Practical Verification of Concurrent Haskell Programs

Michelle Schifferstein - m.schifferstein@student.tudelft.nl
Supervisors: Jesper Cockx (j.g.h.cockx@tudelft.nl) & Lucas Escot (l.f.b.escot@tudelft.nl)
TU Delft, B.Sc. Thesis in Computer Science & Engineering

Github repository

Aim: verification of Haskell concurrency models through Agda2hs.

Haskell: a mainstream strongly typed functional language

Agda: a non-mainstream strongly typed, total functional language with dependent types

Agda2hs: automatically translates Agda code to readable Haskell – combine the best of both worlds!

Concurrent Haskell: *MVar*s are used as mutually exclusive, mutable shared variables. They are 'boxes' that may contain a value, that can be read from when full, and written to when empty.

1. MOTIVATION

- **Formal verification**. As a dependently typed language, Agda allows for formal proofs about program correctness. This significantly improves confidence in the program.
- **Concurrency**. Notoriously difficult to test; instead we can define essential properties that ensure the correctness of our program. These require formal proofs.

2. METHOD

1. Port Haskell concurrency model to Agda. Check it translates to correct Haskell with Agda2hs.
2. Prove correctness of programs with the help of Agda's dependent type system – with a focus on the occurrence of deadlocks.

3. CONCURRENCY MODEL

- Haskell concurrency libraries: low-level implementation, difficult to port directly.
- Claessen 1999 [1]: concurrency monad transformer that adds a simple form of concurrency to any monad:

```
type C m α = (α → Action m) → Action m

data Action m
  = Atom (m (Action m))
  | Fork (Action m) (Action m)
  | Stop
```

Concurrency monad transformer and accompanying Action datatype

```
round :: Monad m ⇒ [Action m] → m ()
round []       = return ()
round (a : as) = case a of
  Atom am    → do a' ← am ; round (as ++ [a'])
  Fork a1 a2 → round (as ++ [a1, a2])
  Stop       → round as
```

Round robin scheduling function for interleaving actions

- Can be used to simulate Haskell MVars
- Combine with pure IO model with memory representation [3], so Agda can evaluate the programs

4. FROM HASKELL TO AGDA (AND BACK) (1)

Circumvent Haskell's *newtype* declarations:

```
record C (m : Set → Set) (a : Set) : Set where
    constructor Conc
    field
        act : (a → Action m) → Action m
open C public
{-# COMPILE AGDA2HS C #-}
```

Agda2hs

```
data C m a = Conc{act :: (a -> Action m) ->
Action m}
```

4. FROM HASKELL TO AGDA (AND BACK) (2)

To meet Agda's termination requirement, add 'fuel' to enforce termination of non-terminating functions:

```
round : { Monad m } → List (Action m) →
      MyNat → m Bool
round [] _ = return True
round xs Zero = return False
round (Stop :: xs) (Suc n) = round xs n
round (Atom x :: xs) (Suc n) = ...
round (Fork x y :: xs) (Suc n) = ...
{-# COMPILE AGDA2HS round #-}
```

5. VERIFICATION

Interesting properties are mostly about specific concurrent *programs*, not about the *models*. We focus on:

- *Absence of deadlocks*. There is no program configuration in which all processes wait indefinitely for each other.

We can use the forced termination of our functions to prove whether our programs terminate or not, i.e. whether a deadlock occurs.

6. DEADLOCK PROOF

We can prove that this program does not terminate because it runs out of fuel:

```
mVarDeadlock : Bool
mVarDeadlock = runIOs (run (do
                a <- newEmptyMVar
                b <- newEmptyMVar
                fork (do
                        takeMVar a fuel
                        writeMVar b 1 fuel)
                takeMVar b fuel
                writeMVar a 2 fuel
                ) (natToMyNat 100000))
{-# COMPILE AGDA2HS mVarDeadlock #-}

deadlock-proof : mVarDeadlock ≡ False
deadlock-proof = refl
```

7. RESULTS

- Ported Claessen's concurrency model to Agda with minor adjustments, generated readable Haskell with Agda2hs.
- Implemented a model for IO (Swierstra 2008) to enable reasoning about concurrent programs in Agda.
- Proven occurrence of deadlocks for simple concurrent programs with *MVar*s.

8. LIMITATIONS

- Agda's requirements for totality and termination
- Deterministic *round robin* scheduler
- Assumed correspondence Haskell's IO with modeled IO

9. FUTURE WORK

- Devising alternatives for round robin scheduler
- Verifying more complex concurrent programs
- Verifying other properties
- Porting a simplified STM model to Agda [2]

10. REFERENCES

[1] K. Claessen, "A poor man's concurrency monad," Journal of Functional Programming, vol. 9, pp. 313–323, 1999.

[2] L. Hu, Compiling Concurrency Correctly Verifying Software Transactional Memory. PhD thesis, University of Nottingham, 6 2012.

[3] W. Swierstra, A functional specification of effects. PhD thesis, University of Nottingham, 11 2008.