Type checking Parametrised Programs and Specifications in ASL+$_{\mathbf{FPC}}$

David Aspinall

School of Informatics, University of Edinburgh, U.K.
David.Aspinall@ed.ac.uk,
WWW: http://homepages.inf.ed.ac.uk/da

Abstract ASL+ [SST92] is a kernel specification language with higher-order parametrisation for programs and specifications, based on a dependently typed λ-calculus. ASL+ has an institution-independent semantics, which leaves the underlying programming language and specification logic unspecified. To complete the definition, and in particular, to study the type checking problem for ASL+, the language ASL+$_{\mathrm{FPC}}$ was conceived. It is a modified version of ASL+ for $\mathcal{FPC}$, and institution based on the paradigmatic programming calculus FPC. The institution $\mathcal{FPC}$ is notable for including *sharing equations* inside signatures, reminiscent of so-called *manifest types* or *translucent sums* in type systems for programming language modules [Ler94,HL94]. This allows type equalities to be propagated when composing modules. This paper introduces $\mathcal{FPC}$ and ASL+$_{\mathrm{FPC}}$ and their type checking systems.

1 Program Development with Institutions

A simple setup for program development with institutions [GB92] is to consider programs to be syntactic expressions denoting models from an institution $\mathcal{I}$, and specifications to be syntactic expressions denoting classes of models. More elaborate views are certainly possible (e.g., programming languages considered as institutions whose satisfaction relation is a function), but perhaps unnecessary.

One issue that must be resolved is the relationship between identifiers in the syntax, and their semantic equivalents. In particular, the possibility of *aliasing*, or as it is known in the context of modular programming, *sharing*, should be considered. While a real language may already include an understanding of sharing, the usual institutional semantics of a specification language such as ASL in equational logic $\mathcal{EQ}$ or first-order logic $\mathcal{FOL}$ does not, simply because there is no way to specify sharing in algebraic signatures. For example, given

$$\begin{array}{l}\Sigma \;=_{\mathrm{def}}\; \mathbf{sig}\\ \qquad\quad \mathbf{sorts}\ s, t\\ \qquad\quad \mathbf{opns}\ c : s,\ d : t\\ \qquad \mathbf{end}\end{array}$$

the equation "c=d" is ill-typed because c and d have distinct sorts; so this equation is not in $\mathbf{Sen}(\Sigma)$. However, flexible ways of parameter passing can

mean that the same sort can be referred to via several different identifiers, so there are occasions when this equation *should* be considered well-typed. The classical example is the "diamond-import" situation [Mac86], illustrated by the Standard ML (SML) functor heading:

```
functor F (structure S1 : sig type intset ... end
           structure S2 : sig type intset ... end
           sharing type S1.intset = S2.intset) = ...
```

The parametrised program `F` has two parameter modules `S1` and `S2`, but requires that any actual parameters have identical implementations of the `intset` type. This means that when type checking the body of the functor, the given type equation can be assumed. In the algebraic case, sometimes we may want to suppose that sorts s and t denote the same set, so $c = d$ *is* type-correct.

This issue may seem simple, but propagating type equalities properly lies at the heart of type-theoretic explanations of programming language module systems, an issue which researchers have worked on for well over a decade (contributions include [HMM90,HL94,Ler95,Ler96,Jon96,Rus99]). The design of a module type system is affected both by the type system of the underlying language, and by the flexibility of the module system: higher-order, first-class and recursive modules have all been considered. The work reported here is a first attempt to design a type system for a language which has higher-order parametrisation of both programs and specifications.

With an institution-based semantics, we have two ways to go:

Ignore sharing: e.g., by extending the satisfaction relation $\models$ to be three valued, so that $A \models \varphi \in \{ \mathbf{true}, \mathbf{false}, \mathbf{wrong} \}$. Then $\mathbf{Sen}(\Sigma)$ is extended to contain all formulae which could possibly have a denotation. So now $c = d \in \mathbf{Sen}(\Sigma)$, but if $A_c \neq A_d$, then $(A \models c = d) = \mathbf{wrong}$. This is a bit like *dynamic type checking* in programming languages, and similarly unattractive: nonsensical sentences accidentally become meaningful.

Handle sharing: e.g., by adding information to signatures, to maintain the idea of *static type checking*. Then $\mathbf{Sen}(\Sigma)$ consists only of formulae which have a denotation in the semantics, as usual. This approach seems desirable when we have languages that can be statically type-checked.

Following the second choice, there are two ways of handling sharing:

External sharing: resolve sharing outwith the institutional notion of signature. For example, we could maintain a map from "external identifiers" to "internal names," the latter being names in an algebraic signature. This is (a bit) like the 1990 SML semantics, and was suggested in the algebraic semantics sketched for Extended ML in [ST86].

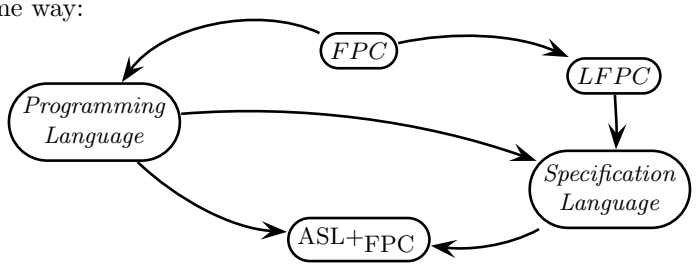
Internal sharing: make sharing part of the notion of signatures in the institution somehow. For example, to handle type sharing, signatures could be equipped with an equivalence relation on sorts.

The advantage of external sharing is keeping our familiar institutions. The considerable disadvantage is that we break the institution-independent framework:

for example, specification building operators of ASL must be lifted to operate on the "external" part of algebraic signatures, and general results must be reproved. (Nonetheless, this route has been followed for the semantics of CASL by introducing *institutions with symbols* [Mos99].)

The internal sharing alternative means that we must modify the institution. But after that, we can apply the general institution-independent framework. We treat signatures as *static typing environments* which contain all that's needed to type-check terms and formulae; $\mathbf{Sen}(\Sigma)$ is exactly the set of well-typed formulae in the abstract syntax over Σ. This is the route that we follow for ASL+$_{\mathrm{FPC}}$.

ASL+$_{\mathrm{FPC}}$ is an attempt to give a complete but small definition of a formal development framework. We start from the fixed-point calculus FPC, which is a prototypical expression language for higher-order functional programming. Then we define syntax and semantics for a programming language, specification language and logic, and fit these into a λ-calculus used for structuring, based on ASL+ [SST92,Asp95b]. The syntax and semantics of each part are put together in the same way:



LFPC, the logic for FPC, is based on higher-order logic with an axiomatisation of the CPO order relation for the underlying fixed-point semantics of FPC. The final result, ASL+$_{\mathrm{FPC}}$, allows higher-order parameterisation of both programs and specifications, as well as the specification of parametrised programs, as studied in the abstract setting of ASL+ [SST92,Asp95b,Asp97].

In Section 2, we give the definitions of the institution $\mathcal{FPC}$. Section 3 introduces syntax and semantics for $\mathcal{FPC}$ signatures and programs in context, and Section 4 describes the full module language ASL+$_{\mathrm{FPC}}$. Section 5 concludes.

2 An Institution for FPC

FPC [Plo85] is an extension of the simply-typed lambda calculus with products, sums, and recursive types. The expressiveness of FPC is well-known: familiar datatypes are built beginning from the empty type $\mu a.a$ and we can define a fixed point operator for each function type $s \multimap p$.

In practice, type expressions are too cumbersome to write out, so we need type abbreviations. Similarly, real programming languages use definitions to avoid repeating functions. So to the minimal FPC calculus we add type and term constants (in algebraic terminology, these are the sort and operation names).

Let *TyVar* and *TyConst* be disjoint countable sets of *type variables* and *type constants*. Types of FPC are given by the grammar:

$$t ::= c \quad | \quad a \quad | \quad t \multimap t \quad | \quad t \times t \quad | \quad t + t \quad | \quad \mu a.t$$

where $a \in \mathsf{TyVar}$ and $c \in \mathsf{TyConst}$. Free and bound variables of a type are defined as usual, and α-convertible types are considered syntactically identical. Substitution of the type s for the type variable a in the type t is written $[s/a]t$. (Similar conventions and notation are used henceforth without note). Given a subset $Ty \subseteq \mathsf{TyConst}$, we write $\mathsf{ProgTypes}(Ty)$ for the set of closed types whose type constants are contained in the set Ty.

Terms of FPC are parametrised on a notion of signature, which is equipped with *sharing equations* for type constants. Let $\mathsf{TmConst}$ be a countable set of term constants.

Definition 1. *An $\mathcal{FPC}$* signature Σ *is a triple* $(Ty^\Sigma, Sh^\Sigma, Tm^\Sigma)$ *where*

- $Ty^\Sigma \subseteq \mathsf{TyConst}$
- $Sh^\Sigma : Ty^\Sigma \to Fin(\mathsf{ProgTypes}(Ty^\Sigma))$
- $Tm^\Sigma : \mathsf{TmConst} \rightharpoonup \mathsf{ProgTypes}(Ty^\Sigma)$

We let Σ stand variously for any component when no confusion would arise, and we write $=_\Sigma$ for the equality relation on $\mathsf{ProgTypes}(Ty^\Sigma)$ defined as the compatible closure of equalities introduced by Sh^Σ (i.e., $c = t$ for $t \in \Sigma(c)$).

The idea is that sharing equations induced by Sh^Σ are used during type checking. In practice, useful signatures will have unifiable equations (considering Ty^Σ as variables); we only want equalities which arise from abbreviations and aliased names. Having Sh^Σ as a partial function would suffice for our purposes, but defining $Sh^\Sigma(c)$ instead as a set of equations allows signatures to be put together easily. Checking that a finite signature is unifiable is a simple first-order unification problem (we do not unfold recursive types).

Terms in FPC are given by the grammar:

$$\begin{array}{l} e ::= v \quad | \quad x \quad | \quad \mathbf{fun}\,(x : t).\, e \quad | \quad e\ e \\ \quad | \quad \langle e, e \rangle \quad | \quad \mathbf{fst}(e) \quad | \quad \mathbf{snd}(e) \\ \quad | \quad \mathbf{inl}_{t+t}(e) \quad | \quad \mathbf{inr}_{t+t}(e) \quad | \quad \mathbf{case}\ e\ \mathbf{of}\ \mathbf{inl}(x) \Rightarrow e\ \mathbf{or}\ \mathbf{inr}(x) \Rightarrow e \\ \quad | \quad \mathbf{intro}_{\mu a.t}(e) \quad | \quad \mathbf{elim}(e) \end{array}$$

where $v \in \mathsf{TmConst}$ ranges over term constants and $x \in \mathsf{TmVar}$ ranges over a set of term variables.

Terms are type-checked with the standard rules, together with a rule for typing term constants and a rule for using type equality:

$$\frac{\Sigma(v) = t}{G \ \triangleright^\Sigma \ v : t} \qquad \frac{G \ \triangleright^\Sigma \ e : s \qquad s =_\Sigma t}{G \ \triangleright^\Sigma \ e : t}$$

As usual, the type checking judgement $G \ \triangleright^\Sigma \ e : t$ uses a context G of type assignments $x : t$ giving types $t \in \mathsf{ProgTypes}(\Sigma)$ to variables.

Definition 2. *A* signature morphism $\sigma : \Sigma \to \Sigma'$ *is a pair* (Ty^σ, Tm^σ) *where* $Ty^\sigma : Ty^\Sigma \to Ty^{\Sigma'}$ *and* $Tm^\sigma : Dom(Tm^\Sigma) \to Dom(Tm^{\Sigma'})$ *are functions such that for all $c \in \Sigma$, $t \in \Sigma(c) \Longrightarrow \sigma(c) =_{\Sigma'} \sigma(t)$ and for all $v \in \Sigma$, $\Sigma(v) = t \Longrightarrow Tm^{\Sigma'}(\sigma(v)) =_{\Sigma'} \sigma(t)$.*

A special case of signature morphism is the inclusion between a signature Σ and a richer one Σ' having more constants or equalities.

Definition 3 (FPC subsignatures and inclusions). *A signature Σ is a* subsignature *of Σ', written $\Sigma \subseteq \Sigma'$, if $Ty^{\Sigma} \subseteq Ty^{\Sigma'}$, $t \in \Sigma(c) \Longrightarrow c =_{\Sigma'} t$, and $\Sigma(v) = t \Longrightarrow Tm^{\Sigma'}(v) =_{\Sigma'} t$. If $\Sigma \subseteq \Sigma'$, then there is a canonical morphism $\iota_{\Sigma,\Sigma'} : \Sigma \hookrightarrow \Sigma'$, the* inclusion of Σ in Σ', *comprising the evident inclusions.*

Example 1. Define three signatures by:

$\Sigma_1 =_{\text{def}}$	$\Sigma_2 =_{\text{def}}$	$\Sigma_3 =_{\text{def}}$
sig	**sig**	**sig**
type c	**type** c	**type** c
val $v : (c \times \mathit{bool})$	**type** d	**type** d
$\times(c \times \mathit{bool})$	**sharing** $d = c \times \mathit{bool}$	**val** $v : (c \times \mathit{bool}) \times d$
end	**val** $v : d \times d$	**end**
	end	

Then $\Sigma_1 \subseteq \Sigma_2$ and $\Sigma_3 \subseteq \Sigma_2$ but Σ_1 and Σ_3 are unrelated.

If $\Sigma \subseteq \Sigma'$ and $\Sigma' \subseteq \Sigma$ we consider Σ and Σ' semantically equivalent. Under this equivalence and a similar one on signature morphisms, we get the category $\mathbf{Sign}^{\mathcal{FPC}}$. We usually work with particular concrete representatives.

2.1 FPC Algebras

FPC has a standard fixed-point semantics given using a universal domain (see e.g., [Gun92]). Using this we define interpretations for FPC types $\mathcal{M}[\![t]\!]_\iota$ in a type environment ι and well-typed FPC terms $\mathcal{M}[\![e : t]\!]_{\iota\rho}$ in a pair of environments ι, ρ. The details are routine, except to require that a Σ-type environment ι is defined on both type variables and constants, and respects the sharing relation in Σ (if $t \in \Sigma(c)$ then $\iota(c) = \mathcal{M}[\![t]\!]_\iota$). Similarly, we call ρ a (G, Σ, ι)-FPC environment if it maps term constants and variables to appropriate domains as required by G, Σ, and ι.

The interpretation of terms is preserved by signature change. This is the main part of the satisfaction condition.

Definition 4 (Environment reducts). *Let $\sigma : \Sigma \to \Sigma'$ be a signature morphism. Suppose ι is a Σ'-type environment and ρ is a $(\sigma(G), \Sigma', \iota)$-FPC environment. We define $\iota|_\sigma$ and $\rho|_\sigma$ by:*

$$\iota|_\sigma(c) = \begin{cases} \iota(\sigma(c)) & \text{for } c \in \Sigma, \\ \text{undefined otherwise.} \end{cases} \qquad \rho|_\sigma(v) = \begin{cases} \rho(\sigma(v)) & \text{for } v \in \Sigma, \\ \text{undefined otherwise.} \end{cases}$$
$$\iota|_\sigma(a) = \iota(a) \qquad \rho|_\sigma(x) = \rho(x)$$

for all c, a, v, x. It follows directly that $\iota|_\sigma$ is a Σ-type environment and $\rho|_\sigma$ is a $(G, \Sigma, \iota|_\sigma)$-FPC environment.

Proposition 1 (FPC meaning is preserved by signature change). *Let $\sigma : \Sigma \to \Sigma'$ be a signature morphism. Suppose ι is a Σ'-type environment and ρ is a $(\sigma(G), \Sigma', \iota)$-FPC environment. Then*

- $\mathcal{M}[\![t]\!]_{\iota|_\sigma} = \mathcal{M}[\![\sigma(t)]\!]_\iota$ *and*
- $\mathcal{M}[\![G \rhd^\Sigma e : t]\!]_{\iota|_\sigma \rho|_\sigma} = \mathcal{M}[\![\sigma(G) \rhd^{\Sigma'} \sigma(e) : \sigma(t)]\!]_{\iota\rho}$.

An *FPC Σ-algebra* A is now defined as a pair (ι_A, ρ_A) of a suitable type environment and term environment for an FPC signature Σ. We can define a model functor by setting $\mathbf{Mod}^{\mathcal{FPC}}(\Sigma)$ to be the discrete category of FPC Σ-algebras.

2.2 LFPC, A Logic for FPC

A suitable logic for FPC can be based on Gordon's HOL logic [GM93], which has equality, implication, and a choice operator as primitive; other connectives are definable. We add FPC types and the cpo order relation $\sqsubseteq$ inherited from the fixed-point semantics, to give the types and terms of LFPC:

$$\tau ::= t \quad | \quad \mathbf{prop} \quad | \quad \tau \to \tau$$
$$h ::= \Lambda z{:}\tau.\, h \quad | \quad h(h) \quad | \quad z \quad | \quad h = h \quad | \quad h \Longrightarrow h \quad | \quad \epsilon z{:}\tau.\, h \quad | \quad e \sqsubseteq e$$

where $t \in \mathit{ProgTypes}(\Sigma)$ and $z \in \mathit{LogVar} \supset \mathit{TmVar}$ ranges over a new countable set LogVar of logical variables. The typing judgement $G \rhd^\Sigma e : \tau$ is defined for a fixed signature Σ and a context of bindings $z : \tau$.

Notice that the logical function space is distinct from the programming language one, and **prop** is distinct from any FPC type of booleans. No base type of individuals is necessary because FPC already includes types denoting countably infinite collections. Adding rules to axiomatise $\sqsubseteq$ gives us a higher-order logic of computable functions (similar to e.g., [MNOS99]). By the anti-symmetry of $\sqsubseteq$, FPC terms are automatically embedded in the logic, since we may express e as $\epsilon x{:}t.\, x \sqsubseteq e \ \bigwedge \ e \sqsubseteq x$. Because $\mathit{LogVar} \supset \mathit{TmVar}$, we can abstract over terms of the programming language inside the logic, but not vice-versa.

The semantics of LFPC is given using the standard set-theoretic construction for HOL. Each type denotes a non-empty set: **prop** is a two element set, $\tau \to \tau$ denotes a set of functions. The FPC type t is interpreted as the underlying set of the domain which interprets t. To define a sentence functor for $\mathcal{FPC}$, we set $\mathbf{Sen}^{\mathcal{FPC}}(\Sigma)$ to be the set of LFPC Σ-terms of type **prop**. The satisfaction condition is straightforward to verify, extending signature morphisms and Proposition 1 to terms of the logic.

Lemma 1 (Satisfaction condition for $\mathcal{FPC}$). *Let A be a Σ'-algebra and $\sigma : \Sigma \to \Sigma'$ a signature morphism. Then $A|_\sigma \models^{\mathcal{FPC}}_\Sigma \varphi$ iff $A \models^{\mathcal{FPC}}_{\Sigma'} \sigma(\varphi)$.*

3 Syntax for Signatures, Algebras, and Renamings

When writing parametrised programs and specifications, or using separate compilation of program parts, we have a *context* of declared programs and speci-

fications; the context corresponds to the formal parameters or module interface. Working in a context, we use signature expressions which may not themselves be closed, but are closed when they are added to the context. Triples $(Ty^{\Sigma}, Sh^{\Sigma}, Tm^{\Sigma})$ which have the same form as signatures but may not be closed are called *pre-signatures*. There is an inclusion between the signature of the context and the overall signature; if Σ_{ctx} is the former, we write $\Sigma_{ctx} \subseteq_{\cup} \Sigma$ to indicate that Σ is pre-signature such that $\Sigma_{ctx} \subseteq \Sigma_{ctx} \cup \Sigma$, where $\cup$ is "sequential" union of signatures. In this case, Σ is a signature-in-context. An algebra-in-context is then given by a function $f : \mathbf{Mod}(\Sigma_{ctx}) \to \mathbf{Mod}(\Sigma)$ which expands any Σ_{ctx}-algebra A to a Σ-algebra $f(A)$, so that $f(A)|_{\Sigma_{ctx}} = A$ (such an f is sometimes called a *persistent constructor* [ST88]).

Definition 5 (Signature morphism in context). *Given Σ_1, Σ_2 such that $\Sigma_{ctx} \subseteq_{\cup} \Sigma_1$ and $\Sigma_{ctx} \subseteq_{\cup} \Sigma_2$, a* signature morphism in context Σ_{ctx} *between them is defined to be an FPC signature morphism $\sigma : \Sigma_{ctx} \cup \Sigma_1 \to \Sigma_{ctx} \cup \Sigma_2$ such that*

$$\begin{array}{ccc} \Sigma_{ctx} & \xrightarrow{\ id\ } & \Sigma_{ctx} \\ {\scriptstyle \iota_1}\downarrow & & \downarrow{\scriptstyle \iota_2} \\ \Sigma_{ctx} \cup \Sigma_1 & \xrightarrow[\ \sigma\ ]{} & \Sigma_{ctx} \cup \Sigma_2 \end{array}$$

(i.e., the action of σ on Σ_{ctx} is the identity).

The grammar for syntactic signatures, signature morphisms, and algebras is:

$$\begin{array}{rcl} S & ::= & \mathbf{sig}\ sdec^{\star}\ \mathbf{end} \\ sdec & ::= & \mathbf{type}\ c \quad | \quad \mathbf{val}\ v : t \quad | \quad \mathbf{sharing}\ c = t \\ P & ::= & \mathbf{alg}\ pdec^{\star}\ \mathbf{end} \\ pdec & ::= & \mathbf{type}\ c = t \quad | \quad \mathbf{val}\ v : t = e \\ s & ::= & [renam^{\star}] \\ renam & ::= & c \mapsto c \quad | \quad v \mapsto v \end{array}$$

Each form has a type checking judgement:

$$\begin{array}{lll} \Sigma_{ctx} \ \triangleright\ S \implies \Sigma & & \text{In } \Sigma_{ctx},\ S \text{ has pre-signature } \Sigma \\ \Sigma_{ctx} \ \triangleright\ P \implies \Sigma & & \text{In } \Sigma_{ctx},\ P \text{ has pre-signature } \Sigma \\ \Sigma_{ctx} \ \triangleright\ s \implies \Sigma \to \Sigma' & & \text{In } \Sigma_{ctx},\ s \text{ is a renaming from } \Sigma \text{ to } \Sigma' \end{array}$$

The first two judgements are inference judgements, since the pre-signature Σ is determined by the syntactic signature S or the program P. A renaming, on the other hand, does not determine its source or destination signature uniquely.

The typing rules are straightforward. They ensure that $\Sigma_{ctx} \cup \Sigma$ and $\Sigma_{ctx} \cup \Sigma'$ are proper signatures. The rule for adding a sharing equation is this:

$$\frac{\Sigma_{ctx} \ \triangleright\ sdecs \implies \Sigma \qquad \begin{array}{l} t \in \mathsf{ProgTypes}(\Sigma_{ctx} \cup \Sigma) \\ \mathit{Unifiable}(\Sigma_{ctx} \cup \Sigma \cup \{\, c = t \,\}) \end{array}}{\Sigma_{ctx} \ \triangleright\ sdecs\ \mathbf{sharing}\ c = t \implies \Sigma \cup \{\, c = t \,\}}$$

The third premise ensures that the new equation is consistent with the equalities known so far. For typable phrases, it is easy to give a semantics.

Definition 6 (Interpretation of syntax in context).

- $[\![\Sigma_{ctx} \triangleright S \implies \Sigma]\!]$ *is the signature in context* $\Sigma_{ctx} \cup \Sigma$.
- $[\![\Sigma_{ctx} \triangleright s \implies \Sigma \to \Sigma']\!]$ *is the signature morphism in context* $\sigma : [\Sigma_{ctx}]\Sigma \to \Sigma'$ *determined by* s.
- $[\![\Sigma_{ctx} \triangleright P \implies \Sigma]\!]$ *is the functor* $f_P : \mathbf{Mod}(\Sigma_{ctx}) \to \mathbf{Mod}(\Sigma_{ctx} \cup \Sigma)$ *given by*

$$f_P(A) = \mathcal{P}[\![\Sigma_{ctx} \triangleright pdecs \implies \Sigma]\!](\iota_A, \rho_A)$$

where $P \equiv$ ***alg*** *pdecs* ***end*** *and* $\mathcal{P}[\![-]\!]_-$ *is defined by induction on the derivation of* $\Sigma_{ctx} \triangleright pdecs \implies \Sigma$, *extending* ι_A *and* ρ_A *in an obvious way.*

4 Modular Programs and Specifications in ASL+$_{\mathbf{FPC}}$

ASL+$_{\mathrm{FPC}}$ is based on the syntax for $\mathcal{FPC}$ of the previous sections. This syntax is combined using ASL-style specification building operators and a λ-calculus. There is a single syntactic category of pre-terms:

$$\begin{array}{rl} M ::= & X \mid P \mid S \\ & \mid \textbf{impose } \varphi \textbf{ on } M \mid \textbf{derive from } M \textbf{ by } s : S \\ & \mid \textbf{translate } M \textbf{ by } s \mid \textbf{enrich } M \textbf{ with } M \\ & \mid \lambda X{:}M.\, M \mid M\, X \mid \Pi X{:}M.\, M \mid \mathsf{Spec}(M) \\ & \mid \mathsf{Let}\ X = M\ \mathsf{in}\ M : M \end{array}$$

Variables X range over a countable set ModVar. Meta-variables SP, A, M are all used to range over the set of pre-terms, with the hint that SP will denote a specification (collections of FPC algebras), and A some arbitrary collection.

Space precludes a complete motivation and explanation of the ASL+ calculus; we give only a brief overview. First, the ASL operators **impose**, etc, have their usual intentions in building specifications. The λ-calculus portion consists of λ-abstraction, application to variables, and Π-quantification for parametric (architectural) specifications. The $\mathsf{Spec}(-)$ operator formalizes specification refinement: $SP' : \mathsf{Spec}(SP)$ asserts that $SP \rightsquigarrow SP'$. This allows parametrised specifications and programs which accept any refinement of their formal parameter, written $\lambda X{:}\,\mathsf{Spec}(SP).\, M$ (semantically, $\mathsf{Spec}(-)$ is understood as a powerset operator). Finally, the let construct allows local definitions of modules, to relieve the restriction on function applications. It also imposes a signature or specification constraint: in $\mathsf{Let}\ X = A\ \mathsf{in}\ M : SP$, the constraining specification SP may be used to hide some details of the implementation M; in particular it *must* hide any mention of X from the result signature of M. (This prevents exporting a hidden symbol; module type systems solve this problem in varying ways).

Contexts for ASL+$_{\mathrm{FPC}}$ contain declarations and definitions for module variables. They may also directly include specifications, to allow "pervasive" datatypes of the language which are visible everywhere (`BOOLEAN`, `INTEGER`, etc).

$$\Gamma ::= \langle\rangle \quad | \quad \Gamma, X : A \quad | \quad \Gamma, X = M \quad | \quad \Gamma, SP$$

For type checking, we extract an FPC signature from a context Γ. This will include all of the pervasive elements, but also, any variables which stand for algebras will be included with their signatures, renamed using a "dot renaming" function to prefix identifiers with the module variable name. (We must assume the existence of suitable dot-renaming functions on TyVar, TmVar). Given a signature Σ, we write $X.\Sigma$ for the dot-renamed signature. Dot notation provides a way for programs to refer to components of modules. The notation can only be used on module variables because the syntax of FPC does not include $\mathrm{ASL}+_{\mathrm{FPC}}$ expressions; this restricts type propagation in the higher-order case.

4.1 Type checking with Rough Types

Now we come to the main novelty in the development. ASL+ is equipped with two formal systems: one for proving satisfaction of a specification by a program, and the other for "rough" typing, which is designed to isolate the "static" type checking component of satisfaction. We follow the same plan in $\mathrm{ASL}+_{\mathrm{FPC}}$, except that rough types are improved to allow type equalities to be propagated from argument to result in parametrised programs. This generalisation is really the crux of the new system. Rough types have the syntax:

$$\kappa ::= \Sigma \quad | \quad \pi X : A.\kappa \quad | \quad \mathsf{P}(\kappa)$$

where Σ ranges over pre-signatures. A program denoting a Σ-algebra will have rough type Σ; a Σ-specification expression will have type $\mathsf{P}(\Sigma)$. The π-types classify functions. The main way that sharing information is propagated is through equations in pre-signatures that refer to the environment (e.g., $c = X.c$). The reason that the domain A of a type $\pi X : A.\kappa$ is a full ASL+ term is to account properly for sharing propagation between successive specification and program parameters; retaining a full term here allows rough types to be recalculated (see [Asp97] for further explanation).

There are three typing judgements:

$\Gamma \Longrightarrow_{\mathsf{sig}} \Sigma_\Gamma$	Σ_Γ is the underlying FPC signature of Γ
$\Gamma \triangleright \kappa \leq \kappa'$	κ is a subtype of κ'
$\Gamma \triangleright M \Longrightarrow \kappa$	M has rough type κ

These judgements are defined in Figures 1–4, described in turn below.

Underlying signature (Figure 1). This judgement also serves to say that the context is well-formed. The underlying FPC signature is made by combining pre-signatures for the pervasive parts of the context, together with the dot-renamed components $X.\Sigma$ for variables X which range over Σ-algebras.[1] Module variables which have non-signature types (the rules assume κ is a non-signature) do not contribute to the FPC signature of the context; there is no way to use them directly in any FPC type or term.

[1] a sort of "flattening" operation, reminiscent of the way Java treats inner classes.

$$\frac{}{\langle\rangle \implies_{\mathsf{sig}} \emptyset}$$

$$\frac{\Gamma \implies_{\mathsf{sig}} \Sigma_\Gamma \qquad \Gamma \triangleright SP \implies \mathsf{P}(\Sigma)}{\Gamma, SP \implies_{\mathsf{sig}} \Sigma_\Gamma \cup \Sigma}$$

$$\frac{\Gamma \implies_{\mathsf{sig}} \Sigma_\Gamma \qquad \Gamma \triangleright SP \implies \mathsf{P}(\kappa)}{\Gamma, SP \implies_{\mathsf{sig}} \Sigma_\Gamma}$$

$$\frac{\Gamma \implies_{\mathsf{sig}} \Sigma_\Gamma \qquad \Gamma \triangleright SP \implies \mathsf{P}(\Sigma)}{\Gamma, X : SP \implies_{\mathsf{sig}} \Sigma_\Gamma \cup X.\Sigma}$$

$$\frac{\Gamma \implies_{\mathsf{sig}} \Sigma_\Gamma \qquad \Gamma \triangleright SP \implies \mathsf{P}(\kappa)}{\Gamma, X : SP \implies_{\mathsf{sig}} \Sigma_\Gamma}$$

$$\frac{\Gamma \implies_{\mathsf{sig}} \Sigma_\Gamma \qquad \Gamma \triangleright M \implies \Sigma}{\Gamma, X = M \implies_{\mathsf{sig}} \Sigma_\Gamma \cup X.\Sigma}$$

$$\frac{\Gamma \implies_{\mathsf{sig}} \Sigma_\Gamma \qquad \Gamma \triangleright M \implies \kappa}{\Gamma, X = M \implies_{\mathsf{sig}} \Sigma_\Gamma}$$

Figure 1. Underlying signature of a context

$$\frac{\Gamma \implies_{\mathsf{sig}} \Sigma_\Gamma \qquad \Sigma_\Gamma \subseteq_\cup \Sigma \qquad \Sigma_\Gamma \subseteq_\cup \Sigma' \qquad (\Sigma_\Gamma \cup \Sigma') \subseteq_{\mathsf{sh}} (\Sigma_\Gamma \cup \Sigma)}{\Gamma \triangleright \Sigma \leq \Sigma'}$$

$$\frac{\Gamma \triangleright A_1 \implies \kappa \qquad \Gamma \triangleright A_2 \implies \kappa \qquad \Gamma, X : A_2 \triangleright \kappa_1 \leq \kappa_2}{\Gamma \triangleright \pi X : A_1.\kappa_1 \leq \pi X : A_2.\kappa_2} \qquad \frac{\Gamma \triangleright \kappa \leq \kappa'}{\Gamma \triangleright \mathsf{P}(\kappa) \leq \mathsf{P}(\kappa')}$$

Figure 2. Subtyping rules for rough types

Subtyping rules (Figure 2). We write $\Sigma_1 \subseteq_{\mathsf{sh}} \Sigma_2$ if $\Sigma_1 \subseteq \Sigma_2$ but $Ty^{\Sigma_1} = Ty^{\Sigma_2}$ and $Tm^{\Sigma_1} = Tm^{\Sigma_2}$. In this case Σ_2 only differs from Σ_1 in having more sharing. The subtyping rules lift this relation to a relation on rough types. The rule for π-rough types appears as if it allows contravariancy in the domain; in fact, it does not because the rough types of A_1 and A_2 are required to be the same.

Programs and ASL terms (Figure 3). The rules for rough typing ASL terms, including FPC signatures and algebras, involve some signature calculation. The first two rules invoke the type checking system for the core-level from Section 3. The rule for **impose** checks that φ is a well-typed proposition.

The rules for **derive** and **translate** use renaming syntax, allowing some polymorphism. Arguments of **derive from** $-$ **by** $s : S$ or of **translate** $-$ **by** s can have any signature which fits suitably with s, according to the type checking rules for signature morphisms. The result signature of **derive** has to be given, but the result of **translate** is inferred, as the smallest image[2] of s. In fact, the rule for **translate** can be understood as constructing a pushout by propagating extra sharing; relying on the natural polymorphism of the syntax for renamings (as opposed to a semantic signature morphism in $\mathbf{Sign}^{\mathcal{FPC}}$), this happens auto-

[2] This means that **translate** only uses surjective signature morphisms; but we can express translation along inclusions **translate** SP **by** $\iota : \Sigma \hookrightarrow \Sigma'$ using **enrich**.

$$\frac{\Gamma \Longrightarrow_{\mathsf{sig}} \Sigma_\Gamma \qquad \Sigma_\Gamma \rhd S \Longrightarrow \Sigma}{\Gamma \rhd S \Longrightarrow \mathsf{P}(\Sigma)} \qquad \frac{\Gamma \Longrightarrow_{\mathsf{sig}} \Sigma_\Gamma \qquad \Sigma_\Gamma \rhd P \Longrightarrow \Sigma}{\Gamma \rhd P \Longrightarrow \Sigma}$$

$$\frac{\Gamma, SP \Longrightarrow_{\mathsf{sig}} \Sigma \qquad \rhd^{\Sigma} \varphi : \mathbf{prop}}{\Gamma \rhd \mathbf{impose}\ \varphi\ \mathbf{on}\ SP \Longrightarrow \mathsf{P}(\Sigma)}$$

$$\frac{\Gamma \rhd SP \Longrightarrow \mathsf{P}(\Sigma') \qquad \Gamma \Longrightarrow_{\mathsf{sig}} \Sigma_\Gamma \qquad \begin{array}{c}\Sigma_\Gamma \rhd S \Longrightarrow \Sigma \\ \Sigma_\Gamma \rhd s \Longrightarrow \Sigma \to \Sigma'\end{array}}{\Gamma \rhd \mathbf{derive\ from}\ SP\ \mathbf{by}\ s : S \Longrightarrow \mathsf{P}(\Sigma)}$$

$$\frac{\Gamma \rhd SP \Longrightarrow \mathsf{P}(\Sigma) \qquad \Gamma \Longrightarrow_{\mathsf{sig}} \Sigma_\Gamma \qquad \Sigma_\Gamma \rhd s \Longrightarrow \Sigma \to s(\Sigma)}{\Gamma \rhd \mathbf{translate}\ SP\ \mathbf{by}\ s \Longrightarrow \mathsf{P}(s(\Sigma))}$$

$$\frac{\Gamma \rhd SP \Longrightarrow \mathsf{P}(\Sigma) \qquad \Gamma, SP \rhd SP' \Longrightarrow \mathsf{P}(\Sigma')}{\Gamma \rhd \mathbf{enrich}\ SP\ \mathbf{with}\ SP' \Longrightarrow \mathsf{P}(\Sigma \cup \Sigma')}$$

Figure 3. Rough typing programs and ASL terms

matically. The rule for **derive**, by contrast, is provided with an explicit target signature, so any sharing in SP beyond that required by Σ' will be disregarded.

The rule for **enrich** is similar to a rule for the dependent sum in type theory: just as x occurs bound in B in the term $\Sigma x{:}A.\, B$, so all the symbols of SP occur bound in SP' in the term **enrich** SP **with** SP'. This non-symmetry in **enrich** isn't revealed by its usual definition in terms of **translate** and **union**. The directly defined semantics of **enrich** SP **with** SP' also shows the dependency: models of the result are extensions of models of SP.

ASL+ terms (Figure 4). First, a variable which ranges over Σ-algebras is given a special *strengthened* type. The signature Σ/X is defined as Σ with the sharing equations augmented, so that $Sh^{\Sigma/X}(c) = Sh^{\Sigma}(c) \cup \{\, c = X.c \,\}$. This reflects the sharing of X with the context, since it denotes a projection on the X.-named part of the underlying environment. Strengthening was introduced by Leroy [Ler94] and a similar rule is present in most module type systems.

Rules for λ-abstractions and Π-abstractions are straightforward. Applications are restricted to variables; it may be necessary to rename the bound variable of the Π-type of the function to match the operand. The application rule is the crucial place where type identities are propagated. Subtyping here allows the actual parameter to have a richer type with more sharing equations than the type of the formal parameter A. Propagation of the type identities occurs because after application any mention of $X.c$ in the result type κ' will refer to a variable declared in the context, possibly having more sharing equations, rather than the bound variable of the Π-type.

$$\frac{\Gamma \Longrightarrow_{\mathsf{sig}} \Sigma_\Gamma \qquad \Gamma \triangleright A \implies \mathsf{P}(\Sigma)}{\Gamma, X : A, \Gamma' \triangleright X \implies \Sigma/X}$$

$$\frac{\Gamma \Longrightarrow_{\mathsf{sig}} \Sigma_\Gamma \qquad \Gamma \triangleright A \implies \mathsf{P}(\kappa)}{\Gamma, X : A, \Gamma' \triangleright X \implies A}$$

$$\frac{\Gamma, X : A \triangleright M \implies \kappa}{\Gamma \triangleright \lambda X{:}A.\, M \implies \pi X : A.\kappa}$$

$$\frac{\Gamma, X : A \triangleright B \implies \mathsf{P}(\kappa)}{\Gamma \triangleright \Pi X{:}A.\, B \implies \mathsf{P}(\pi X : A.\kappa)}$$

$$\frac{\begin{array}{ll}\Gamma \triangleright M \implies \pi X : A.\kappa' & \Gamma \triangleright X \implies \kappa_x \\ \Gamma \triangleright A \implies \mathsf{P}(\kappa) & \Gamma \triangleright \kappa_x \leq \kappa\end{array}}{\Gamma \triangleright M\, X \implies \kappa'}$$

$$\frac{\Gamma \triangleright A \implies \mathsf{P}(\kappa)}{\Gamma \triangleright \mathsf{Spec}(A) \implies \mathsf{P}(\mathsf{P}(\kappa))}$$

$$\frac{\begin{array}{l}\Gamma \triangleright M \implies \kappa_M \\ \Gamma, X : \lceil \kappa_M \rceil \triangleright N \implies \kappa_N \\ \Gamma \triangleright A \implies \mathsf{P}(\kappa) \\ \Gamma, X : \lceil \kappa_M \rceil \triangleright \kappa_N \leq \kappa\end{array}}{\Gamma \triangleright \mathsf{Let}\ X = M\ \mathsf{in}\ N : A \implies \kappa}$$

Figure 4. Rough typing ASL+ terms

The rule for a binding $\mathsf{Let}\ X = M\ \mathsf{in}\ N : A$ allows N to be typed in the context extended by the typing of M and checks that the type of the constraint A is correct. The rough type of A is typed in Γ, so the dependency on X must be removed. The notation $\lceil \kappa \rceil$ in this rule embeds the rough type as a term of the calculus, defined by replacing Σ by its syntax, $\pi X : A.\kappa'$ by $\Pi X{:}A.\lceil \kappa' \rceil$ and $\mathsf{P}(\kappa)'$ by $\mathsf{Spec}(\kappa')$. This is simply a trick to avoid introducing a notion of rough-context (context with rough typing assumptions); when we project from the context, we get the rough type κ (or a strengthened version) again.

4.2 Brief Example

The following example shows how type equalities are propagated. We will build up a context of declarations step-by-step. First:

$$\begin{array}{l}\Gamma_1 =_{\text{def}} \mathtt{ELT} = \mathbf{sig} \\ \qquad\qquad\qquad\quad \mathbf{type}\ \mathtt{elt} \\ \qquad\qquad\quad \mathbf{end}\end{array}$$

If the denotation of this expression is Σ_{ELT}, then we have the rough typing $\langle\rangle \triangleright \mathtt{ELT} \implies \mathsf{P}(\Sigma_{ELT})$. Now we declare a parametrised program for building lists over some Σ_{ELT}-algebra:

$$\begin{array}{l}\Gamma_2 =_{\text{def}} \Gamma_1, \mathtt{List} = \lambda\, \mathtt{Elt} : \mathtt{ELT}\ .\ \mathbf{alg} \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{type}\ \mathtt{elt} = \mathtt{Elt.elt} \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{type}\ \mathtt{list} = \mathit{list}_{\mathtt{elt}} \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{val}\ \mathtt{nil} : \mathtt{list} = \ldots \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{val}\ \mathtt{cons} : \mathtt{list} = \ldots \\ \qquad\qquad\qquad\qquad\qquad\quad \mathbf{end}\end{array}$$

($\mathit{list}_{\texttt{elt}}$ is a type-expression in FPC which expresses the type of lists over the type `elt`; the dots are filled with appropriate terms). This has the rough typing $\Gamma \triangleright \texttt{List} \implies \Pi\texttt{Elt:ELT}.\, \Sigma_{LIST}[\texttt{Elt.elt}]$. The inferred signature of the algebra `List` is $\Sigma_{LIST}[\texttt{Elt.elt}]$, where the square brackets are informal notation to indicate a dependency on `Elt`. To be more exact: this is the signature of lists extended with the equation $\texttt{elt} = \texttt{Elt.elt}$. Now we may apply the `List` program to an algebra, for example:

$$\Gamma_3 =_{\text{def}} \Gamma_2, \texttt{Nat} = \textbf{alg} \quad \textbf{type}\ \texttt{elt} = \mathit{nat} \quad \textbf{end}$$

(where *nat* is an FPC type expression for natural numbers). Then $\Gamma_3 \triangleright \texttt{Nat} \implies \Sigma_{ELT}[\mathit{nat}]$ where $\Sigma_{ELT}[\mathit{nat}] = \Sigma_{ELT} \cup \{\, \texttt{elt} = \mathit{nat} \,\}$. Now we can derive $\Gamma_3 \triangleright \texttt{List Nat} \implies \Sigma_{LIST}[\texttt{Nat.elt}]$. Define $\Gamma_4 = \Gamma_3, \texttt{ListNat} = \texttt{List Nat}$. In the underlying FPC signature (such that $\Gamma_4 \Longrightarrow_{\mathsf{sig}} \Sigma_{\Gamma_4}$), we have the equation $\texttt{ListNat.elt} = \mathit{nat}$, which means that we can apply natural number functions to elements of `ListNat` lists.

This very simple example demonstrates propagation of type equalities for the application of `List`. To prevent it, we could define an opaque version of list:

$$\texttt{OpaqueList} = \mathit{Let}\ L = \texttt{List}\ \mathit{in}\ L : \Pi\texttt{Elt:ELT}.\, \Sigma_{LIST}$$

In a declaration $\texttt{OList} = \texttt{OpaqueList Nat}$ the type identity of the `OList.elt` is unknown, so we could only pass elements of this list around.

4.3 Results and Further Developments

One important and non-trivial result is the decidability of rough type checking.

Theorem 1 (Decidability). *If all signatures are finite, each of the rough typing judgements is decidable.*

Proof. (Outline). First, observe that type checking in FPC and LFPC for finite signatures is decidable. For a slightly different formulation of the rough typing system viewed as an algorithm, we can give a measure on the inputs to each judgement which decreases from conclusions to premises of each rule. □

A set-theoretic semantics for ASL+$_{\text{FPC}}$ is given in [Asp97] together with a soundness proof. It interprets each of the typing judgements given above. However, the interpretation function is partial: rough type checking alone cannot guarantee that specifications are consistent, nor that actual arguments to parametrised programs or specifications meet the axiomatic requirements of their formal parameters.

To guarantee the well-definedness of an ASL+$_{\text{FPC}}$ term, we may need to do theorem proving. This is provided for with the satisfaction system, which incorporates ideas from other research into proof in structured specification.

5 Further Work, Related Work

The work here is mostly taken from Chapters 6 and 7 of my PhD thesis [Asp97], which contains additional results and full definitions. Theorem 1 is a new result. The work began from the conception of adding type equations to algebraic signatures to explain sharing, an idea which occurred earlier to Tarlecki [Tar92]. The system here draws somewhat on later ideas of programming language researchers in investigating type systems for program modules, particularly those of Leroy [Ler95] and Harper and Lillibridge [HL94].

The most closely related and recent work in the algebraic specification community is on CASL's architectural specifications [SMT$^+$01]. This retains institution independence, but at the expense of complexity and for a language more restricted than ASL+.

Research is still highly active in the programming languages community in the quest to find more expressive type systems which are easier to understand and have good properties such as decidability (which failed for [HL94]). See e.g., [Sha98,Sha99,Jud97,Jon96,Rus99,DCH03]. Space precludes a detailed survey, but one aspect is worthy of note: several recent systems employ *singleton kinds* [SH00,DCH03] as an alternative to manifest types, as a way of succinctly internalising type equalities within the type system. The original version of ASL+ [SST92] in fact included a singleton construct (isolated in [Asp95a]) to allow a program to be turned into a trivial specification, and it was suggested how the dot notation could be expressed using this construct. (There are also connections here with the work of Cengarle [Cen94] who defined a syntax with an operator $Sig(-)$ for extracting the signature of an actual parameter; her work is an older relative of ASL+$_{\mathrm{FPC}}$).

In the end, it is a challenge to balance the various requirements and give a feasible system for type checking. The solution here is not ideal and has drawbacks outlined in [Asp97]. Typing modules for a specification language like ASL+ has different requirements to the programming case, and the system proposed here should be regarded only as a first attempt at a type-theoretic solution.

Towards Edinburgh CASL

One future venture we would like to undertake is the design and implementation of a CASL extension for a subset of Standard ML. While the specification constructs and CASL variations have received a great deal of attention, connection to specific programming languages remains relatively unexplored. A significant exception is the work at Bremen on HasCASL [SM02], which has parallels with what we want to do (and connections with work described above). We have early design ideas for a CASL extension called Edinburgh CASL, which is dedicated to specification for a subset of Standard ML, and constructed using a type-theoretic approach similar approach to ASL+$_{\mathrm{FPC}}$. We go beyond FPC in considering additional features of SML like polymorphism and pattern matching.

Since ASL+$_{\mathrm{FPC}}$ was invented, improvements to generic institutional technology were developed which may allow a more abstract approach (for example,

using institutions with symbols and derived signature morphisms, instead of the concrete sharing relation in $\mathcal{FPC}$ signatures); however, these may not help with more advanced features such as first-class modules. And it remains important to verify that abstract constructions produce the desired result in different scenarios, by experimenting with ways of adding programming languages to CASL.

Acknowledgements. I'm grateful to Don Sannella and Andrzej Tarlecki for their guidance during development of this work, and the collaboration with Don since on ideas for Edinburgh CASL described above. Thanks are also due to the HasCASL team in Bremen (especially Till Mossakowski) for discussions.

References

[Asp95a] David Aspinall. Subtyping with singleton types. In *Proc. Computer Science Logic, CSL'94, Kazimierz, Poland*, LNCS 933. Springer-Verlag, 1995.

[Asp95b] David Aspinall. Types, subtypes, and ASL+. In *Recent Trends in Data Type Specification*, LNCS 906. Springer-Verlag, 1995.

[Asp97] David Aspinall. *Type Systems for Modular Programs and Specification*. PhD thesis, Department of Computer Science, University of Edinburgh, 1997.

[Cen94] María Victoria Cengarle. *Formal Specifications with Higher-Order Parameterisation*. PhD thesis, Institut für Informatik, Ludwig-Maximilians-Universität München, 1994.

[DCH03] Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules. In *Proceedings of POPL 2003, New Orleans*, 2003.

[GB92] J. A. Goguen and R. M. Burstall. Institutions: abstract model theory for specification and programming. *Journal of the ACM*, 39:95–146, 1992.

[GM93] M. J. C. Gordon and T. F. Melham. *Introduction to HOL*. Cambridge University Press, 1993.

[Gun92] Carl A. Gunter. *Semantics of Programming Languages*. MIT Press, 1992.

[HL94] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Conference Record of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'94)*, pages 123–137, Portland, Oregon, January 17–21, 1994. ACM Press.

[HMM90] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Conference record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 341–354, San Francisco, CA, January 1990.

[Jon96] Mark P. Jones. Using parameterized signatures to express modular structure. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, St. Petersburg, Florida, 21–24, 1996. ACM Press.

[Jud97] Judicaël Courant. An applicative module calculus. In *TAPSOFT*, Lecture Notes in Computer Science, pages 622–636, Lille, France, April 1997. Springer-Verlag.

[Ler94] Xavier Leroy. Manifest types, modules, and separate compilation. In *Proc. 21st symp. Principles of Programming Languages*, pages 109–122. ACM press, 1994.

[Ler95] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)*, pages 142–153, San Francisco, California, January 22–25, 1995. ACM Press.

[Ler96] Xavier Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(5):667–698, 1996.

[Mac86] David MacQueen. Using dependent types to express modular structure. In *Proceedings, Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 277–286, St. Petersburg Beach, Florida, January 13–15, 1986. ACM SIGACT-SIGPLAN, ACM Press.

[MNOS99] Olaf Müller, Tobias Nipkow, David von Oheimb, and Oskar Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9:191–223, 1999.

[Mos99] Till Mossakowski. Specifications in an arbitrary institution with symbols. In *Proc. 14th WADT 1999*, volume LNCS 1827, pages 252–270, 1999.

[Plo85] Gordon Plotkin. Denotational semantics with partial functions. Lecture at C.S.L.I. Summer School, 1985.

[Rus99] Claudio V. Russo. Non-dependent types for standard ML modules. In *Principles and Practice of Declarative Programming*, pages 80–97, 1999.

[SH00] Christopher A. Stone and Robert Harper. Deciding type equivalence in a language with singleton kinds. In *ACM Symposium on Principles of Programming Languages (POPL), Boston, Massachusetts*, pages 214–227, 19–21, 2000.

[Sha98] Z. Shao. Parameterized signatures and higher-order modules, 1998. Technical Report YALEU/DCS/TR-1161, Dept. of Computer Science, Yale University, August 1998.

[Sha99] Zhong Shao. Transparent modules with fully syntactic signatures. In *International Conference on Functional Programming*, pages 220–232, 1999.

[SM02] Lutz Schrder and Till Mossakowski. HasCASL: Towards integrated specification and development of Haskell programs. In *Proceedings of AMAST 2002*, 2002.

[SMT+01] Lutz Schrder, Till Mossakowski, Andrzej Tarlecki, Bartek Klin, and Piotr Hoffman. Semantics of Architectural Specifications in CASL. In *Proc. FASE 2001*, volume LNCS 2029, pages 253–268, 2001.

[SST92] Donald Sannella, Stefan Sokołowski, and Andrzej Tarlecki. Toward formal development of programs from algebraic specifications: Parameterisation revisited. *Acta Informatica*, 29:689–736, 1992.

[ST86] Donald Sannella and Andrzej Tarlecki. Extended ML: An institution-independent framework for formal program development. In David H. Pitts, editor, *Proc. Workshop on Category Theory and Computer Programming*, LNCS 240, pages 364–389. Springer-Verlag, 1986.

[ST88] Donald Sannella and Andrzej Tarlecki. Toward formal development of programs from algebraic specifications: implementation revisited. *Acta Informatica*, 25:233–281, 1988.

[Tar92] Andrzej Tarlecki. Modules for a model-oriented specification language: a proposal for MetaSoft. In *Proc. 4th European Symposium on Programming ESOP'92*, LNCS 582, pages 452–472. Springer-Verlag, 1992.