HASCASL: Towards Integrated Specification and Development of Functional Programs

Lutz Schröder and Till Mossakowski

BISS, Department of Computer Science, Bremen University

Abstract. The development of programs in modern functional languages such as Haskell calls for a wide-spectrum specification formalism that supports the type system of such languages, in particular higher order types, type constructors, and parametric polymorphism, and that contains a functional language as an executable subset in order to facilitate rapid prototyping. We lay out the design of HASCASL, a higher order extension of the algebraic specification language CASL that is geared towards precisely this purpose. Its semantics is tuned to allow program development by specification refinement, while at the same time staying close to the set-theoretic semantics of first order CASL. The number of primitive concepts in the logic has been kept as small as possible; we demonstrate how various extensions to the logic, in particular general recursion, can be formulated within the language itself.

Introduction

In the application of formal methods to the specification and development of correct software, a critical problem is the transition from the level of specifications to the level of programs. In the HASCASL development paradigm introduced here, this problem is tackled as follows: we choose Haskell, a pure functional language, as the primary (but certainly not exclusive) target language, thereby avoiding the need to deal with side effects and other impure features (the fact that ML has such features was one of the primary obstacles in the design of Extended ML [17]). The specification language HASCASL is designed as an extension of the recently completed first-order based algebraic specification language CASL [3,8] with higher-order concepts such as (partial) function types, polymorphism, and type constructors (going beyond and, at times, deviating from [23]). This extended language is a wide-spectrum language; in particular, it is powerful enough to admit an executable sublanguage that is in close correspondence to Haskell, thus avoiding the need for a specialized interface language as employed in Larch [14]. At the same time, it is simple enough to keep the semantics natural and manageable. In this way, it becomes possible to perform the entire development process, from requirement specifications down to executable prototypes, by successive refinements within a single language.

In more detail, the basic logic of HASCASL consists essentially of the partial λ-calculus [21,22], extended with a form of type class polymorphism that combines features of Haskell and Isabelle [31], and provided with an intensional
Henkin style semantics. The main semantic problem that ensues is the treatment of recursive types and recursive functions, which are not available in classical higher-order logic. The traditional way of dealing with these features is Scott-Strachey denotational semantics [25,32], which needs to talk about notions like complete partial orders (cpos) and least fixed points.

It is indeed possible to add these features to the language without affecting the basic language definition (and hence complicating the semantics), by means of a boot-strap method that consists in writing the relevant extensions as specifications in basic HasCasiL. Much of the force of this approach stems from the fact that there is a secondary ‘internal logic’ that lives within the scope of $\lambda$-abstractions. This internal logic is by no means another primitive concept, but rather grows naturally out of the intensional semantics of partial higher order logic. Through the interaction with the two-valued ‘external’ logic in specifications, the internal logic (which is, in its ‘basic version’, extremely weak) may be observed as well as extended. By means of a suitably extended internal logic, one can then define a type class of cpos in the style of HOLCF [27], so that recursive function definitions become available: similarly, the internal logic allows coding the relevant axioms for recursive datatypes. Thus we arrive at a methodology that allows a smooth transition from abstract requirements to design specifications; the former are specified in partial higher order logic (which is not available in HOLCF), while the latter employ cpos and recursion, with undefinedness represented as $\perp$. In this way, we can, in the requirement specifications, avoid extra looseness arising from the cpos: these are introduced only at the design level, using a fixed set of constructors for generating cpos from recursive datatypes.

The basic logic and design of HasCasiL are laid out in Section 1. This is followed by a presentation of the ‘boot-strap’ features, namely, the internal logic, datatypes, and recursion, in Sections 2–4. The corresponding specifications are presented in detail. Section 5 concludes with a brief example that illustrates the HasCasiL development methodology. For unexplained categorical terminology, the reader is referred to [1,19].

1 The Basic HasCasiL Logic

We shall begin by defining the underlying logic of HasCasiL. We will be as economical as possible with primitive concepts; along the way, we will demonstrate how several important features may be added either as built-in syntactic sugar or, non-conservatively, as HasCasiL specifications. In order to avoid overburdening the presentation, we shall not be overly fussy about details of the syntax; a full description can be found in [29].

HasCasiL is designed as a higher order extension of the first-order algebraic specification language CASL (Common Algebraic Specification Language) developed by CoFI, the international Common Framework Initiative for Algebraic Specification and Development [8]. The features of CASL include first-order logic, partial functions, subsorts, sort generation constraints, and structured and architectural specifications. For the definition of the language cf. [9,3]; a full formal
semantics is laid out in [10]. The semantics of structured and architectural specifications is independent of the logic employed for basic specifications. In order to define our language, it is therefore sufficient to define its logic, i.e. essentially to fix notions of signature, model, sentence, and satisfaction as done below.

Roughly stated, the logic of HasCAsl is Moggi’s partial λ-calculus [21], equipped with a few semantically harmless extensions such as type constructors and type class polymorphism, and embedded into a full-featured external logic much as the one provided in first-order CASL.

1.1 Signatures and sentences

The types associated to a HasCAsl signature are built in the usual way from type constructors of a declared arity (possibly involving type classes; cf. Section 1.3). In particular, there are built-in type constructors \(- \ast \), \(- \ast \ast \), \(- \ast \ast \ast \) etc. for product types, \(- \multimap \) and \(- \rightarrow \) for partial and total function types, respectively, and a unit type unit. Types may be aliased; see [29] for details. Operators are constants of an assigned type. As in first order CASL, the user may specify a sort relation, denoted \(<\), on the declared sorts; in particular, \(s \rightarrow t\) is a subtype of \(s \rightarrow \tau\). Since sub-sorting will not really play a great role in the considerations below, we will mostly omit this feature from the rest of the presentation.

These data determine a notion of term according to the rules given in Figure 1. Notations like \(\bar{x} : \bar{s}\) abbreviate sequences, here: \(x_1 : s_1, \ldots , x_n : s_n\). Explicit projection symbols are absent, since they can be coded by terms such as \(\lambda x : s,y : t \cdot x\); here, we will use fixed abbreviations \(\text{fst}, \text{snd}, \text{and pr}_n\) with the obvious meaning. Application is regarded as a built-in operator with the obvious profile, so that no extra typing rule for application (of arbitrary terms) is required. Further built-in operators come with the subtype relation: whenever \(s < t\), one has a partial downcast operator \(- as s\) and an elementhood predicate \(- \in s\) on \(t\) with the same meaning as in first order CASL. Aside from the partial λ-abstraction of Figure 1, there is a total λ-abstraction \(\lambda \bar{x} : \bar{s} \cdot \alpha\), which abbreviates a downcast to the type of total functions. Additional typing rules for total functions will be used in static analysis.

\[
\begin{align*}
x : s & \in \Gamma \\
\Gamma \vdash x : s & \\
f : s \in \Sigma & \\
\Gamma \vdash f : s & \\
\Gamma \vdash \alpha_i : s_i, \ i = 1, \ldots , n & \\
\Gamma \vdash (\alpha_1, \ldots , \alpha_n) : s_1 \ast \cdots \ast s_n & \\
f : s \rightarrow \tau & \in \Sigma \\
\Gamma \vdash \beta : s & \\
\Gamma \vdash \beta : s & \\
\Gamma, \bar{x} : \bar{s} \vdash \alpha : t & \\
\Gamma \vdash \lambda \bar{x} : \bar{s} \cdot \alpha : s_1 \ast \cdots \ast s_n \rightarrow \tau & \\
\end{align*}
\]

**Fig. 1.** Term Formation and Typing Rules
Such terms may then be used in formulas of an external logic that offers essentially the same ample features as first order CASL, e.g. strong and existential equality, a definedness predicate, and the usual connectives and quantifiers including disjunction, negation, and existence. This external logic is set apart distinctly from a weaker internal logic to be used within $\lambda$-terms; see below.

1.2 Models and satisfaction

Since HASCASL is meant to extend first order CASL as faithfully as possible, its semantics will necessarily be set-theoretic. There is a broad range of choices available for a set-theoretic notion of model of higher order signatures; the principal options are as follows:

- In a *standard* model, all partial function types $s \rightarrow t$ are interpreted by the full set of partial functions from the interpretation of $s$ to that of $t$.
- In an *extensional Henkin* model [15], function types are interpreted by subsets of the full set of functions in such a way that all $\lambda$-terms can be interpreted (the latter property is called *comprehension*).
- In an *intensional* Henkin model, function types are interpreted by arbitrary sets equipped with an application operation of the appropriate type. Comprehension is still required; however, the way $\lambda$-terms are interpreted is now part of the structure of the model rather than just an existence axiom. Intensional Henkin models are discussed, e.g., in [20, 21].

The notion chosen for HASCASL, for reasons of semantics as well as methodology, is that of intensional Henkin models. To begin, moving away from standard models avoids the well-known incompleteness problems. Extensionality carries the disadvantage of destroying the existence of initial models of signatures in a setting with partial functions (see [4] for a simple example); further arguments in favour of intensional models even in the total setting are brought forward in [26]. If, on the other hand, extensionality is for some reason required by the user, it may easily be enforced:

**spec** EXTENSIONALITY =

forall $a, b : Type; f, g : a \rightarrow b$ . $(\forall x : a . f(x) = g(x)) \Rightarrow f = g$

From a categorical point of view (see [12], in particular 1.8, 2.3, 3.1, 4.3, and 5.5, on why we think this point of view is relevant), intensional models are attractive because they are in direct correspondence with the natural class of categorical models; cf. Remark 2 below.

Methodologically speaking, intensional models offer the advantage of allowing a more fine-grained analysis of function spaces. In [6], for instance, a model of the language PCF is defined in a category of sequential algorithms, where functions are considered together with an evaluation strategy. This also allows a distinction between various ways in which a function may fail to yield a value while, in an extensional setting, a function may merely be either defined or undefined.
A peculiarity of the intensional approach is that the intended equality of terms has to be explicitly imposed on the models. We therefore give a deduction system for existential equality \( \overset{\equiv}{=} \), to be read 'both sides are defined and equal', in Figure 2. Deduction takes place in a given context \( \Gamma \); the \( \xi \)-rule uses reasoning with assumptions, marked by square brackets, in a locally enlarged context. \( \text{def} \alpha \) abbreviates \( \alpha \overset{\equiv}{=} \alpha \). Notations like \( \alpha \beta \) refer to substitution of (the components of) \( \beta \) for the variables in the context of \( \alpha \). Rule (pr) is meant to work both ways. The rule (ax) serves the purpose of accommodating axioms that take the form of implications \( \phi \Rightarrow \psi \) (in an indicated context); such axioms are needed in order to ensure correct behaviour w.r.t. suborting and overloading, including the determination of the total function type (see [29] for details).

The deduction system given in [21] is quite similar, but takes strong equality ('if any one side is defined, then so is the other, and both are equal') as the primitive notion; the two systems are easily seen to have the same deductive strength. For the envisaged models, deduction is sound and complete [21, 28]; as shown in [21], this completeness result fails for extensional models (!).

\[
\begin{align*}
\text{var} & \quad \frac{x : s \text{ in } \Gamma}{x \overset{\equiv}{=} x} & \text{op} & \quad \frac{f : s \text{ in } \Sigma}{f \overset{\equiv}{=} f} & \text{sym} & \quad \frac{\alpha \overset{\equiv}{=} \beta \quad \beta \overset{\equiv}{=} \gamma}{\alpha \overset{\equiv}{=} \gamma} \\
\text{def} & \quad \frac{\phi \Rightarrow \psi \text{ axiom}}{\phi \overset{\equiv}{=} \psi} & \text{sym} & \quad \frac{\alpha \overset{\equiv}{=} \beta \quad \beta \overset{\equiv}{=} \gamma}{\alpha \overset{\equiv}{=} \gamma} \\
\text{cong} & \quad \frac{f : t \text{ in } \Sigma}{f \overset{\equiv}{=} f} & \text{ax} & \quad \frac{\phi \alpha \overset{\equiv}{=} \psi \alpha}{\psi \alpha} & \text{str} & \quad \frac{\text{def } f(\alpha)}{\text{def } f(\alpha)} \\
\text{def} & \quad \frac{\text{def } f(\alpha)}{f(\alpha) \overset{\equiv}{=} f(\beta)} & \text{ax} & \quad \frac{\phi \alpha \overset{\equiv}{=} \psi \alpha}{\psi \alpha} & \text{str} & \quad \frac{\text{def } f(\alpha)}{\text{def } f(\alpha)} \\
\text{unit} & \quad \frac{\text{def } a : \text{unit}}{a \overset{\equiv}{=} ()} & \text{pr} & \quad \frac{\text{pr}(\alpha) \overset{\equiv}{=} \text{pr}(\beta), \ i = 1, \ldots, n}{\alpha \overset{\equiv}{=} \beta : s_1 \times \ldots \times s_n} \\
\text{lambda} & \quad \frac{\text{def } \lambda g : f \overset{\equiv}{=} a}{(\beta_1) \frac{\text{def } (\alpha \gamma, \gamma) \overset{\equiv}{=} \alpha \gamma \overset{\equiv}{=} \beta \overset{\equiv}{=} \gamma}{\text{def } (\lambda g : f \overset{\equiv}{=} a)(\gamma) \overset{\equiv}{=} \alpha \gamma \overset{\equiv}{=} \beta \overset{\equiv}{=} \gamma} \\
\text{eta} & \quad \frac{\text{def } \lambda g : f \overset{\equiv}{=} a}{(\xi) \frac{\alpha \overset{\equiv}{=} \beta \overset{\equiv}{=} \gamma}{\lambda g : f \overset{\equiv}{=} \lambda g : f \overset{\equiv}{=} \beta}} \\
\end{align*}
\]

Fig. 2. Deduction rules for existential equality in context \( \Gamma \)
**Definition 1.** A *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

$$M_{n_1} \times \cdots \times M_{n_m} \rightarrow ? M_t$$

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 according to Figure 2 (extended by the above-mentioned subtyping axioms). Moreover, substitution must be modeled as composition of partial functions, and terms of the form $x_1 : s_1, \ldots, x_n : s_n \triangleright x_i : s_i$ must be interpreted by the appropriate product projections.

A *model morphism* between two such models is a family of functions $h_s$, where $s$ ranges over all types, that satisfies the homomorphism condition w.r.t. all interpretation functions for terms. Finally, *satisfaction* of formulae (of the external logic, cf. Section 1.1) in such models is defined in the obvious (classical) way.

**Remark 2.** Intensional Henkin models as defined above are in direct correspondence to categorical interpretations in so-called partial cartesian closed categories (pccc), i.e. essentially categories that have ‘partial function space objects’ representing partial morphisms in much the same way as function spaces in a cartesian closed category represent (total) morphisms (recall that a partial morphism is a span of the form $\bullet \overset{m}{\leftarrow} \bullet \overset{e}{\rightarrow} \bullet$, where $m$ belongs to a given distinguished class of monomorphisms that are admissible as ‘domains’); see [21] for a detailed definition. Equivalence results for partial $\lambda$-calculi on the one hand and pcccs on the other hand are discussed in [28]. An interpretation in a pcoc C gives rise to an intensional Henkin-model by composing the interpretation with the representable functor $\text{hom}_C(1, -)$, where 1 denotes the terminal object, and this process can be reversed; this fact is mentioned for the total case in [7].

### 1.3 Type classes and polymorphism

On top of the syntax given so far, we can now add a type class oriented form of shallow polymorphism without really affecting the semantics in an essential way. Besides improving the usability of the language as such, this also takes us closer to actually incorporating most of Haskell as a sublanguage.

As in Isabelle [31], we regard *type classes* as subsets of the set of all types. Such a type class is declared by writing

```haskell
class C1 or class C1 < C12 or class C1 = CT,
```

where $C_1$ is an existing class of which the class $C_1'$ is declared to be a subclass, and $CT$ is a *class term*. Class terms can be either class names, intersections of classes denoted as comma-separated lists, or built-in classes. Built-in classes are the class $\text{Type}$ of all types and the downsets $\{ a : a < t \}$ of given types $t$ under the subtyping relation. Constants can be declared as polymorphic over
type classes, by giving a type scheme with variables universally quantified over
classes:

\[ \text{op } f : \forall a : C \rightarrow \tau, \]

where \( \tau \) is a type that may contain the type variable \( a \) (the type variable may also be declared globally or locally by means of the keywords \text{var} and \text{forall}). This declaration has the effect of adding, for each type \( t \) of the class \( C \), an instance \( f[t] : \tau[a/t] \) to the signature (at this point, the mechanism is different from Isabelle, where operators are instantiated for \textit{all} types — this is undesirable in situations where the model theory matters). For the sake of readability, we will mostly omit the explicit type information.

Similarly, axioms may be enclosed in a universal quantification over type variables with assigned classes:

\[ \forall a : C \rightarrow \phi. \]

Such axioms are taken to mean an infinite collection of instances, one for each type. Quantification over types is only allowed at the outermost level of axioms.

Instances for a type class are produced by declaring arities for a type constructor in much the same sense as in Isabelle, stating that the result type belongs to a certain class if the argument types belong to certain other classes:

\[ \text{type } F : C \rightarrow \ldots C \rightarrow \tau. \]

In principle, one could leave it at this, except that one will often want to express the fact that axioms associated to the class are implied by the ones for the instance. To this end, one may write a list of axioms and constant declarations in grouping brackets directly after the declaration of the class, and then mark subclass declarations and type constructor declarations with the keyword \text{instance}. For example, part of a specification of a type class with a fixed idempotent endoprojection might look as follows:

class Proj
\[
\{ \text{var } a : \text{Proj} \\
\text{op } pr : a \rightarrow ?a \\
\quad \text{pr } = \lambda x : a \rightarrow pr(pr(x)) \} \\
\text{type } \text{instance } \_ \times \_ : \text{Proj} \rightarrow \text{Proj} \rightarrow \text{Proj} \\
\text{for all } a, b : \text{Proj} \\
\quad \text{pr }[a \times b] = \lambda x : a \rightarrow b \rightarrow (pr(pr(x),pr(y))
\]

This declaration implicitly produces a proof obligation, similar to the \%implies-annotation in Case, which states that all axioms explicitly attached to the class \text{Proj} (only one in the example), instantiated for \( a \times b \), are, similarly as in Isabelle/HOL, meant to follow from the axioms of the specification surrounding the instance, including the instantiated axioms for the arguments \( a \) and \( b \).

Remark 3. The polymorphism introduced above is essentially ML-polymorphism (except that a \textit{let}-construct is missing, which can, however, be regarded as syntactical sugar). The discourse in [11] 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 away' by just replacing polymorphic operators and axioms by all their
instances. The derivation of Girard's paradox in [11], Section 5, is based on the
assumption that terms of the language are identified up to untyped $\beta$-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,
instansiations of polymorphic constants are internally annotated with their types,
the contradiction disappears.

1.4 Predicates and non-strict functions

Two features appear to be still missing in the type system: on the one hand,
predicates, which are provided in CASL, and on the other hand, non-strict func-
tions, which are a feature of Haskell. However, these features can be regarded as
syntactical sugar in the setting built up so far:

HASCASL, like CASL and ML, is strict, i.e. undefined arguments always yield
undefined values, while Haskell functions are allowed to leave arguments uneval-
uated and thus yield results even on 'undefined' arguments. It is well-known that
such non-strict functions may be emulated by means of function types unit $\rightarrow ? t$
as argument types ('proceduring'); we support this concept as follows:

- The type unit $\rightarrow ? t$ is abbreviated as $? t$.
- There are two extra typing rules: a function that expects an argument of type
t may be applied to a term $\alpha$ of type $? t$, which is then automatically replaced
by $\alpha();$ conversely, a function that expects an argument of type $? t$ accepts
arguments $\beta$ of type $t$, which are automatically replaced by $\lambda x : \text{unit} \bullet \beta$.

(N.B.: for algebraic reasons, the type $s \rightarrow ? t$ is not the same as the type $s \rightarrow (? t)$,
the use of which is expressly discouraged.)

Predicates are represented as partial functions into unit. The idea here is
that definedness of such partial functions corresponds to satisfaction of predi-
cates. For types of the form $t \rightarrow \text{unit}$, the abbreviation $\text{pred}(t)$ is provided. The
definedness predicate $\text{def}$ of first order CASL is interpreted, for each type $s$,
as an abbreviation for $\lambda x : s \bullet ()$ (thanks to strictness, this has the expected
behaviour). There is no direct way to use logical connectives in predicate $\lambda$
terms. However, universal conjunctive logic is available: the type $\text{unit}$ can be
regarded as a type of truth values, with truth, written $tt$, abbreviating the term
$\lambda x : \text{unit} \bullet ()$ (of course, this is really an instance of $\text{def}$). The conjunc-
tion operator $\_ \land \_ : \text{unit} \land \text{unit} \rightarrow \text{unit}$ is $\lambda x, y : \text{unit} \bullet ()$. Finally, the univer-
sal quantifier $\forall : \text{pred}(\text{pred}(t))$ is just the predicate $\_ \in t \rightarrow \text{unit}$; falsity can
then be coded as the element $ff = \lambda x : \text{unit} \bullet \forall(\lambda y : \text{unit} \bullet y)$ of $\text{unit}$. The
introduction of further logical operators is possible, but non-conservative; see
Section 2.

2 Internal Equality and the Internal Logic

In basic HASCASL, it is specifically forbidden to use the equality symbol within $\lambda$
terms; however, equality can be sneaked back in by means of an internal equality.
A predicate

\[ \text{eq} : \forall a \cdot \text{pred}(a \times a) \]

is called an internal equality (see also [21]) if \( \text{eq}(x, y) \) is equivalent to \( x \cong y \) in the deduction system of Figure 2 (due to intensionality, this is a stronger property than equivalence of the two formulas for each pair \((x, y)\) of elements of \( a \) in a model).

In fact, internal equality can be specified in HASCASL. Interestingly, the introduction of internal equality turns out to be highly non-conservative, since it makes the logic available within \( \lambda \)-abstracted predicates substantially richer; besides the universal conjunctive logic of Section 1.4, one can define implication, disjunction, negation, and existential quantification as in [18]. The specification of internal equality and the new connectives is given in Figure 3. In order to improve readability, the equality symbol \( \cong \) can, after all, be used within \( \lambda \)-terms, but is, then, implicitly replaced by \( \text{eq} \). The CASL annotation \%def indicates a definitional extension, i.e. an extension that induces a bijection of model classes. Similarly, \%implies precedes axioms that are logical consequences of the previous axioms. It may come as a surprise that the last formula shown in Figure 3 expresses a form of extensionality; however, it is well-known that all categorical models are internally extensional [20].

\begin{figure}[h]
\begin{verbatim}
spec INTERNALLOGIC =
forall a : Type
  op eq : pred(a \times a)
  \begin{itemize}
    \item \( \lambda x : a \cdot \text{eq}(x, x) = \lambda x : a \cdot \text{tt} \)
    \item \( \lambda x, y : a \cdot \text{fst}(x, \text{eq}(x, y)) = \lambda x, y : a \cdot \text{fst}(y, \text{eq}(x, y)) \)
  \end{itemize}
\end{verbatim}
\end{figure}

\textbf{Fig. 3.} Specification of the internal logic

The internal logic is intuitionistic: there may be more than two truth values, and \( \text{neg}(\text{neg}(A)) \) is in general different from \( A \). The obvious deduction rules can
be proved as lemmas; e.g., it is not hard to show that the rule
\[
\phi \implies \psi; \quad \phi
\]
is derivable from the rules in Figure 2 and the definitions in Figure 3. The
external\ logic (cf. Section 1.1) remains classical: as soon as a predicate appears
as an atomic formula, all internal truth values except \(tt\) are collapsed into the
external \textit{false}. Under (external) extensionality (cf. Section 1.2), the internal logic
becomes \textit{almost} classical in the sense that there are \textit{at most} two truth values —
one may still have \(tt = ff\), which implies that all sorts are singletons. Of course,
one can explicitly require \(tt \neq ff\).

3 Recursive Datatypes

In order to actually represent functional programs in \textsc{HasCASL}, recursive
datatypes are an indispensable feature. As in first order \textsc{Casl}, recursive data
types are defined by means of the keyword \textbf{type} (cf. [9]), which may be qualified
by a preceding \textbf{free} or \textbf{generated}: a type declaration defines constructors and
— optionally — selectors for the declared type. The \textbf{generated} constraint introduces
an induction axiom; intuitively, this means that the type is term-generated
(‘no junk’). The \textbf{free} constraint additionally produces a case operator, which
means that the terms are kept distinct (‘no confusion’).

The automatically generated operators and axioms for a free datatype are
shown in Figure 4; the notation assumes that the type declaration was of the form
\[
\textbf{free type } t ::= C_1(t_{i_1}; \ldots ; t_{k_1}) \mid \cdots \mid C_n(t_{i_1}; \ldots ; t_{k_n})
\]
where the \(t_{ij}\) are either non-recursive or equal to \(t\) (the generalization to nested
or mutually recursive types does not present great additional difficulties). The
axioms make use of the internal logic as introduced in Figure 3. Case operators
take one argument \(x\) of the defined type and one function argument for each
constructor, to be chosen depending on which constructor produced \(x\).

Technically, we do not impose any restrictions on the types that appear in the
recursion. However, in the case of recursion on the left hand side of the function
arrow, the usual Russell-type paradoxes appear. Consider, e.g., the type
\[
\textbf{type } L ::= \text{abs}(\text{rep} : L \to \tau) \text{ L}
\]
axiomatizing the untyped partial \(\lambda\)-calculus. In the presence of internal equality,
negation is available in \(\lambda\)-terms, so that we have the term
\[
\lambda x : L \bullet \text{fst}(x, \text{not def rep}(x))(x)
\]
that produces Russell’s contradiction. A contradiction, in this case, means that
all predicates are true, so that all types have at most one element. On the other
hand, the specification has non-trivial models in plain \textsc{HasCASL} without
internal equality since the corresponding domain equation can be solved, e.g., in
the partial cartesian closed category of pointed cpos [25].
## Remark 4. Interestingly, the type

\[
\text{type } U ::= \text{abs (rep : } U \to U)\
\]

required for the total untyped λ-calculus does have non-trivial models even in the presence of internal equality: take a cartesian closed category \( \mathcal{C} \) with reflexive object \( U \). The functor category \( \text{Set}^{\mathcal{C}^{op}} \) is a topos and as such gives rise to a Henkin model with internal equality. Moreover, the Yoneda embedding \( Y : \mathcal{C} \to \text{Set}^{\mathcal{C}^{op}} \) preserves the cartesian closed structure [30], so that \( Y(U) \) is a reflexive object and hence provides us with a (non-extensional) model of the above type.

### 4 Recursive Functions

One problem of a Henkin-semantics for function types (intensional or extensional) is that even primitive recursive function definitions will not in general constitute definitional extensions, since the relevant functions may, in some models, not be present in the interpretation of the function type. It is possible, however, to impose a cpo structure on the relevant types, thus ensuring that all recursive functions actually live in the function type. Just as in the case of the internal logic, we can avoid an actual extension of the language; instead, we introduce the cpo structure by means of suitable specifications, building on the specification of internal equality (cf. Section 2). The overall concept is closely related to that of HOLCF [27]: the crucial difference is that the surrounding logic is (unlike HOL) partial.

The specification of the cpo structure and the fixed point operator is given in Figure 5. \( \text{Nat} \) is a specification of the natural numbers with a sort \( \text{nat} \), operations \( 0 : \text{nat} \) and \( \text{Suc} : \text{nat} \to \text{nat} \), and the usual axioms including induction and primitive recursion (which does not require the cpo structure for its definition). We introduce type classes \( \text{Cpo} \) and \( \text{Cppo} \) of cpos and cpos with bottom,
respectively, with generic instantiations that extend the ordering to products and partial and total continuous function types \( a \rightarrow b \) and \( a \rightarrow b \); the subclass \( Flatpo \) restricts the order to be equality. The continuous function types are subtypes of the built-in function types (due to intensionality, the given definitions of the elementhood predicates as \( \lambda \)-terms are necessary in order to determine the subsort uniquely); partial continuous functions are required to have Scott open domains. Elements of function types are compared pointwise, and elements of product types are compared componentwise. Now, we can introduce a least fixed point operator \( Y \); using this operator, we define a polymorphic undefined constant.

Of course, we cannot expect any of this to be conservative even over the internal logic. However, there are obviously no problems concerning consistency: in standard models, orderings of partial function spaces are cpos with bottom.

A further instance of the class \( Cpo \) is a free datatype \( t \) that has constructor arguments of class \( Cpo \). The instance can be automatically generated; the generation process is invoked by means of the keyword \texttt{deriving} borrowed from Haskell. If \( t \) has constructors \( C_t \) as in Section 3, we obtain

\[
\text{type instance } t : Cpo
\]

\[
\begin{align*}
\text{\quad \{ } \, & = \lambda x, y : t \cdot \\
\text{\quad \{ } \, & \quad (\text{case } x \text{ of } C_1(x_1, \ldots, x_n) \rightarrow \\
\text{\quad \quad & \quad (\text{case } y \text{ of } C_1(y_1, \ldots, y_k) \rightarrow x_1 \leq y_1 \text{ and } \ldots \text{ and } x_n \leq y_k \text{;}} \\
\text{\quad \quad & \quad C_2(y_1, \ldots, y_k) \rightarrow ff; \\
\text{\quad \quad & \quad \ldots)} \\
\text{\quad \quad & \quad \ldots)}
\end{align*}
\]

(where we use built-in syntactical sugar for the \texttt{case} operation), i.e. applications of different constructors are incomparable, while applications of the same constructor are compared argument-wise. There is no circularity here: the definition of the ordering is recursive, but does not use the fixed point operator. Rather, it imposes a particular equation on the ordering, and this equation determines the ordering uniquely thanks to the induction axiom of Figure 4. It is easy to see that the case operation, restricted to continuous arguments, is continuous w.r.t. this ordering, and hence can be used in definitions of recursive functions.

Actual recursive definitions will be expressions that involve \( Y \) and a partial downcast to the total continuous function type. As long as operators are given the right types, such expressions actually denote functions: call a term \( \alpha \) in context \( \check{\alpha} : \check{\delta} \) that has a type of class \( Cpo \) \textit{continuous} if \( \lambda \check{x} : \check{\delta} \cdot \alpha \) is continuous (cf. Figure 5). Moreover, call a type a \textit{cpo-type} if it is built from basic sorts and type variables of class \( Cpo \) by means of the instance declarations for type constructors given in Figure 5 (in particular, cpo-types are of class \( Cpo \)). Then we have

\textbf{Proposition 5.} If, for \( \Gamma \vdash \alpha : t \), all operator constants (besides application) that occur in \( \alpha \), as well as the variables in \( \Gamma \), have cpo-types, and \( t \) is a cpo-type, then \( \alpha \) is continuous.
spec Recursion = InternalLogic then Nat then
class Cpo
{ var a : Cpo
  ops _ ≤ _ ≺ : pred(a × a)
    isChain : pred(nat → a)
    isBound : pred(a × (nat → a))
  sup : (nat → a) → a
  · all(λx : a → x ≤ x)
  · all(λx, y, z : a → (x ≤ y and y ≤ z) → (x ≤ z))
  · all(λx, y : a → (x ≤ y and y ≤ x) → x = y)
  · isChain(a) = λs : nat → a → all(λn : nat → s(n) ≤ s(Suc(n)))
  · isBound(a) = λx : a, s : nat → a → all(λn : nat → s(n) ≤ x)
  · all(λs : nat → a → def sup(a) impl (isBound(sup(s), s) and all(λx : a → isBound(x, s) impl sup(s) ≤ x)))
  · all(λs : nat → a → isChain(s) impl def sup(s)) }

class Cppo < Cpo
{ var a : Cppo
  op bottom : a
  · all(λx : a → bottom ≤ x) }

class instance Flatcpo < Cpo
  · ∀c : Flatcpo → _ ≤ _[a] = eq[a]

vars a, b : Cpo; c : Cppo

type instance a × b : Cpo
  · _ ≤ _[a × b] = λx, y : a × b → fst(x) ≤ fst(y) and snd(x) ≤ snd(y)

type instance _ → _ : Cppo → Cppo → Cppo

type instance unit : Cppo
  · () ≤ ()

type a →? b < a →? b
  · _ ∈ (a →? b) = λf : a →? b → all(λx, y : a × b → (def f(x) and x ≤ y) impl def f(y)) and
    all(λs : nat → a → (isChain(s) and def f(sup(s))) impl ex(λm : nat →
    def f(s(m)) and sup(λn : nat → f(s(n + m)) = f(sup(s))))

type a →? b < a →? b
  · _ ∈ (a →? b) = λf : a →? b → all(λx : a → def f(x) impl f(x) ≤ g(x))

type instance a →? b : Cppo
  · _ ≤ _[a →? b] = λf, g : a →? b → all(λx : a → def f(x) impl f(x) ≤ g(x))

type instance a →? b : Cppo
  · _ ≤ _[a →? b] = λf, g : a →? b → f ≤ [a →? b] g

type instance a →? c : Cppo

then %def
  op Y : (c →? c) → c
  · all(λf : c →? c → f(Y(f)) = Y(f)) and
    all(λx : c → eq(f(x), x) impl Y(f) ≤ x))
  op undefined : unit →? c = Y(λx : unit →? c → x)

Fig. 5. Specification of the cpo structure and the fixed point operator
As a consequence, $\lambda$-abstractions of terms $\alpha$ as in the proposition are continuous and hence possess a least fixed point. Note that the fixed point operator itself is of a cpo-type, provided that its parameter $a$ is instantiated with a cpo-type.

**Remark 6.** A rather different approach to the recursive function problem is pursued in synthetic domain theory (SDT) [16]: the type of propositions (given as the truth value object in a topos) and the 'domain classifier' unit are distinct, so that one can impose the requirement that all functions are continuous, w.r.t. a somewhat differently defined ordering, which in general fails to be antisymmetric (e.g., the order relation on the type of propositions is indiscrete). In the more simplistic view presented above, the internal equality necessarily fails to be even monotone. The SDT approach can be put on top of HasCAsL instead of internal equality as defined above; the methodological effects of this need examination.

In the specification language Spectrum [13], the continuity problem is dealt with in a different fashion: elements of function types are required to be continuous; non-continuous operators (e.g., predicates) are admitted, but cannot be abstracted. In particular, there is no real logic available within $\lambda$-terms: one can use a type of booleans, but for this type, negation has a fixed point. A further, similar approach is found in [2], where there is a two-layer type system with two $\lambda$-abstractions used in programs and specifications, respectively.

5 From HasCAsL Specifications to Haskell Programs — an Example

Consider the following requirement specification of finite maps from keys to elements, following a module of the library of the Glasgow Haskell compiler:

```haskell
spec FiniteMap = Bool then

  type FinMap : type → type → type
  forall key : Eq; elt, elt1, elt2 : Type
  ops empty : FinMap key elt;
    add : FinMap key elt → key → elt → FinMap key elt;
    lookup : FinMap key elt → key → elt;
    map : (key → elt1 → elt2) → FinMap key elt1 → FinMap key elt2
  forall m : FinMap key elt; e : elt; k : key
    - def lookup empty k
    - k1 == k2 = True ⇒ lookup (add m k1 e) k2 = e
    - k1 == k2 = False ⇒ lookup (add m k1 e) k2 = lookup m k2
  forall f : key → elt1 → elt2; m1 : FinMap key elt1; k : key
    - lookup (map f m1) k = f k (lookup m1 k)
```

Here, `Bool` is a specification of a datatype `Bool` of booleans with constants `True` and `False`, as well as a type class `Eq`, corresponding to the Haskell type class of the same name, with a boolean-valued equality function `==`. After several refinement steps, one arrives, e.g., at the following design specification that implements maps as association lists (not necessarily in the most effective way):
spec AssocList = Bool then
vars a, elt, elt1, elt2 : Cpo; key : Cpo, Eq
free type List a ::= [] | _ :: (a; List a) deriving Cpo
type AList key elt := List (key × elt)
program
  empty : AList key elt = []
  add (m : AList key elt) (k : key) (e : elt) :? AList key elt = (k, e) :: m
  lookup ([ ] : AList key elt) (k : key) :? elt = undefined()
  lookup ((k1, e1) :: ml) k = if k == k1 then e1 else lookup ml k
  map (f : key → elt1 → elt2) ([ ] : AList key elt1) ? AList key elt2 = []
  map f ((k1, e1) :: ml) = (k1, f k1 e1) :: map f ml

The program block declares operations and simultaneously provides recursive definitions for them using the pattern matching notation known from most functional programming languages. These definitions are internally coded by the case operator of Section 3 and the fixed point operator Y of Section 4 in the obvious way. The specification is executable, but not monomorphic in the sense that it has, up to isomorphism, a unique model: with a Henkin-style semantics, intensional or extensional, it is generally infeasible (though perhaps not impossible) to eliminate the looseness inherent in the interpretation of function types.

Note in particular that types that are of class Type in FiniteMap are refined by types of class Cpo in AssocList. The refinement can be expressed by means of the standard CASL structuring features:

view IMPLEMENTFM: FiniteMap to AssocList =
  type FinMap → AList

The transition from cpo-types to standard types is achieved by simply forgetting the cpo-structure.

AssocList has, apart from syntax issues, the form of a Haskell program. In fact, a large subset of Haskell can be directly translated to HASCASL in the style of this example and is thereby provided with a denotational semantics. We expect that this denotational semantics will turn out to be compatible with the de facto operational semantics given by the existing Haskell compilers.

6 Conclusions and Future Work

We propose to use HASCASL for the specification and development of functional programs, in particular Haskell programs. HASCASL faithfully extends CASL to intensional partial higher-order logic with type class oriented shallow polymorphism. Its semantics is straightforward as well as flexible. In particular, we have shown that it is possible to extend the internal logic of λ-terms by means of specifications written in HASCASL in such a way that, in turn, recursive datatypes and a HOLCF-style fixed point theory become specifiable; non-continuous function types are retained for purposes of requirement specifications. We have ended
up with a logic that allows one to write functional programs within the specification language itself. Thus, there is no need for mediating logic morphisms or interface logics between specifications and programs.

Future work will partly concern the inclusion of further programming language features in HasCASL. This concerns in particular features of the Glasgow extensions for Haskell, foremost among them existential types, provided that it can be established that this does not lead to inconsistencies (such as the ones arising from full System F polymorphism; cf. [11]). Further, possibly easier, extensions concern multiparameter type classes and constructor classes. Moreover, the semantic properties of the obvious embedding of first order CASL into HasCASL need elaboration, in particular w.r.t. the CASL structuring operations.

As in the case of first order CASL [24], proof support for HasCASL will be supplied by means of an encoding in Isabelle/HOL, with special attention paid to the coding of the intensional function types. In a subsequent step, we aim at an integration of the corresponding tools for both languages into the MAYA environment [5], which provides a management of proof obligations for structured specifications and a management of change (for both specifications and programs) in an evolutionary software development paradigm. Moreover, there will be a tool for translating executable HasCASL specifications to Haskell, which can also be used in order to animate specifications.

Acknowledgements

This work forms part of the DFG-funded project HasCASL (KR 1191/7-1). Partial support by the CoFI Working Group (ESPRIT WG 29432) is gratefully acknowledged, as well as the work of the CoFI Language and Semantics Task Groups. We wish to thank Bernd Krieg-Brückner for his salomonic solution of the non-strict function syntax problem, and Christoph Lüth for useful comments.

References