Introduction into HOL-CASL
– Version 0.82 –

Till Mossakowski

Department of Computer Science
and Bremen Institute for Safe Systems,
University of Bremen, Germany.

Comments to: cofi-hol-casl@brics.dk

May 20, 2002

1 Introduction

The HOL-CASL system provides an interface between the CASL specification language [5] and the theorem proving system Isabelle/HOL [10]. We have chosen Isabelle because it has a very small core guaranteeing correctness, and its provers like the simplifier or the tableaux prover are built on top of this core. Furthermore, there is over ten years of experience with it (several mathematical textbooks have partially been verified with Isabelle). We cannot give an introduction to Isabelle/HOL within this text. We refer to the Isabelle tutorial for this purpose; it is included within the distribution. Here just some brief words how CASL has been connected to Isabelle.

Isabelle is a tactic based theorem prover implemented in standard ML. The main Isabelle logic (called Pure) is some weak intuitionistic type theory with polymorphism. The logic Pure is used to represent a variety of logics within Isabelle; one of them being HOL (higher-order logic). For example, logical implication in Pure (written `==>`, also called meta-implication), is different from logical implication in HOL (written `-->`, also called object implication).

CASL now is in turn represented in Isabelle/HOL. Since subsorting and partiality are present in CASL but not in Isabelle/HOL, we have to encode them out. This means that theorem proving is not done in the CASL logic directly, but in the logic HOL (for higher-order logic) of Isabelle. We have tried to make user's life easy by

- choosing a shallow embedding of CASL into HOL, e.g. CASL's logical implication `=>` is mapped directly to Isabelle/HOL's logical implication `-->` (and the same holds for other logical connectives and quantifiers), and

- adapting Isabelle/HOL's syntax to be conform with the CASL syntax, e.g. Isabelle/HOL's `-->` is displayed as `=>`, as in CASL.

However, it is essential to be aware of the fact that the Isabelle/HOL logic is different from the CASL logic. Therefore, the formulas appearing in subgoals of proofs with HOL-CASL will not fully conform to the CASL syntax: they may use features of Isabelle/HOL such as higher-order functions that are not present in CASL, and moreover, they also may contain syntax from the meta-logic Pure (e.g. meta-implication `==>`, meta universal quantification `!!` and meta-variables `?x`).

2 Getting started

HOL-CASL currently comes for Linux and Sparc-solaris. It can be obtained from `http://www.tzi.de/cofi/HOL-CASL/`.
To install the HOL-CASL system, unpack the distribution file. For Linux, you type

```
tar xzvf HOL-CASL-linux.tar.gz
```

For Solaris, you type

```
gunzip HOL-CASL-solaris.tar.gz
tar xvf HOL-CASL-solaris.tar
```

This will produce a directory `HOL-CASL`. cd into this directory. If you should have set the `CASL_LIB` environment variable, you should make sure that it points to the `HOL-CASL` directory. However, you do not need set it; the current directory will be used in this case.
To start the HOL-CASL system, type

```
HOL-CASL
```

The distribution contains a version of the CASL basic datatypes, in the directory `Basic`. Moreover, we also have included an Isabelle tutorial `Isabelle-tutorial.ps`[1].

3 An easy example

The file `Demo_Nat.casl` contains a small specification of natural numbers. It will serve as our running example:

library DEMO_NAT
%prec $\{\_\_ + \_\_\} < \{\_\_ * \_\_\}$
spec NAT =
  free type $Nat ::= 0 \mid suc(Nat)$

[1] Note that this is not the latest version of the Isabelle tutorial, since the latest version uses Isar, which is currently not supported by HOL-CASL.

op $\_\_ + \_\_, \_\_ * \_\_ : Nat \times Nat \to Nat;$
$\mathit{1}, \mathit{2} : Nat;$
forall $m, n : Nat$
- $\mathit{0} + m = m$ $\%(Nat\_add\_0)\%$
- $suc(n) + m = suc(n + m)$ $\%(Nat\_add\_suc)\%$
- $\mathit{0} * m = \mathit{0}$ $\%(Nat\_mult\_0)\%$
- $suc(n) * m = n * m + m$ $\%(Nat\_mult\_suc)\%$
- $\mathit{1} = suc(\mathit{0})$ $\%(Nat\_1\_def)\%$
- $\mathit{2} = suc(\mathit{1})$ $\%(Nat\_2\_def)\%$

end

You can load this specification by typing

```
use_casl "Demo_Nat";
```

This will produce the following output:

```
HOL-CASL> use_casl"Demo_Nat";

Reading Demo_Nat.casl
Analyzing spec Nat...


Number of encoded axioms: 9
val Nat_1_def = "1 = (suc(0))" : thm
val Nat_2_def = "2 = (suc(1))" : thm
val Nat_add_0 = "(0+ ?m) = ?m" : thm
val Nat_add_suc = "((suc(?n))+ ?m) = (suc((?n+ ?m)))" : thm
val Nat_mult_0 = "(0* ?m) = 0" : thm
val Nat_mult_suc = "((suc(?n))* ?m) = ((?n* ?m)+ ?m)" : thm
val ga_disjoint_0_suc = "not 0 = (suc(?Y0.0))" : thm
val ga_generated_Nat =
  "[| ?PNat 0; !!x1. ?PNat x1 =>?PNat (suc(x1))  |] ==> ?PNat ?xNat" : thm
val ga_injective_suc = "((suc(?X0.0)) = (suc(?Y0.0))) = (?X0.0 = ?Y0.0)" : thm
structure Nat :
  sig
    val axioms : (string * thm) list
    val casl : casltheory
  end
val it = () : unit
HOL-CASL>
```

You get a bunch of ML values. (Notice that Isabelle is implemented in ML, and proving is just executing ML programs. But do not worry - you do not have to learn ML in order to use HOL-CASL.)
Among the ML values are representations of the original axioms, such as `Nat_add_0`, within Isabelle/HOL. Note that the universal quantification of this axiom has

been eliminated, and the variable m has been replaced by a meta variable ?m. This has the advantage that the axiom can be used in proofs much more easily. You also get values like ga_disjoint_0_suc. The ga stands for "generated axiom". The axiom ga_disjoint_0_suc states that 0 is not the successor of anything; it is implicitly generated by CASL's **free type** construct (and so are the other "ga" axioms).
Finally, you get values Nat.casl (the CASL theory Nat) and Nat.axioms (its list of axioms).
You tell HOL-CASL that you want to do proofs in the theory Nat by typing in:

```
CASL_context Nat.casl;
```

It is advisable to add all the axioms of your specification into Isabelle/HOL's simplifier (well, all of some appropriate form). This can be done by

```
 AddsimpAll();
```

Then you can enter a goal:

```
HOL-CASL>  Goal "1+1=2";
Level 0 (1 subgoal)
(1+ 1) = 2
 1. (1+ 1) = 2
val it = [] : thm list
HOL-CASL>
```

N.B.: To save typing, you also can copy and paste the input for HOL-CASL from the file Demo.ML. We did not call it Demo_Nat.ML in order to prevent it from being loaded when loading Demo_Nat.casl. You also can load the demo specifications and proofs by just typing use "Demo". Moreover, a good idea to use Emacs split into two windows: one with the file Demo.ML, and the other one with a shell from which HOL-CASLis called.
The goal is displayed as the one (and only) subgoal that has to be solved. The goal can be solved automatically, using Isabelle's simplifier tactic Simp_tac (the "by" applies a tactic, and the 1 is the number of the subgoal that we want to solve):

```
HOL-CASL>  by (Simp_tac 1);
Level 1
(1+ 1) = 2
No subgoals!
val it = () : unit
HOL-CASL>
```

Finally, you can give a name to the theorem you just have used, and get a value, one_plus_one, for use in further proofs:

```
HOL-CASL> qed "one_plus_one";
val one_plus_one = "(1+ 1) = 2" : thm
val it = () : unit
HOL-CASL>
```

4 Induction

Consider the goal to prove associativity of addition:

```
HOL-CASL>  Goal "forall x,y,z:Nat . (x+y)+z = x+(y+z)";
Level 0 (1 subgoal)
((x+ y)+ z) = (x+ (y+ z))
 1. ((x+ y)+ z) = (x+ (y+ z))
val it = [] : thm list
HOL-CASL>
```

It is clear that we have to use the induction axiom `ga_generated_Nat` (which is the encoding of the sort generation constraint implicit in CASL's **free type** construct, see also section A.3) somehow. Fortunately, there is a tactic `induct_tac` which does the job for us:

```
HOL-CASL> by (induct_tac "x" 1);
Level 1 (2 subgoals)
((x+ y)+ z) = (x+ (y+ z))
 1. ((0+ y)+ z) = (0+ (y+ z))
 2. !!x1. ((x1+ y)+ z) =
          (x1+ (y+ z)) =>(((suc(x1))+ y)+ z) = ((suc(x1))+ (y+ z))
val it = () : unit
HOL-CASL>
```

You now get two subgoals, the first one being the basis of the induction, and the second one the induction step (the `!!` in the second goal denotes universal meta quantification). Both goals can be solved automatically using the simplifier (the `ALLGOALS` here means that the tactic is applied to all of the goals):

```
HOL-CASL>  by (ALLGOALS Simp_tac);
Level 2
((x+ y)+ z) = (x+ (y+ z))
No subgoals!
val it = () : unit
HOL-CASL>
```

That's it! It remains to give a name to the theorem:

```
HOL-CASL> qed "Nat_add_assoc";
val Nat_add_assoc = "((?x+ ?y)+ ?z) = (?x+ (?y+ ?z))" : thm
val it = () : unit
HOL-CASL>
```

5 Loading CASL libraries and tactic scripts

So far for our first small example. The full specification of natural numbers is contained in `Basic/Numbers.casl`. You can load it with

```
use_casl "Basic/Numbers";
```

This will load both the CASL library `Basic/Numbers.casl` and the tactic script `Basic/Numbers.ML`. A tactic script is just a text file with a bunch of user input lines of the form that we have used above. It is advisable to store one's proofs in such scripts in order to be able to later re-load them. By convention, when a CASL library is loaded with `use_casl`, also a try is made to load the tactic script with the same name as the library, apart from that the extension `.casl` is replaced with `.ML`.
You should have a closer look at this tactic script `Basic/Numbers.ML` to see how more serious proofs with HOL-CASL are done. You also can use the other libraries in the folder `Basic`, but there are no or only small tactic scripts in the moment.
The `use_casl` command reloads a CASL library only if it has been changed in the meantime. If you have just changed the ML tactic script and want to reload it, you can enter (for examples, for the Numbers library):

```
use "Basic/Numbers";
```

However, note that the tactic script may fail to go through, since it now does not start with initial versions of simplifier sets, but rather with the versions produces by the former loading of the tactic script. In that case, you should reload the whole library. This can be enforced with

```
update_casl "Basic/Numbers";
```

which will load the library regardless of whether it has been modified or not.

6 Loading and saving sessions

In order to save a session (for example, after having loaded some of the basic datatypes), you type

```
save "<session-name>";
```

In order to leave a session, you type

```
quit();
```

(or just crtl-D).
You can restore a session by typing

```
load <session-name>
```

This just returns to the state where you have been when entering the `save` command. However, notice this works through storing ML heap file, which are quite large (typically 30 to 50 megabytes).

7 The Simplifier

Isabelle's simplifier is a quite powerful rewriting engine. However, when deciding which axioms and theorems to put as rewrite rules into a simplifier set, some care is needed. If one is not careful, the simplifier may loop (with a non-terminating set of rules), or it may do rewriting steps in the wrong direction (with a non-confluent set of rules).
Let us continue with the Demo_Nat example (which just a small part of the library Numbers). Suppose that we want to prove commutativity of addition. We enter:

```
Goal "forall x,y:Nat . x+y=y+x";
by (induct_tac "x" 1);
by (ALLGOALS Simp_tac);
```

We then arrive at:

```
Level 2 (2 subgoals)
(x+ y) = (y+ x)
 1. y = (y+ 0)
 2. !!x1. (x1+ y) = (y+ x1) =>(suc((y+ x1))) = (y+ (suc(x1)))
val it = () : unit
HOL-CASL>
```

For proving the two subgoals, we need induction again. However, since the subgoals may be of interest in their own right, we prove them as lemmas first (omitting the premise of the second subgoal; it already has been used by the simplifier to simplify the conclusion):

```
Goal "forall x:Nat . x+0=x";
by (induct_tac "x" 1);
by (ALLGOALS Simp_tac);
qed "Nat_add_0_right";

Goal "forall x,y:Nat . x+suc(y) = suc(x+y)";
by (induct_tac "x" 1);
by (ALLGOALS Simp_tac);
qed "Nat_add_suc_right";
```

Now we can add the lemmas to the default simplifier set:

```
Addsimps [Nat_add_0_right,Nat_add_suc_right];
```

and prove commutativity of addition, using the enriched simplifier set:

```
Goal "forall x,y:Nat . x+y=y+x";
by (induct_tac "x" 1);
by (ALLGOALS Simp_tac);
qed "Nat_add_comm";
```

However, it is not advisable to throw Nat_add_comm into the simplifier. If we try:

```
Addsimps [Nat_add_comm];

Goal "forall x,y,z:Nat . x+(y+z)=y+(x+z)";
by (induct_tac "x" 1);
by (ALLGOALS Simp_tac);
```

it works well, but in other situations, the simplifier may loop. We therefore should remove the theorem from the simplifier again:

```
Delsimps [Nat_add_comm]; (* Remove it from the default simplifier *)
```

and add it only on-the-fly. Therefore, we use simp_tac instead of Simp_tac, which means that we have to provide an explicit simplifier set. It is constructed from simpset() (the default simplifier set used by Simp_tac) with the infix operation addsimps:

```
Goal "forall x,y,z:Nat . x+(y+z)=y+(x+z)";
by (induct_tac "x" 1);
by (ALLGOALS (simp_tac (simpset() addsimps [Nat_add_comm])));
qed "Nat_add_left_commute";
```

8 The tableaux prover

Besides the simplifier, Isabelle also has a tableaux prover, which can perform transformations of logical connectives and quantifiers. It is used with the tactic Blast_tac:

```
Goal "forall n:Nat . not n = 0 => exists m:Nat.  n = suc(m)";
by (induct_tac "n" 1);
by (Simp_tac 1);
by (Blast_tac 1);
qed "Nat_not0_implies_suc";
```

Note that in this example, neither the simplifier nor the tableaux prover can solve the goal on its own; they have to be combined. This combination can be achieved in one step with Auto_tac:

```
Goal "forall n:Nat . not n = 0 => exists m:Nat.  n = suc(m)";
by (induct_tac "n" 1);
by Auto_tac;
qed "Nat_not0_implies_suc";
```

Note that there also is a variant auto_tac that takes an explicit clasimpset (i.e.simplifier set plus classical reasoning set for the tableaux prover):

```
Goal "forall n:Nat . not n = 0 <=> exists m:Nat.  n = suc(m)";
by (auto_tac (clasimpset() addsimps2 [Nat_not0_implies_suc]));
qed "Nat_not0_equiv_suc";
```

9 Subsorting and partiality

The encoding of CASL into higher-order logic (HOL) is done by encoding partiality via error (or "undefined") elements living in a supersort, encoding subsorting via injections, and encoding sort generation constraints as induction axioms (see [9, 8]). Details of the encoding can be found in the appendix.

Consider the following specification of integers as a supersort of the natural numbers:

library DEMO_INT
%prec $\{\_\_+\_\_, \_\_-\_\_\} < \{\_\_*\_\_\}$
from DEMO_NAT **get** NAT
spec INT =
  NAT
then **generated type** $Int ::= \_\_ - \_\_(Nat;\ Nat)$
  forall $a, b, c, d : Nat$
  • $a - b = c - d <=> a + d = c + b$ %(*Int_equality*)%
then **sort** $Nat < Int$
  forall $a : Nat$
  • $a = a - 0$ %(*Int_Nat_embedding*)%
  ops $0, 1, 2 : Int;$
    $\_\_+\_\_, \_\_-\_\_, \_\_*\_\_ : Int * Int -> Int;$
    $-\_\_ : Int -> Int;$
  forall m,n: Int; a,b,c,d: Nat
  • $(a - b) + (c - d) = (a + c) - (b + d)$ %(*Int_add_def*)%
  • $(a - b) * (c - d) = (a * c + b * d) - (b * c + a * d)$ %(*Int_mult_def*)%
  • $-(a - b) = b - a$ %(*Int_neg_def*)%
  • $m - n = m + (-n)$ %(*Int_sub_def*)%
end

With this specification, we get:

```
HOL-CASL> use_casl"Demo_Int";

Reading Demo_Int.casl
Analyzing from Demo_Nat...
Analyzing spec Int...


Number of encoded axioms: 61
val Int_Nat_embedding = "Nat>Int ?a = (?a- 0)" : thm
val Int_add_def = "((?a- ?b)+ (?c- ?d)) = ((?a+ ?c)- (?b+ ?d))" : thm
val Int_equality = "((?a- ?b) = (?c- ?d)) = ((?a+ ?d) = (?c+ ?b))" : thm
val Int_mult_def =
  "((?a- ?b)* (?c- ?d)) = (((?a* ?c)+ (?b* ?d))- ((?b* ?c)+ (?a* ?d)))" : thm
val Int_neg_def = "(- (?a- ?b)) = (?b- ?a)" : thm
val Int_sub_def = "(?m- ?n) = (?m+ (- ?n))" : thm
val ga_bottom_Int = "not def ?x  = (?x = bottom$)" : thm
```

```
val ga_bottom_Nat = "not def ?x  = (?x = bottom$)" : thm
val ga_definedness_Int = "def Int>?[Int] ?x " : thm
val ga_definedness_Nat = "def Nat>?[Nat] ?x " : thm
val ga_embedding_injectivity_Q_Nat_Q_Int_ =
  "(?[Nat]>?[Int] ?x = ?[Nat]>?[Int] ?y) = (?x = ?y)" : thm
val ga_embedding_injectivity_Int_Q_Int_ =
  "(Int>?[Int] ?x = Int>?[Int] ?y) = (?x = ?y)" : thm
val ga_embedding_injectivity_Nat_Q_Int_ =
  "(Nat>?[Int] ?x = Nat>?[Int] ?y) = (?x = ?y)" : thm
val ga_embedding_injectivity_Nat_Q_Nat_ =
  "(Nat>?[Nat] ?x = Nat>?[Nat] ?y) = (?x = ?y)" : thm
val ga_embedding_injectivity_Nat_Int = "(Nat>Int ?x = Nat>Int ?y) = (?x = ?y)"
  : thm
val ga_existential_equality_Int = "(?x =e= ?y) = (?x = ?y/\\ def ?x )" : thm
val ga_existential_equality_Nat = "(?x =e= ?y) = (?x = ?y/\\ def ?x )" : thm
val ga_generated_Int = "(!!x1 x2. ?P0.0 (x1- x2)) ==> ?P0.0 ?x0.0" : thm
val ga_membership_Int = "def ?x  = (exists y. ?x = Int>?[Int] y)" : thm
val ga_membership_Nat = "def ?x  = (exists y. ?x = Nat>?[Nat] y)" : thm
val ga_membership_Nat_Int = "?x in Nat = (exists y. ?x = Nat>Int y)" : thm
val ga_overload_F_op_Minus_Q_Int_ToQ_Int_op_Minus_IntToInt_ =
  "(- Int>?[Int] ?X1.0) = Int>?[Int] (- ?X1.0)" : thm
val ga_overload_F_op_0_Int_op_0_Nat_ = "0 = Nat>Int 0" : thm
val ga_overload_F_op_1_Int_op_1_Nat_ = "1 = Nat>Int 1" : thm
val ga_overload_F_op_2_Int_op_2_Nat_ = "2 = Nat>Int 2" : thm
val ga_overload_F_op_x_Q_Int_xQ_Int_ToQ_Int_op_x_Q_Nat_xQ_Nat_ToQ_Nat_ =
  "(?[Nat]>?[Int] ?X2.0* ?[Nat]>?[Int] ?X3.0) = ?[Nat]>?[Int] (?X2.0* ?X3.0)"
  : thm
val ga_overload_F_op_x_Q_Int_xQ_Int_ToQ_Int_op_x_IntxIntToInt_ =
  "(Int>?[Int] ?X2.0* Int>?[Int] ?X3.0) = Int>?[Int] (?X2.0* ?X3.0)" : thm
val ga_overload_F_op_x_Q_Int_xQ_Int_ToQ_Int_op_x_NatxNatToNat_ =
  "(Nat>?[Int] ?X2.0* Nat>?[Int] ?X3.0) = Nat>?[Int] (?X2.0* ?X3.0)" : thm
val ga_overload_F_op_x_Q_Nat_xQ_Nat_ToQ_Nat_op_x_IntxIntToInt_ =
  "?[Nat]>?[Int] (Nat>?[Nat] ?X2.0* Nat>?[Nat] ?X3.0) =
   Int>?[Int] (Nat>Int ?X2.0* Nat>Int ?X3.0)" : thm
val ga_overload_F_op_x_Q_Nat_xQ_Nat_ToQ_Nat_op_x_NatxNatToNat_ =
  "(Nat>?[Nat] ?X2.0* Nat>?[Nat] ?X3.0) = Nat>?[Nat] (?X2.0* ?X3.0)" : thm
val ga_overload_F_op_x_IntxIntToInt_op_x_NatxNatToNat_ =
  "(Nat>Int ?X2.0* Nat>Int ?X3.0) = Nat>Int (?X2.0* ?X3.0)" : thm
val ga_overload_F_op_Plus_Q_Int_xQ_Int_ToQ_Int_op_Plus_Q_Nat_xQ_Nat_ToQ_Nat_ =
  "(?[Nat]>?[Int] ?X2.0+ ?[Nat]>?[Int] ?X3.0) = ?[Nat]>?[Int] (?X2.0+ ?X3.0)"
  : thm
val ga_overload_F_op_Plus_Q_Int_xQ_Int_ToQ_Int_op_Plus_IntxIntToInt_ =
  "(Int>?[Int] ?X2.0+ Int>?[Int] ?X3.0) = Int>?[Int] (?X2.0+ ?X3.0)" : thm
val ga_overload_F_op_Plus_Q_Int_xQ_Int_ToQ_Int_op_Plus_NatxNatToNat_ =
  "(Nat>?[Int] ?X2.0+ Nat>?[Int] ?X3.0) = Nat>?[Int] (?X2.0+ ?X3.0)" : thm
val ga_overload_F_op_Plus_Q_Nat_xQ_Nat_ToQ_Nat_op_Plus_IntxIntToInt_ =
```

```
  "?[Nat]>?[Int] (Nat>?[Nat] ?X2.0+ Nat>?[Nat] ?X3.0) =
   Int>?[Int] (Nat>Int ?X2.0+ Nat>Int ?X3.0)" : thm
val ga_overload_F_op_Plus_Q_Nat_xQ_Nat_ToQ_Nat_op_Plus_NatxNatToNat_ =
  "(Nat>?[Nat] ?X2.0+ Nat>?[Nat] ?X3.0) = Nat>?[Nat] (?X2.0+ ?X3.0)" : thm
val ga_overload_F_op_Plus_IntxIntToInt_op_Plus_NatxNatToNat_ =
  "(Nat>Int ?X2.0+ Nat>Int ?X3.0) = Nat>Int (?X2.0+ ?X3.0)" : thm
val ga_overload_F_op_Minus_Q_Int_xQ_Int_ToQ_Int_op_Minus_Q_Nat_xQ_Nat_ToQ_Int_ =
  "(?[Nat]>?[Int] ?X2.0- ?[Nat]>?[Int] ?X3.0) = (?X2.0- ?X3.0)" : thm
val ga_overload_F_op_Minus_Q_Int_xQ_Int_ToQ_Int_op_Minus_IntxIntToInt_ =
  "(Int>?[Int] ?X2.0- Int>?[Int] ?X3.0) = Int>?[Int] (?X2.0- ?X3.0)" : thm
val ga_overload_F_op_Minus_Q_Int_xQ_Int_ToQ_Int_op_Minus_NatxNatToInt_ =
  "(Nat>?[Int] ?X2.0- Nat>?[Int] ?X3.0) = Int>?[Int] (?X2.0- ?X3.0)" : thm
val ga_overload_F_op_Minus_Q_Nat_xQ_Nat_ToQ_Int_op_Minus_IntxIntToInt_ =
  "(Nat>?[Nat] ?X2.0- Nat>?[Nat] ?X3.0) =
   Int>?[Int] (Nat>Int ?X2.0- Nat>Int ?X3.0)" : thm
val ga_overload_F_op_Minus_Q_Nat_xQ_Nat_ToQ_Int_op_Minus_NatxNatToInt_ =
  "(Nat>?[Nat] ?X2.0- Nat>?[Nat] ?X3.0) = Int>?[Int] (?X2.0- ?X3.0)" : thm
val ga_overload_F_op_Minus_IntxIntToInt_op_Minus_NatxNatToInt_ =
  "(Nat>Int ?X2.0- Nat>Int ?X3.0) = (?X2.0- ?X3.0)" : thm
val ga_overload_F_op_bottom_Q_Int_op_bottom_Q_Nat_ =
  "bottom$ = ?[Nat]>?[Int] bottom$" : thm
val ga_overload_F_op_proj_Q_Int_ToQ_Nat_op_proj_IntToQ_Nat_ =
  "Int>?[Int] ?X1.0 as [Nat] = ?X1.0 as [Nat]" : thm
val ga_overload_F_op_suc_Q_Nat_ToQ_Nat_op_suc_NatToNat_ =
  "(suc(Nat>?[Nat] ?X1.0)) = Nat>?[Nat] (suc(?X1.0))" : thm
val ga_overload_P_pred_def_Q_Int_pred_def_Q_Nat_ =
  "def ?[Nat]>?[Int] ?X1.0  = def ?X1.0 " : thm
val ga_overload_P_pred_eeq_Q_Int_xQ_Int_pred_eeq_Q_Nat_xQ_Nat_ =
  "(?[Nat]>?[Int] ?X2.0 =e= ?[Nat]>?[Int] ?X3.0) = (?X2.0 =e= ?X3.0)" : thm
val ga_projection_Int_Nat = "Nat>Int ?x as [Nat] = Nat>?[Nat] ?x" : thm
val ga_projection_injectivity_Int_Nat =
  "(?x as [Nat] = ?y as [Nat]) = (?x = ?y)" : thm
val ga_projection_membership_Nat_Int = "?x in Nat = def ?x as [Nat] " : thm
val ga_strictness_and_totality_op_Minus_IntToInt_ = "def - ?X0.0  = def ?X0.0 "
  : thm
val ga_strictness_and_totality_op_x_IntxIntToInt_ =
  "def ?X0.0* ?X1.0  = (def ?X0.0 /\\ def ?X1.0 )" : thm
val ga_strictness_and_totality_op_x_NatxNatToNat_ =
  "def ?X0.0* ?X1.0  = (def ?X0.0 /\\ def ?X1.0 )" : thm
val ga_strictness_and_totality_op_Plus_IntxIntToInt_ =
  "def ?X0.0+ ?X1.0  = (def ?X0.0 /\\ def ?X1.0 )" : thm
val ga_strictness_and_totality_op_Plus_NatxNatToNat_ =
  "def ?X0.0+ ?X1.0  = (def ?X0.0 /\\ def ?X1.0 )" : thm
val ga_strictness_and_totality_op_Minus_IntxIntToInt_ =
  "def ?X0.0- ?X1.0  = (def ?X0.0 /\\ def ?X1.0 )" : thm
val ga_strictness_and_totality_op_Minus_NatxNatToInt_ =
```

```
  "def ?X0.0- ?X1.0  = (def ?X0.0 /\\ def ?X1.0 )" : thm
val ga_strictness_and_totality_op_suc_NatToNat_ =
  "def suc(?X0.0)  = def ?X0.0 " : thm
val ga_transitivity_Nat_Q_Nat_Q_Int_ =
  "?[Nat]>?[Int] (Nat>?[Nat] ?x) = Nat>?[Int] ?x" : thm
val ga_transitivity_Nat_Int_Q_Int_ = "Int>?[Int] (Nat>Int ?x) = Nat>?[Int] ?x"
  : thm
structure Int :
  sig
    val axioms : (string * thm) list
    val casl : casltheory
  end
val it = () : unit
HOL-CASL>
```

We can see that two new sorts, `?[Nat]` and `?[Int]` are generated, and also quite a lot of axioms. (Note that there are even more axioms, e.g. overloading axioms, that are not provided as ML values, however. The reason is that characters like "?" are not allowed in ML identifiers; hence e.g. axioms involving ?-supersorts cannot be named in ML.) The details of this encoding can be found in the appendix.

Since partiality is encoded via supersorts with error (or "undefined") elements, any use of partiality implies the presence of subsorts in the encoding. However, also the converse holds: any use of subsorts implies the presence of partiality and error supersorts (even in specifications like `Int` above, where there is no partiality present). This is because subsorting comes with partial projections. In case where one does not need partial projections, one can greatly simplify the encoding by omitting them. This can be done with

```
reset use_projections;
```

(while the opposite effect is achieved with a `set use_projections`).

Of course, this also disables the possibility of using casts in the input specifications. For our above example, the list of generated axioms becomes considerably shorter:

```
HOL-CASL> update_casl "Demo_Int";
HOL-CASL>  use_casl"Demo_Int";

Reading Demo_Int.casl
Analyzing from Demo_Nat...
Analyzing spec Int...


Number of encoded axioms: 15
val Int_Nat_embedding = "Nat>Int ?a = (?a- 0)" : thm
val Int_add_def = "((?a- ?b)+ (?c- ?d)) = ((?a+ ?c)- (?b+ ?d))" : thm
```

```
val Int_equality = "((?a- ?b) = (?c- ?d)) = ((?a+ ?d) = (?c+ ?b))" : thm
val Int_mult_def =
  "((?a- ?b)* (?c- ?d)) = (((?a* ?c)+ (?b* ?d))- ((?b* ?c)+ (?a* ?d)))" : thm
val Int_neg_def = "(- (?a- ?b)) = (?b- ?a)" : thm
val Int_sub_def = "(?m- ?n) = (?m+ (- ?n))" : thm
val ga_embedding_injectivity_Nat_Int = "(Nat>Int ?x = Nat>Int ?y) = (?x = ?y)"
  : thm
val ga_generated_Int = "(!!x1 x2. ?P0.0 (x1- x2)) ==> ?P0.0 ?x0.0" : thm
val ga_membership_Nat_Int = "?x in Nat = (exists y. ?x = Nat>Int y)" : thm
val ga_overload_F_op_0_Int_op_0_Nat_ = "0 = Nat>Int 0" : thm
val ga_overload_F_op_1_Int_op_1_Nat_ = "1 = Nat>Int 1" : thm
val ga_overload_F_op_2_Int_op_2_Nat_ = "2 = Nat>Int 2" : thm
val ga_overload_F_op_x_IntxIntToInt_op_x_NatxNatToNat_ =
  "(Nat>Int ?X2.0* Nat>Int ?X3.0) = Nat>Int (?X2.0* ?X3.0)" : thm
val ga_overload_F_op_Plus_IntxIntToInt_op_Plus_NatxNatToNat_ =
  "(Nat>Int ?X2.0+ Nat>Int ?X3.0) = Nat>Int (?X2.0+ ?X3.0)" : thm
val ga_overload_F_op_Minus_IntxIntToInt_op_Minus_NatxNatToInt_ =
  "(Nat>Int ?X2.0- Nat>Int ?X3.0) = (?X2.0- ?X3.0)" : thm
structure Int :
  sig
    val axioms : (string * thm) list
    val casl : casltheory
  end
val it = () : unit
HOL-CASL>
```

The effect is even more dramatic than shown here — the overloading axioms are not shown since they contain special characters not valid in ML identifiers. Altogether, we now have 15 axioms instead of 61 before.
Now we come to the display syntax of terms with subsorts. Usually, injections between subsorts are displayed in subgoals, since they are relevant in proofs. In case that you do not want so see them, enter:

```
print_mode := [""];
```

You can restore the display of injections with:

```
print_mode := ["Inj"];
```

If you also would like to see the types of functions and predicates, enter

```
print_mode := ["InjTypes"];
```

Some of the generated axioms are automatically put into the default simplifier set. However, this does not hold for the overloading axioms. The reason is that it is not clear in advance in which direction the overloading axioms should be applied. Therefore, two functions are provided that enrich the current default simplifier set with the overloading axioms. While `overload_down` simplifies towards profiles living in subsorts, `overload_up` simplifies towards profiles living in supersorts. For example:

```
Goal "forall x:Int . x-x=0";
by (induct_tac "x" 1);
(* We need the overloading equation 0:Int=0:Nat *)
by (simp_tac (overload_down() addsimps [Nat_add_comm]) 1);
qed "Int_minus_0";
```

However, even these simplifier sets are only a very crude starting point. A good treatment of subsorting within simplifier sets is still a subject of current research.

10 Further tactics

For further tactics and tacticals (methods to combine tactics), see the Isabelle manuals. We have adapated the Isabelle tactics to HOL-CASL whenever necessary. The tactic `case_tac` has been split into two tactics: While `case_tac` can be used with some predicate (i.e. `case_tac "n=0"`) to split into two cases, `data_case_tac "n"` assumes that `n` is a variable of a generated or free type, and splits into one subcase for each constructor.
The instantiation tactics, such as `res_inst_tac`, `instantiate_tac` etc., take terms in CASL syntax. However, it is also possible to prefix CASL terms with Isabelle syntax for lambda abstraction here, e.g. as in

```
instantiate_tac [("x","% a b c d . suc(d)")]
```

11 Structured specifications

Isabelle does support extension and union, but neither translation nor hiding nor free specifications. Therefore, we have to eliminate the latter, while keeping the structure given by extensions and unions. This means that theorems are transported along extensions and unions, but not along translations, (some) hidings and free.
Renaming is eliminated by first flattening the specification, and then creating a renamed copy. This means that proofs that you have done for subparts of the renamed specification are lost and have to be redone.
Hiding currently is not properly eliminated — it is just ignored. This works quite well in most cases. In the future, some renaming will be done in order to prevent name clashes between symbols with the same name hidden at different places (however, the renaming will prevent re-use of proofs, see above).
There is some limited support of **free** { *SP* } specifications. *SP* is flattened, and in case it is in positive Horn form, an appropriate sort generation constraint (stating that there is "no junk") is produced. However, an axiom stating that

there is “no confusion” and predicates and domains of functions are minimal is not yet produced (but this will come soon).
There is a further problem: union works differently in CASL and Isabelle — CASL's “same name – same thing” principle is not available in Isabelle. This can lead to problems when same names come from different origins in a union. In the basic datatypes, the problematic uses of unions have been commented out. Also, in some parts that have not been commented out, particularly in the algebra libraries, the different concepts of union cause problems in coding of the axioms, which are not correct in all cases, because Isabelle cannot assign the right type to some constants and functions, and treats them as variables instead (you can notice this by the $ signs in the name). We are looking for a solution for this problem.
Parts of a specification that are marked with an `%implies` annotation are just ignored, because they should not be used in proofs unless they have themselves been proved. Views are also ignored. Of course, it would be desirable to have a management of open proof obligations.
All these deficiencies are overcome with the integration of HOL-CASL into the MAYA system, see [1].

A The encoding of the CASL logic into higher-order logic

We now describe how the CASL logic, namely subsorted partial first-order logic with sort generation constraints ($SubPCFOL^=$), can be encoded into second-order logic ($SOL^=$), which is naturally a sublogic of higher-order logic. The encoding is done in three steps:

1. firstly, partiality is coded out via error elements living in a supersort, giving an encoding from $SubPCFOL^=$ into $SubCFOL^=$;
2. secondly, subsorting is coded out via injections, giving an encoding from $SubCFOL^=$ into $CFOL^=$;
3. and thirdly, sort generation constraints are coded as induction axioms, giving an encoding from $CFOL^=$ into $SOL^=$.

These encodings are not only translations of syntax, but also have a model-theoretic counterpart[2], which provides an implicit soundness and completeness proof for the re-use of HOL-theorem provers for theorem proving in the CASL logic. This is also known as the “borrowing” technique of Cerioli and Meseguer [3], which allows to borrow theorem provers across different logics. Nevertheless, we here only describe the syntax part, i.e. how signatures and sentences are translated. The translation of models and the proofs that these are actually institution representations which admit a faithful translation of logical

[2]Formally, they are institution representations in the sense of [7, 11]

consequence, as well as the definitions of $SubPCFOL^=$, $SubCFOL^=$, $CFOL^=$, $SOL^=$, can be found in [8].

A.1 Encoding $SubPCFOL^=$ into $SubCFOL^=$

The idea here is to add a new element $\perp$ to each sort that represents "undefined", but to retain the old sorts without the undefined elements. That is, the sort set is doubled: each sort s now gets a companion supersort $s^?$, which contains all values of s plus $\perp$. Since we keep the old sorts, the definedness predicate is just membership in the old sorts. Moreover, we add the companion supersort only to those sorts for which it is necessary, i.e. for those where a term involving partial function symbols exists.
In the signature translation, we introduce two versions for each function and predicate symbol: the original one and its (overloaded) strict extension that just propagates the undefined element. The strict extension is necessary if we want to apply a function or predicate symbol to a term involving partial function symbols, since this term can be translated only to a term of sort $s^?$.
Since all partial functions with result sort s are totalized to functions with result sort $s^?$, also the partial projections have to be totalized. For them, we introduce new special function symbols `proj`.
We deviate from the translation (5a') in [8] in the following points: firstly, we also axiomatize existential equality as a predicate; this makes the sentence translation easier. Secondly, since we axiomatize a *unique* undefined element, some axioms are simplified (actually this is not a deviation, but the simplified axioms are not shown in [8]). Moreover, note that, in order to be conform with the CASL syntax, the HOL-CASL system displays the membership predicates $\in s$ as def, since these predicates play the role of definedness predicates.

Translation of signatures Let $\Sigma = (S, TF, PF, P, \leq_S)$ be a $SubPFOL^=$-signature. Let

- $S^? = \{s \in S \mid$ there exists a variable system X and a $\Sigma(X)$-term of sort s built using at least one partial function symbol or projection cast$\}$,
- $S' = S \cup \{s^? \mid s \in S^?\}^3$ and $Q(s) = \begin{cases} s^?, & \text{if } s \in S^? \\ s, & \text{otherwise} \end{cases}$

Then Σ is translated to the $SubFOL^=$-signature $\mathbf{Sig}(\Phi(\Sigma))$ consisting of the sort set S', ordered by

- $s \leq' s'$ iff $s \leq s'$ $(s, s' \in S)$;
- $s \leq' s'^?$ iff $s \leq s'$ $(s, s' \in S)$;
- $s^? \not\leq' s'$ $(s, s' \in S)$, and
- $s^? \leq' s'^?$ iff $s \leq s'$, $(s, s' \in S)$;

augmented by the following total function symbols:

$\perp : s^?$	for $s \in S^?$
$f: s_1, \ldots, s_n \to s$	for $f: s_1, \ldots, s_n \to s \in TF$
$f: s_1, \ldots, s_n \to (s^?)$	for $f: s_1, \ldots, s_n \to ? s \in PF$,
$f: Q(s_1), \ldots, Q(s_n) \to (s^?)$	for $f: s_1, \ldots, s_n \longrightarrow s \in TF \cup PF$, $\{s_1, \ldots, s_n\} \cap S^? \neq \emptyset$
$\mathtt{proj}: s \to ({s'}^?)$	for $s' \leq s$
$\mathtt{proj}: s^? \to ({s'}^?)$	for $s' \leq s$

the following predicate symbols:

$p: s_1, \ldots, s_n$	for $p: s_1, \ldots, s_n \in P$,
$p: Q(s_1), \ldots, Q(s_n)$	for $p: s_1, \ldots, s_n \in P, \{s_1, \ldots, s_n\} \cap S^? \neq \emptyset$
$\stackrel{e}{=}_s : s^? \times s^?$	for $s \in S^?$,

and the following axioms $\mathbf{Ax}(\Phi(\Sigma))$:

$\neg\, x \in s \Leftrightarrow x = \perp$ for $s \in S^?$ (bottom)

$\forall x : s' \bullet \mathtt{proj}_{(s',s^?)}(x) \in s \Leftrightarrow x \in s$ for $s \leq s'$ (projection-membership)

$\forall x : s \bullet \mathtt{proj}_{(s',s^?)}(\mathtt{inj}_{(s,s')}(x)) = \mathtt{inj}_{(s,s^?)}(x)$ for $s \leq s'$ (projection)

$\forall x, y : s' \bullet \mathtt{proj}_{(s',s^?)}(x) = \mathtt{proj}_{(s',s^?)}(y) \Rightarrow x = y$
for $s \leq s'$ (projection-injectivity)

$\forall x, y : s^? \bullet x \stackrel{e}{=}_s y \Leftrightarrow x = y \wedge x \in s$ for $s \in S^?$ (existential-equality)

$\forall x_1 : Q(s_1), \ldots, x_n : Q(s_n) \bullet\ f(x_1, \ldots, x_n) \in s \Leftrightarrow \bigwedge_{s_i \in S^?} x_i \in s_i$
for $f: s_1, \ldots, s_n \to s \in TF$ (strictness-and-totality)

$\forall x_1 : Q(s_1), \ldots, x_n : Q(s_n) \bullet\ f(x_1, \ldots, x_n) \in s \Rightarrow \bigwedge_{s_i \in S^?} x_i \in s_i$
for $f: s_1, \ldots, s_n \to ? s \in PF$ (strictness)

$\forall x_1 : Q(s_1), \ldots, x_n : Q(s_n) \bullet\ p(x_1, \ldots, x_n) \Rightarrow \bigwedge_{s_i \in S^?} x_i \in s_i$
for $p: s_1, \ldots, s_n \in P$ (predicate-strictness)

The axioms (bottom) state that $\perp$ is the unique "undefined" element. The membership, projection and projection-injectivity axioms are those of the axioms used in the definition of $SubPCFOL^=$ (see [6, 2, 8]) that refer to the partial projections. The remaining axioms state strictness of the functions and predicates, while (strictness-and-totality) additionally expresses totality of the total functions.

Translation of sentences A Σ-sentence in $SubPFOL^=$ is translated to a $\Phi(\Sigma)$-sentence in $SubFOL^=$ inductively as follows:

- $\alpha_\Sigma(x : s) = x : s$

[3]We assume that S does not contain sorts of the form $s^?$. In a higher-order extension of CASL, there could be a type constructor "?" behaving like our "?".

- $\alpha_\Sigma(f_{(s_1,\ldots,s_n,s)}(t_1,\ldots,t_n)) = \qquad (f\colon s_1,\ldots,s_n \to s \in TF)$
$\begin{cases} f_{\langle s_1,\ldots,s_n,s)}(\alpha_\Sigma(t_1),\ldots,\alpha_\Sigma(t_n)\rangle \text{ if for } i=1\ldots n,\ \alpha_\Sigma(t_i) \text{ is of sort } s_i \\ f_{(Q(s_1),\ldots,Q(s_n),(s^?))}(\alpha_\Sigma(t_1),\ldots,\alpha_\Sigma(t_n)),\ \text{otherwise} \end{cases}$
- $\alpha_\Sigma(f_{(s_1,\ldots,s_n,s)}(t_1,\ldots,t_n)) = \qquad (f\colon s_1,\ldots,s_n \to ?\ s \in PF)$
$\begin{cases} f_{(s_1,\ldots,s_n,(s^?))}(\alpha_\Sigma(t_1),\ldots,\alpha_\Sigma(t_n)) \text{ if for } i=1\ldots n,\ \alpha_\Sigma(t_i) \text{ is of sort } s_i \\ f_{(Q(s_1),\ldots,Q(s_n),(s^?))}(\alpha_\Sigma(t_1),\ldots,\alpha_\Sigma(t_n)),\ \text{otherwise} \end{cases}$
- $\alpha_\Sigma(p_{s_1,\ldots,s_n}(t_1,\ldots,t_n)) = \qquad (p : s_1,\ldots,s_n \in P)$
$\begin{cases} p_{s_1,\ldots,s_n}(\alpha_\Sigma(t_1),\ldots,\alpha_\Sigma(t_n)) \text{ if for } i=1\ldots n,\ \alpha_\Sigma(t_i) \text{ is of sort } s_i \\ p_{Q(s_1),\ldots,Q(s_n)}(\alpha_\Sigma(t_1),\ldots,\alpha_\Sigma(t_n)),\ \text{otherwise} \end{cases}$
- $\alpha_\Sigma(\mathtt{pr}_{(s_1,s_2)}(t)) = \begin{cases} \mathtt{proj}_{(s_1^?,s_2^?)}(\alpha_\Sigma(t)), & \text{if } \alpha_\Sigma(t) \text{ is of sort } s_1^? \\ \mathtt{proj}_{(s_1,s_2^?)}(\alpha_\Sigma(t)), & \text{otherwise} \end{cases} \quad (s_2 \leq s_1)$
- $\alpha_\Sigma(def(t)) = \begin{cases} \alpha_\Sigma(t) \in s, & \text{if } \alpha_\Sigma(t) \text{ is of sort } s^?, s \in\ S^? \\ T, & \text{otherwise} \end{cases}$
- $\alpha_\Sigma(t \in s_1) = \begin{cases} t \in s_1^?, & \text{if } \alpha_\Sigma(t) \text{ is of sort } s_2^? \\ t \in s_1, & \text{otherwise} \end{cases}$
- $\alpha_\Sigma(t_1 = t_2) = \alpha_\Sigma(t_1) = \alpha_\Sigma(t_2)$
- $\alpha_\Sigma(t_1 \stackrel{\mathrm{e}}{=} t_2) = \begin{cases} \alpha_\Sigma(t_1) = \mathrm{e} =_s \alpha_\Sigma(t_2), & \text{if } \alpha_\Sigma(t_1), \alpha_\Sigma(t_2) \text{ are of sort } s^? \\ \alpha_\Sigma(t_1) = \alpha_\Sigma(t_2), & \text{otherwise} \end{cases}$

 Note that in the last two cases, if the translation of only one term is of a "question mark" sort, the translation of the other one has to be made to be of "question mark" sort as well (by inserting an appropriate injection). The translation can easily be extended from atomic to all formulas.

Apart from mapping definedness and projection to the special symbols introduced above, the translation mainly checks whether the original version of a function or predicate symbol can be used, or if we have to use its strict extension that propagates the undefined element. The strict extension is necessary whenever an argument term involves a partial function symbol.

A Σ-sort generation constraint $(\dot{S}, \dot{F}, \theta\colon \bar{\Sigma} \longrightarrow \Sigma)$ is translated to $(\dot{S}, \dot{F}' \cup \{\bot_s : s | s \in S\}, \theta')$, where $\dot{F}'$ is $\dot{F}$ with all function symbols of form $\mathtt{pr}_{(s,s')}$ replaced by $\mathtt{proj}_{(s,s')}$ and $\theta'\colon \Phi(\bar{\Sigma}) \longrightarrow \Phi(\Sigma)$ is the extension of θ to $\Phi(\bar{\Sigma})$ that is defined by mapping the added symbols in $\Phi(\bar{\Sigma})$ to the corresponding added symbols in $\Phi(\Sigma)$.

A.2 Encoding $SubCFOL^=$ into $CFOL^=$

Translation of signatures Following the CASL semantics, with a subsorted signature $\Sigma = (S, TF, PF, P, \leq_S)$ is translated to many-sorted signature $\hat{\Sigma}$, which is the extension of the underlying many-sorted signature (S, TF, PF, P) with

- a total *injection* function symbol $\mathtt{inj} : s \to s'$, for each pair of sorts $s \leq_S s'$,
- a partial *projection* function symbol $\mathtt{pr} : s' \to ?\ s$, for each pair of sorts $s \leq_S s'$, and
- a unary *membership* predicate symbol $\in^s : s'$, for each pair of sorts $s \leq_S s'$.

We assume that the symbols used for injection, projection and membership are not used otherwise in Σ. In formulas, we also write $t \in s$ instead of $\in^s_{s'}(t)$ if s' is clear from the context.

The translations also adds the following axioms (where the variables are all universally quantified):

$\mathtt{inj}_{(s,s)}(x) \stackrel{e}{=} x$ (identity)

$\mathtt{inj}_{(s,s')}(x) \stackrel{e}{=} \mathtt{inj}_{(s,s')}(y) \Rightarrow x \stackrel{e}{=} y$ for $s \leq_S s'$ (embedding-injectivity)

$\mathtt{inj}_{(s',s'')}(\mathtt{inj}_{(s,s')}(x)) \stackrel{e}{=} \mathtt{inj}_{(s,s'')}(x)$ for $s \leq_S s' \leq_S s''$ (transitivity)

$\in^s_{s'}(x) \Leftrightarrow \exists y \bullet \mathtt{inj}_{(s,s')}(y) = x$ for $s \leq_S s'$ (membership)

$$\mathtt{inj}_{(s',s)}(f_{w',s'}(\mathtt{inj}_{(s_1,s'_1)}(x_1), \ldots, \mathtt{inj}_{(s_n,s'_n)}(x_n))) = \\ = \mathtt{inj}_{(s'',s)}(f_{w'',s''}(\mathtt{inj}_{(s_1,s''_1)}(x_1), \ldots, \mathtt{inj}_{(s_n,s''_n)}(x_n)))$$

for $f_{w',s'} \sim_F f_{w'',s''}$, where $w \leq w', w''$, $w = s_1 \ldots s_n$, $w' = s'_1 \ldots s'_n$, $w'' = s''_1 \ldots s''_n$ and $s', s'' \leq s$ (function-monotonicity)

$p_{w'}(\mathtt{inj}_{(s_1,s'_1)}(x_1), .., \mathtt{inj}_{(s_n,s'_n)}(x_n)) \Leftrightarrow p_{w''}(\mathtt{inj}_{(s_1,s''_1)}(x_1), .., \mathtt{inj}_{(s_n,s''_n)}(x_n))$
for $p_{w'} \sim_P p_{w''}$, where $w \leq w', w''$, $w = s_1 \ldots s_n$, $w' = s'_1 \ldots s'_n$, and $w'' = s''_1 \ldots s''_n$ (predicate-monotonicity)

Note that the axioms for partial projections that are there in the CASL semantics have been omitted here; they are present already in the encoding of $SubPCFOL^=$ to $SubCFOL^=$. Furthermore, the membership predicates cannot be axiomatized using definedness of partial projections, since the latter are total now. Instead, we have to use an existential quantifier that provides an element of the subsort.

Further note that within HOL-CASL, membership of values of sort $s^?$ in sort s is displayed as def. This applies to the membership axioms; part of the membership predicates there are displayed as def.

Finally, within HOL-CASL, we have just refrained from transitively closing the subsort relation also for sorts of form $s^?$; this means that there are no injection symbols from s to $t^?$ for $s \leq t$ (and no corresponding membership symbols either). This works since sentences from $SubPCFOL^=$ can never lead to sentences in $CFOL^=$ containing such symbols. That is, we here exploit special knowledge about the translation from $SubCFOL^=$ to $CFOL^=$ being applied in a specific context, namely in composition with the translation from $SubPCFOL^=$ to $SubCFOL^=$.

Translation of sentences By definition of the institution $SubCFOL^=$, Σ-sentences in $SubCFOL^=$ are already $\hat{\Sigma}$-sentences in $CFOL^=$.

A.3 Encoding $CFOL^=$ into $SOL^=$

$CFOL^=$ adds sort generation constraints to $FOL^=$. These cannot be expressed within $FOL^=$, but can be translated to induction schemes, which can be expressed in second-order logic with equality ($SOL^=$) by second-order quantification over predicates.

Translation of signatures Signatures are translated identically.

Translation of sentences First-order sentences are translated identically. For a sort generation constraint

$$(\overset{\bullet}{S}, \overset{\bullet}{F}, \theta : \bar{\Sigma} \longrightarrow \Sigma)$$

we assume without loss of generality that all the result sorts of function symbols in $\overset{\bullet}{F}$ occur in $\overset{\bullet}{S}$[4]. Let

$$\overset{\bullet}{S} = \{s_1;\ \ldots;\ s_n\}, \quad \overset{\bullet}{F} = \{f_1 : s_1^1 \ldots s_{m_1}^1 \longrightarrow s^1;\ \ldots;\ f_1 : s_1^k \ldots s_{m_k}^k \longrightarrow s^k\}$$

The sort generation constraint is now translated to the $SOL^=$-sentence

$$\begin{aligned} &\forall P_{s_1} : pred(\theta(s_1)) \ldots \forall P_{s_n} : pred(\theta(s_n)) \\ &\qquad \bullet (\varphi_1 \wedge \cdots \wedge \varphi_k) \Rightarrow \textstyle\bigwedge_{j=1,\ldots,n} \forall x : \theta(s_j) \bullet P_{s_j}(x) \end{aligned}$$

where

$$\begin{aligned} \varphi_j &= \forall x_1 : \theta(s_1^j), \ldots, x_{m_j} : \theta(s_{m_j}^j) \\ &\qquad \bullet \left(\textstyle\bigwedge_{i=1,\ldots,m_j;\ s_i^j \in \overset{\bullet}{S}} P_{s_i^j}(x_i) \right) \Rightarrow P_{s^j}\left(\theta(f_j)(x_1, \ldots, x_{m_j})\right) \end{aligned}$$

---

[4]For a sort generation constraint $(\overset{\bullet}{S}, \overset{\bullet}{F}, \theta)$ we can assume without loss of generality that all the result sorts of function symbols in $\overset{\bullet}{F}$ occur in $\overset{\bullet}{S}$. If not, we can just leave out from $\overset{\bullet}{F}$ those function symbols not satisfying this requirement. The satisfaction of the sort generation constraint in any model will not be affected by this: in the $\bar{\Sigma}$-term t witnessing the satisfaction of the constraint, any application of a function symbol with result sort outside $\overset{\bullet}{S}$ can just be replaced by a variable of that sort, which gets then as assigned value the evaluation of the function application.

References

[1] S. Autexier and T. Mossakowski. Integrating HOL-CASL into the development graph manager MAYA. Submitted.

[2] M. Cerioli, A. Haxthausen, B. Krieg-Brückner, and T. Mossakowski. Permissive subsorted partial logic in CASL. In M. Johnson, editor, *Algebraic methodology and software technology: 6th international conference, AMAST 97*, volume 1349 of *Lecture Notes in Computer Science*, pages 91–107. Springer-Verlag, 1997.

[3] M. Cerioli and J. Meseguer. May I borrow your logic? (transporting logical structures along maps). *Theoretical Computer Science*, 173:311–347, 1997.

[4] CoFI. The Common Framework Initiative for algebraic specification and development, electronic archives. Notes and Documents accessible from `http://www.cofi.info`.

[5] CoFI Language Design Task Group. CASL – The CoFI Algebraic Specification Language – Summary. Documents/CASL/Summary, in [4], Mar. 2001.

[6] CoFI Semantics Task Group. CASL – The CoFI Algebraic Specification Language – Semantics. Note S-9 (Documents/CASL/Semantics, version 0.96), in [4], July 1999.

[7] J. Meseguer. General logics. In *Logic Colloquium 87*, pages 275–329. North Holland, 1989.

[8] T. Mossakowski. Relating CASL with other specification languages: the institution level. *Theoretical Computer Science*. To appear.

[9] T. Mossakowski, Kolyang, and B. Krieg-Brückner. Static semantic analysis and theorem proving for CASL. In F. Parisi Presicce, editor, *Recent trends in algebraic development techniques. Proc. 12th International Workshop*, volume 1376 of *Lecture Notes in Computer Science*, pages 333–348. Springer, 1998.

[10] L. C. Paulson. *Isabelle - A Generic Theorem Prover.* Number 828 in LNCS. Springer Verlag, 1994.

[11] A. Tarlecki. Moving between logical systems. In M. Haveraaen, O. Owe, and O.-J. Dahl, editors, *Recent Trends in Data Type Specifications. 11th Workshop on Specification of Abstract Data Types*, volume 1130 of *Lecture Notes in Computer Science*, pages 478–502. Springer Verlag, 1996.