The Logical Framework of SPECTRUM [1]

Radu Grosu
Franz Regensburger

Fakultät für Informatik, Technische Universität München
80290 München , Germany

E–Mail: spectrum@informatik.tu-muenchen.de

March 15, 1994

[1]This work is sponsored by the German Ministry of Research and Technology (BMFT) as part of the compound project "KORSO - Korrekte Software" and by the German Research Community (DFG) project SPECTRUM.

Abstract

The SPECTRUM project concentrates on the process of developing well-structured, precise system specifications. SPECTRUM is a specification language, with a deduction calculus and a development methodology. An informal presentation of the SPECTRUM language with many examples illustrating its properties is given in [BFG+93a, BFG+93b]. The purpose of this article is to describe its formal semantics.

Contents

Chapter 1

Introduction

The SPECTRUM specification language is axiomatic and borrows concepts both from algebraic languages (e.g. LARCH [GHW85]) as well as from type theoretic languages (e.g. LCF [CWMG79]). An informal presentation with many examples illustrating its properties is given in [BFG+93a, BFG+93b]. We briefly summarize its principal characteristics.

1.1 Influences from algebra

In SPECTRUM specifications the influence of algebraic techniques is evident. Every specification consists of a signature and an axioms part. However, in contrast to most algebraic specification languages, the semantics of a specification in SPECTRUM is loose, i.e. it is not restricted to initial models or even term generated ones. Moreover, SPECTRUM is not restricted to equational or conditional equational axioms, since it does not primarily aim at executable specifications. One can use full first order logic to write very abstract and non-executable specifications or only use its constructive part to write specifications which can be understood and executed as programs.

Loose semantics leaves a large degree of freedom for later implementations. It also allows the simple definition of refinement as the reduction of the class of models. This reduction is achieved by imposing new axioms which result from design decisions occurring in the stepwise development of the data structures and algorithms.

Since writing well structured specifications is one of our main goals, a flexible language for structuring specifications has been designed for SPECTRUM. This structuring is achieved by using so called specification building operators which map a list of argument specifications into a result specification. The language

for these operators was originally inspired by ASL [SW83]. The current version borrows concepts also from Haskell [HJW92], LARCH and PLUSS [Gau86].

1.2 Influences from type theory

The influence from type theory is twofold. On the type level SPECTRUM uses shallow predicative polymorphism with type classes in the style of Isabelle [Nip93]. The theory of type classes was introduced by Wadler and Blott [WB89] and originally realized in the functional programming language Haskell. Type classes may be used both to model overloading [CW85, Str67] as well as many instances of parameterized specifications. Like in object oriented languages type classes can be organized in hierarchies such that every class inherits properties from its parent classes. This gives our language an object oriented flavour.

The other influence of type theory can be seen in the language of terms and their underlying semantics. SPECTRUM incorporates the entire notation for typed λ-terms. The definition of the semantics and the proof system was heavily influenced by LCF. Therefore SPECTRUM supports a notion for partial and non-strict functions as well as higher order functions in the sense of domain theory. The models of SPECTRUM specifications are assumed to be certain continuous algebras. All the statements about the expressiveness of LCF due to its foundation in domain theory carry over to SPECTRUM.

Beside type classes there are also two features in the SPECTRUM logic which distinguish SPECTRUM from LCF. SPECTRUM uses three valued logic and also allows in a restricted form the use of non-continuous functions for specification purposes. These non-continuous functions are an extension of predicates and allow to express facts in a functional style that would otherwise have to be coded as relations. The practical usefulness of these features has to be proved in case studies.

In conclusion, all the above features make SPECTRUM a very powerful general purpose specification language. It can be used successfully in data base applications, computationally intensive applications or even distributed applications since it can easily incorporate a theory for streams and stream processing functions [Bro88].

The purpose of this report is to describe the formal semantics of the SPECTRUM kernel language. This semantics incorporates in a uniform and coherent way the properties already mentioned. In comparison with other logics for higher order functions (e.g. LCF family) our main contributions are:

- a denotational semantics based on order sorted algebras for type classes (we are only aware of an operational semantics for Haskell),

- the use of non-continuous functions for specification purposes,
- the identification of predicates with (strong) boolean functions in the context of a three valued logic.

Chapter 2

The SPECTRUM Institution

In this report we concentrate on the formal semantics of the kernel language of SPECTRUM which is pure predicate logic in the style of LCF. Viewed as a specification language SPECTRUM provides constructs for specifying in the large e.g. renaming, hiding, enrichment and parameterisation. Also various induction principles can be coded in SPECTRUM using the techniques known from LCF [Pau84].

In order to give a meaning to the "specifying in the large" constructs, one needs additionally to signatures, sentences and algebras also constructs allowing to relate sentences over different signatures and algebras over either the same or different signatures. These constructs are the signature morphisms, the homomorphisms and the reducts. A logical framework that supports all this constructs is that of institutions [BG84]. The mathematical language for presenting institutions is by tradition that of category theory and therefore we present the semantics of SPECTRUM to some extend in categorical terms.

The SPECTRUM institution is shown in figure 2.1. The different parts of this institution are introduced in the subsequent sections of this chapter.

The language of SPECTRUM consists of two closely interacting parts. The first one is the language of sorts which describes a universe of sorts using sort constructors, sort variables and kinds[1]. The second one is the language on the object level that is used to describe elements living in the above specified sorts[2], the carrier sets. These two levels cannot easily be separated since in our framework of loose model theoretic semantics the semantics of the sorts depends on the semantics of the elements that are specified to inhabit the sorts and vice versa.

[1] in order to avoid (or produce) confusion we will use **kind** for **class** and **sort** for **type**

[2] in a logic with higher order elements functions are also elements of carriers.

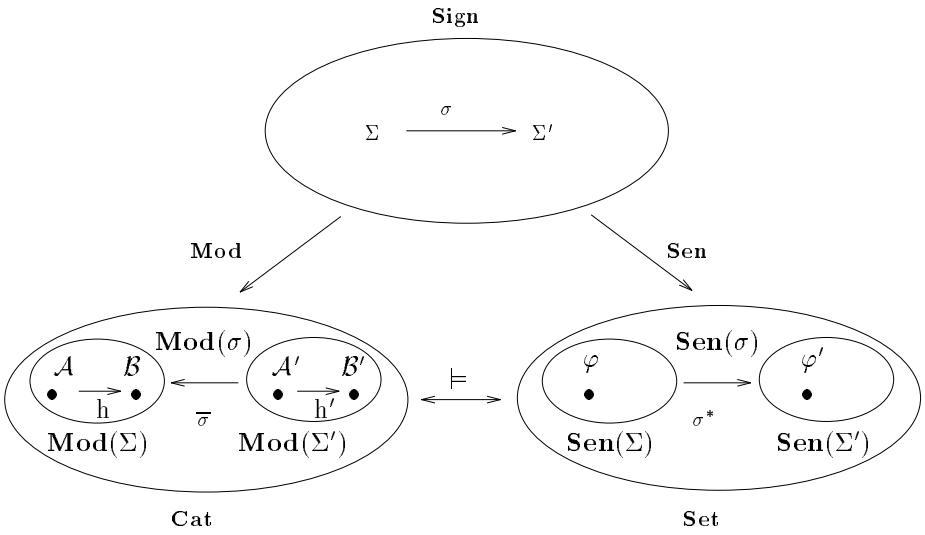

Figure 2.1: The SPECTRUM Institution

This concept of two levels is present in all subsequent parts of the paper and will be symptomatic for all the definitions. We first give a notion of sort signature and then define the concept of a signature on the object level with respect to a given sort signature. We define object terms with respect to a given language for sort terms and interpret these object terms in an algebra with respect to an interpretation for the sorts involved.

2.1 The category of Signatures

2.1.1 Signatures

Signatures introduce a typed alphabet both for building terms and in the sort language and for building terms (and sentences) in the object language. Terms in the sort language are used to type terms in the object language. The more powerful the sort language is the more powerful object terms can be written.

The sort language of SPECTRUM is restrictive enough to assure the existence of static type checking and type inference algorithms but is powerful enough to support advanced features like functional (and user defined) types, shallow polymorphism and type classes. This expressivity is achieved by putting enough structure in the sort signatures.

Definition 1.1 Sort Signature:

A sort signature $\Omega = (K, \leq, SC)$ is an order sorted signature[3], where

- $(K, \leq)$ is a partial order on *kinds*,
- $SC = \{SC_{w,k}\}_{w \in (K \setminus \{map\})^*, k \in K}$ is an indexed set of *sort constructors* with monotonic functionalities i.e.:

$$(sc \in SC_{w,k} \cap SC_{w',k'}) \wedge (w \leq w') \Rightarrow (k \leq k')$$

A sort signature must satisfy the following additional constraints:

- It is *regular, coregular* and *downward complete*. These properties[4] guarantee the existence of principal kinds and sorts.
- It includes the standard sort signature (see below).
- All kinds except *map* and *cpo*, which are in the standard signature, are below *cpo* with respect to $\leq$. In other words, *cpo* is the top kind for all kinds a user may introduce.

□

Kinds are introduced to model Haskell–like type classes. Like type classes they "sort" the types. The subsort order on kinds is intended to model the subclass order. The sort constructors are used to build sort terms.

Definition 1.2 The standard (predefined) sort signature

[3]Order kinded would be more precise; see [GM87, Gog76] for order sorted algebras. In the sequel we will use order sorted and mean order sorted on the level of kinds.

[4]See [SNGM89] for a definition. [NP] use slightly different criteria.

The standard sort signature:

$$\begin{array}{lll} \Omega_{standard} = & (& \{cpo, map\}, \emptyset, \\ & & \{\{\mathsf{Bool}\}_{cpo}, \\ & & \ \{\rightarrow\}_{cpo\ cpo,\ cpo}, \\ & & \ \{\mathbf{to}\}_{cpo\ cpo,\ map}, \\ & & \ \{\times_n\}_{\underbrace{cpo \ldots cpo}_{n\ times},\ cpo} \\ & & \} \\ &) & \end{array}$$

contains two kinds and four sort constructors (actually, we have for each n a sort constructor $\times_n$):

- *cpo* represents the kind of *all complete partial orders*, *map* represents the kind of *all full function spaces*[5],
- Bool is the type of booleans, $\rightarrow$ is the constructor for lifted continuous function spaces, **to** is the constructor for full function spaces and $\times_n$ for $n \geq 2$ is the constructor for Cartesian product spaces.

□

The sort signatures together with a disjoint family $\mathcal{X}$ of sort variables indexed by kinds (a *sort context*) allows us to define the set of monomorphic sort terms.

Definition 1.3 Monomorphic Sort Terms:

$T_\Omega(\mathcal{X})$ is the freely generated order sorted term algebra over $\mathcal{X}$. □

Example 1.1 Some sort terms

Let $\mathsf{Set} \in SC_{cpo,\ cpo}$. Then:
$\mathsf{Set}\ \alpha \rightarrow \mathsf{Bool},\ \mathsf{Bool} \times \mathsf{Bool} \in T_\Omega(\{\alpha\}_{cpo})$ □

The sort expressions obtained by binding the sort variables occurring in the monomorphic sort terms with a universal quantifier (written as Π) are called polymorphic (or Π) sort terms. A polymorphic sort term of the form $\Pi\alpha$:

[5]Complete partial orders are used to model continuous functions and full function spaces are used to model non-continuous functions. The latter ones are never implemented but are extremely useful for specification purposes. An alternative approach is to use only full function spaces in the semantics and to encode continuity of functions in the logic. In [Reg94] HOLCF a higher order version of LCF is embedded into the logic HOL using the generic framework of Isabelle. In this thesis it is shown that the full function space and its subspace of continuous functions over cpo's can live together in one type frame without problems.

$U.e(\alpha)$ denotes the Cartesian product of the family $\{e(t) \mid t \in U\}$[6][7]. In languages with predicative polymorphism the universe U_2 of polymorphic sorts is introduced only after all elements of the universe U of monomorphic sorts are defined. As a consequence polymorphic sorts can neither be nested in sort expressions nor used to instantiate sort variables[8]. The use of predicative polymorphism assures both the existence of classical, set theoretic models and the existence of type inference algorithms (see next section for a discussion about type inference). In our case the role of U is taken by kinds. Moreover, since we do not want to nest the "**to**"s we allow only kinds below cpo to be used in Π–expressions

Definition 1.4 Π-Sort Terms:

$\Pi\alpha_1 : k_1, \ldots, \alpha_n : k_n.e \in T_\Omega^\Pi$ if:

- $e \in T_\Omega(\mathcal{X})$
- $\mathsf{Free}(e) \subseteq \{\alpha_1, \ldots, \alpha_n\}$
- $k_i \leq cpo \qquad$ for $k_i \in K$, $1 \leq i \leq n$

□

Example 1.2 A Π-Sort Term:

$\Pi\alpha : cpo.\mathbf{Set}\ \alpha \longrightarrow \mathsf{Bool} \in T_\Omega^\Pi$ □

In a signature every constant or mapping will have a sort without free sort variables. This motivates the following definition.

Definition 1.5 Closed Sort Terms:

$$T_\Omega = T_\Omega(\emptyset)$$
$$T_\Omega^{closed} = T_\Omega \cup T_\Omega^\Pi$$

□

Note that $T_{\Omega,cpo}^{closed}$ will contain valid sorts for constants while $T_{\Omega,map}^{closed}$ will contain valid sorts for mappings.

Having defined the notion of a signature at the sort level we are able to define polymorphic signatures at the object level.

[6] For simplicity we use the same syntax for terms and their meaning. They will be distinguished when giving the semantics.

[7] Note the similarity with logic by reading "Cartesian product" as "conjunction" and a "sort" as a "formula".

[8] This is why this kind of polymorphism is also known as *shallow* polymorphism.

Definition 1.6 Polymorphic Signature:

A polymorphic signature $\Sigma = (\Omega, F, O)$ is a triple where:

- $\Omega = (K, \leq, SC)$ is a sort signature.
- $F = \{F_\mu\}_{\mu \in T^{closed}_{\Omega, cpo}}$ is an indexed set of constant symbols.
- $O = \{O_\nu\}_{\nu \in T^{closed}_{\Omega, map}}$ is an indexed set of mapping symbols.

It must include the standard signature
$\Sigma_{standard} = (\Omega_{standard}, F_{standard}, O_{standard})$ which is defined as follows:

- Predefined Constants ($F_{standard}$):
  - $\{\mathsf{true}, \mathsf{false}\} \subseteq F_{\mathsf{Bool}}$, $\{\neg\} \subseteq F_{\mathsf{Bool} \to \mathsf{Bool}}$, $\{\wedge, \vee, \Rightarrow\} \subseteq F_{\mathsf{Bool} \times \mathsf{Bool} \to \mathsf{Bool}}$ are the boolean constants and connectives.
  - $\{\bot\} \subseteq F_{\Pi\alpha : cpo.\ \alpha}$ is the polymorphic bottom symbol.
  - $\{\mathsf{fix}\} \subseteq F_{\Pi\alpha : cpo.\ (\alpha \to \alpha) \to \alpha}$ is the polymorphic fixed point operator.
- Predefined mappings ($O_{standard}$):
  - $\{=, \sqsubseteq\} \subseteq O_{\Pi\alpha : cpo.\ \alpha \times \alpha\ \mathbf{to}\ \mathsf{Bool}}$ are the polymorphic equality and approximation predicates.
  - $\{\delta\} \subseteq O_{\Pi\alpha : cpo.\ \alpha\ \mathbf{to}\ \mathsf{Bool}}$ is the polymorphic definedness predicate.

□

The constant and mapping symbols are the typed alphabet for building object terms. They can be either monomorphic (e.g. true, false, $\neg$) or polymorphic (e.g. $\bot$, fix or $=$). Since a polymorphic sort $\Pi\alpha : k.e$ denotes the Cartesian product of the family $\{e(t) \mid t \in k\}$, an element of this product is a tuple $(a_{t_1}, a_{t_2}, \ldots)$ such that $a_t \in e(t)$. In other words, this is a function mapping monomorphic sorts t into elements $a(t) \in e(t)$ where $e(t)$ is also monomorphic. The elements $a(t)$ are also called the *instances* of a. For example $\bot(Nat)$, $\bot(Bool)$ and $\bot(Nat \to Nat)$ are instances of the polymorphic element $\bot$.

2.1.2 Signature Morphisms

In the specification development process we often need to rename a specification or to relate it with a specification over another signature. The connection between an abstract specification and a more concrete version possibly having additional functionality is an example of such a relation.

The basic ingredient for relating specifications are the signature morphisms. Based on signature morphisms we will show how to to translate (or rename) sentences over a signature into sentences over another signature. Signature morphisms will also be used when relating models over different signatures.

Signature morphisms map signatures into signatures. Since a signature is built in two steps, first the sort signature and then the polymorphic signature, the definition of morphisms is also done in two steps.

Definition 1.7 Sort Signature Morphism:

A sort signature morphism ω: $\Omega \rightarrow \Omega'$ is an order sorted signature morphism $\omega = (\omega_K, \omega_{SC})$ (see [GM87, Gog76]) where:

- ω_K is a monotonic map on kinds:
  $\omega_K : (K, \leq) \rightarrow (K', \leq')$
  $k \leq l \;\Rightarrow\; \omega_K(k) \leq' \omega_K(l)$
- ω_{SC} is a family of maps respecting the types and the overloading of sort constructors. More precisely:
  $\omega_{SC} = \{\omega_{w,k} : SC_{w,k} \rightarrow SC'_{\omega^*_K(w),\omega_K(k)}\}_{w \in K^*, k \in K}$
  $sc \in SC_{w,k} \Rightarrow\; \omega_{SC}(sc) \in SC'_{\omega^*_K(w),\omega_K(k)}$ where
  $\omega^*_K(k_1 \ldots k_n) = \omega_K(k_1) \ldots \omega_K(k_n)$
  $sc \in SC_{w,k} \cap SC_{v,l} \;\Rightarrow\; \omega_{SC}(sc) \in SC'_{\omega^*_K(w),\omega_K(k)} \cap SC'_{\omega^*_K(v),\omega_K(l)}$

□

Sort signature morphisms allow us to define sort terms translation.

Definition 1.8 Sort Terms Translation

Given a sort signature morphism ω we denote as usual by ω^* its homomorphic extension to sort terms in $T_\Omega(\mathcal{X})$. We will also denote by ω^* the extension to terms in T^Π_Ω which is defined as follows:

$$\omega^*(\Pi\alpha_1 : k_1, \ldots, \alpha_n : k_n.\ e) \quad = \quad \Pi\alpha_1 : \omega(k_1), \ldots, \alpha_n : \omega(k_n).\ \omega^*(e)$$

□

A polymorphic signature morphisms i.e. a signature morphism between two polymorphic signatures consists of three components: a sort signature morphism ω and two functions $\sigma_F : F \rightarrow F'$ and $\sigma_O : O \rightarrow O'$ mapping constants and respectively operations from the first signature into constants and respectively operations from the second signature. As for the sort signature morphisms we require that σ_F and σ_O are "type" preserving.

Definition 1.9 Signature Morphisms:

A signature morphism $\sigma : \Sigma \rightarrow \Sigma'$ is a triple $\sigma = (\omega, \sigma_F, \sigma_O)$ where:

- $\omega : \Omega \rightarrow \Omega'$ is a sort signature morphism
- $\sigma_F : F \rightarrow F'$ is a $T^{closed}_{\Omega,cpo}$ indexed function:
  $$\sigma_F = \{\sigma_{F,\mu} : F_\mu \rightarrow F'_{\omega^*(\mu)}\}_{\mu \in T^{closed}_{\Omega,cpo}}$$
- $\sigma_O : O \rightarrow O'$ is a $T^{closed}_{\Omega,map}$ indexed function:
  $$\sigma_F = \{\sigma_{O,\nu} : O_\nu \rightarrow O'_{\omega^*(\nu)}\}_{\nu \in T^{closed}_{\Omega,map}}$$

□

2.2 The Sen Functor

2.2.1 The language of Terms

In the previous section we introduced the polymorphic signatures which serve to construct terms in the object language. The construction itself is the purpose of this section.

Like in [Mit90] the core language used to define the semantics of SPECTRUM is explicitly typed i.e. the application of polymorphic constants to sort terms is explicit and the λ–bounded variables are written together with their sorts. This assures that every well formed term has a unique sort in a given context and that the semantics of this term, although given with respect to one of its derivations, is independent from the particular derivation if the sorts of the free variables are the same in all derivation contexts.

For convenience, the *concrete* language of SPECTRUM is like ML, HOL, LCF and Isabelle implicitly typed i.e. the type information is erased from terms. However, like all the above languages, SPECTRUM has principles types i.e. every implicitly typed term t has a corresponding explicitly typed term t' such that erasing all type information from t' yields again t and for every other explicitly typed term t'' having the above property, the type of t'' is an instance of the type of t' for some special notion of instance. Having principles types guaranteed, the semantics of an implicitly typed term t is simply defined to be the semantics of t'[9]. The set of well formed terms is defined in two steps. First we define the

[9]The advantage of this technique is that the problem of defining and finding (rsp. deciding) the principal type property is separated from the definition of the semantics. The drawback is the introduction of two languages namely the one with implicit typing and the one with explicit types. An alternative would be to define the semantics directly on well formed derivations for implicitly typed terms avoiding the introduction of an explicitly typed language. However, since the type system of SPECTRUM is an instance of the type system of Isabelle, we preferred to use an explicit type system and refer to [Nip93, NP] for results about principal typings.

context free syntax of pre terms via a BNF like grammar. In the second step we introduce a calculus for well formed terms that uses formation rules to express the context sensitive part of the syntax.

Context Free Language (Pre Terms)

<term> ::=	ψ	(*Variables*)
	<id>	(*Constants*)
	<Πid> $\underline{[}$ {<sortexp> //$\underline{,}$}$^+$ $\underline{]}$	(*Polyconstant-Inst*)
	<map> <term>	(*Mapping application*)
	<Πmap> $\underline{[}$ {<sortexp> //$\underline{,}$}$^+$ $\underline{]}$<term>	(*Polymapping-Inst*)
	$\underline{\langle}$ {<term> //$\underline{,}$}$^{2+}$ $\underline{\rangle}$	(*Tuple* $n \geq 2$)
	$\underline{\lambda}$ <pattern> $\underline{.}$ <term>	(λ-*abstraction*)
	<term> <term>	(*Application*)
	$\underline{\mathbf{Q}}$ <tid> $\underline{.}$ <term>	($Q \in \{\forall^\perp, \exists^\perp\}$)
	$\underline{(}$ <term> $\underline{)}$	(*Priority*)

<tid> ::= ψ $\underline{:}$ <sortexp> <sortexp> ::= $T_\Omega\ (\mathcal{X})$

<pattern> ::= <tid> | $\underline{\langle}$ {<tid> //$\underline{,}$}$^{2+}$ $\underline{\rangle}$

<id> ::= $F_{T_\Omega, cpo}$ <map> ::= $O_{T_\Omega, map}$

<Πid> ::= $F_{T^\Pi_\Omega, cpo}$ <Πmap> ::= $O_{T^\Pi_\Omega, map}$

In addition all object variables $x \in \psi$ are different from sort variables $\alpha \in \mathcal{X}$ and all variables are different from identifiers in F and O.

Context Sensitive Language

With the pre terms at hand we can now define the well formed terms. We use a technique similar to [Mit90] and give a calculus of formation rules. Since for sort variables there is only a binding mechanism in the language of sort terms but not in the language of object terms, we need no dynamic context for sort variables. The disjoint family $\mathcal{X}$ of sort variables (the sort context) carries enough information. For the object variables, however, there are several binders and therefore we need an explicit variable context.

Definition 2.10 Sort Assertions

The set of sort assertions $\triangleright$ is a set of tuples $(\mathcal{X}, \Gamma, e, \tau)$ where:

- $\mathcal{X}$ is a sort context.
- $\Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ is a set of sort assumptions (a variable context), such that $\tau_i \in T_{\Omega, cpo}(\mathcal{X})$ and no x_i occurs twice in the sort assumptions contained in Γ (valid context condition). This prohibits overloading of variables in one scope.
- e is the pre term to be sorted.
- $\tau \in T_{\Omega, cpo}(\mathcal{X})$ is the derived sort for e.

We define:
$(\mathcal{X}, \Gamma, e, \tau) \in \rhd$ if and only if there is a finite proof tree D for this fact according to the natural deduction system below. □

When we write $\Gamma \rhd_\chi e :: \tau$ in the text we actually mean that there is a proof tree (sort derivation) for $(\mathcal{X}, \Gamma, e, \tau) \in \rhd$. If we want to refer to a special derivation D we write $D : \Gamma \rhd_\chi e :: \tau$. The intuitive meaning of the sort assertion $(\mathcal{X}, \Gamma, e, \tau)$ with $\Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ is that if the variables $x_1, \ldots, x_n$ have sorts $\tau_1, \ldots, \tau_n$ then the pre term e is well formed and has sort τ.

Formation rules for well formed terms

Axioms:

$$(\text{var})\frac{}{x : \tau \rhd_\chi x :: \tau} \qquad (\text{const})\ \frac{}{\emptyset \rhd_\chi c :: \tau}\left\{ c \in F_\tau \right.$$

$$(\Pi\text{-inst})\frac{}{\emptyset \rhd_\chi f[\tau_1, \ldots, \tau_n] :: \tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]}\left\{ \begin{array}{l} f \in F_{\Pi\alpha_1 : k_1, \ldots, \alpha_n : k_n . \tau} \\ \alpha_i : k_i \Rightarrow \tau_i : k_i \end{array} \right.$$

Note that in the above axiom $f[\tau_1, \ldots, \tau_n]$ is part of the syntax
whereas $\tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$ is a meta notation for this presentation of the calculus. The axiom states that given a polymorphic constant $f \in F_{\Pi\alpha_1 : k_1, \ldots, \alpha_n : k_n . \tau}$ every instance of f via the sort expressions $\tau_i : k_i$ yields an explicitly typed term $f[\tau_1, \ldots, \tau_n]$ of sort $\tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$ which is τ after simultaneous replacement of all sort variables α_i by sort expressions τ_i of appropriate kind.

Inference Rules:

$$(\text{weak})\frac{\Gamma \rhd_\chi e :: \tau}{\Gamma \cup \{x_1 : \tau_1, \ldots, x_n : \tau_n\} \rhd_\chi e :: \tau}$$

The 'valid context condition' in the rule (weak) prevents us from building contexts Γ with $x:\tau, x:\sigma \in \Gamma$ and $\tau \neq \sigma$

$$\text{(map-appl)}\ \frac{\Gamma \rhd_\chi e :: \tau_1}{\Gamma \rhd_\chi oe :: \tau_2}\Big\{\ o \in \mathrm{O}_{\tau_1 \mathrm{to} \tau_2}$$

$$\text{(}\Pi\text{map-appl)}\frac{\Gamma \rhd_\chi e :: \sigma_1[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]}{\Gamma \rhd_\chi o[\tau_1, \ldots, \tau_n]e :: \sigma_2[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]}\left\{\begin{array}{l} o \in \mathrm{O}_{\Pi\alpha_1:k_1,\ldots,\alpha_n:k_n.\sigma_1 \mathrm{to} \sigma_2} \\ \alpha_i : k_i \Rightarrow \tau_i : k_i \end{array}\right.$$

The rules (map-appl) and (Πmap-appl) are the formation rules for application of (polymorphic) mappings to terms. They ensure that a symbol for a mapping alone is not a well formed term which means that mappings may only occur in application context. This is another example for the restricted use of the full function space in SPECTRUM.

$$\text{(tuple)}\ \frac{\Gamma \rhd_\chi e_1 :: \tau_1 \ldots \Gamma \rhd_\chi e_n :: \tau_n}{\Gamma \rhd_\chi \langle e_1, \ldots, e_n \rangle :: \tau_1 \times \ldots \times \tau_n}\Big\{\ n \geq 2$$

$$\text{(abstr)}\frac{\Gamma, x:\tau_1 \rhd_\chi e :: \tau_2}{\Gamma \rhd_\chi \lambda x:\tau_1.e :: \tau_1 \rightarrow \tau_2}\ \{e \dagger x$$

$$\text{(patt-abstr)}\frac{\Gamma, x_1:\tau_1, \ldots, x_n:\tau_n \rhd_\chi e :: \tau}{\Gamma \rhd_\chi \lambda\langle x_1:\tau_1, \ldots, x_n:\tau_n\rangle.e :: \tau_1 \times \ldots \times \tau_n \rightarrow \tau}\left\{\begin{array}{l} e \dagger x_i \\ 1 \leq i \leq n \end{array}\right.$$

where $e \dagger x$ is a property of pre terms. A calculus for $e \dagger x$ is presented below.

$$\text{(appl)}\frac{\Gamma \rhd_\chi e_1 :: \tau_1 \rightarrow \tau_2 \qquad \Gamma \rhd_\chi e_2 :: \tau_1}{\Gamma \rhd_\chi e_1 e_2 :: \tau_2}$$

Note that formations for (map-appl) (Πmap-appl) and (appl) use implicit but different application mechanisms. There is no problem in determining the last step in a derivation for a term $e_1 e_2$. If e_1 is not a constant then rule (appl) must be used since there are no variables or composed terms for mappings. If on the other hand e_1 is a constant then the choice is also clear since F and O are disjoint. Of course there remains the problem of guessing the right type τ_1 for the term e_1 in rule (appl) if e_1 is a composed term. But this is another problem of type inference not concerning the distinction between mappings and functions in application context.

$$\text{(quantifier)}\ \frac{\Gamma, x:\tau \rhd_\chi e :: \mathsf{Bool}}{\Gamma \rhd_\chi Qx:\tau.e :: \mathsf{Bool}}\Big\{\ Q \in \{\forall^\perp, \exists^\perp\}$$

$$(\text{priority})\frac{\Gamma \rhd_\chi e :: \tau}{\Gamma \rhd_\chi (e) :: \tau}$$

This concludes the definition of sort derivations. We now present the calculus for $e \dagger x$. The purpose of this side condition is to prohibit the building of λ-terms that do not have a continuous interpretation. Consider the term:

$$\lambda x : \mathsf{Bool}.=[\mathsf{Bool}]\langle x, x\rangle$$

In our semantics the interpretation of the symbol $=$ is the polymorphic identity which is by definition not monotonic. If we allowed the above expression as a well formed term its interpretation would have to be a non-monotonic function.

The property $e \dagger x$ is recursively defined on the structure of the pre term e. It's reading is 'e dagger x' and means 'e is continuous in x'. In the calculus below the set $\Phi(e)$ represents the set of free variables with respect to the binders $\forall^\perp$, $\exists^\perp$ and λ with the obvious definition.

$$(\dagger - \text{var})\frac{}{x \dagger x}$$

$$(\dagger - \text{notfree})\frac{x \notin \Phi(e)}{e \dagger x}$$

$$(\dagger - \text{tuple})\frac{e_1 \dagger x \qquad \ldots \qquad e_n \dagger x}{\langle e_1, \ldots, e_n\rangle \dagger x}$$

$$(\dagger - \text{abstr})\frac{e \dagger x \qquad e \dagger y}{\lambda y : \tau . e \dagger x}$$

$$(\dagger - \text{patt-abstr})\frac{e \dagger x \qquad e \dagger x_1 \qquad \ldots \qquad e \dagger x_n}{\lambda\langle x_1 : \tau_1, \ldots, x_n : \tau_n\rangle . e \dagger x}$$

$$(\dagger - \text{appl})\frac{e_1 \dagger x \qquad e_2 \dagger x}{e_1 e_2 \dagger x}$$

$$(\dagger - \text{quant})\ \frac{e \dagger x \qquad e \dagger y}{Q y : \tau . e \dagger x}\Big\{\ Q \in \{\forall^\perp, \exists^\perp\}$$

$$(\dagger - \text{prio})\frac{e \dagger x}{(e) \dagger x}$$

As we will see later in section 2.4 the quantifiers get a three valued Kleene interpretation. If e is continuous in x and y also $\forall^{\perp} y:\tau.e$ and $\exists^{\perp} y:\tau.e$ are continuous in x. Therefore we can allow terms like $\lambda x:\sigma.\forall^{\perp} y:\tau.e$ provided the dagger test $\forall^{\perp} y:\tau.e \dagger x$ succeeds. For example the test $\exists^{\perp} y:\tau.\delta y \wedge e \dagger x$ will fail since $\delta y \wedge e \dagger y$ fails.

In the report [BFG+93b] we used the phrase 'where x is not free on a mappings argument position' as a context condition for the formation rules (abstr) and (patt-abstr). Looking at the example $\lambda x:\sigma.\exists^{\perp} y:\tau.\delta y \wedge e$ we see that this is too weak for terms with quantifiers inside.

Well formed Terms and Sentences

With the context sensitive syntax of the previous paragraph we are now able to define the notion of well formed terms over a polymorphic signature. Since we use an explicitly typed system, a well formed term is a pre term e together with a sort context $\mathcal{X}$, a variable context Γ and a sort τ.

Definition 2.11 Well formed terms

Let Σ be a polymorphic signature. The set of well formed terms over Σ in sort context $\mathcal{X}$ and variable context Γ with sort τ is defined as follows:

$$T_{\Sigma,\tau}(\mathcal{X},\Gamma) = \{(\mathcal{X},\Gamma,e,\tau) \mid \Gamma \rhd_{\chi} e :: \tau\}$$

The set of all well formed terms in context $(\mathcal{X},\Gamma)$ is defined to be the family

$$T_{\Sigma}(\mathcal{X},\Gamma) = \{T_{\Sigma,\tau}(\mathcal{X},\Gamma)\}_{\tau \in T_{\Omega}(\mathcal{X})}$$

In addition we define the following abbreviations:

$$T_{\Sigma}(\mathcal{X}) = T_{\Sigma}(\mathcal{X},\emptyset) \quad \text{(closed object terms)}$$

$$T_{\Sigma} = T_{\Sigma}(\emptyset) \quad \text{(non-polymorphic closed object terms)}$$

□

Considering a well formed term $(\mathcal{X},\Gamma,e,\tau) \in T_{\Sigma,\tau}(\mathcal{X},\Gamma)$ we see that all the sort derivations $D : \Gamma \rhd_{\chi} e :: \tau$ for this term can only differ in the applications of

the formation rule (weak). Due to the vast type information contained in our pre terms e there are no other possibilities for different sort derivations.

In section 2.4 we will define the interpretation of a well formed term $(\mathcal{X}, \Gamma, e, \tau) \in T_{\Sigma,\tau}(\mathcal{X}, \Gamma)$ with respect to the inductive structure of some sort derivation for this term. To guarantee the uniqueness of our definition we now distinguish the unique and always existing normal form of a sort derivation.

Definition 2.12 Normal Sort Derivation

Let $(\mathcal{X}, \Gamma, e, \tau) \in T_{\Sigma,\tau}(\mathcal{X}, \Gamma)$ be a well formed term. The Normal Sort Derivation $ND : \Gamma \rhd_\chi e :: \tau$ is that derivation where introductions of sort assumptions via the formation rule (weak) occur as late as possible. □

A formal definition of the normal form together with a proof for the existence and uniqueness result is pretty obvious. A thorough discussion of a slightly different technique containing all the definitions and proofs can be found in [Mit93].

Next we define formulae $\mathbf{Form}(\Sigma, \mathcal{X}, \Gamma)$ and sentences $\mathbf{Sen}(\Sigma, \mathcal{X})$ over a polymorphic signature Σ and sort context $\mathcal{X}$. In SPECTRUM the set of formulae $\mathbf{Form}(\Sigma, \mathcal{X}, \Gamma)$ is the set of well formed terms in context $(\mathcal{X}, \Gamma)$ of sort Bool. This leads to a three valued logic. The sentences are as usual the closed formulae.

Definition 2.13 Formulae and Sentences

$$\mathbf{Form}(\Sigma, \mathcal{X}, \Gamma) = T_{\Sigma,\mathsf{Bool}}(\mathcal{X}, \Gamma)$$

$$\mathbf{Sen}(\Sigma, \mathcal{X}) = \mathbf{Form}(\Sigma, \mathcal{X}, \emptyset) \quad \text{(closed formulae are sentences)}$$

$$\mathbf{Sen}(\Sigma) = \mathbf{Sen}(\Sigma, \emptyset) \quad \text{(non-polymorphic sentences)}$$

□

Example 2.3

$$\forall^{\perp} x : \alpha.{=}[\alpha]\langle x, x\rangle \in \mathbf{Sen}(\Sigma, \{\alpha\})$$

$$\forall^{\perp} x : \mathbf{Nat}.{=}[\mathbf{Nat}]\langle x, x\rangle \in \mathbf{Sen}(\Sigma)$$

□

Definition 2.14 Specification

A polymorphic specification $S = (\Sigma, E)$ is a pair where $\Sigma = (\Omega, F, O)$ is a polymorphic signature and $E \subseteq \mathbf{Sen}(\Sigma, \mathcal{X})$ is a set of sentences for some sort context $\mathcal{X}$. □

2.2.2 Terms and Sentences Translation

Remember that a term is a quadruple $(\mathcal{X}, \Gamma, e, \tau)$. As a consequence we first have to define how we translate contexts and pre terms. Remember that sort terms translation was given in section 2.1.2

Definition 2.15 Sort Context Translation

Let $\mathcal{X}$ be a sort context:

$$\mathcal{X} = \{\mathcal{X}_k \mid k \in K\}$$

and $\omega : \Omega \to \Omega'$ be a sort signature morphism. The translation of this context by ω^* is defined as follows:

$$\begin{array}{lcl} \omega^*(\mathcal{X}) & = & \{\omega^*(\mathcal{X})_j \mid j \in K'\} \\ \omega^*(\mathcal{X})_j & = & \bigcup \{\mathcal{X}_k \mid \omega(k) = j\} \end{array}$$

In other words $x \in \mathcal{X}_k$ is translated by ω^* to $x \in \mathcal{X}'_{\omega(k)}$. □

Definition 2.16 Context Translation

Let Γ be a context:

$$\Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$$

The translation of this context by ω^* is defined as follows:

$$\omega^*(\Gamma) = \{x_1 : \omega^*(\tau_1), \ldots, x_n : \omega^*(\tau_n)\}$$

□

Definition 2.17 Pre Terms Translation

Let $\sigma : \Sigma \to \Sigma'$ be a signature morphism with $\sigma = (\omega, \sigma_F, \sigma_O)$. The

extension σ^* of σ to pre terms is defined inductively on their structure as follows:

$$\begin{array}{lcll}
\sigma^\star(x) & = & x, & variable \\
\sigma^\star(c) & = & \sigma_F(c), & c \in F_{T_{\Omega,\mathrm{cpo}}} \\
\sigma^\star(pf[\tau_1,\ldots,\tau_n]) & = & \sigma_F(pf)[\omega^*(\tau_1),\ldots,\omega^*(\tau_n)], & pf \in F_{T^{\Pi}_{\Omega,\mathrm{cpo}}} \\
\sigma^\star(oe) & = & \sigma_O(o)\sigma^\star(e), & o \in O_{T_{\Omega,\mathrm{ops}}} \\
\sigma^\star(po[\tau_1,\ldots,\tau_n]e) & = & \sigma_O(po)[\omega^*(\tau_1),\ldots,\omega^*(\tau_n)]\sigma^\star(e), & po \in O_{T^{\Pi}_{\Omega,\mathrm{ops}}} \\
\sigma^\star(\langle e_1,\ldots,e_n\rangle) & = & \langle\sigma^\star(e_1),\ldots,\sigma^\star(e_n)\rangle, & tuple \\
\sigma^\star(\lambda x:\tau.e) & = & \lambda x:\omega^*(\tau).\sigma^\star(e), & abstr \\
 & & & dito\ pat-abstr \\
\sigma^\star(Qx:\tau.e) & = & Qx:\omega^*(\tau).\sigma^\star(e), & quantifier \\
\sigma^\star((e)) & = & (\sigma^\star(e)), & priority
\end{array}$$

□

Now we can define well formed terms translation.

Definition 2.18 Well Formed Terms Translation

Let $(\mathcal{X},\Gamma,e,\tau)$ be a well formed term and $\sigma : \Sigma \to \Sigma'$ be a signature morphism with $\sigma = (\omega,\sigma_F,\sigma_O)$. The translation function is also denoted by σ^* and is defined as follows:

$$\sigma^*((\mathcal{X},\Gamma,e,\tau)) = (\omega^*(\mathcal{X}),\omega^*(\Gamma),\sigma^*(e),\omega^*(\tau))$$

□

2.3 The Mod Functor

2.3.1 Algebras

The following definitions are standard definitions of domain theory (see [Gun92]). We include them here to get a self-contained presentation.

Definition 3.19 Partial Order

A partial order $\mathcal{A}$ is a pair $(A,\leq)$ where A is a set and $(\leq) \subseteq A \times A$ is a reflexive, transitive and antisymmetric relation. □

Definition 3.20 Chain Complete Partial Order

A partial order $\mathcal{A}$ is ω-chain complete **iff** every chain $a_1 \leq \ldots \leq a_n \leq \ldots,\ n \in \mathbb{N}$ has a least upper bound in $\mathcal{A}$. We denote it by $\sqcup_{i\in\mathbb{N}} x_i$. □

Definition 3.21 Pointed Chain Complete Partial Order (PCPO)

A chain complete partial order $\mathcal{A}$ is pointed **iff** it has a least element. In the sequel we denote this least element by uu_A. □

Definition 3.22 Monotonic Functions

Let $\mathcal{A} = (A, \leq_A)$ and $\mathcal{B} = (B, \leq_B)$ be two PCPOs. A function[10] $f \in B^A$ is *monotonic* **iff**

$$d \leq_A d' \ \Rightarrow\ f(d) \leq_B f(d')$$

□

Definition 3.23 Continuous Functions:

A monotonic function f between PCPOs $\mathcal{A}$ and $\mathcal{B}$ is continuous **iff** for every ω-chain $a_1 \leq \ldots \leq a_n \leq \ldots$ in $\mathcal{A}$:

$$f(\bigsqcup_{i\in\mathbb{N}} a_i) = \bigsqcup_{i\in\mathbb{N}} f(a_i)$$

Since f is monotonic and $\mathcal{A}$ and $\mathcal{B}$ are PCPOs the least upper bound on the right hand side exists. □

Definition 3.24 Product PCPO

If $\mathcal{A} = (A, \leq_A)$ and $\mathcal{B} = (B, \leq_B)$ are two PCPOs then the product PCPO $\mathcal{A} \times \mathcal{B} = (A \times B, \leq_{A\times B})$ is defined as follows:

- $A \times B$ is the usual cartesian product of sets,
- $(d, e) \leq_{A\times B} (d', e')$ **iff** $(d \leq_A d) \wedge (e \leq_B e')$,
- $uu_{A\times B} = (uu_A, uu_B)$

This definitions may be generalized to n-ary products in a straight forward way.

□

Definition 3.25 Function PCPO

If $\mathcal{A} = (A, \leq_A)$ and $\mathcal{B} = (B, \leq_B)$ are two PCPOs then the function PCPO $\mathcal{A} \xrightarrow{c} \mathcal{B} = (A \xrightarrow{c} B, \leq_{A\xrightarrow{c}B})$ is defined as follows:

[10] We write B^A for all functions from A to B.

- $A \xrightarrow{c} B$ is the set of all continuous functions from A to B,
- $f \leq_{A \xrightarrow{c} B} g$ **iff** $\forall a \in A. f(a) \leq_B g(a)$,
- $uu_{A \xrightarrow{c} B} = \lambda x : A. uu_B$

□

Definition 3.26 Lift PCPO

If $\mathcal{A} = (A, \leq_A)$ is a PCPO then the lifted PCPO $\mathcal{A}$ **lift** $= (A$ **lift**$, \leq_{A \textbf{ lift}})$ is defined as follows:

- A **lift** $= (A \times \{0\}) \cup \{uu_{A \textbf{ lift}}\}$ where $uu_{A \textbf{ lift}}$ is a new element which is not a pair.
- $(x, 0) \leq_{A \textbf{ lift}} (y, 0)$ **iff** $x \leq_A y$

  $\forall z \in A$ **lift**$. uu_{A \textbf{ lift}} \leq_{A \textbf{ lift}} z$
- We also define an extraction function $\downarrow$ from A **lift** to A such that

  $$\downarrow uu_{A \textbf{ lift}} = uu_A \qquad ; \qquad \downarrow (x, 0) = x$$

□

We will call the PCPOs also domains (note that in the literature domains are usually algebraic directed complete po's [Gun92]).

The Sort Algebras

Definition 3.27 Sort–Algebras

Let $\Omega = (K, \leq, SC)$ be a sort–signature. An Ω–algebra $\mathcal{SA}$ is an order sorted algebra[11] of domains i.e.:

- For the kind $cpo \in K$ we have a set of domains $cpo^{\mathcal{SA}}$. For the kind $map \in K$ we have a set of full functions spaces $map^{\mathcal{SA}}$.
- For all kinds $k \in K$ with $k \leq cpo$ we have a nonempty subset $k^{\mathcal{SA}} \subseteq cpo^{\mathcal{SA}}$.
- For all kinds $k_1, k_2 \in K$ with $k_1 \leq k_2$ we have $k_1^{\mathcal{SA}} \subseteq k_2^{\mathcal{SA}}$.
- For each sort constructor $sc \in SC_{k_1 \ldots k_n, k}$ there is a domain constructor $sc^{\mathcal{SA}} : k_1^{\mathcal{SA}} \times \ldots \times k_n^{\mathcal{SA}} \to k^{\mathcal{SA}}$ such that if $sc \in SC_{w,s} \cap SC_{w',s'}$ and $w \leq w'$ then

  $$sc^{\mathcal{SA}}_{w',s'} \mid_{w^{\mathcal{SA}}} = sc^{\mathcal{SA}}_{w,s}$$

  In other words overloaded domain constructors must be equal on the smaller domain $w^{\mathcal{SA}} = k_1^{\mathcal{SA}} \times \ldots \times k_n^{\mathcal{SA}}$ where $w = k_1 \ldots k_n$.

[11] See [GM87, Gog76].

We further require the following interpretation for the sort constructors occurring in the standard sort–signature:

- $\mathsf{Bool}^{\mathcal{SA}} = (\{\mathit{ww}_{\mathsf{Bool}}, \mathit{ff}, \mathit{tt}\}, \leq_{\mathsf{Bool}})$ is the flat three–valued boolean domain.
- For $\times_n^{\mathcal{SA}} \in \mathit{cpo}^{\mathcal{SA}} \times \ldots \times \mathit{cpo}^{\mathcal{SA}} \to \mathit{cpo}^{\mathcal{SA}}$:

  $$\times_n^{\mathcal{SA}}(d_1, \ldots, d_n) = d_1 \times \ldots \times d_n,\ n \geq 2$$

  is the n-ary cartesian product of domains.
- For $\rightarrow^{\mathcal{SA}} \in \mathit{cpo}^{\mathcal{SA}} \times \mathit{cpo}^{\mathcal{SA}} \to \mathit{cpo}^{\mathcal{SA}}$:

  $$\rightarrow^{\mathcal{SA}}(d_1, d_2) = (d_1 \stackrel{c}{\rightarrow} d_2)\mathbf{lift}$$

  is the lifted domain of continuous functions. We lift this domain because we want to distinguish between $\perp$ and $\lambda x.\, \perp$.
- For $\mathbf{to}^{\mathcal{SA}} \in \mathit{cpo}^{\mathcal{SA}} \times \mathit{cpo}^{\mathcal{SA}} \to \mathit{map}^{\mathcal{SA}}$

  $$\mathbf{to}^{\mathcal{SA}}(d_1, d_2) = d_2^{d_1}$$

  is the full space of functions between d_1 and d_2.

□

Definition 3.28 Interpretation of sort terms

Let $\nu : \mathcal{X} \to \mathcal{SA}$ be a sort environment and $\nu^* : T_\Omega(\mathcal{X}) \to \mathcal{SA}$ its homomorphic extension. Then $\mathcal{SA}_{[\![\cdot]\!]}\nu$ is defined as follows:

- $\mathcal{SA}[\![e]\!]\nu = \nu^*(e)$ **if** $e \in T_\Omega(\mathcal{X})$
- $\mathcal{SA}[\![\Pi\alpha_1 : k_1, \ldots, \alpha_n : k_n.e]\!] =$
  $\{f \mid f(\nu(\alpha_1), \ldots, \nu(\alpha_n)) \in \mathcal{SA}[\![e]\!]\nu$ **for all** $\nu\}$

For closed terms we write for $\mathcal{SA}[\![e]\!]$ also $e^{\mathcal{SA}}$. □

Sort terms in T_Ω^Π are interpreted as generalized cartesian products (dependent products). By using *n-ary* dependent products we can interpret Π–terms in one step. This leads to simpler models as the ones for the polymorphic λ–calculus.

Polymorphic Algebras

Definition 3.29 Polymorphic Algebra

Let $\Sigma = (\Omega, F, O)$ be a polymorphic signature with $\Omega = (K, \leq, SC)$ the sort–signature. A polymorphic Σ–algebra $\mathcal{A} = (\mathcal{SA}, \mathcal{F}, \mathcal{O})$ is a triple where:

- $\mathcal{SA}$ is an Ω sort algebra,
- $\mathcal{F} = \{\mathcal{F}_\mu\}_{\mu \in T^{closed}_{\Omega,cpo}}$ is an indexed set of constants (or functions), with:

  $$\mathcal{F}_\mu = \{f^{\mathcal{A}} \in \mu^{\mathcal{SA}} \mid f \in F_\mu\}$$

  such that if $f \in F_\mu$ is not the constant $\bot \in \Pi\alpha : cpo.\, \alpha$ then its interpretation $f^{\mathcal{A}}$ is different from ww in $\mu^{\mathcal{SA}}$. If f is polymorphic then all its instances must be different from the corresponding least element.
- $\mathcal{O} = \{\mathcal{O}_\nu\}_{\nu \in T^{closed}_{\Omega,map}}$ is an indexed set of mappings, with:

  $$\mathcal{O}_\nu = \{o^{\mathcal{A}} \in \nu^{\mathcal{SA}} \mid o \in O_\nu\}$$

We further require a fixed interpretation for the symbols in the standard signature. In order to simplify notation we will write $f^{\mathcal{A}}_{d_1,\ldots,d_n}$ for the instance $f^{\mathcal{A}}(d_1,\ldots,d_n)$ of a polymorphic function and $o^{\mathcal{A}}_{d_1,\ldots,d_n}$ for the instance $o^{\mathcal{A}}(d_1,\ldots,d_n)$ of a polymorphic mapping.

- Predefined Mappings ($O_{standard}$):
  - $\{=, \sqsubseteq\} \subseteq O_{\Pi\alpha:cpo.\, \alpha\times\alpha \text{ to } \mathsf{Bool}}$ are interpreted as *identity* and *partial order*. More formally, for every domain $d \in cpo^{\mathcal{A}}$ and $x, y \in d$:

    $$x =^{\mathcal{A}}_d y := \begin{cases} \mathit{tt} & \text{if } x \text{ is identical to } y \\ \mathit{ff} & \text{otherwise} \end{cases}$$

    $$x \sqsubseteq^{\mathcal{A}}_d y := \begin{cases} \mathit{tt} & \text{if } x \leq_d y \\ \mathit{ff} & \text{otherwise} \end{cases}$$
  - $\{\delta\} \subseteq O_{\Pi\alpha:cpo.\, \alpha \text{ to } \mathsf{Bool}}$ is the polymorphic definedness predicate. For every $d \in cpo^{\mathcal{A}}$ and $x \in d$:

    $$\delta^{\mathcal{A}}_d(x) := \begin{cases} \mathit{tt} & \text{if } x \text{ is different from } \mathit{ww}_d \\ \mathit{ff} & \text{otherwise} \end{cases}$$
- Predefined Constants ($F_{standard}$):
  - $\{\mathsf{true}, \mathsf{false}\} \subseteq F_{\mathsf{Bool}}$ are interpreted in the $\mathsf{Bool}^{\mathcal{SA}}$ domain as follows:

    $$\mathsf{true}^{\mathcal{A}} = \mathit{tt} \quad ; \quad \mathsf{false}^{\mathcal{A}} = \mathit{ff}$$
  - The interpretations of $\{\neg\} \subseteq F_{\mathsf{Bool}\to\mathsf{Bool}}$, $\{\wedge, \vee, \Rightarrow\} \subseteq F_{\mathsf{Bool}\times\mathsf{Bool}\to\mathsf{Bool}}$ are pairs in the lifted function spaces such that the function components behave like three–valued Kleene connectives on $\mathsf{Bool}^{\mathcal{SA}}$

as follows:

x	y	$(\downarrow \neg^{\mathcal{A}})(x)$	$x(\downarrow \wedge^{\mathcal{A}})y$	$x(\downarrow \vee^{\mathcal{A}})y$	$x(\downarrow \Rightarrow^{\mathcal{A}})y$
tt	tt	ff	tt	tt	tt
tt	ff	ff	ff	tt	ff
ff	tt	tt	ff	tt	tt
ff	ff	tt	ff	ff	tt
uu	tt	uu	uu	tt	tt
uu	ff	uu	ff	uu	uu
uu	uu	uu	uu	uu	uu
tt	uu	ff	uu	tt	uu
ff	uu	tt	ff	uu	tt

- $\{\bot\} \subseteq F_{\Pi\alpha:cpo.\,\alpha}$ is interpreted in each domain as the least element of this domain. For every $d \in cpo^{\mathcal{SA}}$:

  $$\bot_d^{\mathcal{A}} := uu_d$$

- $\{\mathsf{fix}\} \subseteq F_{\Pi\alpha:cpo.\,(\alpha\to\alpha)\to\alpha}$ is interpreted for each domain d as a pair $\mathsf{fix}_d^{\mathcal{A}} \in (d \to^{\mathcal{SA}} d) \to^{\mathcal{SA}} d$ such that the function component behaves as follows:

  $$(\downarrow \mathsf{fix}_d^{\mathcal{A}})(f) := \bigsqcup_{i\in\mathbb{N}} f^n(uu_d)$$

  where:

  $$f^0(uu_d) := uu_d$$
  $$f^{n+1}(uu_d) := (\downarrow f)(f^n(uu_d))$$

  Note that $\downarrow uu_{(d \xrightarrow{c} d)\,\mathbf{lift}} = uu_{(d \xrightarrow{c} d)}$ and therefore the above definition is sound.

□

2.3.2 The Homomorphisms

Homomorphisms are used to relate algebras over the same signature. They are in our framework different from the algebraic homomorphisms because they are not only required to be compatible with function application but also with function abstraction. This condition is usual for applicative structures and allows inductive reasoning on the term structure (see [Mit90, MM85]).

Similarly with polymorphic algebras, polymorphic homomorphisms are built in two steps, first starting with a homomorphism between sort algebras. More formally:

Definition 3.30 Sort Homomorphisms

Let $\mathcal{SA}$ and $\mathcal{SB}$ be two Ω sort algebras, with $\Omega = (K, \leq, SC)$. An Ω–homomorphism $s : \mathcal{SA} \to \mathcal{SB}$ is an order sorted homomorphism between

sort algebras satisfying:

$$\begin{array}{lcll} s & = & \{s_k : k^{\mathcal{SA}} \rightarrow k^{\mathcal{SB}} \mid k \in K\} & \\ s_k(sc^{\mathcal{SA}}) & = & sc^{\mathcal{SB}}, & sc \in SC_k \\ s_k(sc^{\mathcal{SA}}(d_1,\ldots,d_n)) & = & sc^{\mathcal{SB}}(s_{k_1}(d_1),\ldots,s_{k_n}(d_n)), & sc \in SC_{k_1\ldots k_n,k} \\ k_1 \leq k_2 \Rightarrow \quad s_{k_1} & = & s_{k_2}\,|_{k_1^{\mathcal{SA}}} & \end{array}$$

where $d_1 \in k_1^{\mathcal{SA}}, \ldots, d_n \in k_n^{\mathcal{SA}}$. □

Sort algebras and sort homomorphisms form a category.

Definition 3.31 Polymorphic Homomorphisms

Let $\mathcal{A}_\Sigma = (\mathcal{SA}, \mathcal{F}, \mathcal{O})$ and $\mathcal{B}_\Sigma = (\mathcal{SB}, \mathcal{F}', \mathcal{O}')$ be two Σ–algebras, with $\Sigma = (\Omega, F, O)$ and $\Omega = (K, \leq, SC)$. A polymorphic homomorphism $\mathcal{H} = (s, h)$ is a pair with s an Ω–homomorphism and $h = \{h_d : d \rightarrow s(d) \mid d \in k^{\mathcal{SA}} \wedge k \in K\}$ an indexed function between domains. This function is

logical on constants:

- It *preserves constants* i.e. for every non polymorphic constant $f \in F_\sigma$, polymorphic constant $pf \in F_{\Pi\alpha_1:k_1,\ldots,\alpha_n:k_n.\tau}$ and sort environment $\nu : \mathcal{X} \rightarrow \mathcal{SA}$ such that $\nu_{k_1}(\alpha_1) = d_1, \ \ldots, \ \nu_{k_n}(\alpha_n) = d_n$ and $d = \mathcal{SA}[\![\tau]\!]\nu$ the following holds:

  $h_{\sigma^{\mathcal{SA}}}(f^{\mathcal{SA}}) = f^{\mathcal{SB}}$
  $h_d(pf^{\mathcal{SA}}[d_1,\ldots,d_n]) = pf^{\mathcal{SB}}[s_{k_1}(d_1),\ldots,s_{k_n}(d_n)]$

- It is *compatible both with application and abstraction* i.e. for all functions $f \in d \rightarrow^{\mathcal{SA}} e$ and $g \in s(d) \rightarrow^{\mathcal{SB}} s(e)$ the following holds:

  $h_{d\rightarrow^{\mathcal{SA}} e}(f) = g$ **iff** $\forall a \in d.\ h_e(\downarrow f\ a) = \downarrow g\ h_d(a)$

and *algebraic on operations* :

- It is *compatible with application* i.e. for every non polymorphic operation $o \in O_{\sigma \text{ to } \tau}$, polymorphic operation $po \in O_{\Pi\alpha_1:k_1,\ldots,\alpha_n:k_n.\sigma' \text{ to } \tau'}$, sort environment $\nu : \mathcal{X} \rightarrow \mathcal{SA}$ such that $\nu_{k_1}(\alpha_1) = d_1, \ldots, \nu_{k_n}(\alpha_n) = d_n$, $d = \mathcal{SA}[\![\sigma']\!]\nu$, $e = \mathcal{SA}[\![\tau']\!]\nu$ and elements $a \in \sigma^{\mathcal{SA}}$, $b \in d$ the following holds:

  $h_{\tau^{\mathcal{SA}}}(o^{\mathcal{A}}(a)) = o^{\mathcal{B}}(h_{\sigma^{\mathcal{SA}}}(a))$
  $h_e(po^{\mathcal{A}}[d_1,\ldots,d_n](b)) = po^{\mathcal{B}}[s(d_1),\ldots,s(d_n)](h_d(b))$

□

Homomorphisms are less useful in our framework because they are always required to be bijective.

- *Surjectivity* is imposed by the right to left direction of the **iff** condition (compatibility with abstraction). Suppose $g, g' : s(d) \to s(e)$ and $\forall a \in d.\ h_e(\downarrow f\ a) = \downarrow g\ h_d(a)$. If $\forall a \in d.\ \downarrow g\ h_d(a) = \downarrow g'\ h_d(a)$ then both $h(f) = g$ and $h(f) = g'$. Since h is a function $g = g'$. However if h was not surjective we could have easily constructed two functions g, g' which are distinct on $s(d)$ but equal on the image $h(d)$ of d under h.

- *Injectivity* is imposed by the left to right direction of the **iff** condition (compatibility with application) and the identification of Bool with Truth. Since Bool has in every model only the values $\{\bot, ff, tt\}$, then for every function $f^{\mathcal{SA}} : s^{\mathcal{SA}} \to \mathsf{Bool}^{\mathcal{SA}}$ and every $a \in s^{\mathcal{SA}}$ the value $f(a)$ must be either $\bot$, ff or tt. Hence identification of values in $s^{\mathcal{SA}}$ (congruences) cannot be accomplished by identification of values in $\mathsf{Bool}^{\mathcal{SA}}$. For example, suppose $a, b \in s^{\mathcal{SA}}$, are distinct elements in $\mathcal{SA}$ i.e. $(a ==^{\mathcal{SA}} b) = ff$. Now, if h is not injective and it identifies a and b i.e. $(h(a) ==^{\mathcal{SB}} h(b)) = tt$ then we get a contradiction. On the one hand $h(==^{\mathcal{SA}}) = (==^{\mathcal{SB}})$ implies that $h(a ==^{\mathcal{SA}} b) = (h(a) ==^{\mathcal{SB}} h(b)) = tt$. On the other hand $h(a ==^{\mathcal{SA}} b) = h(ff) = h(\mathsf{false}^{\mathcal{SA}}) = \mathsf{false}^{\mathcal{SB}} = ff$.

2.3.3 The Reduct

Signature morphisms are used not only to translate sentences over one signature into sentences over another signature but also to relate algebras over different signatures. Having a signature morphism $\sigma : \Sigma \to \Sigma'$ we can forget some structure of the algebras and homomorphisms over the signature Σ' and obtain algebras and homomorphisms over the the signature Σ. This construction is known as *reduct* and it is also done in two steps.

Definition 3.32 Sort–Reduct

Let $\omega : \Omega \to \Omega'$ be a a sort morphism between $\Omega = (K, \leq, SC)$ and $\Omega' = (K', \leq', SC')$. The *sort reduct functor* $\overline{\omega} : \mathbf{Mod}(\Omega') \to \mathbf{Mod}(\Omega)$ is defined as follows:

- For every Ω'–sort algebra $\mathcal{SA}'$ the algebra $\overline{\omega}(\mathcal{SA}')$ is a Ω–sort algebra defined by:

$$\begin{array}{lcll} k^{\overline{\omega}(\mathcal{SA}')} & = & \omega(k)^{\mathcal{SA}'} & k \in K \\ sc^{\overline{\omega}(\mathcal{SA}')}_{k_1 \ldots k_n, k} & = & \omega(sc)^{\mathcal{SA}'}_{\omega^*(k_1 \ldots k_n), \omega(k)} & sc \in SC_{k_1 \ldots k_n, k} \end{array}$$

- For every Ω' sort homomorphism $s' = \{s'_k : k^{\mathcal{SA}'} \to k^{\mathcal{SB}'} \mid k \in K'\}$ the homomorphism $\overline{\omega}(s')$ is a Ω sort homomorphism defined by:

$$\begin{array}{lcl} \overline{\omega}(s') & = & \{\overline{\omega}(s')_k : k^{\overline{\omega}(\mathcal{SA}')} \to k^{\overline{\omega}(\mathcal{SB}')} \mid k \in K\} \\ \overline{\omega}(s')_k & = & s'_{\omega(k)} \end{array}$$

□

We now define the polymorphic reducts.

Definition 3.33 Polymorphic–Reduct

Let $\sigma : \Sigma \to \Sigma'$ be a signature morphism between $\Sigma = (\Omega, F, O)$ and $\Sigma' = (\Omega', F', O')$. The *polymorphic reduct* $\overline{\sigma} : \mathbf{Mod}(\Sigma') \to \mathbf{Mod}(\Sigma)$ is defined as follows:

- For every Σ' polymorphic algebra $\mathcal{A}'$ the algebra $\overline{\sigma}(\mathcal{A}')$ is a Σ polymorphic algebra and it is defined by:

$$\begin{array}{lcll} \overline{\sigma}(\mathcal{S}\mathcal{A}') & = & \overline{\omega}(\mathcal{S}\mathcal{A}') & \\ f_{\mu}^{\overline{\sigma}(\mathcal{A}')} & = & \sigma(f)^{\mathcal{A}'}_{\sigma^*(\mu)} & f_\mu \in F_\mu \\ o_{\nu}^{\overline{\sigma}(\mathcal{A}')} & = & \sigma(o)^{\mathcal{A}'}_{\sigma^*(\nu)} & o_\nu \in O_\nu \end{array}$$

- For every Σ'–homomorphism $\mathcal{H}' = (s', h')$ the homomorphism $\overline{\sigma}(\mathcal{H}') = (s, h)$ is a Σ–homomorphism and it is defined by:

$$\begin{array}{lcl} s & = & \overline{\omega}(s') \\ h & = & \{h_d : d \to s(d) \mid d \in k^{\overline{\omega}(\mathcal{S}\mathcal{A}')} \wedge k \in K\} \\ h_d & = & h'_d \end{array}$$

□

Example 3.4 Reducts

As an example consider that we have the following signatures and signature morphism[12]:

Σ = {
  sort σ_1, σ_2 :: k_1;
  sort rc :: (k_1) k_2;
  f : $\sigma_1 \rightarrowtail \sigma_2$;
}

Σ' = {
  sort τ_1, τ_2 :: l_1;
  sort sc :: (l_1) l_2;
  g : $\tau_1 \rightarrowtail \tau_2$;
}

σ = [
  k_1 **to** l_1, k_2 **to** l_2,
  σ_1 **to** τ_1, σ_2 **to** τ_2,
  f **to** g
]

[12] In order to improve readability we use a the SPECTRUM syntax instead of the abstract definition.

Suppose that $\mathcal{A}$ and $\mathcal{B}$ are Σ'–algebras and that $\mathcal{H} = (s, h)$ is a Σ'–homorpism between $\mathcal{A}$ and $\mathcal{B}$. Then we can get the Σ–algebras $\overline{\sigma}(\mathcal{A})$ and $\overline{\sigma}(\mathcal{B})$ and the Σ–homorphism $\overline{\sigma}(\mathcal{H})$ as shown in the figure 2.2. In this figure $d = \sigma_1^{\overline{\sigma}(\mathcal{SA})} = \tau_1^{\mathcal{SA}}$ and $e = \sigma_2^{\overline{\sigma}(\mathcal{SA})} = \tau_2^{\mathcal{SA}}$.

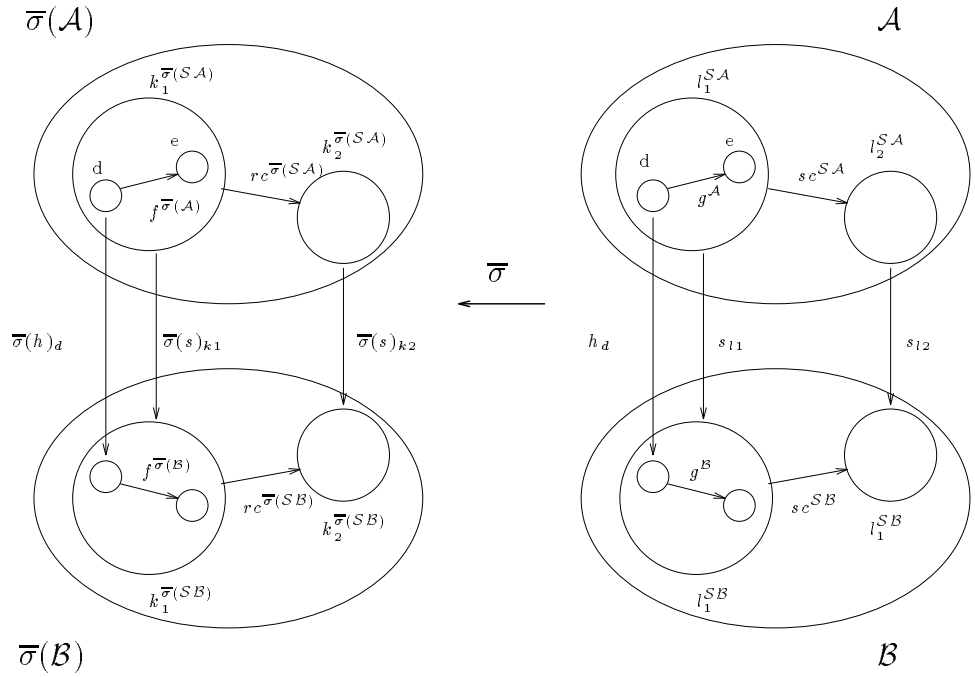

Figure 2.2: Graphical Presentation of Reducts

□

2.4 Models

2.4.1 Interpretation of sort assertions

In this section we define the interpretation of well-formed terms. The interpretation of $(\mathcal{X}, \Gamma, e, \tau) \in T_{\Sigma,\tau}(\mathcal{X}, \Gamma)$ is defined inductively on the structure of the normal sort derivation $ND : \Gamma \rhd_\chi e :: \tau$. The technique used is again due to [Mit90].

Definition 4.34 Satisfaction of a variable context

Let $\Sigma = (\Omega, F, O)$ be a polymorphic signature with $\Omega = (K, \leq, SC)$ and let $\mathcal{A} = (\mathcal{SA}, \mathcal{F}, \mathcal{O})$ be a a polymorphic Σ–algebra.

If Γ is a variable context and

$$\begin{array}{ll} \nu = \{\nu_k : \mathcal{X}_k \to k^{\mathcal{SA}}\}_{k \in K \setminus \{map\}} & \text{sort environment (order-sorted)} \\ \eta : \psi \to \bigcup_{d \in cpo^{\mathcal{SA}}} d & \text{object environment (unsorted)} \end{array}$$

then η satisfies Γ in sort environment ν (in symbols $\eta \models_\nu \Gamma$) iff

$$\eta \models_\nu \Gamma \Leftrightarrow \text{for all } x : \tau \in \Gamma . \eta(x) \in \nu^\star(\tau)$$

□

Definition 4.35 Update of object environments

$$\eta[a/x](y) := \begin{cases} a & \text{if } x = y \\ \eta(y) & \text{otherwise} \end{cases}$$

□

Now we define an order sorted meaning function $\mathcal{A}[\![\cdot]\!]_{\nu,\eta}$ that maps normal sort derivations $ND : \Gamma \rhd_\chi e :: \tau$ to elements in $\mathcal{A}$. Since normal sort derivations always exist and are unique this leads to a total meaning function $\mathcal{A}[\![\cdot]\!]_{\nu,\eta} : T_\Sigma(\mathcal{X}, \Gamma) \to \mathcal{A}$.

Definition 4.36 Meaning of a sort derivation

The meaning of a normal sort derivation $ND : \Gamma \rhd_\chi e :: \tau$ in a polymorphic algebra $\mathcal{A}$ in sort context ν and variable context Γ such that $\eta \models_\nu \Gamma$ is $\mathcal{A}[\![ND : \Gamma \rhd_\chi e :: \tau]\!]_{\nu,\eta}$ which is recursively defined on the structure of ND. The defining clauses are given below. □

Base cases:

$$\text{(var) } \mathcal{A}[\![x : \tau \rhd_\chi x :: \tau]\!]_{\nu,\eta} = \eta(x) \qquad \text{(const) } \mathcal{A}[\![\emptyset \rhd_\chi c :: \tau]\!]_{\nu,\eta} = c^{\mathcal{A}}$$

(Π-inst)

$$\mathcal{A}[\![\emptyset \rhd_\chi f[\tau_1, \ldots, \tau_n] :: \tau]\!]_{\nu,\eta} = f^{\mathcal{A}}(\nu^\star(\tau_1), \ldots, \nu^\star(\tau_n))$$

Inductive cases:

(weak)

$$\mathcal{A}[\![\Gamma \cup \{x_1:\tau_1,\ldots,x_n:\tau_n\} \rhd_\chi e :: \tau]\!]_{\nu,\eta} = \mathcal{A}[\![\Gamma \rhd_\chi e :: \tau]\!]_{\nu,\eta}$$

(map-appl)

$$\mathcal{A}[\![\Gamma \rhd_\chi oe :: \tau_2]\!]_{\nu,\eta} = o^{\mathcal{A}}(\mathcal{A}[\![\Gamma \rhd_\chi e :: \tau_1]\!]_{\nu,\eta})$$

(Πmap-appl)

$$\mathcal{A}[\![\Gamma \rhd_\chi o[\tau_1,\ldots,\tau_n]e :: \sigma_2]\!]_{\nu,\eta} = o^{\mathcal{A}}(\nu^\star(\tau_1),\ldots,\nu^\star(\tau_n))(\mathcal{A}[\![\Gamma \rhd_\chi e :: \sigma_1]\!]_{\nu,\eta})$$

(tuple)

$$\mathcal{A}[\![\Gamma \rhd_\chi \langle e_1,\ldots,e_n\rangle :: \tau_1\times\ldots\times\tau_n]\!]_{\nu,\eta} = (\mathcal{A}[\![\Gamma \rhd_\chi e_1 :: \tau_1]\!]_{\nu,\eta},\ldots,\mathcal{A}[\![\Gamma \rhd_\chi e_n :: \tau_n]\!]_{\nu,\eta})$$

(abstr)[13]

$$\mathcal{A}[\![\Gamma \rhd_\chi \lambda x:\tau_1.e :: \tau_1\rightarrow\tau_2]\!]_{\nu,\eta} = \text{the } \underline{\text{unique}} \text{ pair } (f,0) \in \nu^\star(\tau_1\rightarrow\tau_2) \text{ with } \forall a \in \nu^\star(\tau_1).f(a) = \mathcal{A}[\![\Gamma, x:\tau_1 \rhd_\chi e :: \tau_2]\!]_{\nu,\eta[a/x]}$$

(patt-abstr)

$$\mathcal{A}[\![\Gamma \rhd_\chi \lambda\langle x_1:\tau_1,\ldots,x_n:\tau_n\rangle.e :: \tau_1\times\ldots\times\tau_n\rightarrow\tau]\!]_{\nu,\eta} = \text{the } \underline{\text{unique}} \text{ pair } (f,0) \in \nu^\star(\tau_1\times\ldots\times\tau_n\rightarrow\tau) \text{ with } \forall a_1 \in \nu^\star(\tau_1),\ldots,a_n \in \nu^\star(\tau_n).f((a_1,\ldots,a_n)) = \mathcal{A}[\![\Gamma, x_1:\tau_1,\ldots,x_n:\tau_n \rhd_\chi e :: \tau]\!]_{\nu,\eta[a_1/x_1,\ldots,a_n/x_n]}$$

(appl)

$$\mathcal{A}[\![\Gamma \rhd_\chi e_1e_2 :: \tau_2]\!]_{\nu,\eta} = \downarrow (\mathcal{A}[\![\Gamma \rhd_\chi e_1 :: \tau_1\rightarrow\tau_2]\!]_{\nu,\eta})(\mathcal{A}[\![\Gamma \rhd_\chi e_2 :: \tau_1]\!]_{\nu,\eta})$$

[13] the $\dagger$-test ensures that the clauses for (abstr) and (patt-abstr) are well defined.

(universal quantifier)

$$\mathcal{A}[\![\Gamma \rhd_\chi \forall^\perp x:\tau.e :: \mathsf{Bool}]\!]_{\nu,\eta} =$$

$$= \begin{cases} tt & \text{if } \forall a \in \nu^\star(\tau).(\mathcal{A}[\![\Gamma, x:\tau \rhd_\chi e :: \mathsf{Bool}]\!]_{\nu,\eta[a/x]} = tt) \\ ff & \text{if } \exists a \in \nu^\star(\tau).(\mathcal{A}[\![\Gamma, x:\tau \rhd_\chi e :: \mathsf{Bool}]\!]_{\nu,\eta[a/x]} = ff) \\ uu & \text{otherwise} \end{cases}$$

(existential quantifier)

$$\mathcal{A}[\![\Gamma \rhd_\chi \exists^\perp x:\tau.e :: \mathsf{Bool}]\!]_{\nu,\eta} =$$

$$= \begin{cases} tt & \text{if } \exists a \in \nu^\star(\tau).(\mathcal{A}[\![\Gamma, x:\tau \rhd_\chi e :: \mathsf{Bool}]\!]_{\nu,\eta[a/x]} = tt) \\ ff & \text{if } \forall a \in \nu^\star(\tau).(\mathcal{A}[\![\Gamma, x:\tau \rhd_\chi e :: \mathsf{Bool}]\!]_{\nu,\eta[a/x]} = ff) \\ uu & \text{otherwise} \end{cases}$$

2.4.2 Satisfaction and Models

In this subsection we define the satisfaction relation for boolean terms and sentences (closed boolean terms) and also the notion of a model.

Definition 4.37 Satisfaction

Let

$\mathcal{A} = (\mathcal{SA}, \mathcal{F}, \mathcal{O})$	Σ-Algebra
$\nu = \{\nu_k : \mathcal{X}_k \to k^{\mathcal{SA}}\}_{k \in K \setminus \{map\}}$	sort environment (order-sorted)
$\eta : \psi \to \bigcup_{d \in cpo^{\mathcal{SA}}} d$	object environment (unsorted)

and Γ a variable context with $\eta \models_\nu \Gamma$ then:

$\mathcal{A}$ satisfies $(\mathcal{X}, \Gamma, e, \mathsf{Bool}) \in \mathbf{Form}(\Sigma, \mathcal{X}, \Gamma)$ wrt. sort environment ν and object environment η (in symbols $\mathcal{A} \models_{\nu,\eta} (\mathcal{X}, \Gamma, e, \mathsf{Bool})$) iff

$$\mathcal{A} \models_{\nu,\eta} (\mathcal{X}, \Gamma, e, \mathsf{Bool}) \Leftrightarrow \mathcal{A}[\![\Gamma \rhd_\chi e :: \mathsf{Bool}]\!]_{\nu,\eta} = tt$$

A special case of the above definition is the satisfaction of sentences. Let $(\mathcal{X}, \emptyset, e, \mathsf{Bool}) \in \mathbf{Sen}(\Sigma, \mathcal{X})$ and η_0 an arbitrary environment, then:

$$\mathcal{A} \models_\nu (\mathcal{X}, \emptyset, e, \mathsf{Bool}) \Leftrightarrow \mathcal{A}[\![\emptyset \rhd_\chi e :: \mathsf{Bool}]\!]_{\nu,\eta_0} = tt$$

$$\mathcal{A} \models (\mathcal{X}, \emptyset, e, \mathsf{Bool}) \Leftrightarrow \mathcal{A} \models_\nu (\mathcal{X}, \emptyset, e, \mathsf{Bool}) \text{ for every } \nu$$

□

The satisfaction relation is invariant under translation. More precisely the following institution property holds:

Lemma 4.1 Satisfaction Invariance

Let $(\mathcal{X}, \Gamma, e, \mathsf{Bool})$ be a boolean term over Σ and $\sigma\colon \Sigma \longrightarrow \Sigma'$ a signature morphism. Let $\mathcal{X}' = \sigma^*(\mathcal{X})$ and $\Gamma' = \sigma^*(\Gamma)$ be the translations of $\mathcal{X}$ and Γ under σ. Let $\mathcal{A}$ be a Σ' algebra, ν a sort environment for $\mathcal{X}'$ in $\mathcal{A}$ and η an object environment for Γ' in $\mathcal{A}$. Define $\overline{\nu} = \nu \circ \sigma^*$ and $\overline{\eta} = \eta \circ \sigma^*$. Then the following holds:

$$\overline{\sigma}(\mathcal{A}) \models_{\overline{\nu},\overline{\eta}} (\mathcal{X}, \Gamma, e, \mathsf{Bool}) \Leftrightarrow \mathcal{A} \models_{\nu,\eta} \sigma^*((\mathcal{X}, \Gamma, e, \mathsf{Bool}))$$

Proof Sketch

The theorem is a particular case of the more general formula:

$$\overline{\sigma}(\mathcal{A})\left[\!\left[\Gamma \rhd_\chi e :: \tau\right]\!\right]_{\overline{\nu},\overline{\eta}} = \mathcal{A}\left[\!\left[\sigma^*(\Gamma \rhd_\chi e :: \tau)\right]\!\right]_{\nu,\eta}$$

This formula is proved by induction on the object terms derivation. In order to do this proof we need a similar condition for the sort terms, namely:

$$\overline{\varpi}(\mathcal{A})[\![\tau]\!]\overline{\nu} = \mathcal{A}\left[\!\left[\sigma^*(\tau)\right]\!\right]\nu$$

This condition is also proved by induction, on the sort terms structure. It also holds for π sort terms. □

Now we are able to define models $\mathcal{A}$ of specifications $S = (\Sigma, E)$.

Definition 4.38 Models

Let $S = (\Sigma, E)$ be a specification. A polymorphic Σ-algebra $\mathcal{A}$ is a model of S (in symbols $\mathcal{A} \models S$) **iff**

$$\mathcal{A} \models S \Leftrightarrow \forall p \in E.\, \mathcal{A} \models p$$

□

Chapter 3

Conclusion and acknowledgement

3.1 Conclusions

We have presented the semantics of the kernel part of the SPECTRUM language. Our work differs in many respects from other approaches. In contrast to LCF we allow the use of type classes. Moreover arbitrary non continuous functions can be used for specification purposes. This also permits to handle predicates and boolean functions in a uniform manner. In contrast with other semantics for polymorphic lambda calculus (e.g. [Mit90]) we did not provide an explicit type binding operator on the object level. This is not a restriction for languages having an ML–like polymorphism but allows a more simple treatment of the sort language. More precisely we used order sorted algebras instead of the more complex applicative structures. Order sorted algebras were also essential in the description of type classes.

3.2 Acknowledgement

For comments on draft versions and stimulating discussions we like to thank M. Löwe, B. Möller, F. Nickl, B. Reus, D. Sannella, T. Streicher, A. Tarlecki, M. Wirsing and U. Wolter.

Special thanks go also to our colleagues H. Hussmann, C. Facchi, R. Hettler and D. Nazareth to Tobias Nipkow whose work inspired our treatment of type classes and to Manfred Broy whose role was decisive in the design of the SPECTRUM language.

Bibliography

[BFG+93a] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, O. Slotosch, and K. Stølen. The Requirement and Design Secification Language SPECTRUM. An Informal Introduction. Version 1.0. Part I. Technical Report TUM-I9311, Technische Universität München. Institut für Informatik, May 1993.

[BFG+93b] M. Broy, C. Facchi, R. Grosu, R. Hettler, H. Hussmann, D. Nazareth, F. Regensburger, O. Slotosch, and K. Stølen. The Requirement and Design Secification Language SPECTRUM. An Informal Introduction. Version 1.0. Part II. Technical Report TUM-I9312, Technische Universität München. Institut für Informatik, May 1993.

[BG84] R. M. Burstall and J. A. Goguen. Introducing institutions. volume 164 of *LNCS*. 11984.

[Bro88] M. Broy. Requirement and Design Specification for Distributed Systems. *LNCS*, 335:33–62, 1988.

[CW85] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–523, December 1985.

[CWMG79] R. Milner C Wadsworth M. Gordon. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *LNCS*. Springer, 1979.

[Gau86] M.-C. Gaudel. Towards Structured Algebraic Specifications. *ESPRIT '85', Status Report of Continuing Work (North-Holland)*, pages 493–510, 1986.

[GHW85] J.V. Guttag, J.J. Horning, and J.M. Wing. Larch in Five Easy Pieces. Technical report, Digital, Systems Research Center, Paolo Alto, California, 1985.

[GM87] J.A. Goguen and J. Meseguer. Order–Sorted Algebra Solves the Constructor–Selector, Multiple Representation and Coercion Problems. In *Logic in Computer Science*, IEEE, 1987.

[Gog76] M. Gogolla. Partially Ordered Sorts in Algebraic Specifications. In B. Courcelle, editor, *Proc. 9th CAAP 1984, Bordeaux*. Cambridge University Press, 1976.

[Gun92] C. A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. MIT Press, 1992.

[HJW92] P. Hudak, S. Peyton Jones, and P. Wadler, editors. *Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2)*. ACM SIGPLAN Notices, May 1992.

[Mit90] J.C. Mitchell. Type Systems for Programming Languages. In *Handbook of Theoretical Computer Science*, chapter 8, pages 365–458. Elsevier Science Publisher, 1990.

[Mit93] J. C. Mitchell. *Introduction to Programming Language Theory*. MIT Press, 1993.

[MM85] J.C. Mitchell and A.R. Meyer. Second–Order Logical Relations. In *Logic of Programs*, volume 193 of *LNCS*, pages 225–236, Berlin, June 1985. Springer–Verlag.

[Nip93] T. Nipkow. Order-Sorted Polymorphism in Isabelle. In G. Huet and G. Plotkin, editors, *Logical Environments*, pages 164–188. CUP, 1993.

[NP] Tobias Nipkow and Christian Prehofer. Type checking type classes. In *Proc. 20th ACM Symp. Principles of Programming Languages*, pages 409–418.

[Pau84] L. Paulson. *Deriving Structural Induction in LCF*, volume 173 of *LNCS*. Springer, 1984.

[Reg94] F. Regensburger. *HOLCF: Eine konservative Erweiterung von HOL durch LCF*. PhD thesis, Technische Universität München, 1994. to appear.

[SNGM89] G. Smolka, W. Nutt, J. Goguen, and J. Meseguer. Order-Sorted Equational Computation. In *Resolution of Equations in Algebraic Structures*. Academic Press, 1989.

[Str67] C. Strachey. Fundamental Concepts in Programming Languages. In *Lecture Notes for International Summer School in Computer Programming*, Copenhagen, 1967.

[SW83] D. Sannella and M. Wirsing. A Kernel Language for Algebraic Specification and Implementation. Technical Report CSR-131-83, University of Edinburgh, Edinburgh EH9 3JZ, September 1983.

[WB89] P. Wadler and S. Blott. How to Make Ad-hoc Polymorphism Less Ad hoc. In *16th ACM Symposium on Principles of Programming Languages*, pages 60–76, 1989.