Fine-Grained Stateful Computations

Georgy Lukyanov, Newcastle Univerity

Supervised by Dr Andrey Mokhov, Newcastle University

Motivation

We propose to mitigate the lack of granularity of the `State`[1] monad by forcing ourselves to think of the state as a *key/value* store.
We want to be able to read a value associated with a key, performing the effects caused by reading; and write an effectual value into the store, performing two effects: one associated with the value itself, and the one required for writing.
Moreover, we want to be aware of the structure of the effects we perform, i.e. whether that structure is *static* or *dynamic* in order to distinguish the computations which can be analysed statically.

```haskell
type Read  k f = forall a. k a -> f a

type Write k f = forall a.
                 k a -> f a -> f a

type FS  c k a = forall f. c f =>
               -> Read  k f
               -> Write k f
               -> f a
```

Value Type Dependens on the Key Constructor

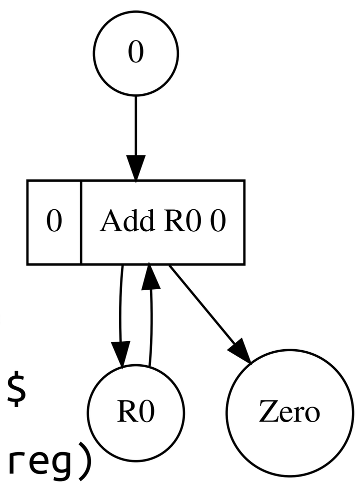
Polymorphic Constraint

```haskell
data MachineKey a where
    Reg  :: Register -> MachineKey Word
    Addr :: MemoryAddress -> MachineKey Word
    F    :: Flag -> MachineKey Bool
    IC   :: MachineKey Word
    IR   :: MachineKey Instruction
    Prog :: IAddr -> MachineKey Instruction
```

Applicative

- static effects
- exact dependencies

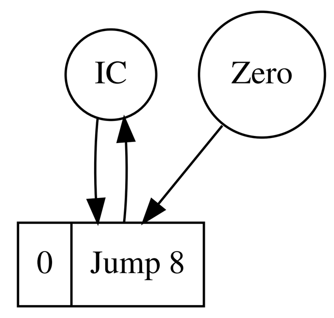
```haskell
add :: Register -> MemoryAddress
    -> FS Applicative MachineKey ()
add reg addr = \read write -> void $
    let result = (+) <$> read (Reg reg)
                     <*> read (Addr addr)
    in write (F Zero)
           ((== 0) <$> write (Reg reg) result)
```



Selective

- branch on effectful values
- approximate dependencies

```haskell
jumpZero :: Offset
         -> FS Selective MachineKey ()
jumpZero offset = \read write ->
    ifS (read (F Zero))
        (void $ write IC ((offset +) <$> read IC))
        (pure ())
```



Monad

- purify effectful values
- unpredictable dependencies

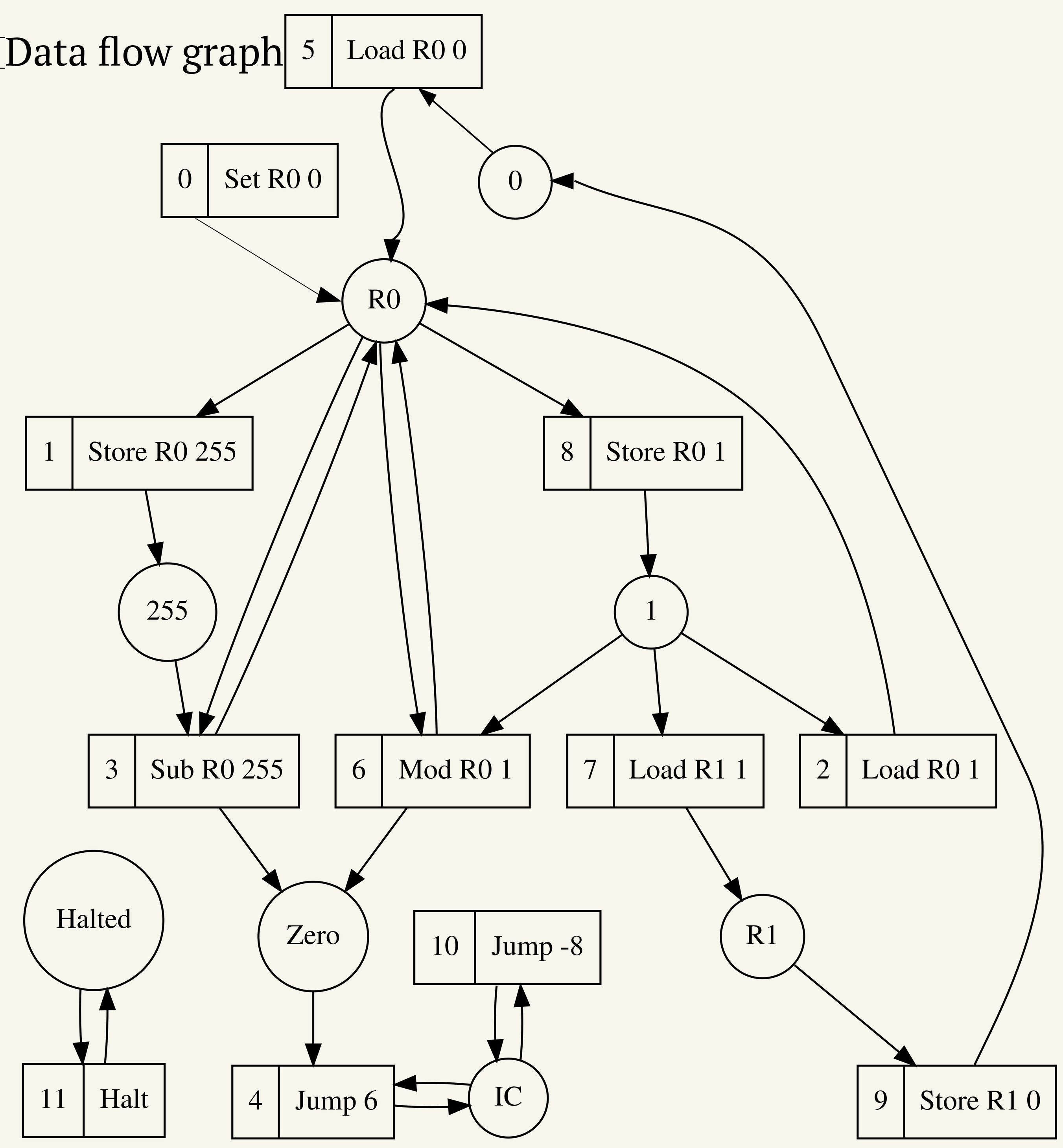
```haskell
loadMI :: Register -> MemoryAddress
       -> FS Monad MachineKey ()
loadMI reg addr = \read write -> void $ do
    pointer <- read (Addr addr)
    write (Reg reg)
          (read (toMemoryAddress pointer))
```

One semantics, multiple interpretations

```haskell
gcdProgram :: Program
gcdProgram =
    [ (0, Set R0 0)
    , (1, Store R0 255)
    , (2, Load R0 1)
    , (3, Sub R0 255)
    , (4, JumpZero 6)
    , (5, Load R0 0)
    , (6, Mod R0 1)
    , (7, Load R1 1)
    , (8, Store R0 1)
    , (9, Store R1 0)
    , (10, Jump (-8))
    , (11, Halt) ]
```

```haskell
read k     = Const [Left (showKey k)]
write k fv = fv *> Const [Right (showKey k)]
```

Data flow graph




```haskell
read  :: k a -> State MachineState a
write :: k a -> State MachineState a
             -> State MachineState a
```

Symbolic Execution
- Paths constraints
- Whole program verification via SBV [2]

Simulation

Results

We apply our approach to instruction set architecture verification. The Haskell source code of a verification framework for a simple hypothetical instruction set is availible on GitHub[6]. The framework describes the semantics of instructions in terms of fine-grained stateful computations. By reinterpreting the polymorphic semantics in the concrete datatypes we obtain (i) a simulator; (ii) a symbolic execution engine which allows to verify programs with an SMT solver; (iii) and a concurrency analysis tool.

Future Work

More applications

- Build Systems a la Carte
- Self-adjusting computatations

Refining Haskell typeclass hierarchy with Selective

```haskell
class Functor f => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

Is there an abstraction between `Applicative` and Monad which would be both statically analysable and efficient?

```haskell
class Applicative f => Selective f where
    select :: f (Either a b) -> f (a -> b) -> f b
```

`select` is a selective function application: given a `Left a`, the function of type `a -> b` must be applied, but it and the associated effects may be skiped when given `Right b`.

`Selective`[7] permits branching on effectful values of any finite types, including `Bool`:

```haskell
ifS :: Selective f => f Bool -> f a -> f a -> f a
```

Every Selective is also a Monad. This allows to retain both *static analysis* and *efficient evaluation.*

```haskell
class Selective m => Monad m where
    (>>=)      :: m a -> (a -> m b) -> m b
    return     :: a -> m a
```

References

[1] Philip Wadler. Monads for Functional Programming. In Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text. Springer-Verlag, Berlin, Heidelberg, 24–52. http://dl.acm.org/citation.cfm?id=647698.734146

[2] Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. 2018. Build Systems à La Carte. Proc. ACM Program. Lang. 2, ICFP, Article 79, https://doi.org/10.1145/3236774

[3] Umut A. Acar. Self-Adjusting Computation. PhD. Thesis. 2005.

[4] Andrey Mokhov, Georgy Lukyanov, and Jakob Lechner. 2018. Formal Verification of Spacecraft Control Programs Using a Metalanguage for State Transformers. arXiv preprint arXiv:1802.01738 (2018).

[5] Levent Erkok, Symbolic Haskell theorem prover using SMT solving. http://hackage.haskell.org/package/sbv

[6] Inglorious Adding Machine, https://github.com/tuura/iam

[7] Andrey Mokhov, Selective Applicative Functors, https://github.com/snowleopard/selective

Contact

g.lukyanov2@ncl.ac.uk
geo2a.info
github/geo2a