Alice ML Through the Looking Glass (Draft)

Andreas Rossberg Didier Le Botlan Guido Tack
Thorsten Brunklaus Gert Smolka

December 21, 2004

intellecttm

Abstract: We present the Alice ML programming language, a functional language that has been designed with strong support for *typed open programming*. Alice incorporates concurrency with data flow synchronisation, higher-order modularity, dynamic modules, and type-safe pickling. Based on these mechanisms it provides a flexible notion of component, and high-level facilities for distributed programming.

0.1 INTRODUCTION

Software is decreasingly delivered as a closed, monolithic whole. As complexity and integration of software grows it becomes more and more important to allow flexible dynamic acquisition of additional functionality. Also, program execution is no longer restricted to one local machine only. With the Internet having gone mainstream and net-oriented applications being omni-present, programs become increasingly distributed across local or global networks. As a result, programs need to exchange larger amounts of data, and the exchanged data is of growing complexity. In particular, programs 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*, the ability to flexibly combine software blocks that were created separately.
- *Dynamicity*, the ability to import *and* export software blocks in running programs.
- *Safety*, the ability to safely deal with unknown or untrusted software blocks.
- *Distribution*, the ability to communicate data and software blocks over networks.
- *Concurrency*, the ability to deal with asynchronous events and perform tasks non-sequentially.

Software blocks intended for dynamic combination 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 [GJS96] – 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.

The programming language Oz [Smo95, Moz04, VH04] has the most advanced support for open programming so far, with an expressive component model and high-level concurrency and mobility, but lacking static typing. The aim of the Alice project [Ali04a] is to systematically reconstruct the essential functionality of Oz on top of a simple and well-understood, typed functional core language.

The result is the programming language *Alice*, a conservative extension of Standard ML [MTHM97]. It adds only few simple, orthogonal high-level concepts that together form a coherent framework supporting all aspects of *typed* open programming. A central notion in this framework are components. However, they are not a primitive concept but actually derived from several simpler concepts. Alice is strongly typed and supports typeful programming [Car91] with abstraction safety (encapsulation) preserved at all times.

The Alice ML language has been implemented in the Alice Programming System, a fully-featured programming environment based on a novel VM with just-in-time compilation, and noticeably 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 concepts introduced in Alice to enable open programming. In order to support concurrency and laziness, Alice provides the concept of futures, described in Section 0.2. A higher-order extension to the SML module system enhances modularity and is briefly sketched in Section 0.3. Type-safe import and export is realized by introducing dynamically typed modules called packages that can be *pickled*, as described in Section 0.4. In Section 0.5 we introduce the component model and show in Section 0.6 how it can be decomposed. Communication through a network based on components is considered in Section 0.7. We illustrate most of the features of Alice in Section 0.8 considering the case study of a distributed solver for constraint programming. We discuss related work in Section 0.10 and conclude in Section 0.11.

0.2 FUTURES

Programs communicating with the outside world usually have to deal with non-deterministic events, at arbitrary points in time. Purely sequential programming cannot adequately handle such scenarios. Alice hence has been designed as a concurrent language throughout.

Concurrency in Alice is based uniformly on the concept of *futures* [NSS02], which has been mostly adapted from Multilisp [Hal85]. A future is a transparent place-holder 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. Futures are a generic mechanism for communication and synchronisation. As such, they are comparatively simple, but expressive enough to enable formulation of all kinds of communication abstractions or explicit synchronisation patterns.

0.2.1 Concurrency

In Alice, any expression can be evaluated in its own thread. A simple expression form allows forking off concurrent computation for evaluating an expression:

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 with a value *v*, the future will be replaced by *v*.

A thread is said to *touch* a future [FF95] 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 future's 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. 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 e_1**) (spawn** e_2**, spawn** e_3**)**

Hence, threads blend perfectly into the "everything is an expression" philosophy of functional programming. For that reason, we call them *functional threads*.

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 0.7), 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 provide for many-to-one 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 individual threads computing its content will all block until `offset` is determined. Consecutive code can access the vector without caring about the progress of the threads. If evaluation depends on a value that is not yet determined, it will automatically block as long as required.

Besides implicit synchronisation, Alice offers library primitives for explicit synchronisation:[1]

```
val await : 'a -> 'a
val awaitEither : 'a * 'b -> ('a,'b) alt
```

The function `await` is a future-strict variant of the identity function: if applied to a future, it blocks until the future has been replaced by a proper value. A straightforward abstraction using this function is the barrier function that implements a *join point* by computing a list of functions concurrently and waiting for all of them to terminate:

```
fun barrier fs = map await (map (fn f => spawn f ()) fs)
```

The function `awaitEither` implements *non-deterministic choice*: given two futures it blocks until at least one has disappeared. It is sufficient as a primitive to encode complex synchronisation with multiple events. As a simple example, consider an abstraction for waiting with time-out:

```
fun awaitTimeout time x =
    case awaitEither (x, spawn sleep time) of
        FST x => x
      | SND _ => raise Timeout
```

0.2.2 Laziness

SML is an eager language. While eager evaluation has advantages (e.g. making algorithmic complexity more predictable), certain algorithms are expressed more elegantly or more efficiently with *lazy evaluation*. It has become a common desire to marry eager and lazy evaluation, and the future mechanism provides an elegant way to do so. Alice keeps eager semantics the default behaviour, but full support for laziness is available through a lazy variant of futures. A *lazy future* is introduced analogously to a concurrent one: 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, associated with a fresh thread performing the computation. Evaluation proceeds as for concurrent futures.

In other words, lazy evaluation can be selected for individual expressions. A fully lazy evaluation regime can be emulated by prefixing *every* subexpression with the `lazy` keyword, but usually only few strategic annotations are necessary. In order to support the definition of lazy functions conveniently, Alice extends the definition of SML's sugared function declaration syntax with support for the `lazy` keyword.

0.2.3 Promises

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, because it demands an initial commitment to the way the information is obtained. Alice hence offers *promises* as a more fine-grained mechanism that allows for creation and elimination of futures in separate operations.

Promises are available through a library structure named `Promise`, with the following signature:

```
type 'a promise
exception Promise
val promise : unit -> 'a promise
val future  : 'a promise -> 'a
val fulfill : 'a promise * 'a -> unit
```

[1] The type `alt` is defined in the Alice library as: `datatype ('a,'b) alt = FST of 'a | SND of 'b`

```
fun append (l1, l2) =
let
    fun iter (p, nil)   = fulfill (p, l2)
      | iter (p, x::xs) = let val p' = promise ()
                          in fulfill (p, x::future p'); iter (p', xs) end
    val p = promise ()
in
    iter (p, l1); future p
end
```

FIGURE 1. Tail-recursive append with promises

A promise is an explicit handle for a future. It virtually states the assurance that a suitable value determining the future will be made available at some later point in time, fulfilling the promise. The *promised future* itself is obtained by applying the `future` function to the promise. It largely behaves like a concurrent future, in particular by allowing data flow synchronisation, except that it 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 the partial and top-down construction of data structures with holes, as exemplified by the tail-recursive formulation of the append function shown in Figure 1. 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, and they provide an important primitive for programming synchronisation, as we will see in the next section.

0.2.4 Thread Safety

When multiple threads share mutable state it is imperative to synchronise access, usually by forms of locking on critical sections. Alice has no locking mechanism built into the language proper. Instead it provides the necessary primitives that enable providing synchronisation mechanisms as library abstractions. A crucial primitive is the atomic exchange operation on references, a variant of the fundamental test-and-set operation [Hal85]:

```
val exchange : 'a ref * 'a -> 'a
```

The exchange operation is sufficient to bootstrap basic synchronisation mechanisms. Without further primitives their implementation would often require forms of polling, though. Along with futures and promises such polling can be circumvented.

As a simple example demonstrating this, Figure 2 presents a higher-order function implementing mutex locks for synchronising an arbitrary number of functions.[2] The following snippet illustrates its use to synchronise concurrent communication to standard output, by preventing execution of `f` and `g` to be interleaved:

```
val mutex = mkMutex ()
val f = mutex (fn x => (print "x = "; print x; print "\n"))
val g = mutex (fn y => (print y; print "\n"))
spawn f "A"; spawn g "B"; spawn f "C"
```

0.2.5 Modules and Types

Futures are not restricted to the core language, entire modules can be futures, too. In particular, module expressions can be evaluated lazily or concurrently, by explicitly prefixing them with the corresponding keywords `lazy` or `spawn`. More importantly, we will see in Section 0.5 that lazy module futures are ubiquitous as a consequence of the lazy linking mechanism for Alice components.

The combination of module futures and dynamic types (Section 0.4) also implies the existence of *type futures*. They are touched only by the `unpack` operation introduced in Section 0.4.1 and by pickling (Section 0.4.2). Touching a type generally can trigger arbitrary computations, e.g. by loading a component (Section 0.5).

[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, similar to the `try`...`finally`... in other languages.

```
(* mkMutex : unit -> ('a -> 'b) -> ('a -> 'b) *)
fun mkMutex () =
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 2. Mutexes for synchronised functions

0.3 HIGHER-ORDER MODULES

For open programming, good language support for modularity is vital. The SML module system is quite advanced, but still limited by its restriction to first-order functors (parameterised modules) and its stratified design (modules cannot be declared locally). Following a long line of work on higher-order modules [DCH03, Ler95, Lil97, Rus98], Alice extends the SML module system in three ways:

- *Higher-order functors*. Functors can be arbitrarily nested and parameterised over other functors.
- *Nested and abstract signatures*. Signatures can be wrapped in structures and be specified abstractly.
- *Local modules*. All module entities can be defined within core `let` expressions.

Local modules are important for dealing with packages (Section 0.4), while the other two extensions allow more general forms of abstraction. In particular, they turn structures into a general container for all language entities, which is crucial for the design of the Alice component system (Section 0.5).

Abstract signatures have received little attention in literature, but they are interesting because they enable the definition of *polymorphic functors*, exemplified by a general 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. We will see an example of this in Section 0.8. The presence of abstract signatures renders module type checking undecidable [Lil97], but this has not turned out to be a problem in practice.

Space consideration preclude a detailed presentation of the Alice module language, but the knowledgeable reader will realise that the above extensions turn it into a higher-order functional language that closely mirrors the module language of OCaml [Ler03], except that functors are not applicative (in particular, OCaml type checking is equally undecidable).

0.4 PACKAGES

When a program is supposed to be open – i.e. able to import and export data and functionality dynamically, from statically unknown sources – then a certain amount of runtime checking is required to ensure the integrity of the program and the runtime system. In a language with a strong static type system, like ML, it particularly must be ensured that dynamic imports cannot undermine the type system. How can the tension be resolved?

Dynamics as a way to complement static typing with dynamic type checking were first proposed by Mycroft [Myc83] and refined by Abadi, Cardelli, Pierce & Plotkin [ACPP91, ACPR95]. Intuitively, they introduce a universal type `dyn` of 'dynamic values' that carry runtime type information. Values of every type can be injected into this type. Projection is a complex type-case operation that dispatches on the runtime type found in the dynamic value.

Dynamics maintain most properties of the static type system by isolating dynamic typing, and they solve the problem of open programming by demanding external values to uniformly have type `dyn`. We see several hurdles that nevertheless prevented the adoption of dynamics in practice: (1) the improper level of granularity they provide for wrapping objects, (2) the complexity of the type-case construct, particularly with respect to polymorphic types, (3) the lack of flexibility with matching types, which makes them fragile against interface changes.

For Alice we hence modified the concept of dynamics slightly: instead of encapsulating core values, dynamics in Alice – called *packages* – contain *modules*. Projection simply matches the runtime *package signature* against a statically specified one – with full respect for subtyping. Reusing module subtyping instead of type matching and dispatch has several advantages: (1) it keeps the language simple, (2) it is flexible, and (3) it allows the programmer to naturally adapt 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 becoming increasingly important with open programming. In Section 0.5 we will see that packages actually are expressive enough to form the basis of a sophisticated component system.

0.4.1 Basics

Packages are the exclusive means for integrating dynamic typing into Alice. A package is a value of the abstract type `package`. Intuitively, it contains a module, along with a dynamic description of its signature.

There are only two basic operations on packages. A package is created by injecting a module, expressed by a structure expressions *strexp* in SML[3] into the type `package`:

`pack` *strexp* `:` *sigexp*

The signature expression *sigexp* defines the package signature. Of course, the module expression *strexp* must statically match this 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, the expression has the signature *sigexp*. If the dynamic check fails, the pre-defined exception `Unpack` is raised.

0.4.2 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 that is transferable to other processes, where an equivalent copy of the original value can be constructed. Since ML is a language with first-class functions, a pickle can naturally include higher-order data, i.e. closures and code. Due 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 takes a file name and a package and writes it to that file. Any eventual future occurring in the package will be touched. If the package contains a local *resource*, i.e. a value that is private or meaningless outside the process, then the exception `Sited` is raised (we return to the issue of resources in Section 0.5.3). The inverse operation `load` takes a file name and retrieves a package from the respective file.

For example, we can write the library structure `Array` to disk, using the following idiomatic code:

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

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

```
structure Array1 = unpack Pickle.load "/tmp/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, no further checks are required. The only possible point of type failure is the `unpack` operation.

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

0.4.3 Dynamic Type Sharing

Note that the type `Array1.array` from the example in the previous section 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
```

The constraint effectively expresses *dynamic type sharing*. By restricting the target signature we ensure static compatibility, but of course we also preclude successful loading of non-standard implementations of arrays. Much like for programming with functors, it depends on the application how much sharing is required. Note that dynamic type sharing can be employed for *typeful programming* [Car91] with dynamic types, when packages themselves contain the implementation of abstract types.

0.4.4 Parametricity

By utilising dynamic type sharing it is possible to dynamically test for type equivalences. In other words, evaluation is not *parametric* [Rey83] in Alice. For example, the functor

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

behaves differently depending on what type it is passed.

Parametricity has important advantages:

- *Theorems for free* [Wad89]. Polymorphic types state strong invariants about terms, which allow deriving a variety of useful laws.
- *Abstraction* [Rey83, MP88]. It is possible to achieve encapsulation solely by abstracting over types.
- *Type erasure*. Programs can be compiled and executed without maintaining costly type information at runtime.

Let us defer the second point to the next section and focus on the other two. Here, losing parametricity is particularly problematic for the core language, where polymorphism is ubiquitous. Alice thus has been designed such that the core language maintains parametricity. As long as the `package` type is excluded, all laws derived by parametricity hold for polymorphic functions. Polymorphic expressions do not depend on any type information and their use is no more costly than in plain SML.

Only on the module level dynamic type information is required – the evaluation of module expressions can be type-dependent. This design reduces the costs for dynamic types and provides a clear model for the programmer: the only type information relevant to the dynamic semantics (and its costs) are *explicitly declared* in the program. Implicit type information on the core level is not relevant.

The restriction of type-driven evaluation to the module language is not without drawback. The fact that ordinary evaluation cannot depend on types significantly restricts the expressive power of the language. In our experience however, there is no strong need for more liberal dynamic typing. Thanks to higher-order and local modules (Section 0.3) it is often possible to lift the procedure in question to a functor. Moreover, packages provide a systematic way to work around the restriction: because they can carry modules and hence explicit types, they can be abused to pass dynamic type information to core functions if really required.

0.4.5 Abstraction Safety and Generativity

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? Viz:

```
signature S      = sig type t; val x : t end
structure M :> S = struct type t = int; val x = 37 end
structure M'     = unpack (pack M : S) : (S where type t = int)
val y = M'.x + 1
```

Fortunately, the `unpack` operation will fail at runtime. This behaviour is achieved by a *dynamically generative* interpretation of type abstraction in Alice: with every abstraction operator `:>` evaluated, fresh type names are generated dynamically [Ros03]. Abstraction safety is maintained even across process boundaries, because type names are globally unique. There is no way within the language to break abstractions, even when their *implementations* are exported.

The generative semantics of abstract types implies that execution of the same implementation of an abstraction will create incompatible instances between runs of the same program, or between different processes loading it. However, this is not a severe restriction, because the availability of module-level pickling enables us to execute it only once and then share the same *evaluated* instance of the abstraction between processes. As we will see in Section 0.5.4, pickling a module actually creates a proper component that is interchangeable with components generated by the compiler.

0.5 COMPONENTS

Software of non-trivial complexity can neither be developed nor deployed as a monolithic block. To keep the development process manageable, and to allow flexible installation and configuration, software has to be split into functional building blocks that can be created separately and configured dynamically. Such building 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. It is the component system that enables closing over free references in a module implementation and hence turning it into a self-enclosed entity.

Alice incorporates a powerful notion of component, that is a refinement and extension of the component system found in the Oz language [DKSS98], which in turn was partially inspired by Java [GJS96]. It provides all of the following:

- *Separate compilation*. Components are physically separate program units that can be translated independently.
- *Lazy dynamic linking*. Loading is performed automatically when needed, by the run-time system.
- *Static linking*. Optionally, components can be bundled into larger components off-line.
- *Dynamic creation*. Programs can compute and export components dynamically.
- *Type safety*. Components carry type information, and linking involves dynamic type checks.
- *Flexibility*. Type checking is based on signature matching and is thus tolerant against interface changes.
- *Sandboxing*. Configurable *component managers* enables setting up custom import policies.

0.5.1 Introduction

Components are the unit of compilation as well as the unit of deployment in Alice. 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* – to realise this definition. Both, import and export interfaces, are fully typed by ML signatures. Each Alice source file defines, and is compiled into, a component. Syntactically, a component definition primarily 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 by importing from them. Syntactically, import is performed by a declaration of the form

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

All import declarations have to appear before the first proper declaration in the component definition. 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 0.3), these entities can include functors and even signatures. All identifiers bound in the specification are in scope in the rest of the program. 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 the component manager and its resolver (Section 0.5.3), but usually it either is a local file, an HTTP web address, or a virtual URL denoting local library components. For example, the following are valid imports:

```
import structure Pickle : PICKLE from "x-alice:/lib/system/Pickle"
```

```
import structure Server :
  sig val run : ('a->'b) -> ('a->'b) 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 from its signature. For example, the previous import declarations could be written as

```
import structure Pickle from "x-alice:/lib/system/Pickle"
import structure Server from "http://domain.org/server"
```

This is particularly needed for large modules, where repeating the signature would be tedious. As an additional service, the compiler automatically thins implicit signatures to the minimum content required by the compiled component, making it maximally robust against eventual changes in parts of an interface that are not accessed.

0.5.2 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* – components are only loaded when needed. This is achieved by treating every cross-component reference – i.e. a reference to an imported entity in a component – as a lazy future (Section 0.2.2). Every component is loaded and evaluated only once.

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 has not been loaded already from the URL, it is done so.
3. *Evaluation*. If the component has been loaded afresh, it is evaluated.
4. *Type Checking*. The component's export signature is matched against the respective import signature.

Each of the steps except for the first 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 the standard exception `Component.Failure` that carries a description of the precise cause of the failure.

0.5.3 Component Managers and Sandboxing

Linking is performed with the help of a *component manager*, which is a module of the runtime library. A component manager is similar to a class loader in Java [GJS96]. It is responsible for locating and loading components, and keeping a table of components already loaded by a process.

In an open setting it is important to be able to deal with untrusted components. For example, an untrusted component should not be given write access to the local file system. Alice inherits the approach taken by Java, where components can be executed in a *sandbox*. Sandboxing relies on two factors: (1) all resources and capabilities a component needs for execution have to be acquired via import through a component manager (in particular, they cannot be taken from a pickle, because they cannot be pickled); (2) it is possible to create custom managers and explicitly link components through them. The implementation of a custom manager can only use capabilities provided by its 'parent' manager, so it can never grant more access than it has itself. A custom manager hence represents a proper sandbox.

0.5.4 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 the compiler, but also dynamically, by a running Alice program. In fact, a pickle created with the `Pickle.save` function (Section 0.4.2) *is* a component and can be imported as if it had been created through compilation. The ability to create components dynamically is particularly important for distributed programming, as we will see in Section 0.8. Basically, it allows the creation of components that *close* over dynamically obtained information, e.g. configuration data or connections to other processes.

```
val table = 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 in table *)
         (fulfill (p, table'); package)                (* release lock, return *)
    | NONE =>                                          (* new *)
      let
         val component = acquire url'                  (* load component *)
         val package = lazy component (link url')      (* evaluate component *)
      in
         fulfill (p, (url', package) :: table');       (* release lock *)
         package
      end
end
```

FIGURE 3. The essence of a component manager

0.6 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 of the structure). In *dec*, every import declaration

```
import spec from s
```

is rewritten as

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

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 component identified by the URL. Imports are then simply structure declarations that lazily unpack that package.

The `link` procedure represents the core of a component manager. Its job is locating components and keeping a table of loaded components. If a component is requested for the first time it is loaded, evaluated and entered into the table. Figure 3 contains a simple model implementation of such a procedure, 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 from a normalised URL. The procedure takes the parent URL as an additional parameter in order to allow the respective resolution. Note that the manager uses promises to implement locking on the component table, to achieve proper reentrancy (Section 0.2.4).

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 simple dot here.

0.7 DISTRIBUTION

This section deals with distributed programming in Alice. Noticeably, only a few high-level primitives suffice to hide all the embarrassing details of low-level communication. We first explain how to set up a client-server connection[4]. Then, we consider a distributed scenario of computation with master and slave processes.

0.7.1 Client-Server

In our model, a client first *establishes* a connection to a server, then *uses* that connection to exchange values.

Establishing a connection The purpose of a server is to offer a service. In Alice, this service 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 [Moz04]. To employ it, a component is first packaged (see packages, Section 0.4), and then made available for download using the following primitive of the `Remote` library:

```
val offer : package -> string
```

Given a package (Section 0.4), the function returns a URL, called a *ticket*, which publicly identifies the package in the network. It is communicable to the outside world by any possible means such as a link on a web page, email, by telephone, or pigeons. The client can use that ticket to fetch the package with `Remote.take`:

```
val take : string -> package
```

This primitive expects a valid ticket and retrieves the corresponding package from the server. The package can then be opened using `unpack`, which dynamically checks that the package signature matches the expected signature. As a result, the downloaded module is available to the rest of the program. Noticeably, this is the only point in the program where a dynamic type check is necessary. Indeed, from now on, static type checking suffices to ensure that all communication is dynamically well-typed.

Two-way connection So far, we can transfer a component from a server to a client. This component may contain code, types and signatures. However, we have not shown yet how the client can communicate values back to the server. This is achieved by implementing some functions in the mobile component as *proxies*. A proxy is a mobile, remote wrapper for a stationary function: when applied on a site, the call is automatically forwarded to the original site as remote procedure call. Arguments and results are automatically transferred between sites by means of pickling. Proxies are created using the following primitive of the `Remote` library:

```
val proxy : ('a -> 'b) -> ('a -> 'b)
```

As a higher-level abstraction, a polymorphic functor (Section 0.3) `Remote.Proxy` is available to transform all functions of a given structure into proxies.

As an example, the expression `proxy (fn x => x+1)` creates a synchronous proxy. All its invocations on the client side will result in a synchronous remote function call. In order to create asynchronous proxies, it suffices to wrap the definition using `spawn`. This immediately returns a future that will be bound to the result of the remote call once it has finished.

Using a Connection We see that using a connection simply consists in calling a proxy function. Furthermore, no dynamic type check is necessary once the connection is established since all remote calls through proxies are necessarily well-typed. This makes a significant difference: if transmission were explicit through `offer` and `take`, all data would have to be dynamically type-checked on unpacking.

In the setting we have described, only the client can call a proxy on the server side. In order to get a more symmetrical connection, it is still possible to transmit a client-side proxy to the server. Indeed, since functions are first-class values, such a proxy can perfectly be shipped as the argument of some server-side function. Such an example of a symmetrical two-way connection is given in Section 0.8, where we describe a distributed constraint solver.

[4]Where "connection" means a logical connection, and is definitely not a permanent connection at the network level.

0.7.2 Distribution: master-slave

In the client-server setting, clients choose independently to connect to a known server. In contrast, in an alternative distribution scenario a *master* initiattes shifting computational tasks to a number of *slave* computers. The Alice `Remote` library provides a functor `Execute` that automatically performs most of the respective procedure. More precisely, it connects to a remote machine by using a low-level service (such as `ssh`), and starts a process that immediately connects to the master. This slave then evaluates a component and sends the result back to the master.

```
functor Execute (val host : string
                 signature RESULT
                 functor Start (CM: COMPONENT_MANAGER) : RESULT) : RESULT
```

`Execute` is a polymorphic functor (Section 0.3) that expects two concrete arguments: the name of the remote host, and a functor `Start` to be executed remotely. `Start` is basically a component that will be evaluated on the slave machine. It expects as an argument a structure representing the local component manager (see Section 0.5.3) that can be used to access local libraries and resources.

By using proxies defined *outside* of the functor `Start`, and by creating proxies *inside* the functor and exported in the `RESULT` signature, a two-way communication is immediately established. We illustrate the use of `Execute` with a practical example in the next section.

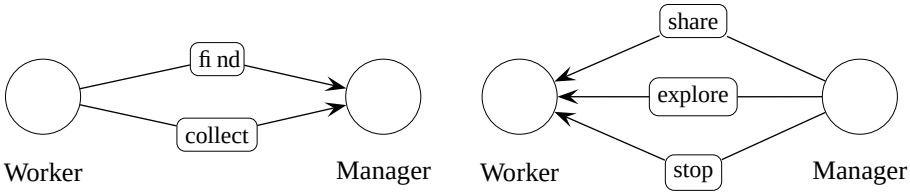
0.8 DISTRIBUTED SOLVER

In this section, we illustrate the main features of Alice by showing the implementation of a distributed application, namely a distributed solver for constraint programming.

Description In the context of constraint programming, a solver is a program that explores a tree in order to find the solutions of a given constraint program. Nodes of the tree represent choice points. Leaves represent failures (previous choices are inconsistent) or solutions. From a logical point of view, searching amounts only to traversing a tree and asking the status of each node.

In a distributed setting, each remote computer (*worker*) explores a different subtree. The interesting information, that is the solutions, are transmitted back to a *manager*. The manager also organises the search. In the following, we focus on the distribution aspect. More information about the search itself can be found in [TL04], which contains a formalisation of the underlying abstractions.

The interface between the workers and the manager can be represented as follows (see Chapter 9 of [Sch02] for a more detailed presentation of this interface):



The `find` message is sent by a worker that is idle. It requests a job, that is, the path of a subtree of the search tree that remains to be explored. When a worker encounters a solution, it sends the `collect` message, along with the solution, to the manager. The `share` message is used by the manager to ask a worker whether it can give away a subtree that remains to be explored. The worker is required to answer either negatively, or positively by providing the path associated to the corresponding unexplored subtree (hence the type `unit -> path option` below), The manager commands a worker to explore a subtree by sending the `explore` message along with the corresponding path. The `stop` message is used to stop the worker when the search is finished (for example when the first solution was found in a one-solution search).

Implementation The implementation of the distributed search engine consists basically of two components: the Manager and the Worker. The manager keeps a list of the workers it is connected to. Each worker has the same interface:

```
signature WORKER =
sig
```

```
    val share   : unit -> path option
    val explore : path -> unit
    val stop    : unit -> unit
end
```

The manager creates workers by using the `Execute` functor. Before that, the proxies of the manager interface have to be defined. As explained above, the manager interface provides two functions: `find` and `collect`:

```
val find    = proxy (fn () => body)
val collect = proxy (fn sol => body)
```

The definition of a worker also takes place in the manager. Basically, we create a dynamic component (Section 0.5.4). The worker interface provides three function proxies: `share`, `explore`, and `stop`. Additionally, the library used for constraint solving, named *Gecode*, is a native library that cannot be pickled, it is sited (Section 0.4.2). Thus, each worker needs to acquire this local library by using the component manager; this is the purpose of the call to the `Link` functor the manager provides.

```
functor StartWorker (CM : COMPONENT_MANAGER) =
struct
  structure Gecode = CM.Link (val url = "x-alice:/lib/Gecode"
                              signature S = GECODE)
  val share   = proxy (fn () => find some unexplored subtree)
  val explore = proxy (fn path => explore the given subtree)
  val stop    = proxy (fn () => OS.Process.exit OS.Process.success)
end
```

In the implementation of `explore`, two special cases are interesting. If the exploration is finished, the worker asks for some more work by calling `find ()`. If a solution `sol` is found, it is transmitted to the manager by performing an asynchronous call `spawn collect sol`. In both cases, a remote procedure call is automatically done since the corresponding functions `find` and `collect` are proxies.

In order to create distant workers, the manager uses the functor `Remote.Execute` repeatedly.

```
structure Worker = Remote.Execute (val host         = host – name
                                   signature RESULT = WORKER
                                   functor Start    = StartWorker)
```

Each newly created worker is stored in the worker list. Then, the search starts by sending the root path of the search tree to the first worker of the list, then ask it for some work to give to other workers.

Concurrency The manager must be able to handle concurrent requests from workers. For example, the `collect` message stores the given solution in a list, which is protected using a locking mechanism (Section 0.2.4). Noticeably, the list of collected solutions is returned immediately when the search engine starts, in the form of a future. The list is built concurrently while solutions are sent to the manager.

0.9 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 state-of-the-art technique for platform independence is to use a virtual machine together with just-in-time (JIT) compilation to native machine code. Futures and lightweight threads are implemented at the core of the system, making thread creation and data flow synchronisation as efficient as possible. More detailed implementation notes can be found in a technical report [BK02].

0.9.1 Pickling

The Alice system provides a generic pickling and unpickling mechanism that can be applied to all objects in the store. The pickler takes a store object and transforms the subgraph of objects reachable from there into a platform

independent external representation (a pickle) that is suitable for transportation over a network or storage in a file. The unpickler recreates a copy of the original subgraph from a pickle. Pickling and unpickling preserve the structure of the original graph, including cycles and sharing of nodes. The low-level pickling service thus implements the transitive closure semantics sketched in Section 0.4.2.

In order to support a platform independent external and an efficient internal representation of data, the system offers a generic transformation mechanism: in a pickle, certain data is marked to be transformed on unpickling. It will be converted to an internal, possibly platform dependent format. The internal data is also marked and preserves enough information to recreate an external, platform independent version during pickling. Floating point numbers, for instance, have different efficient internal representations on different platforms, but their external, pickled representation must be uniform.

The pickler features a minimisation mechanism that removes redundany by maximising sharing of equivalent subgraphs of objects in a pickle. This produces compact and efficient pickles.

0.9.2 Codes and interpreters

Code is just data in the store, it is subject to garbage collection and can be pickled and unpickled. While there is exactly one external representation (called *Alice Abstract Code*), Alice features several internal types of code, and there may be more than one interpreter (or execution unit) for each type. Different codes and different interpreters can coexist and cooperate, and selected on a per-procedure level.

The advantage of this generic model is that for different purposes, different internal codes can be employed: usually, JIT compiled native code delivers the best execution speed. For code only executed once, direct interpretation of the abstract code is superior. A debugging interpreter can make use of additional annotations and check more invariants – the user can decide to run just a certain function in debug mode, with the rest of the system being JIT compiled.

JIT compilation builds on only two mechanisms: the transformation mechanism of the unpickler, and lazy futures. Unpickling transforms abstract code into a lazy future, which triggers JIT compilation of the code on request and results in the respective native machine code. Thus, native code is only created when actually needed.

JIT compilation is the adequate way to deal with a future-based and open programming language: at run time, a lot of optimisations can be applied that are not possible for a static compiler. The JIT compiler can be reflective, taking advantage of the ability to dynamically inspect referenced value. This is particularly important in the optimization of futures and cross-component references represented as futures.

0.9.3 Safety

The Alice implementation does not yet give any safety guarantees – in particular, the integrity and signature information of components is not verified currently. While type-annotated code is rather standard, the ability to dynamically create components via pickling also requires a certain amount of type information about data in the heap. It hence was a conscious decision for the Alice project to focus on language design first and consider implementation-level safety issues in a second phase.

0.10 RELATED WORK

Java [GJS96] was the first major language designed for open programming. It is object-oriented and only weakly typed. Concurrency is very basic. Open programming is based on *reflection*, which not only allows a component to describe itself, but also allows other components to exploit this information constructively. We feel that reflection is expensive, invites abuse, and usually demands a rather limited type system. Packages may be considered as providing a weak form of reflection that avoids these issues. Serialisation in Java is simplistic and requires support from the programmer. Code cannot be serialised, only class names can be used to represent code, which is a weak and fragile abstraction. No structural type checks are performed when a class is loaded, method calls may cause a `NoSuchMethodError`.

The Microsoft ".NET" Common Language Runtime [Mic03] is a framework that is very similar to Java in most aspects related to open programming, but unlike Java is meant to support multiple languages.

Many of the concepts in Alice have been inherited from Oz [Smo95, VH04] and its implementation, the Mozart programming system [Moz04]. It has high-level support for concurrency, pickling, components and distribution similar to Alice. The distribution subsystem is more expressive, supporting distributed state and futures. Unlike

Alice, Oz is not statically typed and it is based on a relational core language with logic variables being the primary means for data flow synchronisation.

Acute [SLW+04] is an experimental, ML-based language for strongly typed open programming. It has an expressive but ad-hoc notion of component with versioning. It supports distribution, but the details are left to the programmer. Resources can be dynamically rebound upon unpickling, using explicit marks in the program. No safety mechanism is built in. Abstraction safety is ensured, with different levels of generativity, but can be broken by explicit means. No implementation is available yet.

JoCaml [FMS01] is a distributed extension of OCaml [Ler03] based on the Join calculus. Concurrency and distribution are more high-level than in Alice: channels achieve both, concurrency and distribution, with expressive means of synchronisation and thread migration. On the other hand, JoCaml is not really open: pickles contain just monomorphic values and can only be stored on a global name server, and there is only a weak component concept. There is only an experimental implementation.

CML [Rep99] is a concurrent extension of SML. It is based on first-class channels and synchronisation events, and does not support data flow synchronisation. It has no distribution features, nor does it address other aspects of open programming.

Erlang [AVWW96] is an untyped, purely functional language with an additional process layer, designed for embedded distributed applications. Processes can only communicate over implicit, process-global message channels. On the other hand, Erlang has a rich repertoire for dealing with failure. Erlang is not designed for open programming, and does not directly support code mobility nor safety policies. As an important feature, it provides the ability for updating code in active processes.

Clean is a concurrent, purely functional language with a type-safe import/export facility based on simple dynamics [Pv00, Pil96]. It does not have advanced support for components or distribution.

0.11 OUTLOOK

We presented the design of Alice, a language for open progamming. Alice provides a novel combination of features to provide concurrency, modularity, a flexible component model and high-level support for distribution. Alice is typed and supports typeful programming (abstraction-safety) dynamically. It is fully implemented as part of a rich programming environment [Ali04b].

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, heap and byte code need to carry sufficient type information to allow creation of verifiable pickles. Research on these issues has been left for future work.

Other open questions we plan to address in the future are – among others – applicative type generativity, a semantics for unloading or even updating components in a manager, and higher-level library abstractions for concurrency and distribution.

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

REFERENCES

[ACPP91] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically-typed language. *TOPLAS*, 13(2), 1991.

[ACPR95] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1), 1995.

[Ali04a] The Alice Project. http://www.ps.uni-sb.de/alice, 2004. Homepage at the Programming Systems Lab, Universität des Saarlandes, Saarbrücken, Germany.

[Ali04b] Alice Team. *The Alice System*. Programming System Lab, Universität des Saarlandes, http://www.ps.uni-sb.de/alice/, 2004.

[AVWW96] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang, Second Edition*. Prentice-Hall, 1996.

[BK02] Thorsten Brunklaus and Leif Kornstaedt. A virtual machine for multi-language execution. Technical report, Universität des Saarlandes, Saarbrücken, Germany, 2002.

[Car91] Luca Cardelli. Typeful programming. In *Formal Description of Programming Concepts*. Springer-Verlag, Berlin, Germany, 1991.

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

[DKSS98] Denys Duchier, Leif Kornstaedt, Christian Schulte, and Gert Smolka. A higher-order module discipline with separate compilation, dynamic linking, and pickling. Technical report, Programming Systems Lab, Universität des Saarlandes, Saarbrücken, Germany, 1998.

[FF95] Cormac Flanagan and Matthias Felleisen. The semantics of future and its use in program optimizations. In *POPL*, San Francisco, USA, 1995.

[FMS01] Cédric Fournet, Luc Maranget, and Alan Schmitt. *The JoCaml Language beta release*. INRIA, http://pauillac.inria.fr/jocaml/htmlman/, 2001.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Programming Language Specification*. Addison–Wesley, 1996.

[Hal85] Robert Halstead. Multilisp: A language for concurrent symbolic computation. *TOPLAS*, 7(4), 1985.

[Ler95] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *POPL*, San Francisco, USA, 1995. ACM.

[Ler03] Xavier Leroy. *The Objective Caml System*. INRIA, 2003.

[Lil97] Mark Lillibridge. *Translucent Sums: A Foundation for Higher-Order Module Systems*. PhD thesis, Carnegie Mellon University, Pittsburgh, USA, 1997.

[Mic03] Microsoft Corporation. *Microsoft .NET*. http://www.microsoft.com/net/, 2003.

[Moz04] Mozart Consortium. The Mozart programming system, 2004. www.mozart-oz.org.

[MP88] John Mitchell and Gordon Plotkin. Abstract types have existential type. *TOPLAS*, 1988.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *Definition of Standard ML (Revised)*. The MIT Press, 1997.

[Myc83] Alan Mycroft. Dynamic types in ML, 1983.

[NSS02] Joachim Niehren, Jan Schwinghammer, and Gert Smolka. Concurrent computation in a lambda calculus with futures. Technical report, Universität des Saarlandes, 2002.

[Pil96] Marco Pil. First class fi le I/O. In *IFL'96*, LNCS, vol.1268. Springer-Verlag, 1996.

[Pv00] Rinus Plasmeijer and Marko van Eekelen. *Concurrent Clean Language Report*, 2000.

[Rep99] John Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[Rey83] John Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing*, Amsterdam, 1983. North Holland.

[Ros03] Andreas Rossberg. Generativity and dynamic opacity for abstract types. In *PPDP*, 2003.

[Rus98] Claudio Russo. *Types for Modules*. Dissertation, University of Edinburgh, 1998.

[Sch02] Christian Schulte. *Programming Constraint Services*, volume 2302 of *LNAI*. 2002.

[SLW+04] Peter Sewell, James J. Leifer, Keith Wansbrough, Mair Allen-Williams, Francesco Zappa Nardelli, Pierre Habouzit, and Viktor Vafeiadis. Acute: High-level programming language design for distributed computation. Technical Report RR-5329, INRIA, 2004.

[Smo95] Gert Smolka. The Oz programming model. In *Computer Science Today*, volume 1000 of *LNCS*. Springer-Verlag, Berlin, Germany, 1995.

[TL04] Guido Tack and Didier Le Botlan. Compositional abstractions for search factories. In *MOZ 2004, Charleroi, Belgium*, LNCS. Springer-Verlag, 2004.

[VH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004.

[Wad89] Philip Wadler. Theorems for free! In *FPCA*. ACM Press, 1989.