HasCasl: Integrated Higher-Order Specification and Program Development

Lutz Schröder and Till Mossakowski [1]

DFKI-Lab Bremen and Dept. of Comput. Sci., Universität Bremen

Abstract

We lay out the design of HasCasl, a higher order extension of the algebraic specification language Casl that serves both as a wide-spectrum language for the rigorous specification and development of software, in particular but not exclusively in modern functional programming languages, and as an expressive standard language for higher-order logic. Distinctive features of HasCasl include partial higher order functions, higher order subtyping, shallow polymorphism, and an extensive type-class mechanism. Moreover, HasCasl provides dedicated specification support for monad-based functional-imperative programming with generic side effects, including a monad-based generic Hoare logic.

Key words: Algebraic specification, functional programming, type classes, polymorphism, Casl, monads, Hoare logic

Introduction

The rigorous development of software from abstract requirements to executable code calls for wide-spectrum languages that are sufficiently powerful and flexible to support both an expressive specification logic and concepts appearing in advanced programming languages, including modern functional languages such as Haskell [50], but also imperative and object-oriented languages. Here, we discuss the design of such a wide-spectrum language, HasCasl. HasCasl is an extension of the standard algebraic specification language Casl (*Common Algebraic Specification Language*) [6,45] developed by the Common Framework Initiative (CoFI) of IFIP WG 1.3, and as such has been adopted by IFIP WG 1.3. It arguably constitutes 'the' natural higher

[1] Research supported by the DFG project HasCASL (KR 1191/7-1/2)

order extension of Casl, and is intended, beyond its purpose as a software specification language, as an expressive standard language for higher order logic. In particular, HasCasl is presently the most expressive language in the logic graph underlying the Bremen heterogeneous tool set Hets [44] and as such serves as a central hub for the interchange of theories between various formalisms in the tool.

The core of HasCasl is a higher order logic of partial functions built on top of Moggi's partial λ-calculus [38]. The semantics and proof theory of this logic have been developed in a companion paper [59]; essentially, one arrives at an intuitionistic partial higher order logic without choice principles (even unique choice). The full HasCasl logic extends the core logic by subtyping and type-classed based shallow polymorphism, including higher-order type constructors and constructor classes; the semantics of the latter is based on models explicitly incorporating signature extensions [64]. Support for general recursive functions is bootstrapped in the style of HOLCF [55] by specifying a theory of fixed point recursion on complete partial orders. Extensive syntactical sugaring of these concepts yields an executable sublanguage which is in close correspondence with a large subset of Haskell.

Part of the technical difficulties arising in the development of these concepts stem from the above-mentioned lack of unique choice in the core logic; in particular, additional effort is required in the construction of inductive datatypes and in setting up the theory of complete partial orders. We believe that this effort is justified, as making do without unique choice allows keeping the model theory more general and forces the use of simpler constructions. A more detailed discussion of this point is found in Sect. 2.

HasCasl is powerful enough to serve as a framework for the definition of further advanced specification logics. We illustrate this point by developing a generic Hoare logic for reasoning about functional-imperative programs with generic side-effects. Following seminal work by Moggi [40], side effects are encapsulated in functional programming via so-called monads; in particular, this is one of the central concepts of Haskell [68]. Monads model a wide range of computational effects: e.g., stateful computations, non-determinism, exceptions, input, and output can all be viewed as monadic computations, and so can various combinations of these concepts such as non-deterministic stateful computations. Our Hoare logic provides a generic *logical* environment for reasoning about monadic computations. In this way, we generalise the suggestions of [40], which were aimed purely at a state monad with state interpreted as global store. We provide both a generic kernel calculus and specialised calculi that provide additional rules dealing with monad-specific operations such as assignment. We end up with an environment that offers not only a combination of functional and imperative programming (as provided in Haskell), but also a surrounding logic that is rather effortlessly adapted to the specification of both functional and imperative aspects.

The material is organised as follows. We give a brief introduction to Casl in Sect. 1. We then recall HasCasl's core logic in Sect. 2. Sections 3–5 deal with the syntax and semantics of type class oriented shallow polymorphism, subtyping, and inductive datatypes, respectively. The HOLCF style modelling of general recursion is treated in Sect. 6. In Sect. 7, we discuss the integration of HasCasl into the heterogenuous tool set, in particular its connection with Casl, Isabelle/HOL, and Haskell. Sections 8–11 are concerned with the monad-based generic Hoare calculus. We give an introduction to monad-based functional-imperative programming, and then discuss generic notions of side-effect freeness and the calculus proper. We illustrate the calculus by means of an extended example, where Dijkstra's non-deterministic implementation of Euclid's algorithm is verified over a generic non-deterministic reference monad. A prelimary version of the HasCasl design has appeared in [61]; the sections on the monad-based generic Hoare logic extend [62].

Related frameworks There are several approaches to tackling the transition from specifications to programs in the literature. Many of them, including Larch [23] and VSE-2 [29], keep the level of specifications in the well-studied realm of first-order logic, while the more problematic features of programming languages are dealt with in intermediate logics (like the dynamic logic of VSE-2) or in programming language specific interface languages (as in Larch). Extended ML [31] avoids such mediating languages by building a higher order specification language on top of a programming language; however, the result is a large and quite unmanageable language. A similar but more manageable approach is taken by the Programatica framework [24], where a specification logic for Haskell called P-logic [25] is provided. In particular, P-logic resembles our approach in the way polymorphism and recursion are supported; the latter is based on an axiomatic treatment of complete partial orders. P-logic differs from our approach in that it is directly built on top of Haskell (with all its specialities like lazy pattern-matching), while we provide a general-purpose higher-order logic that is both a generalization of classical higher-order logic *and* that can be used as a specification logic for Haskell programs. In particular, this means that HasCasl can be used for loose requirement specifications that are later refined into design specifications and programs, which is not possible with the P-logic approach. Moreover, going beyond the scope of P-logic, we cover type class based overloading (also supporting constructor classes, needed for the specification of monads), as well as a Hoare logic for imperative (monad-based) programs.

Other approaches such as CaféOBJ [15] and the related tool Maude [11] opt for making the specification language itself directly executable, however at the expense of a reduced expressivity of the logic. VDM [30] and Z [66] are *model-oriented* specification languages, i.e. a specification typically describes one single intended input-output behaviour. By contrast, CASL and HASCASL allow for loose specifications that abstractly describe whole collections of behaviours in order to avoid overspecification in the early phases of the development.

In terms of the logic employed, HASCASL is in many ways related to Isabelle/HOL [46] and Isabelle/HOLCF [55], respectively, with the crucial difference being that HASCASL works with a more flexible logic that does not impose strong reasoning principles such as excluded middle and choice from the outset (like Coq [13], HASCASL allows adding such principles explicitly as axioms if desired). In fact, Isabelle/HOL presently forms the core of the reasoning support for HASCASL. The gap to be bridged here stems mainly from the fact that HASCASL is a specification language aimed at ease of expression, while the logic of Isabelle/HOL is an input language for a proof tool, and as such more austere. Features of HASCASL not directly supported in Isabelle include higher order type constructors and constructor classes (the latter are needed e.g. for modelling side-effects via monads as explained above), subtyping, partial functions, and advanced structured specification constructs. Similar comments apply to other higher-order theorem provers such as PVS [48].

Existing dedicated higher-order frameworks for software specification include Spectrum [8] and RAISE [19]. Spectrum is in some ways a precursor of HASCASL, in particular supports higher-order functional programming and offers a type class system (without constructor classes). It is however designed entirely as a language for complete partial orders; consequently, it has a three-valued logic admitting undefined truth values and moreover does not include a proper higher-order *specification* language (i.e. non-continuous functions are included for specification purposes, but higher order mechanisms such as λ-abstraction are limited to continuous functions). The RAISE specification language RSL concentrates on direct support for imperative programming and non-determinism, covered in HASCASL by a monad mechanism. The main differences with HASCASL are that RSL has a three-valued logic and does not support polymorphism.

1 CASL

The specification language CASL (*Common Algebraic Specification Language*) has been designed by CoFI, the international *Common Framework Initiative for Algebraic Specification and Development* [12]. Its features include first-order logic, partial functions, subsorts, sort generation constraints, and structured and architectural specifications. For the definition of the language including a full formal semantics cf. [45]. An important point here is that the semantics of structured and architectural specifications is institution-independent, i.e. independent of the logic employed for basic specifications. In order to define the envisaged extension of CASL, it is therefore sufficient to define the underlying logic in the form of an *institution* [20], i.e. essentially to fix notions of signature, model, sentence, and satisfaction as done below.

We briefly point out some particularities of the CASL notation that appear in more or less the same form in HASCASL, but refer to [45,6] for a full explanation of the concepts involved. The CASL logic is *multisorted*; the user may declare sorts by means of the keyword **sort**. Sorts appear in the profiles of operations and predicates. The interpretation of sorts, operations etc. is, by default, *loose*, i.e. a sort may be interpreted by any set and an operation may be interpreted by any map of the given profile, as long as the axioms of the specification are satisfied. The latter, of course, may force an essentially unique interpretation; this holds in particular for axioms implicit in datatype declarations. CASL supports partial operations; the fact that an operation is possibly partial is indicated by a question mark in its profile, i.e. a partial operation f with argument sorts $s_1, \dots, s_n$ and target sort t is declared in the form

op $\quad f : s_1 \times \dots \times s_n \to ?t$

while for a total operation the profile is written in the form $s_1 \times \dots s_1 \to t$. There are atomic formulas for definedness: the formula *def* α asserts that the term α is defined. Partial functions are *strict*, i.e. *def* $f(\alpha)$ always implies *def* α. There are two forms of equations between partial terms: a *strong equation* $\alpha_1 = \alpha_2$ asserts that α_1 is defined iff α_2 is defined, and in this case both terms are equal, while an *existential equation* $\alpha_1 \stackrel{e}{=} \alpha_2$ asserts that α_1 and α_2 are defined and equal. Finally, CASL distinguishes between local and global *variables*: the scope of a global variable, declared using the keyword **var**, is the entire remaining basic specification, while the scope of a local variable, declared using **forall**, is limited to the immediately following list of axioms. Both global and local variables are understood to be universally quantified.

2 The Basic Logic of HASCASL

HASCASL is based on the partial λ-calculus with equality as introduced in [38,39,56]. The model theory of HASCASL uses results of [59] relating the categorical semantics given in [38] with a set-theoretic semantics which is compatible with the existing semantics of CASL.

2.1 The Basic Type System

The (extensible) type system of HasCasl features product types, partial and total function types, and a unit type. Types are built from user-declared basic types introduced by the keyword **type** (for the sake of compatibility with Casl, the keyword **sort** may be used alternatively; moreover, most HasCasl keywords may also be used in their plural forms). E.g. writing

types S, T

declares two basic types S and T. As in Casl, the interpretation of basic types is, by default, loose (cf. Sect. 1). From these basic types and the *unit type* $Unit$, the *types* are inductively generated by taking *product types* $s_1 \times \cdots \times s_n$ and *partial* and *total function types* $s \rightarrow ?t$ and $s \rightarrow t$, respectively, with $s \rightarrow ?t$ a type of *strict* partial functions (cf. Sect. 1).

A type may be abbreviated by means of a *synonym*, using also the keyword **type**, by writing e.g.

type $Binary := (S \times S) \rightarrow T$

The type referred to by a type synonom is called its *expansion*. Although the same keyword is used, synonyms are not basic types. A synonym may be defined only once. Recursive synonym definitions are not allowed.

Terms are formed along with their types; we will introduce the term formation rules informally below. The judgement that a term α has type s is written $\alpha : s$. In fact, term formation depends also on a *context* of typed variables which may include local variables introduced by quantifiers or λ-bindings as well as global variables (introduced by the keyword **var**); we will largely omit this aspect here. Terms are built from variables and user-declared *constants*. A constant (or *operation*) f of type s is declared by writing

op $f : s;$

(with the same mechanisms for declaring several constants at once as in Casl). Instead of **op**, the keyword **fun** may be used (**op** and **fun** differ w.r.t. their behaviour under subtyping; cf. Sect. 4). Since s may be a function type, this provides also a way to declare operations with arguments. As in Casl, constants may be *overloaded*, i.e. a constant $f : s$ is made up of its *name* f and its type s, and the same name may appear in different constants of different types. In fully statically analysed formulas, all constants are explicitly annotated with their types, while they are usually referred to by just their name in the input syntax if the context information suffices for disambiguation. There is a built-in overloaded constant $\stackrel{e}{=}$, called *internal equality*, of type $s \times s \rightarrow ?\,Unit$ for each s, which has a fixed interpretation as equality (due to strictness necessarily existential), exploiting the identification of predicates and partial functions into *Unit* (cf. Sect. 2.3).

The built-in type constructors come with associated term formation rules. Terms of type $s_1 \times \cdots \times s_n$ may be constructed as *tuples* $(\alpha_1, \ldots, \alpha_n)$, where α_i is a term of type s_i for $i = 1, \ldots, n$. The empty tuple () is a term of type *Unit*. Application of a term $\alpha : s \rightarrow ?t$ or $\alpha : s \rightarrow t$ to a term $\beta : s$ is denoted by juxtaposition in the form $\alpha\,\beta$. Given a term α in a context containing an additional variable $x : s$, the partial function that takes x to the term α is denoted by $\lambda\, x : s \bullet \alpha$. If α is defined for all possible values of x, then $\lambda\, x : s \bullet!\, \alpha$ denotes the corresponding total function of type $s \rightarrow t$; otherwise, the term $\lambda\, x : s \bullet!\, \alpha$ is still well-formed, but fails to denote a defined value — contrastingly, the term $\lambda\, x : s \bullet \alpha$ is always defined.

Definition 1 A *basic* HasCasl *signature* consists of sets of basic types, type synonyms, and constants, together with a map associating to each type synonym its expansion as defined above. A *morphism* of basic signatures consists of three maps taking constants to constants, type synonyms to type synonyms, and basic types to basic types or type synonyms, respectively; these maps are required to be compatible in the expected sense with types of operations and expansions of type synonyms. (They are *not* required to preserve name equality of constants; cf. however Definition 27.)

Remark 2 From the point of view of the specifier, the relevance of the notion of signature morphism is mainly that it determines which argument fittings are admissible in instantiations of parametrised specifications [45]. Since the above definition explicitly allows signature morphisms to map basic types to type synonyms, basic types can be instantiated with composite types, albeit at the cost of having to define a type synonym first (allowing basic types to be mapped directly to composite types would strongly increase the number of signature morphisms matching a so-called raw symbol map [45] and thus make symbol maps harder to write and parse). A consequence is that the signature category fails to be cocomplete (while its non-full subcategory consisting of the signature morphisms that map basic types to basic types *is* cocomplete, being essentially the category of models of a Horn theory). However, the pushouts required for instantiating parametrised specifications do exist, which is all that is needed for HasCasl structured specifications.

HasCasl provides the following further term forming operations as convenient syntactic sugar:

Let-terms Local bindings are written *let* $x = \alpha$ *in* β, abbreviating

$(\lambda\, x \bullet \beta)\, \alpha$. Equivalently, the form β *where* $x = \alpha$ may be used. Consecutive bindings may be gathered in the form *let* $x_1 = \alpha_1; \ldots; x_n = \alpha_n$ *in* β.

Iterated abstraction Consecutive λ-abstractions may be combined in the form $\lambda\, x_1\; x_2 \ldots\; x_n \bullet \alpha$, abbreviating $\lambda\, x_1 \bullet!\, \lambda\, x_2 \bullet!\, \ldots \lambda\, x_n \bullet \alpha$. Abstraction over an unused variable of type *Unit* may be written in the form $\lambda\; \bullet \alpha$.

Patterns Variables may be bound within *patterns* in the same way as in functional programming. In the language introduced so far, this means that variables may be bound to components of tuples; e.g. in the term *let* $(x, y) = \alpha$ *in* β, where $\alpha : s \times t$, x is bound to the first component of α and y to the second component. In the full language, patterns may also contain datatype constructors; cf. Sect. 5. Patterns may be arbitrarily nested. HasCasl does *not* include built-in projection functions for product types; these can either be user-defined or replaced by pattern matching. In the meta-theory, we do use fst and snd to denote the projections for a binary product type.

Restriction Given terms $\alpha : s$, $\beta : t$, the term α *res* β abbreviates the term *let* $(x, y) = (\alpha, \beta)$ *in* x. That is, α *res* β is defined iff α and β are defined, and in this case equals α. As a special case, β may be a formula (Section 2.3); in this case, α *res* β is defined iff α is defined and β holds.

Typed terms Again as in Casl, terms as well as patterns may be annotated with their intended type in the form $\alpha : s$. This affects only the static analysis of the term in that fewer possibilities have to be considered in the resolution of overloading.

Further extensions by abbreviation of both the type system and the term formation rules are discussed in Sect. 2.3 and 2.4. To illustrate some features of the basic logic, we give a pedestrian specification of an abstract while operator as a least fixed point (a more concise specification using a built-in notion of general recursion is given in Fig. 10) in Fig. 1; the specification imports a two-valued type of booleans contained in the specification Sums shown further below (Fig. 3).

```
spec  WHILE = SUMS with Bool, if _ then _ else _   then
      type   s
      ops    while: (s → Bool) → (s →? s) → s →? s;
             _ ⊑ _ : Pred ((s →? s) × (s →? s))
      vars   b: s → Bool;  p: s →? s;  x: s
             • _ ⊑ _ = λ(q, r): (s →? s) × (s →? s) •
                    ∀y: s • (r y) res (q y) = q y
             • let F = λq: s →? s;  x: s • if b x then x else q (p x);
                    w = while b p in
                          F w = w  ∧ ∀q: s →? s • F q = q ⇒ w ⊑ q
```

Fig. 1. Specification of an abstract while operator

The semantics of HasCasl extends the set-theoretic semantics of first order Casl; that is, types are interpreted as sets, and constants are interpreted as elements of these sets. The principal issue is then the interpretation of function types; common options include the following.

- In *standard* models, function types $s \to t$ and $s \to ?t$, respectively, are interpreted by the full set of (partial) functions from the interpretation of s to that of t.
- In *extensional Henkin* models [27], function types are interpreted by subsets of the full set of functions in such a way that all λ-terms can be interpreted; the latter property is called *comprehension*. (In the model theory of the total λ-calculus, similar models are called λ*-models*).
- In *intensional* Henkin models (similar to λ*-algebras*), function types are interpreted by arbitrary sets equipped with an application operation. Comprehension is still required; however, the way λ-terms are interpreted is now part of the structure of the model rather than just an existence axiom. Intensionality is discussed e.g. in [37,38].

The notion chosen for HasCasl is that of intensional Henkin models. Intensional models behave well w.r.t. existence of initial models (unlike extensional models; cf. [3]) and, unlike standard models, admit a complete deduction system (completeness for extensional models is at least difficult, cf. [38]). Moreover, they are the natural models for *intuitionistic* higher order logic, which is useful from a computational point of view e.g. in proofs-as-programs paradigms [4,36] and has better proof-theoretic properties than classical logic. (That said, the logic can be specified to be classical by the user if desired; see Sect. 2.3.) We do however introduce variants of HasCasl with extensional and even standard models in order to facilitate the embedding of Casl into HasCasl (cf. Sect. 7).

A peculiarity of the intensional approach is that the definition of model requires a deduction system for equality. Sound and complete deduction systems for equality in the partial λ-calculus are given in [38,59]. These systems are easily extended (e.g. using the results of [59]) to encompass product types and total function types as featured in HasCasl. The rules include partial versions of (β), (η), and (ξ), where attention has to be paid to definedness issues (e.g. the strong equation $(\lambda x \bullet \alpha)\,\beta = \alpha[\beta/x]$ holds only when β is defined). Internal equality is treated by means of two axioms which ensure interpretation as actual equality.

Definition 3 An *(intensional Henkin) model* of a given HasCasl signature is an assignment of a set M_s to each type s, in such a way that *Unit* is interpreted as a singleton set and product types are interpreted as cartesian products, together with an assignment of a partial interpretation function

$$M_{s_1} \times \cdots \times M_{s_n} \rightarrow ? M_t$$

to each term of type t in context $(x_1 : s_1, \ldots, x_n : s_n)$. These interpretation functions are required to respect deducible equality of terms. Moreover, substitution must be modeled as composition of partial functions, and terms of the form $x_1 : s_1, \ldots, x_n : s_n \rhd x_i : s_i$ must be interpreted by the appropriate product projections. (It follows that tuple terms are interpreted by tupling of functions, and that all total functions that live in the partial function type are represented in the total function type; the latter is proved using total λ-abstraction.)

A *model morphism* between two such models is a family of functions h_s, where s ranges over all types, that satisfies the usual (weak) homomorphism condition w.r.t. all terms.

By the results of [59], Henkin models are equivalent to models in partial cartesian closed categories (pccc's) with equality as defined in [38]; typical examples of pccc's with equality are quasitoposes [70]. Indeed giving a pccc model is often the easiest way to construct a Henkin model: given a pccc **C** with equality, a Henkin model of a signature may be defined by interpreting sorts as objects in **C**, and constants by (global) elements of the arising interpretations of the corresponding types; such a model is called a *model over* **C**.

2.3 The Internal Logic

Partial functions into *Unit* can be regarded as predicates; an example is the equality operator mentioned above. The type *Pred* s is a built-in synonym for $s \rightarrow ? \mathit{Unit}$. The type $\mathit{Unit} \rightarrow ? \mathit{Unit}$ serves as a type of truth values, with built-in synonym *Logical*. Using the equality operator, one can *define* a full-blown intuitionistic higher order logic, in which logical operators and quantified formulas are just abbreviations for terms [59,58]. E.g., one can put

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

In fact, the definition of the logic can be written as a HasCasl specification [61]. *Satisfaction* of a formula is just definedness of the corresponding term. We refer to this logic as the *internal logic* (the initial design of HasCasl [61] comprised an alternative external logic). The internal logic inherits model-theoretic intensionality; e.g., satisfaction of $\forall x : s.\, \phi$ in a model M is not the same as satisfaction of ϕ by all elements of M_s.

In HasCasl formulas, the usual logical syntax with quantifiers $\forall$, $\exists$ and connectives $\wedge$, $\vee$, $\Rightarrow$, $\neg$, $\Leftrightarrow$, as well as strong and existential equality (denoted $=$ and $\stackrel{e}{=}$, respectively) and definedness assertions *def* α, is used in the same way as in Casl. Moreover, the Casl notation **pred** $p : t$ is retained as an alternative to **op** p : *Pred* t for the sake of compatibility. The difference with the Casl logic is the richer type and term system on the one hand (including quantification over higher order variables) and the fact that the logic is *intuitionistic* on the other hand. The latter is due to the intensional semantics of function types. In particular, the type *Logical* may contain more elements than just the two set-theoretic partial functions from *Unit* to itself, which correspond to the truth values *true* and *false*. If desired, the user may impose classical logic by importing the specification

spec Classical =
var p: *Logical*
• $p \vee \neg p$

(which does not, however, force extensionality of function types).

Besides its intuitionistic nature, a further point about the internal logic is that it distinguishes between *functions*, i.e. inhabitants of function types $a \rightarrow ?b$, and *functional relations*, i.e. right-definite predicates on $a \times b$ (cf. [59] for details). In other words, the internal logic does not in general have a *unique choice* operator ι that, given a formula $x : a \triangleright \phi$, returns the unique element $\iota x : a \bullet \phi$ of type a satisfying ϕ if a unique such element exists (and is defined iff this is the case). Types a for which such an operator does exist are called *coarse*. In models over a topos, all types are coarse. Generally, every type *Pred* a is coarse, including *Logical*, and coarse types are stable under products, function spaces, and subtypes; moreover, every type a has an *underlying* coarse type, the type of singleton subsets of a.

Example 4 Simple examples of models with a classical internal logic but without unique choice are obtained as models over set-based quasitopoi, such as the categories of pseudotopological spaces or (reflexive, symmetric) relations [2]. In such models, the coarse types are precisely those that are interpreted as *indiscrete objects*; e.g. in the quasitopos of (reflexive, symmetric) relations, a type a is coarse iff its interpretation is a set X equipped with the indiscrete binary relation $X \times X$. It should be noted that in such models, initial datatypes (cf. Sect. 5) typically fail to be coarse; e.g. the natural numbers carry the discrete rather than the indiscrete structure.

Remark 5 It will become apparent below (in particular in Sect. 5 and 6) that a certain amount of additional effort is required to make standard concepts and constructions work in the absence of unique choice. The motivation justifying this effort is twofold:

- Imposing unique choice would amount to limiting the semantics to models over toposes, rather than over the more general quasitoposes. Typical examples of quasitoposes, besides the ones mentioned above, 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 [52,57]. Quasitoposes also play a role in the semantics of parametric polymorphism [7]. It thus seems worthwhile to admit quasitopos models.
- 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 [46], Sec. 5.10).

That said, the user may impose unique choice globally or for selected types: a type a is equipped with unique choice by the specification

op $\quad$ $\mathit{choose}\colon\ (\mathit{Pred}\ a)\ \to ?\, a$
var $\quad$ $p\colon \mathit{Pred}\ a;\ x\colon a$
$\quad\quad\bullet\ \mathit{choose}\ p = x \Leftrightarrow (\forall y\colon a \bullet p\ y \Leftrightarrow x = y)$

which be may either imposed on the individual type a or made polymorphic over some class of types (cf. Sect. 3). The terms $\iota x : a \bullet \phi$ mentioned above can then be written in the form $\mathit{choose}\ \lambda x : a \bullet\ \phi$.

Definition 6 A *basic* HasCasl *theory* is a basic signature together with a set of formulas.

For later use, we fix notions concerning subtypes determined by formulas.

Definition 7 A *generalised type* is a pair $(t.\,\phi)$ consisting of a type t and a predicate $\phi : \mathit{Pred}\ t$ (in the terminology of [59], the generalised types are the objects of the classifying category), to be understood as the subtype of all elements $x : t$ satisfying $\phi\, x$.

Remark 8 Products and (partial or total) function spaces of generalised types can again be described as generalised types [59].

2.4 Non-Strict Functions

In HasCasl, as in the partial λ-calculus, function application is *strict*, i.e. defined values are obtained only from defined arguments. This is in keeping with the semantics of both Casl and ML, but not with the semantics of Haskell, where functions are allowed to leave arguments unevaluated and thus yield defined results on undefined arguments. It is well-known that non-strict functions may be emulated in a strict setting by moving to function types $Unit \to ?a$ as argument types. In order to facilitate the specification of programs in non-strict languages, we include non-strict function types in HasCasl as syntactic sugar:

For a type s, $?s$ abbreviates the type $Unit \to ?s$. Thus, we obtain *non-strict function types* such as $?s \to ?t$. There are two typing rules for application of non-strict functions and application of strict functions to non-strict values:

- If α is a term of type $?s \to ?t$ or $?s \to t$ and β is a term of type s, then $\alpha\,\beta$ is a term of type t, in which β is implicitly replaced by $\lambda \bullet \beta$.
- If α is a term of type $s \to ?t$ or $s \to t$ and β is a term of type $?s$, then $\alpha\,\beta$ is a term of type t, in which β is implicitly replaced by $\beta\,()$.

Corresponding generalised rules apply to functions with several arguments. As a simple example, consider the following specification of a non-strict conjunction on the type *Bool* of booleans (cf. Fig. 3 below):

op $And\colon Bool \to (?Bool) \to Bool$
var $x\colon ?Bool$
$\bullet\ And\ False\ x = False$
$\bullet\ And\ True\ x = x$
$\bullet\ And\ False\ True = False$ % **implied**

Here, the last occurrence of $x : ?Bool$ is implicitly replaced by $x()$, while the occurrence of *True* in the last formula (which, as indicated by the Casl annotation %**implied**, is implied by the others) is replaced by $\lambda \bullet\ True$.

3 Type Class Polymorphism

On top of the basic HasCasl logic, we now introduce a form of syntax-oriented shallow type class polymorphism. That is, we allow types and operations to depend on type variables, including type constructor variables, and axioms to be universally quantified over types at the outermost level. Type variables are understood syntactically, i.e. as ranging over all types expressible in the signature. Similarly as in Haskell [50] and Isabelle [46], the range of a type variable may be restricted to a given type class, understood as a subset of the syntactical universe of all types. This naive approach, explained in Sect. 3.1, leads to the problem that the institutional satisfaction condition fails. We therefore introduce a second level of the semantics, where this defect is repaired by moving to so-called extended models; this is laid out in detail in Sect. 3.2.

The class system of HasCasl, like that of Haskell and unlike that of Isabelle, goes beyond simple type classes in that it caters also for type constructors of arbitrary rank. We therefore begin by introducing a suitable kind universe:

Definition 9 From a given set C of *classes*, which includes a class *Type*, the set K of *kinds* is formed by the grammar

$$K ::= C \mid K \to K.$$

Kinds of the form $Kd_1 \to Kd_2$ are called *constructor kinds*. A kind is called *raw* if it contains no classes other than *Type*. A *subclass relation* is a relation $\leq_C$ between classes and kinds, subject to the following condition. Let S denote the congruence (w.r.t. $\to$) generated by $\leq_C$ on the set of kinds; then the S-equivalence class of each class Cl is required to contain a unique raw kind, denoted $\mathsf{raw}(Cl)$ and called the *raw kind* of Cl. It follows that each kind Kd is S-equivalent to a unique raw kind $\mathsf{raw}(Kd)$, obtained by replacing all classes in Kd with their raw kinds.

Intuitively, *Type* is the syntactical universe of all types, constructor kinds are universes of type constructors, and classes are subsets of these universes as prescribed by $\leq_C$.

A class Cl is declared as a subclass of a given kind Kd by writing **class** $Cl < Kd$. Subclasses of constructor kinds are called *constructor classes*. A class may be declared to be a subclass of several kinds, which however all have to be of the same raw kind due to the requirements of Defn. 9. For the same reason, subclass declarations cannot be recursive (e.g. $Cl < Cl \to Cl$ is illegal). Classes declared without explicit superkinds are subclasses of *Type*. For instance,

classes $BoundedOrd < Ord$;
$Functor < Type \to Type$

declares two classes *BoundedOrd* and *Ord* such that *BoundedOrd* $\leq_C$ *Ord* $\leq_C$ *Type*, and a constructor class $Functor$.

The subclass relation, which we denote in the meta-theory by $\leq_C$, is extended to a subkind relation $\leq_K$ on K by the rules given in Fig. 2. (These rules will be extended in Sect. 4; a syntax-directed version of the entire system is given in Appendix C.) By induction over the derivation length, one shows that $Kd_1 \leq_K Kd_2$ implies that Kd_1 and Kd_2 have the same raw kind.

$$\frac{Cl \leq_C Kd \text{ in } \Sigma}{Cl \leq_K Kd} \qquad \frac{Kd_1 \leq_K Kd_2 \qquad Kd_3 \leq_K Kd_4}{Kd_2 \to Kd_3 \leq_K Kd_1 \to Kd_4}$$

$$\frac{}{Kd \leq_K Kd} \qquad \frac{Kd_1 \leq_K Kd_2 \qquad Kd_2 \leq_K Kd_3}{Kd_1 \leq_K Kd_3}.$$

Fig. 2. Subkinding rules

Type variables are declared along with their kind either by means of the keyword **var** or in local universal quantifications at the outermost level (we will use mostly the former style here). Type variables may then be used in place of types or type constructors, thus making the entity (type, operation, or axiom) where they appear polymorphic over the given kind. For a type, this means that one obtains a *type constructor*; i.e by writing

type $\quad t\ a_1 \dots a_n : Kd$

where $a_1, \dots, a_n$ are type variables of kinds $Kd_1, \dots, Kd_n$, respectively, one declares a type constructor t of kind $Kd_1 \to \cdots \to Kd_n \to Kd$ (in particular of kind Kd when $n = 0$). Polymorphic operations are assigned *type schemes* in the usual sense of ML-polymorphism, i.e. types that are quantified over type variables at the outermost level — HasCasl does *not* admit nested type quantification as in System F. Finally, polymorphic axioms are implicitly universally quantified over their free type variables.

A simple example is a specification of sum types, shown in Fig. 3. It declares a type constructor *Sum* of kind *Type* → *Type* → *Type*, as well as polymorphic operations *inl*, *inr*, and *sumcase*, governed by polymorphic axioms. We immediately obtain the type *Bool* of booleans as *Sum Unit Unit*, with the if-then-else construct arising as a special case of the case construct. Moreover, the specification declares a universal undefined constant *bot*, which has the effect of making the generalised type $(\mathit{Unit}, \lambda \bullet \mathit{false})$ an initial object in the model category. Using *bot*, one can define partial extraction functions *outl*, *outr* for sums as shown in the specification.

Instances of polymorphic operations may be formed explicitly using square brackets; e.g. given basic types S and T, we have an instance $\mathit{sumcase}[S, S, T]$ of type $(S \to ?T) \to (S \to ?T) \to (\mathit{Sum}\ S\ S) \to ?T$ (the order of the type arguments is determined by the order of declaration of type variables). However, types of instances may also be automatically inferred, so that instances can be referred to by just the name of operation, as done in the above specification. Note that instances may involve types containing type variables, as in the %**implied** formula in Fig. 3, where the instance of *sumcase* is for a, b, *Sum a b*.

```
spec  Sums =
      vars   a, b, c : Type
      type   Sum a b
      ops    inl : a → Sum a b;
             inr : b → Sum a b;
             sumcase : (a →? c) → (b →? c) → Sum a b →? c
             bot : ?a
      vars   f : a →? c;  g : b →? c;  h : Sum a b →? c
             • h =sumcase f g ⇔
                  (∀x : a;  y : b • h (inl x) = f x ∧ h (inr y) = g y)
             • ¬def bot : a
             • sumcase inl inr = λz : Sum a b • z        %implied
      then   %def
      vars   a, b : Type
      ops    outl : Sum a b → a;
             outr : Sum a b → b
             • outl = sumcase (λx : a • x) (λy : b • bot)
             • outl = sumcase (λx : a • bot) (λy : b • y)
      type   Bool := Sum Unit Unit
      var    p : Bool;  x, w : a
      ops    True, False : Bool;
             if __ then __ else __ : Bool × a × a → a
             • True = inl ()
             • False = inr ()
             • if p then x else w = sumcase (λ • x) (λ • w) p
```

Fig. 3. A specification of sum types and an implicit initial object

Remark 10 Since polymorphic overloading is permitted, explicit instantiations as explained above may be ambiguous in case the given arguments fit more than one polymorphic profile. In this case, the use of the operation is disambiguated by its own expected type (cf. Rem. 15); if necessary, explicit type annotations must be given. Partial explicit instantiation, e.g. by writing $sumcase[S, S]$ in the above example, is not allowed.

Product and function types are regarded as applications of built-in type constructors $\times$, $\rightarrow$, $\rightarrow?$ of kind $Type \rightarrow Type \rightarrow Type$, and the unit type as a type 'constructor' of kind *Type*; all other type constructors are called *user-declared.* Type constructors are distinguished from parametrised type synonyms; e.g.

```
var    a : Type
type   DList a := List List a
```

defines a parametrised type synonym $DList$ of kind $Type \to Type$. In closed form, the expansion of $DList$ is the *pseudotype*

$$\lambda\, a : Type \bullet List\ List\ a.$$

In order not to overtax the user with yet another kind of λ-abstraction, such λ-types are not included directly in the syntax of HasCasl. They do however play a role for the range of type constructor variables; cf. Sect. 3.2.

The extension of a class is determined by the *instances* derivable for it. The instances are governed by two mechanisms: the kinds assigned to type constructors, and the subclass relation. E.g.

$$\begin{array}{ll} \textbf{var} & a : Ord \\ \textbf{type} & List\ a, Nat : Ord \end{array}$$

declares the type Nat to be of class Ord and the type constructor $List$ to be of kind $Ord \to Ord$. The latter statement means that $List\ t$ is of class Ord whenever t is of class Ord. A type constructor may be given any number of kinds, which however are required to have the same raw kind.

To a class, one can attach both operations and axioms by using appropriate type variables. E.g. the standard operations and axioms for the two classes of orders declared above are specified by

$$\begin{array}{ll} \textbf{vars} & a : Ord;\ b : BoundedOrd \\ \textbf{ops} & \_\_ \leq \_\_ : Pred\ (a * a); \\ & bottom, top : b \\ \textbf{vars} & x, y, z : a;\ v : b \\ & \bullet\ x \leq x \\ & \bullet\ x \leq y \wedge y \leq z \Rightarrow x \leq z \\ & \bullet\ x \leq y \wedge y \leq x \Rightarrow x = y \\ & \bullet\ bottom \leq v \\ & \bullet\ v \leq top \end{array}$$

Then e.g. the comparison operator $\leq$ has instances only for types of class Ord — e.g. given the type declarations above, for Nat, $List\ Nat$, $List\ List\ Nat$ etc. — and also the order axioms hold only at these types. Note that, given the declarations so far, the class $BoundedOrd$ has no instances at all, so that no type has operations *bottom* and *top* (however, types of class $BoundedOrd$ may be declared later); more about this point in Sect. 3.2.

Unlike in Isabelle and Haskell, axioms and operations, respectively, do not as such form part of the definition of a class in HasCasl. The effect of Isabelle's axiomatic type classes can however be emulated as follows. To a class declaration, operations and axioms may be attached in a block marked by curly brackets $\{\ldots\}$, thus declaring the *interface* of the class. E.g. the above declaration of operations and axioms for partial orders could be tied to the class declaration by writing

class Ord $\{$
vars $a : \mathit{Ord};\ x : a$
op $\_\leq\_ : \mathit{Pred}\ (a \times a)$
$\bullet\ x \leq x \ldots \}$

Then, declarations of subclasses of Ord and declarations of type constructors with result class Ord can be marked with the keyword **instance**, thus generating a proof obligation similar to CASL's %**implied** annotation which states that the interface axioms of the class follow from the axioms for the type or the subclass, respectively, together with the axioms of the local environment including the class axioms for type arguments. Here, 'follow' refers to the notion of semantic consequence on the second level of the semantics as introduced in Sect. 3.2 below; this notion of consequence is however just the intuitively expected one (and also the one employed in proof systems such as Isabelle). E.g., given the class interface for Ord, we can declare a generic instance for product types by

vars $a, b : \mathit{Ord}$
type instance $(a \times b) : \mathit{Ord}$
var $x, y : a;\ v, w : b$
$\bullet\ (x, v) \leq (y, w) \Leftrightarrow x \leq y \wedge v \leq w$

which gives rise to the proof obligation that reflexivity, transitivity, and antisymmetry of $\leq$ on $a \times b$ follow from the corresponding laws on a and b and the definition of $\leq$ on $a \times b$. Similarly, proof obligations may be generated for subclasses. E.g.,

class instance $\mathit{DiscreteOrd} < \mathit{Ord}$
vars $a : \mathit{DiscreteOrd};\ x, y : a$
$\bullet\ x \leq y \Leftrightarrow x = y$

expresses that the order axioms follow from the definition of $\leq$ for discrete orders.

As indicated above, HASCASL supports polymorphism over higher kinds. As an example involving constructor classes and type constructors of higher rank, a specification of the classes of monads and monad transformers as used in the Haskell libraries [50] is shown in Fig. 5. The specification of monads slightly

$$
\begin{array}{lll}
\textbf{spec} & \multicolumn{2}{l}{\text{FUNCTOR} =} \\
& \textbf{class} & \mathit{Functor} < \mathit{Type} \to \mathit{Type}\ \{ \\
& \textbf{vars} & F : \mathit{Functor};\ a, b, c : \mathit{Type} \\
& \textbf{op} & \mathit{map} : (a \to c) \to F\ a \to F\ b \\
& \textbf{vars} & x : F\ a;\ f : a \to b;\ g : b \to c \\
& & \bullet\ \mathit{map}\ (\lambda y : a \bullet!\ y)\ x = x \\
& & \bullet\ \mathit{map}\ (\lambda y : a \bullet!\ g\ (f\ x))\ x = (\mathit{map}\ g)\ (\mathit{map}\ f\ x) \\
& & \}
\end{array}
$$

Fig. 4. The constructor class of functors

modifies the usual definition, cf. Sect. 8; the axiomatization of monad transformers follows [33]. The class instance mechanism is illustrated by declaring

$$
\begin{array}{lll}
\textbf{spec} & \multicolumn{2}{l}{\text{MONAD} = \text{FUNCTOR}\ \textbf{then}} \\
& \textbf{class} & \mathit{Monad} < \mathit{Type} \to \mathit{Type}\ \{ \\
& \textbf{vars} & m : \mathit{Monad};\ a, b, c : \mathit{Type} \\
& \textbf{ops} & \_\ \gg=\ \_\_ : m\ a \to (a \to ?\ m\ b) \to ?\ m\ b; \\
& & \_\ \gg=\ \_\_ : m\ a \to (a \to m\ b) \to m\ b; \\
& & \mathit{ret} : a \to m\ a \\
& \textbf{vars} & x, y : a;\ p : m\ a;\ q : a \to ?\ m\ b;\ r : b \to ?\ m\ c;\ f : a \to ?\ b \\
& & \bullet\ \mathit{def}\ \ (q\ x) \Rightarrow ((\mathit{ret}\ x) \gg= q) = q\ x \\
& & \bullet\ p \gg= (\lambda x : a \bullet \mathit{ret}\ (f\ x) \gg= r) = \\
& & \qquad\qquad\qquad p \gg= (\lambda x : a \bullet r\ (f\ x)) \\
& & \bullet\ (p \gg= \mathit{ret}) = p \\
& & \bullet\ ((p \gg= q) \gg= r) = (p \gg= (\lambda x : a \bullet q\ x \gg= r)) \\
& & \bullet\ ((\mathit{ret}\ x) : m\ a) = \mathit{ret}\ y \Rightarrow x = y \\
& & \} \\
& \multicolumn{2}{l}{\textbf{class instance}\ \mathit{Monad} < \mathit{Functor}} \\
& \textbf{vars} & m : \mathit{Monad};\ a, b : \mathit{Type};\ f : a \to b;\ x : m\ a; \\
& & \bullet\ \mathit{map}\ f\ x = x \gg= \lambda y : a \bullet \mathit{ret}\ f\ y \\
\\
\textbf{spec} & \multicolumn{2}{l}{\text{MONADTRANSFORMER} = \text{MONAD}\ \textbf{then}} \\
& \textbf{class} & \mathit{MonadT} : \mathit{Monad} \to \mathit{Monad}\ \{ \\
& \textbf{vars} & t : \mathit{MonadT};\ m : \mathit{Monad};\ a : \mathit{Type}; \\
& \textbf{op} & \mathit{lift} : m\ a \to t\ m\ a \\
& \textbf{vars} & x : a;\ p : m\ a;\ b : \mathit{Type};\ q : a \to m\ b \\
& & \bullet\ \mathit{lift}\ (\mathit{ret}\ x) = (\mathit{ret}\ x) : t\ m\ a \\
& & \bullet\ \mathit{lift}\ (p \gg= q) = (\mathit{lift}\ p) \gg= \lambda y : a \bullet ((\mathit{lift}\ (q\ z)) : t\ m\ a) \\
& & \}
\end{array}
$$

Fig. 5. The classes of monads and monad transformers

Monad to be an instance of the constructor class *Functor*, whose definition is shown in Fig. 4.

Formally, the syntax of polymorphism is captured as follows.

Definition 11 A *polymorphic* HasCasl *signature* Σ consists of a set of classes with a subclass relation according to Defn. 9, kinded type constructors, type synonyms with given expansions, and typed polymorphic operations as described above (basic types are regarded as type constructors without type arguments, similarly for monomorphic operations). Kinds of type constructors are subject to the above-mentioned restriction to coincident raw kinds.

A *morphism* of polymorphic signatures consists of maps taking classes to classes, operations to operations, type synonyms to type synonyms, and type constructors to type constructors or type synonyms, respectively. These maps are required to preserve the subclass relation and the kinding of type constructors in the sense that declared subclass relations and kind assignments for type constructors are mapped to derivable subkinding and kinding judgements, respectively, in the target signature, and to be compatible in the expected sense with expansions of type synonyms and types of constants. Moreover, morphisms must preserve polymorphic overloading: if the source signature contains constants $f : t$ and $f : s$, where t and s are unifiable polymorphic profiles, then the images of $f : s$ and $f : t$ must have the same name in the target signature.

Instances of classes are determined by a kinding system for pseudotypes. Kinding takes place in a *type context* Θ of type variables with assigned kinds according to the (syntax-directed) rules shown in Fig. 6 (to be extended in Sect. 4). A judgement of the form $\Theta \rhd t : Kd$ is to be read 't is a type constructor of kind Kd in context Θ'. In the rules, s and t range over pseudotypes and F over basic type constructors; the premise $F\colon Kd_1$ in Σ means that Kd_1 is an explicitly assigned kind of F. The introduction rule for type constructors applies also to the built-in type constructors $\times$, $\rightarrow$, $\rightarrow?$, *Unit*. A *type* is a pseudotype of kind *Type*.

$$\frac{F : Kd_1 \text{ in } \Sigma \quad Kd_1 \leq_K Kd_2}{\Theta \rhd F : Kd_2} \qquad \frac{a : Kd_1 \text{ in } \Theta \quad Kd_1 \leq_K Kd_2}{\Theta \rhd a : Kd_2}$$

$$\frac{\Theta \rhd t : Kd_1 \quad \Theta \rhd s : Kd_1 \rightarrow Kd_2}{\Theta \rhd s\, t : Kd_2} \qquad \frac{\Theta, a : Kd_1 \rhd t : Kd_2 \quad Kd_3 \leq_K Kd_1}{\Theta \rhd \lambda\, a : Kd_1 \bullet t : Kd_3 \rightarrow Kd_2}$$

Fig. 6. Kinding rules for type constructors

By induction over the type structure, one shows that all kinds derivable for a pseudotype t are of the same raw kind, the *raw kind* of t.

Remark 12 Using the subkinding rules of Appendix C, one shows by induction over the type structure that the kinds derivable for a pseudotype are upwards closed w.r.t the subkind relation (which is why we can require an exact fit in the application rule). Moreover, kinding is invariant under substitution and hence under β-equality (but not under η-equality, which is therefore not imposed on type constructors).

The *first level* of the semantics of polymorphism is defined by reduction to the basic language of Sect. 2 as follows.

Definition 13 The *generalised pseudotypes* of a polymorphic HasCasl signature Σ are formed by the rules of Fig. 6 and an additional rule stating that generalised types (cf. Defn. 7) are generalised pseudotypes of kind *Type*. A *type instance* is a closed generalised pseudotype. The point here is that generalised types may appear as arguments of *user-declared* type constructors — by Rem. 8, we may assume that type instances do not contain applications of the built-in type constructors $\times$, $\rightarrow$, $\rightarrow ?$ to generalised types. A *loose type* is a type instance of kind *Type* which is an application of a user-declared type constructor.

From Σ, we construct a basic theory $\mathsf{B}(\Sigma)$. The sorts of $\mathsf{B}(\Sigma)$ are the loose types of Σ. Operations are translated as follows. Let f have the type scheme $\forall a_1 : Kd_1, \dots, a_n : Kd_n \bullet t$, and let $s_1 : Kd_1, \dots, s_n : Kd_n$ be type instances. By Rem. 8, the type instance $t[s_1/a_1, \dots, s_n/a_n]$ can be interpreted as a generalised type $v = (u_{s_1,\dots,s_n} . \, \phi_{s_1,\dots,s_n})$ in $\mathsf{B}(\Sigma)$. The operation f is then translated into a collection of operations $f_v : u_{s_1,\dots,s_n}$, where $s_1, \dots, s_n$ range over all type instances, together with axioms $\phi_{s_1,\dots,s_n} \, f_v$. Given this construction, it is clear that every morphism $\sigma : \Sigma_1 \rightarrow \Sigma_2$ of polymorphic signatures induces a signature morphism $\mathsf{B}(\sigma) : \mathsf{B}(\Sigma_1) \rightarrow \mathsf{B}(\Sigma_2)$.

The *first level* of the semantics is then given as follows. The *models* and *model morphisms* of Σ are those of $\mathsf{B}(\Sigma)$. Let $\forall a_1 : Kd_1, \dots, a_n : Kd_n \bullet \phi$ be a Σ-formula, where ϕ does not contain further quantification over types. Given type instances $s_1, \dots, s_n$, the formula $\phi[s_1/a_1, \dots, s_n/a_n]$ is translated into a formula $\phi^{s_1,\dots,s_n}$ over $\mathsf{B}(\Sigma)$ by replacing polymorphic operations with the appropriate instances and by eliminating generalised types from quantifiers, i.e. by replacing $\forall x : (t. \, \psi) \bullet \chi$ with $\forall x : t \bullet \psi \, x \Rightarrow \chi$, similarly for $\exists$. A Σ-model M *satisfies* $\forall a_1 : Kd_1, \dots, a_n : Kd_n \bullet \phi$ if M, regarded as a model of $\mathsf{B}(\Sigma)$, satisfies $\phi^{s_1,\dots,s_n}$ for all type instances $s_1, \dots, s_n$.

Remark 14 The polymorphism introduced above is essentially ML-polymorphism. The discourse in [14] may create the impression that the combination of ML-polymorphism and higher order logic is inconsistent. However, this is not the case: as demonstrated above, shallow polymorphism can be coded out by just replacing polymorphic operations and axioms by all their instances. The derivation of Girard's paradox in [14], Sect. 5, is based on the assumption that terms of the language are identified up to untyped β-equality in the absence of type annotations; such an equality is obviously unsound w.r.t. the usual notions of model, and the paradox shows that a language with such an equality is inconsistent. When, as in the usual versions of ML-polymorphism, instantiations of polymorphic constants are internally annotated with their types, the contradiction disappears (i.e. its derivation just produces a type error).

Remark 15 Notice that in the above definition, instances of operations are distinguished by their own types, not by the involved type arguments. This means in particular that polymorphic operations declared with identical names but different profiles agree where their profiles overlap. This is to been seen independently of the fact that for the sake of syntactic convenience, explicit instantiation of polymorphic operations is via their type arguments, which are usually simpler than the type of the operation itself.

For instance, one may write

classes Ord, Num
vars $a\colon Ord;\ b\colon Num$
ops $min\colon a \times a \rightarrow ?\, a;$
$min\colon b \times b \rightarrow ?\, b$

thus giving the function min the two polymorphic profiles $\forall a\colon Ord \bullet a{\times}a \rightarrow ?a$ and $\forall a\colon Num \bullet a \times a \rightarrow ?a$. By Defn. 13, instances for these two profiles at types belonging to both Ord and Num agree. Similarly, overlapping instances of unifiable profiles agree. E.g. one might sensibly first define a polymorphic extension ordering on partial function spaces, and then declare this ordering to be an instance of the class Ord:

vars $a, b\colon Type$
op $\_\_ \leq \_\_\colon Pred\ ((a \rightarrow ?\, b) \times (a \rightarrow ?\, b))$
$\ldots$ *%% Definition of the extension ordering*
type instance $a \rightarrow ?\, b\colon Ord$

One then has two explicit profiles for $\leq$, namely $\forall a\colon Ord \bullet Pred\ (a \times a)$ and $\forall a, b\colon Type \bullet Pred\ ((a \rightarrow ?b) \times (a \rightarrow ?b))$, and the instances of the two operations at the types $Pred\ ((a \rightarrow ?b) \times (a \rightarrow ?b))$ are identical.

3.2 The Extended Model Semantics

As mentioned above, the first level of the semantics of polymorphic HasCasl as defined in the preceding section fails to constitute an institution. We now briefly recall the notion of institution, and discuss the failure of the satisfaction condition at the first level of the semantics. We then go on to define a second level of the semantics which does constitute an institution, making use of a general institution theoretic construction introduced in [64].

Definition 16 [20] An *institution* consists of

- a category of *signatures* and *signature morphisms*;
- a contravariant *model functor* assigning to each signature Σ a category $\mathbf{Mod}(\Sigma)$ of *models* and *model morphisms* and to each signature morphism $\sigma : \Sigma_1 \to \Sigma_2$ a *reduct functor* $\mathbf{Mod}(\Sigma_2) \to \mathbf{Mod}(\Sigma_1)$, whose action on models is denoted by $M \mapsto M|_\sigma$, where $M|_\sigma$ is called the *σ-reduct* of M;
- a covariant *sentence functor* assigning to each signature Σ a set $\mathbf{Sen}(\Sigma)$ of sentences and to each signature morphism $\sigma : \Sigma_1 \to \Sigma_2$ a *translation* $\mathbf{Sen}(\Sigma_1) \to \mathbf{Sen}(\Sigma_2)$, whose action is denoted by $\phi \mapsto \sigma\phi$; and
- for each signature Σ, a satisfaction relation $\models$ on $\mathbf{Mod}(\Sigma) \times \mathbf{Sen}(\Sigma)$

such that for each signature morphism $\sigma : \Sigma_1 \to \Sigma_2$, the *satisfaction condition*

$$M \models \sigma\varphi \iff M|_\sigma \models \varphi$$

holds for all $M \in \mathbf{Mod}(\Sigma_2)$ and all $\phi \in \mathbf{Sen}(\Sigma_1)$.

For HasCasl, we have assembled most of the data required by this definition in the preceding section, with the exception of model reduction and sentence translation; these data are completed as follows.

Definition 17 Recall that a signature morphism $\sigma : \Sigma_1 \to \Sigma_2$ of polymorphic HasCasl signatures (cf. Defn. 13) induces a signature morphism $\mathsf{B}(\sigma) : \mathsf{B}(\Sigma_1) \to \mathsf{B}(\Sigma_2)$ between the associated basic HasCasl signatures. A basic signature morphism induces a reduct functor in the usual way (i.e. $M|_\sigma$ interprets symbols by the interpretations of their σ-translations in M). The model classes of the Σ_i are, by definition, those of the $\mathsf{B}(\Sigma_i)$; the reduct functor for σ is defined to be that of $\mathsf{B}(\sigma)$. The translation map for σ works in the obvious way by replacing all symbols in a formula by their images under σ, which may involve replacing type constructors by type synonyms.

Given these definitions, it is clear that the satisfaction condition fails for the first level of the semantics: e.g., a signature morphism σ may be an inclusion $\Sigma_1 \hookrightarrow \Sigma_2$ into a signature with more types, and thus given a Σ_2-model M, a Σ_1-formula of the form, say, $\forall a : \mathit{Type} \bullet \phi$ may hold for $M|_\sigma$, i.e. for the type instances in Σ_1, but fail to hold for M, i.e. for the additional type instances in Σ_2. As an extreme example, which also illustrates that the notion of model at the first level fails to capture the full intuitive meaning of polymorphic specifications, consider the specification of monads in Fig. 5. This specification introduces only a class, but no types of that class. Therefore, the given axioms for monads do not have any instances, i.e. the specification is, at the first level, model-theoretically vacuous, which is certainly not the intended meaning. A model of an extension of the signature of MONAD where instances of the class *Monad* are declared may very well violate the monad axioms, while its reduct to the signature of MONAD will still trivially satisfy them.

All this is remedied at the second level of the semantics, where models are defined to be first-level models of 'future' extensions of the present signature. This is an instantiation of a generic construction presented in [64]. The definition requires the notion of *derived signature morphisms*, which generalise signature morphisms in that they map

- operation constants to terms;
- type constructors to type instances (cf. Defn. 13); and
- classes to sets of type instances of the appropriate raw kind,

rather than just symbols to symbols. In the compatibility requirements, the subkind relation is then replaced by the subset relation between sets of type instances, derivable kinding judgments by the corresponding elementhood relation, and assigned types of operation constants by derivable types of terms. The notion of model reduction generalises in the obvious way to derived signature morphisms. In order to define a generalised sentence translation, the notion of sentence has to be slightly extended to include universal quantification over type variables ranging over given sets of type instances, with the obvious semantics; given this extended syntax, the definition of translation is straightforward.

Definition 18 The *second level* of the semantics of polymorphic HASCASL is defined as follows. The notions of signature and sentence remain unchanged. An *extended model* of a signature Σ_1 is a pair (N, σ), where $\sigma : \Sigma_1 \to \Sigma_2$ is a derived signature morphism in the sense defined above and N is a (first-level) Σ_2-model. The *reduct* $(N, \sigma)|_\tau$ of (N, σ) along a signature morphism τ is $(N, \sigma \circ \tau)$. The extended model (N, σ) *satisfies* a sentence ϕ if

$$N \models \sigma\phi$$

at the first level.

By the results of [64], we have

Theorem 19 *The second level of the semantics of polymorphic* HASCASL *constitutes an institution.*

Moreover, instead of the pathologies indicated above, we obtain

Theorem 20 *On the second level of the semantics, the Σ-sentence $\forall b_1 : Kd_{21}; \dots; b_m : Kd_{2m} \bullet \psi$ is a semantic consequence of $\forall a_1 : Kd_{11}; \dots; a_n :$*

$Kd_{1n} \bullet \phi$, where the Kd_{ij} are kinds, and ϕ and ψ are formulas not containing further quantification over type variables, iff

$$(\forall a_1 : Kd_{11}; \dots ; a_n : Kd_{1n} \bullet \ \phi) \models \psi$$

on the first level, equivalently on the second level, in the signature obtained from Σ by adding type constructors b_i: Kd_{2i}, $i = 1, \dots, m$.

This is precisely the notion of semantic consequence one would intuitively expect, and also the basis for polymorphic proofs as conducted e.g. in Isabelle [46]. A consequence of the theorem is that the sound and complete proof systems for the partial λ-calculus presented in [38,59] lead to sound and complete proof systems for the second level of the semantics.

A further issue in this context are *model-expansive* or, in CASL terminology, (model-theoretically) *conservative* extensions.

Definition 21 A *theory* in a given institution is a pair $Sp = (\Sigma, \Phi)$ consisting of a signature Σ and a set Φ of Σ-sentences. A *model* of Sp is a Σ-model M such that $M \models \Phi$. A signature morphism $\sigma : \Sigma_1 \to \Sigma_2$ is a *theory morphism* $(\Sigma_1, \Phi_1) \to (\Sigma_2, \Phi_2)$ if

$$\Phi_2 \models \sigma\Phi_1,$$

where $\models$ denotes logical consequence. We say that σ is *(model-theoretically) conservative* if every model M of (Σ_1, Φ_1) has a (Σ_2, Φ_2)-extension, i.e. a model M' of (Σ_2, Φ_2) such that $M'|_\sigma = M$.

It is easy to see that conservative theory morphisms at the second level are sections as derived signature morphisms; conversely, theory morphisms which are sections as derived theory morphisms are conservative [64]. Informally, this means that extensions by syntactic definition are conservative, where thanks to the use of derived signature morphisms in extended models, syntactic definitions of symbols may use e.g. terms to define operation constants and type instances to define type constructors. In particular, equational definitions, well-founded recursive definitions of functions whose result types have unique choice [58], and class declarations are conservative. It will be seen below that, moreover, general recursive definitions over types of a class of domains and subtype definitions are conservative, and inductive datatype definitions are conservative over base theories already containing the natural numbers.

4 Subtyping

For convenience in both writing and reading specifications, HASCASL, like CASL, features coercive subtyping. That is, basic types may be declared to be subtypes of (possibly composite) types; e.g.

$$\begin{array}{ll} \textbf{types} & Nat < Int; \\ & Inj < Int \to Int \end{array}$$

declares Nat to be a subtype of Int, and Inj a subtype of $Int \to Int$ (say, of injective functions). The mutual subtype relation, i.e. type isomorphism, is expressed by '='. Semantically, subtype relations are realised by coercion functions, which are omitted in the notation. Thus, terms of the subtype may be used in terms whenever terms of the supertype are expected. Coercion functions for directly declared subtype relations are required to be injective; however, coercion functions for inferred subtype relations as discussed further below may fail to be injective. Coercion functions are required to be coherent and compatible with overloading; this will be made more precise below. For $s < t$, one has a partial downcast operation $\_\ \mathit{as}\ s : t \to ?s$, defined precisely on the image of s in t, and an elementhood predicate $\_ \in s$ on t, defined as the predicate $\lambda\, x : t \bullet \mathit{def}\ x\ \mathit{as}\ s$. Upcasting of terms may be achieved by explicit type annotations of terms in the form $\alpha : t$.

Subtype relations may also be given polymorphically, i.e. basic type constructors may be declared to be subtypes of pseudotypes. E.g.

$$\begin{array}{ll} \textbf{var} & a : \mathit{Type} \\ \textbf{type} & \mathit{NonrepList}\ a < \mathit{List}\ a \end{array}$$

declares a type constructor $\mathit{NonrepList}$ (say, of nonrepetitive lists) such that instances of $\mathit{NonrepList}$ are subtypes of instances of List. This is briefly expressed by saying that $\mathit{NonrepList}$ is a subtype of List (which may be declared in the form $\mathit{NonrepList} < \mathit{List}$). For the built-in type constructors, we have the subtype relation

$$\_ \to \_ < \_ \to ?\_.$$

Total λ-abstraction is defined as the downcast of partial λ-abstraction to the total function type.

A type constructor F may be declared to be *covariant* or *contravariant*, where the former means that $s < t$ implies $F\,s < F\,t$ and the latter that $t < s$ implies $F\,s < F\,t$. The absence of co- or contravariance is called *non-variance*, while the combination of contravariance and covariance is referred to as *invariance*. Covariance or contravariance of a type constructor are indicated by adding the *variance annotation* $+$ or $-$, respectively, to the corresponding constructor kind or to type variable declarations (similar ideas appear already in [9]; the notation used here is the one applied also e.g. in [1]). E.g., the list constructor might sensibly be declared to be covariant by writing

var	$a\colon +Type$
type	$List\ a\colon Type$

(or shortly $List\colon +Type \to Type$). We do not provide dedicated syntax for invariance; however, invariant type constructors may arise by redeclaration, in unions of specifications, or in instantiations of parametrised specifications where a non-variant type constructor from the formal parameter is e.g. declared to be contravariant in the body of the specification, but instantiated with a covariant type constructor.

A typical example of a contravariant type constructor argument is the function type constructor: if a is a subtype of b, then $b \to c$ is, via function restriction, a subtype of $a \to c$. Explicitly, the built-in type constructors have kinds

$$\begin{aligned} \_ \times \_ &: \ + Type \to + Type \to Type, \\ \_ \to ?\_,\ \_ \to \_ &: \ - Type \to + Type \to Type. \end{aligned}$$

User-declared contravariant type constructors will mostly be related to function types in some way. E.g. one might choose to generalise the type of injective functions to a type constructor, declared by

vars	$a\colon -Type;\ b\colon +Type$
type	$Inj\ a\ b < a \to b$

(this must indeed be declared rather than inferred; e.g. the above declaration would not be sensible for a type of surjective functions).

It is possible to impose *subtyping constraints* on type variables in the form $a < t$, where a is a type variable and t is a pseudotype (similar features are present in the programming language O'Haskell [47]). For instance, the most general way to declare the *twice* function is

vars	$a\colon Type;\ b < a$
op	$twice\colon (a \to ?\, b) \to (a \to ?\, b)$

The effect is that the polymorphic profile of *twice* is annotated with the subtyping constraint $b < a$. Similarly, subtyping constraints may be imposed on polymorphic axioms. It is *not* presently allowed to impose subtyping constraints on type variables appearing as arguments in the declaration of type constructors (since this would ultimately require the introduction of 'dependent kinds'); e.g. in the above context, the declaration **type** $F\ a\ b$ would be illegal. Instances of polymorphic operations with subtyping constraints may be formed only for types that satisfy the constraints (satisfaction of constraints is decidable by means of the syntax-directed set of subtyping rules given in

Appendix D).

Polymorphic operation constants introduced by the keyword **op** are required to be *coherent* under subtyping, i.e. the polymorphic instance for a subtype is required to be mapped to the instance for the supertype under the coercion function, while operators introduced by means of **fun** or **pred** are regarded as non-coherent.

Remark 22 Polymorphic predicates and functions that do not look into the structure of their type arguments, in particular typical polymorphic programs, will be coherent, while polymorphic functions or predicates involving e.g. quantification or equality will often fail to be so. Care should be taken not to accidentally declare functions of the latter kind by **op**, since this will lead to inconsistent specifications.

Subtypes may be *defined* by means of a predicate on the supertype; e.g.

vars $a, b\colon Type$
type $Inj\ a\ b = \{f : a \to b \bullet \forall x, y : a \bullet f(x) = f(y) \Rightarrow x = y\}$

The total function type has the obvious subtype definition built in.

Formally, the subtype relation is a relation between type constructors and pseudotypes. In particular, it is *not* possible to declare composite types to be subtypes of others, nor to declare a subtype relation only for certain instances of a type constructor, e.g. by declaring $NonrepList\ a < List\ a$ only for $a\colon Ord$. A type constructor may be declared a subtype only of pseudotypes of the same raw kind (see below); if a type constructor F is introduced by means of a subtype declaration $F < t$ (such as $NonrepList$ above) and no other kind is declared for F within the same basic specification, then F implicitly inherits the kind of t.

In the meta-theory, we denote the subtype relation by $\leq$. This relation is extended to two preorders $\leq$ and $\leq_*$ on pseudotypes, respectively representing injective and general coercion as suggested in [26], by rules given further below. A typical case where coercions are in general non-injective is coercion by function restriction in subtype relations $b \to c \leq a \to c$ for $a \leq b$. Consequently, application of contravariant type constructors in general is assumed to weaken $\leq$ to $\leq_*$. Signatures are implicitly *embedding closed* [65], i.e. the profiles associated to a given operation constant name are upclosed under $\leq_*$ in the sense that $f : s$ in Σ and $s \leq_* t$ implies $f : t$ in Σ.

The set $\{\pm, +, -, \cdot\}$ of variance annotations, where $\pm$ indicates invariance and $\cdot$ is a placeholder denoting non-variance, will henceforth be denoted by $\mathcal{V}$. The set $\mathcal{V}$ is ordered by taking $\pm$ and $\cdot$ to be the smallest and the greatest element, respectively, and $+$ and $-$ to be mutually incomparable.

The subkind relation is extended by a *variance rule*

$$\frac{}{\mu Kd_1 \to Kd_2 \leq_K \nu Kd_1 \to Kd_2} \quad (\mu, \nu \in \mathcal{V}, \mu \leq \nu),$$

as well as analogous versions of the subkinding rule for constructor kinds (cf. Fig. 2) for covariant and contravariant constructor kinds. The full set of subkinding rules can be found in Appendix A.

Unlike class restrictions, variance annotations are retained in *raw kinds*. This affects the admissibility of kind assignments for type constructors (cf. Sect. 3.1); however, we syntactically relax the previous restrictions as follows: A redeclaration of a type constructor F may omit variances present in the raw kind of the previous declarations and also introduce new variances; these variances are then implicitly combined, and all kinds of F are modified to match the arising raw kind. E.g. the declaration of product types as generic instances of the class *Ord* in Sect. 3.1 is indeed legal, and declares the kind $+Ord \to +Ord \to Ord$ for $\_\times\_$ (since $\_\times\_$ has the built-in kind $+Type \to +Type \to Type$). Similarly, declaring a type to be of kind $+C_1 \to C_2 \to C_3$ and also of kind $C_1 \to -C_2 \to C_3$ results in the kind $+C_1 \to -C_2 \to C_3$. In the same way, the system tries to reconcile different kinding declarations in the case of unions of specifications and instantiations of parametrised specifications. No attempt is made to resolve conflicting variance annotations in left nested occurrences of $\to$; e.g. it is not possible to combine kindings $t\colon (+C_1 \to C_2) \to C_3$ and $t\colon (D_1 \to D_2) \to D_3$. Similar relaxations apply to raw kinds of classes. Formally, we call two kinds Kd_1, Kd_2 *top-level compatible* if $Kd_i = \mu_1^i K_1 \to \cdots \to \mu_n^i K_n \to K$ for suitable kinds K_i, K and $\mu_j^i \in \mathcal{V}$, $i = 1, 2$. We admit redeclarations of types and new subclass declarations $C < Kd$ for existing classes C, provided that the associated raw kinds are top-level compatible with the previous raw kinds; the new raw kind is then the infimum of the newly declared and the previous raw kinds under the subkind relation.

The kinding rules for pseudotypes now require type contexts allowing variance-annotated type variables written in the form $a\colon +Kd$ or $a\colon -Kd$, respectively. Such variance annotations are called *outer variances*. Outer variances may appear also in type variable declarations in pseudotypes and in the actual HasCasl syntax, as already illustrated in the examples above, the effect being a variance declaration for type constructors and type synonyms declared using these type variables. Variance declarations for type synonyms are well-formed only if the associated pseudotypes are kindable by the extended kinding rules below.

The extended kinding rules concern kinding judgements $\Theta \rhd t : Kd$, with Θ a context of variance-annotated type variables, which mean that t depends on the variables in Θ with the indicated variance. (Strictly speaking, pseudotype formation depends also on the declared subtype constraints, but only in the sense that $\lambda\, a : Kd \bullet t$ *is ill-formed if* a *appears in a subtype constraint.*) In type formation, only covariant or non-variant type variables can be introduced (so that e.g. the pseudotype $\lambda\, a\colon -Type \bullet a$ is ruled out). The application rule for type constructors is split into three rules

$$\frac{\Theta \rhd t : Kd_1 \quad \Theta \rhd s : +Kd_1 \to Kd_2}{\Theta \rhd s\ t : Kd_2} \qquad \frac{\Theta^{-1} \rhd t : Kd_1 \quad \Theta \rhd s : -Kd_1 \to Kd_2}{\Theta \rhd s\ t : Kd_2} \qquad \frac{\Theta^{0} \rhd t : Kd_1 \quad \Theta \rhd s : Kd_1 \to Kd_2}{\Theta \rhd s\ t : Kd_2}$$

where the contexts Θ^{-1} and Θ^{0} denote Θ with all outer variances reversed or removed, respectively. For type abstraction, one has rules

$$\frac{\Theta, a : \mu Kd_1 \rhd t : Kd_2 \quad Kd_3 \leq_K Kd_1}{\Theta \rhd \lambda\, a : \mu Kd \bullet t : \nu Kd_3 \to Kd_2}\ (\mu \leq \nu \text{ in } \mathcal{V}).$$

By the above rules, e.g. the pseudotype $\lambda\, a\colon +Type \bullet Pred\ (Pred\ a)$ is of kind $+Type \to Type$, while the pseudotype $\lambda\, a\colon +Type \bullet Pred\ a$ fails to be well-formed. The full set of kinding rules is recorded in Appendix B. By induction over deriviations, one shows that for t, Θ, the set $\{Kd \mid \Theta \rhd t : Kd\}$ is upwards closed w.r.t. subkinding.

Remark 23 It is not in general the case that pseudotypes have smallest kinds w.r.t. the subkind relation. E.g. the user might sensibly assign the additional kind $+Ord \to +Ord \to Ord$ to the product type constructor $\times$; any lower bound of that kind and the built-in kind $+Type \to +Type \to Type$ of $\times$ would then be a subkind of $+Type \to +Type \to Ord$ and hence cannot be expected to be a kind for $\times$. However, the subkinding rule for variances given above introduces a non-trivial ordering also on raw kinds, and the following proposition shows that every pseudotype has a *smallest raw kind*.

Lemma 24 *For all kinds* Kd_1 *and* Kd_2*,* $Kd_1 \leq_K Kd_2$ *implies* $\mathsf{raw}(Kd_1) \leq_K \mathsf{raw}(Kd_2)$*.*

PROOF. Induction over the derivation of $Kd_1 \leq_K Kd_2$. □

Proposition and Definition 25 *For every pseudotype* t *in type context* Θ*, the set* $\{\mathsf{raw}(Kd) \mid \Theta \rhd t : Kd\}$ *has a smallest element, called the* raw kind *of* t *in type context* Θ*.*

(The raw kind may be calculated by recursion along the structure of t.)

PROOF. Induction over the structure of t. The cases for type constructor and variable introduction, as well as type abstraction, are straightforward by Lem. 24. In the cases for type application, the fact is needed that $\nu Kd_2 \to Kd_3 \leq_K \mu Kd_1 \to Kd_4$ implies $Kd_3 \leq_K Kd_4$, which follows from the syntax-directed set of subkinding rules of Appendix C. □

The subtyping relations $\leq$ and $\leq_*$, ranged over by the metavariable $\sqsubseteq$, are defined by the rules of Fig. 7 (a syntax-directed version of the system is given in Appendix D). Subtyping judgements $\Theta; \Lambda \rhd s \sqsubseteq t$ in type context Θ depend on a context Λ of declared subtype constraints of the form $a \leq t$, with a a type variable and t a type in context Θ. Here, Θ is a simple type context without outer variances. The formal difference between the two subtype relations lies in the contravariant application rule, which applies only to $\leq_*$. The subtyping rules assume that all occurring types are well-formed, i.e. kindable in the given context (in particular, the rule for abstractions assumes that a is not mentioned in Λ). The phrase '$F \leq t$ in Σ' means that the type constructor F is declared to be a subtype of t in the signature. Application of the built-in type constructors $\to ?$ etc. is covered by the application rules for arbitrary pseudotypes.

$$\frac{a \leq t \text{ in } \Lambda}{\Theta; \Lambda \rhd a \leq t} \qquad \frac{F \leq t \text{ in } \Sigma}{\Theta; \Lambda \rhd F \leq t} \qquad \frac{\Theta; \Lambda \rhd s \leq t}{\Theta; \Lambda \rhd s \leq_* t} \qquad \frac{}{\Theta; \Lambda \rhd t \leq t} \qquad \frac{\Theta; \Lambda \rhd s \sqsubseteq t \quad \Theta; \Lambda \rhd t \sqsubseteq u}{\Theta; \Lambda \rhd s \sqsubseteq u}$$

$$\frac{\Theta \rhd t : +Kd_1 \to Kd_2 \quad \Theta; \Lambda \rhd s_1 \sqsubseteq s_2}{\Theta; \Lambda \rhd t\ s_1 \sqsubseteq t\ s_2} \qquad \frac{\Theta \rhd t : -Kd_1 \to Kd_2 \quad \Theta \rhd s_2 \leq_* s_1}{\Theta \rhd t\ s_1 \leq_* t\ s_2} \qquad \frac{\Theta; \Lambda \rhd t_1 \sqsubseteq t_2}{\Theta; \Lambda \rhd t_1\ s \sqsubseteq t_2\ s}$$

$$\frac{\Theta, a : Kd; \Lambda \rhd t \sqsubseteq s}{\Theta; \Lambda \rhd \lambda\, a : \mu Kd \bullet t \sqsubseteq \lambda\, a : \mu Kd \bullet s} \ (\mu \in \mathcal{V})$$

Fig. 7. Subtyping rules for pseudotypes (with $\sqsubseteq \in \{\leq, \leq_*\}$)

Lemma 26 *For pseudotypes t_1 and t_2, $\Theta; \Lambda \rhd t_1 \leq_* t_2$ implies that t_1 and t_2 have the same raw kinds.*

Semantically, the interpretation of subtyping is determined by an extension of the translation of polymorphic HasCasl into basic HasCasl to signatures with subtyping, defined as follows.

Definition 27 A *polymorphic* HasCasl *signature* Σ *with subtyping* is defined by extending the notion of polymorphic HasCasl signature (cf. Defn. 11) in the way indicated above: there is additional data in the shape of the subtyping relation $\leq$ between type constructors and pseudotypes, and a *coherence* predicate on the set of polymorphic operations (see above). Moreover, polymorphic operations and axioms are annotated with sets of subtyping constraints of the form described above. For semantic purposes, we admit also subtyping constraints of the form $a \leq_* t$ (such constraints are never generated by user declarations). The restrictions listed above apply, in particular embedding closure. Besides the user-declared symbols, Σ implicitly contains polymorphic operations

$$up : \forall a, b : \mathit{Type}; a \leq_* b \bullet a \to b$$
$$down : \forall a, b : \mathit{Type}; a \leq b \bullet b \to ?a.$$

Similarly, *morphisms* of such signatures are defined by extending the definition of morphism of polymorphic signatures. Signature morphisms map only the user-declared symbols (not the above implicit operations). They are required to preserve coherence and the subtype relation $\leq$, the latter in the sense that subtype declarations are mapped to derivable subtyping judgements. We impose that overloading of symbols is preserved [45], a condition which thanks to embedding closure reduces to the requirement that identically named constants $c : s$ and $c : t$ are mapped to identically named constants whenever $s < t$. Moreover, we require that raw kinds of classes and type constructors are preserved up to top-level compatibility (by Lemma 24 and Proposition 25, it follows already from preservation of the kinding and subclass relations that raw kinds can only decrease under the subkind relation).

Over a polymorphic HasCasl signature Σ with subtyping, we define two kinds of sentences: *explicit coercion* sentences are just the expected polymorphic sentences over Σ, including the built-in symbols, with instances of polymorphic operations admitted only for types satisfying the associated subtyping constraints. *Implicit coercion* sentences additionally may use the above-mentioned subtyping mechanisms, i.e. terms of a subtype can appear wherever terms of a supertype are expected, and downcasts $\_\ \mathit{as}\ s$ and elementhood $\_ \in s$ may be used; however, implicit coercion sentences cannot use the built-in symbols *up* and *down*. Implicit coercion sentences are used in actual specifications, while explicit coercion sentences serve only semantic purposes. Implicit coercion sentences are translated into explicit coercion sentences by

- inserting *up* where terms of a subtype are used in places where terms of the supertype are expected;
- replacing uses of $\in$ with its definition in terms of *down* and definedness;
- replacing downcasts $\_\ \mathit{as}\ s$ with applications of *down*.

The translation of polymorphic signatures into basic signatures is extended by associating to a polymorphic signature Σ with subtyping a basic *theory* $\mathsf{B}(\Sigma)$

as follows. The signature of $\mathsf{B}(\Sigma)$ is defined as before (cf. Defn. 13), except that subtyping constraints are taken into account: a polymorphic operation constant $f : \forall a_1 : Kd_1, \dots, a_n : Kd_n; \Lambda \bullet t$ is instantiated only to those type instances $s_1, \dots, s_n$ that satisfy every subtyping constraint $a_i \sqsubseteq t$ in Λ (with $\sqsubseteq \in \{\leq, \leq_*\}$) in the sense that $(); () \rhd s_i \sqsubseteq t[s_1/a_1, \dots, s_n/a_n]$ is derivable in Σ, using for generalised types the additional rule that $(u.\,\phi) \leq (u.\,\psi)$ if $\forall x : u \bullet \phi\ x \Rightarrow \psi\, x$ is derivable. The axioms of $\mathsf{B}(\Sigma)$ are obtained by translating, in the way described below, the following explicit coercion sentences over Σ:

- coercion from s to t is injective if $s \leq t$, with *down* as a partial left inverse:

$$\begin{aligned}\forall a, b : \mathit{Type}; a \leq b \bullet\ & (\forall x : a \bullet \mathit{down}\ ((\mathit{up}\ x) : b) = x) \wedge \\ & \forall y : b \bullet \mathit{def}\ (\mathit{down}\ y) : a\ \Rightarrow \mathit{up}\ ((\mathit{down}\ y) : a) = y\end{aligned}$$

- subtyping is *coherent*, i.e. coercion functions compose and coercion from a type into itself is the identity:

$$\forall a : \mathit{Type} \bullet \forall x : a \bullet ((\mathit{up}\ x) : a) = x \text{ and}$$
$$\forall a, b, c : \mathit{Type}; b \leq_* c, a \leq_* b \bullet \forall x : a \bullet \mathit{up}\ ((\mathit{up}\ x) : b) = (\mathit{up}\ x) : c.$$

- overloading of operations is compatible with coercion, i.e. for each type context $\Theta = (a_1 : Kd_1; \dots; a_n : Kd_n)$, each polymorphic operation $c : \forall \Theta; \Lambda \bullet s$, and each type t such that $\Theta; \Lambda \rhd s \leq_* t$, there is an axiom

$$\forall \Theta; \Lambda \bullet \mathit{up}\ (c : s) = c : t$$

  (where the profile $c : \forall \Theta; \Lambda \bullet t$ is in Σ by embedding closure);
- correspondingly flagged polymorphic operations are coherent w.r.t. subtyping: if $f : \forall b_1 : Kd'_1, \dots, b_m : Kd'_m \bullet s$ is a coherent polymorphic operation, and for $i = 1, \dots, m$, t_i and u_i are types of kind Kd'_i such that $\Theta; \Lambda \rhd s[t_1/b_1, \dots, t_m/b_m] \leq_* s[u_1/b_1, \dots, u_m/b_m]$, then there is an axiom

$$\forall \Theta; \Lambda \bullet f[u_1, \dots, u_m] = \mathit{up}\ f[t_1, \dots, t_n];$$

- the built-in subtype relations have the expected coercion functions; i.e.

$$\begin{aligned}\forall a, b, c, d : \mathit{Type}; c \leq_* a; d \leq_* b \bullet & \\ & (\forall f : a \to b \bullet (\mathit{up}\ f) : (a \to ?b) = \lambda x : a \bullet\ f\, x) \wedge \\ & (\forall f : a \to ?d \bullet (\mathit{up}\ f) : (c \to ?b) = \lambda x : c \bullet\ \mathit{up}\ (f\ (\mathit{up}\ x))) \wedge \\ & \forall x : c; y : d \bullet (\mathit{up}\ (x, y)) : a \times b = (\mathit{up}\ x, \mathit{up}\ y)\end{aligned}$$

  (note that η does not apply to the right hand side in the first equation, since f has the wrong type).

Finally, explicit coercion sentences are translated into collections of sentences over $\mathsf{B}(\Sigma)$ in the same way as in Defn. 13, with instances restricted to those satisfying the given subtyping constraints. A model of Σ is a model of $\mathsf{B}(\Sigma)$, and such a model satisfies a Σ-sentence if it satisfies all its instances.

Remark 28 The subtyping axioms above imply that the subtype of total functions contains all total functions that live in the partial function type (cf. Definition 3) and that co-contravariant subtype relations for total function types have the right coercion functions, i.e.

$$\forall f : a \to d \bullet (up\ f) : (c \to b) = \lambda\, x : c \bullet!\, up\ (f\ (up\ x)).$$

Remark 29 The presence of the *down* operation implies that subtypes $a \leq b$ are regular subobjects in the categorical models [59].

5 Datatypes

HasCasl supports recursive datatypes in the same style as in Casl. To begin, an unconstrained family of datatypes t_i is declared along with its constructors $c_{ij} : t_{ij1} \to \ldots \to t_{ijk_{ij}} \to t_i$ by means of the keyword **type** in the form

$$\begin{array}{ll}\textbf{type} & t_1 ::= c_{11}\ t_{111}\ \ldots\ t_{11k_{11}}\ \mid\ \ldots\ \mid\ c_{1m_1}\ t_{1m_11}\ \ldots\ t_{1m_1k_{1m_1}} \\ & \ldots \\ & t_n ::= c_{n1}\ t_{n11}\ \ldots\ t_{n1k_{n1}}\ \mid\ \ldots\ \mid\ c_{nm_n}\ t_{nm_n1}\ \ldots\ t_{nm_nk_{nm_n}}\end{array}$$

Here, the t_i may be patterns of the form $C\ a_1\ \ldots\ a_r$, where C is a type constructor and the a_i are type variables, so that C is declared as a polymorphic type. The t_{ijl} are types in the context determined by the C and the a_p. Optionally, selectors $s_{ijl} : t_i \to ?t_{ijl}$ may be declared by writing $(s_{ijl} :?t_{ijl})$ in place of t_{ijl}. All this is just syntactic sugar for the corresponding declarations of types and constants, 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 operation, which implies both induction and a primitive recursion principle. If one of these constraints is used, then recursive occurrences (in the t_{ijl}) of a type constructor C being declared are restricted to the pattern $C\ a_1\ \ldots\ a_r$ 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 the types being declared 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.

For types t_i as above that have only types t_j from the same declaration and types from the local environment as arguments of constructors, the induction axiom states that for any sequence of predicates $P_i : \mathit{Pred}\ t_i$, called the *induction predicates*, the inductive hypothesis implies that $\forall x : t_i.\, P_i(x)$ for all i. Here, the inductive hypothesis expresses that the induction predicates are closed under the constructors in the usual sense. Note that the induction axiom is a higher-order reformulation of the corresponding sort generation constraint in Casl. Unlike in Casl, the induction axioms are however expressible in HasCasl, i.e. generation constraints in HasCasl are just syntactic sugar.

For constructors with composite argument types, the notion of closedness of predicates under the constructor requires extending the induction predicates to *extended induction predicates* P_s on composite types s, as follows.

- *Partial function spaces*:

$$P_{s\to ?t} f \iff \forall x : s.(P_s x \wedge \operatorname{def} f(x)) \Rightarrow P_t f(x).$$

- *Total function spaces*: $P_{s\to t}$ is the restriction of $P_{s\to ?t}$ to $s \to t$.
- *Product types*:

$$P_{s\times t}(x, y) \iff P_s x \wedge P_t y.$$

- For types s from the *local environment*, P_s is taken to be constantly true.
- *Applications* $D\ s_1\ \ldots\ s_q$ of a type constructor D from the local environment to types $s_1, \ldots, s_q$, where at least one s_j contains a recursive occurrence of the t_j: extended induction predicates for such types are required to be closed under all operations with result type $D\ s_1\ \ldots\ s_q$ (which are necessarily newly arising instances of polymorphic operations). Note that they are not in general uniquely defined by this requirement.

Remark 30 If a type constructor D from the local environment has a generation constraint, then of course the closedness requirement on extended induction predicates for applications of D is equivalent to closedness under the operations in the constraint. However, the induction axiom also makes sense if D has no generation constraint; it then states essentially that the types being declared are generated from the reachable part of D. Note that extended induction predicates do not appear in the conclusion of the induction axiom, so that the latter does not imply a sort generation constraint for D.

Generally, every HasCasl specification, in particular every datatype declaration, has a term model [59], and the induction axiom induced by a generatedness constraint is satisfied in the term model. 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 a standard interpretation of functional types (i.e. using the full function space, which cannot be term generated for infinite types).

Finally, note that, due to the flexibility of interpretation of higher types in Henkin models, the higher-order reformulation of generation constraints in HasCasl is weaker than the corresponding generation constraint in Casl, and in particular does not exclude non-standard models. However, proof-theoretically, this difference disappears — at least if the standard Casl proof system with the usual finitary induction rule is used. Only if stronger (e.g. infinitary) forms of induction are used, the difference becomes relevant. It also becomes relevant for monomorphicity: due to possible non-standard interpretations of higher types, even free datatypes (see below) are no longer monomorphic in HasCasl.

Example 31 The following datatype declaration (to be extended by a precise specification of equality on the declared types) might form part of a specification of finite systems with unordered 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 as follows.

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

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 the type *Nat* of natural numbers declared elsewhere), which for *CTree* gives rise to the induction axiom

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

The semantics of free datatypes is determined by a fold operation, i.e. free datatypes are explicitly axiomatised as initial algebras. As indicated above, negative occurrences of the types being declared are forbidden in declarations of free types, i.e.

free type $L ::= abs\ (L \to L)$

is illegal, while

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

is allowed. Thanks to this restriction, we can set about interpreting free datatypes as initial algebras for functors.

To begin, a declaration of datatypes $t_1, \dots, t_n$ as above can be regarded as a fixed point declaration for a family $F = (F_1, \dots, F_n)$ of n-argument type constructors; here, alternatives $A \mid B$ are replaced by sums $A + B$, using a built-in declaration of sum types as in Sect. 3.1. The constructors of the t_i can then be gathered into *structure maps* $c_i : F_i\ t_1\ \dots\ t_n \to t_i$.

We then extend F to a functor, where we view a functor as mapping types to types and functions to functions as in Fig. 4. The action of F on maps is defined by recursion over the structure of F, with the standard clauses for sums, products, function types (where only positive positions appear), and constant types, i.e. types from the local environment. The remaining case in the recursion are types $D\ s_1\ \dots\ s_q$, where D is a type constructor from the local environment. Here, we have to require that the functorial action of D is determined by its specification; that is, the free type is well-formed only if D belongs to the class $\mathit{Functor}_q$, a built-in specification of functors with q arguments that generalises the specification $\mathit{Functor} = \mathit{Functor}_1$ of Fig. 4. (For practical purposes, q can be restricted to small values.)

If the t_i are parametrised over type variables $a_i : \mathit{Type}$, then F is parametrised in the same way. If the a_i appear only in functorial positions, so that the t_i depend functorially on the a_i, then corresponding instances of $\mathit{Functor}_q$ are derived automatically. (Note that this means that the average user will never actually see the classes $\mathit{Functor}_q$ in practice, as instances are generated and exploited automatically for typical sequences of nested datatype declarations.) In order to keep the language design manageable, functorial dependence on variables of higher kinds, although technically possible, is not supported. Cf. [60] for details on the functor mechanism.

The fold operations $fold_i$ for the t_i then have the polymorphic types

$$\begin{aligned} fold_i : \forall b_1, \dots, b_n : Type \bullet (F_1\ b_1\ \dots\ b_n \to b_1) \to \dots \\ \to (F_n\ b_1\ \dots\ b_n \to b_n) \to t_i \to b_i. \end{aligned}$$

(In practice, if F_i is a sum type arising from alternatives, then the argument of type $F_i\ b_1\ \dots\ b_n$ is decomposed into several functions, one for each component of the sum.) The defining property of the fold operations states that, for $b_1, \dots, b_n : Type$, $f_i : t_i \to b_i$, and $d_i : F_i\ b_1\ \dots\ b_n \to b_i$, $i = 1, \dots, n$,

$$f_i = fold_i\ d_1\ \dots\ d_n \text{ for all } i \quad \text{iff} \quad d_i \circ (F_i\ f_1\ \dots\ f_n) = f_i \circ c_i \quad \text{for all } i;$$

i.e. the $fold_i\ f_1\ \dots\ f_n$ constitute the unique F-algebra morphism from the t_i into the F-algebra given by the f_i. Thus, the t_i are determined as the initial F-algebra.

Remark 32 The above requires two warnings. To begin, although we define datatypes as internal initial algebras, they are not in general monomorphic; e.g., the standard definition of the naturals as a free datatype admits non-standard models. This is due to the Henkin semantics — the set of functions to which the fold operation applies does not have a fixed interpretation.

Secondly, unlike in first order CASL, the meaning of **free type** does not coincide with that of the corresponding structured free extension **free { type** ... **}**. The difference is that a free extension also requires all newly arising function types to be freely term generated, which has the undesirable effect of precluding any further function definitions for these types.

Example 33 Consider the following free datatype definitions.

vars $a, b : Type$
free type $List\ a ::= nil \mid cons\ a\ (List\ a)$
free type $Tree\ a\ b ::= leaf\ a \mid branch\ (b \to List\ (Tree\ a\ b))$

For the type constructor *List*, an instance *List* : *Functor* is derived, with *map* defined in the standard way. For *Tree*, we obtain an operation

$$fold : (a \to c) \to ((b \to List\ c) \to c) \to Tree\ a\ b \to c,$$

polymorphic over $c : Type$. This operation is axiomatised as being uniquely determined by the equations

$$\begin{aligned} fold\ f\ g\ (leaf\ x) &= f\ x \\ fold\ f\ g\ (branch\ s) &= g\ (map\ (fold\ f\ g) \circ s). \end{aligned}$$

Remark 34 From the fold operation, one obtains also a primitive recursion principle in the standard way (i.e. by means of a simultanuous recursive definition of the identity). From the latter, in turn, we obtain as a special case a case operator, denoted in the form

$$case\ x\ of\ c\ y_1\ \ldots\ y_l \rightarrow f\ y_1\ \ldots\ y_l \mid \ldots$$

Moreover, free types are generated, i.e. satisfy the induction axiom of Sect. 5.1. In the following, we regard sum types (Fig. 3), which we denote by + in the interest of readability, as free datatypes, and in particular use the case notation for them as well.

Remark 35 Unlike in CASL, declarations of free datatypes in HASCASL are not necessarily a conservative extension of the local environment. Already the naturals may be a non-conservative extension: as discussed in Sect. 3.2, conservative extensions at the second level of the semantics essentially can only introduce names for entities already in the present signature. However, if the naturals, as well as sum types and an initial object as specified in Fig. 3, are already present, then one can construct *finitely branching* free datatypes (thus showing them to be conservative extensions) in a similar way as in standard HOL [49,5] as inductively generated subtypes of a suitable universal type, with some modifications required due to the fact that HASCASL does not impose unique choice (Section 2.3) — essentially, the universal type is a type of trees, represented as partial maps from paths to values (rather than as sets of (node, value)-pairs as in HOL), and annotated explicitly with finite sizes in order to inherit primitive recursion from the naturals. Details are laid out in [60].

6 Recursion

Unlike CASL, HASCASL has a notion of executable specification that includes general recursion and hence possible non-termination, in the style of a strict functional programming language (as laid out in Sect. 2.4, non-strict functions can be modelled as well). This is achieved by explicitly bootstrapping a domain semantics in the style of HOLCF [55]. On the technical side, this requires some adjustments to standard domain theory in order to cope with the austerity of the internal logic; these issues are dealt with in Sect. 6.1. We then go on to discuss initial datatypes in the arising category of domain types, and finally describe how these features are reflected by an appropriate sugaring of the HASCASL syntax.

6.1 Domain Theory in the Internal Logic

We now recast the basics of standard domain theory, phrased in terms of chain-complete partial orders, in the internal logic of HASCASL. The main difficulty here is not so much the intuitionistic aspect (the study of domain theory in toposes goes back at least to [56]), but the fact that due the the absence of unique choice (Section 2.3), 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 $f_i(x)$'. Rather, we have to require existence of suprema of chains in the lifting $?a$ of a domain a; for this purpose, we assume given in this section a type *Nat* of natural numbers.

Definition 36 A partial order a with ordering $\sqsubseteq$ is called a *complete partial order (cpo)* if the lifted type $?a$, equipped with the ordering

$$x \sqsubseteq y \iff (def\, x() \Rightarrow x() \sqsubseteq y()),$$

has suprema of chains and a bottom element. We call chains in $?a$ *partial chains*, as opposed to *total chains*, i.e. chains in a in the usual sense. We say that a cpo a is *pointed* (or a *cppo*) if a has a bottom element. We say that a type a is a *flat cpo* if a is a cpo when equipped with the discrete ordering.

Cpo's can be specified as a class in HasCasl as shown in Fig. 9; the specification imports, besides the natural numbers, a specification of partial orders (Fig. 8), containing in particular the definition of the induced ordering on lifted types. In the discussion below, we denote suprema of (total or partial) chains by $\bigvee$.

Remark 37 Note that the notation for the ordering is changed from $\leq$ to $\sqsubseteq$ in Fig. 9, and the class *Ord* is renamed into *InfOrd* (information ordering) in order to allow the future declaration of the expected instances of the class *Ord*, e.g. the usual ordering on the flat cpo of natural numbers. This nicely illustrates the benefits of combining a class mechanism with Casl's structured specification constructs. In a framework without such constructs, such as Isabelle, it becomes necessary at this point to fully respecify a second copy of the class *Ord* (and indeed this is precisely what happens in Isabelle/HOLCF, where a class `po` of partial orders with ordering $\sqsubseteq$ is newly specified although Isabelle/HOL already includes a class `order` of partial orders with ordering $\leq$).

As in standard domain theory, cppo's in the above sense have least fixed points of continuous endofunctions f, constructed as suprema of (total) chains $(f^n \bot)$. The fixed point operator is denoted by Y. For properties P of $Y\ f$, one has the standard *fixed point induction* principle: if $P\ \bot$, $P\ x \Rightarrow P\ (f\ x)$, and P is *admissible*, i.e. closed under suprema of total chains, then $P\ (Y\ f)$.

To begin, we note that under unique choice, i.e. for coarse types, the definition of cpo coincides with the usual one using total chains.

Proposition 38 *If a coarse type a has suprema of total chains, then a is a cpo in the sense of Fig. 9.*

```
spec  ORD =
      class     Ord {var a: Ord
      fun       _ ≤ __ : Pred (a × a)
      var       x, y, z : a
                • x ≤ x
                • (x ≤ y ∧ y ≤ z) ⇒ x ≤ z
                • (x ≤ y ∧ y ≤ x) ⇒ x = y
                }
      var       a, b : +Ord
      type instance a × b : Ord
      var       x, z : a;  y, w : b
                • (x, y) ≤ (z, w) ⇔ x ≤ z ∧ y ≤ w
      type instance Unit : Ord
                • () ≤ ()
      type instance ?a : Ord
      var       x, y :?a
                • x ≤ y ⇔ (def x() ⇒ x() ≤ y())
```

Fig. 8. Specification of partial orders

PROOF. The supremum of a partial chain (x_i) in a is $\iota x : a \bullet \phi$, where ϕ states that there exists n such that (x_{i+n}) is a total chain with supremum x. The bottom element of $?a = Unit \rightarrow ?a$ is the unique function $\bot$ such that $\neg def\, \bot()$. □

Example 39 The above proposition implies in particular that coarse types become cpos when equipped with the discrete ordering, so that one has the usual concept of flat cpo. There are natural examples of models where *all* types can be made into flat cpos, but also equally natural examples demonstrating that this need not be the case.

As a positive example, consider the quasitopos **ReRe** of reflexive relations [2], whose objects are pairs (X, R) with R a reflexive relation on the set X, and whose morphisms are relation-preserving maps. In a model over **ReRe**, the interpretation $(X_\bot, R_\bot)$ of $?a$ is obtained from the interpretation (X, R) of a by adding a new element $\bot$ to X and putting $xR_\bot\bot$ and $\bot R_\bot x$ for all $x \in X_\bot$. It is easy to check that in his case, a indeed becomes a flat cpo, the crucial point being that the supremum operation is a relation-preserving map from the type of partial chains to $?a$.

A negative example is given by the quasitopos **PsTop** of pseudotopological spaces and continuous functions [28]. A *pseudotopological space* is given in terms of a convergence relation $\rightarrow$ between filters on a set X and points of X, subject to the requirements that $\dot{x} \rightarrow x$, where $\dot{x} = \{A \subseteq X \mid x \in A\}$, and that $\mathfrak{F} \rightarrow x$ iff for all ultrafilters $\mathfrak{U}$ finer than $\mathfrak{F}$ (i.e. $\mathfrak{F} \subseteq \mathfrak{U}$), $\mathfrak{U} \rightarrow x$. A function

```
spec  Recursion = {Ord with Ord ↦ InfOrd, __ ≤ __ ↦ __ ⊑ __} and Nat
      then
      class     Cpo < InfOrd {var a: Cpo
      fun       __ ⊑ __: Pred (a × a)
      op        undefined: ?a
                • ¬def (undefined: ?a)
      type      Chain a = {s: Nat →? a • ∀n: Nat • def s n ⇒ s n ⊑ s (n + 1)}
      fun       sup: Chain a →?a
      var       x :?a; c: Chain a
                • sup c ⊑ [?a]x ⇔ ∀n: Nat • c n ⊑ [?a]x
                }
      class     Cppo < Cpo {var a: Cppo; x: a
      fun       bottom: a
                • bottom ⊑ x }
      class     FlatCpo < Cpo {vars a: FlatCpo; x, y: a
                • x ⊑ y ⇒ x = y }
      vars      a, b: Cpo; c: Cppo; x, y: a; z, w: b
      type instance __ × __: +Cpo → +Cpo → Cpo
      type instance __ × __: +Cppo → +Cppo → Cppo
      type instance Unit: Cppo
      type instance Unit: FlatCpo
      type      a -c->? b = {f: a →?b • ∀c: Chain a •
                sup ((λn: Nat • f (c n)) as Chain b) = f (sup c)}
      type      a -c-> b = {f: a -c->? b • f ∈ a → b}
      type instance __ -c->? __: −Cpo → +Cpo → Cppo
      var       f, g: a -c->? b • f ⊑ g ⇔ ∀x: a • def (f x) ⇒ f x ⊑ g x
      type instance __ -c-> __: −Cpo → +Cpo → Cpo
      var       f, g: a -c-> b • f ⊑ g ⇔ ∀x: a • f x ⊑ g x
      type instance __ -c-> __: −Cpo → +Cppo → Cppo
                • bottom[a -c-> c] = λx: a • bottom[c]
      then      %def
      var       c: Cppo
      fun       Y: (c -c-> c) -c-> c
      var       f: c -c-> c; x: c
                • f (Y f) = Y f
                • f x = x ⇒ Y f ⊑ x
```

Fig. 9. Specification of the cpo structure and the fixed point operator

f between pseudotopological spaces is *continuous* if $f(\mathfrak{F}) \to f(x)$ whenever $\mathfrak{F} \to x$, where $f(\mathfrak{F}) = \{A \mid f^{-1}[A] \in \mathfrak{F}\}$. In **PsTop**, the interpretation $X_\perp$ of $?a$ is obtained from the space X interpreting a type a by adding an element $\perp$ and putting $\mathfrak{F} \to \perp$ for all filters $\mathfrak{F}$ on $X_\perp$, and $\mathfrak{F} \to x$ iff $\mathfrak{F}_X \to x$ for $x \in X$, where $\mathfrak{F}_X = \{A \cap X \mid A \in \mathfrak{F}\}$. Moreover, if types a and b are interpreted by spaces X and Y, respectively, then the function type $a \to b$ is interpreted by the space

$X \to Y$, consisting of the continuous functions $f : X \to Y$, with $\mathfrak{F} \to f$ for a filter $\mathfrak{F}$ on $X \to Y$ iff, whenever $\mathfrak{G} \to x$ in X, then $ev(\mathfrak{F} \times \mathfrak{G}) \to f(x)$, where $\mathfrak{F} \times \mathfrak{G}$ denotes the filter generated by the set $\{A \times B \mid A \in \mathfrak{F}, B \in \mathfrak{G}\}$, and $ev(f, x) = f(x)$. A discrete pseudotopological space is characterised by $\mathfrak{F} \to x$ iff $\mathfrak{F} = \dot{x}$. For discrete spaces X, in particular the natural numbers object (i.e. the set $\mathbb{N}$ equipped with the discrete structure), the above definition of $\mathfrak{F} \to f$ in $X \to Y$ simplifies to $ev_x(\mathfrak{F}) \to f(x)$ for all $x \in X$, where $ev_x(f) = f(x)$.

In models over **PsTop**, no discrete space B with at least two points is a flat cpo in the above sense. The reason is that the supremum map $\sup : \mathit{Chain}\ B \to B_\perp$, where the type $\mathit{Chain}\ B$ of partial chains in B inherits its convergence relation from $\mathbb{N} \to B_\perp$ as a subspace, fails to be continuous. To see this, pick $s \in \mathit{Chain}\ B$ such that s_m is defined for some $m \in \mathbb{N}$. Let $\mathfrak{F}$ be the filter generated by the set

$$\{ev_n^{-1}[C \cup \{\perp\}] \mid C \subseteq B, n \in \mathbb{N}, s_n \in C\}.$$

One can check that $\mathfrak{F} \to s$ in $\mathit{Chain}\ B$. Thus in order for sup to be continuous, one would need $\sup(\mathfrak{F}) \to \sup s = s_m$ in $B_\perp$, whence for all $C \subseteq B$, $\sup^{-1}[C \cup \{\perp\}] \in \mathfrak{F} \iff s_m \in C$ by discreteness of B. In particular, $\sup^{-1}[\{s_m, \perp\}] \in \mathfrak{F}$, i.e. there exist $C_1, \dots, C_k \subseteq B$ and $m_1, \dots, m_k \in \mathbb{N}$ such that

$$\sup{}^{-1}[\{s_m, \perp\}] \supseteq \bigcap ev_{m_i}[C_i \cup \{\perp\}].$$

Now pick $t \in \mathit{Chain}\ B$ such that $t_{m_i} = \perp$ for all i and $\sup t \in B - \{s_m, \perp\}$; then t is contained in the right hand side of the above formula, but not in the left hand side, contradiction.

We now verify that a number of domain theoretic constructions work for our definition of cpo, as claimed by the instance declarations in Fig. 9. Partial suprema have the expected behaviour:

Lemma 40 *Let (x_i) be a partial chain in a. Then $\bigvee_i x_i$ is defined iff $\exists n \bullet \mathit{def}\ x_n$.*

PROOF. The 'if' direction is trivial. Concerning 'only if', just note that

$$(\bigvee x_i)\ \mathit{res}\ \exists n \bullet \mathit{def}\ x_n$$

is an upper bound of (x_i). □

According to Fig. 9, a partial function between cpo's is *continuous* iff it preserves suprema of partial chains. This is equivalent to the standard definition in terms of Scott open domains of definition and preservation of suprema of total chains:

Definition 41 A predicate P : *Pred* a is called *Scott open* if P is upclosed, i.e. $P\ x$ and $x \sqsubseteq y$ imply $P\ y$, and $P\ \bigvee x_i$ for a total chain (x_i) implies $\exists n \bullet P\ x_n$.

Proposition 42 *Let a and b be cpo's, and let $f : a \to ?b$. Then f is continuous iff the predicate $P = \lambda x : a \bullet$ def $f\ x$ is Scott open, f is monotone on P, and f preserves suprema of total chains (x_i) in the sense that if $f\ (\bigvee_i x_i)$ is defined then there exists m such that $f\ x_m$ is defined and $f\ \bigvee_i x_i = \bigvee_i f\ x_{i+m}$.*

PROOF. *'Only if':* If $f\ x$ is defined and $x \sqsubseteq y$, then we have a chain x_i recursively defined by $x_0 = x$ and $x_{i+1} = y$. Thus $f\ y = f\ \bigvee_i x_i = \bigvee_i f\ x_i \geq f\ x$ in $?b$, so that $f\ y$ is defined. This proves monotonicity of f and the first part of Scott openness. For the second part of the latter and preservation of suprema of total chains, let (x_i) be a total chain such that $f\ \bigvee_i x_i$ is defined. By continuity, $\bigvee_i f\ x_i$ is defined and equal to $f\ \bigvee_i x_i$, so that by Lem. 40, there exists m such that $f\ x_m$ is defined; then $f\ \bigvee_i x_i = \bigvee f\ x_i = \bigvee f\ x_{i+m}$.

'If': Let (x_i) be a partial chain. We have to prove the strong equation $f\ \bigvee x_i = \bigvee f\ x_i$. To begin, assume that $f\ \bigvee_i x_i$ is defined. Then $\bigvee_i x_i$ is defined, so that by Lem. 40, there exists m such that x_m is defined. Then (x_{i+m}) is a total chain and $\bigvee x_{i+m} = \bigvee x_i$; hence we have n such that $f\ x_{i+m+n}$ is defined and $f\ \bigvee x_i = \bigvee f\ x_{i+m+n} = \bigvee f\ x_i$.

Conversely, let $\bigvee f\ x_i$ be defined. By Lem. 40, we have n such that $f\ x_n$ is defined. Since P is upclosed, it follows that $f\ \bigvee x_i$ is defined. □

Proposition 43 *Let a and b be cpo's. Then $a \times b$, equipped with the componentwise ordering, is a cpo.*

PROOF. Let (z_i) be a partial chain in $a \times b$. Then $(\mathsf{fst}\ z_i)$ and $(\mathsf{snd}\ z_i)$ are partial chains in a and in b, respectively. We thus obtain $\bigvee z_i$ as

$$(\bigvee \mathsf{fst}\ z_i, \bigvee \mathsf{snd}\ z_i).$$

□

Definition 44 We say that a subtype b of a cpo a is a *sub-cpo* of a if the subtype $?b$ of $?a$ is closed under suprema of chains.

Remark 45 It is automatically the case that for a subtype b of a cpo a, $?b$ inherits the bottom element $\bot$ of $?a$, namely as $\lambda \bullet \bot()$ *as* b (cf. also Rem. 29).

Proposition 46 *Let a and b be cpos. Then the type $a \xrightarrow{c} ?b$ of continuous partial functions is a cppo when equipped with the componentwise ordering. The type $a \xrightarrow{c} b$ of continuous total functions is a sub-cpo of $a \xrightarrow{c} ?b$.*

PROOF. Let (f_i) be a partial chain in $a \xrightarrow{c} ?b$. Then $(f_i(x))$ is a partial chain for all x, so that we obtain $\bigvee f_i$ as

$$\left(\lambda x \bullet \bigvee f_i(x)\right) \; res \; \exists i.\, def\, f_i.$$

It is clear that we can use the same definition for partial chains in $a \xrightarrow{c} b$. The bottom element of $a \xrightarrow{c} ?b$ is $\lambda x \bullet \bot$. □

Proposition 47 *The unit type is a cpo.*

PROOF. The type $?Unit = Logical$ is (internally) even a complete lattice.□

Corollary 48 *If a is a cpo, then $?a$ is a cppo.*

In general, the sum of two cpos is not again a cpo: as shown in Example 39, even $Bool = Unit + Unit$ need not be a (flat) cpo. However, we have

Lemma 49 *In the presence of sum types (cf. Fig. 3), cpos are stable under sums of partial orders, i.e. the sum $a + b$ of two cpos a and b is again a cpo when equipped with the sum ordering, iff Bool is a flat cpo.*

PROOF. 'Only if' is trivial. To prove 'if', let a and b be cpos. Define a function $isLeft : a + b \to Bool$ by

$$isLeft\ z = case\ z\ of\ inl\ x \to True \mid inr\ y \to False.$$

Let s be a partial chain in $a + b$. Then

$$\bigvee s_n = if\ \bigvee isLeft\ s_n\ then\ inl\left(\bigvee outl\ s_n\right)\ else\ inr\left(\bigvee outr\ s_n\right).$$

□

The precondition of the above lemma has a natural sufficient condition:

Lemma 50 *If Nat is a flat cpo, then so is Bool.*

PROOF. *Bool* is isomorphic to the subtype $\{0, 1\}$ of *Nat*. □

For use with the concept of recursion laid out in the previous section, HASCASL offers suitable cpo versions of free datatypes. These are declared in otherwise the same syntax as standard free types by means of the keyword **free domain**. Like in the case of free types, the semantics of free domains is defined by means of a fold operation, which however specifies the interpretation of the free domain to be an initial algebra in the above-defined category of internal cpos — i.e. the fold operation applies only to algebras which are continuous functions on types a of class *Cpo*, returns a continuous function from the initial algebra to a, and is itself continuous. E.g. the specification

var $a : Cpo$
free domain $List\ a ::= nil \mid cons\ a\ (List\ a)$

induces an operation

$$fold : c \xrightarrow{c} (a \xrightarrow{c} c \xrightarrow{c} c) \xrightarrow{c} List\ a \xrightarrow{c} c,$$

polymorphic over $c : Cpo$.

The question then arises whether a conservativity result analogous to the one for free types holds for free domains. The answer is positive in case the types of constructor arguments are either types from the same declaration of mutually recursive types or cpos from the local environment; this is established by defining a suitable cpo structure on the standard free datatype for the same constructor signature; cf. the forthcoming extended version of [60] for details. An interesting open problem is whether the result can be extended to types t with non-strict constructors, i.e. constructors with arguments of type $?t$.

6.3 Programming in HASCASL

General recursive function definitions with a cpo-based fixed point semantics may be written in HASCASL as recursive equations in the standard functional programming style, marked by the keyword **program**; these are implicitly translated into the corresponding fixed point terms. An explicit import of the specification RECURSION is not required. A program block is written as a sequence of so-called pattern equations PE_i in the form

program $\{PE_1 \ldots PE\_n\ \}$

A pattern equation PE_i has as its left hand side a *pattern* and as its right-hand side an arbitrary term. A pattern is an application of a function being recursively defined to argument terms. In the simplest case, the argument terms are applications of constructors to variables; however, more complex argument patterns including nested patterns, wild cards, tuple patterns, and even Haskell-style as-patterns [50] are also admitted. Variables in patterns need not be explicitly declared; their type is inferred. It is statically checked that all involved types are cpo-types; the program block is ill-formed if this check fails. All occurring λ-abstractions, implicit or explicit, are equipped with a downcast to the appropriate continuous function type (so that the user does not have to write these casts explicitly). By consequence, recursively defined functions are undefined if one of the functions involved in their definition fails to be continuous (sufficient criteria for continuity can be statically checked). Recursive functions on free datatypes can be defined by giving a recursive equation for each constructor. This is coded by means of the case operator; an attempt to use this mechanism for non-free datatypes (which do not have case operators) makes the specification ill-formed. On missing constructor patterns, functions are implicitly undefined; in this case, a warning ('non-exhaustive match') is produced.

As a simple example, Fig. 10 shows an implementation of an interpreter for an abstract imperative core language, where programs are regarded as partial functions on a type $s : Cpo$ of states. The program block is translated into a definition of *eval* as a least fixed point of a continuous functional on the type $Prog \xrightarrow{c} s \xrightarrow{c} ?s$.

spec INTERPRETER = SUMS **then**
  var $s : Cpo$
  free domain $Prog\ s ::= skip \mid basic\ (s \xrightarrow{c} ?\, s)$
    $\mid seq\ (Prog\ s)\ (Prog\ s)$
    $\mid if\ (s \xrightarrow{c} Bool)\ (Prog\ s)\ (Prog\ s)$
    $\mid while\ (s \xrightarrow{c} Bool)\ (Prog\ s)$
  op $eval : (Prog\ s) \xrightarrow{c} s \xrightarrow{c} ?\, s$
  program
    $eval\ skip\ s = s$
    $eval\ (basic\ f)\ s = f\ s$
    $eval\ (seq\ p\ q)\ s = eval\ q\ (eval\ p\ s)$
    $eval\ (if\ b\ p\ q)\ s = if\ b\ s\ then\ eval\ p\ s\ else\ eval\ q\ s$
    $eval\ (while\ b\ p)\ s =$
      $if\ b\ s\ then\ eval\ (while\ b\ p)\ (eval\ p\ s)\ else\ s$

Fig. 10. Programming an interpreter for a simple abstract language in HASCASL

7 HasCasl in the Heterogeneous Tool Set

Tool support for HasCasl is implemented within the framework of the Bremen heterogeneuous tool set Hets [44]. This framework is centered around a *logic graph* in which logics, formalised as *institutions* [20], appear as nodes and logic translations, formalised as *comorphisms*, appear as edges. As a node in the logic graph, HasCasl is equipped with tools for parsing and static analysis. Important translations are an embedding of first order Casl into HasCasl, a connection between HasCasl and the interactive higher order theorem prover Isabelle/HOL, and a mapping of executable HasCasl specifications into Haskell. We briefly discuss these translations in the following.

Morphisms and Comorphisms of Institutions Recall from Definition 16 that an institution consists of a category of signatures, equipped with a set-valued sentence functor, a category-valued contravariant model functor, and a satisfaction relation between models and sentences. We briefly recall some standard notions of translations between institutions [21].

Given institutions I and J, an *(institution) comorphism* [21] (also called a *plain map of institutions* [35]) $\mu = (\Phi, \alpha, \beta) : I \to J$ consists of

- a functor Φ from the signature category of I into that of J;
- for each signature Σ in I, a *sentence translation* α_Σ taking Σ-sentences to $\Phi(\Sigma)$-sentences, natural w.r.t. signature morphisms in I; and
- for each signature Σ in I, a *model reduction* functor β_Σ taking $\Phi(\Sigma)$-models to Σ-models, again natural w.r.t. signature morphisms in I,

such that the following *satisfaction condition* holds for all signatures Σ in I, every $\Phi(\Sigma)$-model M, and every Σ-sentence ϕ:

$$M \models_{\Phi(\Sigma)} \alpha_\Sigma \phi \iff \beta_\Sigma M \models_\Sigma \phi.$$

If the model reduction functors β_Σ are surjective on models, then we say that μ is *model-expansive*. In this case, μ admits *borrowing* of entailment systems for basic specifications, i.e. for every signature Σ in I, every set Φ of Σ-sentences, and every Σ-sentence ψ, ψ is a consequence of Φ iff $\alpha_\Sigma(\psi)$ is a consequence of $\alpha_\Sigma[\Phi]$ [10] (where 'only if' holds in general). If μ is even *model-bijective*, i.e. if the β_Σ are bijective on models (but not necessarily on model morphisms), then μ admits borrowing of entailment and refinement also for structured specifications excluding free specifications. If the β_Σ are moreover isomorphisms as functors, then μ admits borrowing of entailment and refinement for structured specifications including free specifications (cf. [42] for a proof and a detailed explanation of the terminology).

Morphisms of specifications are defined dually to comorphisms: a morphism $\mu = (\Phi, \alpha, \beta) : I \to J$ between institutions I and J consists of a functor Φ from the signature category of I to that of J, sentence translation functions α_Σ from $\Phi(\Sigma)$-sentences to Σ-sentences, and *model translation* functors β_Σ from Σ-models to $\Phi(\Sigma)$-models for every signature Σ in I, subject to the obvious satisfaction condition. In heterogeneous specifications, comorphisms appear naturally in translations of structured specifications, while morphisms appear naturally in reductions [43].

Embedding CASL into HASCASL Syntactically, HASCASL is essentially a superset of CASL, so that CASL users can upgrade to HASCASL at liberty. Some subtleties are however attached to the semantic basis of this embedding. It is not possible to define an institution comorphism from CASL into HASCASL: generally, there are no comorphisms from extensional institutions into intensional institutions that work by embedding logical syntax. This is due to the satisfaction condition: e.g., disjunction is extensional in CASL, i.e. for a model M and sentences ϕ, ψ, $M \models \phi \vee \psi$ iff $M \models \phi$ or $M \models \psi$. Now if M is a HASCASL model such that $M \models \phi \vee \psi$ but neither $M \models \phi$ or $M \models \psi$ (such models exist), then there is no possible choice for a reduct $\beta(M)$ of M in the extensional source institution, as by the satisfaction condition $\beta(M)$ would also have to satisfy $\phi \vee \psi$ but none of ϕ and ψ.

The solution to this problem is to connect CASL and HASCASL by a network of morphisms and comorphisms, involving the following modifications of HASCASL.

- The institution HASCASL/FOL of *classical first order* HASCASL is obtained by restricting signatures to first order signatures, in which all operations have types of the form $s_1 \to \ldots \to s_n \to t$ or $s_1 \to \ldots \to s_n \to ?t$, where the s_i are basic types and t is either *Unit* or a basic type, and sentences to formulas not involving λ-abstraction, pairing, quantification over non-basic types, or type variables. Models and satisfaction are inherited from the first level of HASCASL, except that we restrict models to those with a boolan algebra (rather than just a Heyting algebra) of truth values; moreover, we relax the notion of model morphism to the standard notion of morphism of partial first-order structures (i.e. we require only the weak homomorphism property, which in particular amounts to preservation of definedness for partial functions and preservation of satisfaction for predicates, while higher order homomorphisms may additionally impose reflection in both cases; in particular, this is always the case for higher order homomorphisms of standard models). The most relevant difference between CASL and HASCASL/FOL is that the latter does not have sort generation constraints as a separate type of sentence.
- The institution $\text{HASCASL}^{\text{c,uc}}$ is obtained from HASCASL by restricting mod-

els to be classical and to satisfy unique choice. (This is essentially the internal logic of boolean toposes.)

- The institution HasCasl$^{\text{ext,uc}}$ is obtained from HasCasl by restricting models to be extensional (Sect.2.2) and to satisfy unique choice. (This is essentially the internal logic of well-pointed toposes.)
- The institution HasCasl$^{\text{std}}$ is obtained from HasCasl by restricting models to be standard (Sect.2.2).

We then have the following morphisms and comorphisms.

- There is an isomorphism Φ from HasCasl/FOL signatures to Casl signatures, and translations α_Σ of Σ-sentences of HasCasl/FOL into $\Phi(\Sigma)$-sentences of Casl; both Φ and α perform only trivial syntactic rearrangements, such as replacing profiles of operators by the corresponding curried function types. In combination with model reductions β_Σ which map a $\Phi(\Sigma)$-model in Casl to the associated standard Σ-model in HasCasl/FOL, we obtain a comorphism (Φ, α, β) : HasCasl/FOL $\to$ Casl (with functoriality of the β_Σ ensured by the relaxation to first-order homomorphisms in HasCasl/FOL). This comorphism fails to be model-expansive and hence does not admit borrowing of entailment; however, the standard (finitary) proof systems for HasCasl/FOL and Casl differ only w.r.t. the absence of sort generation constraints in HasCasl/FOL. Since Φ is an isomorphism, one obtains also a *morphism* $(\Phi^{-1}, \alpha, \beta)$: Casl $\to$ HasCasl/FOL which, intuitively speaking, imports standard models and hides sort generation constraints.
- One has a comorphism Casl $\to$ HasCasl$^{\text{std}}$ which embeds Casl-signatures and standard sentences into HasCasl by acting as an inverse to the corresponding mappings in the comorphism HasCasl/FOL $\to$ Casl above, translates sort generation constraints into induction axioms, and reduces HasCasl models (of essentially first-order signatures) to Casl models by just forgetting higher order structure. Reduction is bijective on models; hence, this comorphism admits borrowing of entailment and refinement for structured specifications not including free specifications.
- Since extensionality implies excluded middle and standard models satisfy unique choice, one has obvious comorphisms HasCasl $\to$ HasCasl$^{\text{c,uc}}$ $\to$ HasCasl$^{\text{ext,uc}}$ $\to$ HasCasl$^{\text{std}}$. These embeddings are isomorphic on signatures and therefore give rise also to morphisms HasCasl$^{\text{std}}$ $\to$ HasCasl$^{\text{ext,uc}}$ $\to$ HasCasl$^{\text{c,uc}}$ $\to$ HasCasl. Additionally, one has a theoroidal comorphism HasCasl$^{\text{c,uc}}$ $\to$ HasCasl which explicitly adds excluded middle and unique choice to each HasCasl signature. None of the translations between HasCasl$^{\text{std}}$, HasCasl$^{\text{ext,uc}}$, and HasCasl$^{\text{c,uc}}$ are model-expansive; nevertheless, the standard (finitary) proof systems for the three logics coincide, so that transitions between HasCasl$^{\text{std}}$ and HasCasl$^{\text{c,uc}}$ are transparent for the user. Model reduction functors are isomorphisms for the comorphism HasCasl$^{\text{c,uc}}$ $\to$ HasCasl, which hence admits

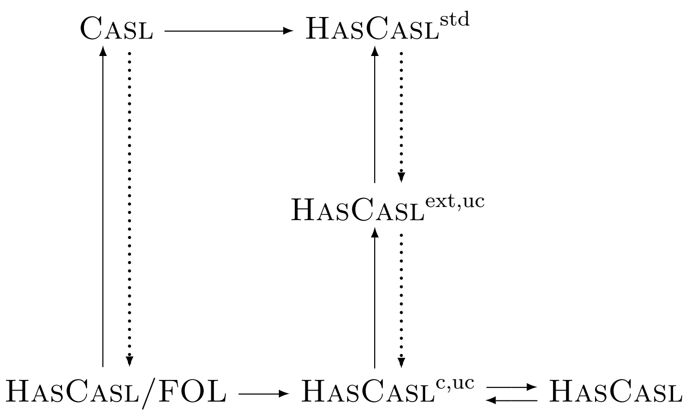

Fig. 11. The CASL-HASCASL network

borrowing of entailment and refinement for structured specifications including free specifications.

- One has a theoroidal comorphism HASCASL/FOL $\to$ HASCASL$^{\text{c,uc}}$ which extends HASCASL/FOL signatures by the axioms of excluded middle and unique choice, and otherwise just acts as a sublogic embedding. In particular, the model reduction functors are isomorphisms, so that this comorphism admits borrowing of entailment and refinement for structured specifications including free specifications.

The ensuing network of morphisms and comorphisms is shown in Fig. 11. Solid lines indicate comorphisms, and dotted lines indicate morphisms.

The comorphism part of the diagram commutes. The sequences of heterogeneous hidings and translations corresponding to the two paths from CASL to HASCASL$^{\text{c,uc}}$ in the diagram, however, are distinct, as the path via HASCASL/FOL syntactically hides sort generation constraints, while the path via HASCASL$^{\text{std}}$ makes them explicit as induction axioms. The path via HASCASL$^{\text{std}}$ will therefore usually be the preferable one; the interest in the path via HASCASL/FOL lies primarily in the fact that it involves only sublogics, rather than fundamental semantic modifications, of HASCASL.

The main question that remains w.r.t. borrowing of entailment systems in the diagram is whether borrowing for structured specifications *including* free specifications is possible between CASL and any of the higher order logics in the diagram. The answer is negative:

Example 51 Consequences of free CASL specifications are not in general preserved by the comorphism into HASCASL$^{\text{std}}$. To see this, consider extending the CASL signature Σ with two sorts s, t and two operations $f, g : s \to t$ to a specification SP with an additional sort v, two constants $c, d : v$, and the axiom

• $(\forall x : s \bullet f(x) = g(x)) \Rightarrow c = d$

In CASL, the free specification $SP' = \Sigma$ **then free** SP has the consequence $c \neq d$, and hence $f \neq g$ (in particular, SP' is not conservative over Σ): SP-models with $f = g$ are never free over their Σ-reducts, since it is always possible to find a homomorphism from their Σ-reduct into the Σ-reduct of an SP-model where $f \neq g$ and $c \neq d$. In HASCASL$^{\mathrm{std}}$, however, the free extension does have models where $f = g$: when f and g are identical functions, then they will remain identical under all higher order homomorphisms. Note that this is a general phenomonon that will occur in any higher order extension of a first order logic that imposes higher order homomorphisms (including intensional higher order logics such as HASCASL, where it may still happen that f and g denote the same element of the function type).

Connecting HASCASL to Isabelle/HOL Proof support for HASCASL is presently based on Isabelle/HOL [46]; the extensible structure of the heterogeneous tool set will however cater for connections to other theorem provers in the future. Indeed the internal logic is to some degree at variance with Isabelle/HOL, as the latter imposes both the law of excluded middle and unique choice (unlike e.g. the Coq proof assistant [13], which therefore presents a future option for alternative proof support). Thus, the comorphism into Isabelle is actually defined on the sublogic HASCASL$^{\mathrm{c,uc}}$ of HASCASL (see above). The mapping from HASCASL$^{\mathrm{c,uc}}$ into Isabelle/HOL codes out subtypes by making subtype injections explicit, and translates partial function types $a \to ?b$ into total function types `'a => 'b option`, where `'b option` is a built-in Isabelle/HOL datatype extending `'b` by an additional element. Finally, the problem that Isabelle does not offer direct support for constructor classes is solved by mappying type constructor variables to loose type constructors, and axioms or implied formulas involving such variables to axioms or proof obligations, respectively (by Theorem 20, this yields a complete proof principle for the extension semantics).

Animating executable HASCASL specifications in Haskell The executable sublogic EXECHASCASL of HASCASL is defined as follows: a specification belongs to EXECHASCASL if

- all types it declares are of class *Cpo*,
- all its operation declarations are coherent (i.e. use the keyword **op**) and have types of class *Cpo* (in particular, such types involve only the continuous function type constructors)
- its only axioms are pattern equations in program blocks. Program blocks are expected to define values of types of class *Cppo*, typically partial continuous

functions. Program blocks defining total continuous functions are admitted; they are interpreted as downcasts of fixed points in the partial function types and generate a corresponding termination proof obligation.

It is straightforward to translate execHasCasl specifications into Haskell programs, noting that standard λ-abstractions in HasCasl need to be translated into strict λ-abstractions in Haskell, while HasCasl terms of the form $\lambda x :?a \bullet t$ can be translated into standard (non-strict) Haskell-terms `\ x -> t`; otherwise, the only real problems that arise concern the management of name spaces. Thus, HasCasl supports a development methodology where abstract requirement specifications are successively refined into design specifications and eventually executable specifications, which are then automatically translated into Haskell.

8 Monads for Functional-Imperative Programming

We now proceed to establish specification support for imperative constructs, which are embedded into functional programming languages such as Haskell [50] by means of monadic encapsulation of side-effects in the spirit of the seminal paper [40]. We give a brief introduction to the basic concepts of monad-based functional-imperative programming, and then introduce a generic monad-based Hoare logic [62]. (Monad-based Hoare logics discussed in [17,40] are specific for particular types of state monad).

Intuitively, a monad associates to each type A a type TA of computations of type A; a function with side effects that takes inputs of type A and returns values of type B is, then, just a function of type $A \to TB$. This approach abstracts away from particular notions of computation such as store, non-determinism, non-termination etc.; a surprisingly large amount of reasoning can in fact be carried out independently of the choice of such a notion.

More formally, a monad on a given category $\mathbf{C}$ can be defined as a *Kleisli triple* $(T, \eta, \_^*)$, where $T : \mathrm{Ob}\,\mathbf{C} \to \mathrm{Ob}\,\mathbf{C}$ is a function, the *unit* η is a family of morphisms $\eta_A : A \to TA$, and $\_^*$ assigns to each morphism $f : A \to TB$ a morphism $f^* : TA \to TB$ such that

$$\eta_A^* = id_{TA}, \quad f^*\eta_A = f, \quad \text{and} \quad g^*f^* = (g^*f)^*.$$

This description is equivalent to the more familiar one via an endofunctor with unit and multiplication [34].

In order to support a language with finitary operations and multi-variable contexts (see below), one needs a further technical requirement: a monad is called *strong* if it is equipped with a natural transformation

$$t_{A,B} : A \times TB \to T(A \times B)$$

called *tensorial strength*, subject to certain coherence conditions (see e.g. [40]); this is equivalent to enrichment of the monad over **C** (see discussion and references in [40]).

Example 52 [40] Computationally relevant monads on **Set** (since strength is equivalent to enrichment, all monads on **Set** are strong) include

- stateful computations with possible non-termination: $TA = (S \to ?(A \times S))$, where S is a fixed set of states and $\_ \to ? \_$ denotes the partial function type;
- non-determinism: $TA = \mathcal{P}(A)$, where $\mathcal{P}$ denotes the power set functor;
- exceptions: $TA = A + E$, where E is a fixed set of exceptions;
- interactive input: TA is the smallest fixed point of $\gamma \mapsto A + (U \to \gamma)$, where U is a set of input values.
- non-deterministic stateful computations: $TA = (S \to \mathcal{P}(A \times S))$, where, again, S is a fixed set of states;
- continuations: $TA = (A \to R) \to R$, where R is a type of results.

In order to accommodate binding also of programs $A \to ?TB$ with intrinsic non-termination, we require a slightly modified specification of monads as already shown in Fig. 5, the main subtlety arising from partiality being the treatment of the first unit law (cf. [63]). The notation is (almost) identical to the one used in Haskell, i.e. the unit is denoted by *ret*, and the binding operator $\_ \gg= \_$ denotes, in Kleisli triple notation, the function $(x, f) \mapsto f^*(x)$. There is a built-in sugaring for the binding operation in the form of a Haskell-style *do*-notation: for monadic expressions p and q,

$$do\ x \leftarrow p;\ q$$

abbreviates $p \gg= \lambda\, x \bullet q$. (This is essentially the same as Moggi's *let*-notation [40].) The intuition behind this notation is that the computations p and q are performed sequentially, with the result of p being bound to x and passed on to q (an expression which may contain the variable x).

As an example of an instance of the class $Monad$, a specification of the state monad is shown in Fig. 12. Note that it is only thanks to the treatment of partial functions in the specification of monads that the state monad is really an instance of $Monad$, since stricter versions of the first monad law fail to hold for the state monad (cf. [63] for a more detailed discussion). Monads specified in HasCasl in the style of Fig. 12 are automatically strong, because the operations of the monad are internalised as functions (recall that strength is equivalent to enrichment).

In the do-notation, the axioms of Fig. 5 take the following shape: Binding is

spec	STATE = MONAD **then**	
	var	$state : Type$
	type instance $ST\ state : Monad$	
	vars	$a, b : Type$
	type	$ST\ state\ a := state \rightarrow ?\ (a \times state)$
	var	$x : a;\ p : ST\ state\ a;\ q : a \rightarrow ?\ ST\ state\ b$
		$\bullet\ (ret\ x) : ST\ state\ a = \lambda s : state \bullet (x, s)$
		$\bullet\ (p \gg= q) = \lambda s1 : state \bullet let\ (z, s2) = p\ s1\ in\ q\ z\ s2$

Fig. 12. Specification of the state monad

associative, i.e. one has

$$do\ y \leftarrow (do\ x \leftarrow p;\ q);\ r = do\ x \leftarrow p;\ do\ y \leftarrow q;\ r$$

if r does not contain x. Moreover, we have *unit laws* stating that

$$(do\ x \leftarrow ret\ a;\ p) = p[x/a], \text{ whenever } p[x/a] \text{ is defined,}$$
$$(do\ y \leftarrow q; x \leftarrow ret\ a;\ p) = do\ y \leftarrow q;\ p[x/a], \text{ and}$$
$$(do\ x \leftarrow p;\ ret\ x) = p.$$

Thanks to associativity, one may safely denote nested *do* expressions like $do\ x \leftarrow p;\ do\ y \leftarrow q;\ \dots$ by $do\ x \leftarrow p; y \leftarrow q;\ \dots$. Repeated nestings such as $do\ x_1 \leftarrow p_1, \dots, x_n \leftarrow p_n;\ q$ are somewhat inaccurately denoted in the form $do\ \bar{x} \leftarrow \bar{p};\ q$. Term fragments of the form $\bar{x} \leftarrow \bar{p}$ are called *program sequences*. Bound variables x_i that are not used later may be omitted from the notation. Terms are generally formed in a context $\Gamma = (x_1 : s_1, \dots, x_n : s_n)$ of variables with assigned types. Following [40], we shall refer to this notation and the associated calculus as the *computational meta-language*.

On top of a monad, one can generically define control structures such as if-then-else or the while loop. The if-then-else construct is defined by

$$if\ b\ then\ p\ then\ q = do\ a \leftarrow b;\ if\ a\ then\ p\ then\ q$$

for $b : T\,Bool$ and $p, q : TA$, where the stateless if-then-else construct on the right hand side is the one defined in Fig. 3. Loops require general recursion, so that one has to restrict to monads that allow lifting a cpo structure on A to a cpo structure on the type TA of computations in such a way that the monad operations become continuous. This is an example of a constructor subclass; the corresponding specification of *cpo-monads* is shown in Fig. 13. Most relevant computational monads including the ones in Example 52 above can be made instances of this subclass.

As an example of a recursively defined control structure we introduce an iteration construct which generalises the while loop by extending it with a default

$$
\begin{array}{|lll|}
\hline
\textbf{spec} & \multicolumn{2}{l|}{\text{CPOMONAD} = \text{RECURSION } \textbf{and} \text{ MONAD } \textbf{then}} \\
& \textbf{class} & \mathit{CpoMonad} < \mathit{Monad}\ \{ \\
& \textbf{vars} & m : \mathit{CpoMonad};\ a : \mathit{Cpo} \\
& \textbf{type} & m\ a : \mathit{Cpo} \\
& \textbf{ops} & \_ \gg= \_ : m\ a \xrightarrow{c} (a \xrightarrow{c}?\ m\ b) \xrightarrow{c}?\ m\ b; \\
& & \mathit{ret} : a \xrightarrow{c} m\ a \\
& & \} \\
\hline
\end{array}
$$

Fig. 13. The constructor subclass of cpo-monads

return value (the while loop as programmed e.g. in the Haskell prelude returns only a unit value) which is fed through the iteration. The (executable) specification of the iteration construct is shown in Fig. 14. Note that the while loop is just iteration with a dummy return value.

$$
\begin{array}{|lll|}
\hline
\textbf{spec} & \multicolumn{2}{l|}{\text{ITERATION} = \text{CPOMONAD } \textbf{and} \text{ BOOL } \textbf{then}} \\
& \textbf{vars} & m : \mathit{CpoMonad};\ a : \mathit{Cpo} \\
& \textbf{op} & \mathit{iter} : (a \xrightarrow{c} m\ \mathit{Bool}) \xrightarrow{c} (a \xrightarrow{c}?\ m\ a) \xrightarrow{c} a \xrightarrow{c}?\ m\ a \\
& \multicolumn{2}{l|}{\textbf{program}} \\
& & \mathit{iter\ test\ f\ x} = \\
& & \mathit{do}\ b \leftarrow \mathit{test}\ x \\
& & \quad \mathit{if}\ b\ \mathit{then} \\
& & \qquad \mathit{do}\ y \leftarrow f\ x \\
& & \qquad\quad \mathit{iter\ test\ f\ y} \\
& & \qquad \mathit{else\ ret}\ x \\
& \textbf{op} & \mathit{while}\ (b : m\ \mathit{Bool})\ (p : m\ \mathit{Unit}) : m\ \mathit{Unit} = \\
& & \qquad\qquad\qquad \mathit{iter}\ (\lambda x \bullet b)\ (\lambda x \bullet p)\ () \\
\hline
\end{array}
$$

Fig. 14. The iteration control structure

9 Generic Side Effect Freeness and Global Evaluation

In preparation for the formulation of the monad-based Hoare calculus, we now summarise material from [63] on generic notions of side-effect freeness, to be required of stateful formulas appearing as pre- and postconditions, and *global evaluation formulas* (called *global dynamic judgements* in [63]). We fix the notation for monads introduced in the previous section (T, η etc.) throughout the remaining development.

Definition 53 [18,67] A program p is called *stateless* if it factors through *ret*, i.e. if it is just a value inserted into the monad. A program p is called *discardable* if

$$(\mathit{do}\ y \leftarrow p;\ \mathit{ret}\ *) = \mathit{ret}\ *,$$

where $*$ is the unique element of the unit type. A program p is called *copyable* if

$$(do\ x \leftarrow p; y \leftarrow p;\ ret\ (x, y)) = do\ x \leftarrow p;\ ret\ (x, x).$$

Moreover, programs p, q *commute* if

$$(do\ x \leftarrow p; y \leftarrow q;\ ret\ (x, y)) = do\ y \leftarrow q; x \leftarrow p;\ ret\ (x, y).$$

Proposition and Definition 54 *[63] Let p be discardable and copyable. Then p commutes with all discardable copyable programs iff p commutes with all discardable copyable Logical-valued programs. In this case, p is called* deterministically side-effect free *(dsef). The subtype of TA formed by the dsef computations will be denoted by DA throughout.*

For details on the relation between the various notions above, cf. [18,63]. Here, we need mainly the notion of deterministic side-effect freeness. Stateless programs are dsef, but not conversely. For example, in the state monad, statelessness means that the program neither changes nor reads the state (p is stateless iff p *exists* in the sense of [40]). Contrastingly, we have

Example 55 A program p is dsef

- in the state monad iff p terminates and does not change the state (p may however read the state);
- in the non-determinism monad iff p has a unique outcome;
- in the exception monad iff p terminates normally;
- in the interactive input monad iff p never reads any input;
- in the non-deterministic state monad iff p does not change the state and always has a unique outcome;
- in the continuation monad (over **Set**) iff p is stateless.

The definition of the semantics of the Hoare logic is based on *global evaluation formulas* $[\bar{x} \leftarrow \bar{p}]\, \phi$, where $\bar{x} \leftarrow \bar{p}$ is a program sequence and $\phi : Logical$ is a formula which may contain $\bar{x}$. Intuitively, $[\bar{x} \leftarrow \bar{p}]\, \phi$ states that ϕ holds for the result $\bar{x}$ after execution of $\bar{x} \leftarrow \bar{p}$ from any initial state. Formally, $[\bar{x} \leftarrow \bar{p}]\, \phi$ abbreviates

$$(do\ \bar{x} \leftarrow \bar{p};\ ret\ (\bar{x}, \phi)) = do\ \bar{x} \leftarrow \bar{p};\ ret\ (\bar{x}, \top)$$

(a strong equation). The degenerate case $[\,]\, \phi$ is (by injectivity of *ret* as specified in Fig. 5) equivalent to ϕ; we shall silently identify the two formulas.

Remark 56 The above semantics of global evaluation formulas is close to Moggi's global semantics of evaluation logic [41] (but not at all to the original local semantics as defined in [54]).

Example 57 In the monads of Example 52, satisfaction of $[x \leftarrow p]\, \phi$, where

$p : TA$, amounts to the following (we freely omit semantic brackets from the notation):

- *states*: terminating execution of p from any initial state yields a result x satisfying ϕ;
- *non-determinism*: all values x in $p \in \mathcal{P}(A)$ satisfy ϕ;
- *exceptions*: if p terminates normally, then its result x satisfies ϕ;
- *interactive input*: the value x eventually produced by p after some combination of inputs always satisfies ϕ;
- *non-deterministic state monad*: all possible results x obtained by execution of p from any initial state satisfy ϕ;
- *continuations*: for $k : A \to R$, $p\ k$ depends only on the restriction of k to the set of values $x : A$ satisfying ϕ.

Figure 15 shows a number of proof rules for global evaluation formulas. In the present setting, this should be regarded as a collection of lemmas rather than as a formally limited calculus; a slightly different calculus for a clearly separated definition of *global evaluation logic* is given in [22]. Double lines indicate that a rule works in both directions. The set of free variables of p is denoted by $FV(p)$. The rules (pre) and (wk) use explicit quantification to enforce the usual variable condition stating that certain variables do not occur freely in assumptions. Soundness of the rules has been established in [63]. The rules shown below the dotted line in Fig. 15 are derivable from the others. We will refer to proofs using only the rules ($\wedge$I) and (wk) as *propositional reasoning.*

For discardable programs, copyability can be reformulated as a global evaluation formula:

Lemma 58 *A discardable program p is copyable iff*

$$[\![x \leftarrow p; y \leftarrow p]\!]\, x = y.$$

Convention 59 Dsef terms can be handled notationally in a more relaxed way, as it is immaterial how often and in which order they are evaluated as long as no other programs interfere. We thus allow dsef programs of type DA to occur in places where a term of type A is expected in programs and formulas. More precisely, if $\bar{x} = (x_1, \dots, x_n)$ is a list of variables of types $A_1, \dots, A_n$ and q is a program, then the program $q[\bar{p}/\bar{x}]$ obtained by substituting terms $p_i : DA_i$ for the x_i is defined as *do* $\bar{x} \leftarrow \bar{p};\ q$, with well-definedness guaranteed by deterministic side-effect freeness (cf. [63] for details). Similarly, $[\![\bar{y} \leftarrow \bar{q}]\!]\, \phi[\bar{p}/\bar{x}]$ abbreviates $[\![\bar{y} \leftarrow \bar{q}; \bar{x} \leftarrow \bar{p}]\!]\, \phi$. Note that this includes the case that $\bar{y} \leftarrow \bar{q}$ is the empty sequence. Since we further identify $[\![]\!]\, \phi$ with ϕ, e.g. $\phi \Rightarrow \psi$ abbreviates $[\![a \leftarrow \phi; b \leftarrow \psi]\!]\, (a \Rightarrow b)$ for $\phi, \psi : D\Omega$. Ambiguities may arise from polymorphic predicates and operations such as equality, e.g. in the equation $p = q$, with $p, q : DA$. In such cases, we will disambiguate formulas by explicit

$$(\wedge\mathbf{I})\ \frac{[\bar{x} \leftarrow \bar{p}]\,\phi \quad [\bar{x} \leftarrow \bar{p}]\,\xi}{[\bar{x} \leftarrow \bar{p}]\,(\phi \wedge \xi)} \qquad (\mathbf{wk})\ \frac{\forall \bar{x}.\, \phi \Rightarrow \xi \quad [\bar{x} \leftarrow \bar{p}]\,\phi}{[\bar{x} \leftarrow \bar{p}]\,\xi} \qquad (\mathbf{eq})\ \frac{[\bar{x} \leftarrow \bar{p}]\, q_1 = q_2 \quad [\bar{x} \leftarrow \bar{p}; y \leftarrow q_1; \bar{z} \leftarrow \bar{r}]\,\phi}{[\bar{x} \leftarrow \bar{p}; y \leftarrow q_2; \bar{z} \leftarrow \bar{r}]\,\phi}$$

$$(\mathbf{app})\ \frac{[\bar{x} \leftarrow \bar{p}]\,\phi \quad y \notin FV(\phi)}{[\bar{x} \leftarrow \bar{p}; y \leftarrow q]\,\phi} \qquad (\mathbf{dis}_0)\ \frac{[\bar{x} \leftarrow \bar{p}; q]\,\phi \quad q \text{ discardable}}{[\bar{x} \leftarrow \bar{p}]\,\phi} \qquad (\mathbf{pre})\ \frac{\forall x.\, [\bar{y} \leftarrow \bar{q}]\,\phi}{[x \leftarrow p; \bar{y} \leftarrow \bar{q}]\,\phi}$$

$$(\eta)\ \frac{[\bar{x} \leftarrow \bar{p}; y \leftarrow \mathit{ret}\ a; \bar{z} \leftarrow \bar{q}]\,\phi}{[\bar{x} \leftarrow \bar{p}; \bar{z} \leftarrow \bar{q}[a/y]]\,\phi[a/y]} \qquad (\mathbf{ctr})\ \frac{[\ldots; x \leftarrow p; y \leftarrow q; \bar{z} \leftarrow \bar{r}]\,\phi \quad x \notin FV(\phi) \cup FV(\bar{r})}{[\ldots; y \leftarrow (\mathit{do}\ x \leftarrow p;\ q); \bar{z} \leftarrow \bar{r}]\,\phi}$$

....

$$(\mathbf{tau})\ \frac{\forall \bar{x}.\, \phi}{[\bar{x} \leftarrow \bar{p}]\,\phi} \qquad (\mathbf{rp})\ \frac{\forall \bar{x}.\, q_1 = q_2 \quad [\bar{x} \leftarrow \bar{p}; y \leftarrow q_1; \bar{z} \leftarrow \bar{r}]\,\phi}{[\bar{x} \leftarrow \bar{p}; y \leftarrow q_2; \bar{z} \leftarrow \bar{r}]\,\phi} \qquad (\mathbf{dis})\ \frac{[\bar{x} \leftarrow \bar{p}; q; \bar{z} \leftarrow \bar{r}]\,\phi \quad q \text{ discardable}}{[\bar{x} \leftarrow \bar{p}; \bar{z} \leftarrow \bar{r}]\,\phi}$$

Fig. 15. Proof rules for global evaluation formulas

type annotations where necessary; e.g., $p =_A q$ abbreviates $[x \leftarrow p; y \leftarrow q]\, x = y$, while $p =_{DA} q$ is just equality of computations. A single warning is required: rule (app) of Fig. 15 is sound only if the formula ϕ is really stateless.

10 The generic Hoare calculus

We now proceed to describe the generic monad-based Hoare-calculus.

Definition 60 A *Hoare triple*, written $\{\phi\}\ \bar{x} \leftarrow \bar{p}\ \{\psi\}$, consists of a program sequence $\bar{x} \leftarrow \bar{p}$, a precondition $\phi : T\Omega$, and a postcondition $\psi : T\Omega$ (which may contain $\bar{x}$), where ϕ and ψ are deterministically side-effect free. This abbreviates the global evaluation formula

$$[a \leftarrow \phi; \bar{x} \leftarrow \bar{p}; b \leftarrow \psi]\,(a \Rightarrow b).$$

The fact that Hoare triples as just defined mention program *sequences* (rather than just programs) reflects the need to actually reason about results of computations, including intermediate results, as opposed to just about state changes as in the traditional case.

Example 61 A Hoare triple $\{\phi\}\ x \leftarrow p\ \{\psi\}$ holds

- in the state monad iff, whenever ϕ holds in a state s and p terminates in state s' with result x when executed in state s, then ψ holds for x in the state s';
- in the non-determinism monad iff, whenever ϕ is true, then ψ holds for all possible results x of p;
- in the exception monad iff, whenever ϕ holds and p terminates normally, returning x, then ψ holds for x;
- in the interactive input monad iff, whenever ϕ holds and p returns x after reading some sequence of inputs, then ψ holds for x.
- in the non-deterministic state monad iff, whenever ϕ holds in a state s, and p possibly terminates in a state s' with result x, then ψ holds for x in s'.

A set of monad-independent Hoare rules is shown in Fig. 16. The rules (dsef), (wk), (disj), and (conj) apply the notation introduced in Convention 59. In particular, $\phi \Rightarrow \psi$ has the same decoding as the Hoare triple $\{\phi\}\ \{\psi\}$, so that (wk) is actually a special case of the sequential rule (seq). Due to discardability, the decoding of $\phi \Rightarrow \psi$ can be simplified to

$$(do\ a \leftarrow \phi, b \leftarrow \psi;\ ret\ (a \Rightarrow b)) = ret\ \top.$$

In the pre- and postconditions, boolean values b are implicitly converted to Ω as $b = true$, and *formulas of type* Ω *are implicity cast to* $D\Omega$ via *ret* when needed (used in Fig. 16 only for the formula $\bot : \Omega$). Square brackets indicate reasoning with local assumptions, discharged by application of the rule; this occurs only in rule (Y). Universal quantifiers on Hoare triples in premises (rules (seq), (wk), (Y)) are, as already in Fig. 15, just a short way of expressing the variable condition. An exception is the universal quantifier on the assumption in (Y), which means that the derivation may use arbitrary instances of the assumption. Arguments in the calculus using only the rules (Ω), ($\bot$), (wk), (conj), and (disj) are referred to as *propositional reasoning*.

The rule (dsef) applies in particular to stateless programs $p = ret\ a$, for which the postcondition simplifies to $x = a$. Although the classical Hoare calculus does not require the usual introduction and elimination rules for logical connectives, such rules are sometimes convenient (see the example below); we have included introduction rules for conjunction and disjunction. One typical Hoare rule that is missing here is the assignment rule; this rule only makes sense in a more specialised context where some sort of store is present. An example of an extension of the calculus by specialised rules for a particular monad is presented below. Rule (Y) refers to the fixed-point operator Y (cf. Sect. 6); this rule applies only to cpo-monads. Application of the Y operator to F requires implicitly that F has the continuous function type $(A \xrightarrow{c} ?TB) \xrightarrow{c} (A \xrightarrow{c} ?TB)$. From (Y), one derives e.g. a rule for the iteration construct from Sect. 8:

$$(\bot)\ \frac{}{\{\bot\}\ p\ \{\varphi\}} \qquad \textbf{(dsef)}\ \frac{p \text{ dsef}}{\{\phi\}\ x \leftarrow p\ \{\phi \wedge x = p\}}\ (x \notin FV(\phi, p))$$

$$(\Omega)\ \frac{}{\{ret\ \phi\}\ p\ \{ret\ \phi\}} \qquad (\eta)\ \frac{\{\phi\}\ x \leftarrow p; y \leftarrow ret\ a; z \leftarrow q\ \{\psi\}}{\{\phi\}\ x \leftarrow p; z \leftarrow q[a/y]\ \{\psi[a/y]\}}$$

$$\textbf{(seq)}\ \frac{\{\phi\}\ \bar{x} \leftarrow \bar{p}\ \{\psi\} \quad \forall \bar{x} \bullet \{\psi\}\ \bar{y} \leftarrow \bar{q}\ \{\chi\}}{\{\phi\}\ \bar{x} \leftarrow \bar{p}; \bar{y} \leftarrow \bar{q}\ \{\chi\}} \qquad \textbf{(wk)}\ \frac{\{\phi\}\ \bar{x} \leftarrow \bar{p}\ \{\psi\} \quad \phi' \Rightarrow \phi \quad \forall \bar{x} \bullet \psi \Rightarrow \psi'}{\{\phi'\}\ \bar{x} \leftarrow \bar{p}\ \{\psi'\}}$$

$$\textbf{(ctr)}\ \frac{\{\phi\}\ \ldots; x \leftarrow p; y \leftarrow q; \bar{z} \leftarrow \bar{r}\ \{\psi\}}{\{\phi\}\ \ldots; y \leftarrow (do\ x \leftarrow p;\ q); \bar{z} \leftarrow \bar{r}\ \{\psi\}}\ (x \notin FV(\bar{r}, \psi))$$

$$\textbf{(if)}\ \frac{\{\phi\}\ a \leftarrow b\ \{if\ a\ then\ \psi\ else\ \xi\} \quad \{\psi\}\ x \leftarrow p\ \{\chi\} \quad \{\xi\}\ x \leftarrow q\ \{\chi\}}{\{\phi\}\ x \leftarrow if\ b\ then\ p\ else\ q\ \{\chi\}}\ (y \notin FV(\psi, \xi))$$

$$\textbf{(conj)}\ \frac{\{\phi\}\ \bar{x} \leftarrow \bar{p}\ \{\psi\} \quad \{\phi\}\ \bar{x} \leftarrow \bar{p}\ \{\chi\}}{\{\phi\}\ \bar{x} \leftarrow \bar{p}\ \{\psi \wedge \chi\}} \qquad \textbf{(disj)}\ \frac{\{\phi\}\ \bar{y} \leftarrow \bar{q}\ \{\chi\} \quad \{\psi\}\ \bar{y} \leftarrow \bar{q}\ \{\chi\}}{\{\phi \vee \psi\}\ \bar{y} \leftarrow \bar{q}\ \{\chi\}}$$

. . . .

$$\textbf{(Y)}\ \frac{\begin{matrix}\left[\forall y \bullet \{\phi\}\ x \leftarrow p\ y\ \{\psi\}\right] \\ \vdots \\ \forall y \bullet \{\phi\}\ x \leftarrow F\ p\ y\ \{\psi\}\end{matrix}}{\{\phi\}\ x \leftarrow Y\ F\ y\ \{\psi\}}$$

Fig. 16. The generic Hoare calculus

Proposition 62 *Given the definition of the iteration construct, the rule*

$$\textbf{(iter)}\ \frac{\forall x \bullet \{\phi\}\ a \leftarrow b\ x\ \{if\ a\ then\ \psi\ else\ \xi\} \quad \forall x \bullet \{\psi\}\ y \leftarrow p\ x\ \{\phi[y/x]\}}{\{\phi[e/x]\}\ y \leftarrow iter\ b\ p\ e\ \{\xi[y/x]\}}\ (y \notin FV(\phi, \xi))$$

is derivable in the generic Hoare calculus.

PROOF. Let F be the functional from the definition of the $iter\ b\ p$, i.e.

$$F\ f\ x = if\ b\ then\ (do\ z \leftarrow p\ x;\ f\ z)\ else\ ret\ x.$$

Assume $\forall x \bullet \{\phi\}\ y \leftarrow f\ x\ \{\xi[y/x]\}$. By rule (Y), it suffices to derive

$$\{\phi\}\ y \leftarrow F\ f\ x\ \{\xi[y/x]\}.$$

By rule (if) and the first premise, this reduces to

$$\{\psi\}\ y \leftarrow (do\ z \leftarrow p\ x;\ f\ z)\ \{\xi[y/x]\} \quad \text{and}$$
$$\{\xi\}\ y \leftarrow ret\ x\ \{\xi[y/x]\}.$$

The second goal is discharged immediately by rules (dsef) and (wk). By the assumption and rules (ctr) and (seq), the first goal reduces to

$$\{\psi\}\ z \leftarrow p\ x\ \{\phi[z/x]\},$$

i.e. to the second premise. □

The rules for if-then-else and iteration have been formulated so as to allow side-effecting expressions as conditions. If the condition b is dsef, then one derives from the given rules and rule (dsef) the usual if rule and a rule for iter corresponding to the standard while rule:

$$\frac{\{\phi \wedge b\}\ x \leftarrow p\ \{\psi\} \qquad \{\phi \wedge \neg b\}\ x \leftarrow q\ \{\psi\}}{\{\phi\}\ x \leftarrow if\ b\ then\ p\ else\ q\ \{\psi\}} \qquad \frac{\{\phi \wedge (b\ x)\}\ y \leftarrow p\ x\ \{\phi[x/y]\}}{\{\phi[e/x]\}\ y \leftarrow iter\ b\ p\ e\ \{\phi[x/y] \wedge \neg(b\ y)\}}.$$

The latter rule specialises to the usual while rule

$$\textbf{(while)}\ \frac{\{\phi \wedge b)\}\ p\ \{\phi\}}{\{\phi\}\ while\ b\ p\ \{\phi \wedge \neg b\}}.$$

The rules of the calculus are sound for arbitrary (cpo-)monads:

Theorem 63 *If a Hoare triple is derivable in a cpo-monad (monad) by the rules of Fig. 16 (excluding (Y)), then the corresponding formula is derivable in the internal language.*

PROOF. We prove each rule as a lemma in the internal language, using the proof rules of Fig. 15:

(Ω), ($\bot$), (η): Straightforward from rules (tau) and (η) of Fig. 15.

(dsef): By rules (conj) and (wk), independently proved sound below, it suffices to show

$$\{\phi\}\ p\ \{\phi\} \quad \text{and} \quad \{\}\ x \leftarrow p\ \{x = p\}.$$

By Lem. 58 (and rule (η) of Fig. 15 to get rid of $ret\ \top$), the second of these claims is equivalent to deterministic side effect freeness of p. The first claim follows by rule (tau) of Fig. 15, since ϕ commutes with p.

(seq): By rules (app) and (pre) of Fig. 15, the premises imply

$$[a \leftarrow \phi; x \leftarrow \bar{p}; b \leftarrow \psi; \bar{y} \leftarrow \bar{q}; c \leftarrow \chi]\, a \Rightarrow b \quad \text{and}$$
$$[a \leftarrow \phi; x \leftarrow \bar{p}; b \leftarrow \psi; \bar{y} \leftarrow \bar{q}; c \leftarrow \chi]\, b \Rightarrow c.$$

By propositional reasoning, we obtain

$$[a \leftarrow \phi; x \leftarrow \bar{p}; \psi; y \leftarrow q; c \leftarrow \chi]\, a \Rightarrow c.$$

The conclusion then follows by the rule (dis) of Fig. 15.

(wk): As indicated above, this is a special case of (two applications of) (seq).

(ctr): Immediate by rule (ctr) of Fig. 15.

(if): Since *if* b *then* p *else* q is just an abbreviation for *do* $a \leftarrow b$; *if* a *then* p *else* q, the conclusion reduces by rules (seq) and (ctr) and the first premise to

$$\{\mathit{if}\ a\ \mathit{then}\ \psi\ \mathit{else}\ \xi\}\ x \leftarrow \mathit{if}\ a\ \mathit{then}\ p\ \mathit{else}\ q\ \{\chi\}$$

for $a : \mathit{Bool}$. We can then perform a case distinction over a. If $a = \mathit{True}$, then the above formula is semantically equivalent to

$$\{\psi\}\ x \leftarrow p\ \{\chi\},$$

i.e. the second premise. The case $a = \mathit{False}$ is analogous.

(conj): By rules (app) and (sef) of Fig. 15, we obtain from the premises

$$[a \leftarrow \phi; \bar{x} \leftarrow \bar{p}; b \leftarrow \psi; c \leftarrow \chi]\, a \Rightarrow b \quad \text{and}$$
$$[a \leftarrow \phi; \bar{x} \leftarrow \bar{p}; b \leftarrow \psi; c \leftarrow \chi]\, a \Rightarrow c.$$

By propositional reasoning, this implies

$$[a \leftarrow \phi; \bar{x} \leftarrow \bar{p}; b \leftarrow \psi; c \leftarrow \chi]\, a \Rightarrow b \wedge c.$$

The conclusion, which is by rule (ctr) of Fig. 15 equivalent to

$$[a \leftarrow \phi; \bar{x} \leftarrow \bar{p}; b \leftarrow \psi; c \leftarrow \chi; d \leftarrow \mathit{ret}\ (b \wedge c)]\, a \Rightarrow d,$$

then follows by (app) and (η+).

(disj): Analogous to (conj).

(Y): Let $F : (A \xrightarrow{c} ?TB) \xrightarrow{c} (A \xrightarrow{c} ?TB)$. As the bottom element $\bot$ of $A \xrightarrow{c} ?TB$ satisfies $\forall y \bullet \{\phi\}\ x \leftarrow \bot\ y\ \{\psi\}$, correctness of the rule follows by fixed point induction if the predicate $\lambda z : A \xrightarrow{c} ?TB \bullet \forall y \bullet \{\phi\}\ x \leftarrow z\ y\ \{\psi\}$ is admissible, i.e. closed under suprema of total chains. This is easily established in the internal logic, noting that $(\bigvee f_i)\ x = \bigvee (f_i\ x)$, that Hoare triples decode into equations between *do*-terms, and that binding commutes with suprema in cpo-monads. □

It is clear that completeness can only be expected in combination with suitable monad-specific rules; e.g., the calculus becomes the usual (complete) Hoare calculus when extended with an assignment rule specific to the store monad. In this sense, the calculus may be regarded as a generic framework for computational deduction systems.

11 Example: Reasoning about dynamic references

We now apply the general machinery developed so far to the (slightly extended) domain of the classical Hoare calculus, namely states consisting of creatable and destructively updatable references (note that this is just one example of a state monad), later to be extended by non-determinism.

The reference monad R uses a type constructor *Ref*, where *Ref* a is the set of references to values of type a. Then, $R\ a$ is the type of reference computations over a. The monad comes with operations for reading from and writing to references (besides the usual monad operations); see Fig. 17. We use the shorthand $\phi(*r)$ for *do* $x \leftarrow$ *read* r; $\phi(x)$. Note that with this notation, *ret* $(x = *r)$ is *not* stateless. Also note the difference between *ret* $(r = s)$ (equality of references, a stateless formula) and *ret* $(*r = *s)$ (equality of contents, a stateful formula). Moreover, recall that *ret* is inserted implicitly where needed.

The axiomatization provides all that is really necessary in order to reason about references, i.e. one does not need to rely on a particular implementation. Axiom *dsef-read* states that reading is deterministically side-effect free. Axiom *read-write* says that after writing to a reference, we can read the value. By contrast, writing to a reference does not change the values of *other* references (*read-write-other*). Note that nothing is said about the nature of references; they could e.g. be integers. The specification of *dynamic* references additionally provides an operation *new* for dynamically creating new references. *read-new* states that after initializing a reference, we can read the initial value. Moreover, creation of new references does not change the values of other references (*read-new-other*). Finally, two newly created references are distinct (*new-distinct*). Note that we do not say anything about reading from references that have not

spec $\textsc{Reference} = \textsc{CpoMonad}$ **then**
var $a\colon Cpo$
types $R\colon CpoMonad;\ Ref\ a\colon Flatcpo$
ops $read\colon Ref\ a \xrightarrow{c} R\ a;$
$\_\_ := \_\_\colon Ref\ a \xrightarrow{c} a \xrightarrow{c} R\ Unit$
forall $x, y\colon a;\ r, s\colon Ref\ a$
• $dsef(read\ r)$ %(dsef-read)%
• $\{\}\ r := x\ \{x = {}^*r\}$ %(read-write)%
• $\{\neg r = s \wedge x = {}^*r\}\ s := y\ \{x = {}^*r\}$ %(read-write-other)%

spec $\textsc{DynamicReference} = \textsc{Reference}$ **then**
var $a, b\colon Type$
op $new\colon a \xrightarrow{c} R(Ref\ a)$
forall $x\colon a;\ y\colon b \to a;\ r\colon Ref\ a;\ p\colon R\ b$
• $\{\}\ r \leftarrow new\ x\ \{x = {}^*r\}$ %(read-new)%
• $\{x = {}^*r\}\ s \leftarrow new\ y\ \{\neg r = s \Rightarrow x = {}^*r\}$ %(read-new-other)%
• $\{\}\ r \leftarrow new\ x;\ z \leftarrow p\ r;\ s \leftarrow new\ (y\ z)\ \{\neg r = s\}$ %(new-distinct)%

Fig. 17. Specification of the reference and the dynamic reference monad

been created yet. In the discussion below, references to rules always refer to the Hoare calculus of Fig. 16.

Using this axiomatization, we now show

$$\{\}\ r \leftarrow new\ x; s \leftarrow new\ y\ \{\neg r = s \wedge x = *r \wedge y = *s\}. \tag{1}$$

We proceed as follows. By *read-new* and rules (Ω) and (seq), we have

$$\{\}\ r \leftarrow new\ x; s \leftarrow new\ y\ \{y = *s\}.$$

By applying rule (seq) to *read-new* and *read-new-other*, we obtain

$$\{\}\ r \leftarrow new\ x; s \leftarrow new\ y\ \{\neg r = s \Rightarrow x = *r\}.$$

Instantiating *new-distinct* with $p = \lambda\_ \bullet\ ret\ ()$ and applying rule (η), we have

$$\{\}\ r \leftarrow new\ x; s \leftarrow new\ y\ \{\neg r = s\}.$$

We then obtain (1) by propositional reasoning with these three formulas.

Another example is the nondeterminism monad, shown in Fig. 18. While *fail* yields no result and hence satisfies arbitrary postconditions, *chaos* yields any result and hence nothing can be said about it. The operation $[]$ is nondeterministic choice (i.e. takes the union of value sets), and *sync* synchronises two nondeterministic values (i.e. takes the intersection of value sets).

spec NONDETERMINISM = CPOMONAD **then**
  var $a: Cpo$
  ops $fail, chaos: N\ a;$
    $\_[\!]\_, \_sync\_ : (N\ a \times N\ a) \xrightarrow{c} N\ a$
forall $x: a;\ p, q: N\ a;\ \varphi, \psi: N\Omega;\ \chi_1, \chi_2: a \rightarrow N\Omega$
• $\{\}\ fail\ \{\psi\}$ %(fail)%
• $\{\varphi\}\ x \leftarrow p\ \{\chi_1 x\}\ \wedge\ \{\varphi\}\ x \leftarrow q\ \{\chi_2 x\}\ \Rightarrow$
  $\{\varphi\}\ x \leftarrow p[\!]q\ \{\chi_1 x \vee \chi_2 x\}$ %(join)%
• $\{\varphi\}\ x \leftarrow p\ \{\chi_1 x\}\ \wedge\ \{\varphi\}\ x \leftarrow q\ \{\chi_2 x\}\ \Rightarrow$
  $\{\varphi\}\ x \leftarrow p\ sync\ q\ \{\chi_1 x \wedge \chi_2 x\}$ %(sync)%

Fig. 18. The nondeterminism monad

One advantage of the looseness of the specifications introduced so far is that we now can combine the specification of references and of nondeterminism and get a specification of nondeterministic reference computations (Fig. 19).

spec NONDETERMINISTICDYNAMICREFERENCE =
  DYNAMICREFERENCE **with** $R \mapsto NR$
  and NONDETERMINISM **with** $N \mapsto NR$

Fig. 19. The nondeterministic dynamic reference monad

As an example, we prove the partial correctness of Dijkstra's nondeterministic version of Euclid's algorithm for computing the greatest common divisor [16] within this monad. Let *euclid* be the program sequence (over *NR Int*)

$r \leftarrow new\ x;$
$s \leftarrow new\ y;$
$while\ ret\ (\neg{*r} == {*s})$
  $(if\ ret\ ({*r} > {*s})\ then\ r := {*r} - {*s}\ else\ fail$
  $[\!]$
  $if\ ret\ ({*s} > {*r})\ then\ s := {*s} - {*r}\ else\ fail)$

Assuming that we have some specification of arithmetic, including *gcd* specified to be the greatest common divisor function, we now prove

$$\{\}\ euclid\ \{{*r} = gcd(x, y)\}.$$

We proceed as follows. Using (dsef), (seq), and propositional reasoning, we can show

$$\begin{aligned}&\{\neg r = s \wedge gcd(*r, *s) = gcd(x, y) \wedge {}^*r > {}^*s\}\\&u \leftarrow read\ r; v \leftarrow read\ s\\&\{\neg r = s \wedge gcd(*r, *s) = gcd(x, y) \wedge {}^*r > {}^*s \wedge u = *r \wedge v = *s\}.\end{aligned}$$

By arithmetic reasoning and (wk), we obtain

$$\begin{aligned}&\{\neg r = s \wedge gcd(*r, *s) = gcd(x, y) \wedge {}^*r > {}^*s\}\\&u \leftarrow read\ r; v \leftarrow read\ s\\&\{\neg r = s \wedge gcd(u, v) = gcd(x, y) \wedge u > v \wedge v = *s\}.\end{aligned} \tag{2}$$

From *read-write* and *read-write-other*, we show by propositional reasoning

$$\begin{aligned}&\{\neg r = s \wedge gcd(u, v) = gcd(x, y) \wedge u > v \wedge v = *s\}\\&r := u - v\\&\{\neg r = s \wedge gcd(u, v) = gcd(x, y) \wedge u > v \wedge v = *s \wedge u - v = *r\}.\end{aligned}$$

By arithmetic reasoning and (wk), we obtain

$$\begin{aligned}&\{\neg r = s \wedge gcd(u, v) = gcd(x, y) \wedge u > v \wedge v = *s\}\\&r := u - v\\&\{\neg r = s \wedge gcd(*r, *s) = gcd(x, y)\}.\end{aligned}$$

By sequencing with (2) and noting that $r := {}^*r - {}^*s$ is shorthand for $u \leftarrow$ $read\ r; v \leftarrow read\ s; r := u - v$, we arrive at

$$\begin{aligned}&\{\neg r = s \wedge gcd(*r, *s) = gcd(x, y) \wedge {}^*r > {}^*s\}\\&r := {}^*r - {}^*s\\&\{\neg r = s \wedge gcd(*r, *s) = gcd(x, y)\}.\end{aligned}$$

By *fail*, we have

$$\begin{aligned}&\{\neg r = s \wedge gcd(*r, *s) = gcd(x, y) \wedge \neg {}^*r > {}^*s\}\\&fail\\&\{\neg r = s \wedge gcd(*r, *s) = gcd(x, y)\}.\end{aligned}$$

Hence by the standard (if) rule

$$\begin{aligned}&\{\neg r = s \wedge gcd(*r, *s) = gcd(x, y)\}\\&if\ ret\ ({}^*r > {}^*s)\ then\ r := {}^*r - {}^*s\ else\ fail\\&\{\neg r = s \wedge gcd(*r, *s) = gcd(x, y)\}.\end{aligned}$$

Analogously, we have

$$\begin{aligned}&\{\neg r = s \wedge gcd(*r, *s) = gcd(x, y)\}\\&if\ ret\ ({}^*s > {}^*r)\ then\ s := {}^*s - {}^*r\ else\ fail\\&\{\neg r = s \wedge gcd(*r, *s) = gcd(x, y)\}.\end{aligned}$$

From these, we obtain by *join* and rule (wk)

$$\begin{array}{l}\{\neg r = s \wedge gcd(*r,*s) = gcd(x,y)\} \\ \quad\; \mathit{if}\ \ ret\ ({}^*r > {}^*s)\ \mathit{then}\ r := {}^*r - {}^*s\ \mathit{else}\ \mathit{fail} \\ \;\; [] \ \mathit{if}\ \ ret\ ({}^*s > {}^*r)\ \mathit{then}\ s := {}^*s - {}^*r\ \mathit{else}\ \mathit{fail} \\ \{\neg r = s \wedge gcd(*r,*s) = gcd(x,y)\}.\end{array}$$

Applying the standard (while) rule and rule (wk) leads to

$$\begin{array}{l}\{\neg r = s \wedge gcd(*r,*s) = gcd(x,y)\} \\ \mathit{while}\ \ ret\ (\neg {}^*r == {}^*s) \\ \;(\ \ \mathit{if}\ \ ret\ ({}^*r > {}^*s)\ \mathit{then}\ r := {}^*r - {}^*s\ \mathit{else}\ \mathit{fail} \\ \;\; [] \ \mathit{if}\ \ ret\ ({}^*s > {}^*r)\ \mathit{then}\ s := {}^*s - {}^*r\ \mathit{else}\ \mathit{fail}) \\ \{\neg r = s \wedge gcd(*r,*s) = gcd(x,y) \wedge *r == *s\}.\end{array}$$

Using the arithmetic fact that $gcd(x, x) = x$, we obtain by (wk)

$$\begin{array}{l}\{\neg r = s \wedge gcd(*r,*s) = gcd(x,y)\} \\ \mathit{while}\ \neg {}^*r == {}^*s \\ \;(\ \ \mathit{if}\ \ ret\ ({}^*r > {}^*s)\ \mathit{then}\ r := {}^*r - {}^*s\ \mathit{else}\ \mathit{fail} \\ \;\; [] \ \mathit{if}\ \ ret\ ({}^*s > {}^*r)\ \mathit{then}\ s := {}^*s - {}^*r\ \mathit{else}\ \mathit{fail}) \\ \{*r = gcd(x,y)\}.\end{array} \tag{3}$$

From (1) above, we obtain by arithmetic reasoning and rule (wk)

$$\{\}\ r \leftarrow \mathit{new}\ x; s \leftarrow \mathit{new}\ y\ \{\neg r = s \wedge gcd(*r,*s) = gcd(x,y)\}, \tag{4}$$

and the result now follows by sequencing (3) and (4).

12 Conclusions

We have presented the design of HasCasl, a wide-spectrum language serving the integrated specification and development of software as well as mathematical modelling on a wider scale. Novel features of HasCasl include the semantic treatment of type class polymorphism by means of an extension semantics and support for inductive datatypes and recursion that does not rely on unique choice. We have moreover laid out the technical aspects of the syntax of the type class mechanism and its interaction with higher order subtyping in some detail.

We have illustrated the expressive strength of HasCasl by means of the development of a Hoare logic for monad-encapsulated generic side effects as used in modern functional-imperative programming. The latter is a contribution in its own right, as it offers modularised reasoning support for monad-based imperative programs, where generic rules are cleanly separated from axiomatisations of specific notions of side-effect (i.e. monads). A stronger generic computational logic of this nature, namely a monad-based dynamic logic, has been presented in [63]; this extension, however, relies on stronger assumptions on the underlying monad. The use of HasCasl outside the realm of software specification as such has been illustrated in [69], where composition tables of region connection calculi are verified in a logically heterogeneous setting in which HasCasl serves the definition of higher-order concepts such as the real numbers.

HasCasl is a central node in the logic graph of the Bremen heterogeneous tool set. As such, it is provided with extensible reasoning support, presently via a translation into Isabelle/HOL, and further connections to other logics in the graph, e.g. a translation of executable specifications to Haskell. These tools are being developed further; in particular, the technical handling of Isabelle proofs on translated HasCasl specifications and the development of suitable dedicated tactics is the subject of ongoing work. An open issue in the language design of HasCasl itself is the specification of nested polymorphism as supported by the Glasgow extensions of Haskell [51], i.e. to find a workaround for the fact that higher order logic is inconsistent with System F [14]. An initial step in this direction would be the support for existential types, which provide a clean way of encapsulating representations of abstract datatypes [32].

Acknowledgements

The authours wish to thank Mihai Codescu, Kathrin Hoffmann, Bernd Krieg-Brückner, Christoph Lüth, and Christian Maeder for useful comments and discussions.

References

[1] A. Abel and R. Mattes. Fixed points of type constructors and primitive recursion. In J. Marcinkowski and A. Tarlecki, editors, *Computer Science Logic, CSL 2004*, vol. 3210 of *Lect. Notes Comput. Sci.*, pp. 190–204. Springer, 2004.

[2] J. Adámek, H. Herrlich, and G. E. Strecker. *Abstract and Concrete Categories.* Wiley Interscience, 1990.

[3] E. Astesiano and M. Cerioli. Free objects and equational deduction for partial conditional specifications. *Theoret. Comput. Sci.*, 152:91–138, 1995.

[4] J. Bates and R. Constable. Proofs as programs. *ACM Trans. Prog. Lang. Systems*, 7:113–136, 1985.

[5] 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 *Lect. Notes Comput. Sci.*, pp. 19–36. Springer, 1999.

[6] M. Bidoit and P. D. Mosses. Casl *User Manual*, vol. 2900 of *Lect. Notes Comput. Sci.* Springer, 2004.

[7] L. Birkedal and R. E. Møgelberg. Categorical models for abadi and plotkin's logic for parametricity. *Math. Struct. Comput. Sci.*, 15, 2005.

[8] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, and K. Stølen. The requirement and design specification language Spectrum, an informal introduction, version 1.0. Technical report, Institut für Informatik, Technische Universität München, Mar. 1993.

[9] L. Cardelli. Notes on $F^{\omega}_{<:}$. Unpublished notes, 1990.

[10] M. Cerioli and J. Meseguer. May I borrow your logic? (transporting logical structures along maps). *Theoret. Comput. Sci.*, 173:311–347, 1997.

[11] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude - A High-Performance Logical Framework How to Specify, Program, and Verify Systems in Rewriting Logic*, vol. 4350 of *Lect. Notes Comput. Sci.* Springer, 2007.

[12] CoFI. The Common Framework Initiative for algebraic specification and development. Electronic Archives under www.cofi.info.

[13] The Coq Development Team. The Coq Proof Assistant – Reference Manual, v8.1. INRIA, 2006, Available under http://coq.inria.fr.

[14] T. Coquand. An analysis of Girard's paradox. In *Logic in Computer Science, LICS 1986*, pp. 227–236. IEEE, 1986.

[15] R. Diaconescu and K. Futatsugi. *CafeOBJ Report.* AMAST series. World Scientific, Singapore, 1998.

[16] E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[17] J.-C. Filliâtre. Proof of imperative programs in type theory. In T. Altenkirch, W. Naraschewski, and B. Reus, editors, *Types for Proofs and Programs, TYPES 1998*, vol. 1657 of *Lect. Notes Comput. Sci.*, pp. 78–92. Springer, 1999.

[18] C. Führmann. Varieties of effects. In M. Nielsen and U. Engberg, editors, *Foundations of Software Science And Computation Structures, FOSSACS 2002*, vol. 2303 of *Lect. Notes Comput. Sci.*, pp. 144–158. Springer, 2002.

[19] C. George, P. Haff, K. Havelund, A. E. Haxthausen, R. Milne, C. B. Nielson, S. Prehn, and K. R. Wagner. *The Raise Specification Language.* Prentice Hall, New York, 1992.

[20] J. Goguen and R. Burstall. Institutions: Abstract model theory for specification and programming. *J. ACM*, 39:95–146, 1992.

[21] J. A. Goguen and G. Rosu. Institution morphisms. *Formal Asp. Comput.*, 13:274–307, 2002.

[22] S. Goncharov, L. Schröder, and T. Mossakowski. Completeness of global evaluation logic. In R. Kralovic and P. Urzyczyn, editors, *Mathematical Foundations of Computer Science, MFCS 2006*, vol. 4162 of *Lect. Notes Comput. Sci.*, pp. 447–458. Springer, 2006.

[23] J. V. Guttag, J. J. Horning, S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *Larch: Languages and Tools for Formal Specification*. Springer, 1993.

[24] T. Hallgren. Haskell tools from the programatica project. In *Haskell Workshop, HASKELL 2003*, pp. 103–106. ACM Press, 2003.

[25] W. L. Harrison and R. B. Kieburtz. The logic of demand in Haskell. *J. Funct. Programming*, 15:837–891, 2005.

[26] A. Haxthausen. Order-sorted algebraic specifications with higher-order functions. *Theoret. Comput. Sci.*, 183:157–185, 1997.

[27] L. Henkin. The completeness of the first-order functional calculus. *J. Symbolic Logic*, 14:159–166, 1949.

[28] H. Herrlich, E. Lowen-Colebunders, and F. Schwarz. Improving Top: PrTop and PsTop. In *Category theory at work*, pp. 21–34. Heldermann, Berlin, 1991.

[29] D. Hutter, H. Mantel, G. Rock, W. Stephan, A. Wolpers, M. Balser, W. Reif, G. Schellhorn, and K. Stenzel. VSE: Controlling the complexity in formal software developments. In *Proceedings of the International Workshop on Applied Formal Methods*, Boppard, Germany, 1998.

[30] C. B. Jones. *Systematic Software Development Using VDM.* Prentice Hall, 1990.

[31] S. Kahrs, D. Sannella, and A. Tarlecki. The definition of extended ML: A gentle introduction. *Theoret. Comput. Sci.*, 173:445–484, 1997.

[32] K. Läufer. Type classes with existential types. *J. Functional Programming*, 6:485–517, 1996.

[33] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Principles of Programming Languages, POPL 95*, pp. 333–343. ACM Press, 1995.

[34] S. Mac Lane. *Categories for the Working Mathematician.* Springer, 1997.

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

[36] G. Mints. *A Short Introduction to Intuitionistic Logic.* Kluwer, 2000.

[37] J. C. Mitchell and P. J. Scott. Typed lambda models and cartesian closed categories. In J. Gray and A. Scedrov, editors, *Categories in Computer Science and Logic*, vol. 92 of *Contemp. Math.*, pp. 301–316. Amer. Math. Soc., 1989.

[38] 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.

[39] E. Moggi. *The Partial Lambda Calculus.* PhD thesis, University of Edinburgh, 1988.

[40] E. Moggi. Notions of computation and monads. *Inform. and Comput.*, 93:55–92, 1991.

[41] E. Moggi. A semantics for evaluation logic. *Fund. Inform.*, 22:117–152, 1995.

[42] T. Mossakowski. Relating CASL with other specification languages: the institution level. *Theoret. Comput. Sci.*, 286:367–475, 2002.

[43] T. Mossakowski. Heterogeneous specification and the heterogeneous tool set. Habilitation thesis, Universität Bremen, 2005.

[44] 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 07*, vol. 4424 of *Lect. Notes Comput. Sci.*, pp. 519–522. Springer, 2007.

[45] P. D. Mosses, editor. CASL *Reference Manual*, vol. 2960 of *Lect. Notes Comput. Sci.* Springer, 2004.

[46] 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.

[47] J. Nordlander. Polymorphic subtyping in O'Haskell. *Sci. Comput. Programming*, 43(2-3):93–127, 2002.

[48] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference, Version 2.4.* SRI International, Menlo Park, 2001.

[49] L. C. Paulson. Mechanizing coinduction and corecursion in higher-order logic. *J. Log. Comput*, 7:175–204, 1997.

[50] S. Peyton Jones, editor. *Haskell 98 Language and Libraries — The Revised Report.* Cambridge, 2003. also: *J. Funct. Programming* 13, 2003.

[51] S. Peyton Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. *J. Funct. Programming.* To appear.

[52] 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.

[53] B. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[54] A. Pitts. Evaluation logic. In G. Birtwhistle, editor, *Higher Order Workshop IV*, Workshops in Computing, pp. 162–189. Springer, 1991.

[55] 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.

[56] G. Rosolini. *Continuity and effectiveness in topoi.* PhD thesis, University of Oxford, 1986.

[57] G. Rosolini and T. Streicher. Comparing models of higher type computation. In *Realizability Semantics and Applications*, vol. 23 of *Electron. Notes Theoret. Comput. Sci.*, 1999.

[58] L. Schröder. The logic of the partial λ-calculus with equality. In J. Marcinkowski and A. Tarlecki, editors, *Computer Science Logic ,CSL 2004*, vol. 3210 of *Lect. Notes Comput. Sci.*, pp. 385–399. Springer, 2004.

[59] L. Schröder. The HasCasl prologue - categorical syntax and semantics of the partial λ-calculus. *Theoret. Comput. Sci.*, 353:1–25, 2006.

[60] 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.

[61] 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.

[62] 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.

[63] L. Schröder and T. Mossakowski. Monad-independent dynamic logic in HasCasl. *J. Logic Comput.*, 14:571–619, 2004.

[64] 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. to appear.

[65] L. Schröder, T. Mossakowski, A. Tarlecki, P. Hoffman, and B. Klin. Semantics of architectural specifications in Casl. In H. Hußmann, editor, *Fundamental Approaches to Software Engineering, FASE 2001*, vol. 2029 of *Lect. Notes Comput. Sci.*, pp. 253–268. Springer, 2001.

[66] M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 1992. 2nd edition.

[67] H. Thielecke. *Categorical Structure of Continuation Passing Style.* PhD thesis, University of Edinburgh, 1997.

[68] P. Wadler. How to declare an imperative. *ACM Computing Surveys*, 29:240–263, 1997.

[69] S. Wölfl, T. Mossakowski, and L. Schröder. Qualitative constraint calculi: Heterogeneous verification of composition tables. In D. Wilson and G. Sutcliffe, editors, *20th International FLAIRS Conference*, pp. 665–670. AAAI Press, 2007.

[70] O. Wyler. *Lecture notes on topoi and quasitopoi.* World Scientific, 1991.

Appendix

A Subkinding Rules

For convenience, the full set of subkinding rules as assembled in Sect.s 3 and 4 is shown in Fig. A.1. Recall that $\mathcal{V}$ denotes the set $\{\pm, +, -, \cdot\}$ of variance annotations, ordered by taking $\pm$ and $\cdot$ to be the smallest and the greatest element, respectively, and $+$ and $-$ to be incomparable.

$$\frac{Cl \le_C Kd \text{ in } \Sigma}{Cl \le_K Kd} \qquad \frac{Kd_1 \le_K Kd_2 \quad Kd_3 \le_K Kd_4}{\mu Kd_2 \to Kd_3 \le_K \mu Kd_1 \to Kd_4}\,(\mu \in \mathcal{V})$$

$$\frac{}{\mu Kd_1 \to Kd_2 \le_K \nu Kd_1 \to Kd_2}\quad(\mu, \nu \in \mathcal{V}, \mu \le \nu),$$

$$\frac{}{Kd \le_K Kd} \qquad \frac{Kd_1 \le_K Kd_2 \quad Kd_2 \le_K Kd_3}{Kd_1 \le_K Kd_3}$$

Fig. A.1. Subkinding rules

B Kinding Rules

The full set of kinding rules for pseudotypes as assembled in Sect. 3 and 4 is shown in Fig. B.1. Recall that Θ^{-1} and Θ^0 denote the context Θ with all outer variances reversed or removed, respectively.

C Syntax-directed Subkinding Rules

For implementation purposes, we give a syntax-directed version of the subkinding rules (cf. Appendix A). The point is to eliminate the transitivity

$$\frac{F : Kd_1 \text{ in } \Sigma \quad Kd_1 \leq_K Kd_2}{\Theta \rhd F : Kd_2} \qquad \frac{a : \mu Kd_1 \text{ in } \Theta, \mu \in \{+, \cdot\} \quad Kd_1 \leq_K Kd_2}{\Theta \rhd a : Kd_2}$$

$$\frac{\Theta^0 \rhd t : Kd_1 \quad \Theta \rhd s : Kd_1 \to Kd_2}{\Theta \rhd s\,t : Kd_2} \qquad \frac{\Theta \rhd t : Kd_1 \quad \Theta \rhd s : +Kd_1 \to Kd_2}{\Theta \rhd s\,t : Kd_2} \qquad \frac{\Theta^{-1} \rhd t : Kd_1 \quad \Theta \rhd s : -Kd_1 \to Kd_2}{\Theta \rhd s\,t : Kd_2}$$

$$\frac{\Theta, a : \mu Kd_1 \rhd t : Kd_2 \quad Kd_3 \leq_K Kd_1}{\Theta \rhd \lambda\, a : Kd_1 \bullet t : \nu Kd_3 \to Kd_2} \; (\mu \leq \nu \text{ in } \mathcal{V})$$

Fig. B.1. Kinding rules for type constructors

and reflexivity rules in the spirit of 'algorithmic subtyping' [53]. The syntax-directed rules are given in Fig. C.1.

$$\textbf{(cl-refl)} \; \frac{}{Cl \leq_K Cl} \qquad \textbf{(cl)} \; \frac{Cl \leq_C Kd_1 \quad Kd_1 \leq_K Kd_2}{Cl \leq_K Kd_2}$$

$$(\to) \; \frac{Kd_1 \leq_K Kd_2 \quad Kd_3 \leq_K Kd_4}{\mu Kd_2 \to Kd_3 \leq_K \nu Kd_1 \to Kd_4} \; (\mu, \nu \in \mathcal{V}, \mu \leq \nu)$$

Fig. C.1. Syntax-directed subkinding rules

Note that rule (cl) is indeed algorithmic since there are only finitely many declarations $Cl \leq_C Kd_1$. In rule $(\to)$, $\mathcal{V}$ is the set of variance annotations, ordered as described in Sect. 4.

Proposition 64 *The rules of Fig. C.1 derive the same subkinding judgements as the rules given in Fig. A.1 above.*

PROOF. (Sketch) It is clear that the rules of Fig. C.1 are derivable from the previous rules and subsume all of the previous rules except reflexivity and transitivity. By induction over the kind structure, it is easy to show that $Kd \leq_K Kd$ is derivable by the rules of Fig. C.1 for all kinds Kd. Finally, the fact that the relation $\leq_K$ generated by the rules of Fig. C.1 and the reflexivity rule is transitive is shown by induction over the combined lengths of the derivations of $Kd_1 \leq_K Kd_2$ and $Kd_2 \leq_K Kd_3$; this involves a case distinction over which rules were applied in the last step in either case.

D Syntax-directed Subtyping Rules

Similarly as for the subkinding system, one can give a syntax-directed set of rules, shown in Fig. D.1, for the subtype relation which is equivalent to the rules presented in Sect. 4. The proof of equivalence is analogous to the one sketched for Prop. 64. The introduction rules for variables and type constructors are, like the rule (cl) of Fig. C.1, algorithmic because there are only finitely many declarations $F \leq t$ and $a \leq t$ in Σ and Λ, respectively. As in Fig. 7, $\sqsubseteq$ ranges over $\{\leq, \leq_*\}$.

$$\frac{a \text{ in } \Theta}{\Theta; \Lambda \rhd a \sqsubseteq a} \qquad \frac{}{\Theta; \Lambda \rhd F \sqsubseteq F} \qquad \frac{a \leq t \text{ in } \Lambda \quad \Theta; \Lambda \rhd t \sqsubseteq s}{\Theta; \Lambda \rhd a \sqsubseteq s} \qquad \frac{F \leq t \text{ in } \Sigma \quad \Theta; \Lambda \rhd t \sqsubseteq s}{\Theta; \Lambda \rhd F \sqsubseteq s}$$

$$\frac{\Theta \rhd t_1, t_2 : +Kd_1 \to Kd_2 \quad \Theta; \Lambda \rhd s_1 \sqsubseteq s_2 \quad \Theta; \Lambda \rhd t_1 \sqsubseteq t_2}{\Theta; \Lambda \rhd t_1\ s_1 \sqsubseteq t_2\ s_2} \qquad \frac{\Theta \rhd t_1, t_2 : -Kd_1 \to Kd_2 \quad \Theta; \Lambda \rhd s_2 \leq_* s_1 \quad \Theta; \Lambda \rhd t_1 \leq_* t_2}{\Theta; \Lambda \rhd t_1\ s_1 \leq_* t_2\ s_2}$$

$$\frac{\Theta, a : Kd; \Lambda \rhd t \sqsubseteq s}{\Theta; \Lambda \rhd \lambda a : \mu Kd \bullet t \sqsubseteq \lambda a : \mu Kd \bullet s}\ (\mu \in \mathcal{V})$$

Fig. D.1. Syntax-directed subtyping rules for pseudotypes