The Missing Link – Dynamic Components for ML

Andreas Rossberg
Universität des Saarlandes
rossberg@ps.uni-sb.de

Abstract

Despite its powerful module system, ML has not yet evolved for the modern world of dynamic and open modular programming, to which more primitive languages have adapted better so far. We present the design and semantics of a simple yet expressive first-class component system for ML. It provides dynamic linking in a type-safe and type-flexible manner, and allows selective execution in sandboxes. The system is defined solely by reduction to higher-order modules plus an extension with simple module-level dynamics, which we call *packages*. To represent components outside processes we employ generic *pickling*. We give a module calculus formalising the semantics of packages and pickling.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Modules, packages; F.3.3 [*Studies of Program Constructs*]: Type structure

General Terms Languages, Theory

Keywords modules, units, separate compilation, dynamic linking, components, dynamic typing, distributed programming, pickling

1. Introduction

ML modules [18, 15, 16, 10] are an indispensable aid for large-scale programming, thanks to their ability to express complex modular abstractions. They provide namespacing, encapsulation, genericity, and architectural configuration in form of a small higher-order language with an expressive, strong type system. However, all this expressive power is purely static: a program must be determined and provided in its entirety at build time, prior to running it. Once built, configuration, functionality and extent of an ML program is fixed – in other words, programs are *closed*.

1.1 Typed Open Programming

This situation is increasingly at odds with the requirements of today's computing reality. Many applications today require an *open* approach to programming, where additional functionality can be acquired at runtime, or behaviour can be exchanged with other, probably remote processes. Frequent requests for features like dynamic linking, marshalling, and distribution in different ML forums indicate that the demand has reached ML.

No serious support for this has yet been provided in practical ML systems, or similar languages. If at all, they only offer basic – and unsafe – marshalling functionality. Ironically, 'lesser' languages, especially Java [4], have long taken the lead with respect to safe integration of respective features. Of course, they have done so only with considerable compromises in language design and semantics For example, Java is safe, but does not have any legitimate notion of *type* safety for open programs (cf. Section 9).

So the question is, how can we add comparable functionality to ML in a cleaner and safer way? That is, how can we support *open programming* without throwing all of ML's nice properties out of the window, especially with respect to its beloved type system?

1.2 Components

To address this question, we present the design and semantics of a simple yet powerful system of *components* for ML. A component is a software block that can be created, deployed, and loaded independently from others. We distinguish components from modules: while modules provide *logical* separation, lexical scoping, genericity, and encapsulation, components provide *physical* separation and dynamic composition. Both mechanisms complement each other. In our system, components *contain* modules. The component system we present provides all of the following:

- *Separate compilation*. Components are independent units.
- *Dynamic linking*. Loading can be performed when needed.
- *Type safety*. Components carry fully checked type information.
- *Subtyping*. Type checking is tolerant against interface changes.
- *Static linking*. Components can be bundled into larger ones.
- *Dynamic creation*. Components can be computed at runtime.
- *Sandboxing*. Component *managers* enable custom policies.

From the user perspective, the most visible extension to ML is allowing compilation units to be preceded by a number of *import declarations* of the form

import *specs* **from** *url*

where *specs* are ML signature specifications and *url* a string denoting the URL under which the component with the respective signature can be located. The necessary linking is performed at runtime and involves a structural dynamic type check on ML signatures.

More interestingly, components are in fact first-class and can be created at runtime: an expression of the form

comp *imports* **in** *specs* **with** *decs* **end**

denotes a value of the abstract type `component` that will be evaluated dynamically. Its export signature is described by *specs* and implemented by *decs*. Both can refer to objects from the surrounding environment, which will be captured in the component's closure. Such *computed components* can be transmitted to other processes, as a flexible means for exchanging behaviour, while abstracting from site-specific resources.

Our component system is defined solely by reduction to *higher-order modules* [10], extended with only two generic mechanisms:

ICFP'06 September 16–21, 2006, Portland, Oregon, USA.

- *Packages.* A variation of dynamics [1] which carries modules.
- *Pickling.*[1] A mechanism for importing/exporting values.

Loading pickles requires dynamic type checks to verify consistent use. To separate concerns, we uncouple dynamic type checking from (un)pickling by introducing packages as a stand-alone feature that provides universal dynamic typing. This choice has several advantages: (1) dynamic typing is usable independent from pickling; (2) a pickle can easily be checked against multiple types; (3) the type of a pickle may be known statically, such that always including type information is redundant. The component system we are going to present will explore the first two points to realise link-time type checking. Ad (1), components are first-class, so they are not necessarily obtained from pickles. Ad (2), a single component may be imported by several others, and hence has to be checked against multiple, potentially different signatures. We only tangentially touch the third point in this paper. As a sketch, consider typed communication channels between processes. A dynamic type check may only be necessary for *establishing* a connection, not for every *transmission*. In particular, we want to avoid transmitting repetitive type descriptions with every single value (in Section 3.2 we briefly mention *proxies* as an example of such a mechanism).

1.3 Alice ML

Packages and pickling, and the presented component system, form a fundamental part of Alice ML [28, 2], a conservative extension of Standard ML [19] designed for open programming. It adds higher-order modules, packages, components, as well as concurrency with futures, laziness, and support for distributed programming.

We will hence use Alice ML for concrete syntax and examples, although our design is equally applicable to other ML dialects with higher-order modules.

The present paper extends on [28], which gave an overview of the concepts found in Alice ML, including components, but omitted much of the details and did not provide any formal treatment.

1.4 Structure

We approach our component design in two parts. First we describe packages (Section 2) and pickling (Section 3) as stand-alone extensions to ML, and present a formal semantics in the framework of a higher-order module calculus (Section 4). In the second part we introduce the notions of component (Section 5) and component manager (Section 6) and show how they are expressible in terms of the former constructs (Section 7). Based on this, we briefly discuss some extensions of the framework (Section 8). Finally, we compare to related work (Section 9) and conclude (Section 10).

2. Packages

The core problem of dynamically exchanging program fragments is how to do type-safe I/O of language-level entities. The solution is well-known: dynamics [1]. They complement the type system with just the amount of dynamic checking necessary to gain the desired flexibility, by adding a single universal type that carries a value along with dynamic type information. Unlike so-called "dynamically typed" languages, or hybrids like Java, dynamics still maintain most of the invariants of a strong type system, because they isolate dynamic type checks to well-defined operations.

Unfortunately, dynamics never really caught on. There are several reasons for this, but we presume the following ones are most severe:

- Dynamics, carrying a single value, do not have the right granularity for most applications.
- The complex typecase construct proposed for unpacking dynamics is too unwieldy. A simple type equality check on the other hand would be too inflexible.
- Omnipresent runtime types are considered too expensive.

We hence propose a variation of dynamics that is based on ML modules, and which we call *packages*. Packages address the above issues: unlike dynamics, a package carries a (possibly higher-order) *module*, along with its dynamic signature. Unpacking only performs a subtype test, but incorporates the full structural subsignature relation of ML modules, providing for a high amount of flexibility. Finally, runtime type passing is confined to the module level.

2.1 Basics

A package is a first-class value of the primitive type package. Intuitively, it contains a module, along with a dynamic description of the module's signature, which we call the *package signature*. Package signatures exist only in the dynamic semantics, they are not tracked by the static type system. That property sets packages apart from other proposals for modules as first-class values [29, 10], where the signature is always fixed statically.

There are only two basic operations on packages. A package is created by injecting a module into the type package. Alice ML employs the following syntax for this purpose:

pack $modexp : sigexp$

This expression creates a package from the module expressed by $modexp$ (which may denote a functor, thanks to higher-order modules). The signature expression $sigexp$ defines the package signature. Of course, the module expression must match this signature.

The inverse operation is projection, eliminating a package. The module expression

unpack $exp : sigexp$

takes a package computed by exp (which needs to have type package) and extracts the contained module – provided that the package signature matches the *target signature* denoted by $sigexp$. That is, unpacking performs a dynamic type check. If the check fails, the pre-defined exception Unpack is raised.[2] Statically, the whole expression has the signature $sigexp$.

For example, we can wrap the SML library structure Array into a package,

```
val p = pack Array : ARRAY
```

and unpack it successfully using the same signature:

```
structure Array' = unpack p : ARRAY
```

Any attempt to unpack p with an incompatible signature will fail. On the other hand, all subsequent accesses to Array' or members of it are statically type-safe, no further checks are required.

Dynamic type checking for packages is performed on full ML signatures. Signatures describe interfaces, and support a rich notion of subtyping, often called *matching* in ML nomenclatura. The Definition of Standard ML [19] formalises it quite intuitively as a relation that consists of two dimensions:

- *Enrichment.* The more specific signature may contain more fields (of more general types) than the less specific one.
- *Instantiation.* Abstract types in the less specific signature can be realised by concrete types in the more specific signature.

[1] Also known as *serialisation* or *marshalling* – the more abstract term *pickling* emphasizes the fact that the representation always is self-contained, which is not the case with most existing serialization or marshalling mechanisms, especially with respect to code.

[2] In the current version of Alice ML that exception is named Mismatch.

Subtyping hence allows a great amount of flexibility with respect to composing modules. In particular, it is robust against extension or refinement of a module interface. That is essential for adequately describing program architectures in a modular manner. Obviously, it is even more desirable in ever-changing dynamic applications.

Note that the same flexibility would *not* be given by simply providing ordinary (core) dynamics plus modules as first-class values: to unwrap a dynamic and retrieve a first-class module value, its *precise* actual signature had to be known. Unwrapping modulo subtyping would require subtyping to pervade the core language.

2.2 Dynamic Type Sharing

Although the module Array' obtained by packing and unpacking the library module Array as shown above is identical to the original, there is a caveat: the ARRAY signature contains an abstract type array; the way we unpacked it, type Array'.array will be statically incompatible with the original type Array.array. Since the types in a package generally cannot be determined statically, all abstract types in the target signature must indeed be considered fully abstract – and hence different from any other – by the (static) type system.

However, type compatibility can be obtained easily – we just need to enforce it in the usual ML way, namely by putting suitable *type sharing constraints* on the target signature:

```
structure Array' = unpack p
                 : ARRAY where type α array = α Array.array
```

With this formulation, the type Array'.array is statically known to be equal to Array.array. Of course, unpacking will only succeed if the package actually meets this requirement at runtime.

The constraint effectively expresses *dynamic type sharing*. By restricting the target signature we ensure static compatibility, at the cost of preventing successful use of non-standard implementations of arrays. Much like for programming with functors, it depends on the application how much sharing is required.

As a fine point, dynamic type sharing works as demonstrated only because the package signature supplied with a pack expression is interpreted transparently. That is, writing

```
val p = pack Array : ARRAY
```

is actually equivalent to

```
val p = pack Array : ARRAY where type α array = α Array.array
```

This behaviour mirrors the semantics of SML's transparent ascription operator (:), but dynamically: the actual package signature is obtained by refining the ascribed signature with the concrete types found in the respective module. Technically, dynamic *selfification* [15], or *strengthening* [16], is performed (Section 4.4). It is worth noting that *without* this behaviour, the fragment

```
val p = pack Array : ARRAY
structure Array' = unpack p
                 : ARRAY where type α array = α Array.array
```

would fail with an Unpack exception (as one would expect), but the contorted, yet seemingly equivalent example

```
val p = pack Array : ARRAY
structure Aux = unpack p : ARRAY
val p' = pack Aux : ARRAY where type α array = α Aux.array
structure Array' = unpack p'
                 : ARRAY where type α array = α Array.array
```

would still succeed (see the formal semantics in Section 4). Obviously, such pathological behaviour is neither desirable nor useful, thus we chose the transparent interpretation.

2.3 Parametricity

Dynamic type sharing can be utilised to dynamically test for type equivalences. Consequently, evaluation is not *parametric* [24, 5], because a program can behave differently depending on the outcome of such a test.

Parametricity is a valuable property for polymorphic languages, because it enables all of the following:

- *Type erasure*. Programs can be compiled and executed without maintaining costly type information at runtime.
- *Theorems* [34]. Polymorphic types state strong invariants about terms, which allow deriving a variety of useful laws.
- *Abstraction* [24, 20]. It is possible to achieve encapsulation solely by abstracting or quantifying over types.

Looking closer, it is obvious that evaluation of *modules* cannot be parametric in the presence of packages – the behaviour of unpack must depend on dynamic type information. However, for Alice ML the semantics of dynamic types has been crafted such that the *core* language, where polymorphism is ubiquitous, is not affected. In particular, unlike functors, polymorphic functions are still fully parametric: the usual laws still hold, and they can be compiled using standard type erasure techniques.

This nicely fits the syntactic setup of ML: on the module level, passing types is always made explicit in the syntax. Core polymorphism, on the other hand, is completely implicit. Thus the syntax provides a clear model to the programmer: only types explicitly supplied in a program, by means of named type declarations, can potentially affect its operational behaviour, and induce a cost.

In order to maintain parametricity for core polymorphism we need strict separation between implicit and explicit types: it is required that no operation consuming dynamic types – i.e. unpack and sealing – may ever depend on the instantiation of a polymorphic type variable. Fortunately, this comes for free: both these operations require an explicit signature annotation. Signatures can only refer to other types and signatures, and neither type declarations nor signatures are allowed to contain free type variables [19]. Consequently, the meaning of a signature expression cannot depend on polymorphic type variables, even in local scopes. For instance, neither of the following definitions is valid:[3]

```
fun α mypack1 (x : α) = pack (val it = x) : (val it : α)
fun α mypack2 (x : α) = let type t = α   (* illegal! *)
                        in  pack (val it = x) : (val it : t) end
```

In the first case, the local α in the signature is considered locally quantified, but x does not have the universal type $\forall\alpha.\alpha$. In the second example, the local type declaration for t, containing a free occurence of α, is syntactically invalid.[4]

The maintenance of parametricity in the core language is not without drawback. The fact that ordinary core-level evaluation cannot directly depend on types may appear to be a severe restriction. The expressive power of dynamic typing remains relatively limited, maybe too limited. However, that objection can be diluted by the existence of work-arounds that allow emulating most of the missing expressiveness:

- Local modules and higher-order functors often enable lifting polymorphic function definitions to the module level, by turning them into functors. For example, the function mypack from above can be reformulated straightforwardly as a functor:

  ```
  functor MyPack (type t; val x : t) =
      struct val p = pack (val it = x) : (val it : t) end
  ```

  However, this technique potentially requires turning all polymorphic functions up the call chain into functors, which might quickly become unwieldly. Moreover, the module language is

[3] Alice ML allows abbreviating **struct** . . . **end** and **sig** . . . **end** as (. . .).

[4] Note that the closedness restriction on local type declarations would not be necessary for plain ML. However, for Alice ML its presence was fortunate.

not Turing-complete, so not all desired functions may be expressible (still, it contains System F_ω).

- A more general work-around is to abuse packages as means for communicating types and modules as first-class values. Of course, that approach could be deemed somewhat questionable, because it essentially means evading the static type system and relying on dynamic typing more than necessary.
- If this should prove to be insufficient, adding conventional first-class modules [29] to the language would be a general solution.

So far, we have only encountered few interesting examples – in the context of what dynamic typing in Alice ML is intended for – which needed to employ any work-around, and the former two were adequate enough in those cases.

2.4 Abstract Types

A remaining problem is the third utilisation of parametricity mentioned in the previous section: type abstraction. Type abstraction is a central feature of the ML module system. Its type system guarantees *abstraction safety*: values of abstract type can only be constructed and deconstructed by the implementation of the abstraction itself. There is no means, within the language, that allows client code to break an abstraction.

The addition of dynamic typing raises delicate issues, because the accompanying loss of parametricity invalidates the standard existential model of type abstraction. To maintain abstraction safety, abstract types must be represented by dynamically generated type names instead, as described in previous work [25]. For Alice ML we hence employ such a dynamically generative semantics of type abstraction. In order to maintain abstraction safety even across process boundaries (e.g. when transferring types via pickling), dynamic type names are globally unique.

3. Pickling

Our main motivation for adding dynamic typing is type-safe import and export. Packages are the key primitive for enabling high-level, type-safe interfaces for exchanging language-level data structures between a process and the outside world. However, in order to export a value, it has to be transformed into a self-contained, platform-independent external representation. Such a representation is called a *pickle*. The transformation is known as *pickling*.

3.1 Persistence

One obvious application of pickling is *persistence*, i.e. I/O to an external medium, e.g. a hard drive. In Alice ML, it is available through two simple library functions:

```
val save : string × package → unit
val load : string → package
```

The save operation writes a package to a file of a given name. The inverse operation load retrieves a package from a file.

All saved pickles contain a single value of the type package – reducing the problem of dynamically checking the type of imported values to the type checking performed by unpack. Since packages contain modules, and modules can embed arbitrary language entities (values, types, higher-order modules, and in Alice ML, even signatures), allowing only packages to be pickled is not a restriction. Note that a pickle contains all reachable code, too.

For example, instead of just wrapping the library module Array into a package as demonstrated in Section 2, we can in fact write it to disk, using the following idiomatic code:

```
Pickle.save ("array.alc", pack Array : ARRAY)
```

The package can be loaded – by the same process, or an arbitrary different one – by composing the inverse operations, in reverse order:

```
structure Array' = unpack Pickle.load "array.alc" : ARRAY
```

The obtained structure Array' can now be used as a substitute for Array – it is an identical copy (however, see Section 2.2 for the issue of type sharing). Any attempt to unpack the loaded package with an incompatible signature will fail with an Unpack exception. All subsequent accesses to Array' or members of it are statically type-safe.

To achieve full safety, two kinds of checks have to be performed:

1. The load operation has to check that the file contains a well-formed pickle. This resembles low-level *verification*, as e.g. performed by the Java Virtual Machine [17] for its byte code – however, unlike Java class files, pickles not only contain (strongly typed) code, but also arbitrary structured data. Failure at this point is considered I/O failure.[5]
2. The unpack operator has to check that the package signature matches the static assumptions, i.e. the target signature. Failure at this point is a dynamic type error.

Note that the nature of the two checks might be quite different in practice, because they operate on different abstraction levels.

3.2 Distribution

Other applications of pickling in Alice ML involve distributed programming. For example, there is a pair of functions that allows a process to make available a module on the net, and enables other processes to grab it:

```
val offer : package → url
val take : url → package
```

The offer function returns a so-called *ticket*, a URL that identifies the package in the network (e.g. via host name, port number, and a running ID). Another process that is communicated this ticket can retrieve the offered package as a pickle.

Usually, the transferred package will contain a structure with *proxy* functions [28], which are mobile wrappers for stationary functions that transparently perform remote function calls when invoked at other sites. Proxy calls are again realised by employing pickling to transmit argument and result values. However, unlike save/load and offer/take, proxy communication is statically typed – it is an example of a typed communication channel as mentioned in the introduction. Hence there is no need to communicate packages. If the sender is not trusted, it may still be necessary to *verify* well-formedness of every received value, though.

3.3 Resources and Security

Pickling is not without restrictions. It is disallowed to export so-called *resources*, i.e. values that are private to a process – this includes critical operations, local information like file handles, and all stateful data. We say that resources are *sited*, and any attempt to pickle a resource will be dynamically detected and yield the exception Sited.

Resources are precluded from pickling for security reasons. Under this restriction, pickles are always secure – code loaded from a pickle cannot perform any critical actions without the receiving site explicitly giving it the capability to do so. In particular, there deliberately is no implicit rebinding [6] in Alice ML. If rebinding is desired, it must be requested explicitly, either by procedural abstraction, or more comfortably, by component abstraction. We

[5] This check is not yet implemented in the current version of Alice.

$$
\begin{array}{lrl}
\text{types} & \tau ::= & \mathsf{Typ}\, m \mid \Pi x{:}\sigma.\tau \mid \tau_1 \times \tau_2 \mid \langle\!|\star|\!\rangle \mid \Psi \\
\text{terms} & e ::= & \mathsf{Val}\, m \mid \mathsf{fix}\, f(x{:}\sigma){:}\tau.e \mid e\, m \mid \langle e_1, e_2\rangle \mid \pi_i e \mid \\
 & & \mathsf{let}\, x{=}m\, \mathsf{in}\, (e:\tau) \mid \mathsf{pack}\, m\, \mathsf{as}\, \sigma \mid \psi(v) \mid \\
 & & \mathsf{pickle}\, e \mid \mathsf{unpickle}\, x{\Leftarrow}e_1\, \mathsf{in}\, e_2\, \mathsf{else}\, (e_3:\tau) \\
\text{signatures} & \sigma ::= & 1 \mid [\![\Omega]\!] \mid [\![\tau]\!] \mid \Pi x{:}\sigma_1.\sigma_2 \mid \Sigma x{:}\sigma_1.\sigma_2 \mid \mathrm{S}(m) \\
\text{modules} & m ::= & x \mid \langle\rangle \mid [\tau] \mid [e:\tau] \mid \lambda x{:}\sigma.m \mid m_1\, m_2 \mid \\
 & & \langle x{=}m_1, m_2\rangle \mid \pi_i m \mid \mathsf{let}\, x{=}m_1\, \mathsf{in}\, (m_2:\sigma) \mid \\
 & & \mathsf{unpack}\, x{\Leftarrow}e\, \mathsf{as}\, \sigma\, \mathsf{in}\, m_1\, \mathsf{else}\, (m_2:\tau)
\end{array}
$$

Figure 1. Calculus syntax

get back to this in Section 5.2, and explain how security policies can be programmed in Section 6.2.

Stateful data is excluded from pickling because stateful pickling would essentially provide a generic cloning operation, which is well-known to break stateful abstractions.[6] An alternative would be distributed state. For Alice ML we decided against it because of the complexity involved, and because it would significantly weaken the self-containment property of pickles (in particular, the potential for dead references in pickles seems high).

4. Formal Semantics

We will now present a formal semantics of an ML module system with packages and pickling. In order to keep the complexity manageable, we idealise the ML language to a much simpler calculus. The basis of this calculus is (a subset of) the higher-order module system by Dreyer, Crary & Harper (DCH) [10]. We equip their system with an operational semantics, and extend it with packages and pickling. For simplicity, we omit applicative functors and first-class modules (though packages provide somewhat similar functionality, so we reuse the syntax); both could easily be reintegrated.

A more serious omission is that the calculus leaves out any form of *sealing* (type abstraction). In a language as rich as the one at hand, reconciling sealing with dynamic typing is not at all straightforward: it would significantly complicate the operational semantics, at least if we wanted to preserve abstraction safety in the presence of dynamic typing. To cope with that, we would have to introduce dynamic type generativity and fully-fledged (dependently typed) higher-order *coercions* to both the core and the module level. We have developed a respective extension for F_ω [25], including an account of applicative generativity (as with weak sealing). The generalisation to a dependently typed language like the one discussed here is certainly interesting, but intricate. We hence leave it for future work.[7]

The Alice ML implementation realises generativity, but because of type erasure it does not have to worry about coercions. The corresponding operational semantics of SML with packages and generative type abstraction has been defined elsewhere [26], in the framework of the SML Definition [19].

4.1 Terms and Modules

Figure 1 presents the full syntax of our calculus. It mostly resembles DCH, and we reuse their type system almost unchanged. The calculus consists of two layers: terms and modules. Terms are assigned types, and modules are assigned signatures, respectively.

[6] The current implementation still allows stateful pickling.

[7] Note that most of the complications already arise with dependent *kinds*, so that specifying the operational semantics in terms of a "phase-splitting" transformation to F_ω enriched with singleton kinds [9] is not any easier.

$$
\begin{array}{rcll}
\sigma \to \tau & := & \Pi x{:}\sigma.\tau & \text{where } x \notin \mathrm{FV}(\tau) \\
\lambda x{:}\sigma.e & := & \mathsf{fix}\, f(x{:}\sigma){:}\tau.e & \text{where } f \notin \mathrm{FV}(e) \\
\sigma_1 \to \sigma_2 & := & \Pi x{:}\sigma_1.\sigma_2 & \text{where } x \notin \mathrm{FV}(\sigma_2) \\
\sigma_1 \times \sigma_2 & := & \Sigma x{:}\sigma_1.\sigma_2 & \text{where } x \notin \mathrm{FV}(\sigma_2) \\
\langle m_1, m_2\rangle & := & \langle x{=}m_1, m_2\rangle & \text{where } x \notin \mathrm{FV}(m_2) \\
\mathsf{unpickle}\, e & := & \mathsf{unpickle}\, x{\Leftarrow}e\, \mathsf{in}\, \mathsf{Val}\, x\, \mathsf{else}\, (\bot_{\langle\!|\star|\!\rangle} : \langle\!|\star|\!\rangle) & \\
\mathsf{unpack}\, e\, \mathsf{as}\, \sigma & := & \mathsf{unpack}\, x{\Leftarrow}e\, \mathsf{as}\, \sigma\, \mathsf{in}\, x\, \mathsf{else}\, (\bot_\sigma : \sigma) &
\end{array}
$$

Figure 2. Syntactic abbreviations

The term language provides recursive functions ($\mathsf{fix}\, f(x{:}\sigma){:}\tau.e : \Pi x{:}\sigma.\tau$) and tuples ($\langle e_1, e_2\rangle : \tau_1 \times \tau_2$), and the ability to project a value from a module ($\mathsf{Val}\, m$). The type language provides a similar operation to project a type from a module ($\mathsf{Typ}\, m$). Functions take modules as arguments, which allows them to also express polymorphism (by taking modules that embed types); consequently, they are dependently typed. Furthermore, the term language includes let, and the new constructs for packages (of type $\langle\!|\star|\!\rangle$) and pickles (of type Ψ), which will be described later. Note that let also binds a module, so that the system only has module variables.

The module level contains trivial modules ($\langle\rangle : 1$), embedded types ($[\tau] : [\![\Omega]\!]$) and embedded values ($[e{:}\tau] : [\![\tau]\!]$) as primitives. From these we can form binary structures ($\langle x{=}m_1, m_2\rangle : \Sigma x{:}\sigma_1.\sigma_2$) and functors ($\lambda x{:}\sigma.m : \Pi x{:}\sigma_1.\sigma_2$); larger structures have to be expressed as nested binary structures. For simplicity, we consider all functors impure (that is, $\Pi x{:}\sigma_1.\sigma_2$ in our system corresponds to $\Pi^{\mathrm{gen}} x{:}\sigma_1.\sigma_2$ in DCH). The module language also provides a let construct, and the unpack operator for packages, described below. Finally, the signature language provides *singleton signatures* $\mathrm{S}(m)$, which are explained in the next section.

For clarity, we will also use a number of abbreviations, which are collected in Figure 2. Moreover, we will often omit type annotations from embedded terms, let expressions, and unpack and unpickle branches (the latter are necessary to maintain a minimal typing property for local binders, as discussed in [10]).

As a concrete example using most of the basic constructs of the calculus, consider the following SML code:

```
functor F (X : sig type t; type u = t × t; val f : t → u end) =
struct
    fun α g (x : X.t) (y : α) = (y, X.f x)
end
```

The functor F can be expressed in the calculus as follows:

$$
\begin{array}{l}
\lambda X : (\Sigma t{:}[\![\Omega]\!].\Sigma u{:}\,\mathrm{S}([\mathsf{Typ}\, t \times \mathsf{Typ}\, t]).\Sigma f{:}[\![[\![\mathsf{Typ}\, t]\!] \to \mathsf{Typ}\, u]\!].1). \\
\quad \langle g = [\lambda\alpha{:}[\![\Omega]\!].\lambda x{:}[\![\mathsf{Typ}(\pi_1 X)]\!].\lambda y{:}[\![\mathsf{Typ}\, \alpha]\!]. \\
\qquad\quad \langle \mathsf{Val}\, y,\ \pi_1(\pi_2(\pi_2 X))\, x\rangle], \\
\quad \langle\rangle\rangle
\end{array}
$$

Note the structure nesting and embedding of types and terms, and the corresponding projections. The role of the singleton signature assigned to u will be explained in the next section.

4.2 Type System

We reuse the type system of DCH almost unchanged, only omitting applicative functors and first-class modules. See [27] for the complete typing rules. For space reasons we will not go into details, and mostly restrict our discussion to the rules that have been added for packages and pickling (Sections 4.4 and 4.5). We refer the interested reader to [10] for an in-depth discussion. Although we will not discuss algorithmic type checking either, we note that the DCH type checking algorithm is easy to adapt to our system (primarily because we only require very minor modifications to the module equivalence algorithm, where two new forms of path arise,

$$
\begin{array}{lrl}
\text{values} & v & ::= \mathsf{fix}\, f(x{:}\sigma){:}\tau.e \mid \langle v_1, v_2\rangle \mid \mathsf{pack}\, w\ \mathsf{as}\ \sigma \mid \psi(v) \\
\text{m-values} & w & ::= \langle\rangle \mid [\tau] \mid [v{:}\tau] \mid \lambda x{:}\sigma.m \mid \langle w_1, w_2\rangle \\
\text{contexts} & E & ::= \_ \mid E\, m \mid \langle E, e\rangle \mid \langle v, E\rangle \mid \pi_i E \mid \\
 & & \quad \mathsf{pickle}\, E \mid \mathsf{unpickle}\, x{\Leftarrow}E\ \mathsf{in}\ e_1\ \mathsf{else}\ (e_2{:}\tau) \mid \\
 & & \quad \bar{E}\{\bar{M}\{E\}\} \\
 & \bar{E} & ::= \mathsf{Val}\, \_ \mid v\, \_ \mid \mathsf{let}\, x{=}\_\ \mathsf{in}\ (e{:}\tau) \mid \mathsf{pack}\, \_\ \mathsf{as}\ \sigma \mid E\{\bar{E}\} \\
\text{m-contexts} & M & ::= \_ \mid M\, m \mid w\, M \mid \langle x{=}M, m\rangle \mid \langle w, M\rangle \mid \pi_i M \mid \\
 & & \quad \mathsf{let}\, x{=}M\ \mathsf{in}\ (m{:}\sigma) \mid \bar{M}\{\bar{E}\{M\}\} \\
 & \bar{M} & ::= [\_{:}\tau] \mid \mathsf{unpack}\, x{\Leftarrow}\_\ \mathsf{as}\ \sigma\ \mathsf{in}\ m_1\ \mathsf{else}\ (m_2{:}\sigma) \mid \\
 & & \quad M\{\bar{M}\}
\end{array}
$$

Figure 3. Values and evaluation contexts

$[\langle\star\rangle]$ and $[\Psi]$, both of which are trivial; the only new module-level expression, unpack, is impure and hence irrelevant).

We have to comment on one particular feature of the type system, because it appears in the operational semantics of unpack: *singleton signatures*. Their purpose is to express non-trivial equivalences between modules: $\mathrm{S}(m)$ is the signature of all modules equivalent to m. For a module $m : [\![\Omega]\!]$ we may derive $m : \mathrm{S}(m)$, a derivation that is sometimes called *selfification*. Given $m' : \mathrm{S}(m)$ we can derive the *static module equivalence* $m' \cong m$. In general however, module expressions may have effects (one pro-forma effect is sealing), DCH hence employ a simple effect system, that precludes impure modules from being used in equivalence judgements and singletons (in our subset, the only two effects distinguished are purity P and impurity W).

Since a type equivalence relation $\mathsf{Typ}\, m_1 \equiv \mathsf{Typ}\, m_2$ is reduced to $m_1 \cong m_2$ in the system, singleton signatures enable the expression of *type sharing*. If, for instance, $m : \mathrm{S}([\tau])$, then $\mathsf{Typ}\, m \equiv \tau$ can be derived. Hence a singleton signature expresses the manifest type specification type u = t×t in the argument of functor F above.

Although singletons are only defined on modules embedding types (i.e. with signature $[\![\Omega]\!]$) a priori, they can be generalised to higher order by means of syntactic sugar. Higher-order singletons allow to selfify arbitrary modules: the rule $m : \mathrm{S}_\sigma(m)$ is admissible for all pure $m : \sigma$, propagating identities of all types embedded in m. [27] defines higher-order singletons $\mathrm{S}_\sigma(m)$ and gives the corresponding admissible rules.

4.3 Operational Semantics

DCH do not include an operational semantics. In order to model dynamic typing, we have to provide such a semantics for the subset of the system considered here. We employ a standard call-by-value, left-to-right evaluation regime. Figures 3 and 4 show the definition of values and evaluation contexts for term and module language, and the reduction rules.

Most reduction rules are standard, except for packages and pickling, which are explained below. The main complication is with respect to the definition of evaluation contexts, because we must accomodate mutual nesting of term and module expressions. The special contexts $\bar{E}$ and $\bar{M}$ encode the possible positions of modules in terms and vice versa.

Given the technical development in the extended version of [10], it is not difficult to show soundness for the calculus:

Proposition 1 (Preservation)

1. *If* $\vdash e : \tau$ *and* $e \to e'$, *then* $\vdash e' : \tau$.
2. *If* $\vdash_\kappa m : \sigma$ *and* $m \to m'$, *then* $\vdash_\kappa m' : \sigma$.

$$
\begin{array}{rll}
\mathsf{Val}[v : \tau] & \to v & \\
(\mathsf{fix}\, f(x{:}\sigma){:}\tau.e)\, w & \to e\{\mathsf{fix}\, f(x{:}\sigma){:}\tau.e/f, w/x\} & \\
\pi_1\langle v_1, v_2\rangle & \to v_1 & \\
\pi_2\langle v_1, v_2\rangle & \to v_2 & \\
\mathsf{let}\, x{=}w\ \mathsf{in}\ e & \to e\{w/x\} & \\
\mathsf{pickle}\, v & \to \psi(v) & \\
\mathsf{unpickle}\, x{\Leftarrow}\psi(v)\ \mathsf{in}\ e_1\ \mathsf{else}\ e_2 & \to e_1\{[v]/x\} & \text{if} \vdash v : \langle\star\rangle \\
\mathsf{unpickle}\, x{\Leftarrow}\psi(v)\ \mathsf{in}\ e_1\ \mathsf{else}\ e_2 & \to e_2 & \text{otherwise}
\end{array}
$$

$$
\begin{array}{rll}
(\lambda x{:}\sigma.m)\, w & \to m\{w/x\} & \\
\langle x{=}w, m\rangle & \to \langle w, m\{w/x\}\rangle & (x \in \mathrm{FV}(m)) \\
\pi_1\langle w_1, w_2\rangle & \to w_1 & \\
\pi_2\langle w_1, w_2\rangle & \to w_2 & \\
\mathsf{let}\, x{=}w\ \mathsf{in}\ m & \to m\{w/x\} & \\
\mathsf{unpack}\, x{\Leftarrow}(\mathsf{pack}\, w\ \mathsf{as}\ \sigma)\ \mathsf{as}\ \sigma'\ \mathsf{in}\ m_1\ \mathsf{else}\ m_2 & \to m_1\{w/x\} & \text{if} \vdash \mathrm{S}_\sigma(w) \le \sigma' \\
\mathsf{unpack}\, x{\Leftarrow}(\mathsf{pack}\, w\ \mathsf{as}\ \sigma)\ \mathsf{as}\ \sigma'\ \mathsf{in}\ m_1\ \mathsf{else}\ m_2 & \to m_2 & \text{otherwise}
\end{array}
$$

Figure 4. Reduction Rules

Proposition 2 (Progress)

1. *If* $\vdash e : \tau$ *and* $e \neq v$, *then* $e \to e'$.
2. *If* $\vdash_\kappa m : \sigma$ *and* $m \neq w$, *then* $m \to m'$.

4.4 Packages

Packages are supported by an additional type, which we write $\langle\star\rangle$, and the corresponding introduction and elimination constructs:

$$\mathsf{pack}\, m\ \mathsf{as}\ \sigma$$
$$\mathsf{unpack}\, x{\Leftarrow}e\ \mathsf{as}\ \sigma\ \mathsf{in}\ m_1\ \mathsf{else}\ m_2$$

Since the calculus does not have exceptions, unpack requires branching to handle the failure case. Two reduction rules define the operational semantics of packages:

$$
\begin{array}{rll}
\mathsf{unpack}\, x{\Leftarrow}(\mathsf{pack}\, w\ \mathsf{as}\ \sigma)\ \mathsf{as}\ \sigma'\ \mathsf{in}\ m_1\ \mathsf{else}\ m_2 & \to m_1\{w/x\} & \text{if} \vdash \mathrm{S}_\sigma(w) \le \sigma' \\
\mathsf{unpack}\, x{\Leftarrow}(\mathsf{pack}\, w\ \mathsf{as}\ \sigma)\ \mathsf{as}\ \sigma'\ \mathsf{in}\ m_1\ \mathsf{else}\ m_2 & \to m_2 & \text{otherwise}
\end{array}
$$

The subtyping test is defined by the subtyping judgement of the static semantics. If the involved signatures are not in the required subtyping relation the else branch will be invoked. In the rest of the paper we will mostly omit the branching for clarity, and just write $\mathsf{unpack}\, e\ \mathsf{as}\ \sigma$, courtesy of the syntactic abbreviation given in Figure 2. It diverges in case of failure, approximating the use of exceptions (all signatures are inhabited by diverging computations, see the definition of $\bot_\sigma$ in [27].

Since evaluation is call-by-value, the environment Γ in the subtyping judgement will always be empty. The target signature is not simply checked against the package signature σ, but against the singleton signature of the contained module w, which formally realises the selfification of the package signature σ we already mentioned in Section 2.2. We can make the motivating example for this behaviour more concrete now – consider:

$$
\begin{array}{l}
\mathsf{let}\, p = [\mathsf{pack}\ [\tau]\ \mathsf{as}\ [\![\Omega]\!]]\ \mathsf{in} \\
\mathsf{let}\, x = \mathsf{unpack}\, \mathsf{Val}\, p\ \mathsf{as}\ \mathrm{S}([\tau])\ \mathsf{in}\ \ldots
\end{array}
$$

Without the selfification in the unpack evaluation rule, evaluation of x would fail (diverge), because $[\![\Omega]\!] \not\leq \mathrm{S}([\tau])$. However, the type identity can still be propagated and rediscovered by an indirection;

$$\frac{\Gamma \vdash_\kappa m : \sigma}{\Gamma \vdash \mathsf{pack}\, m\ \mathsf{as}\ \sigma : \langle\!|\star|\!\rangle} \quad (1)$$

$$\frac{\Gamma \vdash e : \langle\!|\star|\!\rangle \qquad \Gamma, x{:}\sigma \vdash_\kappa m_1 : \sigma' \qquad \Gamma \vdash_\kappa m_2 : \sigma' \qquad \Gamma \vdash \sigma' : \square}{\Gamma \vdash_{\mathrm{W}} \mathsf{unpack}\, x \Leftarrow e\ \mathsf{as}\ \sigma\ \mathsf{in}\ m_1\ \mathsf{else}\ (m_2 : \sigma') : \sigma'} \quad (2)$$

$$\frac{\Gamma \vdash e : \langle\!|\star|\!\rangle}{\Gamma \vdash \mathsf{pickle}\, e : \Psi} \quad (3) \qquad \frac{\Gamma \vdash \square}{\Gamma \vdash \psi(v) : \Psi} \quad (4)$$

$$\frac{\Gamma \vdash e_1 : \Psi \qquad \Gamma, x{:}[\![\langle\!|\star|\!\rangle]\!] \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau \qquad \Gamma \vdash \tau : \square}{\Gamma \vdash \mathsf{unpickle}\, x \Leftarrow e_1\ \mathsf{in}\ e_2\ \mathsf{else}\ (e_3 : \tau) : \tau} \quad (5)$$

Figure 5. Selected typing rules

evaluation of

$$\begin{array}{l} \mathsf{let}\, p \ = [\mathsf{pack}\ [\tau]\ \mathsf{as}\ [\![\Omega]\!]]\ \mathsf{in} \\ \mathsf{let}\, x' = \mathsf{unpack}\, \mathsf{Val}\, p\ \mathsf{as}\ [\![\Omega]\!]\ \mathsf{in} \\ \mathsf{let}\, p' = [\mathsf{pack}\, x'\ \mathsf{as}\ \mathrm{S}(x')]\ \mathsf{in} \\ \mathsf{let}\, x \ = \mathsf{unpack}\, \mathsf{Val}\, p'\ \mathsf{as}\ \mathrm{S}([\tau])\ \mathsf{in}\ \ldots \end{array}$$

would still succeed, yielding the original module $[\tau]$ for x, under the statically transparent signature $\mathrm{S}([\tau])$. The reason is that $[\tau]$ will be substituted for x' in the package signature of p' during evaluation, leaving

$$\begin{array}{l} \mathsf{let}\, p' = [\mathsf{pack}\ [\tau]\ \mathsf{as}\ \mathrm{S}([\tau])]\ \mathsf{in} \\ \mathsf{let}\, x \ = \mathsf{unpack}\, \mathsf{Val}\, p'\ \mathsf{as}\ \mathrm{S}([\tau])\ \mathsf{in}\ \ldots \end{array}$$

after reduction of the first two let-bindings. Consequently, it is more natural to make the former example succeed already (in a language with sealing a possible alternative is to have pack perform sealing, but experiments in early versions of Alice ML revealed it to be a nuisance in practice: for instance, the dynamic type sharing example in Section 2.2 would require unexpected and tedious annotations at pack time to *prevent* unwanted sealing).

Figure 5 contains the typing rules for packages (1 and 2). Rule 1 enforces that the module wrapped in a package matches the provided package signature. Rule 2 defines that unpack delivers a module of the target signature σ, bound to x in the success branch – or takes the failure branch. Both branches need to match σ'.

Note that unpacking is an impure operation (specified by the W effect annotation in the conclusion). For obvious reasons, this treatment is essential for soundness in presence of a potentially impure core language, where the argument might evaluate to different package values at different points in time.

We should note that the calculus is only intended as an idealised model language, not as an *internal language* targettable by translation (as e.g. the original DCH) – the use of the subtyping judgement in the operational semantics prevents a direct translation from a richer external language. For that purposes, the unpack side condition had to be replaced by a separate judgement mirroring the external subtyping relation. Moreover, reduction would probably have to perform dynamic conversions to bridge between both signatures. Such considerations are outside the scope of this paper. See [26] for an operational semantics that encompasses full SML.

4.5 Pickling

Our calculus provides an abstract account of pickling, in the form of three constructs:

- pickle e creates a pickle of the value computed by e.
- unpickle $x \Leftarrow e$ in e_1 else e_2 takes a pickle e and extracts its value, binding it to x in e_1; if the pickle is malformed, e_2 is evaluated instead.
- $\psi(v)$ represents a pickle itself, i.e. a value that is some serialized representation of the value v (think of a byte string or a file). Pickles are assigned type Ψ.

The key characteristic of this mechanism is that there is no requirement for the term v in a pickle $\psi(v)$ to actually be well-formed! This models the fact that in practice, pickles can be created outside the runtime, by extra-linguistic means, and the language has no way to enforce that they are well-formed. Indeed, a pickle may be deliberately forged by an attacker. This liberty is visible in the typing rules, shown in Figure 5. In particular, rule 4 does not have any premise regarding v.

If pickles may be malformed, how can we establish soundness? How can the runtime guarantee its integrity? This requires *verification* when the pickle is loaded. In our model, verification simply amounts to a well-formedness side condition in the reduction rules for unpickle:

$$\begin{array}{ll} \mathsf{unpickle}\, x \Leftarrow \psi(v)\ \mathsf{in}\ e_1\ \mathsf{else}\ e_2 \to e_1\{[v]/x\} & \text{if} \vdash v : \langle\!|\star|\!\rangle \\ \mathsf{unpickle}\, x \Leftarrow \psi(v)\ \mathsf{in}\ e_1\ \mathsf{else}\ e_2 \to e_2 & \text{otherwise} \end{array}$$

Upon loading, it is checked that a pickle actually contains a well-formed representation of a value (of type $\langle\!|\star|\!\rangle$). Because a pickle is self-contained, i.e. a closed expression, the environment in the side condition's judgement is empty.

Despite this robustness, we want to preclude that malformed pickles can be created from within the language. A programmer has to invoke pickle e to create pickles (the form $\psi(v)$ should be considered inaccessible in the surface language). This first evaluates e and then creates a pickle from the result. That is, we have a simple reduction rule

$$\mathsf{pickle}\, v \to \psi(v)$$

Unlike rule 4, typing rule 3 requires the operand of pickle to be a well-formed term.

Consider the following four examples:

1. unpack (unpickle pickle(pack $\lambda x{:}1.\langle\rangle$ as $1 \to 1$)) as $1 \to 1$
2. unpack (unpickle pickle(pack $\lambda x{:}1.\langle\rangle$ as $1 \to 1$)) as $1 \to 1 \to 1$
3. unpack (unpickle ψ(pack $\lambda x{:}1.x\ \langle\rangle$ as $1 \to 1$)) as $1 \to 1$
4. unpack (unpickle pickle(pack $\lambda x{:}1.x\ \langle\rangle$ as $1 \to 1$)) as $1 \to 1$

The first three are all statically well-typed, but only the first will evaluate successfully. The second will fail (i.e. diverge) due to a dynamic type error in unpack, the third due to a verification error during unpickling. The last example is rejected by the (static) type system, because the pickled value is not denoted by a well-formed expression. Contrast this to the third example, which represents a (statically valid) malformed pickle.

For simplicity, our calculus does not model resources. Proper handling of resources would be relatively straightforward: it amounts to extending the pickle operator to conditional form and adding a side condition to its primary reduction rule that ensures that no resource names occur in v (assuming resources are simply represented as names). Otherwise it would fail.

5. Components

Packages already allow dynamic loading and exchanging of modules. However, these modules have to be fully evaluated and closed. If we wanted to delay evaluation, or have it depend on other modules then we had to resort to functional (or functorial) abstraction. The notion of *components* we are introducing now provides a much more comfortable and flexible means for achieving the same effect. Nevertheless, in Section 7 we will see that it can actually be expressed as a – relatively simple – higher-order abstraction.

5.1 Compilation Units

A program in our system consists of a potentially open set of components that are created separately and loaded dynamically [28]. Every component provides a module (its *export*) and accesses an arbitrary number of modules retrieved from other components (its *imports*). Import and export interfaces are typed by ML signatures.

Each source file defines a component: the contained ML declarations are interpreted as a structure body, forming the export module. Other components can be accessed through a prologue of import declarations:

```
import specs from url
```

The signature specifications *specs* describe the entities used from the imported structure, along with their types. All identifiers bound in the specification are in scope in the rest of the component. Thanks to higher-order modules, these entities can include functors. For instance, the following are valid imports:

```
import structure Server : sig val run : (α→β) → (α→β) end
    from "http://my.org/server"
import functor MkRedBlackMap (Key : ORDERED) : MAP
    from "x-alice:/lib/data/MkRedBlackMap"
```

The string in an import declaration contains the URL under which the component is to be acquired at runtime. Although the URL is hardwired into the code, its interpretation is completely up to the responsible *component manager* (Section 6), and hence configurable. Usually it is either a local file, an HTTP address, or a virtual URL denoting system library components (Alice ML uses the x-alice: scheme for this purpose).

To execute a program, a designated *root* component is *evaluated*, meaning that its defining declarations are evaluated in sequence, according to the dynamic semantics of the language. Evaluation of an import declaration triggers loading and evaluation of the respective component, a process referred to as *dynamic linking*. However, every component is loaded at most once. We defer discussion of the details of linking until Section 6.

Compilation units are always syntactically closed. There are no free identifiers, not even for most primitive operators like op+. They are all bound by imports, thus enabling separate compilation.

However, writing down the signatures for all imported modules would be tedious in practice. As syntactic sugar, we hence allow the type annotations on import specifications to be dropped. It suffices that the imported components are accessible (in compiled form) during compilation, so that the compiler can insert the respective types from their export signatures. For example, the previous import declarations could be abbreviated to

```
import structure Server from "http://my.org/server"
import functor MkRedBlackMap
    from "x-alice:/lib/data/MkRedBlackMap"
```

As an additional service, the Alice ML compiler automatically thins implicit signatures by removing all entities that are not directly or indirectly referred in the remainder of the component. Doing so makes the compiled component maximally robust against eventual changes to unused parts of an interface.

For convenience, and for compatiblity with Standard ML, Alice ML furthermore allows to omit imports for modules from the Standard ML Basis library [13] by default. The respective import declarations are implicitly prepended to every compiled source. Again, the compiler thins signatures, and removes all redundant imports.

5.2 Computed Components

Compilation is the most obvious, but not the only way to create components. Nor do components necessarily live in files. In fact, components are first-class entities in our design, and can be constructed dynamically by an ML process. We call such components *computed components*, as opposed to compiled components (strictly speaking, a compiled component is just the special case of a component computed by the compiler, though).

```
structure Component :
sig
    type component
    exception Failure of url × exn
    val fromPackage : package → component
    val save : string × component → unit
    val load : string → component
    ...
end
```

Figure 6. The Component structure

Within the language, a component is a value of the abstract type component, which is defined in the library structure Component. Figure 6 shows an excerpt of its signature. Values of this type can be constructed with a new syntactic form:[8]

```
comp imports in specs with decs end
```

A component expression has a syntactic structure similar to a compilation unit. The main difference is that its export signature must be given explicitly, in form of a sequence of signature specifications *specs* between the keywords **in** and **with**. The environment obtained from *decs* must match this signature. Naturally, imports and declarations are not evaluated when the component is constructed, but when it is linked. Thus it can not only import other components, it can also perform sited operations and generate side effects (through functionality obtained from imported library components).

Note also that the imports scope over the signature – an export signature may depend on types defined in other components.

The less visible but more important difference between compiled and computed components is that the latter need not be closed. Hence computed components can embody information that is obtained dynamically. That is useful for at least two purposes:

- *Pre-computation*. Through a staged building process, components can be created that readily provide data structures that are expensive to compute, or should be "statically generated".
- *Mobility*. Dynamic behaviour that depends on resources can be wrapped into a component and be passed to other processes.

The former application requires pickling: by calling Component.save a component can be pickled to disk. Component files created this way behave exactly like components created by the compiler. In fact, there is only one uniform file format: all pickle files are actually components and can be used as such – the package-based pickling operations presented in Section 3 are simply convenient wrapper functions.

As an example, the following program creates a component that will print its creation date when invoked later:

```
val date = Date.toString (Date.fromTimeLocal (Time.now ()))
val component =
  comp
    import structure TextIO from "x-alice:/lib/system/TextIO"
  in
    val hello : unit → unit
  with
    fun hello () =
        TextIO.print ("Hello world! Created at " ^ date ^ "\n")
  end
val _ = Component.save ("hello", component)
```

[8] This syntax is not yet available in the Alice System; the library functor Component.Create is provided as a substitute.

The created component can be loaded or imported. The simplest possible program utilising it is the following:

```
import val hello : unit → unit from "hello"
val _ = hello ()
```

Packages can be directly converted to components with the Component.fromPackage function, which is sometimes convenient. However, the key difference between a component and a simple package is that a component enables access to resources and other functionality local to the target site. For instance, the previous example would not work if we omitted the import declaration for TextIO – then the local instance of TextIO.print would be in the closure of component, and pickling would fail with a Sited exception because print is a resourceful operation. With the import, we effectively enforce *rebinding* of all scoped references to TextIO on the target site (the target site may choose to prevent the import, see Section 6.2).

Rebinding is particularly important for distributed programming. Exchanging dynamic behaviour between processes can be achieved by creating a *mobile* component serving two purposes:

- it closes over all entities obtained at creation site, thus containing the necessary dynamic information (like date above),
- it abstracts over all entities to be obtained at the target site, thus enabling pickling and (re)binding (like TextIO above).

The former is handled automatically by the semantics of pickling. The latter can be controlled by the use of import declarations in the definition of the mobile component. The rebinding itself is automatic, by the process of dynamic linking.

The Alice ML library provides infrastructure for different distribution scenarios based on components. The simplest would be to use the offer/take mechanism (Section 3.2) to communicate a component and execute it locally. This is suitable for a client/server scenario, for instance. Another possible scenario is distributed computation based on a master/slave architecture: a master site distributes computational tasks to a number of slaves (or workers), which return their results. To spawn new processes on remote sites, the Alice ML library offers the function

```
val run : string × component → package
```

which takes a host name and a component and evaluates the component on the respective site. It communicates back the component's export module as a package. More detailed explanation and examples can be found in [28].

6. Component Managers

6.1 Dynamic Linking

Evaluation of an import declaration triggers dynamic linking. Linking a component involves four steps:

1. *Resolution*. The import URL is normalised relative to the URL of the current component.
2. *Acquisition*. If the component is being requested for the first time, it is loaded.
3. *Evaluation*. If the component has been loaded afresh, its body is evaluated and its dynamic export signature computed.
4. *Type Checking*. The component's export signature is matched against the respective import signature.

Each of the steps can fail: the component might be inaccessible or malformed, evaluation may terminate with an exception, or type checking may discover a mismatch. Under each of these circumstances the exception Component.Failure (Figure 6) is raised. The URL it carries denotes the requested component, and the nested exception describes the precise cause of the failure. In particular, it can be an I/O exception, the exception Unpack (Section 2.1), or a number of other exceptions defined in the library.

```
signature COMPONENT_MANAGER =
sig
    exception Conflict
    val acquire : url → component
    val eval : component → package
    val enter : url × package → unit
    val lookup : url → package option
    val link : url → package
end
```

Figure 7. The signature of a component manager

To enable control over this process, linking is performed with the help of a *component manager*. It is the entity responsible for locating and loading components, and keeping a table of components loaded already. The default component manager is a module of the runtime library that is initialised on startup of an Alice ML process. It starts with an empty table and incrementally fills it as required by evaluation of the root component or any of its imports.

To a program, the responsible component manager is accessible not only implicitly for imports, but also explicitly as the library structure ComponentManager (Figure 7). Using that structure, a program can operate on first-class components and influence the manager in more direct ways. It provides several basic services on component values, most of which implement one of the above steps:

- acquire retrieves a component, without actually evaluating or entering it into the table,
- eval evaluates a component into a package containing its export, without entering it into the table,
- enter enters an export package under a specified URL, raising Conflict if the URL is already taken,
- lookup retrieves a component from the table,
- link loads and enters a component from a specified URL.

The function link combines the sequence of operations usually required to link a component given its URL, i.e. aquires, evaluates and enters a component, in a single operation.

Ultimately, eval is the only operation that actually consumes a value of type component – no other ways exist to access it. Possible example scenarios for its use are a client that sends a component to a server to utilise special services local to the server; the server has to apply eval from its local component manager to run it. In a master/slave architecture the slaves will evaluate a component received from the master – the run function from Section 5.2 can actually be implemented in terms of some lower-level service like SSH and a stub component on the slave site that retrieves and evaluates a first-class component.

6.2 Sandboxing

The relevance of component managers lies in their ability to control imports. In an open setting it is important to handle untrusted components, and to restrict their capabilities. For example, they should not be given unrestricted access to the local file system.

To deal with this, we adopt the approach taken by Java: components can be executed in a *sandbox*. Sandboxing relies on two factors: (1) all critical resources and capabilities are sited (Section 3.3) and hence have to be acquired via import through local system components; (2) it is possible to create custom managers that restrict the access to certain components.

A custom manager simply is a user-defined implementation of the COMPONENT_MANAGER signature. For example, the Alice ML library provides a functor to create new managers with spe-

cialised behaviour. When a component is evaluated by a specific manager (e.g. by using its eval or link procedure), this manager is inherited by all components that are directly or indirectly requested by the first one. Thus, if the manager restricts access to critical system libraries, none of these components can gain access to them.

There are several possible ways in which a custom manager can restrict access: (1) It can simply reject loading from specific system URLs. (2) It can restrict the signature under which specific system components are made available, by forwarding the request to its parent manager, but repackaging the result with a thinner signature (e.g. removing the operations for opening output files from TextIO). (3) It can substitute critical components by security-sensitive wrappers, that dynamically check for more fine-grained policies with each operation (this is basically what Java does).

So far there is only minimal infrastructure for sandboxing in the Alice ML library. We are still exploring the design space, and plan to extend the library in future versions of the system.

7. Decomposing Components

At first, components may look like a complex mechanism. In this section we will show that this is not the case, by giving a simple reduction of components to functions and packages. The merit of this reduction is three-fold: (1) it keeps the language conceptually simple, (2) it defines the semantics of components without the need for additional technical machinery, (3) type soundness follows.

7.1 Components

A component can be understood as a function that evaluates to a package containing the export module. More precisely, the abstract type component can be implemented as a higher-order function type:

```
type component = (url → package) → package
```

Its argument encapsulates the component manager, needed to acquire imports. Applying the function evaluates the component.

Accordingly, a component expression

```
comp imports in specs with decs end
```

can be viewed as syntactic sugar for the function

```
fn import ⇒ let importdecs in pack (decs) : (specs) end
```

where *import* is a reserved identifier and *importdecs* is obtained from *imports* by rewriting every import declaration

```
import specs from url
```

to

```
structure strid = unpack import url : (specs)
open strid
```

where *strid* is a fresh identifier.

This simple transformation fully determines the semantics of components. In particular, it makes obvious how dynamic type checking is performed for imports, and how acquisition of imported component is delegated to the component manager: every component receives the function *import* for acquiring its imports. It evaluates to a package that contains its own export.

The most subtle point is that export signatures are actually determined dynamically, due to the transparent interpretation of pack (Section 2.1). This enables complex type sharing: an export signature may mention a type that has been imported abstractly, still other components further down the dependency graph may match this type concretely.

7.2 Component Managers

It remains to be shown how component managers themselves can be implemented. Figure 8 contains a simple model implementation.

```
exception Conflict
val table = ref [] : (url × package) list ref

fun import' parent =
let
    fun acquire url =
        Component.load url handle e ⇒ raise Failure (url, e)

    fun lookup "x-alice:/lib/system/ComponentManager" =
        pack (
          exception Conflict = Conflict
          val acquire = acquire
          val lookup = lookup  val enter = enter
          val eval = eval' "."   val link = import' "."
        ) : COMPONENT_MANAGER
      | lookup url =
        List.find (fn (x,_) ⇒ x = url) (!table)

    and enter (url, package) =
         if isSome (lookup url) then raise Conflict
         else table := (url, package) :: !table

    and eval' url component =
        component (import' url) handle e ⇒ raise Failure (url, e)

    fun link url =
    let  val url' = resolve (parent, url) in
         case lookup url' of
              SOME package ⇒ package
            | NONE ⇒
              let val package = eval' url' (acquire url') in
                    enter (url', package); package
              end
    end
in
    link
end
```

Figure 8. A canonical component manager

Apparently, most functions are straightforward. However, we show a quite limited version of acquire, which can only handle file URLs. For other URL schemes (particularly http:) additional services may be accessed, which we will not describe here.

The main complication is URL resolution: all URLs in an import declaration have to be interpreted relative to the domain and path of the URL under which the importing component was acquired. This is necessary to make groups of components relocatable across directory structures and network domains. Consequently, the *import* function passed to a component must know about that component's associated URL, that URL has to be passed as an additional parent argument. Assuming existence of an auxiliary function *resolve* : url × url → url, the internal link function can then perform the necessary resolution. When evaluating a component C, link passes along C's URL to the evaluation function eval', which constructs an *import* function from it that is suitable for loading C's own imports.

Explicit access to the component manager is enabled in this implementation by special-casing the internal lookup function on the system URL of the component manager. If applied to that URL, lookup just returns an appropriate package containing the required functionality. Note that these functions do not interpret URLs relative to a parent – no unambiguous notion of parent exists in their case, because the functions can be passed around first-class. Instead, URLs are pragmatically resolved relative to the local host and current working directory, which we indicate with "." here.

Given the described reduction of components and component managers, execution of a program can be thought of as evaluation of the simple application

```
import' "." root
```

where *root* is the URL of the program's root component, and "." again denotes the current working directory. All observable properties of program execution follow from this decomposition.

8. Extensions

8.1 Lazy linking

So far, we have assumed that all dynamic linking is to be performed eagerly. However, in practice it is often desirable to delay linking of individual components until they are actually accessed, in order to keep the working set smaller, minimize startup times, or access remote locations just in time.

Given a language with support for laziness, it is straightforward to extend our model to lazy dynamic linking. There is only one crucial change: the right-hand side of a rewritten import declaration has to be evaluated lazily. Taking Alice ML as our vehicle, which supports module-level laziness through its future mechanism [21], this amounts to simply rewriting an import declaration to

```
structure strid = lazy (unpack import url : (specs))
open strid
```

(Note that **open** just affects scoping, it does not trigger evaluation. Evaluation is triggered when the first field is needed.)

This is the semantics that is actually used in Alice ML. More details can be found in [28], which also describes how the component manager has to be refined to deal with concurrency.

8.2 Static Linking

When *developing* an application, it usually is good advice to split it into small enough components, so that they can be modified and compiled independently. When the final application is *deployed*, however, it is preferable to have it consist of as few parts as possible, to ease installation and minimize potential for failure. It is thus important to decrease the level of granularity of components when moving from individual programming tasks to the final product.

The Alice System supports this with a simple notion of *static linking*, or *bundling*: it provides a linker tool that takes a set of components, including one designated root component, and bundles them to form a single large component (in practice, the set is computed automatically from the dependency graph and URL-based cut-off rules given by the user). The resulting component has the same export as the root component, and the collective imports of all components required by one of the bundled components, but not in the set. The linker checks that all pairs of import/export signatures of internalized import edges match, otherwise the whole operation is rejected with a type error message.

Interestingly, the semantics of static linking can be defined in terms of custom component managers. Here is a brief sketch: consider linking of two components A and B, were B imports A and is used as the root component. Slightly simplifying (and ignoring URL resolution), what the static linker does is to create a wrapper component that looks as follows in desugared form (assuming the original component values are bound to a and b in the context, and hence will be in the closure):

```
fn import ⇒ let
    val aExport = lazy (unpack a import : A_SIG)
    fun import' "A" = aExport
      | import' url = import url
in b import' end
```

Basically, the new component creates a trivial custom component manager that treats requests for the URL "A" of the bundled component specially. All other component requests are just forwarded to the parent manager. This specialised manager is then passed on to the root component, ensuring that it will see the internalized version of A when requesting it. We here show the lazy semantics, where linking does not change the relative evaluation order of the linked components: A is still evaluated lazily.

The linker of the actual Alice System performs an additional optimization: since import/export signatures are checked at bundling time, it is statically known that the inserted unpack operations will not fail. The linker hence can safely remove the corresponding dynamic type checks, saving space and time. However, this transformation cannot be expressed on the language level, due to the lack of static first-class modules in Alice ML.

9. Related Work

There is extensive literature on dynamics as well as ML-style module systems, but to the best of our knowledge, combining the two has not been considered before. There have been different proposals for defining separate compilation for ML [7, 32], but they do not provide dynamic linking. The problem of software configuration and dynamic linking has been approached from a more general direction by many authors [8, 12, 3]. Most of this work is concerned with basic calculi, that introduce components as primitive concepts. There is relatively little work that investigates concrete language design in the context of ML or similar languages.

Facile [33] was an earlier extension of Standard ML for distributed programming. It provides *structure servers* for making persistent ML structures. A structure is retrieved from the server by requesting a module with a suitable signature, which implies a form of dynamic signature check. If several structures match a given signature, the last one stored is returned.

Closest to ours is the work on Acute [30], which like Alice ML extends an ML-like language with marshalling, concurrency, and support for dynamic loading. The mechanisms in Acute are comparatively complex, with no obvious reduction into simpler constructs like in our approach. Acute has no equivalent to computed components, a different concept of linear *marks* is needed to control the extent of rebinding, which is not directly comparable. No security mechanisms is provided, and due to uncontrolled implicit rebinding it is not clear how a sandboxing mechanism could be programmed. On the other hand, Acute provides other features not directly available in Alice ML, e.g. versioning constraints and different levels of type generativity. In Alice ML, the latter can only be simulated, to a certain degree, using pre-computation (Section 5.2).

Many of the concepts in Alice ML were inherited from Oz [31, 11], which has a component system similar to the one described here, equally based on pickling. However, Oz is an untyped language, no checking is performed for imports. Moreover, the decomposition of Oz components (called *functors*) differs from the one described in Section 7 and linking relies on untypable reflective capabilities on varyadic procedures.

Java [14] was the first major language with a serious focus on open programming. Instead of modules, Java components (*class files*) carry classes. Our approach to dynamic linking and sandboxing through component managers has clearly been inspired by Java's concept of *class loader*. However, there is no signature language or structural checking when a class is loaded, subsequent method calls may cause a NoSuchMethodError exception any time, undermining the type system. Java has no generic pickling mechanism, the provided serialisation requires considerable support from the programmer. Code cannot be serialised; consequently, there is no equivalent to computed components.

Scala [22] provides a much more expressive static type system, specifically intended for modular components. However, Scala reuses Java's runtime and library infrastructure and hence suffers from the same shortcomings regarding dynamic type checking.

Other functional languages have incorporated variants of dynamics to support type-safe I/O. Most notably, Clean supports a comparably rich form, with a full `typecase` construct [23]. However, no existing language seems to have generic pickling support for modules, and none has a higher-level component system that unifies pickles and compiled binaries.

10. Conclusion

We have described a simple but expressive approach for enriching ML with dynamic components suitable for open and distributed programming. It reuses much of what is already provided by ML modules and does require only very few new concepts. The central one are packages, which lift the well-known idea of dynamics to the level of modules. Another highlight of the design is its uniform use of pickling for representing components externally.

The system forms the basis of Alice ML and has been successfully implemented and used for a non-trivial code base. Pickling works surprisingly well in practice. In particular, pickles are usually pretty compact (in the order of few to few ten kilobytes), even when they contain computed components with significant amounts of code in their closure.

There are several open questions and directions for future work: formally, we want to reconcile the package semantics with abstraction-safe sealing, and integrate laziness into the calculus. It also might be interesting to look at type systems that trace use of resources to avoid sitedness errors. Implementation-wise, the main remaining issue is the integration of pickle verification into the system. This requires the ability to type-check the heap representation of data structures. Finally, pragmatically we would like to design suitable abstractions for making sandboxing accessible to the programmer.

Acknowledgments

The author thanks Didier Le Botlan, Jan Schwinghammer, and Guido Tack for discussion and comments on earlier drafts.

References

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. *Transactions on Programming Languages and Systems*, 13(2), 1991.

[2] Alice Team. *The Alice System*. Programming System Lab, Universität des Saarlandes, `http://www.ps.un-sb.de/alice/`, 2003.

[3] D. Ancona and E. Zucca. A calculus of module systems. *Journal of Fucntional Programming*, 12(2), 2002.

[4] K. Arnold and J. Gosling. *The Java Programming Language*. Addison–Wesley, 1996.

[5] E. Bainbridge, P. Freyd, A. Scedrov, and P. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70(1), 1989.

[6] G. Bierman, M. Hicks, P. Sewell, G. Stoyle, and K. Wansbrough. Dynamic rebinding for marshalling and update, with destruct-time lambda. In *Proc. 8th International Conference on Functional Programming*, 2003.

[7] M. Blume and A. Appel. Hierarchical modularity. *Transactions on Programming Languages and Systems*, 21(4), 1999.

[8] L. Cardelli. Program fragments, linking, and modularization. In *Proc. 24th Symposium on Principles of Programming Languages*, 1997.

[9] D. Dreyer. *Understanding and Evolving the ML Module System*. Phd thesis, Carnegie Mellon University, 2005.

[10] D. Dreyer, K. Crary, and R. Harper. A type system for higher-order modules. In *Proc. 30th Symposium on Principles of Programming Languages*, 2003. Expanded version available as CMU Technical Report CMU-CS-02-122R.

[11] D. Duchier, L. Kornstaedt, C. Schulte, and G. Smolka. A higher-order module discipline with separate compilation, dynamic linking, and pickling. Technical report, Universität des Saarlandes, 1998.

[12] M. Flatt and M. Felleisen. Units: Cool modules for HOT languages. In *Proc. Conference on Programming Language Design and Implementation*, 1998.

[13] E. Gansner and J. Reppy. *The Standard ML Basis Library*. Cambridge University Press, 2004.

[14] J. Gosling, B. Joy, and G. Steele. *The Java Programming Language Specification*. Addison–Wesley, 1996.

[15] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proc. 21st Symposium on Principles of Programming Languages*, 1994.

[16] X. Leroy. Manifest types, modules, and separate compilation. In *Proc. 21st Symposium on Principles of Programming Languages*, 1994.

[17] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison–Wesley, 1996.

[18] D. MacQueen. Modules for Standard ML. In *Symposium on LISP and Functional Programming*, Austin, USA, 1984.

[19] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[20] J. Mitchell and G. Plotkin. Abstract types have existential type. *Transactions on Programming Languages and Systems*, 10(3), 1988.

[21] J. Niehren, J. Schwinghammer, and G. Smolka. A concurrent lambda calculus with futures. *Theoretical Computer Science*, forthcoming.

[22] M. Odersky. *Programming in Scala*. École Polytechnique Fédérale de Lausanne, 2005.

[23] M. Pil. First class file I/O. volume 1268 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[24] J. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing*. North Holland, 1983.

[25] A. Rossberg. Generativity and dynamic opacity for abstract types. In *Proc. Principles and Practice of Declarative Programming*, 2003.

[26] A. Rossberg. The definition of Standard ML with packages. Technical report, Universität des Saarlandes, 2005. www.ps.uni-sb.de/Papers.

[27] A. Rossberg. The Missing Link – Dynamic Components for ML (extended). Technical report, Universität des Saarlandes, 2006.

[28] A. Rossberg, D. Le Botlan, G. Tack, T. Brunklaus, and G. Smolka. Alice ML through the looking glass. In *Trends in Functional Programming*, volume 5. Intellect, 2006.

[29] C. Russo. First-class structures for Standard ML. In *Proc. 9th European Symposium on Programming*, 2000.

[30] P. Sewell, J. Leifer, K. Wansbrough, F. Z. Nardelli, M. Allen-Williams, P. Habouzit, and V. Vafeiadis. Acute: high-level programming language design for distributed computation. In *10th International Conference on Funcitonal Programming*, 2005.

[31] G. Smolka. The Oz programming model. In *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*. Springer, 1995.

[32] D. Swasey, T. Murphy, K. Crary, and R. Harper. A separate compilation extension to standard ml. Technical Report CMU-CS-06-104, Carnegie Mellon University, 2006.

[33] B. Thomsen, L. Leth, and T.-M. Kuo. A facile tutorial. In *7th International Conference on Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*, 1996.

[34] P. Wadler. Theorems for free! In D. MacQueen, editor, *Proc. 4th International Conference on Functional Programming and Computer Architecture*, 1989.