Implementing a "CCSVP to CCS" translator and interpreter

pietronky

...

Introduction

Goal

Implementing a program able to

- Receive a CCSVP source code as input
  and translate it into a pure CCS source code
- Interpret the resulting pure CCS source code

Note that CCSVP theory allows passed data to range among all natural numbers. This fact may not be easily reproduced during the coding phase, since you would end up dealing with processes whose definition is infinite.
That's why allowed data types will be integers taken from a user defined range and user defined enum values.

Chosen programming language

The program is **entirely** written in `Haskell`.
`Haskell` has been chosen mainly because of these characteristics

- **Expressiveness**
  Using an expressive language means focusing mostly on the application domain related concepts during the development process.
- **Functional paradigm**
  Using a functional language means encountering no bugs related to a program's changing state during the development process, thanks to *referential transparency*.
- **High-level**
  Using a high-level language means being free from explicit memory management, thanks to *garbage collection*.

CCSVP source code structure

```
{-# OPTIONS_CCS -nat-max 10 #-}
F2 = in(x:Nat).F1(x);
F1(x:Nat) = out(x).F2 + in(y:Nat).F0(x,y);
F0(x:Nat,y:Nat) = out(x).F1(y);
G = in(5).G;
Main = (G | F2) \ {in};
```

The program above, which implements a 2-sized FIFO buffer, is an example of CCSVP source code

Note that the programmer must specify a roof for integers and parametric constants must have a typed signature.

Having a roof for integers means that any arithmetic expression (including numbers) overflowing such roof would be evaluated as the roof itself.

CCSVP source code structure

Pay attention to channels, as they're not typed.
More specifically, for what concerns input channels, even if the programmer must specify a type for input variables, this doesn't mean that he's also forced to send equally typed values along the corresponding output channel.
In other words, sending a value of different type would simply disallow synchronization.

This choice respects Plotkin's structural operational semantics, as the rule for synchronization in CCSVP is the following, given P, Q, P' and Q' generic processes, a a channel and v a value (of any type)

$$\frac{P \xrightarrow{a(v)} P' \qquad Q \xrightarrow{\overline{a}(v)} Q'}{P|Q \xrightarrow{\tau} P'|Q'}$$

CCSVP source code structure

```
{-# OPTIONS_CCS -nat-max 15 #-}
Is_odd(x:Nat)  = if x = 0 then return(False).0 +
                 if ¬(x = 0) then Is_even(x - 1);
Is_even(x:Nat) = if x = 0 then return(True).0 +
                 if ¬(x = 0) then Is_odd(x - 1);
Main = Is_odd(15);
```

The program above, which is a naïve implementation of parity checking, shows the utilization of enum values.
The choice is not to force the programmer to explicitly specify a series of enums. Instead, before translation happens, every enum value is read from the source code and the resulting set is considered as the whole set of usable enum values.
The overall universe of values will be the union of such a set with $[0, \texttt{nat-max}]$.

CCSVP source code structure

```
{-# OPTIONS_CCS -nat-max 10 #-}
Main = in(x:Enum).Waldo(x);
Waldo(x:Enum) = out(x).0;
```

Of course the aforementioned approach has some drawbacks.
For instance, consider the program above.
Channel `in` is willing to accept an enum value as input, but in the source code no enum value appears.
Note that, however, cases like this are automatically managed in the translation process.
As a matter of fact, given $[\![\ ]\!] : \text{CCSVP} \to \text{CCS}$ the translation function, `in(x:Enum).Waldo(x)` would be translated as $\sum_{v \in \varnothing} \texttt{in}_{\texttt{v}}.[\![\texttt{Waldo(x)}]\!]$, which would be equivalent to 0 (the inaction).

CCSVP source code structure

```
{-# OPTIONS_CCS -nat-max 25 #-}
Main = Garply(5, 5, 10);
Garply(x:Nat,y:Nat,z:Nat) = if y = 5 then (in(x:Nat).Waldo(x,z) + Waldo(x,z));
Waldo(x:Nat,z:Nat) = out(x + z).0;
```

Last but not least, programmers must be aware of some scoping rules.
For example, in `Garply`, `x` is shadowed due to `in(x:Nat)` in the first call of `Waldo(x,z)`, but it is not shadowed in the second call of `Waldo(x,z)`.
Of course the programmer can choose to force a specific scope,
for example writing `in(x:Nat).(Waldo(x,z) + Waldo(x,z))` (even if in this specific case it wouldn't make sense due to bisimilarity rules).

CCSVP source code parsing

In order to perform CCSVP source code translation more easily and interpret the translation's result afterwards, CCSVP source code must be parsed to produce an adequate data structure.

A CCSVP source code is basically a series of definitions of constants, among which the special `Main` constant must be defined. `Main` will be the translated program's entry point during interpretation.

The aforementioned series of definitions can be represented with a `Map` data type, where keys will be the definitions *l-values* and values will be pairs composed by a signature (meaning a sequence of "parameter-type" pairs) and the definitions *r-values*, represented with a new *ad hoc* data type.
Such data type's structure will be gradually discovered in the following slides.

The Expression module

Since CCSVP theory allows to receive and send values on channels, one of the main data types to define is `Expr`.
Such data type represents an expression which can be sent on an output channel.

```
data Op
= Div
| Minus
| Mul
| Plus
deriving (Eq)
```

```
data Expr
= Arithmetic Expr Op Expr
| EEnum String
| ENat Int
| Var String
```

The Expression module

Moreover, CCSVP theory offers conditional constructs and therefore also boolean expressions are needed as data types.

```
data BOp
= Eq
| Neq
| Lt
| Geq
```

```
data LOp
= And
| Or
```

```
data BExpr
= Comparison Expr BOp Expr
| Not BExpr
| Binary BExpr LOp BExpr
```

The Process module

Since processes could perform some actions, having a data type for those seems sensical.
Such data type is defined as follows

```
data Act
= In (Chan, Maybe (Var, Ty))
| Out (Chan, Maybe Expr)
| Tau
```

The `In` data constructor is for input channels and it accepts as input a pair composed by

- A `Chan` (which is a `String`, since `Chan` is only a type alias)
- A `Maybe` value, which in its `Just` case holds a pair composed by a `Var` (always a type alias for a `String`) and a `Ty` value, which represents a type ranging on $\{\texttt{Enum}, \texttt{Nat}\}$.
  When such `Maybe` value is `Nothing`, the represented action is an input on a "non value passing" channel

The Process module

```
data Act
= In (Chan, Maybe (Var, Ty))
| Out (Chan, Maybe Expr)
| Tau
```

The `Out` data constructor is for output channels and it accepts as input a pair composed by

- A `Chan`
- A `Maybe` value, which in its `Just` case holds an `Expr`.
  In a similar way with respect to the `In` data constructor, when such `Maybe` value is `Nothing`, the represented action is an output on a "non value passing" channel

`Tau` represents the silent action.

The Process module

```
data Proc
= ActPrefx Act Proc
| Choice [Proc]
| Cond BExpr Proc
| Const Name (Maybe [Expr])
| Inaction
| Par Proc Proc
| Relabeling Proc [(Chan, Chan)]
| Restriction Proc [(Chan, Bool)]
```

The data type shown above represents a CCSVP process.
A few clarifications

- When the list of choices of the `Choice` data constructor is empty, the process is seen as an inaction (this is in complete agreement which CCSVP theory).
  Since we have a **list** rather than a **set** like in CCSVP theory, given P an arbitrary process one may perfectly write $P + P$
- For what concerns the `Const` data constructor, while `Name` is still an alias of `String`, the second argument is a `Just` if and only if the constant is parametric
- Even if relabeling would require a (relabeling) function rather than a list of "new name - old name" pairs, only this kind of relabeling is actually supported.
  Current euristics is that it is the kind of relabeling most likely to be used.
- The `Restriction` data constructor accepts a list of pairs such that their second elements are boolean values which are `True` if and only if the restricted channel is an input one.

CCSVP source code checking

Once the CCSVP source code has been parsed, thanks to `Alex` (`Lex`) in synergy with `Happy` (`Yacc`), a few checks must be performed in order to guarantee that code is semantically consistent.
The performed checks are the followings, taken in order

1. Check for duplicated definitions
2. Type checking
3. Check for channels which are used both as value passing and as "standard" within the same process

Translation rules

$[\![a.P]\!] = a.[\![P]\!]$

$[\![a(x : t).P]\!] = \sum_v a_v.[\![P\{v/x\}]\!]$

where v is a value of type t

and only the free occurences of x are substituted by v

$[\![\overline{a}.P]\!] = \overline{a}.[\![P]\!]$

$[\![\overline{a}(e).P]\!] = \overline{a_v}.[\![P]\!]$

where e evaluates to v

$[\![\tau.P]\!] = \tau.[\![P]\!]$

$[\![\sum_i P_i]\!] = \sum_i [\![P_i]\!]$

$[\![P|Q]\!] = [\![P]\!]|[\![Q]\!]$

$[\![P \setminus L]\!] = [\![P]\!] \setminus (L_1 \cup \{c_v \mid c \in L_2 \wedge v \in \mathcal{V}\})$

where L_1 is the subset of L including only "standard" channels,

L_2 is the subset of L including only value-passing channels

and $\mathcal{V}$ is the overall universe of values

$[\![P[c_1'/c_1, \ldots, c_n'/c_n]]\!] = [\![P]\!][f(c_1'/c_1), \ldots, f(c_n'/c_n)]$

where $f(c_i'/c_i)$ is c_i'/c_i if c_i is a "standard" channel

and a list of c_{i_v}'/c_{i_v} (where $v \in \mathcal{V}$) otherwise

---

$$[\![\text{if } e \text{ then } P]\!] = \begin{cases} [\![P]\!] & \text{if } e \text{ evaluates to true} \\ 0 & \text{otherwise} \end{cases}$$

$[\![K(e_1, \ldots, e_n)]\!] = K_{n_1,\ldots,n_n}$

where e_1 evaluates to n_1, …, e_n evaluates to n_n,

$K(x_1, \ldots, x_n) = P$ for some P

and $K_{n_1,\ldots,n_n} = [\![P\{n_1/x_1, \ldots, n_n/x_n\}]\!]$

The Translation module

As the previous slide suggests, the main increment to the Haskell code must be a function able to perform the replacement of a variable's free occurrences with a value taken from the overall universe (recall that such universe comes from the parsing process).

Note that expressions, boolean expressions and processes share a common characteristic: each of those can be subject to a variable replacement.
Therefore it seems sensical to exploit a Parametric type class declaring a funcion replace and make Expr, BExpr and Proc instances of such class.

```
class Parametric a where
    replace :: Var -> Val -> a -> a
```

The Translation module

```haskell
class Parametric a where
    typeCheck :: [(Var, Ty)] -> a -> Either String a
```

Note that the `Parametric` type class also declares a `typeCheck` function, exploted during the type checking process.

For instance, this is how a boolean expression is type checked

```haskell
getType :: Scope -> Expr -> Either String Ty
getType scope e@(Arithmetic e1 _ e2) =
    do t1 <- getType scope e1
       t2 <- getType scope e2
    case (t1, t2) of
        (Nat, Nat) -> return Nat
        _ -> Left ("type mismatch [Nat, Nat] / " ++
                   show [t1, t2] ++ " in " ++ show e)
getType _ (EEnum _) = return Enum
getType _ (ENat _) = return Nat
getType scope (Var x) =
    case find (\(y, _) -> x == y) scope of
        Nothing -> Left ("Couldn't solve " ++
                         x ++ " within scope")
        Just (_, t) -> return t
```

```haskell
typeCheck scope e@(Comparison e1 op e2) =
    do t1 <- getType scope e1
       t2 <- getType scope e2
       case (t1, op, t2) of
           (Nat, _, Nat) -> return e
           (Enum, Eq, Enum) -> return e
           (Enum, Neq, Enum) -> return e
           _ -> Left ("type mismatch " ++
                      show [t1, t2] ++ " in " ++ show e)
typeCheck scope (Not e) = typeCheck scope e
typeCheck scope e@(Binary e1 _ e2) =
    typeCheck scope e1 >> typeCheck scope e2 >> return e
```

The Translation module

```
{-# OPTIONS_CCS -nat-max 5 #-}
Main = Bar(5);
Bar(x:Nat) = if x = 5 then Bar(x);
```

```
{-# OPTIONS_CCS -nat-max 20 #-}
Main = Foo(5);
Foo(x:Nat) = Fie(x + 5);
Fie(x:Nat) = Fee(x + 10);
Fee(x:Nat) = Foo(x - 15);
```

Having implemented all the aforementioned stuff, the translation function shown before can be encoded in `Haskell` almost for free.
There is however still a problem: cyclic definitions, like in the snippets above.
A naïve implementation could therefore not account such type of definitions and result in possibly looping translation processes.

The Translation module

In order to smoothly complete translation the following approach is chosen, starting from `Main`'s definition.
Note that during translation a set S_1 is carried on in order to track the names of processes which have already been translated.

1. Let R be the *r-value* of current process and let C be its name.
   C is added to S_1.
   R is translated, following the rules which have already been shown.

   Constants occurring in R are managed as follows:

   - any non parametric constant K is not recursively translated, but instead added to a new set S_2, tracking the names of processes which **must** still be translated.
   - any parametric constant K' is replaced with a new constant $K'_{v_1,\ldots,v_n}$ (where each v_i is the evaluation of K''s i^{th} argument).
     $K'_{v_1,\ldots,v_n}$ also becomes an element of S_2.

2. The environment is updated, setting C to R', which is R's translation.

3. S_2 is updated as $S_2 \setminus S_1$ and the translation stops is $\varnothing$ is the result.

4. Now assume that S_2 is not empty.
   Let $K'_{v_1,\ldots,v_n}$ be any new constant in S_2 resulting from the translation of a parametric constant K'.
   Let $x_1, \ldots, x_n$ be K''s (untyped) signature and let P be K''s *r-value*.
   Then $K'_{v_1,\ldots,v_n}$ is added to the environment, having $P\{v_1/x_1, \ldots, v_n/x_n\}$ as *r-value*

5. Every element left in S_2 is recursively translated

The `Execution` module

Once all the definitions required for `Main` have been translated, the CCS source code can be finally executed.
For this purpose a type alias `Step` is defined, standing for a pair `(Act, Proc)`.
A function `nextSteps` is defined in order to return all possible steps that a process can perform according to Plotkin's structural operational semantics.
Starting from `Main`, such function is repeatedly called, and a random element is taken from the set of possible steps of current process.
If such a set gets empty the interpreter terminates warning the user about a deadlock.