CCS model checker in Haskell

Arnar Birgisson
Reykjavik University, 2008

Abstract

We give a high-level overview of a basic CCS/HML model checker written in Haskell, a pure, lazy, functional language. The model checker handles recursive CCS definitions and non-recursive HML formulas. It can apply search on the process graph to look for reachable processes having certain properties.

1 Introduction

This report outlines a rudimentary implementation of a CCS model checker using the purely functional language Haskell. The resulting model checker is able to verify logical expressions of non-recursive Hennessy-Milner logic against CCS expressions. CCS expressions can contain (previously defined) variables and variable definitions may be recursive. The model checker does not support checking process equivalence such as bisimilarity [1].

Our main motivation is to show an example of how both operational and denotational semantics can be translated to Haskell in a simple manner.

The complete source code discussed here is available at `http://www.hvergi.net/arnar/code/haskell-ccs/`.

Thanks to my instructor, Luca Aceto, for reading the drafts and providing me with useful comments.

2 Overview

The implementation consists of four main components. All parts rely on the module `CCS` which defines the datatypes used to represent CCS and HML expressions.

Parser The module `Parser` uses Parsec [2] parser combinators to construct parsers for both CCS and HML. These convert strings to values of the types `Process` and `HMLFormula` from the `CCS` module.

Evaluation The semantics of both CCS and HML are implemented in the module `Semantics`. This module is a simple translation of the SOS rules for CCS and the HML evaluation function, both of which are defined in [1].

Search The `Search` module implements several graph search algorithms, such as BFS, DFS, iterative deepening DFS [4] and the heuristic search method A* [5]. Most of these are not relevant to the model checker, but A* is used to check for reachability of a state satisfying a given HML formula.

Console driver The `Console` module makes use of the Shellac [3] library to set up an interactive shell where the user can define CCS parameters and issue commands to list successors of processes, test a process against a HML formula and perform reachability searches.

3 Basic types and functions

To represent CCS expressions internally, we make use of a basic algebraic data type `Process`. It in turn relies on a simple type for actions and a few type synonyms.

```
type Signal = String
type ProcessName = String
type Renaming = Map.Map Signal Signal
type Restriction = Set.Set Action

data Action
    = Input Signal
    | Output Signal
    | Tau
    deriving (Eq, Show, Ord)

data Process
    = Name ProcessName
    | Prefix Action Process
    | Choice [Process]
    | Parallel Process Process
    | Rename Process Renaming
    | Restrict Process Restriction
    deriving (Eq, Show, Ord)
```

To keep track of defined process variables, we use a simple map.

```
type ProcessDefinitions = Map.Map ProcessName Process
```

Finally, a handy constant to have around is the null process, which may be represented as an empty choice.

```
nullProcess :: Process
nullProcess = Choice []
```

We have a similar type for HML expressions

```
data HMLFormula
    = HMLTrue
    | HMLFalse
    | HMLAnd HMLFormula HMLFormula
    | HMLOr HMLFormula HMLFormula
    | HMLBox Action HMLFormula
    | HMLDiamond Action HMLFormula
    deriving (Eq,Show)
```

4 Parsing

We use the Parsec [2] parser combinator library to construct a simple parser so the user can easily construct values of the above types. It is not our intention to detail it here as it is entirely standard and only makes use of Parsec's basic features, but instead we give just a taste of how the parsers are combined and show a neat trick on how to handle the relabeling and restriction operators.

CCS parsing begins with the choice operator, which has the lowest precedence. We simply parse a sequence of parallel constructs (the next operator in increasing precedence order) separated by a `+`. `parallelProcess` is in turn built on the next operator in precedence order. The whole parser is built like this by stacking such parsers that deal with only the operators of a certain precedence each.

```
process :: Parser Process
process = choiceProcess

choiceProcess :: Parser Process
choiceProcess =
    do ps <- sepBy1 parallelProcess (symbol "+")
       if null $ tail ps
          then return $ head ps
          else return $ Choice ps

parallelProcess :: Parser Process
parallelProcess =
    do whiteSpace
       chainl1 (lexeme prefixedProcess) pipe
       where
         pipe = do symbol "|"
                   return Parallel
```

Eventually we reach the highest precedence operators, relabeling and restriction. These operators are unary postfix operators, which means that when it comes to actually parsing the operator itself, we already have in hand a fully parsed process on which the operator shall act. Since both have equal precedence and should be applied in the order they appear, it is not immediately obvious how to do so. By using the fact that functions are first class values in Haskell, we can write a parser that parses just a single relabeling or restriction operator and returns not a `Process`, but a function that given a `Process` P, returns a `Process` where the operator has been applied on P.

```
process_adapter :: Parser (Process -> Process)
process_adapter =
    do restr <- restriction
       return (\p -> Restrict p restr)
    <|>
    do ren <- renaming
       return (\p -> Rename p ren)
```

As we can see, this is very simple. The `process_adapter` parser simply parses a restriction or a relabeling operator, and for each returns a function that wraps a process in the relevant constructor from the `Process` type.

We'll make do with showing only the parser for a restriction,

```
restriction :: Parser Restriction
restriction =
   do symbol "\\"
      oneAction <|> (braces manyActions)
      where
        manyActions = do ss <- commaSep1 action
                         return $ Set.fromList ss
        oneAction  = do a <- action
                        return $ Set.singleton a
```

Now, with the parser `process_adapter` in hand, what we need to do is to write a parser that parses a CCS atom (i.e. the null process, a process variable or a parenthesized full process), followed by a sequence of *process adapters*, applies each one in turn and returns the result.

```
adaptedProcess :: Parser Process
adaptedProcess =
    do p <- simpleProcess
       adapters <- many process_adapter
       return $ foldr (flip (.)) id adapters p
```

Note: the `foldr (flip (.)) id adapters` expression may look a little exotic, but simply put it takes a list of functions and composes them in order, starting with the identity function. In other words,

```
foldr (flip (.)) id [f1, f2, f3] x
  ==  (f3 . f2 . f1 . id) x
  ==  (f3 (f2 (f1 (id x))))
```

5 Evaluation

We now turn our attention towards the module `Semantics`, which implements the semantics detailed in appendix 7. The main function for CCS semantics is `ccsSucc`, which has the type signature

```
ccsSucc :: ProcessDefinitions
        -> Process
        -> [(Action, Process)]
```

This function, given declaration for the defined CCS variables and a process P, returns all possible pairs of actions α and processes P' such that

$$P \xrightarrow{\alpha} P'$$

is provable using the SOS rules for CCS. In other words, it lists the possible successor processes of P and the corresponding actions. Note that due to Haskell's lazyness, there is no problem with returning an infinite list. This function is then defined structurally using pattern matching on the CCS constructors. The simplest case is action prefixing,

```
ccsSucc defs (Prefix a p) = [(a, p)]
```

A little more complex are the cases corresponding to rules *(SUM)*, *(RES)*, *(REL)* and *(CON)*. `nub` is a standard function that eliminates duplicates from a list.

```
-- Rule SUM
ccsSucc defs (Choice ps) =
    nub $ concatMap (ccsSucc defs) ps

-- Rule RES
ccsSucc defs (Restrict p r) =
    nub $ map wrap $ filter f $ ccsSucc defs p
    where
      f (a,_) = (Set.notMember a r) &&
                (Set.notMember (complement a) r)
      wrap (a,p') = (a, Restrict p' r)

-- Rule REL
ccsSucc defs (Rename p r) =
    nub $ map ren $ ccsSucc defs p
    where
      f = renameFunction r
      ren (a,p') = (f a, Rename p' r)

-- Rule CON
ccsSucc defs (Name pn) =
    case Map.lookup pn defs of
      Just p  -> ccsSucc defs p
      Nothing -> error (pn ++ "␣is␣undefined")

-- Helper function for relabelings
renameFunction :: Renaming -> Action -> Action
renameFunction r (Input a) =
    case Map.lookup a r of
       Just a' -> Input a'
       Nothing -> Input a
renameFunction r (Output a) =
    case Map.lookup a r of
       Just a' -> Output a'
       Nothing -> Output a
```

The most interesting case however is the implementation of parallel composition, rules *(COM1)-(COM3)*.

```
-- Rules COM1 - COM3
ccsSucc defs (Parallel p q) =
  nub $ concat [
      map (wrap (\x -> Parallel x q)) lefts,
      map (wrap (\x -> Parallel p x)) rights,
      catMaybes $ map (uncurry sync)
                      [(l,r) | l <- lefts,
                               r <- rights]
    ]
   where
     lefts  = ccsSucc defs p
     rights = ccsSucc defs q
     wrap f (a, p) = (a, f p)
     sync (a,p) (a',p')
          | a == complement(a') =
               Just (Tau, Parallel p p')
          | otherwise = Nothing
```

Here, the helper method `sync` matches up two action/process pairs if the actions complement each other. The helpers `lefts` and `rights` perform one step on each side of the operator. The possible successor states from all three cases are then concatenated and duplicates removed.

Evaluation of HML formulas against processes is defined in a similarly structured manner.

```
-- HML evaluation
hmlEvaluate :: ProcessDefinitions
            -> HMLFormula -> Process -> Bool

hmlEvaluate _ HMLTrue _ = True
hmlEvaluate _ HMLFalse _ = False

hmlEvaluate defs (HMLAnd f1 f2) p =
   (hmlEvaluate defs f1 p) && (hmlEvaluate defs f2 p)

hmlEvaluate defs (HMLOr  f1 f2) p =
   (hmlEvaluate defs f1 p) || (hmlEvaluate defs f2 p)

hmlEvaluate defs (HMLBox a f) p =
   all (hmlEvaluate defs f)
       (map snd (filter ((== a) . fst)
                        (ccsSucc defs p)
                )
       )

hmlEvaluate defs (HMLDiamond a f) p =
    any (hmlEvaluate defs f)
        (map snd (filter ((== a) . fst)
                         (ccsSucc defs p)
                )
        )
```

We can see how the symmetry in the box and diamond operators is nicely reflected in the Haskell functions `all` and `any`.

We will skip discussing the implementation of the search algorithms as they are not relevant to this discussion.

6 Console interaction

The mode of user interaction with the model checker is through an interactive console. For this purpose we used the Shellac [3] package, which makes it particularly easy to write interactive console applications in Haskell. While our code is a good example of Shellac usage, we will not go into the details here. It suffices to mention that throughout the console session we maintain a global instance of the `ProcessDefinitions` type. The user can define process variables that are stored in this map, which is then passed to every evaluation of a process.

To see this in action, consider the classic *coffee and tea machine* [1]. (In the examples below, letters in bold indicate text entered by the user. Actions prefixed with ^ correspond to *output actions*, often indicated with an overbar in the literature.)

```
ccs> def CTM = coin.(^coffee.CTM + ^tea.CTM)
ccs> list
Process definitions
  CTM = coin.(^coffee.CTM + ^tea.CTM)
```

The `list` command lists all defined process names together

with their definitions. The `succ` command tells us what are the possible actions and successor processes of a process,

```
ccs> succ CTM
--(coin)--> (^coffee.CTM + ^tea.CTM)

ccs> succ (^coin.0 | CTM)
--(^coin)--> (0 | CTM)
--(coin)--> (^coin.0 | (^coffee.CTM + ^tea.CTM))
--(tau)--> (0 | (^coffee.CTM + ^tea.CTM))

ccs> succ (^coin.0 | CTM) \ coin
--(tau)--> (0 | (^coffee.CTM + ^tea.CTM)) \ {coin}
```

The command `test` takes an HML formula and a process and tests if the process satisfies the formula.

```
ccs> test [coin]<^coffee>tt CTM
True
ccs> test [coin](<^coffee>tt & <^tea>tt) CTM
True
ccs> test [coin]<^beer>tt CTM
False
```

Since our implementation does not handle recursive HML formulas, we cannot define meaningful operators based on fixpoints, such as *possibility*, *invariantly* etc. Instead, we can check for reachability by using search. This functionality is provided by the `reachable` command, which, given an HML formula and a process, traverses the state graph generated by the process and looks for a process that satisfies the formula. We can for example model a computer scientist in the classical way and look for deadlocks.

```
ccs> def CS = ^coin.coffee.^pub.CS
ccs> reachable (
...>    [coin]ff & [^coin]ff
...>  & [coffee]ff & [^coffee]ff
...>  & [tea]ff & [^tea]ff
...>  & [pub]ff & [^pub]ff
...>  & [tau]ff )  (CTM | CS)\{coin,coffee,tea}
Not found
```

The formula we chose here matches a state where no action is possible, we do this by simply creating a conjunction of $[a]ff$ for any possible action a, as $[a]ff$ simply means that it is not possible to perform an a-action.

`Not found` means that no state in the graph satisfied the formula, i.e. exhibited a deadlock. A faulty coffe and tea machine can however easily generate a deadlock.

```
ccs> def BadCTM = ( coin.^coffee.BadCTM
...>              + coin.^tea.BadCTM )
ccs> reachable (
...>    [coin]ff & [^coin]ff
...>  & [coffee]ff & [^coffee]ff
...>  & [tea]ff & [^tea]ff
...>  & [pub]ff & [^pub]ff
...>  & [tau]ff )   (BadCTM | CS)\{coin,coffee,tea}
Found: (^tea.BadCTM | coffee.^pub.CS) \ {coffee,coin,
    tea}
```

The search yielded a state where we can see that the machine offers tea but the computer scientist is only willing to accept coffee, hence a deadlock arises.

7 Appendix: CCS and HML semantics

The semantics of CCS and HML are thoroughly described in [1], we reproduce parts of it here for quick reference.

We assume a countably infinite set of *action* (or *signal*) names $\mathcal{A}$. From this we construct the set of complementary names

$$\overline{\mathcal{A}} = \{\overline{a} \mid a \in \mathcal{A}\}$$

Intuitively, if two parallel processes are willing to perform the actions a and $\overline{a}$ respectively, they are able to communicate by performing these actions in tandem. We then define the set

$$\mathsf{Act} = \mathcal{A} \cup \overline{\mathcal{A}} \cup \{\tau\}$$

where τ represents the special *silent action*, i.e. an action where no input- or output signal is observed. By convention, we assume that $\overline{\overline{a}} = a$ for any action a.

We also assume a countably infinite set of process names $\mathcal{K}$. Our convention will be to write elements from $\mathcal{A}$ in all-lowercase and elements from $\mathcal{K}$ starting with a capital letter.

We can now describe the syntax of CCS expressions by the grammar

$$P, Q ::= K \mid \alpha.P \mid \sum_{i \in I} P_i \mid P|Q \mid P[f] \mid PL$$

where

- $K \in \mathcal{K}$
- $\alpha \in \mathsf{Act}$
- I is a (possibly infinite) index set
- $f : \mathsf{Act} \to \mathsf{Act}$ is a relabeling function having $f(\tau) = \tau$ and $f(\overline{a}) = \overline{f(a)}$ for all actions a.

We also assume that for any K appearing in a CCS expression, we have a defining equation of the form

$$K \stackrel{\text{def}}{=} P$$

We use $\mathbf{0}$ as a shorthand for the empty sum of processes, representing the *nil* process, one that affords no actions. Note that we can define recursive processes, such as

$$X \stackrel{\text{def}}{=} a.X$$

The formal semantics of CCS expressions can now be de-

fined with structural operational semantic rules.

$$\frac{}{\alpha.P \xrightarrow{\alpha} P} \quad \text{(ACT)}$$

$$\frac{P_j \xrightarrow{\alpha} P'_j}{\sum_{i\in I} P_i \xrightarrow{\alpha} P'_j} \quad \text{where } j \in I \quad (\text{SUM}_j)$$

$$\frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \quad (\text{COM1})$$

$$\frac{Q \xrightarrow{\alpha} Q'}{P|Q \xrightarrow{\alpha} P|Q'} \quad (\text{COM2})$$

$$\frac{P \xrightarrow{a} P' \quad Q \xrightarrow{\overline{a}} Q'}{P|Q \xrightarrow{\tau} P'|Q'} \quad (\text{COM3})$$

$$\frac{P \xrightarrow{\alpha} P'}{P\backslash L \xrightarrow{\alpha} P'\backslash L} \quad \text{where } \alpha, \overline{\alpha} \notin L \quad (\text{RES})$$

$$\frac{P \xrightarrow{\alpha} P'}{P[f] \xrightarrow{f(\alpha)} P'[f]} \quad (\text{REL})$$

$$\frac{P \xrightarrow{\alpha} P'}{K \xrightarrow{\alpha} P'} \quad \text{where } K \stackrel{\text{def}}{=} P \quad (\text{CON})$$

Hennessy-Milner logic

In our implementation, we can test processes both for syntactic equivalence and also if they satisfy a finite (non-recursive) HML formula. The formal syntax of HML formulae is as follows

$$F, G ::= tt \mid ff \mid F \wedge G \mid F \vee G \mid \langle a \rangle F \mid [a]F$$

where $a \in \text{Act}$. Formal, denotational, semantics are presented in [1] on page 91, we make do with informal descriptions here.

- All processes satisfy tt.
- No processes satisfy ff.
- $\wedge$ and $\vee$ have their normal, boolean-logic meaning
- $\langle a \rangle F$ is satisfied by a process P iff P can do an a action and become a process that satisfies F.
- $[a]F$ is satisfied by a process P iff however P does an a action, the resulting process satisfies F.

Note that the last one is vacuously true if P cannot do an a action, while the second last is always false if P cannot do an a, regardless of the formula F.

References

[1] Luca Aceto, Anna Ingolfsdottir, Kim G. Larsen and Jiří Srba, *Reactive Systems – Modelling, Specification and Verification*, Cambridge University Press, 2007.

[2] Daan Leijen, *Parsec, monadic parser cominator library for Haskell*, `http://legacy.cs.uu.nl/daan/parsec.html`

[3] Robert Dockins, *Shellac, a library for creating read-eval-print shells*, `http://www.cs.princeton.edu/~rdockins/shellac/home/`

[4] Richard E. Korf. *Depth-first iterative-deepening: An optimal admissible tree search*, Artificial Intelligence, Vol. 27, No. 1, 1985, pp.97–109.

[5] P.E. Hart, N.J. Nilsson and B. Raphael, *A formal basis for the heuristic determination of minimum cost paths*, IEEE Transactions on Systems Science and Cybernetics, Vol. SSC-4, No. 2, July 1968, pp.263–313.