

Bootstrapping Types and Cotypes in HASCASL

Lutz Schröder*

DFKI-Lab Bremen and Department of Computer Science, University of Bremen

Abstract. We discuss the treatment of initial datatypes and final process types in the wide-spectrum language HASCASL. In particular, we present specifications that illustrate how datatypes and process types arise as bootstrapped concepts using HASCASL's type class mechanism, and we describe constructions of types of finite and infinite trees that establish the conservativity of datatype and process type declarations adhering to certain reasonable formats. The latter amounts to modifying known constructions from HOL to avoid unique choice; in categorical terminology, this means that we establish that quasitoposes with an internal natural numbers object support initial algebras and final coalgebras for a range of polynomial functors, thereby partially generalizing corresponding results from topos theory.

Introduction

The formally stringent development of software in a unified process calls for wide-spectrum languages that support all stages of the development process, including requirements, design, and implementation. In the CASL language family [3], this role is played by the higher-order CASL extension HASCASL [19, 24]. Like in first-order CASL, a key feature of HASCASL is support for inductive datatypes, which appear in the specification of the functional correctness of software. In the algebraic-coalgebraic language COCASL [11], this concept is complemented by coinductive types, which appear as state spaces of reactive processes. Many issues revolving around types of either kind gain in complexity in the context of the enriched language HASCASL; this is related both to the presence of additional language features such as higher order types and type class polymorphism and to the nature of the underlying logic of HASCASL, an intuitionistic higher order logic of partial functions without unique choice which may, with a certain margin of error, be thought of as the internal logic of quasitoposes.

Here, we discuss several aspects of HASCASL's concept of inductive datatype, as well as the perspective of adding coinductive types to HASCASL. To begin, we present the syntax and semantics of inductive datatypes, which may be equipped with reachability constraints or intiality constraints; both types of constraints may be relatively involved due to the fact that constructor arguments may have complex composite types. We then go on to show how initial datatypes may be specified in terms of HASCASL's type class mechanism. On the one hand,

* This work forms part of the DFG-project HASCASL (KR 1191/7-2)

this shows that initial datatypes need not be regarded as a built-in language feature, but may be considered as belonging into a ‘HASCASL prelude’. On the other hand, the specifications in question give a good illustration of how far the type class mechanism may be stretched. We then briefly discuss how a simple dualization of these specifications describes final process types in the style of CoCASL; thus, the introduction of such types into HASCASL would merely constitute additional syntactic sugar (although concerning the relationship to CASL and CoCASL, for both datatypes and process types certain caveats apply related to HASCASL’s Henkin semantics).

Finally, we tackle the issue of the conservativity of datatype and process type declarations. We follow the method employed in standard HOL [14, 2], which consists in defining a universal type of trees and then carving out the desired inductive or coinductive types. However, the constructions need to be carefully adapted in order to cope with the lack of unique choice; in particular, it turns out that the existence of final process types hinges on the presence of an *internal* natural numbers object (NNO). Abstracting our results to the categorical level, we prove, in partial generalization of corresponding results for toposes [8], that any quasitopos with an internal NNO supports initial algebras and final coalgebras for certain classes of polynomial functors.

1 HASCASL

The wide-spectrum language HASCASL [19] extends the standard algebraic specification language CASL by intuitionistic partial higher order logic, equipped with a set-theoretic Henkin semantics, an extensive type class mechanism, and HOLCF-style support for recursive programming. HASCASL moreover provides support for functional-imperative specification and programming in the shape of monad-based computational logics [20, 22, 21, 25]. Tool support for HASCASL is provided in the framework of the Bremen heterogeneous tool set Hets [10]. We expect the reader to be familiar with the basic CASL syntax (whose use in our examples is, at any rate, largely self-explanatory), referring to [3, 12] for a detailed language description. Below, we briefly review the HASCASL language features most relevant for the understanding of the present work, namely type class polymorphism and certain details of HASCASL’s higher order logic; cf. [24] for a full language definition.

The logic of HASCASL is based on the partial λ -calculus [9]. It is distinguished from standard HOL by having intuitionistic truth values and partial function types $t \rightarrow? s$ (besides total function types $t \rightarrow s$). The set-theoretic semantics is given by *intensional Henkin models*, where function types are equipped with application operators but are neither expected to contain all set-theoretic functions nor indeed to consist of functions; in particular, different elements of the function type may induce the same set-theoretic function. Such models are essentially equivalent to models in (varying!) partial cartesian closed categories (pccc’s) with equality [18]; these categories are slightly more general than quasitoposes [1], which can be seen as finitely cocomplete pccc’s with equality.

The difference between the HASCASL logic and the more familiar topos logic [7] is the absence of unique choice [18], where we say that a type a admits *unique choice* if a supports *unique description* terms of the form $(\iota x : a. \phi) : a$ designating the unique element x of a satisfying the formula ϕ (which may of course mention x), if such an element exists uniquely (this is like Isabelle/HOL’s THE [13]). In HASCASL, the unique choice principle may be imposed if desired by means of a polymorphic axiom. The lack of unique choice requires additional effort in the construction of tree types establishing the conservativity of datatype and process type declarations; this is the main theme of Sect. 5. The motivation justifying this effort is twofold:

- Making do without unique choice essentially amounts to admitting models in quasitoposes (in fact, pccc’s with equality) rather than just in toposes. Interesting set-based quasitoposes include pseudotopological spaces and reflexive relations; further typical examples are categories of extensional presheaves, including e.g. the category of reflexive logical relations, and categories of assemblies, both appearing in the context of realizability models [15, 16]. Quasitoposes also play a role in the semantics of parametric polymorphism [4].
- A discipline of avoiding unique choice leads to constructions which may be easier to handle in machine proofs than ones containing unique description operators (cf. e.g. the explicit warning in [13], Sec. 5.10).

HASCASL’s shallow polymorphism revolves around a notion of type class. Type classes are syntactic subsets of *kinds*, where kinds are formed from *classes*, including a base class *Type* of all types, and the type function arrow \rightarrow . Classes are declared by means of the keyword **class**; e.g.

class *Functor* < *Type* \rightarrow *Type*

declares a class *Functor* of type constructors, i.e. operations taking types to types. Types are declared with associated classes (or with default class *Type*) by means of the keyword **type**; e.g. a type constructor F of class *Functor* is declared by writing

type $F : \textit{Functor}$

Such declarations may be generic; e.g. if *Ord* is a class, then we may write

var $a, b : \textit{Ord}$
type $a \times b : \textit{Ord}$

thus imposing that the class *Ord* is closed under products; note how the keyword **var** is used for both standard variables and type variables. Operations and axioms may be polymorphic over any class, i.e. types of operations and variables may contain type variables with assigned classes.

In order to ensure the institutional satisfaction condition (invariance of satisfaction under change of notation), polymorphism is equipped with an *extension semantics* [23]; the only point to note for purposes of this work is that as a consequence, a specification extension is, in CASL terminology, (model-theoretically)

conservative (i.e. admits expansions of models) iff it introduces names for entities already expressible in the present signature. In the case of types, this means that e.g. a datatype declaration is conservative iff it can be implemented as a subtype of an existing type.

2 Datatypes in HASCASL

HASCASL supports recursive datatypes in the same style as in CASL. To begin, an unconstrained datatype t is declared along with its constructors $c_i : t_{i1} \rightarrow \dots \rightarrow t_{ik_i} \rightarrow t$ by means of the keyword **type** in the form

$$\mathbf{type} \ t ::= \ c_1 \ t_{11} \ \dots \ t_{1k_1} \mid \dots \mid \ c_n \ t_{n1} \ \dots \ t_{nk_n}$$

(mutually recursive types are admitted as well, but omitted from the presentation for the sake of readability). Here, t is a pattern of the form $C \ a_1 \ \dots \ a_r$, $r \geq 0$, where C is the type constructor (or type if $r = 0$) being declared and the a_i are type variables. The t_{ij} are types whose formation may involve C and the a_i . Optionally, selectors $sel_{ij} : t \rightarrow? t_{ij}$ may be declared by writing $(sel_{ij} :?t_{ij})$ in place of t_{ij} . All this is syntactic sugar for the corresponding declarations of types and operations, and equations stating that selectors are left inverse to constructors.

Data types may be qualified by a preceding **free** or **generated**. The **generated** constraint introduces an induction axiom; this corresponds roughly to term generatedness ('no junk'). The **free** constraint ('no junk, no confusion') instead introduces an implicit fold operator, which implies both induction and a primitive recursion principle. If one of these constraints is used, then recursive occurrences (in the t_{ij}) of C are restricted to the pattern $C \ a_1 \ \dots \ a_r$ appearing on the left hand side; i.e. HASCASL does not support polymorphic recursion. If a **free** constraint is used, then additionally recursive occurrences of C are required to be strictly positive w.r.t. function arrows, i.e. occurrences in the argument type of a function type are forbidden. In more detail, the semantics of the constraints is as follows.

2.1 Generated types

If t as above has only t and types from the local environment as arguments of constructors, then a generatedness constraint for t induces an induction axiom for t as in CASL. Unlike in CASL, the induction axiom is however expressible in HASCASL, i.e. generation constraints in HASCASL are just syntactic sugar. For constructors with composite argument types, the induction axiom more generally states that a predicate P on t holds universally if a family of *extended induction predicates* P_s on composite types s is closed under the constructors, where the P_s are subject to the following conditions. The base cases are $P_t = P$ and $P_s = \top$ if s is from the local environment or a type variable. The remaining clauses are

- *Partial function spaces*: $P_{s \rightarrow ?u} f \iff \forall x : s. (P_s x \wedge \text{def } f(x)) \Rightarrow P_u f(x)$.

- *Total function spaces*: $P_{s \rightarrow u}$ is the restriction of $P_{s \rightarrow ?u}$ to $s \rightarrow u$.
- *Product types*: $P_{s \times u}(x, y) \iff P_s x \wedge P_u y$.
- *Applications* $D s_1 \dots s_q$ of a type constructor D from the local environment to types s_1, \dots, s_q , where at least one s_j contains a recursive occurrence of t : $P_{D s_1 \dots s_q}$ is required to be closed under all operations with result type $D s_1 \dots s_q$ (which are necessarily newly arising instances of polymorphic operators). Note that $P_{D s_1 \dots s_q}$ is not in general uniquely defined by this requirement.

Remark 1. If a type constructor D from the local environment has a generation constraint, then the closedness requirement on extended induction predicates for applications of D already follows from closedness under the operators in the constraint. However, the induction axiom also makes sense if D has no generation constraint; it then states that t is generated from the reachable part of D .

Generally, every HASCASL specification, in particular every datatype declaration, has a term model [18], which satisfies the above induction axiom. However, we stress that the induction axiom does *not* imply that elements of a generated datatype whose constructors have functional arguments are reachable by the constructors and λ -abstraction. In particular, induction axioms do not preclude interpreting functional types by full function spaces, which cannot be term generated for infinite types.

Finally, note that, due to Henkin semantics, generation constraints in HASCASL are weaker than in CASL, and in particular do not exclude non-standard models. However, proof-theoretically, this difference disappears — at least if the standard finitary induction rule is used. Only if stronger (e.g. infinitary) forms of induction are used, the difference becomes relevant.

Example 2. The following datatype declaration might form part of a specification of systems with unordered finite branching:

generated type $Set\ a ::= empty \mid add\ a\ (Set\ a)$
generated type $Sys\ b ::= node\ b\ (Set\ (Sys\ b))$

The induction axiom for Set is as in CASL; the induction axiom for $Sys\ b$ is

$$\left. \begin{array}{l} (\forall x : b; s : Set\ (Sys\ b) \bullet Q\ s \Rightarrow P\ (node\ x\ s)) \wedge \\ Q\ empty \wedge \\ (\forall s : Set\ (Sys\ b); t : Sys\ b \bullet (Q\ s \wedge P\ t) \Rightarrow Q\ (add\ t\ s)) \end{array} \right\} \Rightarrow \forall t : Sys\ b \bullet P\ t.$$

As an example with functional constructors, consider a datatype of at most countably branching trees,

generated type $CTree\ a ::= leaf\ a \mid branch\ (Nat \rightarrow? CTree\ a)$

(with Nat previously declared), which for $CTree$ gives rise to the induction axiom

$$\left. \begin{array}{l} (\forall x : a \bullet P\ (leaf\ x)) \wedge \\ (\forall f : Nat \rightarrow? CTree \bullet \\ (\forall x : Nat \bullet def\ f\ x \Rightarrow P\ (f\ x)) \Rightarrow P\ (branch\ f)) \end{array} \right\} \Rightarrow \forall t : CTree \bullet P\ t.$$

2.2 Free types

The semantics of free datatypes is determined by a fold operator, i.e. free datatypes are explicitly axiomatized as initial algebras. As indicated above, recursive occurrences of free types must be strictly positive, i.e. types like $L ::= \text{abs } (L \rightarrow L)$ and $L a ::= \text{abs } ((L \rightarrow a) \rightarrow a)$ are illegal, while

free type $\text{Tree } a b ::= \text{leaf } b \mid \text{branch } (a \rightarrow \text{Tree } a b)$

is allowed. Free datatypes are seen as initial algebras for functors. In the standard case, the functors in question are polynomial functors, with multiple arguments of constructors represented as products and alternatives represented as sums. E.g. the signature of the tree type above induces the functor F_{ab} given by

$$F_{ab}c = b + (a \rightarrow c).$$

The general mechanism for extracting functors from datatype declaration is explained in more detail in Sect. 3. This mechanism relies on type classes to ensure that user-defined type constructors appearing in constructor arguments are actually functors. The latter will in particular be the case if type constructors are defined as free datatypes with functorial parameters; e.g. the above declaration induces a functor taking b to $\text{Tree } a b$.

For now, we take for granted that a free datatype t as in the beginning of this section can be regarded as an initial algebra $\alpha : F t \rightarrow t$ for a functor F . Initiality is expressed by means of a polymorphic fold operation

$$\text{fold} : (F b \rightarrow b) \rightarrow t \rightarrow b$$

for $b : \text{Type}$, and an axiom stating that, for $d : F b \rightarrow b$, $\text{fold } d$ is the unique F -algebra morphism from α to d , i.e. the unique map f satisfying $d \circ (F f) = f \circ \alpha$.

Remark 3. Unlike in CASL, the meaning of **free type** does not coincide with that of the corresponding structured free extension **free { type ... }**, which would require all newly arising function types to be also freely term generated.

Remark 4. The reason for using an explicit fold operation in place of a combination of induction ('no junk') and term distinctness ('no confusion') is the absence of a unique choice operator, without which the existence of folds fails to be derivable from the no-junk-no-confusion principles. (E.g. in the quasitopos of pseudotopological spaces, the no-junk-no-confusion axioms determine the underlying set of an initial algebra but not its topological structure.) Conversely, one easily sees that initiality implies induction and term distinctness.

From the fold operation, one obtains a primitive recursion principle in the standard way (i.e. by means of a simultaneous recursive definition of the identity). From the latter, in turn, we obtain as a special case a case operator, denoted in the form

$$\text{case } x \text{ of } c y_1 \dots y_l \rightarrow f y_1 \dots y_l \mid \dots$$

Example 5. Consider the following free datatype definitions.

```

free type List a ::= nil | cons (a; List a)
free type Tree a b ::= leaf a | branch (b → List (Tree a b))

```

The declaration of *List* *a* induces the standard fold operation for lists. Moreover, the type class mechanism (cf. Sect. 3) recognizes automatically that the type constructor *List* is a functor, and in particular generates the standard *map* operation. For *Tree*, we obtain a polymorphic fold operation

$$\text{fold} : (a \rightarrow c) \rightarrow ((b \rightarrow \text{List } c) \rightarrow c) \rightarrow \text{Tree } a \ b \rightarrow c.$$

This operation is axiomatized as being uniquely determined by the equations

$$\text{fold } f \ g \ (\text{leaf } x) = f \ x \quad \text{and} \quad \text{fold } f \ g \ (\text{branch } s) = g \ (\text{map } (\text{fold } f \ g) \circ s).$$

3 Initiality via the Type Class Mechanism

The concept of free datatype described in the previous section may be regarded as bootstrapped, i.e. as being a HASCASL library equipped with built-in syntactic sugar rather than a basic language feature. The crucial point here is that HASCASL’s type class mechanism allows talking about functorial signatures, algebras for a functor, and, importantly, homomorphisms. The specifications shown below are real HASCASL specifications, parsed and prettyprinted using the Heterogeneous Tool Set (Hets) [10].

Figure 1 shows the constructor class of functors. Mutually recursive or parametrized datatypes require n -ary functors for $n \in \mathbb{N}$, and in fact occasionally type constructors which are functorial only in some of their arguments; since HASCASL does not feature dependent classes, the corresponding classes need to be specified one by one, as exemplified in Fig. 1 by a specification of bifunctors. This is not a problem in practice, as typically only small values of n are needed; the specification of bifunctors illustrates how $n + 1$ -ary functors can be specified recursively in terms of n -ary functors.

Remark 6. One might envision a single specification of functors of arbitrary finite arity by abuse of syntax, as follows: declare a class *Typelist* and type constructors *Nil* : *Typelist*, *Cons* : *Type* → *Typelist* → *Typelist*, and define *Functor* as a subclass of *Typelist* → *Type*. (Undesired semantic side effects may be eliminated by specifying the types *Nil*, *Cons* *a* *Nil* etc. to be singletons.) Similar tricks work in Haskell [6] but rely on multi-parameter type classes, which are currently excluded from the HASCASL design.

For purposes of conservativity of datatype declarations, the class of polynomial functors (bifunctors etc.), shown in Fig. 2, plays an important role. An n -ary functor is polynomial if it can be generated from projection functors (the identity functor if $n = 1$) and constant functors by taking finite sums and products. To avoid circularity, the required type constructors in have been specified

```

spec FUNCTOR =
vars  a, b, c : Type; x : a; f : a → b; g : b → c
ops   _comp_ : (b → c) × (a → b) → a → c;
        id : a → a
    • id x = x
    • (g comp f) x = g (f x)
class Functor < Type → Type
{vars  a, b, c : Type; F : Functor; f : a → b; g : b → c
op    map : (a → b) → F a → F b
    • map id = id : F a → F a
    • map (g comp f) : F a → F c = (map g) comp (map f)
}
class Bifunctor < Type → Functor
{vars  a, b, c, d : Type; F : Bifunctor; f : a → b; g : b → c
op    parmap : (a → b) → F a d → F b d
    • parmap id = id : F a d → F a d
    • parmap (g comp f) : F a d → F c d = (parmap g) comp (parmap f);
    • (parmap f) comp (map h) : F a c → F b d = (map h) comp (parmap f)
}

```

Fig. 1. HASCASL specification of functors

without recourse to free type declarations. Recall that constructor/selector pairs such as *mkId/getId* impose that the selector *getId* is left inverse to the constructor *mkId*. Axioms stating surjectivity of constructors are omitted in the figure except in the case of *Prod* (for *Sum*, joint surjectivity of *inl* and *inr* follows from the axiom for *sumcase*). Similarly, the obvious definitions of the associated *map* operations are omitted, except in the case of sums. Note that HASCASL does not provide a way to exclude unwanted ('junk') further instance declarations for the class *PolyFunctor*, i.e. to say that the class is generated by the given generic instances.

In Fig. 3, we present a specification of algebras for a functor. The set of algebra structures for a functor *F* on a type *a* is given by the type constructor *Alg*, which depends on both *F* and *a* and thus has the profile *Functor* → *Type* → *Type*; it is given as a type synonym for the type *F a* → *a*. Similarly, the type constructor *AlgMor* for algebra morphisms depends on *F* and types *a*, *b* forming the carriers of the domain and the codomain, respectively. Algebra morphisms are treated as triples consisting of two algebra structures and a map between the carriers.

Initial algebras are then specified by means of two operations: a type constructor *InitialCarrier* that assigns to a functor the carrier set of its initial algebra, and a polymorphic constant *initialAlg* which represents the structure map of an initial algebra for *F* on this carrier. Initiality of this algebra is specified by means of an explicit fold operation. As initial algebras will exist only

```

spec POLYFUNCTORS = FUNCTOR
then class PolyFunctor < Functor
vars F, G : PolyFunctor; a, b : Type
types Const a, Id, Sum F G, Prod F G : PolyFunctor
type Const a b ::= mkConst (getConst : a)
type Id b ::= mkId (getId : b)
type Sum F G b ::= inl (F b) | inr (G b)
vars f : F b →? a; g : G b →? a; h : Sum F G b →? a
op sumcase : (F b →? a) → (G b →? a) → (Sum F G) b →? a
• h = sumcase f g ⇔
  ∀ x : F b; y : G b • h (inl x) = f x ∧ h (inr y) = g y
type Prod F G b ::= pair (outl : F b; outr : G b)
var k : a → b
• (map k : Sum F G a → Sum F G b) =
  sumcase (inl comp (map k)) (inr comp (map k))
• ...
• ∀ z : Prod F G b • z = pair (outl z, outr z);
• ...
class PolyBifunctor < Type → PolyFunctor; PolyBifunctor < Bifunctor
...

```

Fig. 2. HASCASL specification of polynomial functors

for some functors, the abovementioned operations are defined only on a subclass *DTFunctor* (‘datatype functor’) of *Functor*. We declare the class *PolyFunctor* (Fig. 2) to be a subclass of *DTFunctor*, thus stating that all polynomial functors have initial algebras as proved in Sect. 5; due to possible junk in the class *PolyFunctor* (see above), this is consistent but non-conservative. Moreover, we state that initial algebras depend functorially on parameters in the case of polynomial bifunctors and that the arising functor again has an initial algebra (as nested recursion may be coded by mutual recursion in the standard way [5]).

We conclude with a brief description of how the data above are generated by the static analysis of actual HASCASL specifications. The functor F associated to the declaration of a datatype t as in the beginning of Sect. 2 is a sum of n functors F_i , one for each constructor c_i ; the functor F_i , in turn, is a product of k_i functors F_{ij} , corresponding to the t_{ij} . The t_{ij} are, by the restrictions laid out in Sect. 2, inductively generated from types in the local environment, t , and the a_i by taking products, exponentials $s \rightarrow t$ or $s \rightarrow? t$, where s is a type formed from the a_i and the local environment, and applications $D s_1 \dots s_l$ of type constructors from the local environment, the latter subject to the restriction that if s_i contains a recursive occurrence of t , then the dependence of D of its i -th argument must be functorial. The latter property is tracked by means of the type class mechanism; in particular, instances of *Functor* are generated automatically for parametrized datatypes such as the type *List a* of Example 5. Given this format of the t_{ij} , it is

straightforward to associate a functor to each t_{ij} (using further generic instances of *Functor*, in particular exponentials and closure under functor composition). Finally, an instance $F : DT\text{Functor}$ is generated if not already induced by the generic instances shown in Fig. 3, in which case consistency of the datatype declaration becomes the responsibility of the user.

```

spec ALGEBRA = POLYFUNCTORS
then vars  F : Functor; a, b : Type
type  Alg F a := F a → a
op    ---→--- : Pred ((a → b) × (Alg F a) × (Alg F b))
vars  f : a → b; alpha : Alg F a; beta : Alg F b
• (f :: alpha → beta) ⇔ (beta comp (map f)) = (f comp alpha)
type  AlgMor F a b = {(f, alpha, beta) : (a → b) × Alg F a × Alg F b •
                        f :: alpha → beta }

classes DTFunctor < Functor; PolyFunctor < DTFunctor
{vars  F : DTFunctor; a : Type
type  InitialCarrier F
ops  initialAlg : Alg F (InitialCarrier F);
      ifold : Alg F a → InitialCarrier F → a
vars  alpha : Alg F a; g : InitialCarrier F → a;
• (g :: initialAlg → alpha) ⇔ g = ifold alpha;
}
var   G : PolyBifunctor
type  ParamDT G a ::= mkPDT (getPDT : InitialCarrier (G a))
• ∀ l : ParamDT G a • mkPDT (getPDT l) = l
type  ParamDT G : DTFunctor
vars  l : ParamDT G a; b : Type; f : a → b
• map f l = mkPDT (ifold (initialAlg comp (parmap f)) (getPDT l))

```

Fig. 3. HASCASL specification of initial algebras

Using the *sumcase* operation of Fig. 2, one can now gather the constructors into a structure map $c : F t \rightarrow t$; the freeness constraint in the above datatype declaration then translates into the declaration of a two-sided inverse g of *ifold* c . The fold operation on t is obtained as $fold \alpha = (ifold \alpha) \circ g$.

4 Process Types in HASCASL

Although process types in the style of CoCAsL, so-called *cotypes* [11], are not presently included in the HASCASL design, the results of the previous section indicate that cotypes could be integrated seamlessly into HASCASL. A cotype is a syntactic representation of a coalgebra for a signature functor. Cotypes are declared in a similar style as types; the crucial difference is that, while selectors

are optional in a datatype, they are mandatory in a cotype, as they constitute the actual structure map of the coalgebra, and constructors are optional. Moreover, a cotype induces axioms guaranteeing that the domains of selectors in the same alternative agree, and that the domains of all alternatives are disjoint and jointly exhaustive. Thus, the intended models e.g. of the cotype

$$\mathbf{cotype} \textit{ Proc} ::= (\textit{out} :? a; \textit{next} :? \textit{Proc}) \mid (\textit{spawnl}, \textit{spawnr} :? \textit{Proc})$$

are coalgebras for the functor F given by $FX = a \times X + X \times X$. The extraction of functors from cotype signatures is analogous to the case of types as explained in Sect. 3. The semantics of cotypes, in particular *cofree* (i.e. final) cotypes, builds on a dual of the specification of algebras (Fig. 3), where the type of algebras is replaced by a type $\mathit{Coalg} F a := a \rightarrow F a$, the definition of homomorphisms is correspondingly modified, and initiality is replaced by finality, i.e. unique existence of morphisms *into* the final coalgebra given by an *unfold* operation. The domains of the selectors can in principle be arbitrary types; of course, this will not in general guarantee existence of final coalgebras. We omit the discussion of cogeneratedness of cotypes.

The only subtle point in the matching between cotype declarations in HAS-CASL and coalgebras is that, unlike in a set-based framework such as COCASL, the abovementioned conditions on domains of selectors of a cotype t with associated functor F are insufficient to guarantee existence of a single structure map $t \rightarrow F t$, the point being, again, the absence of unique choice. We thus impose, instead of just disjointness and joint exhaustiveness of the domains, that the cotype is the coproduct of the domains, by introducing a polymorphic partial case operation similar to the *sumcase* operation of Fig. 2. E.g. for the cotype *Proc* above and $f, g : \textit{Proc} \rightarrow? a$, $\textit{case } f \textit{ } g = h : \textit{Proc} \rightarrow a$ is defined whenever the domains of f and g equal the domains of *out* and *spawnl*, respectively, and in this case h extends f and g . (Under unique choice, *case* is definable.)

5 Conservativity of Datatypes and Process Types

Free datatypes in HASCASL are not necessarily conservative extensions of the local environment. Already the naturals may be non-conservative: as discussed in Sect. 1, conservative extensions can only introduce names for entities already in the present signature, and a given model might interpret all types as finite sets. This problem arises already in standard HOL, where the construction of initial datatypes [14, 2] is based on the naturals. The said construction makes heavy use of unique choice, so that the question arises whether similar constructions are possible in HASCASL.

It turns out that the construction of finitely branching datatypes can be modified to avoid unique choice, assuming, besides a type *Nat* of natural numbers with associated primitive recursion principle, sum types (denoted by $+$ in the interest of readability, with injection functions *inl* and *inr* as usual and extraction functions $\textit{outl} : a + b \rightarrow? a$, $\textit{outr} : a + b \rightarrow? b$) and an initial type, which shows up in the form of an undefined partial constant $\textit{bot} :? a$ at every type a (under

unique choice, sum types and an initial type can be constructed). We describe the construction for the simplified situation where we have a single unparametrized datatype t with n constructors c_i of arity k_i , with arguments either of type t or of some type a (extending this to mutually recursive types, complex argument types — excluding function types —, and parameters is straightforward).

To begin, the type $Path$ is defined as $Nat \rightarrow? Nat$. We put $nil = \lambda m \bullet m \text{ res bot}$ and $cons\ m\ s = \lambda k \bullet \text{case } k \text{ of } 0 \rightarrow m \mid r + 1 \rightarrow s\ r$. The crucial point is now the definition of the universal type: while in the construction of [14, 2], this is a type of sets of nodes, we put

$$DTree\ a = Path \rightarrow? ((a + Nat) \times Nat),$$

where for $f : DTree\ a$ and $p : Path$, $f\ p = (x, n)$ indicates that the subtree at p has size n and is either a leaf labelled y , if $x = inl\ y$, or a node labelled by the constructor c_i , if $x = inr\ i$. We embed a into $DTree\ a$ and define the constructors c_i as operations on $DTree\ a$ in the obvious way (with sizes determined by counting 1 for each leaf or constructor), and then take t to be the smallest subtype of $DTree\ a$ closed under the c_i . We put $size\ z = snd\ (z\ nil)$ for $z : t$, and for $j : Nat$, we define a generic j -th selector by $sel_j\ z = z \circ (cons\ j)$. Note that $size\ z > 0$ for all $z : t$.

It remains to be shown that we can construct the function $fold\ d_1 \dots d_n$, for functions d_i representing an algebra on a type b . We define a primitive recursive function $f : Nat \rightarrow ((DTree\ a) \rightarrow? (a + b))$ by

$$\begin{aligned} f\ 0\ z &= bot \\ f\ (n + 1)\ z &= (\text{case } fst\ (z\ nil) \text{ of } inl\ y \rightarrow inl\ y \\ &\quad | inr\ i \rightarrow inr\ (d_i\ (f\ n\ (sel_1\ z)) \dots (f\ n\ (sel_{n_i}\ z)))) \end{aligned}$$

(with extraction functions $outl, outr$ on $a + b$ appropriate for the argument types of d_i left implicit), and put $fold\ d_1 \dots d_n\ z = outr\ (f\ (size\ z)\ z)$.

In summary, the main changes w.r.t. standard HOL with unique choice concern the universal type, which is a type of partial functions rather than relations (reflecting the fact that functional relations need not be functions in the absence of unique choice), and the construction of primitive recursive functions, which can no longer rely on an inductive construction of their graphs. It is an open problem whether our use of the size function for this purpose can be either generalized or avoided so as to cover also infinitely branching datatypes such as the type $Tree\ a\ b$ from Example 5.

Somewhat surprisingly, a quite similar construction works also for final coalgebras. For simplicity, assume a cotype declaration of the form

$$\mathbf{cotype}\ t ::= (sel_{11} : t_{11}; \dots; sel_{1m_1} : t_{1m_1}) \mid \dots \mid (sel_{n1} : t_{n1}; \dots; sel_{nm_n} : t_{nm_n}),$$

where for some $1 \leq k_i \leq m_i$ and some type a from the local environment (the output type), $t_{ij} = t$ if $j \leq k_i$, and otherwise $t_{ij} = a$. (The generalization to mutually recursive cotypes, several output types, and types t_{ij} of the form $b \rightarrow s$, where b is an input type from the local environment, is straightforward.) Given

that initial datatypes have already been constructed, we may now take the type $Path$ to be the type $List\ Nat$ of lists of natural numbers, with constructors nil , $cons$ and the standard $snoc$ operation. Our universal type is then

$$PTree\ a = Path \rightarrow? (Nat + a)$$

(where it is crucial that we omit the additional Nat component present in $DTree\ a$). For $f : PTree$ and $p : Path$, the intended reading of $f\ p = x$ is that position p in the tree is a leaf labelled with output y if $x = inr\ y$, and a branch according to the i -th alternative in the above declaration if $x = inl\ i$. The carrier of the final cotype for the above declaration is then the subtype c of $PTree\ a$ consisting of those f such that

$$\begin{aligned} & \text{def } (outl\ (f\ nil)) \quad \text{and} \\ & \text{def } (f\ (snoc\ p\ j)) \iff (\exists i : Nat. f\ p = inl\ i \wedge 1 \leq j \leq k_i). \end{aligned}$$

(Note that $f\ p = inl\ i$ entails that $f\ p$ is defined.) For $1 \leq j \leq m_i$, we can then define $sel_{ij}\ f$ to be defined iff $f\ nil = inl\ i$, and in this case

$$sel_{ij}\ f = \begin{cases} \lambda p. f\ (cons\ j\ p) & j \leq k_i \\ outr\ (f\ (cons\ j\ nil)) & \text{otherwise,} \end{cases}$$

and these selectors can be gathered into a single coalgebra structure on c for the appropriate functor using the information from $outl\ f\ nil$. Given a further cotype d matching the above declaration, with selectors sel_{ij}^d gathered into a coalgebra structure α , we may, using the *case* operator on d discussed in Sect. 4, define a function alt giving, for $x : d$, the alternative $i = alt\ x : Nat$ relevant for x . We then define the morphism $u = unfold\ \alpha : d \rightarrow c$ recursively by

$$u\ x\ nil = inl\ i \quad \text{and} \quad u\ x\ (cons\ j\ p) = \begin{cases} u\ (sel_{ij}^d\ x)\ p & 0 \leq j \leq k_i \\ inr\ sel_{ij}^d\ x & k_i < j \leq m_i, p = nil \\ bot & \text{otherwise.} \end{cases}$$

Importantly, primitive recursion on lists is given by an operator (rather than just by a unique existence axiom), so that the above definition can be expressed as a term defining $unfold\ \alpha$ and indeed $unfold$ as a function.

While the above seems rather entangled in the specificities of HASCASL, the arguments are in fact general enough to work in any quasitopos, or indeed any pccc with equality (cf. Sect. 1). We thus obtain results of independent interest stating that a quasitopos supports certain datatypes and process types, thus partially generalizing known results for W -types in toposes (e.g. [8]). It should be noted that the definition of initial algebras given in Sect. 3 in fact relates to *internal* initial algebras, where initiality is defined by an internal universal quantifier and moreover embodied as an explicit fold operation; this requirement is stronger than the external definition of initial algebras phrased in terms of the existence of algebra morphisms in a category. An *internal natural numbers*

object (NNO) in this sense is not strictly needed in the construction of initial datatypes given above, but ensures that the datatypes constructed are, in turn, internal initial algebras. For the construction of final process types, however, the requirement that the NNO is internal is crucial. We record explicitly

Theorem 7. *Let \mathbf{C} be a quasitopos.*

- (a) *If \mathbf{C} has a NNO, then \mathbf{C} has initial algebras for functors T of the form $TX = \sum_{i=1}^n A_i \times X^{k_i}$ (with $k_i \in \mathbb{N}$ and constant parameter objects A_i).*
- (b) *If \mathbf{C} has an internal NNO, then \mathbf{C} has internal initial algebras for functors T of the form $TX = \sum_{i=1}^n A_i \times X^{k_i}$.*
- (c) *If \mathbf{C} has an internal NNO, then \mathbf{C} has internal final coalgebras for functors T of the form $TX = \sum_{i=1}^n (B_i \rightarrow A_i \times X^{k_i})$.*

Example 8. In every topological universe (i.e. well-fibred topological quasitopos over \mathbf{Set} [1]) \mathbf{C} , the set \mathbb{N} equipped with the discrete structure is an internal NNO: we have to show that the fold map $A \times (A \rightarrow A) \times \mathbb{N} \rightarrow A$, $(x, f, n) \mapsto f^n(x)$ is a \mathbf{C} -morphism. As \mathbb{N} is discrete, it suffices to show that the map $(x, f) \mapsto f^n(x)$ is a morphism for every n , which holds in any cartesian closed category.

6 Conclusion

We have laid out how initial datatypes and final process types are incorporated into HASCASL, and we have established the existence of such types for a broad class of signature formats. The main contribution in the latter respect is the avoidance of the unique choice principle, which means that, on a more abstract level, our constructions work in any quasitopos.

It remains to be seen whether the datatype constructions can be adapted to more general classes of signatures, in particular to datatype signatures with infinite branching. Support for datatypes with polynomial signatures is already implemented in Hets; support for more complex signatures, intertwined with HASCASL's type class mechanism, is forthcoming.

References

- [1] J. Adámek, H. Herrlich, and G. E. Strecker. *Abstract and Concrete Categories*. Wiley Interscience, 1990.
- [2] S. Berghofer and M. Wenzel. Inductive datatypes in HOL - lessons learned in formal-logic engineering. In *Theorem Proving in Higher Order Logics*, vol. 1690 of *LNCS*, pp. 19–36. Springer, 1999.
- [3] M. Bidoit and P. D. Mosses. *CASL User Manual*, vol. 2900 of *LNCS*. Springer, 2004.
- [4] L. Birkedal and R. E. Møgelberg. Categorical models for abadi and plotkin's logic for parametricity. *Math. Struct. Comput. Sci.*, 15, 2005.
- [5] E. L. Gunter. A broader class of trees for recursive type definitions for HOL. In *Higher Order Logic Theorem Proving and Its Applications*, vol. 780 of *LNCS*, pp. 141–154. Springer, 1993.

- [6] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Haskell Workshop*, pp. 96–107. ACM, 2004.
- [7] J. Lambek and P. J. Scott. *Introduction to Higher-Order Categorical Logic*. Cambridge, 1986.
- [8] I. Moerdijk and E. Palmgren. Wellfounded trees in categories. *Ann. Pure Appl. Logic*, 104:189–218, 2000.
- [9] E. Moggi. Categories of partial morphisms and the λ_p -calculus. In *Category Theory and Computer Programming*, vol. 240 of *LNCS*, pp. 242–251. Springer, 1986.
- [10] T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set. In *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 4424 of *LNCS*, pp. 519–522. Springer, 2007.
- [11] T. Mossakowski, L. Schröder, M. Roggenbach, and H. Reichel. Algebraic-co-algebraic specification in CoCASL. *J. Logic Algebraic Programming*, 67:146–197, 2006.
- [12] P. D. Mosses, editor. *CASL Reference Manual*, vol. 2960 of *LNCS*. Springer, 2004.
- [13] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, vol. 2283 of *LNCS*. Springer, 2002.
- [14] L. C. Paulson. Mechanizing coinduction and corecursion in higher-order logic. *J. Log. Comput.*, 7:175–204, 1997.
- [15] W. Phoa. An introduction to fibrations, topos theory, the effective topos and modest sets. Research report ECS-LFCS-92-208, Lab. for Foundations of Computer Science, University of Edinburgh, 1992.
- [16] G. Rosolini and T. Streicher. Comparing models of higher type computation. In *Realizability Semantics and Applications*, vol. 23 of *ENTCS*, 1999.
- [17] L. Schröder. The logic of the partial λ -calculus with equality. In *Computer Science Logic*, vol. 3210 of *LNCS*, pp. 385–399. Springer, 2004.
- [18] L. Schröder. The HASCASL prologue - categorical syntax and semantics of the partial λ -calculus. *Theoret. Comput. Sci.*, 353:1–25, 2006.
- [19] L. Schröder and T. Mossakowski. HASCASL: Towards integrated specification and development of functional programs. In *Algebraic Methodology and Software Technology*, vol. 2422 of *LNCS*, pp. 99–116. Springer, 2002.
- [20] L. Schröder and T. Mossakowski. Monad-independent Hoare logic in HASCASL. In *Fundamental Aspects of Software Engineering*, vol. 2621 of *LNCS*, pp. 261–277, 2003.
- [21] L. Schröder and T. Mossakowski. Generic exception handling and the Java monad. In *Algebraic Methodology and Software Technology*, vol. 3116 of *LNCS*, pp. 443–459. Springer, 2004.
- [22] L. Schröder and T. Mossakowski. Monad-independent dynamic logic in HASCASL. *J. Logic Comput.*, 14:571–619, 2004.
- [23] L. Schröder, T. Mossakowski, and C. Lüth. Type class polymorphism in an institutional framework. In *Recent Developments in Algebraic Development Techniques, 17th International Workshop, WADT 04*, vol. 3423 of *LNCS*, pp. 234–248. Springer, 2004.
- [24] L. Schröder, T. Mossakowski, and C. Maeder. HASCASL – Integrated functional specification and programming. Language summary. Available under http://www.informatik.uni-bremen.de/agbkb/forschung/formal_methods/CoFI/HasCASL
- [25] D. Walter, L. Schröder, and T. Mossakowski. Parametrized exceptions. In *Algebra and Coalgebra in Computer Science, CALCO 05*, vol. 3629 of *LNCS*, pp. 424–438. Springer, 2005.