Generating CSP Models for Communicating Haskell Processes

Neil Brown

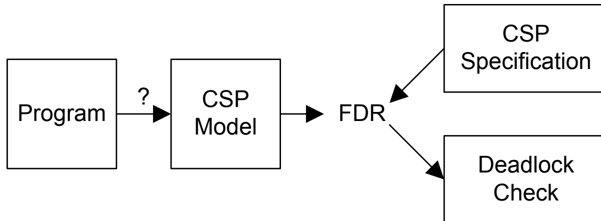
Computing Laboratory
University of Kent
UK

7 August 2009

University of Kent | Computing

Communicating Sequential Processes (CSP)

- Process calculus, developed by Hoare and Roscoe
- Synchronous message-passing concurrency
- Supports model-checking (using FDR) of model against specification



University of Kent Computing

Communicating Haskell Processes (CHP)

- Haskell imperative concurrency library
- Everything happens inside the CHP monad
  - The CHP monad is really a transformer on top of the IO monad, so CHP has `liftIO`
- Already available on Hackage

University of Kent Computing

Dining Philosophers

University of Kent | Computing

Philosopher Model

```haskell
phil :: Event -> Event -> CHP ()
phil left right
  = do randomDelay -- thinking
       sync left <||> sync right
       randomDelay -- eating
       sync left <||> sync right
       phil left right
  where randomDelay = liftIO $ ...
```

$$
\begin{aligned}
&\mathit{PHIL}(\mathit{left}, \mathit{right}) \\
&\quad = (\mathit{left} \rightarrow \mathit{SKIP} \mathbin{|||} \mathit{right} \rightarrow \mathit{SKIP}) \\
&\quad \mathbin{⨟} (\mathit{left} \rightarrow \mathit{SKIP} \mathbin{|||} \mathit{right} \rightarrow \mathit{SKIP}) \\
&\quad \mathbin{⨟} \mathit{PHIL}(\mathit{left}, \mathit{right})
\end{aligned}
$$

University of Kent Computing

Overview

Problem: turn Haskell program (using CHP library/monad) into its CSP model

University of Kent Computing

Get Analysing!

We’ll need:

- Source code

Get Analysing!

We'll need:

- Source code
    - Including any libraries (probably on Hackage)

Get Analysing!

We'll need:

- Source code
  - Including any libraries (probably on Hackage)
- Haskell parser (haskell-src)

Get Analysing!

We'll need:

- Source code
  - Including any libraries (probably on Hackage)
- Haskell parser (haskell-src)
  - Including any extensions used (haskell-src-exts)

University of Kent Computing

Get Analysing!

We'll need:

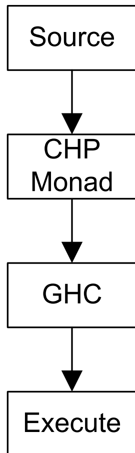
- Source code
  - Including any libraries (probably on Hackage)
- Haskell parser (haskell-src)
  - Including any extensions used (haskell-src-exts)
- Haskell semantics (hmmm...)

Get Analysing!

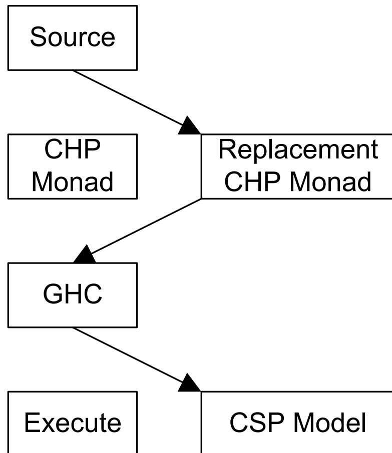
We'll need:

- Source code
  - Including any libraries (probably on Hackage)
- Haskell parser (haskell-src)
  - Including any extensions used (haskell-src-exts)
- Haskell semantics (hmmm. . .)
  - Including any extensions used

Program “Stack”



Program “Stack”



University of Kent | Computing

Take a Short-Cut

Why not just redefine the monad and let the compiler/run-time do all the work?

```
type CHP a ≈ Writer CSPModel a

sync (Event id)  ≈  tell  (EventSync id)
a <||> b ≈ tell  (Par (execWriter a) (execWriter b))
```

University of Kent Computing

Unchanged Code

```
phil :: Event -> Event -> CHP ()
phil left right
  = do randomDelay -- thinking
       sync left <||> sync right
       randomDelay -- eating
       sync left <||> sync right
       phil left right
  where randomDelay = liftIO $ ...
```

University of Kent | Computing

Get Analysing!

We'll need:

- Source code (GHC must compile it)
  - Including any libraries (ditto)
- Haskell parser (GHC)
  - Including any extensions used (GHC)
- Haskell semantics (GHC must be able to run it)
  - Including any extensions used (ditto)

Overview

Problem: turn Haskell program (using CHP library/monad) into its CSP model

Solution: substitute CHP monad; new problems:

- Recursion/repetition
- Non-determinism
- Tracking values

Repetition

```
phil :: Event -> Event -> CHP ()
phil left right
  = do randomDelay -- thinking
       sync left <||> sync right
       randomDelay -- eating
       sync left <||> sync right
       phil left right
  where randomDelay = liftIO $ ...
```

University of Kent | Computing

Repetition

Problem: phil never terminates, so neither does generating its model!

Solution: annotation for recursive processes:

```
phil = process "phil" $ \left right ->
    do randomDelay -- thinking
       sync left <||> sync right
       randomDelay -- eating
       sync left <||> sync right
       phil left right
  where randomDelay = liftIO $ ...
```

University of Kent | Computing

Process Annotation

```
class ProcessAnnotation a where
  process :: String -> a -> a

instance (Eq a, ProcessAnnotation b) =>
           ProcessAnnotation (a -> b) ...
instance ProcessAnnotation (CHP a) ...
```

- Process annotation captures repetition
- Also allows knowledge of parameters
  - Assume a process acts the same given the same parameters
  - Only need to examine a process again if its parameters change

University of Kent | Computing

Philosopher Model

```
phil :: Event -> Event -> CHP ()
phil = process "phil" $
  \ left right ->
    do randomDelay -- thinking
       sync left <||> sync right
       randomDelay -- eating
       sync left <||> sync right
       phil left right
  where randomDelay = liftIO $ ...
```

$$
\begin{aligned}
&\mathit{PHIL}(\mathit{left}, \mathit{right}) \\
&\quad = (\mathit{left} \rightarrow \mathit{SKIP} \mathbin{|||} \mathit{right} \rightarrow \mathit{SKIP}) \\
&\quad \mathbin{⨾} (\mathit{left} \rightarrow \mathit{SKIP} \mathbin{|||} \mathit{right} \rightarrow \mathit{SKIP}) \\
&\quad \mathbin{⨾} \mathit{PHIL}(\mathit{left}, \mathit{right})
\end{aligned}
$$

But this is deterministic…

Sources of Non-Determinism

Non-determinism in a process must arise from one or more of:

- Parameters
- Results of IO actions (liftIO)
- Reading from channels

Only interested when *communication behaviour* is affected.

Non-determinism

```
liftIO  readBool >>= \b -> if b then sync c else sync d

liftIO  _ >>= _
```

University of Kent | Computing

Problem: Black Box!

- We know where the external data comes in, but we don’t know how (or if!) it affects program behaviour.

Problem: Black Box!

- We know where the external data comes in, but we don't know how (or if!) it affects program behaviour.
- Need to input different arbitrary data into black box, and see when the behaviour (output, in a way) varies.

Problem: Black Box!

- We know where the external data comes in, but we don't know how (or if!) it affects program behaviour.
- Need to input different arbitrary data into black box, and see when the behaviour (output, in a way) varies.
- Similar to property-based testing! We can use QuickCheck

Problem: Black Box!

- We know where the external data comes in, but we don't know how (or if!) it affects program behaviour.
- Need to input different arbitrary data into black box, and see when the behaviour (output, in a way) varies.
- Similar to property-based testing! We can use ~~QuickCheck~~ Lazy SmallCheck

Lazy SmallCheck

```
foo :: Maybe String -> [Int]
fooResultCheck :: [Int] -> Bool
```

Lazy SmallCheck

```
foo :: Maybe String -> [Int]
fooResultCheck :: [Int] -> Bool

undefined :: Maybe String
```

Lazy SmallCheck

```
foo :: Maybe String -> [Int]
fooResultCheck :: [Int] -> Bool

undefined :: Maybe String
Nothing :: Maybe String
Just undefined :: Maybe String
```

Lazy SmallCheck

```
foo :: Maybe String -> [Int]
fooResultCheck :: [Int] -> Bool

undefined :: Maybe String
Nothing :: Maybe String
Just undefined :: Maybe String
Just [] :: Maybe String
Just (undefined : undefined) :: Maybe String
```

University of Kent Computing

Lazy SmallCheck

Lazy SmallCheck can tell us:

- All the different behaviours produced by varying the input values, up to a given search depth
- And whether the search was exhaustive
  - If not, the model is only an approximation

Non-determinism

```
liftIO  readBool >>= \b -> if b then sync c else sync d
```

$$(retvalA \rightarrow c \rightarrow SKIP) \square (retvalB \rightarrow d \rightarrow SKIP)$$

- Can handle external data that has a small domain or that is barely used

Tracking Values

```
identityProcess :: Chanin a -> Chanout a -> CHP ()
identityProcess input output
  = foreverP $ readChannel input >>= writeChannel output
```

$$identity(IN, OUT) = in?x \rightarrow out!x \rightarrow identity(IN, OUT)$$

- We need a way to track the values from one channel to another

University of Kent | Computing

Tracking Values: Current Solution

Use error for the input return, along with try and evaluate in the output to test for special values. Not perfect:

```
succProcess :: Chanin Int -> Chanout Int -> CHP ()
succProcess input output
  = foreverP $ do x <- readChannel input
                  writeChannel output (x + 1)
```

(Better suggestions welcome after the talk)

University of Kent | Computing

Conclusions

- The approach works! (With caveats)
- Haskell-specific
- CSP-specific
- Haskell's Strengths:
  - Monads can form an intermediate *alterable* layer
  - Type system and purity show where side effects are, and where external data can enter the program
  - Laziness allows for clever techniques with bottom (and thus Lazy SmallCheck)

Questions?

- Paper accepted at AVoCS 2009
- Hope to release software soon