Automatic Generation of CINNI Instances for the Maude System[*]

Jonas Eckhardt[†] Tobias Mühlbauer[‡] José Meseguer[§]

September 27, 2011

Abstract

Many formal languages use the concept of names to range over essential entities of the language and are usually equipped with special binding constructs for names. For example, the λ-calculus uses variables as names and λ-abstractions as a name binders; Milners π-calculus uses the action prefixes *in* and *new* to bind variables in a subsequent term; and first-order logic uses variables as names, which can be bound by the $\forall$ and $\exists$ quantifiers.

CINNI is a calculus of explicit substitutions that contributes a first-order representation of terms which takes variable bindings into account and captures free substitutions. The CINNI calculus is parametric in the syntax of the object language, which allows it to be applied to many different object languages.

The createCINNI tool makes the parametric nature of CINNI available to the Maude system by means of an automatic module transformation which — given a Maude module specifying the syntax of an object language L — generates a Maude module containing the instantiation CINNI$_L$.

1 Introduction

Many formal languages use the concept of *names* to range over essential entities of the language and are usually equipped with special *binding constructs* for names. For example, the λ-calculus uses variables as names and λ-abstractions as name binders; Milner's π-calculus [6] uses the action prefixes *in* and *new* to bind variables in a subsequent term; and first-order logic uses variables as names, which can be bound by the $\forall$ and $\exists$ quantifiers.

Stehr [7] proposes CINNI, a calculus of explicit substitutions that contributes a first-order representation of terms which takes variable bindings into account

[*]Research partially supported by NSF Grant CCF 09-05584.

[†]eckharjo@in.tum.de

[‡]muehlbau@in.tum.de

[§]meseguer@cs.uiuc.edu

and captures free substitutions. The CINNI calculus is parametric in the syntax of an object language, which allows it to be applied to many different object languages. Stehr shows applications of CINNI to the λ-, ς-, and π-calculi in Maude. In other work [1, 8], the CINNI calculus is applied to manage names and bindings in various host languages.

In this work, we make the parametric nature of CINNI available to Maude users by means of an automatic module transformation which — given a Maude module specifying the syntax of an object language L — generates a Maude module containing the instantiation CINNI_L.

We first provide an overview of the CINNI calculus. Then, as a running example, we apply the CINNI calculus to Millner's π-calculus, and give, based on that example, an idea how the transformation can be generated automatically using reflection. Finally, we describe how the transformation works in more detail and present the Maude tool *createCINNI*, which performs the module transformation in Full Maude.

2 CINNI

Stehr [7] proposes CINNI, a calculus of explicit substitutions that contributes a first-order representation of terms which takes variable bindings into account and captures free substitutions. For a given language L and its defining syntax, the instantiation of CINNI for L is denoted by CINNI_L. Stehr tries to stay as close as possible to the standard name notation while at the same time including the canonical representation of the de Bruijn notation [5] as a special case, in which a single name is used. CINNI uses the Berklin notation [2, 3] that unifies indexed and named notations. In the Berklin notation, each variable name X is annotated with an index $i \in \mathbb{N}$ which represents the position of the binder in the term that binds X_i. The index i of X_i thereby indicates that the binder that binds the variable X is the ith binder to the left of the variable in the term.

Example 1: Berklin notation
The following example illustrates the Berklin notation. Variable X_0 is bound by the second binder while variable X_1 is bound by the first binder in the term.

$$\begin{tikzcd}[column sep=small] \forall X. & \forall X. & f(X_0) \arrow[l, bend left] & \wedge & f(X_1) \arrow[llll, bend left] \end{tikzcd}$$

CINNI extends a given language L with explicit substitutions as shown in equations 1, 2 and 3. The simple substitution $[X := M]$ replaces variable X_0 with value M and reduces the index of any other equally named variable X_{n+1} to X_n. The shift substitution $\uparrow_X$ for variable X increases the index of variables with the same name X. The lifted substitution $\Uparrow_X (S)$ is defined in equations 4, 5, and 6. It decreases the index of variables with the name X, performs the substitution S, and finally lifts the variable.

$$[X := M] \qquad \text{(simple substitution)} \tag{1}$$

$$\uparrow_X \qquad \text{(shift substitution)} \tag{2}$$

$$\Uparrow_X (S) \qquad \text{(lifted subsitution)} \tag{3}$$

$$\Uparrow_X (S)X_0 = X_0 \tag{4}$$

$$\Uparrow_X (S)X_{n+1} = \uparrow_X ((S)X_n) \tag{5}$$

$$\Uparrow_X (S)Y_n = \uparrow_X ((S)Y_n) \text{ if } X \neq Y \tag{6}$$

For each syntactical constructor f of the language L, CINNI adds a *syntax-specific equation* which automatically shifts the bound variables in each argument of the constructor. Let $j_{i,1}, \ldots j_{i,m_i}$ be the arguments that are bound by f in argument i, then the *syntax-specific equation* is defined by:

$$S\, f(P_1, \ldots, P_n) = f(\Uparrow_{P_{j_{1,1}}} (\ldots \Uparrow_{P_{j_{1,m_1}}} (S))P_1, \ldots, \Uparrow_{P_{j_{n,1}}} (\ldots \Uparrow_{P_{j_{n,m_n}}} (S))P_n)$$

3 Running Example: CINNI$_\pi$

In the following, we will describe the application of CINNI to Millner's π-calculus. We will later use this running example to illustrate out module transformation. Process terms are represented by the sort `Trm` and channels by the sort `Chan`. Process terms can be concatenated by the associative and commutative parallel composition operator `P|Q` for which the null process `nil` acts as identity. For a process term `P`, the term `out CX <CY>. P` represents a process that sends the channel `CY` over the channel `CX` and then continues with `P`. The term `in CX [Y]. P` represents a process term that receives a channel name over the channel `CX` and then continues with `P`. Finally, the term `new [Y] P` represents a process term that creates a new local name that can be used in `P` and then continues with `P`. The functional Maude module

```
fmod PI-SYNTAX is
  protecting QID .
  sorts Chan Trm .

  op _{_} : Qid Nat -> Chan .
  op nil : -> Trm [ctor] .
  op _|_ : Trm Trm -> Trm [ctor assoc comm id: nil] .
  op new[_]_ : Qid Trm -> Trm [ctor] .
  op out_<_>._ : Chan Chan Trm -> Trm [ctor] .
  op in_[_]._ : Chan Qid Trm -> Trm [ctor] .
endfm
```

describes the syntax of the π calculus.

The two terms `in CX [Y] P` and `new [Y] P` bind the channel name `Y` in the subsequent process `P`. The variables

```
vars X Y : Qid .
vars CX CZ : Chan .
vars P Q : Trm .
var S : Subst .
vars n : Nat .
```

are used in the following equations.

The application of CINNI to the syntax of the π-calculus adds a sort that represents substitutions (`sort Subst`), new operators and equations for the three kinds of substitution: simple, shift and lifted.

```
op [_:=_] : Qid Chan -> Subst .
op [shift_] : Qid -> Subst .
op [lift__] : Qid Subst -> Subst .
op __ : Subst Chan -> Chan .
op __ : Subst Trm -> Trm .

eq [X := CZ] X{0} = CZ .
eq [X := CZ] X{suc(n)} = X{n} .
ceq [X := CZ] Y{n} = Y{n} if X =/= Y .

eq [shift X] X{n} = X{suc(n)} .
ceq [shift X] Y{n} = Y{n} if X =/= Y .

eq [lift X S] X{0} = X{0} .
eq [lift X S] X{s(n)} = [shift X] (S (X{n})) .
ceq [lift X S] Y{n} = [shift X] (S (Y{n}))
   if X =/= Y .
```

Additionally, *syntax-specific equations* are added to the module. A substitution that is applied to the `nil` process is discarded. If a substitution is applied to the parallel composition of two process terms, it is applied to each process term individually. As `nil` acts as the identity for the associative and commutative parallel composition operator, it is important not to apply the substitution to a composition where one of the subterms is the process term `nil`. Similarly to the parallel composition, if a substitution is applied to the process term `out CX<CZ>.M`, the substitution is simply applied to all subterms. The two process terms `in CX[Y].M` and `new[Y]M` bind the name `Y` in the subsequent process term `M`, so the lifted substitution `[lift Y S]` has to be applied to `M`.

```
eq S nil=nil.
ceq S (P | Q) = (S P) | (S Q)
   if P =/= nil and Q =/= nil .
eq S (out CX<CZ>.P) = out (S CX)<S CZ>.(S P).
eq S (in CX[Y].P) = in(S CX)[Y].([lift Y S] P).
eq S (new[Y]P) = new[Y]([lift Y S]P).
```

Discussion

Using Berklin's representation, a requirement for an automatic generation of CINNI specifications is that the language differentiates between *names*, *indexed names*, and *values*. *Names* are used in binding expressions, *indexed names* are names quantified by an index that are bound to a *name* by binding operators, and *values* are terms of the language that can be substituted for *indexed names*. In our example of Millner's π calculus, these entities were mapped to entities of the target language as follows:

$$\begin{aligned} \textit{name} &\mapsto \textbf{sort}\ \texttt{Qid} \\ \textit{indexed name} &\mapsto \textbf{sort}\ \texttt{Chan} \\ \textit{value} &\mapsto \textbf{sort}\ \texttt{Chan} \end{aligned}$$

If this mapping is given, the CINNI transformation can be fully automated. The transformation consists of two main steps:

1. A sort that represents substitutions and operators for the simple, shift, and lifted substitutions are added. Additionally, for each constructed sort of the language, e.g., `Chan` and `Trm`, an operator to prefix substitutions is added. Then, equations defining the semantics of the simple, shift, and lifted substitution are added. These equations need to be aware of the actual sorts for *names*, *indexed names*, and *values*.

2. A *syntax-specific* equation is added for each syntactic constructor of the language. Basically, there are two types of syntactic constructors: those that bind *names* in one or more of the arguments and those that don't. For example, the syntactic constructor `new[_]_` binds the first argument in the second whereas the parallel operator `_|_` does not bind any names. The *syntax-specific* equations describe the effect of applying a substitution a term built with the constructor by passing down the substitution to each of the arguments. If the constructor binds a *name* in an argument, the *name* is lifted in the substitution of that argument.

The information about which *names* are bound in which argument cannot be derived directly from the first-order declaration of a syntactic constructor. Our transformation uses Maude's `metadata` attribute so that the user can add this binding information. The two constructors `new[_]_` and `in_[_]` bind the first in the second argument. Thus they are be annotated accordingly.

```
op new[_]_ : Qid Trm -> Trm [ctor metadata "1->2"] .
op in_[_]._ : Chan Qid Trm -> Trm [ctor metadata "2->3"] .
```

Having the mapping between *names*, *indexed names*, and *values* and the corresponding sorts in the language together with the information about which *name* is bound in which argument by the syntactic constructors enables us to create a fully automated module transformation. The next section describes the transformation in more detail.

4 The Transformation

Figure 1 shows a top level view of the transformation. The transformation lifts the source module to the meta-level, renames the module, and adds sorts, operators, and equations to the meta-representation of the module.

The two overloaded operators

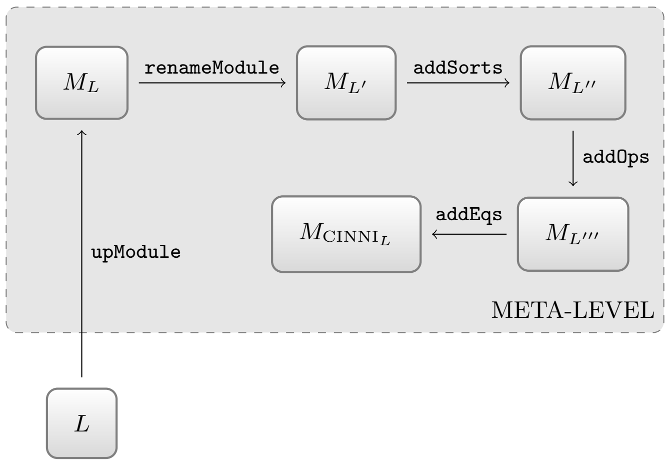

Figure 1: Top level view on the module transformation

```
op cinni : Qid Type Type Type Type -> Module .
op cinni : Module Type Type Type Type -> Module .
```

only differ in their first argument: One can either specify the module's name or give directly the meta representation of the source module. Additionally, one has to specify the sorts that will be used for: (i) substitutions, (ii) *names*, (iii) *indexed names*, and (iv) *values*, where sorsts (i)–(iv) are sorts of the source module. To apply the transformation on our running example, the transformation is executed using the command

```
red cinni('PI-SYNTAX, 'Subst, 'Qid, 'Chan, 'Chan) .
```

The equation for the `cinni` operator, which takes the name of the source module as an argument, uses the `upModule` operator to create the meta representation of the source module. The invocation of the `cinni` operator with the resulting meta-representation of the module is then evaluated by the second equation, which performs the transformation.

```
ceq cinni(MOD, SUBSTT, NAMET, INAMET, VALT) =
  cinni(M, SUBSTT, NAMET, INAMET, VALT)
  if M := upModule(MOD, false) .

ceq cinni(M, SUBSTT, NAMET, INAMET, VALT) = MWE
  if RM := renameModule(M, qid("CINNI-" + string(getModuleName(M)))
      {getParameterDeclList(M)} )
    /\ CTL := removeDoubles(getConstructedTypes(getOps(M)))
    /\ MWS := addSorts(RM, SUBSTT)
    /\ MWO := addOps(MWS, simpleSubst(NAMET, VALT, SUBSTT)
                   shiftSubst(NAMET, SUBSTT)
                   liftSubst(NAMET, SUBSTT)
                   substOps(removeDoubles(CTL INAMET),SUBSTT))
    /\ MWE := addEqs(MWO, substBase(NAMET, VALT)
                   shiftEqs(NAMET)
                   liftEqs(NAMET, SUBSTT)
```

```
                        createSpecificEqs(getOps(M), SUBSTT,
                          NAMET, INAMET, CTL)
                        createIdentityEq(CTL, SUBSTT)) .
```

The effect of applying the second equation is as follows. First, the name of the source module is prefixed with the string `CINNI` using the `renameModule` operator. Then, the following sorts, operators, and equations are added using the `addSorts`, `addOps`, and `addEqs` operators, respectively:

- The specified sort for substitutions.
- Operators for the three types of substitutions.
- For each constructed sort in the source module, one operator for prefixing substitutions.
- Equations defining the semantics of the simple, shift, and lifted substitution.
- One *syntax-specific equation* for each syntactic constructor of L.
- Identity equations are added to avoid unnecessary substitutions.

The operators to rename a module, and to add sorts, operators, or equations are not described here. A detailed description of these operators is given in the subsection "A Deadlock-Freedom Transformation" of [4, p. 480]. The automatic creation of the CINNI operators and equations is described in more detail in the next subsections. Subsection 4.1 describes how the CINNI operators are created, and Subsection 4.2 shows how the CINNI equations, including the *syntax-specific equations*, are created.

4.1 Creation of CINNI operators

The meta-representation of a Maude operator is a term of sort `OpDecl` and is defined in the Maude module `META-MODULE`. Terms of sort `OpDeclSet` represent sets of operators and can be concatenated using the associative and commutative operator `__` for which the term `none` acts as identity. We construct the operators for the simple, shift, and lifted substitution by using the auxiliary operators `simpleSubst`, `shiftSubst`, and `liftSubst`, respectively. The equation

```
eq simpleSubst(NAMET, VALT, SUBSTT)
  = (op '`[_:=_`] : NAMET VALT -> SUBSTT [none] . ) .
```

takes the type of *names, values*, and *substitutions*, as argument, and yields the meta-representation of the simple substitution operator. In our running example, the term

```
simpleSubst('Qid, 'Chan, 'Subst)
```

would be reduced to the following term, which is the meta-representation of the simple substitution operator of CINNI$_\pi$.

```
op '`[_:=_`] : 'Qid 'Chan -> 'Subst [none] .
```

The equations

```
eq shiftSubst(NAMET, SUBSTT)
  = (op '`[shift_`] : NAMET -> SUBSTT [none] . ) .
eq liftSubst(NAMET, SUBSTT)
  = (op '`[lift__`] : NAMET SUBSTT -> SUBSTT [none] . ) .
```

take the type of *names* and *substitutions*, and create the meta-representation of the shift and lifted substitution.

Additionally, for each sort that is constructed in the source module, the corresponding operators to prepend substitutions are added. The operator `substOps` is defined recursively on the structure of the first argument. The base case takes a term of sort `Type`, and the sort of substitutions and creates the meta representation of the prefixing operator for that type.

```
eq substOps(T, ST) = (op '__ : ST T -> T [none] . ) .
```

The two recursive cases

```
ceq substOps(T TL, ST) = substOps(T, ST) substOps(TL, ST) if T =/= nil .
ceq substOps(T TL, ST) = substOps(TL, ST) if T == nil .
```

decompose the first argument — of sort `TypeList` — structurally.

4.2 Creation of CINNI equations

As for the meta-representation of operators, the meta-representation of equations can be found in the Maude module `META-MODULE`. Equations and sets of equations are represented at the meta-level by terms of the sorts `Equation` and `EquationSet`.

We first describe the creation of the equations that define the semantics of the CINNI substitution operators. Then, we describe the creation of the *syntax-specific equations*.

4.2.1 Creation of the equations for the CINNI operators

The auxiliary operators `substBase`, `shiftEqs`, and `liftEqs` create the meta representation of the equations defining the semantics of substitutions. For example, the equations

```
eq shiftEqs(NAMET) =
  shiftEq1(NAMET) shiftEq2(NAMET) .

eq shiftEq1(NAMET) =
  (eq '__['`[shift_`][qid("X:" + string(NAMET))],
    '_`{_`}[qid("X:" + string(NAMET)), 'M:Nat]]
  = '_`{_`}[qid("X:" + string(NAMET)), 's_['M:Nat]] [none] .) .

eq shiftEq2(NAMET) =
  (ceq '__['`[shift_`][qid("X:" + string(NAMET))],
    '_`{_`}[qid("Y:" + string(NAMET)), 'M:Nat]]
  = '_`{_`}[qid("Y:" + string(NAMET)), 'M:Nat]
  if '_=/=_[(qid("Y:" + string(NAMET)),
    qid("X:" + string(NAMET)))] = 'true.Bool [none] .) .
```

$$\begin{tikzcd}
 & \texttt{getOperators(MOD)} \arrow[d, "\texttt{createSpecificEq}"] & \\
 & \textbf{op}\ \texttt{f:T}_1\ \texttt{...}\ \texttt{T}_N\ \texttt{->}\ \texttt{T [ctor metadata M]} \arrow[dl, "\texttt{createLeft}"'] \arrow[dr, "\texttt{createRight}"] & \\
\texttt{X}_1\texttt{:T}_1\texttt{,...,X}_N\texttt{:T}_N \arrow[d] & & \texttt{(S}_1\ \texttt{X}_1\texttt{:T}_1\texttt{),...,(S}_N\ \texttt{X}_N\texttt{:T}_N\texttt{)} \arrow[d] \\
\textbf{eq}\ \texttt{S f(X}_1\texttt{:T}_1\texttt{,...,X}_N\texttt{:T}_N\texttt{)} & \texttt{=} & \texttt{f((S}_1\ \texttt{X}_1\texttt{:T}_1\texttt{),...,(S}_N\ \texttt{X}_N\texttt{:T}_N\texttt{))}
\end{tikzcd}$$

Figure 2: Creation of a *syntac-specific* equation

take the type of *names* as an argument and return the meta-representation of the following two equations of the CINNI calculus:

$$\begin{aligned} \uparrow_X X_m &= X_{m+1} \\ \uparrow_X Y_n &= Y_n \text{ if } X \neq Y \end{aligned}$$

The equations defining the operators `substBase` and `shiftEqs` are omitted for the sake of brevity.

4.2.2 Creation of the *syntax specific equations*

In a last step, the *syntax-specific equations* are created. A schematic overview of the auxiliary functions, that are used, is shown in Figure 2. *Syntax-specific equations* have to be created for each operator of the source module. The operator `getOperators` returns the meta-representation of the operators defined in the given module. For each of these operators, say `f`, the `createSpecific` operator is executed. Using the two auxiliary functions `createLeft` and `createRight`, which create the left-hand and right-hand side of the *syntax-specific equation*, the equations are created.

Let us assume that the declaration for the operator `f` contains as a `metadata` attribute,

$$\textbf{op}\ \texttt{f : T1 ... TN -> T [ctor metadata i}_1\texttt{->j}_1\texttt{,...,i}_M\texttt{->j}_M\texttt{]}$$

with $\texttt{i}_1,\ldots,\texttt{i}_M,\texttt{j}_1,\ldots,\texttt{j}_M \in \{1,...,N\}$. Thus, the operator `f` binds the argument $\texttt{i}_k$ in the argument $\texttt{j}_k$ for $k \in \{1,\ldots,M\}$. Furthermore, we assume that no two *names* are bound in the same argument. This is expressed by the requirement that $\texttt{j}_k \neq \texttt{j}_l$ for all $l \neq k$.

If we are applying a substitution `S` to $\texttt{f(}t_1,\ldots,t_n\texttt{)}$, then the substitutions $\texttt{S}_j$ for the jth argument t_j is of the form $\texttt{[shift X}_i\texttt{:T}_i\ \texttt{S]}$ if `i->j` appeared in the metadata declaration of `f`, i.e., if there is an argument i that is bound in the argument j. Otherwise, if no argument is bound in the argument j, the substitution $\texttt{S}_j$ is equal to `S`. To better illustrate how the algorithm works, we create the *syntax specific equation* based on the operator declaration of the

```
op new[_]_ : Qid Trm -> Trm [ctor metadata "1->2"] .
```

operator of our running example. The meta-representation of the *syntax specific equation* is thereby defined by the equation

```
eq __[S:'Subst, 'new`[_`]_[VAR0:'Qid, VAR1:'Trm]] =
  'new`[_`]_[VAR0:'Qid,'lift[VAR0:'Qid, S:'Subst]] .
```

Creating the left-hand side of a *syntax-specific equation.* Given a list of types and the index of the first variable as argument, the `createLeft` operator creates the meta-representation of the argument list of an equation.

```
op createLeft : TypeList Nat -> NeTermList .

eq createLeft(T TL, INDEX) =
  createLeft(T, INDEX) , createLeft(TL, INDEX + 1) .
eq createLeft(T, INDEX) =
  qid("VAR" + string(INDEX, 10) + ":" + string(T)) .
```

The first equation recursively decomposes the given term of sort `TypeList`, and counts the current argument position in the second argument. If the first argument is a term of sort `Type`, the second equation creates the meta-representation $\texttt{VAR}_i$ of a variable at argument position i. The term

```
createLeft('Qid 'Trm, 0)
```

in our running example is reduced to

```
VAR0:'Qid, VAR1:'Trm
```

which is the left-hand side of the *syntax specific equation* of the operator

```
op new[_]_ : Qid Trm -> Trm [ctor metadata "1->2"] .
```

Creating the right-hand side of a *syntax-specific equation.* The right-hand side of a *syntax-specific equation* also depends on the `metadata` attribute. Thus, the operator `createRight` takes the list of types from the operator declaration, the current index of the variable, the variable for the substitution, a list of types that should be lifted, the mapping between arguments and the bound arguments, and the type of *names* as arguments, and creates the meta-representation of the list of variables for the right-hand side of the equation.

```
op createRight : TypeList Nat Variable TypeList Map{Nat, Nat} Type
  -> NeTermList .
```

The first equation recursively decomposes the first argument of `TypeList`, and increases the index of the current variable. The second equation creates the meta-representation, if the first argument is a term of sort `Type`. If the type is not included in the list of relevant types for substitution, the substitution is omitted. Otherwise, as discussed above, if the argument at the current index binds a *name*, i.e., `$hasMapping(MAP, INDEX)= true`, then the lifted substitution is created. Otherwise, the substitution is simply passed down to the argument.

```
eq createRight(T TL, INDEX, SVAR, TTL, MAP, VNT) =
  createRight(T, INDEX, SVAR, TTL, MAP, VNT) ,
  createRight(TL, INDEX + 1, SVAR, TTL, MAP, VNT) .

eq createRight(T, INDEX, SVAR, TTL, MAP, VNT) =
  if T in TTL then
    if ($hasMapping(MAP, INDEX)) then
      '__['`[lift__`][qid("VAR" + string(MAP[INDEX], 10) + ":" +
      string(VNT)) ,SVAR], qid("VAR" + string(INDEX, 10) + ":" +
      string(T))]
    else
      '__[SVAR, qid("VAR" + string(INDEX, 10) + ":" + string(T))]
    fi
  else
    qid("VAR" + string(INDEX, 10) + ":" + string(T))
  fi .
```

In our running example, the right-hand side of the equation is created by the term

```
createRight('Qid 'Trm, 0, 'S:Subst, 'Qid 'Trm, 1->2, 'Qid)
```

This results in the meta-representation

```
VAR0:'Qid,'lift[VAR0:'Qid, S:'Subst]
```

Creating the *syntax-specific* equation. Bringing it all together, the operators `createSpecificEqs`, and `createSpecificEq` create the *syntax specific equations.*

```
op createSpecificEqs : OpDeclSet Type Type Type TypeList
  -> EquationSet .
op createSpecificEq : OpDecl Type Type Type TypeList -> EquationSet .
```

Except for the first parameter — either a set of operator declarations, or a single operator declaration — both operators take the same parameters: the type of substitutions, the type of *names*, the type of `indexed names`, and a list of types that can be substituted. The operator `createSpecificEqs` recursively creates the *syntax-specific* equations using the `createSpecificEq` operator.

```
eq createSpecificEqs(OPD, SUBSTT, VARNAMET, VART, TERMTL)
  = createSpecificEq(OPD, SUBSTT, VARNAMET, VART, TERMTL) .
ceq createSpecificEqs(OPD OPDS, SUBSTT, VARNAMET, VART, TERMTL)
  = createSpecificEqs(OPD, SUBSTT, VARNAMET, VART, TERMTL)
    createSpecificEqs(OPDS, SUBSTT, VARNAMET, VART, TERMTL)
  if OPDS =/= none .
```

The behavior of the `createSpecificEq` operator is defined in three equations. First, if the operator's definition contains the `ctor` and `metadata S` attributes, `S` is parsed, and the `createLeft` and `createRight` operators are used to construct the resulting meta-representation of the *syntax-specific* equation.

```
ceq createSpecificEq((op N : TL -> TERMT [ctor metadata(S) AS].),
  SUBSTT, VARNAMET, VART, TERMTL) =
  (eq '__[SVAR, N[createLeft(TL, 0)]]
    = N[createRight(TL, 0, SVAR, (TERMTL VART), getPairs(S),
      VARNAMET)] [none] .)
if SVAR := qid("S:" + string(SUBSTT))
  /\ TL =/= nil /\ TL in (TERMTL VART) .
```

Second, if the operator's definition contains the `ctor` and `id(ID)` attribute (but no `metadata` attribute), then the substitution is simply passed down to the arguments. A conditional equation is created, since all arguments are required to be unequal to the identity element of the operator to prevent infinite loops.

```
ceq createSpecificEq((op N : TL -> TERMT [ctor id(ID) AS].),
  SUBSTT, VARNAMET, VART, TERMTL) =
  (ceq '__[SVAR, N[createLeft(TL, 0)]]
    = N[createRight(TL, 0, SVAR, (TERMTL VART), empty, VARNAMET)]
   if createUnequalToId(TL, ID, 0) [none] . )
if SVAR := qid("S:" + string(SUBSTT))
  /\ TL =/= nil /\ TL in (TERMTL VART) .
```

Finally, if the `ctor` attribute is contained in the operator's definition (and no `metadata` or `id` attribute), the *syntax-specific* equation is created using the `createLeft` and `createRight` operators.

```
eq createSpecificEq((op N : TL -> TERMT [AS].),
  SUBSTT, VARNAMET, VART, TERMTL) =
  if ctor in AS and TL =/= nil and TL in (TERMTL VART) then
    (eq '__[qid("S:" + string(SUBSTT)), N[createLeft(TL, 0)]]
    = N[createRight(TL, 0, qid("S:" + string(SUBSTT)),
      (TERMTL VART), empty, VARNAMET)] [none] .)
  else
    none
  fi [owise] .
```

5 The createCINNI Tool

The Full Maude `show module` command is used to retrieve the Maude representation of the meta-module that is created using the `cinni` command. Thus, the created meta-module has to be loaded in the Full Maude database, then printed using the `show module`, and finally the result has to be filtered. The createCINNI tool is defined by the shell script

```
#!/bin/bash
if [ $# -ne 6 ]
then
echo "Usage: ./createCINNI.sh {module name} {substitution sort}
  {name sort} {indexed name sort} {value sort} {module file}"
exit 65
fi

echo "
(select META-LEVEL .)
(fmod CREATE-CINNI is
  ex META-LEVEL .
  ex CINNI-META .
  op module : -> Module .
  eq module = cinni('$1, '$2, '$3, '$4, '$5) .
endfm)
(load module .)
(show module CINNI-$1 .)
```

```
q" | maude -no-prelude -no-banner -no-advise -no-wrap -no-ansi-color
     prelude.maude CINNIMETA.maude $6 full-maude26.maude | awk '/fmod/,/
     endfm/' > CINNI-$6
```

which takes six parameters: The name of the source module, the required sort of substitutions, the sort of *names*, the sort of *indexed names*, the sort of *values*, and a the name of the file containing the source module.

Basically, a new Full Maude module with name `CREATE-CINNI` is created, which extends the `META-LEVEL` and the `CINNI-META` module. Additionally, a constant operator `module` is created that is reduced to the meta-representation of the new module. Then, the new module is loaded into the Full Maude database using the `load` command. Finally, the module is printed using the `show module` command. The result is then filtered with an regular expression and the module is written in a file.

In our example of Milner's π calculus, the createCINNI tool is executed using the command

```
./createCINNI PI-SYNTAX Subst Qid Chan Chan pi-syntax-file
```

The resulting Full Maude module

```
(fmod CREATE-CINNI is
  ex META-LEVEL .
  ex CINNI-META .
  op module : -> Module .
  eq module = cinni('PI-SYNTAX, 'Subst, 'Qid, 'Chan, 'Chan) .
endfm)
```

is loaded in the Full Maude database. The resulting module is then printed and written to the file `CINNI-pi-syntax-file`.

The create CINNI Maude tool can be found on the Maude homepage: http://maude.cs.uiuc.edu/tools/createcinni.

References

[1] M. AlTurki and J. Meseguer. Dist-Orc: A Rewriting-based Distributed Implementation of Orc with Formal Analysis. In *Electronic Proceedings in Theoretical Computer Science*, pages 26–45, 2010.

[2] K. Berkling. *A Symmetric Complement to the Lambda Calculus*, volume 76–77 of *Bonn Interner Bericht ISF*. Gesellschaft für Mathematik und Datenverarbeitung mbH, September 1976.

[3] K. Berkling and E. Fehr. A Consistent Extension of the Lambda Calculus as a Base for Functional Programming Languages. *Information and Control*, 55(1–3):89–101, 1982.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude - A High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[5] N. G. De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 75(5):381–392, 1972.

[6] R. Milner. *Communication and concurrency.* Prentice-Hall, 1989.

[7] M. Stehr. CINNI - A Generic Calculus of Explicit Substitutions and its Application to λ- ς- and π-Calculi. *Electronic Notes in Theoretical Computer Science*, 36:70–92, 2000.

[8] P. Thati. *A theory of testing for asynchronous concurrent systems.* PhD thesis, University of Illinois at Urbana-Champaign, 2003. AAI3101979.