Chapter 1

Alice Through the Looking Glass

Andreas Rossberg[1] Didier Le Botlan[1] Guido Tack[1]
Thorsten Brunklaus[1] Gert Smolka[1]

Abstract. We present Alice, a functional programming language that has been designed with strong support for *typed open programming*. It incorporates concurrency with data flow synchronisation, higher-order modularity, dynamic modules, and type-safe pickling as a minimal and generic set of simple, orthogonal features providing that support. Based on these mechanisms Alice offers a flexible notion of component, and high-level facilities for distributed programming.

1.1 INTRODUCTION

Software is decreasingly delivered as a closed, monolithic whole. As its complexity and the need for integration grows it becomes more and more important to allow flexible dynamic acquisition of additional functionality. Program execution is no longer restricted to one local machine only: with net-oriented applications being omni-present, programs are increasingly distributed across local or global networks. As a result, programs need to exchange more data with more structure. In particular, they need to exchange behaviour, that is, data may include code.

We refer to development for the described scenario as *open programming*. Our understanding of open programming includes the following main characteristics:

- *Modularity*, to flexibly combine software blocks that were created separately.
- *Dynamicity*, to import *and* export software blocks in running programs.
- *Security*, to safely deal with unknown or untrusted software blocks.
- *Distribution*, to communicate data and software blocks over networks.
- *Concurrency*, to deal with asynchronous events and non-sequential tasks.

[1]Programming Systems Lab, Saarland University, Saarbrücken, Germany;
Email: rossberg,botlan,tack,brunklaus,smolka@ps.uni-sb.de

Dynamic software blocks are usually called *components*.

Most practical programming languages today have not been designed with open programming in mind. Even the few that have been – primarily Java [9] – do not adequately address all of the above points. For example, Java is not statically type-safe, has only weak support for import/export, and rather clunky distribution and concurrency mechanisms.

Our claim is that only a few simple, orthogonal concepts are necessary to provide a flexible, *strongly typed* framework supporting all aspects of open programming. Components are a central notion in this framework, but instead of being primitive they can be derived from the following simpler, orthogonal concepts:

- *Futures*, which allow for light-weight concurrency and *laziness*.
- *Higher-order modules*, to provide genericity and encapsulation.
- *Packages*, to wrap modules into self-describing, dynamically typed entities.
- *Pickling*, to enable generic persistence and distribution.
- *Proxy functions*, to enable remote calls into other processes.

To substantiate our claim, we developed the programming language *Alice ML*, a conservative extension of Standard ML [13]. It is largely inspired by Oz [26, 14, 28], a relational language with advanced support for open programming, but lacking any notion of static typing. The aim of the Alice project is to systematically reconstruct the essential functionality of Oz on top of a typed functional language.

Alice has been implemented in the Alice Programming System [2], a full-featured programming environment based on a VM with just-in-time compilation, support for platform-independent persistence and platform-independent mobile code, and a rich library for constraint programming.

Organisation of the paper. This paper describes the design of the open programming features of Alice. Futures provide for concurrency (Section 1.2). Higher-order modules enhance modularity (Section 1.3). Type-safe marshalling is enabled by packages (Section 1.4). Components (Section 1.5) are built on top of these mechanism (Section 1.6). Distribution is based on components and RPCs (Section 1.7). We briefly discuss the implementation (Section 1.8), compare to some related work (Section 1.9), and conclude with a short outlook (Section 1.10).

1.2 FUTURES

Programs communicating with the outside world usually have to deal with non-deterministic, asynchronous events. Purely sequential programming cannot adequately handle such scenarios. Support for concurrency hence is vital.

Concurrency in Alice is based uniformly on the concept of *futures*, which has been mostly adapted from Multilisp [10]. A future is a transparent placeholder for a yet undetermined value that allows for implicit synchronisation based on data flow. There are different kinds of futures, which we will describe in the following sections. A formal semantics can be found in [16]. Futures are a generic mechanism for communication and synchronisation. As such, they are

comparatively simple, but expressive enough to enable formulation of a broad range of concurrency abstractions.

1.2.1 Concurrency

Future-based concurrency is very light-weight: any expression can be evaluated in its own thread, a simple keyword allows forking off a concurrent computation:

```
spawn exp
```

This phrase immediately evaluates to a fresh *concurrent future*, standing for the yet unknown result of *exp*. Simultaneously, evaluation of *exp* is initiated in a new thread. As soon as the thread terminates, the result globally replaces the future.

A thread is said to *touch* a future [7] when it performs an operation that requires the actual value the future stands for. A thread that touches a future is suspended automatically until the actual value is determined. This is known as *data flow synchronisation*.

If a concurrent thread terminates with an exception, the respective future is said to be *failed*. Any operation touching a failed future will cause the respective exception to be synchronously re-raised in the current thread.

Thanks to futures, threads give results, and concurrency can be orthogonally introduced for arbitrary parts of an expression. We hence speak of *functional threads*. For example, to evaluate all constituents of the application $e_1(e_2,e_3)$ concurrently, it is sufficient to annotate the application as follows:

```
(spawn e1) (spawn e2, spawn e3)
```

Functional threads allow turning a synchronous function call to a function f into an *asynchronous* one by simply prefixing the application with spawn:

```
val result = spawn f (x, y, z)
```

The ease of making asynchronous calls even where a result is required is important in combination with distributed programming (Section 1.7), because it allows for *lag tolerance*: the caller can continue its computation while waiting for the result to be delivered. Data flow synchronisation ensures that it will wait if necessary, but at the latest possible time, thus maximising concurrency.

Futures already provide for complex communication and synchronisation. Consider the following example:

```
val offset = spawn (sleep(Time.fromSeconds 120); 20)
val table = Vector.tabulate (40, fn i ⇒ spawn fib(i + offset))
```

The first declaration starts a thread that takes two minutes to deliver the value 20. The computation for the table entries in the second declaration depends on that value, but since the entries are computed concurrently, construction of the table can proceed without delay. However, the spawned threads will all block until offset is determined. Consecutive code can already access the table, but if it touches an entry that is not yet determined, it will automatically block.

Besides implicit synchronisation, Alice offers primitives for explicit synchronisation, including non-deterministic choice. They are sufficient to encode complex synchronisation with multiple events or timeouts [22].

1.2.2 Laziness

It has become a common desire to marry eager and lazy evaluation, and the future mechanism provides an elegant way to do so. While keeping eager application semantics, full support for laziness is available through *lazy futures*: the phrase

lazy *exp*

will not evaluate *exp*, but instead returns a fresh lazy future, standing for the yet unknown result of *exp*. Evaluation of *exp* is triggered by a thread first touching the future. At that moment, the lazy future becomes a concurrent future and evaluation proceeds as for concurrent futures.

In other words, lazy evaluation can be selected for individual expressions. For example, the expression (fn x ⇒ 5) (lazy raise E) will *not* raise E. A fully lazy evaluation regime can be emulated by prefixing *every* subexpression with lazy.

1.2.3 Promises and Locking

Functional threads and lazy evaluation offer convenient means to introduce and eliminate futures. However, the direct coupling between a future and the computation delivering its value often is too inflexible. *Promises* are a more fine-grained mechanism that allows for creation and elimination of futures in separate operations, based on three basic primitives:

type α promise
val promise : unit → α promise
val future : α promise → α
val fulfill : α promise × α → unit

A promise is an explicit handle for a future. Creating one by calling promise virtually states the assurance that a suitable value determining that *promised future* will be made available at some later point in time, fulfilling the promise. The future itself is obtained by applying the future function to the promise. A promised future is not replaced automatically, but has to be eliminated by explicitly applying the fulfill function to its promise. A promise may only be fulfilled once, any further attempt will raise the exception Promise.

Promises allow for partial and top-down construction of data structures with holes, e.g. a tail-recursive formulation of the append function [22]. However, they are particularly important for concurrent programming: for example, they can be used to implement streams and channels as lists with a promised tail. They also provide an important primitive for programming synchronisation.

For instance, Alice requires no primitive locking mechanisms, they can be fully bootstrapped from promises plus atomic exchange on references, a variant of the fundamental test-and-set operation [10]:

val exchange : α ref × α → α

As a demonstrating example, Figure 1.1 presents a function implementing mutex locks for synchronising an arbitrary number of argument functions.[2] The follow-

[2] Alice defines exp_1 finally exp_2 as syntactic sugar for executing a finaliser exp_2 after evaluation of exp_1 regardless of any exceptional termination.

```
(* mutex : unit → (α → β) → (α → β) *)
fun mutex () = let
    val r = ref ()                                  (* create lock *)
in
    fn f ⇒ fn x ⇒ let
        val p = promise ()
    in
        await (exchange (r, future p));             (* take lock *)
        f x
        finally fulfill (p, ())                     (* release lock *)
    end
end
```

FIGURE 1.1. Mutexes for synchronised functions

---

ing snippet illustrates its use to ensure non-interleaved concurrent output:

```
val sync = mutex ()
val f = sync (fn x ⇒ (print "x = "; print x; print "\n"))
val g = sync (fn y ⇒ (print y; print "\n"))
spawn f "A"; spawn g "B"
```

1.2.4 Modules and Types

Futures are not restricted to the core language, entire modules can be futures, too: module expressions can be evaluated lazily or concurrently by explicitly prefixing them with the corresponding keywords lazy or spawn. In Section 1.5 we will see that lazy module futures are ubiquitous as a consequence of the lazy linking mechanism for components.

The combination of module futures and dynamic types (Section 1.4) also implies the existence of *type futures*. They are touched only by the unpack operation (Section 1.4.1) and by pickling (Section 1.4.2). Touching a type generally can trigger arbitrary computations, e.g. by loading a component (Section 1.5).

1.3 HIGHER-ORDER MODULES

For open programming, good language support for modularity is essential. The SML module system is quite advanced, but still limited. Adopting a long line of work [5, 11, 12, 23], Alice extends it with higher-order functors and local modules (for dealing with packages, Section 1.4).

Less standard is the support for nested and abstract signatures: as in Objective Caml, signatures can be put into structures and even be specified abstractly in other signatures. Abstract signatures have received little attention previously, but they are interesting because they enable the definition of *polymorphic functors*, exemplified by a generic application functor:

```
functor Apply (signature S signature T) (F: S → T) (X: S) = F X
```

Polymorphic functors are used in the Alice library to provide certain functionality at the module level (see e.g. Section 1.7.3). More importantly, nested signatures

turn structures into a general container for all language entities, which is crucial for the design of the component system (Section 1.5). The presence of abstract signatures renders module type checking undecidable [12], but this has not turned out to be a problem in practice.

1.4 PACKAGES

When a program is able to import and export functionality dynamically, from statically unknown sources, a certain amount of runtime checking is inevitable to ensure the integrity of the program and the runtime system. In particular, it must be ensured that dynamic imports cannot undermine the type system.

Dynamics [15, 1] complement static typing with isolated dynamic type checking. They provide a universal type dyn of 'dynamic values' that carry runtime type information. Values of every type can be injected, projection is a complex type-case operation that dispatches on the runtime type found in the dynamic value. Dynamics adequately solve the problem of typed open programming by demanding external values to uniformly have type dyn. We see several hurdles that nevertheless prevented the wide-spread adoption of dynamics in practice: (1) their too fine level of granularity, (2) the complexity of the type-case construct, (3) the lack of flexibility with matching types.

We modified the concept of dynamics slightly: dynamics in Alice, called *packages*, contain *modules*. Projection simply matches the runtime *package signature* against a static one – with full respect for subtyping. Reusing module subtyping has several advantages: (1) it keeps the language simple, (2) it is flexible and sufficiently robust against interface evolution, and (3) it allows the programmer to naturally adopt idioms already known from modular programming. Moreover, packages allow modules to be passed as first-class values, a capability that is sometimes being missed from ML, and becomes increasingly important with open programming. A formal semantics for packages can be found in [21].

1.4.1 Basics

A package is a value of the abstract type package. Intuitively, it contains a module, along with a dynamic description of its signature. A package is created by injecting a module, expressed by a structure expressions *strexp* in SML:[3]

pack *strexp* : *sigexp*

The signature expression *sigexp* defines the package signature. The inverse operation is projection, eliminating a package. The module expression

unpack *exp* : *sigexp*

takes a package computed by *exp* and extracts the contained module, provided that the package signature matches the *target signature* denoted by *sigexp*. Statically,

[3] Since Alice supports higher-order modules, *strexp* includes functor expressions.

the expression has the signature *sigexp*. If the dynamic check fails, the pre-defined exception Unpack is raised.

A package can be understood as a statically typed first-class module as proposed by Russo [24, 23], wrapped into a conventional dynamic. However, coupling both mechanisms enables unpack to exploit subtype polymorphism, which is not possible otherwise, due to the lack of subtyping in the ML core language.

1.4.2 Pickling

The primary purpose of packages is to type dynamic import and export of high-level language objects. At the core of this functionality lies a service called *pickling*. Pickling takes a value and produces a transitively closed, platform-independent representation of it, such that an equivalent copy can be constructed in other processes. Since ML is a language with first-class functions, a pickle can naturally include code. Thanks to packages, even entire modules can be pickled.

One obvious application of pickling is *persistence*, available through two primitives in the library structure Pickle:

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

The save operation writes a package to a file of a given name. Any future occurring in the package (including lazy ones) will be touched (Section 1.2.1). If the package contains a local *resource*, i.e. a value that is private to a process, then the exception Sited is raised (we return to the issue of resources in Section 1.5.3). The inverse operation load retrieves a package from a file.

For example, we can write the library structure Array to disk:

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

It can be retrieved again with the inverse sequence of operations:

```
structure Array1 = unpack Pickle.load "array.alc" : ARRAY
```

Any attempt to unpack it with an incompatible signature will fail with an Unpack exception. All subsequent accesses to Array1 or members of it are statically type-safe, the only possible point of type failure is the unpack operation.

Note that the type Array1.array will be statically incompatible with the original type Array.array, since there is no way to know statically what type identities are found in a package, and all types in the target signature must hence be considered abstract. If compatibility is required, it can be enforced in the usual ML way, namely by putting *sharing constraints* on the target signature:

```
structure Array1 = unpack Pickle.load "/tmp/array.alc"
                        : ARRAY where type array = Array.array
```

1.4.3 Parametricity, Abstraction Safety and Generativity

By utilising dynamic type sharing it is possible to dynamically test for type equivalences. In other words, evaluation is no longer *parametric* [19]. For example,

```
functor F (type t) = unpack load file : (val it : t)
```

is a functor that behaves differently depending on what type t it is passed.[4]

Parametricity is important because it induces strong static invariants about polymorphic types, that particularly guarantee *abstraction* [19] and enable efficient type erasing compilation. On the other hand, packages enforce the presence of non-parametric behaviour. Alice thus has been designed such that the *core* language, where polymorphism is ubiquitous, maintains parametricity. Only the *module* level employs dynamic type information – module evaluation can be type-dependent. This design reduces the costs for dynamic types and provides a clear model for the programmer: only *explicit* types can affect the dynamic semantics.

The absence of parametricity on the module level still raises the question of how dynamic typing interferes with type abstraction. Can we sneak through an abstraction barrier by dynamically discovering an abstract type's representation? For instance:

```
signature S = (type t; val x : t)
structure M = (type t = int; val x = 37) :> S
structure N = unpack (pack M : S) : (S where type t = int)
val y = N.x + 1
```

Fortunately, the unpack operation will fail at runtime, due to a *dynamically generative* interpretation of type abstraction: with every abstraction operator :> evaluated, fresh type names are generated dynamically [20]. Abstraction safety is always maintained, even when whole modules cross process boundaries, because type names are globally unique.

Note that when fully dynamic type generativity is too strong to achieve proper type sharing between processes, the implementation of an abstract type can simply be exported as a pre-evaluated component (Section 1.5.4).

1.5 COMPONENTS

Software of non-trivial complexity has to be split into functional building blocks that can be created separately and configured dynamically. Such blocks are called *components*. We distinguish components from modules: while modules provide name spacing, genericity, and encapsulation, components provide physical separation and dynamic composition. Both mechanisms complement each other.

Alice incorporates a powerful notion of component that is a refinement and extension of the component system found in the Oz language [6], which in turn was partially inspired by Java [9]. It provides all of the following:

- *Separate compilation*. Components are physically separate program units.
- *Lazy dynamic linking*. Loading is performed automatically when needed.
- *Static linking*. Components can be bundled into larger components off-line.
- *Dynamic creation*. Components can be computed and exported at runtime.

[4] Alice allows abbreviating signatures sig ... end with (...), likewise for structures.

- *Type safety*. Components carry type information and linking checks it.
- *Flexibility*. Type checking is tolerant against interface changes.
- *Sandboxing*. Custom *component managers* enable selective import policies.

1.5.1 Introduction

A program consists of a – potentially open – set of components that are created separately and loaded dynamically. Static linking allows both to be performed on a different level of granularity by bundling given components to form larger ones. Every component defines a module – its *export*, and accesses an arbitrary number of modules from other components – its *imports*. Import and export interfaces are fully typed by ML signatures.

Each Alice source file defines a component. Syntactically, it is a sequence of SML declarations that is interpreted as a structure body, forming the export module. The respective export signature is inferred by the compiler. A component can access other components through a prologue of import declarations:

import *spec* **from** *string*

The SML signature specification *spec* in an import declaration describes the entities used from the imported structure, along with their type. Because of Alice's higher-order modules (Section 1.3), these entities can include functors and even signatures. The string contains the URL under which the component is to be acquired at runtime. The exact interpretation of the URL is up to the component manager (Section 1.5.3), but usually it is either a local file, an HTTP web address, or a virtual URL denoting local library components. For instance:

```
import structure Server : sig val run : (α→β) → (α→β) end
    from "http://domain.org/server"
```

For convenience, Alice allows the type annotations in import specifications to be dropped. In that case, the imported component must be accessible (in compiled form) during compilation, so that the compiler can insert the respective types.

1.5.2 Program Execution and Dynamic Linking

A designated *root* is the main component of a program. To execute a program, its root component is evaluated. Loading of imported components is performed *lazily*, and every component is loaded and evaluated only once. This is achieved by treating every cross-component reference as a lazy future (Section 1.2.2). The process of loading a component requested as import by another one is referred to as *dynamic linking*. It involves several steps:

1. *Resolution*. The import URL is normalised to a canonical form.
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.
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 respective future is failed with a standard exception that carries a description of the precise cause of the failure.

1.5.3 Component Managers and Sandboxing

Linking is performed with the help of a *component manager*, which is a module of the runtime library, similar to a class loader in Java [9]. It is responsible for locating and loading components, and keeping a table of loaded components.

In an open setting it is important to be able to deal with untrusted components. For example, they should not be given write access to the local file system. Like Java, Alice can execute components in a *sandbox*. Sandboxing relies on two factors: (1) all resources and capabilities a component needs for execution are *sited* and have to be acquired via import through a component manager (in particular, they cannot be stored in a pickle); (2) it is possible to create custom managers and link components through them. A custom manager can never grant more access than it has itself. A custom manager hence represents a proper sandbox.

1.5.4 Dynamic Creation of Components

The external representation of a component is a pickle. It is hence possible to create a component not only statically by compilation, but also dynamically, by a running Alice program. In fact, a pickle created with the Pickle.save function (Section 1.4.2) *is* a component and can be imported as such.

The ability to create components dynamically is particularly important for distribution (Section 1.7.3). Basically, it enables components to capture dynamically obtained information, e.g. configuration data or connections to other processes.

1.6 DECOMPOSING COMPONENTS

What *are* components? The close relation to concepts presented in previous chapters, like modules, packages and futures is obvious, so one might hope that there exists a simple reduction from components to simpler concepts. And indeed, components are merely syntactic sugar. Basically, a component defined by a sequence of declarations *dec* is interpreted as a higher-order procedure:

```
fn link ⇒ pack struct dec end : sigexp
```

where *link* is a reserved identifier and *sigexp* is the component signature derived by the compiler (the principal signature). In *dec*, every import declaration

```
import spec from s
```

is rewritten as[5]

```
structure strid = lazy unpack link s : sig spec end
open strid
```

[5] An open declaration merely affects scoping, it does not touch its argument.

```
val table = ref () : (url × package) list ref

fun link parent url = let
   val url' = resolve (parent, url)                    (* get absolute URL *)
   val p = promise ()
   val table' = exchange (table, future p)             (* lock table *)
in
   case List.find (fn (x,y) ⇒ x = url') table' of
      SOME package ⇒                                   (* already loaded *)
         (fulfill (p, table'); package)                (* unlock, return *)
    | NONE ⇒ let                                       (* not loaded yet *)
         val component = acquire url'                  (* load component *)
         val package = lazy component(link url')       (* evaluate *)
      in
         fulfill (p, (url',package) :: table');        (*  unlock  *)
         package
      end
end
```

FIGURE 1.2. The essence of a component manager

---

where *strid* is a fresh identifier. The expansion makes laziness and dynamic type checking of imports immediately obvious. Component acquisition is encapsulated in the component manager represented by the *link* procedure. Every component receives that procedure for acquiring its imports and evaluates to a package that contains its own export. The *link* procedure has type string → package, taking a URL and returning a package representing the export of the respective component. Imports are structure declarations that lazily unpack that package.

When a component is requested for the first time the *link* procedure loads it, evaluates it and enters it into a table. Figure 1.2 contains a simple model implementation that assumes existence of two auxiliary procedures *resolve*, for normalising URLs relative to the URL of the parent component, and *acquire* for loading a component. The parent URL is required as an additional parameter to enable the respective resolution. To achieve proper re-entrancy, the manager uses promises to implement locking on the component table (Section 1.2.3), and unlocks it *before* evaluating the component (hence the lazy application).

Giving this reduction of components, execution of an Alice program can be thought of as evaluation of the simple application

link *"."* *root*

where link is the initial component manager and *root* is the URL of the program's root component, resolved relative to the "current" location, which we indicate by a dot here.

1.7 DISTRIBUTION

Distributed programming can be based on only a few high-level primitives that suffice to hide all the embarrassing details of low-level communication.

1.7.1 Proxies

The central concept for distribution in Alice are *proxies*. A proxy is a mobile wrapper for a stationary function: it can be pickled and transferred to other processes without transferring the wrapped function itself. When a proxy function is applied, the call is automatically forwarded to the original site as a remote invocation, where argument and result are automatically transferred by means of pickling.

Proxies are created using the primitive

val proxy : $(\alpha \to \beta) \to (\alpha \to \beta)$

For instance, the expression proxy (**fn** x $\Rightarrow$ x+1) creates a simple proxy.

All invocations of a proxy are synchronous. In order to make an asynchronous call, it suffices to wrap the application using spawn (Section 1.2.1). This immediately returns a future that will be replaced by the result of the remote call automatically once it terminates.

Note that all calls through proxies are statically typed.

1.7.2 Client/Server

To initiate a proxy connection, a pickle must be first transferred between processes by other means. The Alice library supports two main scenarios. In the *client/server* model a client establishes a connection to a known server. A service offered by a server takes the form of a local component, which we refer to as the *mobile* component. Mobile components can be made available in a network through a simple transfer mechanism adapted from Mozart [14]. To employ it, a component is first packaged (Section 1.4), and then made available for download:

val offer : package $\to$ url

Given a package (Section 1.4), this function returns a URL, called a *ticket*, which publicly identifies the package in the network. A ticket can be communicated to the outside world by conventional means such as web pages, email, phone, or pigeons. A client can use a ticket to retrieve the package from the server:

val take : url $\to$ package

The package can then be opened using unpack, which dynamically checks that the package signature matches the client's expectations. Noticeably, this is the only point where a dynamic type check is necessary.

In order to establish a permanent connection, the mobile component must contain proxies. More complex communication patterns can be established by passing proxies back and forth via other proxies. They can even be forwarded to third parties, for instance to enable different clients to communicate directly.

1.7.3 Master/Slave

In an alternative scenario a *master* initiates shifting computational tasks to a number of *slave* computers. The library functor Run performs most of the respective

procedure: it connects to a remote machine by using a low-level service (such as ssh), and starts a slave process that immediately connects to the master. It evaluates a component and sends back the result.

```
functor Run (val host : string
             signature RESULT
             functor F : COMPONENT'MANAGER → RESULT) : RESULT
```

Run is a polymorphic functor (Section 1.3) with two concrete arguments: the name of the remote host, and a functor that basically defines a dynamic component (Section 1.5.4). It takes a structure representing a component manager, which can be used to access local libraries and resources on the remote host.

We illustrate a two-way connection by sketching the implementation of a distributed computation. A master process delegates computations to slaves that may dynamically request data from the master, by calling the proxy getData. For simplicity, we assume that the result of the computation is an integer.

```
(* Slaves use getData to acquire specific data. *)
val getData = proxy (fn key ⇒ return associated data)

(* Create a slave process on the given host. *)
fun computeOn (hostname, parameter) = let
    functor Start (CM : COMPONENT'MANAGER) = (val result = computation)
    structure Slave = Run (val host = hostname
                           signature RESULT = (val result : int)
                           functor F = Start)
in
   Slave.result
end
```

In the definition of result, the computation may use getData, which is a proxy to the master. It can access local libraries through the component manager CM. Note also that the computation is parameterised by the argument parameter. Then, the following code suffices to perform distributed computations in parallel.

```
val res1 = spawn computeOn ("machine1", parameter1)
val res2 = spawn computeOn ("machine2", parameter2)
```

The extended version of this paper contains a more extensive example [22].

1.8 IMPLEMENTATION

An implementation of Alice must meet two key requirements: dealing efficiently with the future-based concurrency model, and supporting open programming by providing a truly platform-independent and generic pickling mechanism.

The appropriate technology to achieve platform independence is to use a virtual machine (VM) together with just-in-time (JIT) compilation to native machine code. Futures and light-weight threads are implemented at the core of the system, making concurrency and data flow synchronisation efficient.

Pickling and unpickling are core VM services and available for all data in the store. They preserve cycles and sharing, which is vital for efficient pickling. Alice

also features a minimisation mechanism that merges all equivalent subgraphs of a pickled data graph [27].

Code is just heap-allocated data, it is subject to garbage collection and can be pickled and unpickled. The VM allows different internal types of code – e.g. JIT compiled native code and interpreted byte code – to coexist and cooperate. Different codes and interpreters can be selected on a per-procedure basis. Pickler and unpickler automatically convert between these internal codes and a well-defined external format. More details can be found in [22] and in a technical report [4].

Thanks to the pickling-based approach to distribution no complex distributed garbage collection is required. The only inter-process references are proxies. Distributed collection of proxies will be implemented in a future version of Alice.

1.9 RELATED WORK

There is numerous work and languages concerned with some of the issues addressed by Alice. A more detailed and extensive comparison can be found in [22].

Java [9] was the first major language designed for open programming. It is object-oriented and open programming is based on *reflection*, which allows other components to exploit type information constructively. We feel that general reflection is expensive, invites abuse, and in practice demands a rather limited type system, while packages avoid these issues. Concurrency and serialisation require considerable support from the programmer, code cannot be serialised. No structural type checks are performed when a class is loaded, subsequent method calls may cause a NoSuchMethodError, undermining the type system.

Scala [17] is a hybrid object-oriented/functional language on top of the Java infrastructure. It has a very powerful static type system with expressive abstraction mechanisms. For example, *bounded views* can specify a lower bound as well as an upper bound for an abstracted type. For comparison, a signature in Alice is a lower bound of the abstracted component. Still, concurrency and distribution are inherited from Java and suffer from the same shortcomings. In particular, the expressiveness of the type system does not carry over to dynamic typing, because Scala types are *erased* to Java types during compilation.

Oz/Mozart [26, 14, 28] inspired many of the concepts in Alice. Oz has very similar high-level support for concurrency, pickling and components. The Mozart distribution subsystem is more ambitious than Alice, supporting distributed state and futures, for the price of considerably higher complexity. Unlike Alice, Oz is based on a relational core language. It has no type system.

Acute [25] is probably closest in spirit to Alice. It is an experimental ML-based language for typed open programming that guarantees safe interaction between distributed applications, although the details of distribution are left to the programmer. Pickling allows resources to be dynamically rebound, no respective safety mechanism is built in. Components support versioning, but look more ad-hoc compared to Alice. Unlike in Alice, abstractions can be broken by explicit means, for the sake of flexible versioning.

JoCaml [8] is an innovative distributed extension of OCaml based on the Join

calculus. Concurrency and distribution uses channels and is more high-level than in Alice, allowing for complex declarative synchronisation patterns and thread migration. However, JoCaml is not open: pickling is restricted to monomorphic values stored on a global name server and there is no explicit component concept.

CML [18] is a mature concurrent extension of SML. It is based on first-class channels and synchronisation events, where synchronization has to be fully explicit. CML does not address distribution or other aspects of open programming.

Erlang [3] is a language designed for embedded distributed applications, applied successfully for industrial telecommunication. It is purely functional with an additional, impure process layer. Processes communicate over implicit, process-global message channels. Erlang is untyped, but has a rich repertoire for dealing with failure. It is not designed for open programming and does not directly support code mobility, but a distinctive feature is code update in active processes.

1.10 OUTLOOK

We presented Alice, a functional language for open programming. Alice provides a novel combination of coherent features to provide concurrency, modularity, a flexible component model and high-level support for distribution. Alice is strongly typed, incorporating a module-based variant of dynamics to embrace openness. It is fully implemented with a rich programming environment [2], and some small to medium-size demo applications have already been implemented with it, including a distributed constraint solver, a multi-player network game, and a simple framework for web scripting.

There is not yet a formal specification of the full language. Moreover, the implementation does not yet provide extra-lingual safety and security on the level of pickles. To that end, byte code *and* heap need to carry sufficient type information to allow creation of verifiable pickles. It was a deliberate decision to defer research on these issues to future work.

Acknowledgements We thank our former colleague Leif Kornstaedt, who co-designed Alice and also invested invaluable amounts of work into making it fly.

REFERENCES

[1] M. Abadi, L. Cardelli, B. Pierce, and D. Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1), 1995.

[2] The Alice Project. http://www.ps.uni-sb.de/alice, 2004. Homepage at the Programming Systems Lab, Universität des Saarlandes, Saarbrücken, Germany.

[3] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang, Second Edition*. Prentice-Hall, 1996.

[4] T. Brunklaus and L. Kornstaedt. A virtual machine for multi-language execution. Technical report, Universität des Saarlandes, Saarbrücken, Germany, 2002.

[5] D. Dreyer, K. Crary, and R. Harper. A type system for higher-order modules. In *Principles of Programming Languages*, New Orleans, USA, 2003.

[6] 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, Saarbrücken, Germany, 1998.

[7] C. Flanagan and M. Felleisen. The semantics of future and its use in program optimizations. In *Principled of Programming Languages*, San Francisco, USA, 1995.

[8] C. Fournet, L. Maranget, and A. Schmitt. *The JoCaml Language beta release*. INRIA, http://pauillac.inria.fr/jocaml/htmlman/, 2001.

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

[10] R. Halstead. Multilisp: A language for concurrent symbolic computation. *TOPLAS*, 7(4), 1985.

[11] X. Leroy. Applicative functors and fully transparent higher-order modules. In *Principles of Programming Languages*, San Francisco, USA, 1995. ACM.

[12] M. Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, USA, 1997.

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

[14] Mozart Consortium. The Mozart programming system, 2004. www.mozart-oz.org.

[15] A. Mycroft. Dynamic types in ML, 1983. Draft article.

[16] J. Niehren, J. Schwinghammer, and G. Smolka. Concurrent computation in a lambda calculus with futures. Technical report, Universität des Saarlandes, 2002.

[17] M. Odersky. *Programming in Scala*. École Polytechnique Féd. de Lausanne, 2005.

[18] J. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

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

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

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

[22] A. Rossberg, D. Le Botlan, G. Tack, T. Brunklaus, and G. Smolka. Alice through the looking glass (Extended mix). Technical report, Universität des Saarlandes, Saarbrücken, Germany, 2004. http://www.ps.uni-sb.de/Papers/.

[23] C. Russo. *Types for Modules*. Dissertation, University of Edinburgh, 1998.

[24] C. Russo. First-class structures for Standard ML. In *ESOP*, Berlin, Germany, 2000.

[25] P. Sewell, J. J. Leifer, K. Wansbrough, M. Allen-Williams, F. Z. Nardelli, P. Habouzit, and V. Vafeiadis. Acute: High-level programming language design for distributed computation. Technical Report RR-5329, INRIA, 2004.

[26] G. Smolka. The Oz programming model. In *Computer Science Today*, volume 1000 of *LNCS*. Springer-Verlag, Berlin, Germany, 1995.

[27] G. Tack. Linearisation, minimisation and transformation of data graphs with transients. Diploma thesis, Programming Systems Lab, Universität des Saarlandes, 2003.

[28] P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004.