A Syntactic Approach to Structure Generativity

Franz-Josef Grosch

Informatik-Bericht Nr. 96-05
Juli 1996

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

A Syntactic Approach to Structure Generativity

Franz-Josef Grosch*

Abteilung Softwaretechnologie
Technische Universität Braunschweig, Germany

July 12, 1996

Abstract

Advanced module systems like Standard ML's [13, 17] support *structure generativity*. Structure generativity denotes the mechanism that parameterized modules (*functors*) generate a "new" module instance (*structure*) for every application to a suitable argument. This operational behaviour is essentially a side effect. Since interfaces in general depend on module instances, module instances are computationally characterised as "stamps" for the purpose of interface-checking.

This paper presents a typed module calculus that makes instances of modules syntactically apparent in expressions and interfaces. The module calculus has a simple rewriting semantics without side effects. Interface-checking is based on a type system with dependent functions, strong sums and additionally a non-standard variant of weak sums. A module system derived from the module calculus forms a separate layer above some typed core language. As a demonstrating example, we sketch M/SML, a module language on top of core SML.

Key words: module language, ML, type systems, dependent types

1 Introduction

Module systems provide support for factoring large software systems into separate but dependent program units. These program units, called *simple modules* (or just *modules* for short) in the following and variously known as clusters, packages, or structures, in their simplest form define collections of identifiers: types and values in a purely functional setting, additionally variables and exceptions for imperative features, and even classes for the object-oriented paradigm. The types of simple modules, called *simple interfaces* for convenience, are known as definition modules, package specifications, and signatures. They classify a simple module's exported identifiers with types for values and with kinds for type constructors. In general, simple interfaces as well as simple modules depend on preceding, so-called imported modules. That is, simple interfaces are essentially *dependent types*.

In order to build a large system, simple modules and simple interfaces have to be organized in a hierarchical, or more exactly, topological order with respect to the relationship "depends-on". This can be done rigidly, by defining modules one after another in a language like Modula-2 [24] or more flexible in a language like SML [17, 16], where dependencies are expressed via functor parameters and structures are linked by functor applications.

An essential feature of SML's module system is *structure generativity*. Structure generativity deals with the question when a structure is instantiated in order to be part of a system. This is no problem in a static module system like Modula-2's modules where declaration and instantiation

* This work is funded by the Deutsche Forschungsgemeinschaft, grant Sn11/4-1. Author's current address: Technische Universität Braunschweig, Abteilung Softwaretechnologie, Bültenweg 88, D-38106 Braunschweig/Germany. E-mail: grosch@ips.cs.tu-bs.de.

of a module coincide and an instance of a module is simply identified by the module's name. In SML, structures are instantiated as the result of functor applications, and structure generativity denotes the computational behaviour that a "new" instance of a structure is generated for every functor application. Essentially, this is a side effect at the module level.

Since signatures in general depend on structures (or more exactly on instances of structures) structure generativity is also important for the static semantics of SML's modules. In the static semantics instances are identified by stamps. This computational characterisation of SML's type-checker is standardized in [17] and has been extended to higher-order functors in [15].

In this work we want to treat module instances in a more abstract, type-theoretic framework. We abandon the side effect of module instantiation and develop a module calculus that has a simple rewriting semantics and makes instances of simple modules syntactically apparent in expressions. For the purpose of interface-checking, we develop a type system, based on dependent functions, strong sums and a non-standard variant of weak sums that also makes module instances apparent. In this module calculus a complete software system is simply an expression, and the expression's type represents its interface. In a sense, the module calculus is purely functional. Furthermore, we will sketch a module language on top of core SML that is derived from the module calculus.

To reach our goals, we reassess MacQueen's proposal for the module system of SML [13]. As we will see, this leads us to a module system similar in spirit but technically quite different to SML's modules. We adopt MacQueen's definition that a module, in its simplest form, is a named collection of declarations whose purpose is to define an environment. That means, a simple module is identified by a name, classified by a simple interface, and realized by an implementation.

The purpose of our module calculus is to describe *systems*. Declaring a simple module m with interface M and implementation i, for later use within some expression u with interface U is, roughly, a *dependent pair*:

$$(m : M = i, u\{m\}) : (m : M \times U\{m\}),$$

where m may occur free in expression $u\{m\}$ and interface $U\{m\}$. We prefer the notation:

$$\mathsf{dec}[m{:}M\ \mathsf{is}\ i]\ u\{m\} : \mathsf{some}[m{:}M]\ U\{m\}.$$

We call this binding construct a *system*. The expression u which uses the simple module m is also called the *export part*.

However, as MacQueen points out, the implementation as well as the interface of a simple module often contain free identifiers referring to other modules or to predefined declarations. Therefore, he argues, a module in general is a *dependent function* (called *functor* in SML) from an *environment of definition* to the *defined environment*:

$$\mathsf{fun}(x{:}A)\ m\{x\} : \mathsf{all}(x{:}A)\ M\{x\}.$$

There seems to be a contradiction between MacQueens explanation of a module as the result of a function application and our understanding, where a simple module is declared as a part of a system. A closer look shows that the first is the implementor's view (What is necessary to implement a module?), while the second is the system architect's view (How can I use a certain module in a system's architecture?). But both views do not exclude each other.

It is no problem to add an additional parameter module x with interface A to our notion of a system:

$$F = \mathsf{fun}(x{:}A)\ (\mathsf{dec}[m{:}M\{x\}\ \mathsf{is}\ i\{x\}]\ (u\{x\}\{m\})).$$

We get a parameterized system (a function), where the interface of module m, its implementation i, and the export part u may refer to the parameter module x. If F is applied to a matching module a, x is substituted by a. The result is a system which declares simple modules a and m and exports u.

$$\mathsf{dec}[a{:}A\ \mathsf{is}\ i']\ (Fa) \equiv \mathsf{dec}[a{:}A\ \mathsf{is}\ i']\ (\mathsf{dec}[m{:}M\{x \leftarrow a\}\ \mathsf{is}\ i\{x \leftarrow a\}]\ (u\{x \leftarrow a\}\{m\})).$$

The simplest system we can imagine declares one simple module and exports it.

$$\mathsf{dec}[a\!:\!A\ \mathsf{is}\ i_a]\ a$$

What seems rather useless becomes more meaningful if we imagine a system which declares two modules a and b and exports module b.

$$\mathsf{dec}[a\!:\!A\ \mathsf{is}\ i_a]\ (\mathsf{dec}[b\!:\!B\{a\}\ \mathsf{is}\ i_b\{a\}]\ b)$$

In general, it should be possible that a system exports more than one module. Therefore, we additionally introduce labelled tupels (which are essentially strong sums, see for example [7, 22]) in order to be able to group several simple modules of a system into one exported expression[1].

$$\mathsf{dec}[a\!:\!A\ \mathsf{is}\ i_a]\ (\mathsf{dec}[b\!:\!B\{a\}\ \mathsf{is}\ i_b\{a\}]\ \langle l_a = a, l_b = b\rangle$$

Now, both modules a and b are exported as a tuple where a is labelled with l_a and b is labelled with l_b.

So far, we have separated two concerns: instances of simple modules are made explicit by declarations, and the export part specifies which components of a system can be used from outside. The question arises, how to use such a system. First, we can imagine a (sub-)system as a component of a larger system. Since a system is simply an expression of a calculus with a rewriting semantics (see section 4 below) using a system or applying a parameterized system is equivalent to textually inserting the system itself. Second and more important, if a system (a client) depends on another system (the supplier), the supplier system has to be *opened* in order to enable access to the internal structure.

2 Using systems—module instances made explicit

Now, if we want to use a system partially or as a whole in another context, we somehow have to *open* or to *eliminate* the system. Since we regard a system as a dependent pair combined of a module declaration and an export part we have to use an appropriate elimination form. There are several syntactic formulations for eliminating dependent pairs [22, 7, 18]. We prefer the following syntactic form:

$$\mathsf{use}\ p\ \mathsf{as}\ [x]y\ \mathsf{in}\ e$$

This means, bind the first resp. the second part of system p to variables x resp. y in expression e. The standard computation rule for dependent pairs using our notation is as follows:

$$\mathsf{use}\ (\mathsf{dec}[a\!:\!A\ \mathsf{is}\ i_a]\ b)\ \mathsf{as}\ [x]y\ \mathsf{in}\ e \to e\{x \leftarrow i_a\}\{y \leftarrow b\{a \leftarrow i_a\}\}.$$

In words, i_a substitutes x in e, and since b depends on a, a is substituted by i_a before b substitutes y in e.

Because we are interested in making module instances explicit in interfaces, we could regard systems as standard *weak sums* (see for example, [22]). The typing rule for weak sums has the following form:

$$\frac{\Gamma \vdash p : \mathsf{some}[a\!:\!A]\ B \quad \Gamma, x\!:\!A, y\!:\!B \vdash e : C}{\Gamma \vdash \mathsf{use}\ p\ \mathsf{as}\ [x]y\ \mathsf{in}\ e : C} \quad x, y \notin C.$$

If we assume weak sums, every use of a system "generates" an instance of its declared module. This is very similar to Mitchell and Plotkin's type-theoretic explanation of abstract types [18]. But, for our purpose weak sums are lacking in several respects. Generating module instances is still implicitly connected to the reduction of use. Furthermore, dependencies from expression e to the used system p expressed via variables x and y must not be propagated into the interface C of

[1] In general, a labelled tuple contains arbitrary expressions

e. This leads to the well-known critique [6, 14, 4] that weak sums have to be eliminated globally for all their clients.

What we really need is an elimination form that makes instances explicit in expressions and allows for the propagation of dependencies through interfaces. Therefore, we propose the following non-standard computation rule for use:

$$\mathsf{use}\ (\mathsf{dec}[a\!:\!A\ \mathsf{is}\ i_a]\ b)\ \mathsf{as}\ [x]y\ \mathsf{in}\ e \to \mathsf{dec}[a\!:\!A\ \mathsf{is}\ i_a]\ (e\{x \leftarrow a\}\{y \leftarrow b\})\ .$$

Instead of the implementation i_a the module's name a substitutes x. At the same time, we (re)bind a outside, ensuring that a does not escape its scope. Consequently, the implementation i_a remains the same. What seems to be a "cheap trick" solves several problems at once.

First, instances are made explicit, because we can regard every dec-bound module name as an instance of the assigned implementation. If we need two instances of a simple module, we have to use a system two times, and we get two instances due to α-conversion. For example:

$$\begin{aligned} S &\equiv \mathsf{dec}[a\!:\!A\ \mathsf{is}\ i_a]\ b \\ \mathsf{use}\ S\ \mathsf{as}\ [x]y\ \mathsf{in}\ (\mathsf{use}\ S\ \mathsf{as}\ [u]v\ \mathsf{in}\ f\ x\ u) &\to \mathsf{dec}[a\!:\!A\ \mathsf{is}\ i_a]\ ((\mathsf{use}\ S\ \mathsf{as}\ [u]v\ \mathsf{in}\ f\ x\ u)\{x \leftarrow a\}\{y \leftarrow b\}) \\ &\to \mathsf{dec}[a\!:\!A\ \mathsf{is}\ i_a]\ (\mathsf{dec}[a'\!:\!A\ \mathsf{is}\ i_a]\ (f\ a\ u)\{u \leftarrow a'\}\{v \leftarrow b\}) \\ &\to \mathsf{dec}[a\!:\!A\ \mathsf{is}\ i_a]\ (\mathsf{dec}[a'\!:\!A\ \mathsf{is}\ i_a]\ (f\ a\ a')). \end{aligned}$$

Here, using system S two times, "generates" two instances of implementation i_a, named a and a'.

Second, we get an *system closure mechanism* for free. An arbitrary part of a system can only be used in its context. Imagine a system $S \equiv \mathsf{dec}[a\!:\!A\ \mathsf{is}\ i_a]\ \mathsf{dec}[b\!:\!B\ \mathsf{is}\ i_b]\ \mathsf{dec}[c\!:\!C\ \mathsf{is}\ i_c]\ d$; using b from system S automatically rebuilds the context, b depends on:

$$\mathsf{use}\ S\ \mathsf{as}\ [u]v\ \mathsf{in}\ (\mathsf{use}\ v\ \mathsf{as}\ [x]y\ \mathsf{in}\ f\ x) \twoheadrightarrow \mathsf{dec}[a\!:\!A\ \mathsf{is}\ i_a]\ (\mathsf{dec}[b\!:\!B\ \mathsf{is}\ i_b]\ f\ b)$$

At the same time, those parts of system S which are not needed are disregarded in the result.

If we look at the appropriate non-standard typing rule, this closure mechanism (the complete modular structure of the result) is also apparent in the resulting interface:

$$\frac{\Gamma \vdash p : \mathsf{some}[a\!:\!A]\ B \quad \Gamma, x\!:\!A, y\!:\!B \vdash c : C}{\Gamma \vdash \mathsf{use}\ p\ \mathsf{as}\ [x]y\ \mathsf{in}\ c : \mathsf{some}[a\!:\!A]\ (C\{x \leftarrow a\})}, y \notin C.$$

The result of using a system again is a system. The interface shows the modular structure an expression will have after a reduction is possible. Similar to weak sums, y must not occur in C, because it would otherwise escape its scope.

To sum up, the proposed elimination form makes instances of simple modules apparent in expressions and interfaces and supplies the relevant context when using a system partially or as a whole. It allows the propagation of dependencies through interfaces and eliminates the need to use a system globally for all its clients. Unfortunately, use cannot be called an elimination construct because the result of using a system again is a system. But, it does exactly what we need.

Together with function abstractions and labelled tuples, as mentioned in the introduction, we get a complete module calculus. But before we formally describe the module calculus and its typing system, we take a look at the connection of the core and the module language.

3 Embedding a core language

From the module calculus' point of view a simple module is an indivisible unit, an atom, identified by a name for later reference and specified by a simple interface hiding local declarations of the accompanying implementation. Simple interfaces and implementations are determined by some typed core language and connect the separated layers of module and core language.

For the purpose of demonstration we will use core SML [17] as our core language, consequently implementation modules are called *structures* and simple interfaces are called *signatures*. Signatures classify core language declarations. These are types, values, and exceptions.

Contrary to SML, signatures do not contain structure specifications. Furthermore, implementations should be separately compilable, and interface-checking module language expressions should not utilize implementation details. Therefore, we regard type specifications as abstract (opaque) and additionally introduce specifications of type synonyms (*manifest types* [9]). Similar to SML, we have a notion of *subtyping* between signatures that allows to define different views of a simple module. According to standard techniques, signatures may be thinned and type synonyms may be converted to abstract types. The paradigmatic signature of a module implementing a stack of integer values has the following form:

$$
\begin{array}{l}
\mathsf{interface}\ \mathit{INT\_STACK} = \mathsf{sig} \\
\qquad \mathsf{type}\ \mathit{item} = \mathit{int} \qquad (*\ \text{type synonym}\ *) \\
\qquad \mathsf{type}\ \mathit{stack} \qquad\quad (*\ \text{abstract type}\ *) \\
\qquad \mathsf{val}\ \mathit{push} : \mathit{item} \to \mathit{stack} \to \mathit{stack} \\
\qquad \mathsf{val}\ \mathit{pop} : \mathit{stack} \to \mathit{item} * \mathit{stack} \\
\qquad \ldots \\
\mathsf{end}
\end{array}
$$

If we now declare a module *stack* with interface *INT_STACK* for use in some module expression u, we write:

$$\mathsf{dec}[\mathit{Stack} : \mathit{INT\_STACK}\ \mathsf{is}\ \mathsf{struct}\ \ldots\ \mathsf{end}]\ u$$

The combination of module name, interface and implementation struct ... end is very similar to a *structure binding* in SML, apart from the fact that the visibility of the module name is restricted to the expression u. Signatures occurring inside u now can refer to types from *Stack* by the dot notation, writing *Stack.item* or *Stack.stack*. Structures (module implementations) scoped inside u can refer to types as well as to values and exceptions, e. g. *Stack.push*.

Technically, we guarantee that the identity of an abstract type declared inside a simple module is connected to the module's name, in other words, a *new* abstract type is generated for every instance of a simple module. Similar to *translucent sums* [6] or *manifest types* [9, 10], we use a *strengthening* operation (see section 4) to give identities to abstract types. In the example, the abstract type *stack* is manifestly equivalent to *Stack.stack* inside expression u.

We have seen that signatures and structures naturally depend on module names. Technically, the module calculus is a programming language with dependent types. Therefore, due to reductions, references to a module name may become references to an arbitrary module expression. As a consequence, interface-checking has to decide the equality of module language expressions, and reductions are also necessary inside structures. As we will discuss in the next section, the module calculus is *confluent* and *strongly normalising*, which makes structural equality of normal forms an adequate formulation for equality and guarantees a terminating reduction of module expressions.

The goal of reducing a module program is the production of an executable software system. Such a system coincides with our notion of a system. If a module language expression only consists of a sequence of module declarations together with an export expression, all simple modules involved in the complete system are in some topological order with respect to the relationship "depends-on". If all implementations of simple modules have been supplied, the dependencies indicate how to link these implementations in order to form an executable program.

4 The $\lambda\delta$-module calculus

Essentially, the foundation of our module calculus is a lambda calculus with types depending on values. Since, *system declaration* and *use* are the main novell features in our calculus we call it the *$\lambda\delta$-module calculus*[2]. The $\lambda\delta$-module calculus forms a separate layer above some typed core

[2] λ for abstractions and δ for declarations

Expressions			
	m	$::=$ x	variable/name
		$\mid$ l	label
		$\mid$ $\mathsf{fun}(x{:}I)\ m$	(function) abstraction
		$\mid$ $m_1\ m_2$	application
		$\mid$ $\langle l_1 = m_1, \ldots, l_n = m_n \rangle$	tuple construction
		$\mid$ $m\#l$	tuple selection
		$\mid$ $\mathsf{dec}[x{:}S]\ m$	(system) declaration
		$\mid$ $\mathsf{use}\ m_1\ \mathsf{as}\ [x_1]x_2\ \mathsf{in}\ m_2$	use
Interfaces			
	I	$::=$ S	simple interface
		$\mid$ $\mathsf{all}(x{:}I_1)\ I_2$	function interface
		$\mid$ $\langle l_1 : I_1, \ldots, l_1 : I_n \rangle$	tuple interface
		$\mid$ $\mathsf{some}[x{:}S]\ I$	system interface
Signatures			
	S	$::=$ $\mathsf{sig}\ E\ \mathsf{end}$	signature
	E	$::=$ D, E	signature entries
		$\mid$ ε	
	D	$::=$ $\mathsf{type}\ t$	opaque type specification
		$\mid$ $\mathsf{type}\ t = T$	manifest type specification
		$\mid$ $\mathsf{val}\ v : T$	value specification
	T	$::=$ t	type name
		$\mid$ $m.t$	type component
		$\mid$ $\mathit{int} \mid \mathit{bool}$	predefined simple types
		$\mid$ $T_1 \to T_2$	function type

Figure 1: Syntax of the $\lambda\delta$-module calculus

language. Therefore, we first take a look at syntax and reduction of the module calculus and then explain the connection to the core language.

4.1 Syntax and reduction

Figure 1 shows the syntax of the $\lambda\delta$-module calculus. There are two distinct identifier classes *variables/names* (x) and *labels* (l). Expressions are composed from variables/names, labels, *function abstraction* and *application*, *tuple construction* and *selection*, and *system declaration* and *use*. Abstraction, declaration, and use introduce variables/names. Tuples introduce labels. Labels are visible in subsequent tuple components and are used to select a tuple component from outside.

There are suitable interfaces for every kind of expression. *Function interfaces* are dependent functions. *Tuple interfaces* are a generalized form of strong sums. *System interfaces* are a non-standard variant of weak sums, as explained before. *Simple interfaces* play the role of simple types, which characterize simple modules, the simple values of the calculus. The interface of a module name introduced in a system declaration is syntactically restricted to a simple interface. This reflects the fact that system declarations only introduce simple modules. Simple modules and their interfaces are determined by the embedded core language. Since we assume an ML-like core language, simple interfaces are called *signatures*.

Signatures allow for the specification of abstract types (*opaque type specifiation*), type synonyms

Reduction for the $\lambda\delta$-module calculus

$$
\begin{array}{lcl}
(\mathsf{fun}(x\!:\!A)\ m_1)\ m_2 & \rightarrow & m_1\{x \leftarrow m_2\} \\
\langle l_1 = m_1, \ldots, l_k = m_k, \ldots, l_n = m_n\rangle \# l_k & \rightarrow & m_k\{l_1 \leftarrow m_1\} \cdots \{l_{k-1} \leftarrow m_{k-1}\} \\
\mathsf{use}\ (\mathsf{dec}[x_1\!:\!S]\ m_1)\ \mathsf{as}\ [x_2]x_3\ \mathsf{in}\ m_2 & \rightarrow & \mathsf{dec}[x_1\!:\!S]\ (m_2\{x_2 \leftarrow x_1\}\{x_3 \leftarrow m_1\})
\end{array}
$$

Figure 2: Reduction rules

(*manifest type specification*), and core language values (*value specification*). Essentially, type names (t) and values (v) are two further identifier classes distinct from variables/names and labels. These identifiers can only be used inside signatures. Type expressions (T) are composed from predefined *simple types*, *type names*, *type components* and *function types*. Type components from "imported" modules are referenced via the dot notation; a module expressions (m) qualifies the module and a type name (t) selects the component. Due to type components, interfaces are value-dependent types.

Although we have signatures, structure construction is not part of the module calculus. We regard a *structure* as the implementation of a simple module which is part of the core language. In the module calculus itself, a module is either declared by a name in a system declaration or bound via a variable as a function parameter or in the use construct. The module's signature completely classifies the structure. This makes module expressions independent from representation details and allows for separate compilation of structures.

The reduction rules for the $\lambda\delta$-module calculus are shown in figure 2. Substitution ($\{x \leftarrow m\}$) is defined as usual. For simplicity, α-conversion, which has to be considered, is not made explicit in the reduction rules, and compatibility rules for selecting redexes are omitted. Since simple modules are indivisible units, there is no reduction rule for the dot notation—the dot notation is simply a qualified identifier. Note that substitutions and reductions also occur inside interfaces, because interfaces depend on module expressions.

4.2 The core language

The core language is connected to the module calculus via module implementations. For every *declaration* of a simple module a *structure* may (but need not) be supplied as its *implementation*. The simple module's signature and the implementation are in the same scope. The additional syntax which connects structures to the module calculus is shown in figure 3. To go with signatures, structures contain type and value declarations.

The core language distinguishes declarations of type synonyms (*non-generative type binding*) and of new type constructors (*generative type creation*). For simplicity, the declaration of constructor functions is not part of the syntax, and new type constructors are always nullary. The addition of constructor functions, n-ary type constructors and exception declarations is straightforward.

Similar to signatures, structures also may depend on module expressions, when values and types from "imported" modules are used via the dot notation. As a consequence, the reduction of module expressions also leads to substitutions inside structures.

If all declarations of modules have been implemented by structures, and if the reduced expression does not contain any abstractions, module expressions inside implementations can be reduced to module names, indicating how the separately compiled structures have to be linked. The result is a sequence of structures, topologically ordered according to the relationship "depends-on". The export part indicates the "main module" of the whole system.

4.3 Interface inference

Interface-checking module expressions is a delicate issue, because interfaces may depend on arbitrary module expressions. The complete typing rules are given in the appendix. Here we only

Structure implementations:

m	::=	...	
		$\mathsf{dec}[x\!:\!S\,\mathsf{is}\,s]\,m$	declaration and implementation
		...	
s	::=	$\mathsf{struct}\ e\ \mathsf{end}$	structure
e	::=	d, e	structure entries
		ε	
d	::=	$\mathsf{type}\ t = T$	type binding (non-generative)
		$\mathsf{datatype}\ t$	type creation (generative)
		$\mathsf{val}\ v : t = c$	value declaration
c	::=	$m.v$	(value) component
		...	core language expressions

Figure 3: Syntax for the core language

discuss the essential and the most unusual rules shown in figure 4.

The rules are written down in a non-deterministic manner, in order to show the principal inference steps. All derivations start from an *axiom*. There are three axioms: an empty tuple interface, an empty signature and the kind type are wellformed. The symbol $\Box$ denotes wellformedness and symbol [ ] denotes an empty *basis*. There is a *start* and a *weakening* rule for variables/names, labels, opaque types, type synonyms and values[3]. *Interface formation* rules for function, system, and tuple interfaces, and rules for *signature formation* and *type formation* allow to derive wellformedness of complex and simple interfaces.

Function interfaces are dependent functions introduced by *abstractions*. Tuple interfaces are a generalized form of strong sums, which are introduced by *tuple construction*. System interfaces are introduced by system *declarations* and may be regarded as a non-standard variant of weak sums. Note that the syntactic restriction that names introduced by declarations must have simple interfaces is reflected by the declaration rule and by the formation rule for system interfaces. The elimination rules for function interfaces (*application*) and tuple interfaces (*tuple selection*) are standard. Rather unusual is the rule *use*.

The syntactic construct use matches an expression m_1 that has a system interface to variables x_2 and x_3 for use in expression m_2. x_2 binds a simple module and, therefore, has a simple interface. x_3 binds the export part of the used system. The resulting interface again is a system interface, where the declared name is rebound by x_1 in the resulting system interface. That means, I_2 may very well contain x_2 but must not contain x_3, because x_2 is substituted by x_1 while x_3 would escape its scope. Figuratively, the typing rule *use* reflects the fact that the module declared in the used system is always part of the using expression, ensuring the system closure mechanism mentioned in section 2.

Type specifications in signatures may refer to internally declared type names. Therefore, the *module* rules extend the basis ensuring that type names do not escape their scope when typing type components and value components referenced via the dot notation.

The *conversion* rule allows to convert interfaces. It integrates several important conversion operations. First of all, our type system supports the idea that abstract types are connected to the instance of the module they are declared in. This approach is very similar to the approaches of Leroy [9, 10] or Harper and Lillebridge [6]. Similar to Leroy's system, we use a *strengthening* operation

[3] *stmt* denotes an arbitrary type statement

(axiom) $[\,] \vdash \langle\,\rangle : \Box \qquad [\,] \vdash \mathsf{sig\ \ end} : \Box \qquad [\,] \vdash \mathsf{type} : \Box$

(start) $$\frac{\Gamma \vdash I : \Box}{\Gamma, x{:}I \vdash x : I}, x \notin \Gamma$$ other start rules omitted

(weakening) $$\frac{\Gamma \vdash \mathit{stmt} \quad \Gamma \vdash I : \Box}{\Gamma, x{:}I \vdash \mathit{stmt}}, x \notin \Gamma$$ other weakening rules omitted

(interface formation) $$\frac{\Gamma \vdash S : \Box \quad \Gamma, x{:}S \vdash I : \Box}{\Gamma \vdash \mathsf{some}[x{:}S]\ I : \Box}$$ other interface formation rules omitted

(signature formation) omitted

(type formation) omitted

(module) $$\frac{\Gamma \vdash m : \mathsf{sig\ type}\ t, E\ \mathsf{end}}{\Gamma, \mathsf{type}\ t \vdash m : \mathsf{sig}\ E\ \mathsf{end}}$$ other module rules omitted

(abstraction) $$\frac{\Gamma, x{:}I_1 \vdash m : I_2 \quad \Gamma \vdash \mathsf{all}(x{:}I_1)\ I_2 : \Box}{\Gamma \vdash \mathsf{fun}(x{:}I_1)\ m : \mathsf{all}(x{:}I_1)\ I_2}$$

(application) $$\frac{\Gamma \vdash m_1 : \mathsf{all}(x{:}I_2)\ I_1 \quad \Gamma \vdash m_2 : I_2}{\Gamma \vdash m_1\ m_2 : I_1\{x \leftarrow m_2\}}$$

(tuple construction) $$\frac{\Gamma \vdash \langle l_2 = m_2, \ldots, l_n = m_n\rangle\{l_1 \leftarrow m_1\} : \langle l_2{:}I_2, \ldots, l_n : I_n\rangle\{l_1 \leftarrow m_1\} \quad \Gamma \vdash m_1 : I_1 \quad \Gamma \vdash \langle l_1{:}I_1, \ldots, l_n{:}I_n\rangle : \Box}{\Gamma \vdash \langle l_1 = m_1, \ldots l_n = m_n\rangle : \langle l_1{:}I_1, \ldots, l_n{:}I_n\rangle}$$

(tuple selection) $$\frac{\Gamma \vdash m : \langle l_1{:}I_1, \ldots, l_k{:}I_k, \ldots, l_n{:}I_n\rangle}{\Gamma \vdash m\#l_k : I_k\{l_1 \leftarrow m\#l_1\}\cdots\{l_{k-1} \leftarrow m\#l_{k-1}\}}$$

(declaration) $$\frac{\Gamma, x{:}I_1 \vdash m : I \quad \Gamma \vdash \mathsf{some}[x{:}S]\ I : \Box}{\Gamma \vdash \mathsf{dec}[x{:}S]\ m : \mathsf{some}[x{:}S]\ I}$$

(use) $$\frac{\Gamma \vdash m_1 : \mathsf{some}[x_1{:}S]\ I_1 \quad \Gamma, x_2{:}S, x_3{:}I_1 \vdash m_2 : I_2}{\Gamma \vdash \mathsf{use}\ m_1\ \mathsf{as}\ [x]y\ \mathsf{in}\ m_2 : \mathsf{some}[x_1{:}S]\ (I_2\{x_2 \leftarrow x_1\})}, x_3 \notin I_2$$

(conversion) $$\frac{\Gamma \vdash m : I_1 \quad \Gamma \vdash I_2 : \Box \quad I_1/m <:_\Gamma I_2}{\Gamma \vdash m : I_2}$$

Figure 4: Interface inference rules

(figure 5) to represent the fact that abstract type components in signatures are not arbitrary types, but come from the module expression they are connected to. Different to Leroy's approach we do not restrict strengthening to *path expressions*, but allow arbitrary module expressions.

Beside strengthening, our module calculus also supports a *subtyping* relation (see figure 5) between interfaces in order to allow for different views of a simple module. Subtyping between interfaces ($<:_\Gamma$) is defined as usual: signatures may be thinned, manifest type specifications are considered subtypes of opaque type specifications, function interfaces are contra-variant, tuple and system interfaces are covariant.

Strengthening

$$
\begin{aligned}
(\mathsf{sig}\ E\ \mathsf{end})/m &= \mathsf{sig}\ E/m\ \mathsf{end} \\
\varepsilon/m &= \varepsilon \\
(\mathsf{type}\ t, E)/m &= \mathsf{type}\ t = m.t, E/m \\
(\mathsf{type}\ t = T, E)/m &= \mathsf{type}\ t = T, E/m \\
(\mathsf{val}\ v : T, E)/m &= \mathsf{val}\ v : T, E/m \\
I/m &= I
\end{aligned}
$$

Subtyping of interfaces ($<:_\Gamma$)

$$
\begin{array}{lcl}
\mathsf{all}(x{:}I_1)\ I_2 <:_\Gamma \mathsf{all}(x{:}I_1')\ I_2' & \Leftarrow & I_1' <:_\Gamma I_1,\ I_2 <:_{\Gamma'} I_2' \\
 & & \quad \text{where } \Gamma' = \Gamma, x{:}I_1' \\
\langle\,\rangle <:_\Gamma \langle\,\rangle & & \\
\langle l_1{:}I_1, \ldots, l_n{:}I_n\rangle <:_\Gamma \langle l_1{:}I_1', \ldots, l_n{:}I_n'\rangle & \Leftarrow & I_1 <:_\Gamma I_1', \\
 & & \quad \langle l_2{:}I_2, \ldots, l_n{:}I_n\rangle <:_{\Gamma'} \langle l_2{:}I_2', \ldots, l_n{:}I_n'\rangle \\
 & & \quad \text{where } \Gamma' = \Gamma, l_1{:}I_1 \\
\mathsf{some}[a{:}S]\ I <:_\Gamma \mathsf{some}[a{:}S']\ I' & \Leftarrow & S <:_\Gamma S',\ I <:_{\Gamma'} I' \\
 & & \quad \text{where } \Gamma' = \Gamma, a{:}S \\
\mathsf{sig}\ E\ \mathsf{end} <:_\Gamma \mathsf{sig}\quad \mathsf{end}; & & \\
\mathsf{sig}\ D_1, \ldots, D_n\ \mathsf{end} <:_\Gamma \mathsf{sig}\ D_1', \ldots, D_m'\ \mathsf{end} & \Leftarrow & D_{\sigma(i)} <:_{\Gamma'} D_i' \text{ for } i \in \{1, \ldots, m\} \\
 & & \quad \text{where } \Gamma' = \Gamma, D_1, \ldots, D_n \\
 & & \quad \text{and}\quad \sigma : \{1, \ldots, m\} \to \{1, \ldots, n\} \\
\mathsf{type}\ t <:_\Gamma \mathsf{type}\ t & & \\
\mathsf{type}\ t = T <:_\Gamma \mathsf{type}\ t & & \\
\mathsf{val}\ v : T_1 <:_\Gamma \mathsf{val}\ v : T_2 & \Leftarrow & T_1 \approx_\Gamma T_2 \\
\mathsf{type}\ t = T_1 <:_\Gamma \mathsf{type}\ t = T_2 & \Leftarrow & T_1 \approx_\Gamma T_2
\end{array}
$$

Manifest equality of types ($\approx_\Gamma$) (reflexivity and transitivity omitted)

$$
\begin{array}{lcl}
T_1 \to T_2 \approx_\Gamma T_1' \to T_2' & \Leftarrow & T_1 \approx_\Gamma T_1',\quad T_2 \approx_\Gamma T_2' \\
t \approx_\Gamma T & \Leftarrow & \Gamma \vdash t = T : \mathsf{type} \\
m.t \approx_\Gamma T & \Leftarrow & \Gamma \vdash m : \mathsf{sig}\ \mathsf{type}\ t = T, E\ \mathsf{end} \\
t \approx_\Gamma t & & \\
m_1.t \approx_\Gamma m_2.t & \Leftarrow & m_1 = m_2
\end{array}
$$

Syntactic equality of expressions modulo reduction ($=$)
(simple syntactic equality of expressions ($\equiv$) and reduction ($\twoheadrightarrow$) omitted)

$$
m_1 = m_2 \quad \Leftarrow \quad m_1' \equiv m_2', \quad m_1 \twoheadrightarrow m_1', \quad m_2 \twoheadrightarrow m_2'
$$

Figure 5: Conversion

Due to dependencies, the subtyping relation has to consider *manifest equality* of types ($\approx_\Gamma$). Subtyping and manifest equality need basis Γ as an additional parameter because manifest type specifications which determine manifest type equality are recorded in the basis. Two type components are manifestly equal if the qualifying module expressions are syntactically equal ($=$) modulo reduction ($\twoheadrightarrow$).

Syntactic equality ($\equiv$) is the last step in converting interfaces. We have chosen a simplified form of syntactic equality which is obviously decidable. For simplicity, we regard any two expressions containing interface annotations as different. For practical purposes this form of equality seems

to be sufficient. We are not sure whether syntactic equality is decidable if we take interface annotations into account.

Without proof, we state the following proposition:

Proposition 1 *Reduction in the $\lambda\delta$-module calculus is confluent and strongly normalising.*

These properties guarantee a terminating reduction and make syntactic equality of expressions modulo reduction a meaningful relation.

Type-checking structures is fairly standard. The rules are given in the appendix. Basis Γ is the same for type-checking a simple module's implementation and the system declaration the module is declared in. Since type-checking structures and interface-checking module expressions use the same basis, there are also start and weakening rules for value specifications.

5 M/SML—a module language for core SML

As an example for a module language based on the $\lambda\delta$-module calculus, we are implementing M/SML, a module language for core SML. For several reason we have chosen the core of SML as our core language. Core SML has a rich type system that is formally defined [17], SML itself has a powerful module system, and variants of SML's module system are the subject of ongoing research (see section 6). In this section we will sketch additional features of M/SML necessary for a module language which are not covered by the $\lambda\delta$-module calculus.

First of all, interface declarations are an essential feature of a module language. Interface declarations define a name for a signature or an arbitrary complex interface. Without any restriction, function, tuple, and system interfaces may be declared. The example shows the well-known interface characterizing a module that represents an order.

```
interface ORD = sig
                  type t
                  val less: t -> t -> bool
                end
```

MacQueen's *signature closure rule* [13] claims that a signature should contain the structures it depends on as substructures. In SML, the names of these substructures are *free* and are instantiated with every use of the signature. M/SML does not allow for substructures. In order to express the dependency of an interface from certain modules, we use parameterized interfaces. Parameterized interfaces are functions from modules to interfaces which can be "instantiated" at different uses by application to a matching module.

The interface `DICT` for a dictionary depends on a module that realizes keys and can be characterized as an order.

```
interface DICT (Key: ORD) = sig
                              type key = Key.t
                              type 'a dict
                              val empty: dict
                              val add: key -> 'a -> 'a dict -> 'a dict
                            end
```

An SML signature that serves the same purpose has the following form:

```
signature DICT = sig
                   structure Key: ORD
                   type key = Key.t
                   ... (* rest as above *)
                 end
```

The main difference is that the SML signature is instantiated implicitly, while parameterized interfaces in M/SML have to be applied explicitly.

Since system declarations are the most unusual part of M/SML, we will start with the most simple system.

```
IntOrder = dec [IntOrd: INT_ORD] IntOrd
```

The system `IntOrder` consists of a single simple module `IntOrd` that is also exported. The interface `INT_ORD` is a subinterface of `ORD` and can be characterized by the following interface declaration.

```
interface INT_ORD = ORD with sig type t = int end
```

`MakeDict` represents a dictionary that is parameterized by a key. Note that the interface of `Dict` is characterized by applying parameterized interface `DICT` to module `Key`. `Key`—or more exactly what `Key` will be, when `MakeDict` has been applied—and `Dict` are exported as a tuple.

```
MakeDict = fun (Key: ORD)
           dec [Dict: DICT Key]
             <Key = Key, Dictionary = Dict>
```

In order to build a dictionary with integer keys we `use` system `IntOrder` and apply `MakeDict`. Since we use `IntOrder` via the export part the declaration is matched with a wild card (`_`).

```
IntKeyDict = use IntOrder as [_] Key in
               MakeDict Key
```

Usually, the use of a system happens only via the export part, therefore, we could abbreviate the `as`-part of `use`.

```
IntKeyDict = use IntOrder as Key in
               MakeDict Key
```

The resulting system `IntKeyDict` can be characterized by interface `INT_KEY_DICT`.

```
interface INT_KEY_DICT = some [IntKey: INT_ORD]
                           some [IntKeyDict: DICT IntKey]
                             <Key: INT_ORD, Dictionary: DICT IntKey>
```

If we want to add an extension module that defines additional operations on dictionaries, e.g. a domain function that returns a set of keys, we need a set module and the extension itself. These modules are characterized by parameterized interfaces `SET` and `EXT`.

```
interface SET (Elt: ORD) = sig
                             type elt = Elt.t
                             type set
                             val empty: set
                             val member: elt -> set -> bool
                             ...
                           end

interface EXT (Key:ORD) (KeySet: SET Key) =
                      sig
                        val domain: 'a dict -> KeySet.set
                      end
```

The set as well as the extension can be realized very generally, in order to allow for reuse in different contexts.

```
MakeSet    = fun (Elt: ORD)
             dec [Set: SET Elt]
               Set
ExtendDict = fun (Elt:ORD)
             fun (EltDict: DICT Elt)
             fun (EltSet: SET Elt)
             dec [Extend: EXT Elt EltSet]
               <Key        = Elt,
                Keyset     = EltSet,
                Dictionary = EltDict,
                Extensions = ExtendDict>
```

If we now want to build an extension of the integer dictionary above, we have to `use` system `IntKeyDict`, combine it with a `KeySet`, and build an appropriate extension. Normally, a system consists of several declarations. Therefore, `use` allows to match such a "curried" system in one step.

```
IntKeyExtDict = use IntKeyDict          as [_][_] Dict    in
                use MakeSet (Dict#Key)  as [_]    KeySet  in
                  ExtendDict (Dict#Key) (Dict#Dictionary) IntSet
```

We can abbreviate matching all leading declarations, and accessing the "last" export just the same way as matching one declaration and accessing the export part.

```
IntKeyExtDict = use IntKeyDict          as Dict    in
                use MakeSet (Dict#Key)  as KeySet  in
                  ExtendDict (Dict#Key) (Dict#Dictionary) KeySet
```

If we additionally introduce pattern matching for tuples, `IntKeyExtDict` can be written down very elegantly:

```
IntKeyExtDict = use IntKeyDict   as <Key, Dictionary>  in
                use MakeSet Key  as Keyset             in
                  ExtendDict Key Dictionary Keyset           .
```

Until now, no implementations of simple modules have been supplied. Preferably, every declaration defines a unique name global to the top-level expression it occurs in. Then, the accompanying implementation can be separated, provided top-level declarations are also unique. The implementations needed above can be added separately.

```
implementation IntOrd  of IntOrder    = struct ... end
implementation Dict    of MakeDict    = struct ... end
implementation Set     of MakeSet     = struct ... end
implementation Extend  of ExtendDict  = struct ... end
```

The environment for the implementation remains the same as the environment for the declaration. Separate compilation of implementations is possible and reorganization of module programs is rather easy.

If all implementations have been supplied, a system like `IntKeyExtDict` can be compiled into an executable program. Currently we compile implementations into SML structures which maintain reduction of module expressions via import lists. Several instances of an implementation are simply represented as duplicated code.

6 Related work

The central aim of the $\lambda\delta$-module calculus is to integrate the side effect of generating module instances into a calculus with a simple rewriting semantics. The idea to use names for representing

instances is inspired by calculi which integrate side effects into functional languages, especially Odersky's $\lambda\nu$-calculus [19]. While $\lambda\nu$ is essentially a calculus of local names, $\lambda\delta$ may be characterized as a calculus of global names.

The $\lambda\delta$-module calculus is a predicative calculus, where types depend on values. A non-standard variant of weak sums is used to explain module generativity. Similar in spirit, Mitchell and Plotkin [18] use the SOL-calculus to give a type-theoretic account of abstract types. A type and its operations may be packaged into an abstract type—essentially a weak sum. An abstract type has to be opened, thereby generating a "new" type before it can be used. Although the SOL-calculus is impredicative, with values depending on types, using a system in the $\lambda\delta$-module calculus is similar to combining open and pack in SOL, with systems instead of abstract types.

MacQueen's original explanation of SML's module system using strong sums [14] has been further developed in the XSML-calculus [7]. The XSML-calculus is a predicative calculus with strong sums that is used to give a type-theoretic explanation of Standard ML regarding, both, the core and the module language. Structure generativity is not explained. Since structures may occur in interfaces, type-checking XSML is undecidable, because of the need for checking the equality of arbitrary core language expressions. In [8] Harper et. al. show how to introduce a phase distinction that makes type-checking decidable. The $\lambda\delta$-module calculus also embodies strong sums, but they are only used to form labelled tuples of module expressions. These tuples do not contain any core language expressions. Module instances can only occur in tuples, if they have been bound before in a declaration.

Type generativity and abstract types are not explained by the XSML-calculus. In order to prevent the need for weak sums, which have to be opened globally and to allow access to abstract types via the dot notation [4], translucent sums [6] or manifest types [9, 12] have been proposed. Both concepts identify abstract types with the module expression they come from. This leads to a more abstract and less operational treatment of type generativity than in the definition of SML [17]. The identity of abstract types (generative type declarations) is intimately connected to the generation of structure instances. While the operational approach of SML generates "new" types for every structure instance, in the manifest-types approach abstract types are compatible if they come from the same *path expression*. This can be a doubtful feature if two module instances with compatible abstract types belong to one software system and we want to restructure the system by substituting one instance by another implementation. This clearly hurts the principle of information hiding. The $\lambda\delta$-module calculus also uses manifest types to give identities to abstract types, but abstract types are new in every module instance, since the instances are explicit. Furthermore, structure construction is not part of the calculus and, therefore, manifest type expressions are not restricted to path expressions.

All the theoretical work above is used to develop module systems of the ML-family. Manifest types constitute the foundation of the module system of Objective Caml (formerly known as Caml Special Light [11]). SML-1996 [21] is a proposal for a language based on XSML and translucent sums. Besides theoretical foundations and the handling of type generativity, the integration of higher-order functors has been a major goal. All approaches scale up to higher-order functors, the more abstract approaches [10, 8] as well as the more operational approach [23, 15]. The $\lambda\delta$-module calculus also allows higher-order expressions, but it is not so closely connected to the ML-family.

7 Summary and future work

The $\lambda\delta$-module calculus introduces a syntactic way to handle the identity of module instances. System declaration and the dual use-construct allow to deal with module instances in a calculus with a simple rewriting semantics instead of using an operational semantics which describes structure generation as a side effect at the module level.

The type system describes interfaces of expressions using types (interfaces) that depend on values (module expressions). The most simple interfaces are signatures, the interfaces of simple modules. A non-standard variant of weak sums, called system interfaces, makes module instances

explicit in interfaces. Together with dependent functions, characterizing parameterized systems, strong sums for tupling arbitrary module expressions, and manifest types for expressing dependencies, we get a rich type system appropriate for a module language.

Essentially, the $\lambda\delta$-module calculus is a predicative calculus. Simple modules are its simple values, and signatures may be regarded as "simple" types. In a sense, the calculus is completely functional. The notion of a software system coincides with our notion of a system. It is simply a closed expression of the module language. Consequently, the interface reveals the complete modular structure an expression will have when all parameters have been supplied and the expression is reduced to normal form. Equational reasoning at the module level is possible, for example, to show that two different systems are equivalent[4]. Simple modules (and arbitrary module expression) that interfaces and implementations depend on are always represented syntactically in the module program. Therefore, it is not possible to declare *defective signatures* [1] (also called "monsters" in [16]) which can not be implemented by any structure.

The module language M/SML is just a testbed for the $\lambda\delta$-module calculus. Currently we are developing a concept-proof implementation of M/SML using parts of the ML-Kit [3] for the static analysis of module implementations. Since we are mainly interested in the module language, module language expressions which form an executable system are compiled into a sequence of SML structures.

Our original motivation was to support the description and development of software architectures [5], using a module language similar to the module language of SML. In principle, SML or one of its variants could serve this task [2], but in SML the elaboration of module expressions is only possible if all structures have been implemented. Architectional and implementational concerns cannot be separated. In M/SML a software achitecture can be designed and module expressions can be elaborated without supplying implementations for every dec-bound module.

The strict separation of module and implementation languages opens the door for several interesting applications of the $\lambda\delta$-module calculus. For example, we could allow a restricted form of mutual dependencies between module implementations, similar to Modula-2, where two or more implementation modules may mutually depend on each other's definition module.

```
system mutual = dec [a : A and b : B a]
                <a = a, b = b>
implementation a of mutual = struct ... b.f ... end
implementation b of mutual = struct ... a.f ... end
```

Furthermore, we could apply the $\lambda\delta$-module calculus to the problem of namespace management for an object-oriented language. In a strongly-typed object-oriented language, a class could play the role of a simple module, and it seems possible to develop a powerful module system for an object-oriented core language.

References

[1] Maria Virginia Aponte. Extending record typing to type parametric modules with sharing. In *20th Symposium on Principles of Programming Languages*, pages 465–478. ACM Press, January 1993.

[2] Edoardo Biagioni, Robert Harper, and Peter Lee. Implementing software architectures in Standard ML. In *ICSE-17 Workshop on Research Issues in the Intersection of Software Engineering and Programming Languages*, 1994. Position Paper.

[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]The topological order according to the depends-on relation between simple modules is not unique

[4] 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.

[5] 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.

[6] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In POPL-21 [20], pages 123–137.

[7] 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.

[8] 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.

[9] Xavier Leroy. Manifest types, modules and separate compilation. In POPL-21 [20], pages 109–122.

[10] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *22th Symposium on Principles of Programming Languages*, pages 142–153. ACM Press, January 1995.

[11] Xavier Leroy. Le système Caml Special Light: modules et compilation efficace en Caml. Technical Report 2721, INRIA, November 1995. In french.

[12] Xavier Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 1996. To appear.

[13] David MacQueen. Modules for Standard ML. In *Proc. 1984 ACM Conference on LISP and Functional Programming*, pages 198–207, New York, 1984. ACM Press.

[14] David MacQueen. Using dependent types to express modular structure. In *13th Symposium on Principles of Programming Languages*, pages 277–286. ACM Press, January 1986.

[15] 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.

[16] Robin Milner and Mads Tofte. *Commentary on Standard ML.* The MIT Press, Cambridge, Massachusetts, 1991.

[17] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* The MIT Press, Cambridge, Massachusetts, 1990.

[18] 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.

[19] Martin Odersky. A functional theory of local names. In POPL-21 [20], pages 48–59.

[20] *Conference Record of POPL '94: 21th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages.* ACM Press, January 1994.

[21] Chris Stone and Robert Harper. A type-theoretic account of Standard ML 1996. Technical Report CMU-CS-96-136, School of Computer Science, Carnegie Mellon University, May 1996.

[22] Simon Thompson. *Type Theory and Functional Programming.* Addison-Wesley, 1991.

[23] Mads Tofte. Principal signatures for higher-order program modules. In *19th Symposium on Principles of Programming Languages*, pages 189–199. ACM Press, January 1992.

[24] Niklaus Wirth. *Programming in Modula-2.* Springer-Verlag, 1988.

A Appendix

A.1 Syntax

A.1.1 Module calculus

Expressions

m	::=	x	variable/name
	|	l	label
	|	$\mathsf{fun}(x{:}I)\ m$	(function) abstraction
	|	$m_1\ m_2$	application
	|	$\langle l_1 = m_1, \ldots, l_n = m_n \rangle$	tuple construction
	|	$m\#l$	tuple selection
	|	$\mathsf{dec}[x{:}S]\ m$	(system) declaration
	|	$\mathsf{use}\ m_1\ \mathsf{as}\ [x_1]x_2\ \mathsf{in}\ m_2$	use

Interfaces

I	::=	S	simple interface
	|	$\mathsf{all}(x{:}I_1)\ I_2$	function interface
	|	$\langle l_1{:}I_1, \ldots, l_1{:}I_n \rangle$	tuple interface
	|	$\mathsf{some}[x{:}S]\ I$	system interface

Signatures S	::=	$\mathsf{sig}\ E\ \mathsf{end}$	signature
E	::=	D, E	signature entries
	|	ε	
D	::=	$\mathsf{type}\ t$	opaque type specification
	|	$\mathsf{type}\ t = T$	manifest type specification
	|	$\mathsf{val}\ v : T$	value specification
T	::=	t	type name
	|	$m.t$	type component
	|	$int \mid bool$	predefined simple types
	|	$T_1 \to T_2$	function type

A.1.2 Core language

Implementations

m	::=	...	
	|	$\mathsf{dec}[x{:}S\ \mathsf{is}\ s]\ m$	declaration and implementation
	|	...	
s	::=	$\mathsf{struct}\ e\ \mathsf{end}$	structure
e	::=	d, e	structure entries
	|	ε	
d	::=	$\mathsf{type}\ t = T$	type binding (non-generative)
	|	$\mathsf{datatype}\ t$	type creation (generative)
	|	$\mathsf{val}\ v : t = c$	value declaration
c	::=	$m.v$	(value) component
	|	...	core language expressions

A.2 Typing rules

A.2.1 Module calculus

(axiom) $$[\,] \vdash \langle\,\rangle : \Box \qquad [\,] \vdash \mathsf{sig\ \ end} : \Box \qquad [\,] \vdash \mathsf{type} : \Box$$

(start) $$\frac{\Gamma \vdash I : \Box}{\Gamma, x{:}I \vdash x : I}, x \notin \Gamma \qquad \frac{\Gamma \vdash I : \Box}{\Gamma, l{:}I \vdash l : I}, l \notin \Gamma$$

$$\frac{\Gamma \vdash \mathsf{type} : \Box}{\Gamma, \mathsf{type}\ t \vdash t : \mathsf{type}}, t \notin \Gamma \qquad \frac{\Gamma \vdash T : \mathsf{type} \quad \Gamma \vdash \mathsf{type} : \Box}{\Gamma, \mathsf{type}\ t = T \vdash t = T : \mathsf{type}}, t \notin \Gamma$$

$$\frac{\Gamma \vdash T : \mathsf{type}}{\Gamma, \mathsf{val}\ v : T \vdash \mathsf{val}\ v : T}, v \notin \Gamma$$

(weakening)

$$\frac{\Gamma \vdash \mathit{stmt} \quad \Gamma \vdash I : \Box}{\Gamma, x{:}I \vdash \mathit{stmt}}, x \notin \Gamma \qquad \frac{\Gamma \vdash \mathit{stmt} \quad \Gamma \vdash I : \Box}{\Gamma, l{:}I \vdash \mathit{stmt}}, l \notin \Gamma$$

$$\frac{\Gamma \vdash \mathit{stmt} \quad \Gamma \vdash \mathsf{type} : \Box}{\Gamma, \mathsf{type}\ t \vdash \mathit{stmt}}, t \notin \Gamma \qquad \frac{\Gamma \vdash \mathit{stmt} \quad \Gamma \vdash T : \mathsf{type} \quad \Gamma \vdash \mathsf{type} : \Box}{\Gamma, \mathsf{type}\ t = T \vdash \mathit{stmt}}, t \notin \Gamma$$

$$\frac{\Gamma \vdash \mathit{stmt} \quad \Gamma \vdash T : \mathsf{type}}{\Gamma, \mathsf{val}\ v : T \vdash \mathit{stmt}}, v \notin \Gamma \qquad \textit{stmt} \text{ denotes an arbitrary typing statement}$$

(interface formation)

$$\frac{\Gamma \vdash I_1 : \Box \quad \Gamma, x{:}I_1 \vdash I_2 : \Box}{\Gamma \vdash \mathsf{all}(x{:}I_1)\ I_2 : \Box} \qquad \frac{\Gamma \vdash S : \Box \quad \Gamma, x{:}S \vdash I : \Box}{\Gamma \vdash \mathsf{some}[x{:}S]\ I : \Box}$$

$$\frac{\Gamma \vdash I_1 : \Box \quad \Gamma, l_1{:}I_1 \vdash \langle l_2{:}I_2, \ldots, l_n{:}I_n\rangle : \Box}{\Gamma \vdash \langle l_1{:}I_1, \ldots, l_n{:}I_n\rangle : \Box}$$

(signature formation)

$$\frac{\Gamma \vdash T : \mathsf{type} \quad \Gamma, \mathsf{type}\ t = T \vdash \mathsf{sig}\ E\ \mathsf{end} : \Box}{\Gamma \vdash \mathsf{sig\ type}\ t = T, E\ \mathsf{end} : \Box}$$

$$\frac{\Gamma \vdash \mathsf{type} : \Box \quad \Gamma, \mathsf{type}\ t \vdash \mathsf{sig}\ E\ \mathsf{end} : \Box}{\Gamma \vdash \mathsf{sig\ type}\ t, E\ \mathsf{end} : \Box} \qquad \frac{\Gamma \vdash T : \mathsf{type} \quad \Gamma \vdash \mathsf{sig}\ E\ \mathsf{end} : \Box}{\Gamma \vdash \mathsf{sig\ val}\ v : T, E\ \mathsf{end} : \Box}$$

(type formation)

$$\frac{\Gamma \vdash T_1 : \mathsf{type} \quad \Gamma \vdash T_2 : \mathsf{type}}{\Gamma \vdash T_1 \to T_2 : \mathsf{type}}$$

$$\frac{\Gamma \vdash m : \mathsf{sig\ type}\ t, E\ \mathsf{end}}{\Gamma \vdash m.t : \mathsf{type}} \qquad \frac{\Gamma \vdash m : \mathsf{sig\ type}\ t = T, E\ \mathsf{end} \quad \Gamma \vdash T : \mathsf{type} \quad \Gamma \vdash t : \mathsf{type}}{\Gamma \vdash m.t = T : \mathsf{type}}$$

(module)

$$\frac{\Gamma \vdash m : \mathsf{sig\ type}\ t, E\ \mathsf{end}}{\Gamma, \mathsf{type}\ t \vdash m : \mathsf{sig}\ E\ \mathsf{end}}$$

$$\frac{\Gamma \vdash m : \mathsf{sig\ type}\ t = T, E\ \mathsf{end}}{\Gamma, \mathsf{type}\ t = T \vdash m : \mathsf{sig}\ E\ \mathsf{end}} \qquad \frac{\Gamma \vdash m : \mathsf{sig\ val}\ v : T, E\ \mathsf{end}}{\Gamma, \mathsf{val}\ v : T \vdash m : \mathsf{sig}\ E\ \mathsf{end}}$$

(abstraction)
$$\frac{\Gamma, x{:}I_1 \vdash m : I_2 \quad \Gamma \vdash \mathsf{all}(x{:}I_1)\, I_2 : \Box}{\Gamma \vdash \mathsf{fun}(x{:}I_1)\, m : \mathsf{all}(x{:}I_1)\, I_2}$$

(application)
$$\frac{\Gamma \vdash m_1 : \mathsf{all}(x{:}I_2)\, I_1 \quad \Gamma \vdash m_2 : I_2}{\Gamma \vdash m_1\ m_2 : I_1\{x \leftarrow m_2\}}$$

(tuple construction)
$$\frac{\begin{array}{c}\Gamma \vdash \langle l_2 = m_2, \ldots, l_n = m_n\rangle\{l_1 \leftarrow m_1\} : \langle l_2{:}I_2, \ldots, l_n : I_n\rangle\{l_1 \leftarrow m_1\} \\ \Gamma \vdash m_1 : I_1 \quad \Gamma \vdash \langle l_1{:}I_1, \ldots, l_n{:}I_n\rangle : \Box\end{array}}{\Gamma \vdash \langle l_1 = m_1, \ldots l_n = m_n\rangle : \langle l_1{:}I_1, \ldots, l_n{:}I_n\rangle}$$

(tuple selection)
$$\frac{\Gamma \vdash m : \langle l_1{:}I_1, \ldots, l_k{:}I_k, \ldots, l_n{:}I_n\rangle}{\Gamma \vdash m\#l_k : I_k\{l_1 \leftarrow m\#l_1\} \cdots \{l_{k-1} \leftarrow m\#l_{k-1}\}}$$

(declaration)
$$\frac{\Gamma, x{:}I_1 \vdash m : I \quad \Gamma \vdash \mathsf{some}[x{:}S]\, I : \Box}{\Gamma \vdash \mathsf{dec}[x{:}S]\, m : \mathsf{some}[x{:}S]\, I}$$

(use)
$$\frac{\Gamma \vdash m_1 : \mathsf{some}[x_1{:}S]\, I_1 \quad \Gamma, x_2{:}S, x_3{:}I_1 \vdash m_2 : I_2}{\Gamma \vdash \mathsf{use}\ m_1\ \mathsf{as}\ [x]y\ \mathsf{in}\ m_2 : \mathsf{some}[x_1{:}S]\ (I_2\{x_2 \leftarrow x_1\})}, x_3 \notin I_2$$

(conversion)
$$\frac{\Gamma \vdash m : I_1 \quad \Gamma \vdash I_2 : \Box \quad I_1/m <:_\Gamma I_2}{\Gamma \vdash m : I_2}$$

A.2.2 Core language

(declaration)
$$\frac{\Gamma, x{:}S \vdash m : I \quad \Gamma \vdash i : S \quad \Gamma \vdash \mathsf{some}[x{:}S]\, I : \Box}{\Gamma \vdash \mathsf{dec}[x{:}S\,\mathsf{is}\,i]\, m : \mathsf{some}[x{:}S]\, I}$$

(structure)
$$\frac{\Gamma \vdash \mathsf{sig}\ \ \mathsf{end} : \Box}{\Gamma \vdash \mathsf{struct}\ \ \mathsf{end} : \mathsf{sig}\ \ \mathsf{end}}$$

$$\frac{\Gamma \vdash T : \mathsf{type} \quad \Gamma \vdash \mathsf{struct}\ e\ \mathsf{end}\{t \leftarrow T\} : \mathsf{sig}\ E\ \mathsf{end}\{t \leftarrow T\} \quad \Gamma \vdash \mathsf{sig\ type}\ t = T, E\ \mathsf{end} : \Box}{\Gamma \vdash \mathsf{struct\ type}\ t = T, e\ \mathsf{end} : \mathsf{sig\ type}\ t = T, E\ \mathsf{end}}$$

$$\frac{\Gamma, \mathsf{type}\ t \vdash \mathsf{struct}\ e\ \mathsf{end} : \mathsf{sig}\ E\ \mathsf{end} \quad \Gamma \vdash \mathsf{sig\ type}\ t, E\ \mathsf{end} : \Box}{\Gamma \vdash \mathsf{struct\ datatype}\ t, e\ \mathsf{end} : \mathsf{sig\ type}\ t, E\ \mathsf{end}}$$

$$\frac{\Gamma, v{:}T \vdash c : T \quad \Gamma, v{:}T \vdash \mathsf{struct}\ e\ \mathsf{end} : \mathsf{sig}\ E\ \mathsf{end} \quad \Gamma \vdash \mathsf{sig\ val}\ v : T, E\ \mathsf{end} : \Box}{\Gamma \vdash \mathsf{struct\ val}\ v : T = c, e\ \mathsf{end} : \mathsf{sig\ val}\ v : T, E\ \mathsf{end}}$$

(value component)
$$\frac{\Gamma \vdash m : \mathsf{sig\ val}\ v : T, E\ \mathsf{end}}{\Gamma \vdash m.v : T}$$

further core language rules omitted

A.3 Conversion and reduction

A.3.1 Strengthening (/)

$$
\begin{aligned}
(\mathsf{sig}\ E\ \mathsf{end})/m &= \mathsf{sig}\ E/m\ \mathsf{end} \\
\varepsilon/m &= \varepsilon \\
(\mathsf{type}\ t, E)/m &= \mathsf{type}\ t = m.t, E/m \\
(\mathsf{type}\ t = T, E)/m &= \mathsf{type}\ t = T, E/m \\
(\mathsf{val}\ v : T, E)/m &= \mathsf{val}\ v : T, E/m \\
I/m &= I
\end{aligned}
$$

A.3.2 Subtyping of interfaces ($<:_\Gamma$)

$$
\begin{array}{lcl}
\mathsf{all}(x{:}I_1)\ I_2 <:_\Gamma \mathsf{all}(x{:}I_1')\ I_2' & \Leftarrow & I_1' <:_\Gamma I_1,\ I_2 <:_{\Gamma'} I_2' \\
 & & \quad \text{where } \Gamma' = \Gamma, x{:}I_1' \\
\langle\,\rangle <:_\Gamma \langle\,\rangle & & \\
\langle l_1{:}I_1, \ldots, l_n{:}I_n\rangle <:_\Gamma \langle l_1{:}I_1', \ldots, l_n{:}I_n'\rangle & \Leftarrow & I_1 <:_\Gamma I_1', \\
 & & \quad \langle l_2{:}I_2, \ldots, l_n{:}I_n\rangle <:_{\Gamma'} \langle l_2{:}I_2', \ldots, l_n{:}I_n'\rangle \\
 & & \quad \text{where } \Gamma' = \Gamma, l_1{:}I_1 \\
\mathsf{some}[a{:}S]\ I <:_\Gamma \mathsf{some}[a{:}S']\ I' & \Leftarrow & S <:_\Gamma S',\ I <:_{\Gamma'} I' \\
 & & \quad \text{where } \Gamma' = \Gamma, a{:}S \\
\mathsf{sig}\ E\ \mathsf{end} <:_\Gamma \mathsf{sig}\ \ \mathsf{end} & & \\
\mathsf{sig}\ D_1, \ldots, D_n\ \mathsf{end} <:_\Gamma \mathsf{sig}\ D_1', \ldots, D_m'\ \mathsf{end} & \Leftarrow & D_{\sigma(i)} <:_{\Gamma'} D_i' \text{ for } i \in \{1, \ldots, m\} \\
 & & \quad \text{where } \Gamma' = \Gamma, D_1, \ldots, D_n \\
 & & \quad \text{and} \quad \sigma : \{1, \ldots, m\} \to \{1, \ldots, n\} \\
\mathsf{type}\ t <:_\Gamma \mathsf{type}\ t & & \\
\mathsf{type}\ t = T <:_\Gamma \mathsf{type}\ t & & \\
\mathsf{val}\ v : T_1 <:_\Gamma \mathsf{val}\ v : T_2 & \Leftarrow & T_1 \approx_\Gamma T_2 \\
\mathsf{type}\ t = T_1 <:_\Gamma \mathsf{type}\ t = T_2 & \Leftarrow & T_1 \approx_\Gamma T_2
\end{array}
$$

A.3.3 Manifest equality of types ($\approx_\Gamma$)

$$
\begin{array}{lcl}
T_1 \to T_2 \approx_\Gamma T_1' \to T_2' & \Leftarrow & T_1 \approx_\Gamma T_1',\quad T_2 \approx_\Gamma T_2' \\
t \approx_\Gamma T & \Leftarrow & \Gamma \vdash t = T : \mathsf{type} \\
m.t \approx_\Gamma T & \Leftarrow & \Gamma \vdash m : \mathsf{sig}\ \mathsf{type}\ t = T, E\ \mathsf{end} \\
t \approx_\Gamma t & & \\
m_1.t \approx_\Gamma m_2.t & \Leftarrow & m_1 = m_2 \\
T_2 \approx_\Gamma T_1 & \Leftarrow & T_1 \approx_\Gamma T_2 \\
T_1 \approx_\Gamma T_3 & \Leftarrow & T_1 \approx_\Gamma T_2,\quad T_2 \approx_\Gamma T_3
\end{array}
$$

A.3.4 Syntactic equality of expressions modulo reduction (=)

$$
m_1 = m_2 \quad \Leftarrow \quad m_1' \equiv m_2',\quad m_1 \twoheadrightarrow m_1',\quad m_2 \twoheadrightarrow m_2'
$$

A.3.5 Simple syntactic equality of expressions ($\equiv$)

$$
\begin{array}{lcll}
x \equiv x & & & \\
m_1\ m_2 \equiv m_1'\ m_2' & \Leftarrow & m_1 \equiv m_1', & m_2 \equiv m_2' \\
m\#l \equiv m'\#l & \Leftarrow & m \equiv m' & \\
\langle\,\rangle \equiv \langle\,\rangle & & & \\
\langle l_1 = m_1, \ldots, l_n = m_n\rangle & & & \\
\quad \equiv \langle l_1 = m_1', \ldots, l_n = m_n'\rangle & \Leftarrow & m_1 \equiv m_1', & \langle l_2 = m_2, \ldots l_n = m_n\rangle\{l_1 \leftarrow m_1\} \\
 & & & \equiv \langle l_2 = m_2', \ldots, l_n = m_n'\rangle\{l_1 \leftarrow m_1\} \\
\mathsf{use}\ m_1\ \mathsf{as}\ [x]y\ \mathsf{in}\ m_2 & & & \\
\quad \equiv \mathsf{use}\ m_1'\ \mathsf{as}\ [x]y\ \mathsf{in}\ m_2' & \Leftarrow & m_1 \equiv m_1', & m_2 \equiv m_2'
\end{array}
$$

A.3.6 Reduction ($\twoheadrightarrow$)

$$
\begin{array}{lcll}
m \twoheadrightarrow m & & & \\
m_1 \twoheadrightarrow m_2 & \Leftarrow & m_1 \to m_2 & \\
m_1 \twoheadrightarrow m_3 & \Leftarrow & m_1 \twoheadrightarrow m_2, & m_2 \twoheadrightarrow m_3
\end{array}
$$

A.3.7 One-step reduction ($\to$)

The compatibility rules for selecting a redex and α-conversion are omitted.

$$
\begin{array}{lcl}
(\mathsf{fun}(x\!:\!A)\ m_1)\ m_2 & \to & m_1\{x \leftarrow m_2\} \\
\langle l_1 = m_1, \ldots, l_k = m_k, \ldots, l_n = m_n\rangle\#l_k & \to & m_k\{l_1 \leftarrow m_1\}\cdots\{l_{k-1} \leftarrow m_{k-1}\} \\
\mathsf{use}\ (\mathsf{dec}[x_1\!:\!S]\ m_1)\ \mathsf{as}\ [x_2]x_3\ \mathsf{in}\ m_2 & \to & \mathsf{dec}[x_1\!:\!S]\ (m_2\{x_2 \leftarrow x_1\}\{x_3 \leftarrow m_1\})
\end{array}
$$

Technische Universität Braunschweig
Informatik-Berichte ab Nr. 93-07

93-07	S.Schwiderski, 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 *light*-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
95-06	V.S.Cherniavsky	Über semantische und formalistische Beweismethoden in den exakten Wissenschaften
95-07	A.Zeller, D.Lütkehaus	DDD - A Free Graphical Front-End for UNIX Debuggers
95-08	A.Zeller	Smooth Operations with Square Operators – The Version Set Model in ICE
95-09	P. Funk, A. Lewien, G. Snelting	Algorithms for Concept Lattice Decomposition and their Application
96-01	A. Zeller, G. Snelting	Unified Versioning Through Feature Logic
96-02	M. Goldapp, U. Grottker, G. Snelting	Validierung softwaregesteuerter Meßsysteme durch Program Slicing und Constraint Solving
96-03	C. Lindig, G. Snelting	Modularization of Legacy Code Based on Mathematical Concept Analysis
96-04	J. Adámek, J. Koslowski, V. Pollara, W. Struckmann	Workshop Domains II (Proceedings)
96-05	F.-J. Grosch	A Syntactic Approach to Structure Generativity