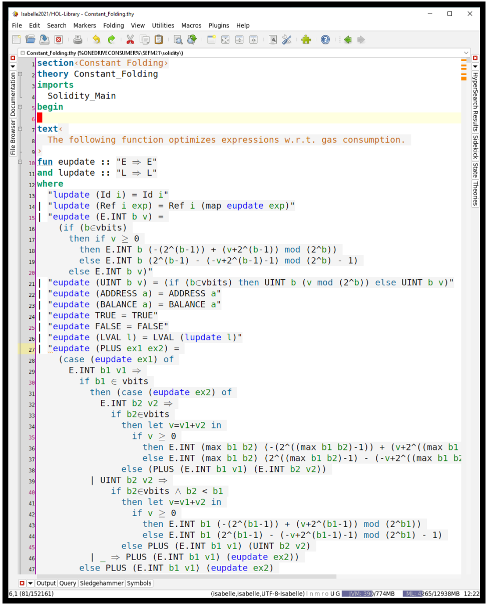
Verified Constant Solving

Solidity

```
int16 x;

// costs 20 Gas
x = int16(250) + uint8(500);
```

Solidity

```
int16 x;

// costs 8 Gas
x = int16(494);
```



Isabelle2021/HOL-Library - Constant_Folding.thy
File Edit Search Markers Folding View Utilities Macros Plugins Help
Constant_Folding.thy (%ONEDRIVECONSUMER%\SEFM21\solidity\)
File Browser Documentation
HyperSearch Results Sidekick State Theories
Output Query Sledgehammer Symbols
6,1 (81/152161)
(isabelle,isabelle,UTF-8-Isabelle) I n m r o U G JVM: 390/774MB ML: 4265/12938MB 12:22

```
section‹Constant Folding›
theory Constant_Folding
imports
  Solidity_Main
begin

text‹
  The following function optimizes expressions w.r.t. gas consumption.
›
fun eupdate :: "E ⇒ E"
and lupdate :: "L ⇒ L"
where
  "lupdate (Id i) = Id i"
| "lupdate (Ref i exp) = Ref i (map eupdate exp)"
| "eupdate (E.INT b v) =
    (if (b∈vbits)
      then if v ≥ 0
        then E.INT b (-(2^(b-1)) + (v+2^(b-1)) mod (2^b))
        else E.INT b (2^(b-1) - (-v+2^(b-1)-1) mod (2^b) - 1)
      else E.INT b v)"
| "eupdate (UINT b v) = (if (b∈vbits) then UINT b (v mod (2^b)) else UINT b v)"
| "eupdate (ADDRESS a) = ADDRESS a"
| "eupdate (BALANCE a) = BALANCE a"
| "eupdate TRUE = TRUE"
| "eupdate FALSE = FALSE"
| "eupdate (LVAL l) = LVAL (lupdate l)"
| "eupdate (PLUS ex1 ex2) =
    (case (eupdate ex1) of
      E.INT b1 v1 ⇒
        if b1 ∈ vbits
          then (case (eupdate ex2) of
            E.INT b2 v2 ⇒
              if b2∈vbits
                then let v=v1+v2 in
                  if v ≥ 0
                    then E.INT (max b1 b2) (-(2^((max b1 b2)-1)) + (v+2^((max b1
                    else E.INT (max b1 b2) (2^((max b1 b2)-1) - (-v+2^((max b1 b
                else (PLUS (E.INT b1 v1) (E.INT b2 v2))
          | UINT b2 v2 ⇒
              if b2∈vbits ∧ b2 < b1
                then let v=v1+v2 in
                  if v ≥ 0
                    then E.INT b1 (-(2^(b1-1)) + (v+2^(b1-1)) mod (2^b1))
                    else E.INT b1 (2^(b1-1) - (-v+2^(b1-1)-1) mod (2^b1) - 1)
                else PLUS (E.INT b1 v1) (UINT b2 v2)
          | _ ⇒ PLUS (E.INT b1 v1) (eupdate ex2))
        else PLUS (E.INT b1 v1) (eupdate ex2)
```

SSCalc

Specification

- Invariant over member variables and balance
- Pre/post-conditions for internal methods

Verification

- Constructor establishes invariant
- External methods preserve invariant
- Preconditions imply postconditions for internal methods

Solidity

```
contract Example {
  uint x;
  constructor(uint y, ...) public {
    ... x = y; ...
  }
  function int1(uint y, ...) internal {
    ... ad1.call.value(1 ether)(abi.
        encodeWithSignature("ext()")); ...
  }
  function ext() external {
    ... int1(5, ...); ...
    ... ad2.transfer(1 ether); ...
  }
  function () external payable {
    ...
  }
}
```

DM and B. Thornton. SSCalc: A calculus for Solidity smart contracts. SEFM 2023.

UKRI Engineering and Physical Sciences Research Council

Verification of Banking Contract

$$\sum_{a} \texttt{balances}(a) \leq \mathit{balance}$$

Solidity

```
contract Bank {
    mapping(address => uint256) balances;

    function deposit() public payable {
        balances[msg.sender] = balances[msg.sender] + msg.value;
    }

    function withdraw() public {
        uint256 bal = balances[msg.sender];
        balances[msg.sender] = 0;
        msg.sender.transfer(bal);
    }
}
```