

**No Type Stamps and No Structure Stamps —
a Referentially-Transparent Higher-Order
Module Language**

Franz-Josef Grosch



Informatik-Bericht Nr. 95-05
Juli 1995

© Abteilung Softwaretechnologie
Institut für Programmiersprachen
Technische Universität Braunschweig
Gaußstraße 17
D-38106 Braunschweig
Germany

No Type Stamps and No Structure Stamps – a Referentially-Transparent Higher-Order Module Language

Franz-Josef Grosch*

Abteilung Softwaretechnologie
Technische Universität Braunschweig, Germany

Abstract

A language for programming-in-the-large should support architectural descriptions of an entire product line. The evaluation of programs-in-the-large should generate the architecture of individual products and should link implementations-in-the-small to form an executable product. The Standard ML (SML) module language [20] could serve this task, but it is not referentially transparent. Furthermore, it does not distinguish between architectural and implementational concerns.

This paper presents a general module language that is based on a typed λ -calculus extended by *systems*. Overcoming SML’s drawbacks, the module language permits syntactic control of type and structure generativity resulting in referential transparency. Furthermore, splitting up architectural and implementational concerns, the implementation of elementary modules is not part of the module language. The module calculus is essentially simply typed, with elementary modules and types as simple values, and interfaces and kinds as types. Function and system interfaces are value-dependent types.

Key words: module language, ML, type systems

1 Introduction

A main task in software engineering is to arrange modules into a sensible system architecture. Today a good architectural design should describe an entire product line in terms of a *reference architecture* [7]. The reference architecture should enable individual product architectures to be instantiated, refined or generated. Following the observations that “programming languages are frozen software engineering

knowledge” [13] and “programming-in-the-large is essentially typed functional programming” [27] a language for programming-in-the-large should generate individual product architectures by the evaluation of functional architectural programs.

The Standard ML (SML) module language [20, 19, 16] supports a functional style for programming-in-the-large: *Structures* are elementary components, *functors* are component-valued functions and *signatures* are component specifications. Large systems can be built by successively applying component-valued functions to previously constructed components. This approach has successfully been used in the development of large software systems [3, 6].

The SML module language can serve as a model for a general module language, independent of the embedded implementation language [27]. However, it has some disadvantages:

- it is not referentially transparent, and
- programming-in-the-large is not clearly separated from implementing single components.

Referential transparency and equational reasoning about large systems are inhibited by the fact that every functor application generates a new structure. Thus, structure and type generativity may be regarded as side-effects on the module level. They require a rather complicated stamp-based semantics, especially in combination with higher-order functors [18].

From a software-engineering point of view, architectural concerns like defining interfaces and designing the system architecture should be completely separated from the implementation of single components. The module system of Modula-2 [30] supports this separation. Interfaces are defined in *definition modules* and the system architecture is designed by import lists referring to module names. Completely separated, the implementations of single components

*This work is funded by the Deutsche Forschungsgemeinschaft, grant Sn11/4-1. Author’s current address: Technische Universität Braunschweig, Abteilung Softwaretechnologie, Gaußstr. 17, D-38092 Braunschweig/Germany. E-mail: grosch@ips.cs.tu-bs.de.

are contained in *implementation modules*. But, a system architecture in Modula-2 is frozen. SML’s functors (parameterized modules) enable a flexible architecture to be designed, but a functorized architecture cannot be defined without supplying implementations; architectural and implementational concerns are mixed up.

In this work we present a typed, referentially transparent and higher-order module language that may serve as a general module language for several typed implementation languages. Module programs describe architectures for a product line. The evaluation of a module program generates a product architecture, and, if possible, links implementations in order to form an executable program. The formal foundation for the module language is an extended typed λ -calculus called the $\lambda\delta$ -calculus, which permits the syntactic control of generativity. The typing is based on dependent types [2, 29] due to manifest specifications [10, 14, 15] in signatures. The calculus is essentially simply typed, with elementary modules and types as simple values, with elementary signatures and kinds as simple types, and with value-dependent functions and systems as type constructors. Implementations of elementary modules are not part of the calculus, i. e. the calculus constitutes a separate layer above any implementation language.

1.1 Syntactic control of generativity

We now intuitively explain the syntactic control of generativity using Core-SML as the paradigmatic implementation language. Partially adopting SML’s terminology, we call an elementary module a *structure* and the interface for a structure a *signature*.

The basis for our language is a typed lambda calculus. The well-known identity function of the lambda calculus ($\lambda x:T. x$) becomes the identity function for module expressions:

```

module id =
  fun x: sig
    type t
    val equal: t * t -> bool
  end
-> x

```

The interface of parameter x must describe a structure containing a type definition for t and a function `equal`. Now assume there is a structure named `int0rd` with a matching interface. Then every application `(id int0rd)` reduces to the same component `int0rd` according to the usual β -conversion.

`int0rd` is the name of a structure that is neither **fun**-bound nor predefined. In order to bind such

components we introduce a second binding operator called (*system*) *declaration*. Such a declaration introduces (**dec**) the name of a structure (or a type) and its interface description—very similar to a definition module in Modula-2. The name can be used inside the module expression following the back arrow `<-`. This module expression is called *export part*, because it allows the access to a system.

```

module makeInt0rd =
  dec int0rd: sig
    type t = int
    val less: t * t -> bool
    val equal: t * t -> bool
  end
<- (id int0rd)

```

Since `(id int0rd)` reduces to `int0rd`, the system `makeInt0rd` only declares the structure name `int0rd` and exports this structure. The module expression `(id int0rd)` is correctly typed, because the interface of `int0rd` is a subinterface of the required interface for the argument x , according to the usual notion of module subtyping [10, 14].

The normal form for `makeInt0rd` is a complete module expression without any free identifiers. In order to access the export part an additional construct **bind** is needed. For example

```
bind m to makeInt0rd in (id m)
```

permits to use the export part of system `makeInt0rd` by the name m in the module expression `(id m)` following keyword **in**. In order to prevent `int0rd` from escaping its scope, the scope for `int0rd` is expanded:

```

bind m to makeInt0rd in (id m)
≡ bind m to
  (dec int0rd: sig .. end <- (id int0rd))
  in (id m)
≡ dec int0rd: sig .. end
  <- (id m)[m := (id int0rd)]
≡ dec int0rd: sig .. end <- (id (id int0rd))
≡ dec int0rd: sig .. end <- int0rd

```

The reduction shows that **dec**-bindings never disappear, only their scope can be expanded. Similar to β -conversion the reduction for **bind** must consider α -conversion to prevent name clashes. As expected the expression reduces to an expression equivalent to `makeInt0rd`, since only the identity function is applied to the export part.

Function abstractions and system declarations can be mixed as desired. For example, generic modules, similar to *generic packages* in Ada [1] or *templates* in C++ [28], are described as module functions returning a system.

```

module makeSet =
  fun elements : sig
    type t
    val equal: t * t -> bool
  end
-> dec set : sig
  type element = elements.t
  type set
  val empty : set
  val member : element -> set
    -> bool
  end
<- set

```

makeSet is a module function parameterized with a structure elements that requires a type definition t and a function equal. It returns a system (**dec** set : .. <- set) describing a set that depends on the parameter elements. The dependency is expressed in the *manifest type specification* for element. Manifest specifications substitute *sharing constraints* as used in SML. Besides types also substructures can be specified manifestly.

Using the module expression linkIntOrdSet a product architecture can be generated:

```

module linkIntOrdSet =
  bind intElems to makeIntOrd
  in (makeSet intElems)

```

Evaluating linkIntOrdSet results in a specific system declaring the name of an element and a set structure and exporting the set structure.

```

linkIntOrdSet
≡ bind intElems to
  (dec intOrd: sig .. end <- intOrd)
  in (makeSet intOrd)
≡ dec intOrd: sig .. end
  <- (makeSet intElems)[intElems := intOrd]
≡ dec intOrd: sig .. end
  <- (dec set: sig .. intOrd.t .. end
    <- set)

```

So far, no structure implementations have been supplied and the evaluation of the module expression only generates a skeletal product architecture. However, the resulting system is generated in a referentially transparent way by the evaluation of interface-correct module expressions.

1.2 Supplying implementations

Structure implementations establish the connection between module and implementation language. If an implementation is given for each structure name, the

evaluation of a module expression not only generates a skeletal architecture but also links implementations forming an executable program. In order to achieve this, an implementation has to be supplied for every **dec**-bound name. The syntax requires the implementation to follow the signature. The implementation can access those modules also visible in the signature. In our toy example, two structure declarations have to be implemented.

```

module makeIntOrd =
  dec intOrd: sig ... end
  is struct
    type t = int
    fun less (x, y) = ...
    fun equal (x, y) = (x = y)
    ...
  end
<- intOrd

module makeSet =
  fun elements : sig ... end
-> dec set : sig ... end
  is struct
    type element = elements.t
    type set = element list
    val empty = []
    fun member x s = ..elements.equal..
    ...
  end
<- set

```

While the implementation language determines the module language's signatures, structure implementations do not belong to the module language. Structure implementations are only connected to the module name and have no influence on the evaluation of module expressions. If the structure implementations match the given signatures, they are linked during evaluation.

2 The $\lambda\delta$ module calculus

We now formally introduce the $\lambda\delta$ module calculus, $\Lambda\delta$ for short.

Although unusual from a theoretical point of view, generativity, especially type generativity, is a common feature in many programming languages. In SML's module language, the generation of type and structure names may be regarded as a side effect. The central idea of $\Lambda\delta$ is to introduce a new binding construct (δ) which controls the generation of new type and structure names syntactically.

Module expressions:

t	$::=$	t_p	module path
		$\lambda x:T. t$	(function) abstraction
		$t_1 t_2$	application
		$\delta x:T. t$	(system) declaration
		bind x to t_1 in t_2	use

t_p	$::=$	x	module identifier
		l	structure/type label
		$t.l$	structure component

Interface expressions:

T	$::=$	T_s	simple interface
		$\Pi x:T_1. T_2$	function interface
		$T t$	application interface
		$\exists x:T_1. T_2$	system interface
		Bind x To t In T	use interface

T_s	$::=$	$\{ E \}$	signature
		type	kind

E	$::=$	D, E	signature entries
		ε	

D	$::=$	$l:T$	abstract structure/ type specification
		$l:T = t$	manifest structure/ type specification
		$l:\tau$	value specification

τ	$::=$	t_p	type path
		$int \mid bool$	implementation
		$\tau \rightarrow \tau \mid list(\tau) \mid \dots$	language types

Figure 1: Syntax for $\Lambda\delta$

2.1 Module expressions

Figure 1 shows the syntax for $\Lambda\delta$. Module expressions (t) are built from function abstraction and application and from system declaration and use. The reduction rule for application is β -reduction in consideration of α -conversion (Figure 2).

A module expression may be referred to by a module path t_p . A module path is either a module identifier x or a label qualified by a module expression. Of course, the qualifying expression must denote a structure name. Inside a signature (see 2.2), a module expression may be referred to by a label directly.

A system declaration introduces a structure or type name, which is bound in the body. The body is called *export part*, since it can be accessed from outside us-

Module expressions:

$(\lambda x:T. t_1) t_2$	\rightarrow_t	$t_1[x := t_2]$
bind x_2 to $(\delta x_1:T. t_1)$		
in t_2	\rightarrow_t	$\delta x_1:T_1. (t_2[x_2 := t_1])$

Interface expressions:

$(\Pi x:T_1. T_2) t$	\rightarrow_T	$T_2[x := t]$
Bind x_2 To $(\delta x_1:T_1. t)$		
In T_2	\rightarrow_T	$\exists x_1:T_1. (T_2[x_2 := t])$

Manifest specifications:

$$l:T = t \rightarrow_D l:(T/t) = t$$

Definition of $/$:

$$\begin{aligned} \{ E \}/t &= \{ E/t \} \\ (l:T, E)/t &= (l:(T/(t.l)) = t.l, E/t) \\ (l:T = u, E)/t &= (l:T = u, E/t) \\ (l:\tau, E)/t &= (l:\tau, E/t) \\ T/t &= T \end{aligned}$$

Figure 2: Reduction rules

ing **bind**. The system use **bind** x **to** t_1 **in** t_2 enables the export part of system t_1 to be named x in module expression t_2 . The reduction rule for system use is shown in figure 2. Again α -conversion is assumed to prevent name clashes. A system is used by extending the scope of a δ -bound name and accessing the export part. The reduction rule shows that δ -bound names never disappear. Due to α -conversion two uses of the same system “generate” distinct δ -bound names:

$$\begin{aligned} &\lambda f:F. \text{let } m = \delta a:A. a \text{ in} \\ &\quad \text{bind } x \text{ to } m \text{ in} \\ &\quad \text{bind } y \text{ to } m \text{ in } f x y \\ \rightarrow_t &\lambda f:F. \delta a:A. \text{bind } y \text{ to } \delta a:A. a \text{ in } f a y \\ \rightarrow_t &\lambda f:F. \delta a:A. \delta a':A. f a a' \end{aligned}$$

There is no notion of *structure implementation* in the module calculus. Structures are introduced by a δ -bound name and its signature; they are definition modules in the sense of Modula-2. Structure implementations have no influence on static or operational semantics of $\Lambda\delta$.

A module expression containing only δ s is regarded as *observable* and describes a (sub)system architecture.

2.2 Interfaces

$\Lambda\delta$ is typed. The types of module expressions are interfaces. Simple interfaces (T_s) comprise signatures,

and the kind **type** of all types. Signature entries can be selected by labels; they specify types, substructures, and values of the implementation language. Types as well as substructures may be specified abstractly or manifestly. We assume that abstract signature entries are named together with the surrounding structure. Therefore, the reduction rule \rightarrow_D uses the operation $/$ in order to propagate manifest equalities. Operation $/$ (Figure 2) recursively traverses all signature entries and incorporates manifest equalities for abstract type and abstract structure specifications. The following example demonstrates the use of \rightarrow_D :

$$\begin{array}{l} \delta a : \{ t : \mathbf{type} \} . \\ \delta b : \{ m : \{ t : \mathbf{type} \} = a, v : m.t \} . b \\ \rightarrow_D \delta a : \{ t : \mathbf{type} \} . \\ \delta b : \{ m : \{ t : \mathbf{type} = a.t \} = a, v : m.t \} . b \end{array}$$

Due to manifest specifications we obtain a notion of value-dependent types or, more precisely, module-dependent interfaces. Abstractions have dependent function interfaces of the form $(\Pi x : T_1 . T_2)$, where x may occur free in T_2 . $(T t)$ determines the interface for a function application. A system interface is similar to a weak dependent sum $(\exists x : T_1 . T_2)$, where again x may occur free in T_2 . **(Bind x To t In T)** determines the interface for a system use. Figure 2 shows the reduction rules for dependent interfaces.

$$\begin{array}{ll} (i) & (\delta a : A . a) : (\exists a : A . A) \\ (ii) & (\mathbf{bind } m \text{ to } (\delta a : A . a) \text{ in } m) \\ & \quad : (\mathbf{Bind } m \text{ To } (\delta a : A . a) \text{ In } A) \\ (iii) & \mathbf{Bind } m \text{ To } (\delta a : A . a) \text{ In } A \\ & \xrightarrow{T} \exists a : A . (A[m := a]) \\ & \equiv \exists a : A . A \end{array}$$

The examples show: (i) a very simple system and its interface, (ii) a system use and its interface, and (iii) the reduction of a system-use interface.

2.3 Interface inference rules

The presentation of $\Lambda\delta$'s interface inference rules follows Barendregt's presentation of typed lambda calculi in [2]. There are constants $*$ and $*_s$. $*$ is the interface kind of all interfaces and $*_s$ is the interface kind of all simple interfaces, which in turn is an interface subkind of $*$. A *declaration* D is of the form $x : T, l : T_s, l : T_s = t$ or $l : \tau$, where x is a module identifier and l is a label. A *statement* S is of the form $t : T, T : *, T : *_s$. $\mathit{lab}(E)$ denotes the set of labels used in a sequence of signature entries. A

context Γ is a finite, ordered sequence of declarations each with different identifiers or labels.

The notion $\Gamma \vdash S$ is defined by the axioms and rules shown in figure 3.

There are two axioms. According to the four forms of declarations, there are four start rules and corresponding weakening rules. The interface formation rules ensure that interface expressions are wellformed. Note the restriction in the formation rule for system interfaces: interfaces for a δ -bound name have to be simple (kind $*_s$), i. e. a δ declares a type or a structure name. The interface formation rules also include restrictions for abstractions: the interface for a λ -bound identifier must be either a system interface, a function interface or a simple interface, but no application or use interface.

The rules for abstraction and application are the standard rules for value-dependent function types [2, 29].

A system interface is described by $\exists x : T_1 . T_2$. There are similarities to weak existential sums which justify this notation. $\delta x : T_2 . t$ might be considered as a pair consisting of a module name x and a module expression $t[x]$ containing x as a free variable. The interface $\exists x : T_2 . T_1$ is a module-dependent sum of interfaces $T_1[x]$, where x ranges over implementations satisfying T_2 . In fact it is a *weak* dependent sum, since the implementation itself is hypothetical; it is not even part of the calculus.

The commonly used elimination rule for weak dependent sums [29, 5] is:

$$\frac{\Gamma \vdash t_1 : (\exists y : T_1 . T_3) \quad \Gamma, y : T_1, x : T_3 \vdash t_2 : T_2}{\Gamma \vdash (\mathbf{open } t_1 \text{ as } (y, x) \text{ in } t_2) : T_2}$$

with the additional constraint that y must not escape its scope. This is not appropriate if we consider y being a structure and x being a module expression depending on y . In contrast, it is necessary that t_2 and T_2 depend on y . In order to solve the problem, we use **bind** as described above instead of **open**. Unlike **open**, **bind** enables only the export part of a system declaration to be accessed. The corresponding typing rule is:

$$\frac{\Gamma \vdash t_1 : (\exists y : T_1 . T_3) \quad \Gamma, y : T_1, x : T_3 \vdash t_2 : T_2}{\Gamma \vdash (\mathbf{bind } x \text{ to } t_1 \text{ in } t_2) : (\mathbf{Bind } x \text{ To } t_1 \text{ In } T_2)}$$

The assumptions are the same as in the rule for **open**, while the inferred interface is calculated by **Bind**.

The interfaces of structure components are deduced by the structure and access rules. The opacity rule allows to forget manifest type or structure identities. The transparency rule guarantees that a structure

(axiom) $\langle \rangle \vdash \{ \} : *_s \quad \langle \rangle \vdash \mathbf{type} : *_s$

(start) $\frac{\Gamma \vdash T : *}{\Gamma, x:T \vdash x : T}, x \notin \Gamma \quad \frac{\Gamma \vdash T : *}{\Gamma, l:T \vdash l : T}, l \notin \Gamma \quad \frac{\Gamma \vdash t : T \quad \Gamma \vdash T : *}{\Gamma, l:T = t \vdash l : T = t}, l \notin \Gamma \quad \frac{\Gamma \vdash \tau : \mathbf{type}}{\Gamma, l:\tau \vdash l : \tau}, l \notin \Gamma$

(weakening) $\frac{\Gamma \vdash S \quad \Gamma \vdash T : *}{\Gamma, x:T \vdash S}, x \notin \Gamma \quad \frac{\Gamma \vdash S \quad \Gamma \vdash T : *}{\Gamma, l:T \vdash S}, l \notin \Gamma \quad \frac{\Gamma \vdash S \quad \Gamma \vdash t : T \quad \Gamma \vdash T : *}{\Gamma, l:T = t \vdash S}, l \notin \Gamma$

$\frac{\Gamma \vdash S \quad \Gamma \vdash \tau : \mathbf{type}}{\Gamma, l:\tau \vdash S}, l \notin \Gamma$ (subkind) $\frac{\Gamma \vdash T : *_s}{\Gamma \vdash T : *}$

(interface formation) $\frac{\Gamma \vdash T_1 : * \quad \Gamma, x:T_1 \vdash T_2 : *}{\Gamma \vdash (\Pi x:T_1 . T_2) : *} \quad \frac{\Gamma \vdash T_1 : *_s \quad \Gamma, x:T_1 \vdash T_2 : *}{\Gamma \vdash (\exists x:T_1 . T_2) : *}$

$\frac{\Gamma \vdash T : *_s \quad \Gamma, l:T \vdash \{ E \} : *_s, l \notin \mathit{lab}(E)}{\Gamma \vdash \{ l:T, E \} : *_s}, l \notin \mathit{lab}(E) \quad \frac{\Gamma \vdash \tau : \mathbf{type} \quad \Gamma, l:\tau \vdash \{ E \} : *_s, l \notin \mathit{lab}(E)}{\Gamma \vdash \{ l:\tau, E \} : *_s}, l \notin \mathit{lab}(E)$

$\frac{\Gamma \vdash t : T \quad \Gamma \vdash T : *_s \quad \Gamma, l:T = t \vdash \{ E \} : *_s, l \notin \mathit{lab}(E)}{\Gamma \vdash \{ l:T = t, E \} : *_s}, l \notin \mathit{lab}(E)$

(abstraction) $\frac{\Gamma, x:T_1 \vdash t : T_2 \quad \Gamma \vdash (\Pi x:T_1 . T_2) : *}{\Gamma \vdash (\lambda x:T_1 . t) : (\Pi x:T_1 . T_2)}$ (application) $\frac{\Gamma \vdash t_1 : (\Pi x:T_2 . T_1) \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash (t_1 t_2) : ((\Pi x:T_2 . T_1) t_2)}$

(declaration) $\frac{\Gamma, x:T_1 \vdash t : T_2 \quad \Gamma \vdash (\exists x:T_1 . T_2) : *}{\Gamma \vdash (\delta x:T_1 . t) : (\exists x:T_1 . T_2)}$ (use) $\frac{\Gamma \vdash t_1 : (\exists y:T_1 . T_3) \quad \Gamma, y:T_1, x:T_3 \vdash t_2 : T_2}{\Gamma \vdash (\mathbf{bind } x \mathbf{ to } t_1 \mathbf{ in } t_2) : (\mathbf{Bind } x \mathbf{ To } t_1 \mathbf{ In } T_2)}$

(structure) $\frac{\Gamma \vdash t : \{ l:T, E \}}{\Gamma \vdash t : (\{ E \}[l := t.l])} \quad \frac{\Gamma \vdash t_1 : \{ l:T = t_2, E \}}{\Gamma \vdash t_1 : (\{ E \}[l := t_2])} \quad \frac{\Gamma \vdash t : \{ l:\tau, E \}}{\Gamma \vdash t : \{ E \}}$

(component) $\frac{\Gamma \vdash t : \{ l:T, E \}}{\Gamma \vdash t.l : T} \quad \frac{\Gamma \vdash t_1 : \{ l:T = t_2, E \}}{\Gamma \vdash t_1.l : T = t_2} \quad \frac{\Gamma \vdash t : \{ l:\tau, E \}}{\Gamma \vdash t.l : \tau}$

(opacity) $\frac{\Gamma \vdash t_1 : T = t_2}{\Gamma \vdash t_1 : T}$ (transparency) $\frac{\Gamma \vdash t : T}{\Gamma \vdash t : T/t}$

(conversion) $\frac{\Gamma \vdash t : T_1 \quad \Gamma \vdash T_2 : * \quad T_1 \equiv_R T_2}{\Gamma \vdash t : T_2} \quad \frac{\Gamma \vdash t : T_1 \quad \Gamma \vdash T_2 : * \quad T_1 <: T_2}{\Gamma \vdash t : T_2} \quad \frac{\Gamma \vdash t : T_1 \quad \Gamma \vdash T_2 : * \quad T_1 \approx_\Gamma T_2}{\Gamma \vdash t : T_2}$

Figure 3: Interface inference rules

determines the identity for its generative signature entries—it makes abstract identities manifest.

The conversion rules permit three different conversion relations between interfaces. Relation \equiv_R is a congruence relation, defining syntactic equality between interfaces under consideration of the reduction rules \rightarrow_t , \rightarrow_T and \rightarrow_D , up to α -conversion. Relation $<:$ defines subtyping between interfaces; function interfaces are contravariant, while system interfaces are covariant. Relation \approx_Γ defines manifest equality of

interfaces. Figure 4 shows the definitions for $<:$ and \approx_Γ .

Structure implementations are not part of $\Lambda\delta$. The link to the implementation language is established by value specifications. Value specifications in signatures do not contribute to the static and dynamic semantics of $\Lambda\delta$. They simply specify exported values which can be used by other implementations. However, interface formation rules and access rules take value specifications into consideration. At this point, the

Subtyping of interfaces:

$$\begin{array}{lcl}
\tau <: \tau, & T <: T & \\
E_1 <: E_2 & & \Rightarrow \{E_1\} <: \{E_2\} \\
E <: \varepsilon & & \\
E_1 <: E_2 & & \Rightarrow (l:T, E_1) <: E_2 \\
E_1 <: E_2 & & \Rightarrow (l:T = t, E_1) <: E_2 \\
E_1 <: E_2 & & \Rightarrow (l:\tau, E_1) <: E_2 \\
T_1 <: T_2, E_1 <: E_2 & & \Rightarrow (l:T_1 = t, E_1) <: (l:T_2, E_2) \\
T_1 <: T_2, E_1 <: E_2 & & \Rightarrow (l:T_1, E_1) <: (l:T_2, E_2) \\
T_1 <: T_2, E_1 <: E_2 & & \Rightarrow (l:T_1 = t, E_1) <: (l:T_2 = t, E_2) \\
\tau_1 <: \tau_2, E_1 <: E_2 & & \Rightarrow (l:\tau_1, E_1) <: (l:\tau_2, E_2) \\
T_2 <: T_1, U_1 <: U_2 & & \Rightarrow (\Pi x:T_1. U_1) <: (\Pi x:T_2. U_2) \\
T_1 <: T_2 & & \Rightarrow (T_1 t) <: (T_2 t) \\
T_1 <: T_2, U_1 <: U_2 & & \Rightarrow (\exists a:T_1. U_1) <: (\exists a:T_2. U_2) \\
T_1 <: T_2 & & \Rightarrow (\mathbf{Bind} x \mathbf{To} t \mathbf{In} T_1) <: (\mathbf{Bind} x \mathbf{To} t \mathbf{In} T_2)
\end{array}$$

Manifest equality of interfaces:

$$\begin{array}{lcl}
T_1 \approx_{\Gamma} T_2, \Gamma \vdash t_1 : T_1 = t_2, E_1 \approx_{\Gamma'} E_2 & \Rightarrow & (l:T_1 = t_1, E_1) \approx_{\Gamma} (l:T_2 = t_2, E_2), \text{ where } \Gamma' = (\Gamma, l:T_1 = t_1) \\
T_1 \approx_{\Gamma} T_2, E_1 \approx_{\Gamma'} E_2 & \Rightarrow & (l:T_1, E_1) \approx_{\Gamma} (l:T_2, E_2), \text{ where } \Gamma' = (\Gamma, l:T_1) \\
E_1 \approx_{\Gamma} E_2 & \Rightarrow & (l:\tau, E_1) \approx_{\Gamma} (l:\tau, E_2) \\
E_1 \approx_{\Gamma} E_2 & \Rightarrow & \{E_1\} \approx_{\Gamma} \{E_2\} \\
T_1 \approx_{\Gamma} T_2, U_1 \approx_{\Gamma'} U_2 & \Rightarrow & (\Pi x:T_1. U_1) \approx_{\Gamma} (\Pi x:T_2. U_2), \text{ where } \Gamma' = (\Gamma, x:T_1) \\
T_1 \approx_{\Gamma} T_2 & \Rightarrow & (T_1 t) \approx_{\Gamma} (T_2 t) \\
T_1 \approx_{\Gamma} T_2, U_1 \approx_{\Gamma'} U_2 & \Rightarrow & (\exists x:T_1. U_1) \approx_{\Gamma} (\exists x:T_2. U_2), \text{ where } \Gamma' = (\Gamma, x:T_1) \\
T_1 \approx_{\Gamma} T_2 & \Rightarrow & (\mathbf{Bind} x \mathbf{To} t \mathbf{In} T_1) \approx_{\Gamma} (\mathbf{Bind} x \mathbf{To} t \mathbf{In} T_2) \\
\\
T \approx_{\Gamma} T & & \\
T \approx_{\Gamma} U & \Rightarrow & U \approx_{\Gamma} T \\
T \approx_{\Gamma} U, U \approx_{\Gamma} V & \Rightarrow & T \approx_{\Gamma} V
\end{array}$$

Figure 4: Conversion relations

rules are incomplete, due to the fact that rules for implementation-language types that permit to infer $\Gamma \vdash \tau : \mathbf{type}$ are omitted.

To summarize, $\Lambda\delta$ is essentially a simply-typed module calculus, with structures and types as simple values and signatures and kinds as simple types. Manifest specifications cause a notion of value-dependent interfaces, and a simple form of polymorphism is gained due to subtyping between interfaces.

3 A module language on top of Core-SML

The $\lambda\delta$ module calculus as introduced in the last section, is the foundation of a general module language, that can be used on top of several typed implementation languages. In order to compare our module

language to the SML module language and its variants, we now sketch a module language on top of core-SML as the implementation language. In general, a module language based on $\Lambda\delta$ is influenced by the embedded implementation language. Other implementation languages are possible, e.g. in [9] we use a module language on top of Modula-2.

A module program is a sequence of top-level declarations. A top-level declaration is either a named module expression (**module**) or a named interface definition (**interface**). Module expressions are translated straight forward to $\Lambda\delta$. Figure 5 shows the translation.

The interfaces for elementary modules are signatures as known from SML. The main difference is that types and substructures can be specified manifestly—this eliminates the need for *sharing constraints* as used in SML. For example, the interfaces describing an abstract EQUALITY or a manifest INTORDER (as used in

```

λ x:T.t    ~ fun x:T -> t
δ x:T.t    ~ dec x:T <- t

Π x:T1.T2 ~ all x:T1 => T2
∃ x:T1.T2 ~ some x:T1 <= T2

{ E }      ~ sig E end

```

Figure 5: Translation of $\Lambda\delta$

the introduction) can be named by the following interface definitions:

```

interface EQUALITY =
sig
  type t
  val equal: t * t -> bool
end
interface INTORDER =
sig
  type t = int
  val less: t * t -> bool
  val equal: t * t -> bool
end

```

The definition of subinterfaces is facilitated by the syntactic short cut **with** which enables signatures to be extended and signature entries to be overwritten. For example, INTORDER could be defined as a subinterface of EQUALITY:

```

interface INTORDER =
  EQUALITY with sig
    type t = int
    val less: t * t -> bool
  end

```

Additionally, we introduce parameterized interfaces. Parameterized interfaces are not part of $\Lambda\delta$, they only occur as top-level definitions. The interface INTORDER can be computed by a parameterized interface MANIFEST_ORD applied to int:

```

interface MANIFEST_ORDER =
  Fun x: type
  => (EQUALITY with sig
    type t = x
    val less : t * t -> bool
  end )
interface INTORD = (MANIFEST_ORDER int)

```

Parameterized interfaces map module expressions to module interfaces. The difference to function interfaces is quite subtle. Function interfaces can be

implemented by module functions, technically spoken, their interface kind is *. Parameterized interfaces cannot be implemented by module expressions, they are only used to compute other interfaces. For example, the kind of MANIFEST_ORDER is **type** \rightarrow *. This difference maintains the distinction

parameterized (program specification) \neq
(parameterized program) specification

required by Sanella, Sokolowski and Tarlecki [26] in the context of algebraic specifications.

3.1 Representation of software systems

In the following we compare our module language to SML's module language and its variants. First, we consider the representation of a software system. Usually, a software system is represented as a hierarchy of structures. In SML, a software system is regarded as a structure containing the next-lower level of the system hierarchy as substructures. In our module language, a software system is a module expression containing declarations only. The structures constituting a system are in topological order w.r.t. their dependencies. The export part is a single structure, usually the last structure in the topological order, possibly containing substructures describing a complex export. The essential difference to SML is that no implementations are necessary to build a concrete software system from a functorized architecture.

3.2 Fully transparent higher-order functors

Another interesting aspect is the interface description for the paradigmatic higher-order functor `apply` that has been considered by Leroy [15] and Biswas [4]. The explicit control of type and structure generativity in our calculus influences the way how type and structure equalities are propagated.

In our module language the higher-order module function `apply` is defined as:

```

interface S = sig
  type t
end
module apply =
  fun f: (all x : S => (some m : S <= S))
  -> fun x: S
  -> f x

```

`apply` has a “fully transparent” behaviour, similar to [18]. This is best explained by an example. The parameter `f` of `apply` must be a function returning a system of the given interface. The following module expression `makePair` could be a matching parameter for `f`:

```

interface RESULT =
  Fun x : S => sig
    type t = x.t * x.t
  end
module makePair =
  fun x : S
  -> dec pair : (RESULT x)
  <- pair

```

In order to use `apply` we need a second parameter for `x` that satisfies the interface constraints:

```

module makeIntMod =
  dec intMod : sig type t = int end
  <- intMod
module useApply1 =
  bind x to makeIntMod
  in bind y to (apply makePair) x
  in y

```

Interface checking `useApply1` propagates the type equalities of `pair.t` and deduces for `y` that

$$y.t \approx_{\Gamma} \text{pair.t} \approx_{\Gamma} (\text{intMod.t} * \text{intMod.t})$$

in a context Γ , where $\text{intMod.t} \approx_{\Gamma} \text{int}$.

If we decide to declare `abstractPair` as a structure containing an abstract type `abstractPair.t`:

```

module makeAbstractPair =
  fun x : S
  -> dec abstractPair : S
  <- abstractPair

```

the abstract identity is propagated in

```

module useApply2 =
  bind x to makeIntMod
  in bind y to (apply makeAbstractPair) x
  in y

```

and interface checking only can deduce for `y` that

$$y.t \approx_{\Gamma} \text{abstractPair.t}$$

However, it is possible to describe the propagation of type and structure equalities using manifest specifications. For example, `apply` could be defined as:

```

module apply =
  fun f : (all x : S =>
    (some m : (RESULT x) <= (RESULT x)))
  -> fun x : S
  -> f x

```

limiting the usability of `apply` to `useApply1`. Interface checking `useApply2` detects an error, because the interface of `makeAbstractPair` is no longer a subinterface of parameter `f`.

3.3 Generativity

In SML, generativity is sometimes used to create different instances of a structure by repeated functor applications. In our calculus, a similar effect is gained by the use of `bind`. Imagine a generic module `queue` that is to be implemented by the use of two back-to-back stacks. This could be expressed at the module level in the following way.

```

interface STACK =
  Fun x : type => sig
    type elem = x
    val push : elem -> ()
    val pop : () -> elem
    ...
  end
module makeStack =
  fun elem : type
  -> dec stack : (STACK elem)
  <- stack
interface QUEUE =
  Fun x : type => sig
    type elem = x
    val enqueue: elem -> ()
    ...
  end
module makeQueue =
  fun elem : type
  -> let elemStack = (makeStack elem)
  in bind front to elemStack
  in bind back to elemStack
  in dec queue : (QUEUE elem)
  is struct
    ... front.push ... back.push
  end
  <- queue

```

There is one parameterized subsystem `makeStack` which is instantiated to the subsystem `elemStack`. `elemStack` is used two times and “generates” two different stack structures due to the reduction rule for `bind`. The resulting subsystem is parameterized by the element type `elem`. It consists of two stacks `front` and `back` and one queue `queue`, where the queue structure is exported.

3.4 Applicative semantics

Due to the explicit control of generativity, our module language is referentially transparent. Further, the separation of module language and implementation language guarantees that module expressions cannot depend on the store. In fact, there is no notion of store at the module level. This is different to Leroy’s [15] variant of SML’s module language. He argues

that in a stratified language like SML applicative semantics for types can be sound, while applicative semantics for structures is unsound. The reason is that in SML module expressions may depend on the store and side effects prevent a referentially transparent behaviour of functor application. Leroy further argues that in a module language with side effects and modules as first class values [10], applicative semantics for function applications is always unsound.

3.5 Functorial polymorphism

In [4], Biswas identifies a new form of polymorphism found in higher-order functors. He argues that in a certain context, where for example a module function (functor) with interface

```
all x : sig type t = int end
=> sig type t = int end
```

is expected, a module function with interface

```
all x : sig type t end
=> sig type t = x.t end
```

would do as well. He calls this form of polymorphism *functorial polymorphism*. In our module language, the necessary type conversion is simply a combination of subtyping ($<:$) with manifest equality (\approx_{Γ}):

```
all x : sig type t end
=> sig type t = x.t end
<:
all x : sig type t = int end
=> sig type t = x.t end
≈Γ
all x : sig type t = int end
=> sig type t = int end
```

3.6 Phase distinction

As pointed out in [12], the use of dependent types conflicts with compile-time type checking since a type expression (expected to evaluate at compile time) may depend on arbitrary runtime expressions. Checking the equality of type expressions may involve deciding the equality of arbitrary run-time expressions, which in general is not decidable.

In our module language, the equality of run-time module expressions is decided syntactically (\equiv_R). Since the module language does not allow recursive functions, syntactic equality is sufficient for interface checking.

Furthermore, both, interface checking as well as the evaluation of module expressions is considered to happen at system compile-time. Interface checking

corresponds to the usual type checking. Evaluation of the module program generates a specific system architecture; if all implementations are supplied, the evaluation links the specific software system.

The evaluation of a linked software system, usually considered to happen at system run-time, is not part of the dynamic semantics of our module language. Therefore, using dependent interfaces does not destroy the desirable phase distinction between system compile-time and system run-time.

4 Related work

SML's module language and its variants have been the main inspiration for our module language. Manifest type specifications have also been used in [15, 14, 10]. Contrary to these approaches, we also allow manifest structure specifications.

The semantics of higher-order functors in SML's module language has been treated by several authors. In [11, 17], a treatment based on strong sums has been given. Leroy [14] provides a description based on weak sums, which is able to treat generative and nongenerative type specifications. The stamp-based approach by Biswas [4] does not handle type generativity. Only MacQueen and Tofte's work [18] takes into account structure generativity. In our approach, structure and type generativity is controlled syntactically, therefore, type and structure stamps are no longer required.

The syntactic control of generativity is inspired by Mitchell and Plotkin's work [21] on the type-theoretic characterisation of abstract types. Roughly, a system declaration is very similar to a packed abstract type, while a system use corresponds to a combination of unpacking and repacking.

The idea of embedding names into a lambda calculus for controlling side effects can also be found in Odersky's work [22]. While Odersky proposes a calculus of local names, our module calculus which controls the side effect caused by structure and type generation could be called a calculus of global names.

While we favour a strict separation of implementation and module language, other authors try to integrate module and implementation language into a single calculus. Harper and Lillebridge [10] suggest an impredicative calculus with modules as first-class values based on Girards F_{ω} (see for example [8, 2]). The seminal work of MacQueen [17] uses a predicative (ramified) calculus based on strong sums.

5 Conclusion

We started this work from a software-engineering point of view. The motivation was to describe ref-

erence architectures for an entire product line. The functorized style suggested by SML's module language [23] was the main inspiration for our module language. It supports the development of a general architecture for a whole line of similar products. The architecture for a single product is generated by the evaluation of a module program. The missing separation of architectural and implementational concerns, the rather difficult stamp-based static semantics and the non-referential transparent behaviour of functor applications in SML's module language led to the development of our calculus.

We have presented a module calculus that treats type and, for the first time, structure generativity, in a referentially transparent way without using any type or structure stamps. Manifest specifications and higher-order expressions substitute SML's sharing constraints.

Currently, we are working on a prototypical interpreter for the module language with core-SML as the implementation language. Our aim is the design of domain-specific architectures using the module language. Perhaps we can construct the implementation as a domain-specific architecture that can be instantiated for different implementation languages and, if all the implementations are supplied, generates executable interpreters simply by evaluating module expressions.

References

- [1] *Reference Manual for the Ada Programming Language*, 1983. ANSI/MIL-STD 1815 A.
- [2] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors, *Background: Computational Structures*, volume 2 of *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, New York, 1992.
- [3] Lars Birkedal, Nick Rothwell, Mads Tofte, and David N. Turner. The ML Kit version 1. Technical Report 93/14, DIKU, University of Copenhagen, Denmark, 1993.
- [4] Sandip K. Biswas. Higher-order functors with transparent signatures. In POPL-22 [25], pages 154–163.
- [5] Luca Cardelli and Xavier Leroy. Abstract types and the dot notation. In Manfred Broy and Cliff B. Jones, editors, *Proc. IFIP TC2 working conference on Programming Concepts and Methods*, pages 479–504. North-Holland, 1990.
- [6] Eric Cooper, Robert Harper, and Peter Lee. The Fox project: Advanced development of systems software. Technical Report CMU-CS-91-178, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1991.
- [7] David Garlan and Dewayne E. Perry. Introduction to the special issue on software architecture. *IEEE Transactions on Software Engineering*, 21(4):269–274, April 1994.
- [8] Jean-Yves Girard. The system F of variable types, fifteen years later. *Theoretical Computer Science*, 45:159–192, 1980.
- [9] Franz-Josef Grosch and Bernd Fischer. Architectural programming: from modules to systems (and beyond). Technical report, Abteilung Softwaretechnologie, Technische Universität Braunschweig, 1995. Forthcoming.
- [10] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In POPL-21 [24], pages 123–137.
- [11] Robert Harper and John C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, April 1993.
- [12] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *17th Symposium on Principles of Programming Languages*, pages 341–254. ACM Press, January 1990.
- [13] Gilles Kahn. Contribution to a discussion at Dagstuhl-Seminar: Future directions in software engineering. Dagstuhl-Seminar-Report: 32, February 1992.
- [14] Xavier Leroy. Manifest types, modules and separate compilation. In POPL-21 [24], pages 109–122.
- [15] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In POPL-22 [25], pages 142–153.
- [16] David MacQueen. Modules for Standard ML. In *Proc. 1984 ACM Conference on LISP and Functional Programming*, pages 198–207, New York, 1984. ACM Press.
- [17] David MacQueen. Using dependent types to express modular structure. In *13th Symposium on Principles of Programming Languages*, pages 277–286. ACM Press, January 1986.

- [18] David B. MacQueen and Mads Tofte. A semantics for higher-order functors. In Donald Sannella, editor, *Proc. 5th European Symposium on Programming*, volume 788 of *Lecture Notes in Computer Science*, pages 409–423, Edinburgh, U.K., April 1994. Springer-Verlag.
- [19] Robin Milner and Mads Tofte. *Commentary on Standard ML*. The MIT Press, Cambridge, Massachusetts, 1991.
- [20] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, Massachusetts, 1990.
- [21] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.
- [22] Martin Odersky. A functional theory of local names. In POPL-21 [24], pages 48–59.
- [23] Laurence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, Cambridge, Great Britain, 1991.
- [24] *Conference Record of POPL '94: 21th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM Press, January 1994.
- [25] *Conference Record of POPL '95: 22th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM Press, January 1995.
- [26] Donald Sannella, Stefan Sokolowski, and Andrzej Tarlecki. Towards formal development of programs from algebraic specifications: Parameterisation revisited. Technical Report 6/90, Fachbereich Informatik, Universität Bremen, 1990.
- [27] D. T. Sannella and L. A. Wallen. A calculus for the construction of modular prolog programs. *Journal of Logic Programming*, 12:147–177, 1992.
- [28] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.
- [29] Simon Thompson. *Type Theory and Functional Programming*. Addison-Wesley, 1991.
- [30] Niklaus Wirth. *Programming in Modula-2*. Springer-Verlag, 1988.

92-01	F.-J.Grosch, G.Snelting	Polymorphic Components for Monomorphic Languages
92-02	S.Conrad, M.Gogolla, R.Herzig	TROLL <i>light</i> : A Core Language for Specifying Objects
93-01	B.Fischer	A New Feature Unification Algorithm
93-02	G.Snelting	Perspektiven der Softwaretechnologie
93-03	G.Snelting, A.Zeller	Inferenzbasierte Werkzeuge in NORA
93-04	W.Rönsch, J.Schüle	Parallelisierung im Wissenschaftlichen Rechnen
93-05	P.Löhr-Richter, G.Reichwein	Object Oriented Life Cycle Models
93-06	M.Krone, G.Snelting	On the Inference of Configuration Structures from Source Code
93-07	S.Schwidderki, T.Hartmann, G.Saake	Monitoring Temporal Preconditions in a Behaviour Oriented Object Model
93-08	T.Hartmann, G.Saake	Abstract Specification of Object Interaction
93-09	G.Snelting, B.Fischer, F.-J.Grosch, M.Kievernagel, A.Zeller	Die inferenzbasierte Softwareentwicklungsumgebung NORA
93-10	C.Lindig	STYLE – A Practical Type Checker for SCHEME
93-11	H.-D.Ehrich	Beiträge zu KORSO- und TROLL <i>light</i> -Fallstudien
94-01	A.Zeller	Configuration Management with Feature Logics
94-02	J.Schönwälder, H.Langendörfer	Netzwerkmanagement — Beschreibung des Exponats auf der CeBIT'94
94-03	T.Hartmann, G.Saake, R.Jungclaus, P.Hartel, J.Kusch	Revised Version of the Modelling Language TROLL (Version 2.0)
94-04	A.Zeller, G.Snelting	Incremental Configuration Management Based on Feature Unification
94-05	S.Conrad	A Basic Calculus for Verifying Properties of Synchronously Interacting Objects
94-06	M.Gogolla, N.Vlachantonis, R.Herzig, G.Denker, S.Conrad, H.-D.Ehrich	The KORSO Approach to the Development of Reliable Information Systems
94-07	C.Lindig	Inkrementelle, rückgekoppelte Suche in Software-Bibliotheken
94-08	B.Fischer, M.Kievernagel, W.Struckmann	VCR: A VDM-based software component retrieval tool
95-01	V.S.Cherniavsky	Philosophische Aspekte des Unvollständigkeitstheorems von Gödel
95-02	G.Snelting	Reengineering of Configurations Based on Mathematical Concept Analysis
95-03	A.Zeller	A Unified Configuration Management Model
95-04	H.Bickel, W.Struckmann	The Hoare Logic of Data Types
95-05	F.-J.Grosch	No Type Stamps and No Structure Stamps – a Referentially-Transparent Higher-Order Module Language