Types, Subtypes, and ASL+

David Aspinall

Department of Computer Science, University of Edinburgh, U.K.
e-mail: `da@dcs.ed.ac.uk`

Abstract. ASL+ is a formalism for specification and programming in-the-large, based on an arbitrary institution. It has rules for proving the satisfaction and refinement of specifications, which can be seen as a type theory with subtyping, including *contravariant refinement* for Π-abstracted specifications and a notion of *stratified equality* for higher-order objects. We describe the syntax of the language and a partial equivalence relation semantics. This style of semantics is familiar from subtyping calculi, but a novelty here is the use of a hierarchy of typed domains instead of a single untyped domain. We introduce the formal system for proving satisfaction and refinement and describe how it is linked to proof systems of the underlying programming and specification languages.

1 Introduction

There is a simple correspondence between the worlds of type theory and algebraic specification:

$$\begin{array}{ccc} \textit{elementhood},\ M : A & \Longleftrightarrow & \textit{satisfaction},\ M \in \textit{Mod}(A) \\ \textit{subtyping},\ A \leq A' & \Longleftrightarrow & \textit{refinement},\ A' \rightsquigarrow A \end{array}$$

The *elementhood* of a term M in a type A is like *satisfaction* of a specification A by a program M; to say that a type A is a *subtype* of another type A' is like saying that the specification A is a *refinement* of the specification A'. This connection holds because we take a model-theoretic approach to specification semantics: a specification amounts to the collection of models that are permissible implementations, and the refinement $A' \rightsquigarrow A$ just means that every model of A is also a model of A'. The difference is one of syntax and semantics: type theory is a game of symbol-pushing which often begins from syntactically defined relationships, whereas in our view of algebraic

This paper appears in *Recent Trends in Data Type Specification*, Springer Lecture Notes in Computer Science 906, 1995.

specification a model-theoretic semantics is paramount, and we search for *sound* proof-theoretic counterparts afterward.

An example of this is a formal system introduced by Sannella, Sokołowski, and Tarlecki [15] for proving satisfaction in ASL+, an extended version of the ASL specification language. The rules of their formal system are remarkably similar to those of a subtyping system with *power types*, designed by Cardelli as a programming language type-system [5]. Cardelli's system has terms of the form $Power(A)$, which stands for the type of all subtypes of A. He uses the elementhood $A : Power(A')$ to stand for the subtyping $A \leq A'$. In ASL+, there is a corresponding term written $Spec(A)$, and $A : Spec(A')$ is the syntactic counterpart of $A' \rightsquigarrow A$. Like Cardelli's system, ASL+ also has λ and Π-abstractions; we use them for expressing parameterised programs and specifications, and specifications of parameterised programs (an example follows below). The similarity between the systems of Cardelli and Sannella *et al.* is intriguing, because they were designed in isolation and from quite different motivations. It is explained, of course, by the correspondence above.

The work reported here is part of an attempt to consciously exploit this connection, by applying recent ideas from type theory to help understand and develop ASL+. The original treatment of the language and its formal systems was restricted in scope and depth. In particular, it turns out that the satisfaction system has a surprisingly complex meta-theory due to its combination of several concepts. The main focus of this paper, however, is on a new syntax and semantics for ASL+, which extend and improve the original treatment. The remainder of this section contains a brief overview of the framework; further explanation and motivations can be found in [15]. Section 2 describes the formal syntax of ASL+ and the kind-checking system used to define the semantics. In Section 3 we describe the semantics based on partial equivalence relations. Section 4 provides only an outline of the formal system for proving satisfaction; a fuller treatment will appear in [2]. Section 5 concludes.

1.1 Overview of the ASL+ framework

ASL+ is proposed as a kernel language for writing programs and specifications in-the-large; it extends the λ-calculus parameterisation mechanism of ASL. Suppose we wish to specify symbol tables for various kinds of lookup key: let KEY be a specification of a datatype of keys. A *parameterised specification* of symbol tables has the form:

$$SYMTAB =_{\mathrm{def}} \lambda X \leq KEY \ . \ \ldots$$

Here the actual parameter that instantiates the variable X is required to be a specification that is a refinement of KEY. The body of the abstraction (indicated by dots) is a specification written using the parameter X. We can also write a *parameterised program* implementing symbol tables, parameterised on a program that implements the datatype KEY (this is like a *functor* in Standard ML):

$$SymTab =_{\mathrm{def}} \lambda X : KEY \,.\, \ldots$$

But $SymTab$ isn't directly an implementation of $SYMTAB$ (even if it is correct); this is because $SYMTAB$ is a mapping from specifications to specifications, not a specification itself. To write a *specification of a parameterised program* (a *functor signature* in Extended ML), we use Π-abstraction:

$$SYMTAB^{\#} =_{\mathrm{def}} \Pi X : KEY \,.\, \ldots$$

The denotation of $SYMTAB^{\#}$ is a collection of functions from algebras to algebras, which might include the function denoted by $SymTab$. Some parameterised specifications may be viewed as specifications of corresponding parameterised programs, but in general we need both; see [15] for further details, including an explanation of a meaning for '#' (which we use here simply as a part of the names for Π-abstractions).

ASL+ is a higher-order framework, so we may freely write higher-order specifications parameterised on $SYMTAB$, as well as specifications parameterised on KEY or $SYMTAB^{\#}$, and also higher-order parameterised programs. We now highlight some aspects connected with subtyping systems which will be important later.

Subsumption and Refinement. For reference sake, we recall the characteristic rule of subtyping systems, known as *subsumption*:

$$\frac{M : A \qquad A \leq B}{M : B} \tag{SUB}$$

It expresses the idea that if A is a subtype of B, then every term of type A also has type B. In our setting, this rule is sound because of the definition of specification refinement.

Refinement of Π-specifications. There are two ways to refine specifications like $SYMTAB^{\#}$. First, we can make design decisions by imposing further constraints on the body. Alternatively, we can *remove* constraints from the argument specification KEY. Both result in a smaller class of models. A simple example is to imagine that KEY includes axioms that specify an ordering on keys. But we may wish to use an implementation *SymFinMap* of finite maps from some type *ELT* to symbols, which is surely good enough for the job, and satisfies a more general specification:

$$\mathit{SYMFINMAP}^{\#} =_{\mathrm{def}} \Pi X : \mathit{ELT} \ldots.$$

If $\mathit{KEY} \leq \mathit{ELT}$, we have that $\mathit{SYMFINMAP}^{\#} \leq \mathit{SYMTAB}^{\#}$ and so via (SUB), $\mathit{SymFinMap} : \mathit{SYMTAB}^{\#}$ too (modulo some renaming, perhaps). The principle is captured by this rule (where Γ is a list of assumptions about variables):

$$\frac{\Gamma \vdash A' \leq A \qquad \Gamma, X : A' \vdash B \leq B'}{\Gamma \vdash \Pi X{:}A.B \leq \Pi X{:}A'.B'} \qquad (\textsc{sub-}\Pi)$$

This demonstrates the contravariance of the domain of function spaces. We might call this the rule of *contravariant refinement*, as its counterpart is known colloquially as the "contravariant rule" amongst subtyping enthusiasts.

Equality of models of Π-specifications. We would like to have a simple notion of equality on the terms of our calculus which entitles us to substitute like for like. Accepting (SUB-Π), we can go further than the usual β-conversion equalities. Suppose that we have two implementations *SymTab* and *SymFinMap*, and moreover, that these are equal as functions when restricted to the domain *KEY*. Then they are indistinguishable when used as implementations of $\mathit{SYMTAB}^{\#}$, and we write $\mathit{SymTab} = \mathit{SymFinMap} : \mathit{SYMTAB}^{\#}$ to mean that we may replace one for the other in any context which expects an implementation of $\mathit{SYMTAB}^{\#}$. The rule for equality of λ-terms is:

$$\frac{\Gamma \vdash A' \leq A \qquad \Gamma, X : A' \vdash M = M' : B'}{\Gamma \vdash \lambda X{:}A.M = \lambda X{:}A'.M' \ : \Pi X{:}A'.B'} \qquad (\textsc{eq-}\lambda)$$

When terms have several types, it is possible to have different equality relations at different types. The rule (EQ-λ) demonstrates that viewing terms in larger types can allow more terms to be considered equal; informally this is because there are fewer ways of distinguishing them. In general equalities become finer as the subtype hierarchy is descended. Conversely, moving from type to supertype equality is preserved; this monotonicity property that can be derived via (SUB) and the rules for β-conversion. These aspects of a *stratified equality* (stratified over a subtyping relation) are well known in subtyping type-systems.

2 The Formalism ASL+

ASL+ is based on an arbitrary institution [12], together with an associated programming language and specification language. This means we are given a collection of signatures $\Sigma \in \mathbf{Sign}$, and for each signature a collection of algebras $\mathbf{Alg}(\Sigma)$. The denotation of a program is an element of $\mathbf{Alg}(\Sigma)$ for some Σ, and the denotation of a specification is a collection of algebras, an element of $\mathbf{Spec}(\Sigma)$, writing $\mathbf{Spec}(\Sigma)$ as a shorthand for $Pow(\mathbf{Alg}(\Sigma))$. The underlying programming language is formalized as a collection of operations for putting together programs. An *arity* is an element of $\mathbf{Sign}^* \times \mathbf{Sign}$. A *program building operator* (PBO) p with arity $(\Sigma_1 \cdots \Sigma_k, \Sigma)$ is a mapping $\mathbf{Alg}(\Sigma_1) \times \cdots \times \mathbf{Alg}(\Sigma_k) \to \mathbf{Alg}(\Sigma)$. Similarly, a *specification building operator* (SBO) s with the same arity is a mapping $\mathbf{Spec}(\Sigma_1) \times \cdots \times \mathbf{Spec}(\Sigma_k) \to \mathbf{Spec}(\Sigma)$.

The set $\mathcal{T}$ of *terms* of ASL+ is the least set such that:

1. $X \in \mathcal{T}$;
2. If $M_1 \in \mathcal{T} \ldots M_k \in \mathcal{T}$ then $p[M_1 \cdots M_k] \in \mathcal{T}$;
3. If $A_1 \in \mathcal{T} \ldots A_k \in \mathcal{T}$ then $s[A_1 \cdots A_k] \in \mathcal{T}$;
4. If $A \in \mathcal{T}$ and $M \in \mathcal{T}$ then $\lambda X{:}A.M \in \mathcal{T}$ and $\{M\}_A \in \mathcal{T}$;
5. If $M \in \mathcal{T}$ and $N \in \mathcal{T}$ then $MN \in \mathcal{T}$;
6. If $A \in \mathcal{T}$ and $B \in \mathcal{T}$ then $\Pi X{:}A.B \in \mathcal{T}$, $A \;\&\; B \in \mathcal{T}$, and $Spec(A) \in \mathcal{T}$.

where X ranges over a fixed set of variables. The meta-variables A, B, M, N and variants range over $\mathcal{T}$; the intuition is that A and B stand for terms which are specifications (also called types because of the parallels drawn in Section 1). We write $\lambda X \leq A.\, M$ as shorthand for $\lambda X{:}\, Spec(A).M$, and omit the empty bracket pairs from terms like $s[\,]$ and $p[\,]$.

A couple of term constructors were not mentioned in the previous section. $A \;\&\; B$ is the *conjunction* of the specifications A and B: this can be thought of taking the union of the requirements of A with those of B, which corresponds to the intersection of model classes in the semantics; for this reason $A \;\&\; B$ behaves as an *intersection type*. The *singleton type* $\{M\}_A$ provides a way of linking the levels of programs and specifications which is useful for writing Π-specifications. When $\{M\}_A \leq B$ we know that M is an implementation of B. We treat the singleton type as the equivalence class of terms equal to M at the type A; the tag A is needed to keep track of which type we wish to think of M as inhabiting.

A *context* Γ is a list of declarations $X_1 : A_1, \ldots, X_n : A_n$ in which no variable X_i is declared more than once. We write $\langle\rangle$ for the empty context, $Dom(\Gamma)$ for the set $\{X_i \mid 1 \leq i \leq n\}$, and define the projection $\Gamma(X_i)$ to be A_i. A context Γ' *extends* another context Γ, written $\Gamma \subseteq \Gamma'$, if every declaration $X_i : A_i$ in Γ also appears in Γ'. The free and bound variables of a term are defined as expected. Syntactical identity is modulo alpha-convertibility, and $M[X := N]$ denotes the capture-avoiding substitution of the term N for the variable X inside the term M. Substitution extends to contexts pointwise on the A_i. There are three varieties of reduction on terms: β-reduction and η-reduction are as usual; additionally we use Γ-reduction, defined relative to a context Γ, as $\mathbf{C}[X] \longrightarrow_{\Gamma} \mathbf{C}[M]$ provided, for some M and A, $\Gamma(X) \equiv \{M\}_A$ ($\mathbf{C}[-]$ indicates a term with a hole in it). The idea behind Γ-reduction is that if we know a variable X inhabits a type $\{M\}_A$, then X must be equal to M.

Example 2.1 ASL+ is intended to be instantiated with the institution independent SBOs of ASL described in [16]. These include families of operators such as the simple specification constructor $\langle\Sigma, \Phi\rangle$ which has arity (ϵ, Σ) where Φ is a set of Σ-sentences, and the hiding and renaming operator $D_\sigma \equiv$ **derive from** $-$ **by** σ which has arity (Σ', Σ) for each signature morphism $\sigma : \Sigma \to \Sigma'$. If we use the institution of equational logic $\mathcal{EQ}$, then signatures and algebras are the usual algebraic notions and sentences are equations; an orthogonal term rewriting system [9] is a suitable notion of program. PBOs might include $R_{\vec{\Phi}}$ with arity (ϵ, Σ) for a set of (oriented) Σ-equations $\vec{\Phi}$ that define an orthogonal term rewriting system, and $+_{\Sigma_1,\Sigma_2}$ with arity $(\Sigma_1\,\Sigma_2, \Sigma_1 \uplus \Sigma_2)$, for creating disjoint unions of rewrite systems.

2.1 Kind checking

The set $\mathcal{T}$ contains many nonsensical terms. Some of them are eliminated by *kind checking*, a syntax examination process that ensures structural well-formedness (*kindability*) of terms and contexts, by attaching a *kind* to each term according to the rules in Table 1. Kinds play a dual role, in fact; they are also used to define the semantic domains over which terms are interpreted. The kinds, ranged over by K, are formal expressions generated by the grammar:

$$K \ ::= \ \Sigma \quad | \quad K \to K \quad | \quad Spec(K)$$

where Σ ranges over **Sign**. Intuitively, a base kind Σ classifies terms that denote algebras with signature Σ; the arrow kind $K \to K'$ classifies mappings from terms with kind K to terms with kind K', and the specification kind

$Spec(K)$ classifies collections of terms of kind K. Kind-checking involves two judgements:

Γ K-context	Γ is a kindable context.
$\Gamma \vdash M : K$	In context Γ, M has kind K.

In the case that the second judgement holds, we say that M is *kindable* in Γ.

We briefly describe the system. The rules (K-EMPTY) and (K-EXTEND) manage contexts. Each declaration in a context must have a *Spec*-kind, so $X : A$ means that X has kind K when A has kind $Spec(K)$. The rule (K-VAR) derives this. The rules for PBOs (K-PBO) and SBOs (K-SBO) are similar: in each case, we must check that the operator is correctly instantiated and the final premise ensures that Γ is a kindable context in case $k = 0$. The rule (K-λ) gives an arrow kind to λ-abstractions; it should be compared with (K-Π) which shows that Π-abstractions stand for collections of mappings. The rule (K-APP) is the obvious rule for function application. A singleton term $\{M\}_A$ has kind $Spec(K)$ by (K-$\{\}$) when $M : K$ and $A : Spec(K)$, so A stands for a collection of terms with the same kind as M. By (K-&), we can only form conjunctions of specifications that have the same *Spec* kind. Finally, (K-*Spec*) characterises the *Spec* term constructor as forming collections of collections.

Example 2.2 Suppose (for simplicity) that *KEY* is a nullary SBO s_k, with arity (ϵ, Σ_K). Let s_s be an SBO that builds symbol table specifications and p_s a PBO that builds symbol table implementations, with $Arity(s_s) = Arity(p_s) = (\Sigma_K, \Sigma_S)$. Now we have the following kindings:

$\langle\rangle \vdash SYMTAB : Spec(\Sigma_K) \to Spec(\Sigma_S)$	$SYMTAB \equiv \lambda X \leq s_k.\, s_s[X]$
$\langle\rangle \vdash SymTab : \Sigma_K \to \Sigma_S$	$SymTab \equiv \lambda X{:}s_k.p_s[X]$
$\langle\rangle \vdash SYMTAB^\# : Spec(\Sigma_K \to \Sigma_S)$	$SYMTAB^\# \equiv \Pi X{:}s_k.s_s[\{X\}_{s_k}]$

Properties of kind-checking. The kind-checking system is well-behaved. The rules are *syntax-directed* by the term M and context Γ and so the kind of M is unique. The proposition below states desirable properties of the kind-checking system; the proofs are routine inductions on kinding derivations.

Proposition 2.3 (Kind-checking)

1. *Suppose Γ K-context, where $\Gamma \equiv X_1 : A_1, \ldots, X_n : A_n$. Then for all m such that $0 \leq m < n$, $X_1 : A_1, \ldots, X_m : A_m$ is also a kindable context and for some K_{m+1}, we have $X_1 : A_1, \ldots, X_m : A_m \vdash A_{m+1} : Spec(K_{m+1})$.*

$$\frac{}{\langle\rangle \ \text{K-context}} \qquad (\text{K-EMPTY})$$

$$\frac{\Gamma \vdash A : \mathit{Spec}(K) \qquad X \notin \mathit{Dom}(\Gamma)}{\Gamma, X : A \ \text{K-context}} \qquad (\text{K-EXTEND})$$

$$\frac{\Gamma \ \text{K-context} \qquad \Gamma \equiv \Gamma_1, X : A, \Gamma_2 \qquad \Gamma_1 \vdash A : \mathit{Spec}(K)}{\Gamma \vdash X : K} \qquad (\text{K-VAR})$$

$$\frac{\mathit{Arity}(p) = (\Sigma_1 \cdots \Sigma_k, \Sigma) \qquad \text{for } 1 \leq i \leq k,\ \Gamma \vdash M_i : \Sigma_i \qquad \Gamma \ \text{K-context}}{\Gamma \vdash p[M_1 \cdots M_k] : \Sigma} \qquad (\text{K-PBO})$$

$$\frac{\mathit{Arity}(s) = (\Sigma_1 \cdots \Sigma_k, \Sigma) \qquad \text{for } 1 \leq i \leq k,\ \Gamma \vdash A_i : \mathit{Spec}(\Sigma_i) \qquad \Gamma \ \text{K-context}}{\Gamma \vdash s[A_1 \cdots A_k] : \mathit{Spec}(\Sigma)} \qquad (\text{K-SBO})$$

$$\frac{\Gamma \vdash A : \mathit{Spec}(K) \qquad \Gamma, X : A \vdash M : K'}{\Gamma \vdash \lambda X{:}A.M \ : K \to K'} \qquad (\text{K-}\lambda)$$

$$\frac{\Gamma \vdash M : K \to K' \qquad \Gamma \vdash N : K}{\Gamma \vdash M\,N : K'} \qquad (\text{K-APP})$$

$$\frac{\Gamma \vdash M : K \qquad \Gamma \vdash A : \mathit{Spec}(K)}{\Gamma \vdash \{M\}_A : \mathit{Spec}(K)} \qquad (\text{K-}\{\})$$

$$\frac{\Gamma \vdash A : \mathit{Spec}(K) \qquad \Gamma \vdash B : \mathit{Spec}(K)}{\Gamma \vdash A \mathbin{\&} B : \mathit{Spec}(K)} \qquad (\text{K-}\&)$$

$$\frac{\Gamma \vdash A : \mathit{Spec}(K) \qquad \Gamma, X : A \vdash B : \mathit{Spec}(K')}{\Gamma \vdash \Pi X{:}A.B : \mathit{Spec}(K \to K')} \qquad (\text{K-}\Pi)$$

$$\frac{\Gamma \vdash A : \mathit{Spec}(K)}{\Gamma \vdash \mathit{Spec}(A) : \mathit{Spec}(\mathit{Spec}(K))} \qquad (\text{K-}\mathit{Spec})$$

Table 1: Kind checking in ASL+

2. *Suppose* $\Gamma \vdash M : K$. *Then (a)* Γ *K-context, and (b)* $FV(()M) \subseteq Dom(\Gamma)$.

3. Weakening. *If* $\Gamma \vdash M : K$, $\Gamma \subseteq \Gamma'$ *and* Γ' *K-context, then* $\Gamma' \vdash M : K$.

4. Kind unicity. *If* $\Gamma \vdash M : K$, *then* K *is the unique such kind.*

5. Decidability. *It is decidable whether* $\Gamma \vdash M : K$ *for any given* Γ, M, *and* K*; furthermore, given just* Γ *and* M, *it is decidable whether there is such a* K.

6. Substitution. *If* $\Gamma, X : A, \Gamma' \vdash M : K$, $\Gamma \vdash A : Spec(K')$ *and* $\Gamma \vdash N : K'$, *then* $\Gamma, \Gamma'[X := N] \vdash M[X := N] : K$.

7. Single-step subject reduction. *Suppose* $\Gamma \vdash M : K$. *If any of* $M \longrightarrow_\beta M'$, $M \longrightarrow_\eta M'$ *or* $M \longrightarrow_\Gamma M'$, *then* $\Gamma \vdash M' : K$ *too.*

It is important to realise that kind checking is a necessary, but *not sufficient* condition to ensure that a term has a well-defined meaning. There are several ways that a kindable term may fail to have a denotation in the semantics given later:

- We can write λ-abstractions over specifications. In general, we have to check that a program satisfies a specification to see that an application of such a term is defined. For example, if s_0 is an inconsistent Σ-specification, and p denotes a Σ-algebra, so $Arity(s_0) = Arity(p) = \Sigma$, then $\vdash (\lambda X{:}s_0.X)\ p : \Sigma$ holds in the kind-checking system, but we choose to regard $(\lambda X{:}s_0.X)\ p$ as meaningless since p cannot satisfy s_0.

- A singleton specification $\{M\}_A$ denotes the collection of algebras equal to M which satisfy the specification A. Again, for this to be defined, we must check that M satisfies A, which is a stronger condition than the rule (κ-$\{\}$) ensures.

- We allow for partiality of PBOs and SBOs so that some instantiations of an operator may fail to denote an algebra or collection of algebras. Whilst for specifications we might instead suppose that the operator returned the inconsistent specification, it is necessary for PBOs. For example, if we wished to allow a general union operation $\cup$ of term rewriting systems in $\mathcal{EQ}$, it may be that $P_1 \cup P_2$ is not confluent (hence not orthogonal); so $\cup$ is at best a partial operator.

3 Semantics of ASL+

The semantics presented here is an altered form of the set-theoretic semantics in [15]. Along lines suggested there, we use the institution to construct a hierarchy of set-like domains $\mathcal{D}_K$, indexed by kinds K. Programs built using PBOs have kind Σ for some specification Σ, and are interpreted as elements of $\mathcal{D}_K = \mathbf{Alg}(\Sigma)$. Rather than sets, specifications with kind $\mathit{Spec}(K)$ for some K are interpreted as partial equivalence relations (PERs) over $\mathcal{D}_K$ (for specifications built using SBOs, we have PERs over $\mathbf{Alg}(\Sigma)$ for some Σ). Parameterised terms with kind $K \to K'$ are interpreted as set-theoretic partial functions in $\mathcal{D}_K \rightharpoonup \mathcal{D}_{K'}$. We pause to explain these choices before presenting the definitions.

Partial equivalences. The use of partial equivalence relation models for interpreting subtyping and polymorphic type-systems is well-known (*e.g.,* [6]). Recall that a *partial equivalence relation* R is a set $|R|$ (the carrier) together with a symmetric and transitive relation $-R- \subseteq |R| \times |R|$. The reason why we use PERs is that they model the typed equational theory of λ-calculi, and in a simple way that works well with subtyping. A type is interpreted as a PER over a set of values which are denotations of terms. Subtyping is just inclusion of PERs; if $A \leq B$ then for all a, b, $a \ [\![A]\!] \ b \Longrightarrow a \ [\![B]\!] \ b$ should hold. It follows that if two terms are equal at A, they are also equal at B. Typically, PER models follow an "untyped" approach, based on a global value space (*e.g.,* the domain D of a model of the untyped λ-calculus) from which all types are carved out. Here we use the hierarchy $\mathcal{D}_K$, retaining a "typed" aspect; this is because algebras over different signatures live in different categories, so it seems natural to preserve the different value spaces in the semantics.

Partial functions. Partial functions are used because, as explained above, the true domain of a function with kind $K \to K'$ may be smaller than K. Their use also allows a neat way of capturing the rule (SUB-Π) shown in Section 1.1, via inclusion. If $\Pi X{:}A.B$ has kind $K \to K'$, then we let $[\![\Pi X{:}A.B]\!]$ stand for the collection of partial functions in $\mathcal{D}_{K \to K'}$ which are total *at least* on $[\![A]\!]$, and map elements $a \in [\![A]\!]$ to elements of $[\![B]\!]_{[X \to a]}$. So the inclusion $[\![\Pi X{:}A.B]\!] \subseteq [\![\Pi X{:}A'.B']\!]$ holds provided $[\![A']\!] \subseteq [\![A]\!]$ and $[\![B]\!]_{[X \to a]} \subseteq [\![B']\!]_{[X \to a]}$ for all $a \in [\![A']\!]$. Notice that this does not work if we define $[\![\Pi X{:}A.B]\!]$ as a set of total functions defined on exactly $[\![A]\!]$.

3.1 The semantic interpretation

We recall some nomenclature: the domain of a PER is $dom(R) = \{a \mid a\ R\ a\}$ and $a \in R$ abbreviates $a \in dom(R)$. If $a \in R$ then the equivalence class of a in R is $[a]_R = \{b \mid a\ R\ b\}$. Inclusion of PERs $R \subseteq R'$ is defined for PERs with the same carrier by $a\ R\ b \Longrightarrow a\ R'\ b$ for all a, b. PER$[A]$ stands for the set of all PERs with carrier A.

Domains for kinds and contexts. The semantic domain $\mathcal{D}_K$ associated to each kind K is a set defined by induction on the structure of K:

$$\begin{array}{lcl}
\mathcal{D}_\Sigma & = & \mathbf{Alg}(\Sigma) \\
\mathcal{D}_{K \to K'} & = & \mathcal{D}_K \rightharpoonup \mathcal{D}_{K'} \\
\mathcal{D}_{Spec(K)} & = & \mathrm{PER}[\mathcal{D}_K]
\end{array}$$

where $A \rightharpoonup B$ is the set of partial functions from A to B. We assume that for each $\Sigma \in \mathbf{Sign}$, the collection of algebras over Σ is built within some fixed universe, and so $\mathbf{Alg}(\Sigma)$ is a set; this provides a simplistic justification of the existence of the hierarchy $\mathcal{D}_K$.

Contexts are interpreted by PERs over products of sets: $A \times B$ is the set of all ordered pairs $\langle a, b\rangle$ with $a \in A, b \in B$ and π_1, π_2 are the projections. Let $1 =_{\mathrm{def}} \{*\}$ be a single point set. The domain $\mathcal{D}_\Gamma$ for a kindable context Γ is defined by induction on a kinding derivation:

$$\begin{array}{lcl}
\mathcal{D}_{\langle\rangle} & = & 1 \\
\mathcal{D}_{\Gamma, X:A} & = & \mathcal{D}_\Gamma \times \mathcal{D}_K, \text{ where } \Gamma \vdash A : Spec(K)
\end{array}$$

The free variables in a term will be interpreted by projection from an *environment*, which is a tuple g in the domain of some context. The projections along g corresponding to a context Γ are given by:

$$\begin{array}{rcl}
g^{\langle\rangle}(Y) & & \text{undefined, for all } Y. \\
g^{\Gamma, X:A}(Y) & = & \begin{cases} \pi_2(g), & \text{if } Y \equiv X, \\ (\pi_1(g))^\Gamma(Y) & \text{if } Y \not\equiv X. \end{cases}
\end{array}$$

We write ${g_1}^{\Gamma_1} \subseteq {g_2}^{\Gamma_2}$ if ${g_1}^{\Gamma_1}(X) = {g_2}^{\Gamma_2}(X)$ for all $X \in Dom(\Gamma_1)$.

Semantic constructions. Here are the semantic constructions used in the interpretation. The interpretation is partial (some terms fail to denote even if they are kindable, as we have explained), so we adopt the convention that the application of any of the constructions below to an undefined argument gives an undefined result.

Let A and B be sets; R, R' and R'' elements of PER$[A]$, $F \in dom(R) \to B$ and $G \in dom(R) \to$ PER$[B]$ such that $a \ R \ b$ implies $G(a) = G(b)$. Then:

- $Id(A, A') \ \in$ PER$[A]$ provided $A' \subseteq A$ and where:

  $$a \ Id(A, A') \ a \qquad \text{for all } a \in A'$$

- $\{\}(R, a) \ \in$ PER$[A]$ provided $a \in A$, and where:

  $$b \ \{\}(R, a) \ c \qquad \text{iff} \qquad b \ R \ c \quad \text{and} \quad b \ R \ a$$

- $\lambda(R, F) \ \in A \rightharpoonup B$ is the partial function f defined by:

  $$f(a) = \begin{cases} F(a) & \text{for all } a \in R \\ \text{undefined} & \text{otherwise.} \end{cases}$$

- $\Pi(R, G) \ \in$ PER$[A \rightharpoonup B]$ where:

  $$f \ \Pi(R, G) \ g \qquad \text{iff} \qquad \forall a, b. a \ R \ b \Longrightarrow f(a) \ G(a) \ g(b)$$

- $\Sigma(R, G) \ \in$ PER$[A \times B]$ where:

  $$c \ \Sigma(R, G) \ d \quad \text{iff} \quad \pi_1(c) \ R \ \pi_1(d) \quad \text{and} \quad \pi_2(c) \ G(\pi_1(c)) \ \pi_2(d)$$

- $\cap(R, R') \ \in$ PER$[A]$ where:

  $$a \cap(R, R') \ b \qquad \text{iff} \qquad a \ R \ b \quad \text{and} \quad a \ R' \ b$$

- $\mathcal{P}(R) \ \in$ PER[PER$[A]$] where:

  $$R' \ \mathcal{P}(R) \ R'' \qquad \text{iff} \qquad R' = R'' \quad \text{and} \quad R' \subseteq R$$

Constructions such as λ, Π, Σ and $\cap$ are well-known. For example, $\Pi(R, G)$ is a PER in which two partial functions are related if they map related arguments to related results (and both must be defined on at least $dom(R)$). Less familiar ones include $Id(A, B)$, which is simply a PER over elements of A with domain B on which the equivalence relation is just identity; the PER $\{\}(R, a)$ which is the equivalence class of a in R, and $\mathcal{P}(R)$ in which two PERs are equivalent if they are equal as PERs and included in R. It is a simple matter to check that the constructions lead to values in the domains claimed, given defined arguments in the domains stated.

$$
\begin{array}{lcl}
[\![\langle\rangle]\!] & = & Id(1,1) \\
[\![\Gamma,\, X : A]\!] & = & \Sigma([\![\Gamma]\!],\, [\![\Gamma \,\vdash\, A : Spec(K)]\!]) \\
[\![\Gamma \,\vdash\, X : K]\!]_g & = & g^{\Gamma}(X) \\
[\![p[M_1 \cdots M_k] : \Sigma]\!] & = & [\![p]\!]\,([\![M_1 : \Sigma_1]\!],\ \ldots,\ [\![M_k : \Sigma_k]\!]) \\
[\![s[A_1 \cdots A_k] : Spec(\Sigma)]\!] & = & Id(\,\mathcal{D}_\Sigma,\, [\![s]\!]\ (dom([\![A_1 : Spec(\Sigma_1)]\!]), \\
 & & \qquad\qquad \ldots,\, dom([\![A_k : Spec(\Sigma_k)]\!])\,)\,) \\
[\![\{M\}_A : Spec(K)]\!] & = & \{\}([\![A : Spec(K)]\!],\ [\![M : K]\!]) \\
[\![A \ \& \ B : Spec(K)]\!] & = & \cap([\![A : Spec(K)]\!],\ [\![B : Spec(K)]\!]) \\
[\![Spec(A) : Spec(Spec(K))]\!] & = & \mathcal{P}([\![A : Spec(K)]\!]) \\
[\![M\ N : K']\!] & = & [\![M : K \to K']\!]\ \ ([\![N : K]\!])
\end{array}
$$

$$
\begin{array}{l}
[\![\Gamma \,\vdash\, \lambda X{:}A.M \ \ : K \to K']\!]_g \\
\qquad = \lambda([\![\Gamma \,\vdash\, A : Spec(K)]\!]_g,\ a \mapsto [\![\Gamma,\, X : A \,\vdash\, M : K']\!]_{\langle g,\, a\rangle})
\end{array}
$$

$$
\begin{array}{l}
[\![\Gamma \,\vdash\, \Pi X{:}A.B \ \ : Spec(K \to K')]\!]_g \\
\qquad = \Pi([\![\Gamma \,\vdash\, A : Spec(K)]\!]_g,\ a \mapsto [\![\Gamma,\, X : A \,\vdash\, B : Spec(K')]\!]_{\langle g,\, a\rangle})
\end{array}
$$

Table 2: Semantics of ASL+

Definition of semantics. The interpretation of contexts and terms is defined in Table 2 by induction on kind-checking derivations. The interpretation of a context $[\![\Gamma]\!]$ is a PER over $\mathcal{D}_\Gamma$, and the interpretation of a term, $[\![\Gamma \vdash M : K]\!]$, is a map from $dom([\![\Gamma]\!])$ to $\mathcal{D}_K$. If $g \notin dom([\![\Gamma]\!])$, then $[\![\Gamma \vdash M : K]\!]_g$ is undefined by convention. There is an equation in the table for each kinding rule; for readability Γ and g are omitted when they don't play a part. We shall sometimes also omit K when it is clear from the context, writing just $[\![M]\!]$.

We rely upon an externally defined semantics for PBOs and SBOs: as mentioned before, $[\![p]\!]$ and $[\![s]\!]$ are partial maps between algebras or classes of algebras. SBOs ignore the equivalence relation on their arguments, and the instantiation of an SBO is interpreted as an identity PER. This means that two terms with kind Σ are equal just when they denote exactly the same algebra. (It would be interesting to extend this to allow coarser institution-dependent equalities, such as isomorphism, or even to allow $[\![s]\!]$ to be a PER constructor).

Example 3.1 We calculate the meaning of the closed term $SYMTAB^{\#}$:

$$\begin{array}{rcl}
R =_{\mathrm{def}} [\![s_k[\,]]\!] & = & Id(\mathcal{D}_{\Sigma_K}, [\![s_k]\!]) \\
[\![SYMTAB^{\#}]\!] & = & [\![\Pi X{:}s_k.s_s[\{X\}_{s_k}]]\!] \\
 & = & \Pi(R, a \mapsto Id(\mathcal{D}_{\Sigma_S}, [\![s_s]\!](dom(\{\}(R, [\![X]\!]_{\langle\star,\, a\rangle}))))) \\
 & = & \Pi(R, a \mapsto Id(\mathcal{D}_{\Sigma_S}, [\![s_s]\!](\{a\})))
\end{array}$$

By the definition of Π, $[\![SYMTAB^{\#}]\!]$ is interpreted as the set of partial functions f from $\mathbf{Alg}(\Sigma_K)$ to $\mathbf{Alg}(\Sigma_S)$ such that $\forall a \in [\![s_k]\!].\, f(a) \in [\![s_s]\!](\{a\})$. Two such functions f and f' are equivalent if they are equal as functions when restricted to $[\![s_k]\!]$.

Properties of the semantics. It would be nice to establish that the interpretation is well-defined in the sense that whenever $[\![M : K]\!]$ is defined, then $[\![M]\!] \in \mathcal{D}_K$. Unfortunately, this isn't true! The reason is that, because of the use of partial functions, a function application $f(a)$ in the semantics may be defined "by accident" without a being in the domain of the PER that f is assumed to inhabit. This isn't ensured by kind-checking or definedness of the interpretation function, but it is ensured by the stronger formal system of the next section. Some useful properties of the semantics can be established at this stage, nevertheless, under the assumption that everything we deal with is defined and in the domains required.

Lemma 3.2 (Semantic properties for well-defined interpretations)

1. Weakening. *Let* Γ, Γ' *be kindable contexts with* $\Gamma \subseteq \Gamma'$. *Let* $g \in [\![\Gamma]\!]$ *and* $g' \in [\![\Gamma']\!]$ *where* $g^{\Gamma} \subseteq g'^{\Gamma'}$. *Then* $[\![\Gamma \vdash M : K]\!]_g = [\![\Gamma' \vdash M : K]\!]_{g'}$.

2. Semantic contexts. *Let* Γ *be a kindable context and* $\Gamma \vdash X : K$.
Then: $\forall g_1, g_2.\ g_1 \quad [\![\Gamma]\!] \quad g_2 \implies {g_1}^{\Gamma}(X) \quad [\![\Gamma \vdash \Gamma(X) : \mathit{Spec}(K)]\!]_{g_1} \quad {g_2}^{\Gamma}(X)$

3. Semantic substitution. *Let* $\Gamma, X : A, \Gamma' \vdash M : K'$, $\Gamma \vdash A : \mathit{Spec}(K)$, $g \in [\![\Gamma]\!]$, *and let* $n = [\![\Gamma \vdash N : K]\!]_g$. *If* $g_1 \in [\![\Gamma, \Gamma'[X := N]]\!]$, *and* $g_2 \in [\![\Gamma, X : A, \Gamma']\!]$, *then:*
$$[\![\Gamma, \Gamma'[X := N] \vdash M[X := N] : K']\!]_{g_1} = [\![\Gamma, X : A, \Gamma' \vdash M : K']\!]_{g_2}$$
provided $g^{\Gamma} \subseteq {g_1}^{\Gamma, \Gamma'[X:=N]} \subseteq {g_2}^{\Gamma, X:SP, \Gamma'}$ and ${g_2}^{\Gamma, X:SP, \Gamma'}(X) = n$.

4. β and η-reduction. *Suppose* $\Gamma \vdash M : K$ *and* $g \in [\![\Gamma]\!]$. *If either* $M \longrightarrow_{\beta} N$ *or* $M \longrightarrow_{\eta} N$, *then* $[\![M]\!]_g = [\![N]\!]_g$.

4 Satisfaction in ASL+

So far we have a formal syntax of terms, and a semantics for the kindable terms. However, not all kindable terms have a denotation in the semantics, and we have no way of proving when a program satisfies a specification (elementhood in the semantics). We now introduce the system of rules[1] in Table 3 for proving satisfaction. Two judgements are defined:

Γ Context	Γ is a *denoting* context.
$\Gamma \vdash M : A$	In context Γ, M *satisfies* A.

and we use some derived judgements:

$\Gamma \vdash A \leq B$	$=_{\text{def}}\ \Gamma \vdash A : \mathit{Spec}(B)$
$\Gamma \vdash M = N : A$	$=_{\text{def}}\ \Gamma \vdash M : \{N\}_A$
$\Gamma \vdash A$	$=_{\text{def}}\ \Gamma \vdash A \leq B$ for some B

We have explained the "subtyping" abbreviation already. A relation of typed-equality $\Gamma \vdash M = N : A$ is included in the calculus via singleton types; this has been investigated elsewhere [1].

The judgement $\Gamma \vdash A$ asserts that A is a *denoting* type. More generally, a denoting term M is one for which there exists an A such that $\Gamma \vdash M : A$. Thus the rules in Table 3 can also be used to establish that a term has a denotation in the semantics. From the viewpoint of ASL+, a program term

[1] These are a re-design of those in [15], which had several omissions.

is meaningful just in case there is some specification that it satisfies, and similarly a specification is meaningful if there is another which it refines. This may seem like an odd way of determining which terms are well-formed, but typically the SBOs include operators for constructing trivial specifications, which for example may amount to some simple type-checking requirements. So if M is built from PBOs, it is natural to expect there to be some A built from SBOs such that $M : A$. Furthermore, we would expect $A \leq A$ if A is a well-formed specification. We examine this more closely next.

Basic programs and specifications. To do anything useful, we must first have some way to prove things about the underlying programming language and specification language. A *basic program* P is a term built using only PBOs and *basic program variables* from a **Sign**-sorted set V_{prog}; similarly a *basic specification* S (or T) is built using only SBOs and variables from a **Sign**-sorted set V_{spec}. We abstract away from the details of the underlying proof systems and assume that there are two *consequence relations* [3] which describe the properties of interest:

$$\begin{array}{ll} \Delta \models\!\!\!-_{V}^{sat} P : S & \textit{basic satisfaction;} \\ \Delta \models\!\!\!-_{V}^{ref} S \leq T & \textit{basic refinement.} \end{array}$$

In each case, V is the set of variables (from V_{prog} and V_{spec}) free in the program or specification terms and Γ is a finite set of statements with the same form as the right hand side. Variables are needed because it is essential to have *schematic* assertions that allow us to deal with SBOs and PBOs applied to ASL+ terms, and in particular, ASL+ variables. The rules (PBO) and (SBO) formalize this.[2] To apply one of these rules (backwards), one must find appropriate substitutions θ from variables in V to ASL+ terms which unify the right hand side of a sequent with the desired conclusion in ASL+. If the left hand side of the sequent Δ is non-empty, this leads to further statements to be proved in ASL+. (Of course, the substitutions θ must respect the sorts of the variables; we omit details here).

Example 4.1 In $\mathcal{EQ}$, we would expect the consequence relations to include, for sets of sentences Φ_1, Φ_2 over signatures Σ_1, Σ_2: $\models\!\!\!-_{\{\}}^{sat} R_{\vec{\Phi_1}} + R_{\vec{\Phi_2}} : \langle \Sigma_1 \uplus \Sigma_2, \Phi_1 \cup \Phi_2, \rangle$ (the union of two rewrite systems satisfies a specification consisting of a union of the equations), and $R_{\sigma(\vec{\Phi})} : \mathcal{S} \models\!\!\!-_{\{\mathcal{S}\}}^{sat} R_{\vec{\Phi}} : D_\sigma(\mathcal{S})$ (which allows renaming), where $\sigma : \Sigma \to \Sigma'$ is a signature morphism, Φ is

[2]The vector notation $\bar{P}$, *etc.* abbreviates vectors of arguments to PBOs and SBOs. We have glossed over some details: there should be extra premises to ensure that the arguments are denotable: $\Gamma \vdash \bar{A}$ *etc.*, and we should ensure that the arity of the operators is respected.

$$\frac{}{\langle\rangle \ \text{Context}} \qquad (\textsc{empty})$$

$$\frac{\Gamma \vdash A \qquad X \notin Dom(\Gamma)}{\Gamma, X : A \ \text{Context}} \qquad (\textsc{extend})$$

$$\frac{\Gamma \ \text{Context}}{\Gamma \vdash X : \Gamma(X)} \qquad (\textsc{var})$$

$$\frac{\Delta \mathrel{\vdash\!\!\!-}^{sat}_{V} p[\bar{P}] : s[\bar{S}] \qquad \theta_p(\bar{P}) = \bar{M},\ \theta_s(\bar{S}) = \bar{A} \qquad \Gamma \vdash \theta_p(\theta_s(\Delta)) \qquad \Gamma \ \text{Context}}{\Gamma \vdash p[\bar{M}] : s[\bar{A}]} \qquad (\textsc{pbo})$$

$$\frac{\Delta \mathrel{\vdash\!\!\!-}^{ref}_{V} s[\bar{S}] \leq t[\bar{T}] \qquad \theta_s(\bar{S}) = \bar{A},\ \theta_s(\bar{T}) = \bar{B} \qquad \Gamma \vdash \theta_s(\Delta) \qquad \Gamma \ \text{Context}}{\Gamma \vdash s[\bar{A}] : \mathit{Spec}(t[\bar{B}])} \qquad (\textsc{sbo})$$

$$\frac{\Gamma, X : A \vdash M : B}{\Gamma \vdash \lambda X{:}A.M : \Pi X{:}A.B} \qquad (\Pi\text{-I})$$

$$\frac{\Gamma \vdash M : \Pi X{:}A.B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B[X := N]} \qquad (\Pi\text{-E})$$

$$\frac{\Gamma \vdash M : A_1, A_2 \qquad \Gamma \vdash A_1, A_2 : \mathit{Spec}(B)}{\Gamma \vdash M : A_1 \ \&\ A_2} \qquad (\&\text{-I})$$

$$\frac{\Gamma \vdash M : A_1 \ \&\ A_2 \qquad j = 1, 2}{\Gamma \vdash M : A_i} \qquad (\&\text{-E})$$

$$\frac{\Gamma, X : A \vdash X : B \qquad \Gamma \vdash B}{\Gamma \vdash A : \mathit{Spec}(B)} \qquad (\mathit{Spec}\text{-I})$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash A : \mathit{Spec}(B)}{\Gamma \vdash M : B} \qquad (\mathit{Spec}\text{-E})$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : A \qquad M =_{\beta\eta\Gamma} N}{\Gamma \vdash M : \{N\}_A} \qquad (\{\}\text{-I})$$

$$\frac{\Gamma \vdash M : \{N\}_A}{\Gamma \vdash M : A} \qquad (\{\}\text{-E})$$

Table 3: Satisfaction Rules of ASL+

a set of Σ-sentences and $\mathcal{S}$ stands for a basic Σ'-specification. Examples of proof rules which generate a consequence relation for $\vdash^{ref}$ on the institution independent operators of ASL can be found in [11].

Some assumptions must be made of $\vdash^{sat}$ and $\vdash^{ref}$, the most important of which is that they are sound with respect to the semantics of basic programs and specifications. For example, whenever $P_1 : S_1 \ldots P_n : S_n \vdash^{sat}_{V} P : S$ then

$$\forall \rho, \gamma.\, [\![P_1]\!]_\rho \in [\![S_1]\!]_\gamma \;\bigwedge\; \cdots \;\bigwedge\; [\![P_n]\!]_\rho \in [\![S_n]\!]_\gamma \Longrightarrow [\![P]\!]_\rho \in [\![S]\!]_\gamma$$

where ρ and γ are evaluations mapping Σ-variables from V_{prog} to Σ-algebras and Σ-variables from V_{spec} to sets of Σ-algebras. There are other requirements which reflect the ideas above: there should always be an S such that $\vdash^{sat}_{\{\}} P : S$ whenever P is a well-formed basic program with a denotation, and in this case we should have $\vdash^{ref}_{\{\}} S \leq S$ to indicate that S too is well-formed and has a denotation.

Higher-order constructors. In Table 3 there is a rule of *introduction* and a rule of *elimination* for each of the higher-order term constructors.

The rules for λ-introduction (Π-I) and elimination (Π-E) are usual in all type systems with dependent Π types. For conjunction the rules are also well-known, except that the introduction rule (& -I)[3] restricts intersections to being formed over types which have a common supertype. This reflects the semantics: we only form collections within the same value space.

The elimination rule (*Spec*-E) for the *Spec* operator is simply the subsumption rule in a different guise. The introduction rule (*Spec*-E) is rather unusual: it has a simple reading (if whenever X is assumed to inhabit A, it can be shown to also inhabit B, then deduce $A \leq B$), but its simplicity belies its power when combined with the other introduction and elimination rules. It is common in subtyping systems to axiomatise a separate subtyping relation with rules for each type constructor; this one rule captures the full relation. In particular, the (derived) relation $\leq$ is easily shown to be reflexive and transitive on denoting types, and familiar rules for subtyping intersection types follow. Also, the rule (SUB-Π) shown in Section 1.1 is admissible when we close the system under a rule of η-subject reduction.

The meaning of the singleton terms is captured by ($\{\}$-I). Whenever M is convertible to N via β,η or Γ reductions, and both satisfy S, we assert that M inhabits the singleton $\{N\}_S$: intuitively, M is in the equivalence class of terms equal to N when considered at S. The elimination rule is weaker, it just says that when $M : \{N\}_A$, then M must also satisfy A.

[3] Notice that it has four premises, abbreviated in a hopefully obvious way.

Example 4.2 As a simple example, we can show that $\mathit{SymTab} : \mathit{SYMTAB}^{\#}$:

$$\dfrac{\mathcal{P} : s_k, \mathcal{P} : \mathcal{S} \models\!\!-^{sat}_{\{\mathcal{S},\mathcal{P}\}} p_s[\mathcal{P}] : s_s[\mathcal{S}] \qquad \dfrac{\vdots}{X : s_k} \qquad \dfrac{\vdots}{X : \{X\}_{s_k}} \qquad \dfrac{\dfrac{\models\!\!-^{ref}_{\{\}}\ s_k \leq s_k}{\vdash\ s_k \leq s_k}}{X : s_k\ \text{Context}}}{\dfrac{X : s_k\ \vdash\ p_s[X] : s_s[\{X\}_{s_k}]}{\langle\rangle\ \vdash\ \lambda X{:}s_k.p_s[X] : \Pi X{:}s_k.s_s[\{X\}_{s_k}]}}$$

(the dots are filled by repeating the few rules on the right, and using the rule ($\{\}$-I) in the second case; we omitted the context $X : s_k$ in these branches too).

This example serves to show how the proof theoretic side of the ASL+ framework is glued together, rather than illustrate the power of the language. We assumed that the main proof arises from a single sequent in $\models\!\!-^{sat}$; in reality s_k and p_s would probably be composed of several SBOs and PBOs, and this proof could involve several invocations of the underlying proof systems.

Properties of satisfaction. Unlike the kind-checking system, it is not obvious how to apply the rules in the satisfaction system: for a given conclusion, several rules may be able to derive it. It would be interesting to examine ways of directing proof search in this system; this corresponds to looking for a type-checking algorithm for the related subtyping system. Of course, theorem proving is almost always needed at the level of basic programs and specifications anyway, so we can't hope for a complete algorithm.

Space precludes any treatment of the meta-theory of the rules here. We mention only the crucial aspect of the system mentioned at the outset: soundness with respect to the semantics defined in Section 3. This also establishes the well-definedness of the interpretation on denotable terms and contexts.

Proposition 4.3 (Soundness and Definedness properties)

1. Agreement with kind-checking.
   If Γ *Context, then* Γ *K-context.*
   If $\Gamma\ \vdash\ M : A$ *then for some (unique) kind* K, $\Gamma\ \vdash\ M : K$ *and* $\Gamma\ \vdash\ A : \mathit{Spec}(K)$.

2. Definedness.
   If Γ *Context, then* $[\![\Gamma]\!]\ \in\ \mathrm{PER}[\mathcal{D}_\Gamma]$.
   If $\Gamma\ \vdash\ M : A$ *and* $g \in [\![\Gamma]\!]$, *then* $[\![M : K]\!]_g\ \in\ \mathcal{D}_K$.

3. Γ-reduction.
   Suppose $\Gamma\ \vdash\ M : A$ *and* $g \in [\![\Gamma]\!]$. *Then* $M \longrightarrow_\Gamma N$ *implies* $[\![M]\!]_g\ [\![A]\!]_g\ [\![N]\!]_g$.

4. Soundness.
If $\Gamma \vdash M : A$ *and* $g\ [\![\Gamma]\!]\ g'$, *then* $[\![M]\!]_g\ [\![A]\!]_g\ [\![M]\!]_{g'}$

The proofs proceed by induction on derivations in the satisfaction system. Part 1 uses Proposition 2.3; parts 2–4 are proved together, using Proposition 3.2. Now we have established:

Theorem 4.4 (Soundness of Satisfaction)
$\Gamma \vdash M : A$ *implies* $[\![\Gamma \vdash M : K]\!]_g \in [\![\Gamma \vdash A : Spec(K)]\!]_g$ *for all* $g \in [\![\Gamma]\!]$.

Proof *A corollary of the final part of Proposition 4.3, taking* $g = g'$. □

5 Conclusions

ASL+ provides a unified framework for programming and specification in-the-large which seems powerful. Some larger and more detailed examples than those shown here have been studied in [17, 15, 2] but work is still needed to appreciate the practical merits of the different mechanisms of parameterisation. Real specification languages built on ASL+ might not use all of the parameterisation possibilities (Extended ML, for example, lacks parameterised specifications), but it makes sense to study a complete kernel formalism to see that the distinct mechanisms can be safely combined.

There are many avenues to be explored starting from the satisfaction system of Section 4. We would like to be able to characterise the relationship between the consequence relations for PBOs and SBOs and their corresponding fragments inside ASL+. The best case would be if the extension is conservative, and complete (*i.e.,* a converse of Theorem 4.4 holds) when the underlying relations are complete. As mentioned, we should try to find ways of directing proof search in the system, so that showing satisfaction in ASL+ is reduced to showing underlying consequences in a canonical way. Unfortunately, similar things can be tricky to show for related subtyping systems [8, 14], and the added complication of dependent Π-types makes things harder still. Also, the concise presentation of the rules shown here is somewhat provisional; although it is nice to describe, it may prove more fruitful to study a longer equivalent version where more of the rules are spelled out.

Certain extensions of the satisfaction system would be interesting. One is to allow subtyping between signatures: whenever $\iota : \Sigma_1 \hookrightarrow \Sigma_2$ is an inclusion in a sub-category of signature inclusions, we can consider $\Sigma_2 \leq \Sigma_1$ by virtue of the reduct functor. This would allow many renamings to be avoided at the level of syntax; in the semantics, we would have to reconstruct them. The subtyping semantics in [4] uses a similar technique.

As well as subtyping parallels, there are some specification languages related to ASL+. Cengarle and Wirsing [7] describe a language with higher-order parameterisation, including parameterisation on signatures. Drossopoulou and Paterson [10] have explored similar type theory connections when designing a language for writing parameterised programs and their interfaces. SPECTRAL in [13] is a design for a rich language that combines an institution-level language that has subtyping, polymorphism and dependent types with a module-level language similarly endowed for specifications and programs.

Acknowledgements. I would like to sincerely thank Don Sannella, Benjamin Pierce and Andrzej Tarlecki for their coaching and encouragement during the course of this work. Thanks are also due to Claudio Russo and Martin Hofmann for discussions about ASL+, and to the anonymous referees of ADT'94 for suggestions to improve the presentation. I was supported by a UK EPSRC postgraduate studentship.

References

[1] David R. Aspinall. Subtyping with singleton types. In *Proc. Computer Science Logic, CSL'94, Kazimierz, Poland*, Lecture Notes in Computer Science 933. Springer-Verlag, 1995.

[2] David R. Aspinall. Type systems for algebraic specification in-the-large. Forthcoming PhD thesis, Department of Computer Science, University of Edinburgh, 1996.

[3] Arnon Avron. Simple consequence relations. *Information and Computation*, 92:105–139, 1991.

[4] Val Breazu-Tannen, Thierry Coquand, Carl A. Gunter, and Andre Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991.

[5] Luca Cardelli. Structural subtyping and the notion of power type. In *Fifteenth Annual ACM Symposium on Principles of Programming Languages*, 1988.

[6] Luca Cardelli and Giuseppe Longo. A semantic basis for Quest. *Journal of Functional Programming*, 1(4):417–458, 1991.

[7] María Victoria Cengarle and Martin Wirsing. A calculus of parameterization for algebraic specifications. Technical Report 94/198, Department of Computer Science, Monsahs University, 1994.

[8] Pierre-Louis Curien and Giorgio Ghelli. Coherence of subsumption, minimum typing and type-checking in $F_{\leq}$. *Mathematical Structures in Computer Science*, 2:55–91, 1992.

[9] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In van Leeuwen et.al., editor, *Handbook of Theoretical Computer Science*, chapter 6. North-Holland, 1990.

[10] Sophia Drossopoulou and Ross Paterson. Higher order module parameterization. Technical report, Department of Computing, Imperial College, October 1993.

[11] Jorge Farrés-Cassals. *Verification in ASL and related Specification Languages*. PhD thesis, Edinburgh University, 1992.

[12] J. A. Goguen and R. M. Burstall. Institutions: abstract model theory for specification and programming. *Journal of the ACM*, 39:95–146, 1992.

[13] Bernd Krieg-Brückner and Donald Sannella. Structuring specifications in-the-large and in-the-small: Higher-order functions, dependent types and inheritance in SPECTRAL. In *TAPSOFT '91: Proc. Joint Conf. on Theory and Practice of Software Development*, Brighton, 1991.

[14] Benjamin C. Pierce. Bounded quantification is undecidable. *Information and Computation*, 112(1), July 1994.

[15] Donald T. Sannella, Stefan Sokołowski, and Andrzej Tarlecki. Toward formal development of programs from algebraic specifications: Parameterisation revisited. *Acta Informatica*, 29:689–736, 1992.

[16] Donald T. Sannella and Andrzej Tarlecki. Specifications in an arbitrary institution. *Information and Computation*, 76(2/3):165–210, 1988.

[17] Stefan Sokołowski. Parametricity in algebraic specifications: A case study. Technical report, Institute of Computer Science, Polish Academy of Sciences, Gdańsk, 1989.