HasCASL: a logic combining higher-order logic, type classes, polymorphism, subsorting and partial functions

Till Mossakowski

Oberseminar 26.04.2022

Motivation

- wish for integration of theorem provers for *induction axioms* into Hets
- Daniel Wand's prover *pirate* supports induction
  - Underlying logic:
    Polymorphic first-order logic with type constructors and type classes
  - HasCASL is a logic that supports such features (and more)
- Original motivation for HasCASL:
  - general-purpose higher-order extension of CASL
  - specification and development of Haskell programs
  - specification and functional programming within a single language

CASL and HasCASL

- CASL is a conservative extension of first-order logic:
  - partial functions
  - subtypes
  - induction axioms for datatypes
- HasCasl is a conservative extension of CASL:
  - Partial λ-calculus
  - type-class oriented shallow polymorphism
  - type constructors
  - HOLCF-style fixed-point recursion

CASL

```
spec StrictPartialOrder =
  %% Let's start with a simple example !
  sort Elem
  pred __<__ : Elem * Elem %% pred abbreviates predicate
  forall x , y, z : Elem
  . not x < x  %(strict)%
  . x < y => not y < x %(asymmetric)%
  . x < y /\ y < z => x < z %(transitive)%
  %{ Note that there may exist x, y such that
     neither x < y nor y < x. }%
end
```

Natural numbers in CASL

```
spec Nat =
  %% Natural numbers
  sort Nat
  ops 0 : Nat
    suc : Nat -> Nat
    pre : Nat ->? Nat
  %% subsort of positive numbers
  sort Pos < Nat
  forall n:Nat
  . n in Pos <=> not n=0
  %% alternative shorthand syntax
  sort Pos = { n:Nat . not n=0 }
end
```

Semantics of CASL

Many-sorted first-order structures:

- one universe of discourse per sort
- subsorts are interpreted as injective functions
- extensions for predicates
- functions may be partial

Evaluation of sentences

- as in first-order logic
- terms may not denote, due to presence of partial functions
- atomic formulas with non-denoting terms evaluate to false
  - as in negative free logic
- normal equality = holds if both terms do not denote, or denote the same thing
- existential equality =e= holds if both terms denote, and do denote the same thing

Free datatypes in CASL

```
spec Nat =
  free type Nat ::= 0 | suc(Nat)
end
```

This is shorthand for

```
spec Nat =
  sort Nat
  ops 0 : Nat
      suc : Nat -> Nat
  generated type Nat ::= 0 | suc(Nat)
  forall x,y : Nat
  . not 0 = suc(x)   %% disjoint images of constructors
  . suc(x)=suc(y) => x=y %% injectivity of constructors
end
```

Generated datatypes in CASL

The sentence

generated type `Nat ::= 0 | suc(Nat)`

holds in a first-order structure M, if for each element in $a \in M_{Nat}$, there is a term t built with `0` and `suc` such that

$$M(t) = a$$

This cannot be expressed in first-order logic!

Free datatypes in CASL, cont'd

```
spec List =
  sort Elem
  free type List ::= nil | cons(Elem,List)
end
```

This is shorthand for

```
spec List =
  sorts Elem, List
  ops nil : List
      cons : Elem * List -> List
  generated type List ::= nil | cons(Elem,List)
  forall x,x1,x2:Elem; l,l1,l2 : List
  . not nil = cons(x,l)
      %% disjoint images of constructors
  . cons(x1,l1) = cons(x2,l2) => (x1=x2 /\ l1=l2)
      %% injectivity of constructors
```

Generated datatypes in CASL, cont'd

```
generated type List ::= nil | cons(Elem,List)
```

holds in a first-order structure M, if for each element in $a \in M_{List}$, there is

- a term t built with `nil` and `cons` and variables over sort `Elem`
- a variable valuation ν,

such that

$$M_\nu(t) = a$$

Recursion over free datatypes in CASL

```
spec Nat =
  free type Nat ::= 0 | suc(Nat)
then %def
  ops   __ + __, __ * __   :   Nat * Nat ->  Nat;
  forall m,n,k : Nat
  . 0 + m = m                     %(add_0_Nat)%
  . suc(n) + m = suc(n + m)       %(add_suc_Nat)%
  . 0 * m = 0                     %(mult_0_Nat)%
  . suc(n) * m = (n * m) + m      %(mult_suc_Nat)%
then %implies
  . m + 0 = m                     %(add_0_Nat_right)%
  . m+(n+k) = (m+n)+k             %(add_assoc_Nat)%
  . m+suc(n) = suc(m+n)           %(add_suc_Nat)%
  . m+n = n+m                     %(add_comm_Nat)%
```

The implied sentences are *inductive theorems*.

Recursive definitions

Theorem

A recursive definition over free datatypes such that the definition has non-overlapping and exhaustive patterns is a definitional extension of the free datatype.

Hets can check this (i.e. the validity of the **%def** annotation).

Recursion over free datatypes in CASL, cont'd

```
spec List =
    free type Nat ::= 0 | suc(Nat)
    sort Elem %% loose interpretation
    free type List ::= nil | cons(Elem; List)
then %def
    ops concat : List * List -> List;
        length : List -> Nat;
    forall x:Elem; K, L, M:List
    . concat(nil, K) = K %(concat_nil)%
    . concat(cons(x,K), L) = cons(x, concat(K, L)) %(concat_
    . length(nil) = 0 %(length_nil)%
    . length(cons(x, L)) = suc(length(L)) %(length_NeList)%
then %implies
    forall K, L, M:List
    . concat(concat(K,L),M) = concat(K,concat(L,M))
```

Polymorphism and higher-order functions in HasCASL

```
logic HasCASL
spec List =
 var a : Type
 free type List a ::= Nil | Cons a (List a)
 ops head        : forall a:Type . List a ->? a;
     foldr       : forall a, b:Type
                     . (a * b ->? b) * b * List a ->? b;
     foldl       : forall a, b:Type
                     . (a * b ->? a) * a * List b ->? a;
     map         : forall a, b:Type
                     . (a ->? b) * List a ->? List b;
     __++__      : forall a:Type
                     . List a * List a -> List a;
```

Polymorphism and higher-order relations in HasCASL

```
%{ Relations and partial equivalence relations (PERs) }%
logic HasCASL
spec Relation =
  var S : Type
  ops reflexive, symmetric, transitive : Pred(Pred(S*S))
  forall r:Pred(S*S)
  . reflexive r <=> forall x:S . r(x,x)
  . symmetric r <=> forall x,y:S . r(x,y) => r(y,x)
  . transitive r <=>
     forall x,y,z:S . r(x,y) /\ r(y,z) => r(x,y)
  type PER S = {r : Pred(S*S) . symmetric r /\ transitive r}
  op dom : PER S -> Pred S
  forall x:S; r: PER S
   . x isIn dom r <=> (x,x) isIn r
```

Generated datatypes in HasCASL

The sentence

```
generated type Nat ::= 0 | suc(Nat)
```

can be directly expressed as induction axiom in second-order logic:

```
forall M:Pred(Nat)
. (M(0) /\ forall n:Nat . M(n) => M(suc(n))) =>
     forall n:Nat . M(n)
```

Type classes in HasCASL

```
logic HasCASL
spec Ord =
  class  Ord {
    var    a: Ord
    fun    __<=__ : Pred (a * a)
    var    x, y, z: a
    . x <= x
    . x <= y /\ y <= z => x <= z
    . x <= y /\ y <= x => x = y
  }
var a, b: Ord
type  instances a * b: Ord
vars  x, z: a; y, w: b
. (x, y) <= (z, w) <=> x <= z /\ y <= w
```

Type Classes and Polymorphism

- *Type classes* can be declared or defined:
  classes $Ord;\ Eq < Ord;\ Num = \{a : Type \bullet a < Int\}$
- *Type constructors* may have classes in their *arities*:
  var $a : Eq$
  type $List\ a : Eq$
- Operators and axioms may be *polymorphic* over classes:
  var $a : Ord$
  op $max : List\ a \rightarrow ?\ a$

The Partial λ-Calculus

- Terms need not denote
- *Partial* function types $s_1 \dots s_n \rightarrow ?t$
- λ-abstraction produces *partial* functions
- correspondingly geared deduction

The partial λ-calculus (cont'd)

- *Predicates* are partial functions into unit, but...
- *logic* within λ-abstractions initially limited to truth, conjunction
- Thus: *no Russell-type paradoxes*

Semantics

- Models are *syntactical λ-algebras* (e.g. Breazu-Tannen/Meyer 1985):
  - Interpret *all terms* as partial functions
  - Substitution = composition, variables = projections
  - *Require* compatibility with deduction
- These are 'the same' as the natural categorical models — i.e. functors into partial cartesian closed categories (CSL 03)

Intensionality

Models are *intensional*

- Allows e.g. topos models
- Avoids problems such as incompleteness and non-existence of initial models
- If desired, extensionality can be *specified*
- *Internally*, everything is extensional (Mitchell/Scott 1989)

Semantics of Polymorphism

- *Classes* are subsets of the (syntactical) type universe
- Polymorphic types, operators, and axioms are at the first level coded by collections of instances
  (second level: extension models, $\rightarrow$ institution)
- Axioms and operators *may* be 'attached' to classes to express proof obligations for instances:

  class *Ord* {...} %% Order relation & axioms
  type **instance** *Nat*
  ... %% Ordering on the naturals

N.B.: System F + HOL is known to be inconsistent!

The Internal Logic

- *Specify* internal *equality*
- *Define* internal logic $((\forall x.\,\phi) = ((\lambda x.\,\phi) = \lambda x.\,tt)$ etc.)
- The internal logic is *intuitionistic* (essentially topos logic minus unique choice; *codes* dependent types)
- Extensionality implies *classical* logic!
- Datatypes: no junk/no confusion axioms in the internal logic

Recursion

- HOLCF-like specification of cpo's (chain complete) as a type class
- Continuous function spaces $s \overset{\text{cont}}{\longrightarrow} t$, $s \overset{\text{cont}}{\longrightarrow}? \, t$
- fixed point operator $Y : (a \to a) \to a$, where $a : Cppo$

Recursion: Syntax

- Standard functional programming syntax within **program** blocks
- Abbreviate $f = Y(\lambda f \bullet \alpha)$ by

$$f\ x = \alpha\ x$$

- Pattern syntax for recursive functions on datatypes, *let*-expressions, type inference
- In short: **program** blocks 'look and feel' a lot like Haskell

Tool support

- Implemented as part of HETS
- Parser
- Static analysis: typing, mixfix analysis
- Encoding of HasCasl subset into Isabelle/HOL
- Translation of executable subset into Haskell.

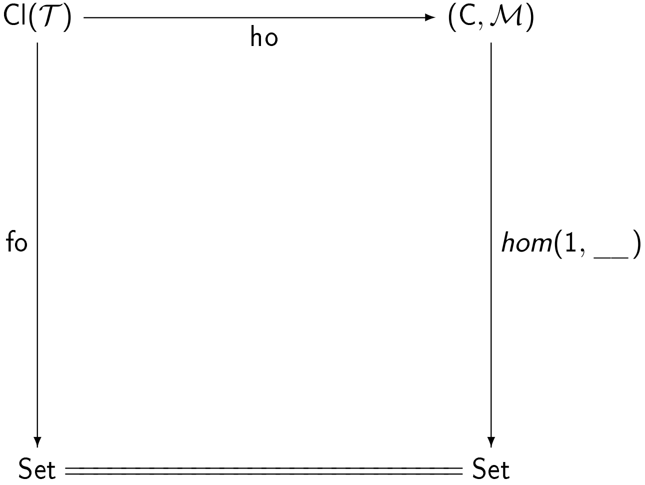
Conclusion

- HasCasl is conceptually simple
- ... and yet accommodates both logic and programming.
- Novel notion of semantics of the partial λ-calculus
- Programming features are *specified* within the language
- Executable HasCasl corresponds reasonably closely to Haskell

Future and 'Future' Work

- Complete the tool support
- Specification methodology
- Case study
- Basic libraries
- HasCasl for functional-imperative programming: do-notation, monadic computational logics
  (latest: computational logic with exceptions, AMAST 04)

The Global Element Construction



Process can be reversed: given first order $\mathsf{Cl}(\mathcal{T}) \to \mathsf{Set}$, construct higher order $\mathsf{Cl}(\mathcal{T}) \to (\mathsf{C}, \mathcal{M})$.

(*'Henkin models demote higher order to first order'*)