

BOOTSTRAPPING TYPES AND COTYPES IN HASCASL

LUTZ SCHRÖDER

DFKI-Lab Bremen and Department of Computer Science, University of Bremen, Germany
e-mail address: Lutz.Schroeder@dfki.de

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. Moreover, we present similar constructions in categories of internal complete partial orders.

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 [26, 25]. 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 [12], 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

2000 ACM Subject Classification: D.2.1 [Software Engineering]: Requirements/Specifications — languages; E.1 [Data Structures]; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs — specification techniques; F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages — denotational semantics; F.4.1 [Mathematical Logic and Formal Languages]: Mathematical Logic — lambda calculus and related systems, recursive function theory. *General Terms:* Languages, Theory, Verification.

Key words and phrases: datatypes, process types, software specification, quasitoposes, partial λ -calculus. This work is an extended version of [24].

It forms part of the DFG-project HASCASL (KR 1191/7-2).

functions without unique choice which may, with a certain margin of error, be thought of as the internal logic of quasitoposes (more precisely, it is the internal logic of partial cartesian closed categories with equality [22, 23]).

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 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` [16, 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. Abstracting our results to the categorical level, we prove, in partial generalization of corresponding results for toposes [9], that any quasitopos (indeed, any partial cartesian closed category with equality and finite coproducts) with `NNO` supports initial algebras and final coalgebras for certain classes of polynomial functors. Moreover, we obtain corresponding results for datatypes and process types equipped with complete partial orders (the former are called free domains in `HASCASL`). These types serve as the correspondent of programming language datatypes in `HASCASL`'s internal modelling of denotational semantics.

The material is organised as follows. We recall some aspects of the syntax and semantics of `HASCASL`, including the relationship between `HASCASL`' Henkin models and categorical models, in Sect. 1. In Sect. 2, we expand on the semantics of generated and free datatypes in `HASCASL`. These two sections summarise material from [23, 25]. We then go on to present the bootstrapped specification of the syntax and semantics of signature functors and inductive datatypes using the type class mechanism in Sect. 3. In Sect. 4 we discuss how these concepts extend naturally to coinductive process types. We present the constructions establishing the conservativity of datatype and process type declarations in Sect. 5. Finally, we recall the modelling of general recursive programs by means of an adapted version of domain theory in Sect. 6, and show how the constructions of plain initial datatypes and final process types can be modified to obtain corresponding constructions on domains.

1. `HASCASL`

The wide-spectrum language `HASCASL` [26] 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 [27, 29, 28, 31]. Tool support for HASCASL is provided in the framework of the Bremen heterogeneous tool set Hets [11]. 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, 13] for a detailed language description. Below, we 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. [25] for a full language definition. Moreover, we recall HASCASL’s Henkin semantics and its relation to categorical models in quasitoposes and, more generally, partial cartesian closed categories with equality [23].

1.1. The internal logic of HASCASL. The logic of HASCASL is based on the partial λ -calculus [10]. It is distinguished from standard HOL by having intuitionistic truth values and partial function types $t \rightarrow? s$ (besides total function types $t \rightarrow s$); λ -abstractions $\lambda x : t \bullet \alpha$ denote partial functions, i.e. inhabitants of partial function types $t \rightarrow? s$, while total λ -abstractions, inhabiting the total function type $t \rightarrow s$, are denoted $\lambda x : s \bullet! \alpha$. There are moreover a unit type $Unit$, with unique inhabitant $()$, and product types $s \times t$. Predicates then arise as partial functions into $Unit$, where definedness is understood as satisfaction, and the type of truth values is $Logical = Unit \rightarrow? Unit$. We denote application of a function f to an argument x as $f x$. As in [23], we moreover denote by $\alpha \upharpoonright \phi$ the *restriction* of a term α to a formula ϕ , i.e. $\alpha \upharpoonright \phi$ is defined iff α is defined and ϕ holds, and in this case is equal to α (essentially, \upharpoonright is just the first projection).

In a partial setting, there are numerous readings of equality; here, we require *strong equality*, denoted $=$ and read ‘one side is defined iff the other is, and in this case, both sides are equal’, as well as *existential equality*, denoted $\stackrel{e}{=}$ and read ‘both sides are defined and equal’. Equality of terms in the partial λ -calculus is defined largely as expected [10, 23], with a few subtleties attached to partiality — e.g. β -equality $(\lambda x : t \bullet \alpha)\gamma = \alpha[\gamma/x]$ holds only if the term γ is defined. We assume that equality is *internal*, i.e. that there exists on every type a binary predicate representing existential equality, also written $\stackrel{e}{=}$; this defines the *partial λ -calculus with equality*. In the partial λ -calculus with equality, an intuitionistic predicate logic is defined in the standard way (see e.g. [22, 23]) by abbreviations such as

$$\forall x : t \bullet \phi = ((\lambda x : t \bullet \phi) \stackrel{e}{=} (\lambda x : a \bullet ())),$$

where $()$ is the unique inhabitant of $Unit$.

The difference between the HASCASL logic and the more familiar topos logic [8] is the absence of unique choice [23], 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** [14]). In HASCASL, the unique choice principle may be imposed if desired by means of a polymorphic axiom [25]. 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 rather than just in toposes (cf. Sect. 1.3). 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 [17, 21]. 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 from [14], Sec. 5.10:

“Description operators can be hard to reason about. Novices should try to avoid them. Fortunately, descriptions are seldom required.”

1.2. Type class polymorphism. 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* : *Functor*

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

var *a, b* : *Ord*
type *a* \times *b* : *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* [30]; 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 only 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.

1.3. Henkin Models and Partial Cartesian Closed Categories. The set-theoretic semantics of HASCASL 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 [23]; these categories are slightly more general than quasitoposes [1], which can be seen as finitely cocomplete pccc’s with equality. Below, we summarise some of the details of the categorical viewpoint; we refer to [23, 25] for the full definition of intensional Henkin models.

A *dominion* [20] on a category \mathbf{C} is a class \mathcal{M} of monomorphisms in \mathbf{C} which contains all identities and is closed under composition and pullback stable, the latter in the sense that pullbacks, or inverse images, of \mathcal{M} -morphisms along arbitrary morphisms exist and are in \mathcal{M} . The pair $(\mathbf{C}, \mathcal{M})$ is called a *dominional category*. A *partial morphism* $X \multimap Y$ in $(\mathbf{C}, \mathcal{M})$ is a span $X \xleftarrow{m} D \xrightarrow{f} Y$, where $m \in \mathcal{M}$. Partial morphisms (m, f) are composed by pullback formation. Intuitively, (m, f) is a partial map defined on the subobject D of X . The partial morphisms in $(\mathbf{C}, \mathcal{M})$ form a category $\mathbf{P}(\mathbf{C}, \mathcal{M})$, into which \mathbf{C} is embedded by mapping a morphism f to the partial morphism (id, f) . If \mathbf{C} is cartesian, i.e. has a terminal object 1 and binary products $A \times B$, then \mathbf{C} is a *partial cartesian closed category (pccc)* if the functor

$$\mathbf{C} \xrightarrow{- \times A} \mathbf{C} \hookrightarrow \mathbf{P}(\mathbf{C}, \mathcal{M})$$

has a right adjoint for each object A in \mathbf{C} . In case \mathcal{M} contains all diagonal morphisms $A \rightarrow A \times A$, the classes of extremal and regular monomorphisms in \mathbf{C} , respectively, and the class \mathcal{M} coincide. In this case, \mathbf{C} is called a *pccc with equality*.

It has been shown in [23] that one has an equivalence between theories in the partial λ -calculus with equality and pccc's with equality. Here, a *theory* consists of a set of basic types, from which composite types are obtained inductively by forming partial function types $s_1 \times \cdots \times s_n \multimap t$, a set of basic operations with assigned types, and a set of axioms, expressed as *existentially conditioned equations (ece's)* in this signature. Here, an existentially conditioned equation is a sentence of the form $\bigwedge_{i=1}^n \text{def } \alpha_i \Rightarrow \beta \stackrel{e}{=} \gamma$, where β , γ , and the α_i are terms formed from the basic operations, application, λ -abstraction, and typed variables from a given context, and $\text{def } \alpha$ abbreviates the formula $\alpha \stackrel{e}{=} \alpha$, which states that the term α is defined.

In the correspondence between categories and theories, one associates to every pccc with equality, \mathbf{C} , an *internal language* $\text{Th}(\mathbf{C})$ which has the objects of \mathbf{C} as basic types and the partial morphisms as operations, as well as all ece's expressed in this language which hold in \mathbf{C} as axioms. Conversely, one associates to every theory \mathcal{T} in the partial λ -calculus with equality a pccc with equality, $\text{Cl}(\mathcal{T})$, the *classifying category* of \mathcal{T} . The objects of $\text{Cl}(\mathcal{T})$ are pairs (Γ, ϕ) consisting of a finite *context* $\Gamma = (x_1 : s_1, \dots, x_n : s_n)$ of variables x_i with assigned types s_i and a formula (i.e. by the correspondence between predicates and partial functions discussed above just a definedness assertion) ϕ in context Γ . Morphisms $\sigma : (\Gamma, \phi) \rightarrow (\Delta, \psi)$ are (type-correct) substitutions of the variables in Δ by terms in context Γ such that ϕ entails $\psi\sigma$ as well as definedness of $\sigma(x)$ for every variable x in Δ ; morphisms are taken modulo provable equality of terms under ϕ .

The central facts establishing that the above correspondence is actually an equivalence are that

- the pccc \mathbf{C} is equivalent to $\text{Cl}(\text{Th}(\mathbf{C}))$, and
- the theory $\text{Th}(\text{Cl}(\mathcal{T}))$ is a conservative extension of \mathcal{T} .

When reasoning about a pccc with equality, \mathbf{C} , one may thus assume that \mathbf{C} is actually of the form $\text{Cl}(\text{Th}(\mathbf{C}))$, i.e. freely move back and forth between logical and categorical arguments, and in particular construct objects in \mathbf{C} as subtypes of types formed from \mathbf{C} -objects. The constructions of initial datatypes and final process types in Sect. 5 will be based on this principle. They will, by the above equivalence, amount simultaneously to conservativity results in HASCASL and to existence theorems for datatypes and process types in the categorical semantics. In the latter incarnation, they apply in particular to quasitoposes [32], which may be defined as finitely cocomplete pccc's with equality. This

class of categories is technically related to toposes, the essential difference being that the internal logic of a topos speaks about *all* subobjects, while that of a quasitopos speaks only about the extremal subobjects (or more formally that the classifier of a topos classifies all subobjects, and that of a quasitopos only the extremal subobjects).

As mentioned above, the range of examples is much broader in the case of quasitoposes; e.g. there are many interesting non-trivial concrete quasitoposes over **Set**, while concrete toposes over **Set** are always full subcategories of **Set**. Intuitively, quasitoposes support a distinction between ‘maps’, i.e. functional relations, and ‘morphisms’, i.e. functions, while the two concepts coincide in toposes. In the internal logic, the difference is captured precisely by the fact that toposes admit unique choice, while quasitoposes do not. Objects A in a quasitopos (or in a pccc with equality) that do admit unique choice in the sense described in Sect. 1.1 are called *coarse*. Explicitly, A is coarse iff there exists a function c from the type $Sg(A)$ of singleton subsets of A to A such that $c(p)$ is in p (and hence $p = \{c(p)\}$) for every $p : Sg(A)$; in this case, the unique description term $\iota x : A \bullet \phi$ can be defined as $c(\lambda x : A \bullet \phi)$.

To give the reader a basic feeling for the above issues, we recall one of the simplest examples of a non-trivial set-based quasitopos, the category **ReRe** of reflexive relations [1]. The objects of **ReRe** are pairs (X, R) with R a reflexive relation on the set X , and morphisms $f : (X, R) \rightarrow (Y, S)$ are relation-preserving maps $f : X \rightarrow Y$, i.e. $f(x)Sf(y)$ whenever xRy . We say that (X, R) is *discrete* if R is equality, and *indiscrete* if $R = X \times X$. The coarse objects of **ReRe** are precisely the indiscrete objects. The category **ReRe** has a natural numbers object, i.e. an initial algebra for the functor $- + 1$, namely the discrete structure on the set of natural numbers.

2. DATATYPES IN HASCASL

HASCASL supports recursive datatypes in the same style as in CASL [3, 13]. 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; their handling requires essentially no more than adding more indices). 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 , the type variables a_i , and any types declared in the *local environment*, i.e. the context of preceding declarations. 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 $t = C \ a_1 \ \dots \ a_r$ 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 t are required to be strictly positive w.r.t. function arrows, i.e. occurrences in the argument type of a function type are forbidden. We omit a detailed discussion of generatedness constraints [25]. The semantics of freeness constraints is defined in more detail as follows.

Standardly, initial datatypes are characterised by the abovementioned induction axioms (*no junk*) and additionally by the *no confusion* condition, stating essentially that all terms formed from the constructors and given elements of the types in the local environment denote distinct values. By the discussion in Sect. 1.3, it is clear that these conditions are insufficient in the setting of HASCASL's internal logic: in the maps vs. morphisms metaphor, they constrain only the underlying set of a datatype, not its structure. E.g. in the quasitopos **ReRe** of reflexive relations, the no-junk-no-confusion axioms for the datatype of natural numbers, i.e. the Peano axioms, will be satisfied by any object whose underlying set is the set of natural numbers. In particular, one will not be able to prove a recursion principle from the Peano axioms (which is possible under unique choice [15]), as models of the Peano axioms in general fail to be initial algebras.

As mentioned above, the semantics of free datatypes in HASCASL is therefore determined by a fold operator, i.e. free datatypes are explicitly axiomatised 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 may thus be 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

$$d \circ (F f) = f \circ \alpha.$$

Initiality implies induction and term distinctness, i.e. the usual no-junk/no-confusion conditions: term distinctness follows from the fact that structure maps of initial algebras are isomorphisms (Lambek's lemma); induction for a predicate P on t is proved by applying fold at the type $b = (x : t. P x)$. (The semantics of polymorphism in HASCASL precribes that polymorphic operators such as fold do have instances at subtypes [25]. For polynomial functors, the use of such instances can be circumvented; cf. Remark 5.4 for more comments on this point.) Moreover, one obtains a *primitive recursion* principle by means of a simultaneous recursive definition of the identity (as suggested in [5]): The fold operation allows

defining recursive functions $f : t \rightarrow b$, where $\alpha : F t \rightarrow t$ is the initial datatype for the functor F , using the iteration scheme, i.e.

$$f (\alpha x) = d (F f x) \quad \text{for } x : F t$$

(which is just a restatement of the previous equation). One may thus in particular define a function $g : t \rightarrow t \times b$ by

$$g (\alpha x) = (\lambda y : F(t \times b) \bullet (\alpha (F \pi_1 y), d y)) (F g x)$$

where π_1 denotes the first projection $\lambda(x, y) : F(t \times b) \bullet x$. Here, the actual body of the recursive definition is the map $d : F(t \times b) \rightarrow b$. One proves by induction that the first component of g is the identity on t ; the second component $f = \pi_2 \circ g$, where $\pi_2 : F(t \times b) \rightarrow b$ denotes the second projection, satisfies

$$f (\alpha x) = d (F(\lambda y : t \bullet (y, f y)) x),$$

and conversely every solution f of this equation yields a solution $g = \lambda y : t \bullet (y, f y)$ of the iteration equation for g . Thus we may define $f : t \rightarrow b$ by *primitive recursive equations*, whose right hand side may depend on applications $f x$ of f to constructor arguments x appearing in the pattern αx on the left hand side, as in the case of iteration, and additionally on the constructor arguments x themselves.

Since by Lambek's lemma, the structure map of an initial algebra is an isomorphism, free datatypes $\alpha : F t \rightarrow t$ inherit a case operator from the decomposition of $F t$ as a sum; such an operator

$$\text{case } x \text{ of } c_1 y_{11} \dots y_{1k_1} \rightarrow f_1 y_{11} \dots y_{1k_1} \mid \dots \mid c_l y_{l1} \dots y_{lk_l} \rightarrow f_l y_{l1} \dots y_{lk_l}$$

is provided explicitly in HASCASL.

Remark 2.1. 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.

Example 2.2. Consider the following free datatype definitions.

free type $List\ a ::= nil \mid cons\ (a; List\ a)$

free type $Tree\ a\ b ::= leaf\ a \mid branch\ (b \rightarrow List\ (Tree\ a\ b))$

The declaration of $List\ a$ induces the standard fold operation for lists. Moreover, the type class mechanism (cf. Sect. 3) recognises 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

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

This operation is axiomatised as being uniquely determined by the equations

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

```

spec FUNCTOR =
  vars    $a, b, c : \text{Type}; x : a; f : a \rightarrow b; g : b \rightarrow c$ 
  ops    $\_comp\_ : (b \rightarrow c) \times (a \rightarrow b) \rightarrow a \rightarrow c;$ 
          $id : a \rightarrow a$ 
  •  $id\ x = x$ 
  •  $(g\ comp\ f)\ x = g\ (f\ x)$ 
  class  $Functor < Type \rightarrow Type$ 
  {vars   $a, b, c : Type; F : Functor; f : a \rightarrow b; g : b \rightarrow c$ 
  op     $map : (a \rightarrow b) \rightarrow F\ a \rightarrow F\ b$ 
  •  $map\ id = id : F\ a \rightarrow F\ a$ 
  •  $map\ (g\ comp\ f) : F\ a \rightarrow F\ c = (map\ g)\ comp\ (map\ f)$ 
  }
  class  $Bifunctor < Type \rightarrow Functor$ 
  {vars   $a, b, c, d : Type; F : Bifunctor; f : a \rightarrow b; g : b \rightarrow c$ 
  op     $parmap : (a \rightarrow b) \rightarrow F\ a\ d \rightarrow F\ b\ d$ 
  •  $parmap\ id = id : F\ a\ d \rightarrow F\ a\ d$ 
  •  $parmap\ (g\ comp\ f) : F\ a\ d \rightarrow F\ c\ d = (parmap\ g)\ comp\ (parmap\ f);$ 
  •  $(parmap\ f)\ comp\ (map\ h) : F\ a\ c \rightarrow F\ b\ d = (map\ h)\ comp\ (parmap\ f)$ 
  }

```

Figure 1: HASCASL specification of functors

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 algebra homomorphisms.

Figure 1 shows the constructor class of functors. Mutually recursive or parametrised 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 3.1. 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 $Nil : \text{Typelist}$, $Cons : \text{Type} \rightarrow \text{Typelist} \rightarrow \text{Typelist}$, and define *Functor* as a subclass of $\text{Typelist} \rightarrow \text{Type}$. (Undesired semantic side effects may be eliminated by specifying the types *Nil*, $Cons\ a\ Nil$ etc. to be singletons.) Similar tricks work in Haskell [7] 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. These operations, and similar constructions in

```

spec POLYFUNCTORS =
  FUNCTOR
then classes PolyFunctor < Functor;
               PolyBifunctor < Type → PolyFunctor;
               PolyBifunctor < Bifunctor
vars F, G : PolyFunctor; H, K : PolyBifunctor; a, b, c : Type
type Sum b c ::= inl b | inr c
vars f : b → a; g : c → a; h : Sum b c → a
op sumcase : (b → a) → (c → a) → Sum b c → a
• h = sumcase f g
  ⇔ ∀ x : b; y : c • h (inl x) = f x ∧ h (inr y) = g y;
types Fst a b := a;
        Snd a b := b;
        Id b := b;
        ProdF F G b := F b × G b;
        ProdBF H K b c := H b c × K b c;
        SumF F G b := Sum (F b) (G b);
        SumBF H K b c := Sum (H b c) (K b c)
types Fst, Id, ProdF F G, SumF F G : PolyFunctor;
        Fst, Snd, ProdBF H K, SumBF H K : PolyBifunctor
var k : a → b
• (map k : SumF F G a → SumF F G b)
  = sumcase (inl comp map k) (inr comp map k)
• (parmap k : SumBF H K a c → SumBF H K b c)
  = sumcase (inl comp parmap k) (inr comp parmap k)
• ... %% definitions of map and parmap for the other cases

```

Figure 2: HASCASL specification of polynomial functors

Fig. 3 below, are defined as *type synonyms*, i.e. as mere abbreviations of existing types.¹ The obvious definitions of the *map* and *parmap* operations are omitted for most of the functors introduced in Fig. 2, 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. As in Fig. 1, we show only the specification for functors of arity at most 2; the extension to higher arities is obvious.

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, thus circumventing the absence of dependent types — such as

¹Consequently, the specifications, while correct according to the HASCASL language definition, fail to pass the static analysis in the present version of the heterogeneous tool set Hets [11], as type synonyms are currently immediately expanded and β -reduced; this will be remedied in future versions of the tool. In [24], we have used type declarations with explicit constructors as a workaround in place of type synonyms; for purposes of the present work, we have given preference to readability of specifications.

the ‘type’ of algebra morphisms between algebras *alpha* and *beta* — in HASCASL (these can be conservatively added to the language [22], however at the price of making type checking undecidable).

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, called *ifold* in the specification. As initial algebras will exist only for some functors, the abovementioned operations are defined only on a subclass *DTFunction* (‘datatype functor’) of *Function*. We declare the class *PolyFunction* (Fig. 2) to be a subclass of *DTFunction*, thus stating that all polynomial functors have initial algebras as proved in Sect. 5; due to possible junk in the class *PolyFunction* (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 [6]) by defining a type constructor *ParamDT* which maps a polynomial bifunctor *G* to the functor that takes a type *a* to the initial algebra of the polynomial functor *G a*, and by declaring *ParamDT G* to be an instance of *DTFunction*.

Remark 3.2. Note that functors induced by parametrised initial datatype declarations are declared as instance of *DTFunction* in Fig. 3 only if the signature functor is polynomial. It is not possible to extend this mechanism to arbitrary parametrised datatypes, as the corresponding functors need not themselves have initial algebras. As a simple example, consider the declaration

```

var    a : Type
free type C a ::= abs (Pred a)
    
```

which defines *C a* as the initial algebra of the functor *F* given by $F a b = \text{Pred } a$, i.e. *F* takes the powerset of its first argument and ignores the second. Thus, *C a* is isomorphic to *Pred a* and hence, by Russel’s paradox, the functor *C* does not have an initial algebra.

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, $t = C a_1 \dots a_n$, and the type variables *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 *Function* are generated automatically for parametrised datatypes such as the type *List a* of Example 2.2. Given this format of the *t_{ij}*, it is straightforward to associate a functor to each *t_{ij}* (using further generic instances of *Function*, in particular exponentials and closure under functor composition). Finally, an instance $F : \text{DTFunction}$ is generated. If this instance is already induced by the generic instances shown in Fig. 3, then the datatype declaration is guaranteed to be conservative (cf. Sec. 5); otherwise, conservativity and in fact consistency of the datatype

```

spec ALGEBRA = POLYFUNCTORS
then vars   $F : Functor; a, b : Type$ 
type   $Alg\ F\ a := F\ a \rightarrow a$ 
op     $--::--\rightarrow-- : Pred\ ((a \rightarrow b) \times (Alg\ F\ a) \times (Alg\ F\ b))$ 
vars   $f : a \rightarrow b; alpha : Alg\ F\ a; beta : Alg\ F\ b$ 
•  $(f :: alpha \rightarrow beta) \Leftrightarrow (beta\ comp\ (map\ f)) = (f\ comp\ alpha)$ 
type   $AlgMor\ F\ a\ b = \{(f, alpha, beta) : (a \rightarrow b) \times Alg\ F\ a \times Alg\ F\ b \bullet$ 
•  $f :: alpha \rightarrow beta \}$ 

classes  $DTFuncor < Functor; PolyFuncor < DTFuncor$ 
{vars   $F : DTFuncor; a : Type$ 
type   $InitialCarrier\ F$ 
ops    $initialAlg : Alg\ F\ (InitialCarrier\ F);$ 
         $ifold : Alg\ F\ a \rightarrow InitialCarrier\ F \rightarrow a$ 
vars   $alpha : Alg\ F\ a; g : InitialCarrier\ F \rightarrow a;$ 
•  $(g :: initialAlg \rightarrow alpha) \Leftrightarrow g = ifold\ alpha;$ 
}
var    $G : PolyBifuncor$ 
type   $ParamDT\ G\ a := InitialCarrier\ (G\ a)$ 
type   $ParamDT\ G : DTFuncor$ 
vars   $l : ParamDT\ G\ a; b : Type; f : a \rightarrow b$ 
•  $map\ f\ l = ifold\ (initialAlg\ comp\ parmap\ f)\ l$ 

```

Figure 3: HASCASL specification of initial algebras

declaration becomes the responsibility of the user. This happens in particular when constructor arguments involve either type constructors from the local environment which are not themselves declared as initial datatypes or exponentiation with types from the local environment. Whether or not datatype declarations are conservative in the latter case, which in particular includes the case of *infinite branching*, remains an open problem; under unique choice, declarations of infinitely branching datatypes are conservative [9, 16, 2]. If F is moreover of the class *PolyBifuncor* (or a corresponding class of functors of higher arity), then an instance $C : DTFuncor$ is generated.

Using the *sumcase* operation of Fig. 2, one can gather the constructors of t into a structure map $c : F\ t \rightarrow t$; the freeness constraint 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* [12], 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. Thus, the core of a cotype declaration has the form

$$\mathbf{cotype}\ t ::= (sel_{11} : t_{11}; \dots; sel_{1k_1} : t_{1k_1}) \mid \dots \mid (sel_{n1} : t_{n1}; \dots; sel_{nk_n} : t_{nk_n}),$$

where t is a pattern $C a_1 \dots a_n$ consisting of a newly declared type constructor C and type variables a_1, \dots, a_n . A cotype induces axioms guaranteeing that the domains of selectors in the same alternative agree, and that the domains of all alternatives form a coproduct decomposition of the cotype. Thus, e.g. the models of the cotype

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

are coalgebras for the functor F given by $FX = a \times X + X \times X$. 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 $\textit{Coalg } F a := a \rightarrow F a$, 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. For cofree cotypes, the codomains of the selectors are, as in the case of initial datatype, required to depend functorially on t ; of course, this will not in general guarantee existence of final coalgebras. The extraction of functors from cotype signatures is analogous to the case of types as explained in Sect. 3, with two differences:

- the class of functors that admit final coalgebras contains a generalised class of polynomial functors that allows replacing identity functors by exponentiation with constant exponents (cf. Sect. 5);
- unlike for free types, functors arising from cofree cotype declarations, even for polynomial functors, typically do not have final coalgebras.

We omit the discussion of cogeneratedness of cotypes.

The only subtle point in the matching between cotype declarations in HASCASL and coalgebras is that the conditions imposed in COCASL to ensure that a cotype t with associated functor F decomposes as a disjoint union of the domains of the selectors would in HASCASL be insufficient to guarantee existence of a single structure map $t \rightarrow F t$, the point being, again, the absence of unique choice. As indicated above, 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 $f, g : Proc \rightarrow? a$, $\textit{case } f \textit{ } g = h : Proc \rightarrow a$ 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, $\textit{case } f \textit{ } g$ is definable as $\lambda p : Proc \bullet \iota x : a \bullet x = f(p) \vee x = g(p)$.)

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 [16, 2] is based on the naturals. The constructions given in *loc. cit.* make heavy use of unique choice, so that the question arises whether similar constructions are possible in HASCASL. Below, we answer this question in the affirmative for the case of finitely branching datatypes; it remains open for the infinitely branching case. By the equivalence of HASCASL with the internal logic of partial cartesian closed categories with equality (Sect. 1.3), our results extend to pccc's with equality and finite coproducts, and hence in particular to quasitoposes.

To begin, we fix the required additional infrastructure. As seen above, already the construction of signature functors for standard datatypes requires finite sums. These are specified in HASCASL by declaring a type constructor Sum as in Fig. 2, and moreover by making the subtype $0 = (x : Unit. \perp)$ of $Unit$ an initial type. The latter is achieved by declaring a polymorphic constant

$$bot : Unit \rightarrow? a$$

for all types a , along with the axiom $\neg def\ bot\ ()$. Below, we denote sums by $+$ in the interest of readability, with injection functions inl and inr as usual; moreover, we use the standard *case* notation as discussed in the case of initial datatypes in Sec. 2, and denote the unit type $Unit$ by 1 .

The declaration of Sum in Fig. 2 is non-conservative. One can however reduce the existence of sums to the existence of a type $Bool = 1 + 1$ of booleans; in particular, one needs only a type constant $Bool$, not a type constructor Sum . As usual, we denote the case operator on $Bool = 1 + 1$ as *if – then – else*, and the terms $inl\ ()$ and $inr\ ()$ as *true* and *false*, respectively. By the results summarised in Sect. 1.3, we may phrase this as a result on pccs with equality:

Proposition 5.1. *A pcc with equality, \mathbf{C} , has finite coproducts (sums) iff the sum $Bool = 1 + 1$ exists in \mathbf{C} .*

Proof. Given $Bool = 1 + 1$, one can construct the sum $a + b$ of objects a, b in \mathbf{C} as

$$(x : 1 \rightarrow? a, y : 1 \rightarrow? b, z : Bool. (def\ (x\ ()) \Leftrightarrow z = true) \wedge (def\ (y\ ()) \Leftrightarrow z = false))$$

with injections $inl\ x = (\lambda z : 1 \bullet x, \lambda z : 1 \bullet bot, true)$, and $inr\ y = (\lambda z : 1 \bullet bot, \lambda z : 1 \bullet y, false)$. The copairing $h = sumcase\ f\ g$ of functions $f : a \rightarrow c$, $g : b \rightarrow c$ is then defined as

$$h\ (x, y, z) = if\ z\ then\ f\ (x\ ())\ else\ g\ (y\ ()).$$

It is easy to see that the copairing is uniquely determined. \square

Remark 5.2. Under unique choice, $Bool = 1 + 1$ coincides with the type

$$(p : Logical. p \vee \neg p),$$

with injections $inl\ () = \top$ and $inr\ () = \perp$. The copairing $h = sumcase\ f\ g$ of two functions $f, g : 1 \rightarrow a$ is then defined as $h\ p = \iota x : a \bullet (p \Rightarrow f\ () = x) \wedge (\neg p \Rightarrow g\ () = x)$. By Proposition 5.1, this reproves the well-known fact that toposes have finite coproducts. In a quasitopos, one cannot in general construct $Bool$ as subtype of $Logical$ — the latter is typically an indiscrete space, while $Bool$ is typically discrete. E.g. in the quasitopos of reflexive relations, $Logical$ carries the universal relation, while $Bool$ carries the equality relation.

Remark 5.3. In a cartesian closed category, finite coproducts are always internal in the sense that copairing is embodied in an operation (*sumcase* in the above notation) which satisfies the relevant laws (cf. Fig. 2) internally (cf. e.g. [18]). Thus, the internal language of a pcc with equality and finite coproducts has sum types as specified in Fig. 2. Conversely, the classifying category of a theory with sum types as in Fig. 2 has finite coproducts: one has to check that the binary coproduct of subtypes $(x : a. \phi)$, $(y : b. \psi)$ exists (even when one does not assume an instance of Sum at these subtypes); but this is just the subtype

$$(z : a + b. case\ z\ of\ inl\ x \rightarrow \phi \mid inr\ y \rightarrow \psi).$$

In combination with Proposition 5.1, it follows that

the partial λ -calculus with equality, bot, and Bool is the internal logic of pccc's with equality and finite coproducts,

where we understand *bot* and *Bool* as being equipped with the operations and axioms discussed above, in particular *if – then – else*.

The search for the internal logic of quasitoposes, i.e. a logic that would be equivalent to quasitoposes via an internal language/classifying category correspondence, remains open. Recall that a quasitopos is a pccc with equality and finite colimits, i.e. finite coproducts and coequalisers. Hence, the missing ingredient is a suitable logical representation of coequalisers, i.e. quotients. We conjecture that the key to this is a generalisation from subtypes $(x : a. \phi)$ to subtypes with replacement, i.e. types of the form $(f(x). x : a; \phi)$, representing the quotient of $(x : a. \phi)$ by the kernel of a function $f : a \rightarrow b$.

As indicated above, we shall also need the standard notion of natural numbers object (NNO). Categorically, an NNO is an initial algebra for the functor $- + 1$; in HASCASL, a corresponding type of natural numbers is specified as

free type $Nat ::= 0 \mid suc\ Nat$

Remark 5.4. In a cartesian closed category, every NNO is internal in the sense that the unique existence of an algebra morphism from the NNO into a given $- + 1$ -algebra holds as a formula of the internal logic and is embodied by an operation, *fold* in the above notation [18]; the same holds for initial algebras, and dually for final coalgebras, of arbitrary strong functors, in particular polynomial functors. The explicit distinction of internal NNOs used in [24] is thus superfluous. It follows that the internal language of a pccc with equality, sums, and NNO always has a type *Nat* as specified above. Conversely, the classifying category of a theory containing sums (i.e. *bot* and *Bool*) and the type *Nat* will have $(n : Nat. \top)$ as an NNO. To see this, one has to show that the fold operation applies also to $- + 1$ -algebras on subtypes $(x : a. \phi)$. This can be proved without resorting to instances of *fold* at such subtypes, as indicated in Sect. 2; as announced there, the argument presented in the following is general enough to apply to arbitrary polynomial functors. The relevance of this point is that assuming instances of *fold* at subtypes essentially amounts to postulating induction as a separate axiom, rather than deriving it from recursion.

To begin, note that the induction principle on *Nat* may be proved using *fold* only at the type *Logical*: given a predicate P on *Nat* such that $P\ 0$ and $\forall n : Nat \bullet P\ n \Rightarrow P\ (suc\ n)$, define a predicate Q on *Nat* recursively by

$$\begin{aligned} Q\ 0 &= P\ 0 \\ Q\ (suc\ n) &= Q\ n \wedge P\ (suc\ n). \end{aligned}$$

Then Q has the defining property of *fold g*, where $g : Logical + 1 \rightarrow Logical$ is the copairing of the identity and \top . As the constantly true predicate on *Nat* also has this property, it follows that Q and hence P hold universally.

Then, a morphism $d : (x : a. \phi) + 1 \rightarrow (x : a. \phi)$ induces in the obvious way a morphism $d' : ((1 \rightarrow? a) + 1) \rightarrow (1 \rightarrow? a)$. One thus obtains $f = fold\ d' : Nat \rightarrow (1 \rightarrow? a)$. It remains to show that f factors through $(x : a. \phi)$, i.e. that $f\ n$ is defined and satisfies $\phi[f\ n/x]$ for all n ; this is proved by induction.

We have thus established that

the partial λ -calculus with equality, bot, Bool, and Nat is the internal language of pcccs with equality, finite coproducts, and NNO.

We shall now prove the existence of initial algebras and final coalgebras for certain classes of functors the categorical semantics, thus partially generalizing known results for W -types in toposes (e.g. [9]).

Definition 5.5. The class of *polynomial* functors is inductively generated from the identity functor and constant functors by taking finite sums and products. The class of *extended polynomial functors* is inductively generated from the exponential functors with constant exponent (including the identity functor by taking exponent 1) and constant functors by taking finite sums and products.

Of course, the intended interpretation of the constructor class *PolyFunctor* from Sect. 3 is the class of polynomial functors, and correspondingly for the more general constructor class appearing in the semantics of cotypes (Sect. 4) and the class of extended polynomial functors.

Theorem 5.6. *Let \mathbf{C} be a pccc with equality, finite coproducts, and NNO (e.g. a quasitopos with NNO). Then*

- (a) \mathbf{C} has initial algebras for polynomial functors;
- (b) \mathbf{C} has final coalgebras for extended polynomial functors.

The constructions employed in the proof are essentially subtype definitions in the internal logic. By the discussion in Sect. 1.2 and 1.3, it follows, that, as an extension of the specification of sums and the natural numbers, the declaration of a datatype $t = C a_1 \dots a_n$ with constructor arguments t_{ij} as in the beginning of Sect. 2 is conservative, provided that the t_{ij} are built from t , the a_i , and types from the local environment using only product and sum type formation. Moreover, the declaration of a final process type $t = C a_1 \dots a_n$ as in Sect. 4 is conservative as an extension of the specification of sums and the natural numbers, provided that the codomains t_{ij} of the selectors are built from t , the a_i , and types from the local environment using only product and sum type formation and exponentiation with exponents not depending on t . Since the class of pccc's with equality, finite coproducts, and NNO is stable under taking products of categories, Theorem 5.6 implies moreover that analogous results hold for declarations of several mutually recursive types or cotypes, respectively.

We begin by proving the existence of a particular datatype, the type of lists:

Lemma 5.7. *Let \mathbf{C} be a pccc with equality, finite coproducts, and NNO. Then \mathbf{C} has list objects, i.e. for every object A , the functor $1 + A \times _$ has an initial algebra.*

Proof. Put

$$List = (l : Nat \rightarrow? A, n : Nat. \forall m : Nat \bullet \text{def } (l\ m) \Leftrightarrow m < n)$$

(where $<$ is defined recursively) and define operations $nil : PList$ and $cons : A \times PList \rightarrow PList$ by

$$\begin{aligned} nil &= (\lambda m \bullet bot, 0) \\ cons(x, (l, n)) &= (\lambda k \bullet \text{case } k \text{ of } 0 \rightarrow x \mid suc\ r \rightarrow l\ r, suc\ n). \end{aligned}$$

Then the copairing of nil and $cons$ is a $1 + A \times _$ -algebra structure on $List$. Given another $1 + A \times _$ -algebra on an object B , given by operations $e : B$, $f : A \times B \rightarrow B$, we define

$g = \text{fold } e \ b : \text{List} \rightarrow B$ by recursion over Nat :

$$\begin{aligned} g(l, 0) &= e \\ g(l, \text{suc } n) &= f(l, 0) (g((\lambda k : \text{Nat} \bullet l(\text{suc } k)), n)). \end{aligned}$$

It is easy to check that g satisfies, and is uniquely determined by, the defining equation for $\text{fold } e \ b$. Moreover, as recursion on natural numbers is embodied as an operation, so is the recursion principle on lists. \square

Proof of Theorem 5.6. By Remark 5.4, we can conduct the proof in the classifying category of the internal language of \mathbf{C} , the latter being a partial λ -theory with equality, *bot*, finite sums, and NNO.

(a): We can assume that the given functor F is of the normal form $FX = \sum_{i=1}^n A_i \times X^{k_i}$ with $k_i \in \mathbb{N}$ and constant parameter objects A_i . Let $A = \sum_{i=1}^n (A_i + 1)$, with outer coproduct injections in_i , and let Path be the type of lists of natural numbers (which exists in \mathbf{C} according to Lemma 5.7). We now define a universal type of trees, from which the desired initial algebra will be carved out as a subtype, by

$$DTree = (l : \text{Path} \rightarrow? A; d : \text{Path} \rightarrow? \text{Nat}),$$

where for $(l, d) : DTree$ and $p : \text{Path}$, $l \ p = \text{in}_i \ x$ indicates that the subtree at p is either a leaf labelled y , if $x = \text{inl } y$, or a node labelled by the i -th constructor, if $x = \text{inr } ()$, and $d \ p = n$ indicates that the subtree at p has depth n . We put $\text{depth}(l, d) = d \ \text{nil}$ for $(l, d) : DTree$, and for $j : \text{Nat}$, $j > 0$, we define a generic j -th selector by $\text{sel}_j(l, d) = (l \circ (\text{cons } j), d \circ (\text{cons } j))$. We define constructors $c_i : A_i \times DTree^{k_i} \rightarrow DTree$ by

$$c_i(x, (l_1, d_1), \dots, (l_{k_i}, d_{k_i})) = (l, d)$$

where l and d are defined by case distinction as

$$\begin{aligned} l \ \text{nil} &= \text{in}_i(\text{inr } ()) & l(\text{cons } j \ p) &= (\text{if } j = 0 \ \text{then } \text{in}_i(\text{inl } x) \ \text{else } l_j \ p) \\ d \ \text{nil} &= 1 + \max(d_1 \ \text{nil}, \dots, d_{k_i} \ \text{nil}) & d(\text{cons } j \ p) &= (\text{if } j = 0 \ \text{then } 0 \ \text{else } d_j \ p). \end{aligned}$$

(Here, access to the l_j and the d_j is by long case distinctions over j , with l_j and d_j undefined for $j > k_i$, the maximum is defined by recursion on the naturals, with $\max() = 0$, and the *if* expressions abbreviate obvious case expressions.) We then take the carrier T of the desired initial algebra to be the smallest subtype of $DTree$ closed under the c_i . Note that for all $(l, d) : T$, $\text{depth}(l, d) > 0$ and $l \ \text{nil} = \text{in}_i(\text{inr } ())$ for some i . We then have $l(\text{cons } 0 \ \text{nil}) = \text{in}_i(\text{inl } x)$ for some $x : A_i$, which we can access via a partial extraction function $\text{leaf}_i : DTree \rightarrow? A_i$.

It remains to be shown that we can construct the function $\text{fold } b_1 \ \dots \ b_n : T \rightarrow B$ for functions b_i representing an algebra on a type B . We define a primitive recursive function $f : \text{Nat} \rightarrow T \rightarrow? B$ by

$$\begin{aligned} f \ 0 \ (l, d) &= \text{bot} \\ f(\text{suc } n) \ (l, d) &= \text{case } l \ \text{nil} \ \text{of} \\ &\quad (\text{in}_i(\text{inr } ())) \rightarrow \\ &\quad b_i(\text{leaf}_i(l, d), f \ n \ (\text{sel}_1(l, d)), \dots, f \ n \ (\text{sel}_{k_i}(l, d)))_{i=1, \dots, n}, \end{aligned}$$

and put

$$\text{fold } b_1 \ \dots \ b_n \ z = f(\text{depth } z) \ z.$$

One verifies directly from the defining equations for f that this definition satisfies the fold equation. Moreover, one shows using the definition of Z as the least subtype of $DTree$ closed under the constructors that $fold\ b_1 \dots b_n$ is total, and uniquely determined by the fold equation.

(b): We can assume the the given functor F has the normal form $FX = \sum_{i=1}^n A_i \times (B_i \rightarrow X)$, with constant parameter objects A_i, B_i : it is easy to see that the class of functors isomorphic to such normal forms contains all exponential functors and all constant functors (noting that parameter objects can also be 1 or 0) and is closed under sums; to see closure under products, note that a product of two such normal forms is a sum of summands of the form $(A \times (B \rightarrow X)) \times (A' \times (B' \rightarrow X))$. Such a summand is isomorphic to $(A \times A') \times ((B + B') \rightarrow X)$.

Now put $A = \sum_{i=1}^n A_i$ and $B = \sum_{i=1}^n B_i$, with injections in_i in both cases. Define $Path$ as the type of lists over B , and equip it with the standard $snoc$ operation $Path \times Nat \rightarrow Path$. The universal type of infinite trees is

$$PTree = Path \rightarrow? A$$

(where it is crucial that we omit the depth component present in the universal type $DTree$ for initial datatypes). For $f : PTree$ and $p : Path$, the intended reading of $f\ p = in_i\ x$ is that position p in the tree behaves according to the i -th alternative and outputs $x : A_i$. The carrier of the final F -coalgebra is then the subtype C of $PTree$ consisting of those f such that

$$def\ (f\ nil) \quad \text{and} \quad (5.1)$$

$$def\ (f\ (snoc\ p\ (in_i\ y))) \iff \exists x : A_i \bullet f\ p = in_i\ x \quad (5.2)$$

for all $i = 1, \dots, n$. (Note that $f\ p = in_i\ x$ entails that $f\ p$ is defined.) We then define a F -coalgebra structure $c : C \rightarrow \sum_{i=1}^n (A_i \times (B_i \rightarrow C))$, with injections again denoted in_i , by

$$c\ f = case\ f\ nil\ of\ (in_i\ x \rightarrow in_i\ (x, \lambda y : B_i; p : Path \bullet f\ (cons\ (in_i\ y)\ p)))_{i=1, \dots, n}$$

Given a further F -coalgebra $d : D \rightarrow \sum_{i=1}^n (A_i \times (B_i \rightarrow D))$, we define the morphism $u = unfold\ d : D \rightarrow C$ recursively by

$$\begin{aligned} u\ z\ nil &= case\ d\ z\ of\ (in_i\ (x, g) \rightarrow in_i\ x)_{i=1, \dots, n} \\ u\ z\ (cons\ (in_i\ y)\ p) &= case\ d\ z\ of\ in_i\ (x, g) \rightarrow u\ (g\ y)\ p \end{aligned}$$

where omitted cases in the second case statement are understood to be *bot*. Since primitive recursion on lists is given by an operator, the above definition can be expressed as a term defining $unfold\ d$ and indeed $unfold$ as a function. It is immediate that $u\ z$ satisfies (5.1); one proves by induction over p that $u\ z$ satisfies also (5.2) and therefore indeed belongs to C . One verifies directly that u satisfies the defining equation for $unfold\ d$. Finally, one shows by induction over $Path$ that u is uniquely determined by the unfold equation. \square

Remark 5.8. The crucial difference between the above proof and the constructions of [16, 2] is the definition of the universal types as partial function spaces rather than types of sets of nodes, reflecting the fact that functional relations need not be functions in the absence of unique choice. Moreover, the construction of primitive recursive functions can no longer rely on an inductive construction of their graphs. It is an open problem whether our use of the depth function for this purpose in the case of initial datatypes can be generalised so as to cover also infinitely branching datatypes such as the type $Tree\ a\ b$ from Example 2.2.

6. DOMAINS

The treatment of general recursion in HASCASL is based on a HOLCF-style [19] internal representation of domains, phrased in terms of chain-complete partial orders. Some adaptations to this theory are necessary in order to cope with the absence of unique choice [25]. We briefly recall the relevant definitions and results below, and then go on to discuss the existence of initial datatypes in the category of domains. As already in the case of datatypes, we work in the internal language of a pccc with equality, sums, and NNO.

The main difficulty is that without unique choice, we can no longer e.g. define the value at x of the supremum of a chain of partial functions f_i as ‘the value (if any) eventually assumed by the $f_i(x)$ ’. Hence the modified definition

Definition 6.1. A partial order A with ordering \sqsubseteq is called a *complete partial order (cpo)* if the type $1 \rightarrow? A$, equipped with the ordering

$$x \sqsubseteq y \iff (\text{def } x () \Rightarrow x () \sqsubseteq y ()),$$

has suprema of chains, denoted by \bigsqcup , and a bottom element \perp . We call chains in $1 \rightarrow? A$ *partial chains*. We say that a cpo A is *pointed* (or a *cpo*) if A has a bottom element. We say that A is a *flat cpo* if A is a cpo when equipped with the discrete ordering. A partial function between cpo’s is *continuous* iff it preserves suprema of partial chains. The types of total and partial continuous functions from A to B are denoted $A \xrightarrow{c} B$ and $A \xrightarrow{c} ? B$, respectively.

Lemma 6.2. *Let (x_i) be a partial chain. Then $\bigsqcup_i x_i$ is defined iff $\exists n \bullet \text{def } x_n$.*

Cpo’s can be specified as a class in HASCASL; this is carried out in detail in [25]. While under unique choice, all types can be made into flat cpos, this need not be the case without unique choice. Cppo’s in the above sense have least fixed points of continuous endofunctions f , constructed as suprema of (total) chains $(f^n \perp)$; this is the basis of the interpretation of general recursive functions. Cpo’s are closed under the usual type constructors:

Proposition 6.3. *Let A and B be cpo’s. Then $A \times B$, equipped with the componentwise ordering, is a cpo.*

Proposition 6.4. *Let A and B be cpos, and let C be a type. Then the types $C \rightarrow B$, $C \rightarrow ? B$, $A \xrightarrow{c} B$, and $A \xrightarrow{c} ? B$ are cpo’s when equipped with the componentwise ordering; $C \rightarrow ? B$ and $A \xrightarrow{c} ? B$ are moreover pointed.*

Proposition 6.5. *The unit type is a cpo.*

Corollary 6.6. *If A is a cpo, then $1 \rightarrow ? A$ is a cppo.*

In general, the sum of two cpos, even $\text{Bool} = 1 + 1$, need not be a cpo when equipped with the sum ordering. However, we have

Lemma 6.7. *Cpo’s are stable under sums of partial orders iff Bool is a flat cpo.*

The syntactic sugaring of domains in HASCASL includes a **free domain** construct that declares initial algebras in the category of cpo’s and continuous functions (rather than in the category of types and functions as in the case of **free type**). We now show that the initial datatypes and final process types for polynomial and extended polynomial functors F , respectively, constructed in the proof of Theorem 5.6 can be made (respectively, in the case of final process types, slightly modified) into cpo’s in such a way that they become

initial algebras and final coalgebras, respectively, for the corresponding functor, denoted \bar{F} , on the category of cpo's and continuous functions, where in the case of extended polynomial functors, function spaces are replaced by continuous function spaces. It is an important open problem whether this result can be extended to datatypes t with non-strict constructors, i.e. with arguments of type $1 \rightarrow? t$, such as the type of lazy lists. In the following, we assume that *Nat* is a flat cpo (this may or may not be the case in concrete models [25]); consequently, *Bool* is also a flat cpo, and hence cpo's are stable under sums by Lemma 6.7.

Initial Datatypes as Cpo's. Let T be the initial algebra for the functor $FX = \sum_{i=1}^n (A_i \times X^{k_i})$ as in Sect. 5, where the parameter objects A_i are cpo's. Then the ordering on T is inherited, reusing here and below the notation from the proof of Theorem 5.6, from *DTree* (this is equivalent to the obvious recursive definition of a componentwise ordering), which by the above results and under the given assumptions is a cppo.

Proposition 6.8. *With the above ordering, T is an initial \bar{F} -algebra in the category of cpo's and continuous functions.*

Proof. It is easy to see that the constructors c_i as defined in the proof of Theorem 5.6 are continuous. To prove that T is a cpo, it suffices to show that the supremum in *DTree* of a partial chain s in T is again in T , provided that $\sup s$ is a defined value in *DTree*. We proceed by induction over *depth* $\sqcup s_m$. Let $s_m = (l_m, d_m)$ for all m , and let $\sup s_m = (l, d)$. Then $l \text{ nil} = \text{in}_i ()$ for some i . By the definition of the sum ordering and Lemma 6.2, there is some m such that $l_r \text{ nil} = \text{in}_i ()$ for all $r \geq m$. Since l_r is in T , we have $s_r = c_i (\text{leaf}_i s_r, \text{sel}_1 s_r, \dots, \text{sel}_{k_i} s_r)$ and $\text{sel}_j s_r : T$ for $j = 1, \dots, k_i$ and $r \geq m$. By continuity of c_i , it now follows from the inductive assumption that $\sup s_m$ belongs to T .

It remains to be shown that for continuous functions b_i representing a \bar{F} -algebra on a cpo B , the function $\text{fold } b_1 \dots b_n : T \rightarrow B$ is continuous. It is easy to see that, given the auxiliary function $f : \text{Nat} \rightarrow T \rightarrow? B$ from the proof of Theorem 5.6, the function $f \ n$ is continuous for every n in *Nat*. Since *Nat* is equipped with the flat ordering, it follows that f itself is continuous. Continuity of $\text{fold } b_1 \dots b_n = \lambda z \bullet f (\text{depth } z) z$ then follows by the (obvious) continuity of *depth*. In fact, f even depends continuously on the b_i , so that fold itself is continuous. \square

We have thus established that

*the category of cpo's in a pccc with equality, finite coproducts, and NNO has
initial algebras of polynomial functors if the NNO is a flat cpo,*

and hence that declarations of **free domains** for polynomial functors in HASCASL are conservative as extensions of the specification of sums and a flat cpo of natural numbers; moreover, the above proof shows additionally that the fold operator, and hence the primitive recursion operator, is a continuous higher order function.

Final Process Types as Cpo's. Unlike in the case of initial datatypes, we have to modify the universal type *PTree* to the type

$$CPTree = \text{Path} \xrightarrow{c} ?A,$$

again reusing the notation from the proof of Theorem 5.6, in order to obtain a coalgebra structure for \bar{F} . By the above results, including the fact that list types are cpos, *CPTree* is a cppo. The definition of the subtype C , the structure map $c : C \rightarrow \bar{F}C$, and the function

$u = \text{unfold } d : D \rightarrow C$ for a continuous \bar{F} -coalgebra d on a cpo D are otherwise literally the same as in the proof of Theorem 5.6. It is easy to see that C is closed under suprema of chains in $CPTree$ and hence a cpo. Since C consists of continuous maps, $c \circ f$ is really in $\bar{F}C$ (where functions $B_i \xrightarrow{c} C$ must be continuous) for $f : C$. It is straightforward to check that c and u are continuous, using in the latter case the fact established above that primitive recursive functions (here, on $Path$), as well as the primitive recursion operator itself, are continuous. We have thus shown that

the category of cpo's in a pccc with equality, finite coproducts, and NNO has final coalgebras of extended polynomial functors if the NNO is a flat cpo.

and hence that corresponding declarations of final process types as cpos for extended polynomial functors in HASCASL are conservative as extensions of the specification of sums and a flat cpo of natural numbers. (Recall that such declarations are not a HASCASL language feature as such, but can be emulated according to Sect. 3 and 4.)

7. 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 (more precisely, in any partial cartesian closed category with equality and finite coproducts) with a natural numbers object. We have moreover discussed how the constructions can be adapted to yield corresponding types with a domain structure as used in HASCASL's internal modelling of general recursion.

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 and lazy constructors. Support for datatypes with polynomial signatures is already implemented in the heterogeneous tool set Hets [11]; support for more complex signatures, intertwined with HASCASL's type class mechanism as described here, is forthcoming.

ACKNOWLEDGEMENTS

The author wishes to thank Till Mossakowski for collaboration on HASCASL, and Peter Johnstone and Sam Staton for helpful remarks.

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 Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics, TPHOLs 1999*, vol. 1690 of *Lect. Notes Comput. Sci.*, pp. 19–36. Springer, 1999.
- [3] M. Bidoit and P. D. Mosses. CASL *User Manual*, vol. 2900 of *Lect. Notes Comput. Sci.* 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] J.-Y. Girard. *Proofs and Types*. Cambridge University Press, 1989. Translated and with appendices by P. Taylor and Y. Lafont.
- [6] E. L. Gunter. A broader class of trees for recursive type definitions for HOL. In J. J. Joyce and C.-J. H. Seger, editors, *Higher Order Logic Theorem Proving and Its Applications, HUG 1993*, vol. 780 of *Lect. Notes Comput. Sci.*, pp. 141–154. Springer, 1993.

- [7] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In H. Nilsson, editor, *Haskell Workshop, Haskell 2004*, pp. 96–107. ACM, 2004.
- [8] J. Lambek and P. J. Scott. *Introduction to Higher-Order Categorical Logic*. Cambridge, 1986.
- [9] I. Moerdijk and E. Palmgren. Wellfounded trees in categories. *Ann. Pure Appl. Logic*, 104:189–218, 2000.
- [10] E. Moggi. Categories of partial morphisms and the λ_p -calculus. In D. H. Pitt, S. Abramsky, A. Poigné, and D. E. Rydeheard, editors, *Category Theory and Computer Programming*, vol. 240 of *Lect. Notes Comput. Sci.*, pp. 242–251. Springer, 1986.
- [11] T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set. In O. Grumberg and M. Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2007*, vol. 4424 of *Lect. Notes Comput. Sci.*, pp. 519–522. Springer, 2007.
- [12] T. Mossakowski, L. Schröder, M. Roggenbach, and H. Reichel. Algebraic-co-algebraic specification in CoCASL. *J. Logic Algebraic Programming*, 67:146–197, 2006.
- [13] P. D. Mosses, editor. *CASL Reference Manual*, vol. 2960 of *Lect. Notes Comput. Sci.* Springer, 2004.
- [14] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, vol. 2283 of *Lect. Notes Comput. Sci.* Springer, 2002.
- [15] L. C. Paulson. Set theory for verification. II: Induction and recursion. *J. Autom. Reasoning*, 15:167–215, 1995.
- [16] L. C. Paulson. Mechanizing coinduction and corecursion in higher-order logic. *J. Log. Comput.*, 7:175–204, 1997.
- [17] 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.
- [18] A. Pitts. *Categorical Logic*, vol. 5, Algebraic and Logical Structures, chapter 2. Oxford University Press, 2000.
- [19] F. Regensburger. HOLCF: Higher order logic of computable functions. In E. T. Schubert, P. J. Windley, and J. Alves-Foss, editors, *Theorem Proving in Higher Order Logics, TPHOLS 1995*, vol. 971 of *Lect. Notes Comput. Sci.*, pp. 293–307, 1995.
- [20] G. Rosolini. *Continuity and Effectiveness in Topoi*. PhD thesis, Merton College, Oxford, 1986.
- [21] G. Rosolini and T. Streicher. Comparing models of higher type computation. In L. Birkedal, J. van Oosten, G. Rosolini, and D. S. Scott, editors, *Realizability Semantics and Applications*, vol. 23 of *Electron. Notes Theoret. Comput. Sci.*, 1999.
- [22] L. Schröder. The logic of the partial λ -calculus with equality. In J. Marcinkowski and A. Tarlecki, editors, *Computer Science Logic*, vol. 3210 of *Lect. Notes Comput. Sci.*, pp. 385–399. Springer, 2004.
- [23] L. Schröder. The HASCASL prologue - categorical syntax and semantics of the partial λ -calculus. *Theoret. Comput. Sci.*, 353:1–25, 2006.
- [24] L. Schröder. Bootstrapping types and cotypes in HASCASL. In T. Mossakowski and U. Montanari, editors, *Algebra and Coalgebra in Computer Science, CALCO 2007*, vol. 4624 of *Lect. Notes Comput. Sci.*, pp. 447–462. Springer, 2007.
- [25] L. Schröder and T. Mossakowski. HASCASL: Integrated higher-order specification and program development. Available under http://www.informatik.uni-bremen.de/agbkb/forschung/formal_methods/CoFI/HasCASL.
- [26] L. Schröder and T. Mossakowski. HASCASL: Towards integrated specification and development of Haskell programs. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methodology and Software Technology, AMAST 2002*, vol. 2422 of *Lect. Notes Comput. Sci.*, pp. 99–116. Springer, 2002.
- [27] L. Schröder and T. Mossakowski. Monad-independent Hoare logic in HASCASL. In M. Pezzè, editor, *Fundamental Approaches to Software Engineering, FASE 2003*, vol. 2621 of *Lect. Notes Comput. Sci.*, pp. 261–277. Springer, 2003.
- [28] L. Schröder and T. Mossakowski. Generic exception handling and the Java monad. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Algebraic Methodology and Software Technology, AMAST 2004*, vol. 3116 of *Lect. Notes Comput. Sci.*, pp. 443–459. Springer, 2004.
- [29] L. Schröder and T. Mossakowski. Monad-independent dynamic logic in HASCASL. *J. Logic Comput.*, 14:571–619, 2004.
- [30] L. Schröder, T. Mossakowski, and C. Lüth. Type class polymorphism in an institutional framework. In J. Fiadeiro, editor, *Recent Developments in Algebraic Development Techniques, 17th International Workshop, WADT 04*, vol. 3423 of *Lect. Notes Comput. Sci.*, pp. 234–248. Springer, 2004.

- [31] D. Walter, L. Schröder, and T. Mossakowski. Parametrized exceptions. In J. Fiadeiro and J. Rutten, editors, *Algebra and Coalgebra in Computer Science, CALCO 05*, vol. 3629 of *Lect. Notes Comput. Sci.*, pp. 424–438. Springer, 2005.
- [32] O. Wyler. *Lecture notes on topoi and quasitopoi*. World Scientific, 1991.