Status Report: HOT Pickles, and How To Serve Them

Andreas Rossberg
Saarland University
rossberg@ps.uni-sb.de

Guido Tack
Saarland University
tack@ps.uni-sb.de

Leif Kornstaedt
Microsoft Corp.
kornstaedt@ps.uni-sb.de

Abstract

The need for flexible forms of serialisation arises under many circumstances, e.g. for doing high-level inter-process communication or to achieve persistence. Many languages, including variants of ML, thus offer pickling as a system service, but usually in a both unsafe and inexpressive manner, so that its use is discouraged. In contrast, safe generic pickling plays a central role in the design and implementation of Alice ML: components are defined as pickles, and modules can be exchanged between processes using pickling. For that purpose, pickling has to be *higher-order* and *typed* (*HOT*), i.e. embrace code mobility and involve runtime type checks for safety. We show how HOT pickling can be realised with a modular architecture consisting of multiple abstraction layers for separating concerns, and how both language and implementation benefit from a design consistently based on pickling.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; E.2 [*Data*]: Data Storage Representations

General Terms Languages, Design

Keywords Pickling, serialisation, marshalling, persistence, distributed programming, components, virtual machines

1. Introduction

Pickling is a service for externalising language-level values from a process's heap soup. Many languages offer such services, often known as *serialisation* or *marshalling*. But most of these services do not deliver pickles *HOT* – as higher-order, typed, self-contained object representations, like the functional programming gourmet would prefer them.

In this paper we present a recipe for such pickles: we discuss the pickling mechanism in *Alice ML* [27, 25, 26] and central aspects of its implementation in the Alice Programming System [3]. Our recipe is truly *higher-order*, in that it includes pickling of code, and even entire modules. It also is *typed*: where necessary, pickles carry structural type information that is checked upon unpickling. Both programming language design *and* implementation can benefit from being designed with HOT pickling in mind.

In previous work, we have already taken a high-level look at pickling as a language feature with abstract semantics [25], and we have discussed the implementation of an efficient raw pickling service at the lowest level [31]. In this paper, we discuss the higher levels of a realistic implementation, filling the (wide) gap between the previous papers. In particular, we lay the focus on the treatment of code and types, as these are the most unique aspects of pickling in our system, and we show how the whole mechanism can be realised using an architecture of layered abstractions.

1.1 Motivation

The main purpose of pickling is the support of *open programming*. By that we mean development of programs that are open to dynamic extension, persistence, communication with other processes, and exchange of behaviour. We characterise this by a number of abilities [27]: *modularity*, flexible combination of separate components; *dynamicity*, import *and* export of components at runtime; *portability*, exchange of components across different platforms; *safety*, graceful treatment of erroneous components; *security*, graceful treatment of untrusted components; *distribution*, communication of components across networks; and *concurrency*, handling asynchronous events and non-sequential tasks.

Alice ML is an extension of Standard ML [16] that has been specifically designed for open programming. In particular, Alice ML features a flexible system of components [25] that can be exchanged between processes as pickles. We will give a brief overview of Alice ML in Section 2.

A primary goal in the design of the Alice ML language as well as its implementation was to identify core primitives and generic abstractions that enable the realisation of open programming features in a modular manner. To this end, a vital decision was to base the language semantics on generic pickling as a central primitive. Modular implementation of this service in turn induced a number of interesting design decisions in the implementation, such as a generic store abstraction modelling the heap.

It turns out that this modular design approach not only makes the architecture more manageable, it also increases the expressive power of the language in considerable ways. As we will see, Alice ML is able to make components first-class, it can provide first-class access to the compiler, it uses one uniform representation for compiled "binaries" and serialised runtime values, and it even allows arbitrarily mixing the notions of compilation and computation, while processes can still dynamically load, create, and exchange such components. And all that in a strongly typed context.

1.2 Requirements and Architecture

To be adequate as a basis for language-level open programming, we require a number of properties from pickling:[1]

- *Transparency*: the result of unpickling a pickled value should be an observationally equivalent copy of the original.

[1] Note that it is an entirely different problem to interoperate with foreign languages and existing protocols, which we do not consider here.

ML'07, October 5th, 2007, Freiburg, Germany

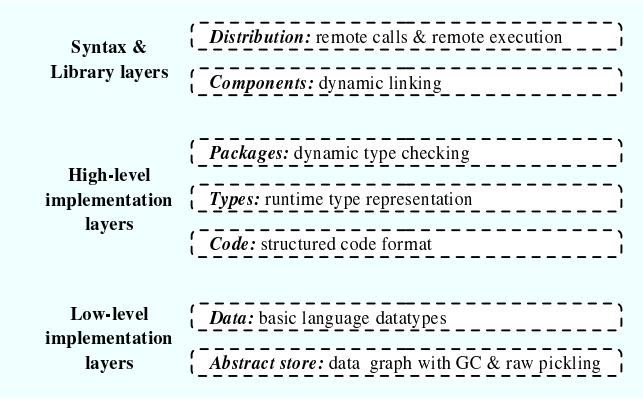

Figure 1. Layered architecture for higher-order typed pickling

- *Universality*: any type of value should be picklable. Particularly functional and user-defined types should be readily supported.
- *Closedness*: a pickle has to be self-contained. That is, it has to include the whole transitive closure of a value – for functions it has to carry all necessary code, including library functionality.
- *Verifiability*: it should be possible to check for malicious values.
- *Security*: names in a pickle should not be able to silently capture security-relevant *resources* in a process.
- *Portability*: representation should be independent from hardware platform, operating system, or other such concerns.
- *Adaptivity*: on the other hand, the pickling service has to address these concerns when *unpickling* in a given environment.
- *Efficiency*: the size of a pickle should be linear in the size of the represented value in memory. In particular, cycles and all *sharing* have to be maintained. Moreover, pickling and unpickling should take linear time.

The pickling architecture of the Alice system we describe meets all these requirements, except for verifiability. We believe this can be added as well, but have left the details for future work – Section 12 offers a brief discussion. We are not aware of any other implementation of pickling that meets all these requirements.

Our pickling architecture consists of a number of abstraction layers. In particular, the *representation* of pickles is defined by a tower of layers shown in Figure 1. Each layer is comparably small and easy to implement, more complex aspects can be defined in high-level terms. Specifically, only the implementations for the bottom two layers in the figure necessarily need to drop down to a machine-level language such as C – although for efficiency, parts of the third do, too. The middle representation layers are defined in terms of ML itself. However, implementation of respective operations requires access to unsafe primitives defined by the lower layers, and hence the middle layers have to be part of the "trusted base" of the system. In contrast, the top layers can be defined completely in terms of the language proper, either as simple library functionality, or as syntactic sugar.

The lowest two layers have already been described in a previous ML Workshop paper [31], and we will only briefly recap them here. Likewise, most details on the upper two layers can be found in previous work [25]. Consequently, in this paper we focus on the middle layers, which implement code and types. In the end, this should also draw the bigger picture of how all the parts integrate into the overall design.

Before we start off describing the above architecture, we reiterate central points of the design of Alice ML (Secton 2). We then describe the lower five implementation layers (Sections 3–7, after which we recap the component system (Section 8). We next discuss issues of compilation (Section 9) and the addition of distribution (Section 10) and concurrency (Section 11). Before we wrap up with related work, we sketch a possible approach for adding pickle verification (Section 12).

2. Alice ML

Alice ML is a conservative extension of Standard ML [16] designed for open programming. It adds three central features:

- *Packages* [25], a variation of dynamics [1] carrying modules.
- *Pickling*, generic higher-order serialisation as described above.
- *Futures* [19], enabling laziness and light-weight concurrency with implicit data-flow synchronisation.

Packages are essential to make pickling type-safe where types are not known statically. Futures are convenient to deal with asynchronicity and delay in external communication. Based on these primitives, Alice ML provides derived features for high-level open programming:

- *Persistence*, the ability to store values or entire modules in files.
- *Components* [25], lazily linked, first-class program fragments.
- *Proxies* [26], mobile RPC wrappers for stationary functions.

All these features uniformly use the pickling mechanism: persistent values are pickles, arguments of remote procedure calls are passed as pickles, and component files are pickles. Note particularly that Alice ML has no separate notion of binary code format – the compiler simply produces pickled component values!

As a simple example, the following program fragment dynamically *computes* a component using the first-class component syntax **comp**...**end**. The component exports a function hello that prints its own creation date. The component is saved to a file, using the pickling mechanism:

```
val date = Date.fromTimeLocal (Time.now ())
val component =
    comp
        import structure TextIO : TEXT_IO
                from "x-alice:/lib/system/TextIO"
    in
        val hello : unit → unit
    with
        fun hello () = TextIO.print (Date.toString date)
    end
do Component.save ("hello", component)
```

Once saved, the now persistent component can be imported as if it was conventionally created by the compiler:

```
import val hello : unit → unit from "hello"
do hello ()
```

Note how the component captures the dynamically pre-computed date when it is *created*, while importing the print function in the process where it is *linked*.

We delay an example of the use of proxies until Section 10.3. For a more thorough presentation of the design of the Alice ML open programming features we refer to other sources [27, 25, 26].

3. Abstract Store

The lowest-level abstraction in our architecture is the *abstract store*. All data, comprising everything a program allocates on the heap, but also code and runtime type information, is ultimately represented in the abstract store.

The abstract store is part of the SEAM framework for extensible virtual machines [28], on which the Alice system is built. We only give an overview here. For a more technical description of abstract store and pickling, we refer you to [31, 14]. A technical report contains a more detailed discussion of the SEAM design [8].

3.1 Data graph

The abstract store contains a graph representing the *heap* of a process. Despite this, it is completely agnostic of the language, its data types, code format, or type system. In fact, there are only four different *kinds* of nodes that can occur in the graph:

- *Scalars*, leaves in the graph representing bounded integers.
- *Chunks*, leaf nodes carrying unstructured data of arbitrary size.
- *Blocks*, inner nodes with a fixed number of *slots* referring to children (successors in the graph).
- *Transforms*, marking subgraphs to be modified for pickling.

Naturally, scalars are optimised using an unboxed representation, but this is immaterial to the abstraction. Blocks and chunks are attributed with an optional integer *label*, and a flag indicating whether the node is *mutable*, i.e., whether it allows modification of its contents. Transforms will be discussed in Section 3.3.

The store provides an interface for *allocating* nodes, for *initialising* them, for *accessing* them, and for *mutating* mutable nodes. It also offers functionality for inquiring kind and attributes of a node. Note that the graph can be cyclic.

3.2 Store services

Apart from these primitives, the abstract store provides three central *services*:

- *Garbage collection*, ensures that no explicit de-allocation of nodes is necessary, and that the abstract store is internally consistent (no dangling pointers).
- *Raw pickling*, allows to create an external, platform-independent binary representation of the subgraph of the store that is reachable from a given node. The inverse *unpickling* operation recreates a copy of the subgraph, preserving its structure.
- *Minimisation*, can be used to minimise the representation of a given subgraph by inducing maximum sharing, i.e. forming a minimal equivalent graph [31]. Note that this includes loop folding for cyclic structures.

Like garbage collection, pickling and minimisation are entirely generic services in that they can be applied to anything in the store. Pickling complete reachable subgraphs is essential for our approach. We will see that our encoding of Alice ML data structures in terms of abstract store nodes satisfies our requirement that a pickle contains the complete transitive closure of a value.

The raw pickling service currently supports pickling of stateful data, i.e. mutable nodes. This induces significant complications to maintain the illusion of atomicity in the presence of concurrent threads, in particular when futures (Section 11) are present – see [31] for details. Although the approach described there successfully addresses these issues, we intend to reconsider this choice for future versions of Alice ML, because unrestricted stateful pickling can grossly break encapsulation of stateful programming abstractions.

3.3 Transformations

For some types of data, it is too strong a requirement to use the same representation in the abstract store as in a pickle. For instance, floating point numbers could be stored using the native format of the hardware platform, whereas in a pickle the format must be in platform-independent IEEE with fixed endianness. Or, we want to execute code in an optimised format, which could be native code for the host processor architecture, while the pickled code again must be independent of the hardware. Some data structures may maintain caches to speed up computation, but pickling the cache is not necessary as it can be recreated upon unpickling, saving space in the pickle.

To this end, the pickling service provides a way of transforming data between internal and external representations. It is the task of *transform* nodes to mark data that requires transformation. A transform node contains a pointer to a transformation function and has a child node representing the actual value. The transformation function is called by the pickling service whenever it encounters a transform node, passing the child node as argument. Its result is a (possibly newly allocated) node in the abstract store that is pickled in place of the child. Conversely, a dual transformation function must be registered with the unpickling service so that the external value can be transformed back to the internal representation when a transform node is encountered in a pickle.

The concrete transformation is thus given as a pair of functions. That way pickling can be implemented as a generic service of the store, while being parametric over, but adaptive to, language- or library-specific, system-dependent representation choices.

If a transform node does not define a corresponding transformation function, pickling will fail. This can be used deliberately to mark nodes that are not allowed to be pickled. We make use of this for the representation of resources (Section 4.1).

3.4 Pickle Format

The pickles created by the abstract store service have a well-defined, platform-independent binary format. Each pickle contains a version number, such that incompatible changes in the implementation can be detected.

A binary pickle consists of a program for a simple stack-based unpickling machine. It contains a sequence of instructions. Each instruction may consume abstract store nodes from the stack and leave a node on the stack. There is one instruction corresponding to every node kind. Instructions for scalars and chunks simply construct the corresponding node and push it on the stack. A block instruction has two arguments, a label and the number of slots the block has. Interpreting it consumes as many stack values as the block has slots, creates the block node using the arguments, and pushes it on the stack. A transform instruction consumes the argument of the transform from the stack, performs the transformation (identified by a string argument of the instruction), and pushes the root node of the result back on the stack.

In order to support sharing in the graph, the unpickling machine also has a set of registers, plus instructions to push and pop registers. For cyclic graphs, an additional pair of instructions is needed that creates a placeholder for one node on the cycle and stores it in a register, and later, after constructing the rest of the cycle, replaces the placeholder.

For pickling, the store service traverses the data graph in a depth-first, post-order manner, starting at the root node to be pickled. Sharing and cycles can be detected by keeping track of all nodes that have already been visited. The transformation of a node is triggered as soon as a transform node is found, and the pickler descends into the result of the transformation.

In a final step, the generated “pickle program” is compressed using the Zlib compression library.

4. Data

The *data layer* defines how language-specific data structures are mapped to the abstract store. Most basic ML values have a straightforward encoding in terms of the abstract store. For instance,

- *integers*, *characters*, *nullary constructors* and *native pointers* to non-heap "foreign" data are represented as scalars,
- *floats* and *strings* are boxed as immutable chunks,
- n-ary *tuples* or *records* are immutable blocks with n slots,
- unary constructors are immutable blocks with arity dependent on their argument type (i.e., record arguments are *merged*); the block label encodes the constructor index (for large datatypes, a secondary index is stored in an additional slot),
- *references* are mutable unary blocks; *arrays* are either mutable blocks or chunks, depending on their type,
- function *closures* are immutable blocks carrying a *code object* (see Section 5) and a vector of environment values.

4.1 Resources

Some values have a semantics that is defined only local to the current process. Classic examples are file handles or native pointers to data that lives outside the abstract store (e.g. imported through the foreign function interface).

We collectively call these values *resources*, and we do not allow to pickle them. Any value that contains a resource is called *sited* – any attempt to pickle it will be rejected. This can be achieved easily by wrapping all values representing resources in a transform node without a transformation function.

Alternatively, it would be possible to use transforms to implicitly *rebind* certain resources on the target site. For example, the `stdOut` stream could easily be connected to the respective object in the target process. However, we consciously refrain from using this option – the Alice ML security model is such that a pickle can never contain any resources. Any resource is considered private, and must be explicitly acquired on a site, giving rise to a form of capability-based security. The dynamic component concept of Alice ML makes it possible to formulate respective abstractions for distributed programming, as well as controlling resource acquisition through user-defined *sandboxing* [26, 25].

A third alternative would be to mirror sitedness in types, using some suitable effect system, and rule out pickling of sited values statically. This is an interesting option, but would induce substantial changes to the type system of the language. We hence leave investigation of this idea for future work.

5. Code

The most outstanding property of our pickling architecture is its full support for higher-order values, i.e. functions. Obviously, this not only requires transferring closures, but in fact, full code mobility.

Supporting code mobility as part of pickling induces a set of requirements on the way code is represented:

1. The (primary) code format has to be platform-independent and relocatable.
2. Code must be movable (and garbage-collectable) with per-function granularity.
3. The transitive closure of all code fragments reachable from a given function must be easy to compute.

In the Alice system, these requirements are addressed by the following central design choices for *code objects*:

- The external representation of code, the so-called *abstract code*, is a high-level format suitable for efficient runtime *compilation*.
- Optionally, just-in-time compilation (*jitting*) can produce a format more adequate for efficient *execution* on a given platform (either native code or byte code).
- Abstract code is represented as a regular ML data structure on the heap; jitted code lives in a chunk node on the heap.
- Every jitted code object is wrapped by a transform yielding the respective abstract code object.
- Every source-level function (counting nested functions separately) is represented by an individual code object.

The last point is the most important: it is what allows pickling and garbage collection of code to be performed on the granularity of single functions.[2] The use of transforms for code objects is what achieves platform-independence. We discuss the other points in the following subsections. More detailed discussion of the design space can be found in a technical report describing SEAM [8].

5.1 Abstract Code Format

The Alice abstract code is an instruction-based code format with SSA-style variable bindings. It is graph-structured: every instruction contains its continuation(s), thus representing control flow. Variables and instructions are annotated with liveness information that allows efficient register or stack allocation in a just-in-time compiler. Most of the instruction set is fairly standard and immaterial to the problem of pickling, thus we do not describe it here.

Thanks to our layered approach, the representation of code can be defined in terms of a set of ML datatypes. Thus, it can be constructed and processed in ML itself. In particular, the compiler can directly generate code as ML values.

More importantly, a consequence of this choice is that the code's representation in the store is implicitly defined via the data layer. Transitively, this also defines its external representation in pickles, in a platform-independent manner. To produce a "binary", the compiler simply pickles a value from its own heap.

The Alice abstract code currently is untyped. See Section 12 for respective discussion.

5.2 Jitting and Interpreters

Two kinds of jitted code are currently supported in the Alice runtime: *byte code*, available on all platforms [17], and *native code*, used only on x86 platforms. The byte code jitter performs various dynamic optimisations, especially with respect to futures (Section 11). Furthermore, all optimisations can cross component boundaries, which is especially important when software is deployed as many small components.

The type of code can be selected per function – every code object contains a reference to a suitable *interpreter* for executing the code. Every interpreter also defines its own layout of stack frames. Calls may freely mix functions executing with different interpreters. Thus, jitting can be performed selectively for "hot spots". In fact, the byte code jitter is even capable of selective rejitting of heavily used functions [17].

5.3 Reachable Code

As usual, first-class functions are represented by closures at runtime, which contain a reference to a respective code object.

The transitive closure of code reachable from a given function coincides with the data graph reachable from its representing closure: when the function can call another function, a reference to the latter has to appear either in the environment of the function closure or directly in its code (see next section), rendering it a suc-

[2] In experiments we found that by forming the transitive closure at that granularity, the size of pickled modules with non-trivial imports usually was in the range of 100%–300% of the original component size, while the size of the transitive closure at *module* granularity typically lied in the range of 400%–1400%. Moreover, cutting off library modules as often proposed rarely reduced this size by more than a third.

cessor in the data graph. Thus, pickling and garbage collection of code become trivial.

Given the fine granularity of code objects, no special treatment is necessary for "persistent" library code: we can afford to include all referenced library functions in a pickle, thus achieving proper closedness and at the same time vastly simplifying the implementation by avoiding ad-hoc cut-off and rebinding mechanisms.

5.4 Embedded Values

Because abstract code is a regular ML value, and "binaries" are written by pickling, code is not restricted to sequential form but can be arbitrarily structured. In particular, code can contain structured values as "immediates" – loading an immediate value simply loads a node, whether scalar or not.

The Alice ML compiler exploits this by performing a general form of constant folding, or *value propagation*, that is not limited to scalar types, but can compute values of arbitrary type – especially algebraic datatypes, such as lists or trees – and embed the result into the generated code. It induces no extra cost if the same value is embedded more than once, since pickling maintains sharing.

Even closures can be constructed at (static) compile time and embedded at call sites where their environment is statically known. Moreover, in the presence of value propagation, a statically known environment coincides with an empty one.

The Alice runtime also utilises embedded values for dynamic *specialisation*: the abstract code features a special variant of a closure creation instruction that does not construct a conventional closure object, but instead creates a copy of the underlying abstract code object and dynamically embeds the environment values as immediates. Consecutive jitting can take advantage of this to perform additional optimisations, especially inlining. The static compiler generates this instruction for functors and top-level functions with comparable usage profile.

Last but not least, the ability to embed arbitrary values in generated code also gives rise to an expressive dynamic compilation facility (Section 9.1). Dynamic compilation can embed values from the actual heap of the host process. By using an internal form of reflection, value propagation is able to exploit such values as well.

6. Types

Achieving type safety for open programming requires strategic use of dynamic type checking. At its core, Alice ML employs *packages* for this purpose (Section 7). To support them, a certain amount of type information has to be computed at runtime. This includes ML core types as well as module signatures. Hence, we need a runtime representation for both.

We implement runtime types by *reifying* type and signature expressions to the term level, that is, transforming them into expressions that compute suitable type descriptions at runtime. Regarding our layered architecture, the implementation of types is thereby done in terms of *code* that constructs type data structures.

6.1 Type representation

Like code, types and signatures are represented in a high-level manner by regular ML data structures. The details of the representation are again immaterial to our approach – type representations are realised as abstract data types. Somewhat idealised signatures for these abstract types are given in Figure 2.

The modules implementing these signatures have to be part of the system's trusted kernel. In practice, they use relatively standard term structures with optimisations like lazy substitution, caching of signature look-up tables, and others described in the literature [30].

A peculiar point of Alice ML's type system is its semantics of type abstraction: to maintain abstraction safety in the presence of

```
structure Type :
sig
    type ty and tyrow
    type tyvar = string
    eqtype tycon
    val tyvar : tyvar → ty
    val tycon : tycon → ty
    val arrow : ty × ty → ty
    val record : tyrow → ty
    val apply : ty list × ty → ty
    val func : tyvar list × ty → ty
    val poly : tyvar list × ty → ty
    val emptyRow : unit → tyrow
    val extendRow : tyrow × string × ty → tyrow
    val new : int → tycon
end

structure Sig :
sig
    type sign and spec
    type id = string
    type longid = id list
    type rea = (longid × ty) list
    val sign : spec → sign
    val fct : sign × sign → sign
    val emptySpec : unit → spec
    val valSpec : id × ty → spec
    val typSpec : id × int × ty option → spec
    val strSpec : id × sign → spec
    val sigSpec : id × sign → spec
    val seqSpec : spec × spec → spec
    val findVal : sign × longid → ty
    val findTy : sign × longid → ty option
    val findStr : sign × longid → sign
    val findSig : sign × longid → sign
    val matches : sign × sign → bool
    val realise : sign × rea → sign
end
```

Figure 2. ADTs for runtime type and signatures

dynamic typing, it is based on dynamic generation of fresh type names [23]. This is mirrored by the function Type.new for generating fresh type constructors. They are represented by globally unique identifiers (GUIDs, as provided by suitable operating system mechanisms) to emulate a global type heap.

As the only non-obvious operation, Sig.realise instantiates a signature given a *realisation* mapping all its abstract types (identified by relative long identifiers) to concrete type constructors.

6.2 Type reification and erasure

Runtime types are generated by reifying type expressions and declarations to term-level expressions performing appropriate calls to the type ADTs. Figure 3 gives a semi-formal sketch of a source-to-source translation performing this reification. It is similar in spirit to the type erasure translation described by Crary et al. [10]. But since our code format is untyped, we treat the target language as typeless, such that we do not have to deal with the introduction of singleton representation types here.

The translation assumes an injective mapping •$ from type and signature identifiers to fresh value identifiers. This mapping is extended to long identifiers in the obvious way.

Note that the transformation only involves *explicit* type and signature information. This is no accident: Alice ML has been designed such that core language polymorphism stays fully parametric, and types can be erased on that level [25]. This not only simplifies compilation, it also avoids substantial operational overhead that would otherwise arise from the presence of dynamic types,

$$
\begin{array}{ll}
[\![\textsf{type}\ \overline{\alpha}\ tycon{=}ty]\!] & = \textsf{val}\ tycon\$ = [\![\lambda\overline{\alpha}.ty]\!] \\
[\![\textsf{signature}\ sigid{=}sigexp]\!] & = \textsf{val}\ sigid\$ = [\![sigexp]\!] \\
\\
[\![\lambda\overline{\alpha}.ty]\!] & = \mathsf{Type.func}([\overline{"\alpha"}], [\![ty]\!]) \\
[\![\alpha]\!] & = \mathsf{Type.tyvar}\ "\alpha" \\
[\![\overline{ty}\ longtycon]\!] & = \mathsf{Type.apply}([\overline{[\![ty]\!]}], longtycon\$) \\
[\![ty_1 \to ty_2]\!] & = \mathsf{Type.arrow}([\![ty_1]\!], [\![ty_2]\!]) \\
[\![\{\langle tyrow \rangle\}]\!] & = \mathsf{Type.record}\ [\![\langle tyrow \rangle]\!] \\
[\![\ ]\!] & = \mathsf{Type.emptyRow}() \\
[\![lab : ty\ \langle, tyrow \rangle]\!] & = \mathsf{Type.extendRow}("lab", [\![ty]\!], [\![\langle tyrow \rangle]\!]) \\
\\
[\![longsigid]\!] & = longsigid\$ \\
[\![\textsf{sig}\ spec\ \textsf{end}]\!] & = \mathsf{Sig.sign}\ [\![spec]\!] \\
[\![\textsf{fct}\ strid{:}sigexp_1 & \\
\quad \to sigexp_2]\!] & = \mathsf{Sig.fct}("strid", [\![sigexp_1]\!], [\![sigexp_2]\!]) \\
[\![spec]\!] & = \ldots
\end{array}
$$

Figure 3. Type and signature reification

particularly given that separate compilation and dynamic linking of components precludes many respective optimisations.

7. Packages and Modules

Alice ML features *packages* [25], first-class values containing a module paired with its signature – essentially a module-based variation of *dynamics* [1]. They are created using the expression form

pack $strexp$: $sigexp$

To access the module from a package, it has to be unpacked against a *target signature*: the module expression

unpack exp : $sigexp$

checks that the signature of the package exp matches $sigexp$, and only if that is the case, evaluates to the embedded module – otherwise, the exception Unpack is raised. This is the *only* form of dynamic type checking in Alice ML.

From a more type-theoretic point of view, the type package is a form of existential type $\exists\alpha.\alpha$ over modules. Since the very purpose of packages is dynamic type checking, the witness type of the existential – the package signature – cannot be erased. Consequently, the obvious representation of a package is an actual pair of the underlying module and the representation of the signature.

As for pickling, the two library operations

Pickle.save : string × package → unit
Pickle.load : string → package

allow creating and reading a package as a pickle. Because the package contains a runtime representation of the signature, so will the pickle. There is no direct way to access the raw pickling mechanism from the language level, consequently all pickle files will contain runtime type information. We call these "*cooked*" pickles.

7.1 Modules

An ML module can either be a structure or a (potentially higher-order) functor. Operationally, structures behave like simple records and functors like functions. As in other ML implementations, that also is how they are realised in the Alice system. However, there are noticeable differences:

- To support dynamic typing, type and signature fields are not erased from structures. Instead, they carry the runtime representation of the respective type-level objects.
- Sealing (using the :> operator) replaces all type fields specified abstractly in the ascribed signature with fresh type names.
- Subtyping is *not* coercive. That is, use of signature subsumption does not imply narrowing in the representation (however, narrowing is performed where signatures are *explicitly* ascribed). Consequently, the runtime layout of a structure generally is not known statically. Projection hence is realised via cached hash look-up, with efficient support from the virtual machine.

The latter design choice makes unpacking and dynamic linking in Alice ML simpler and potentially more efficient. A tupled representation of structures with narrowing is straightforward for implementations where every use of subtyping is static: in that case, both source and target arity of a structure are statically known, and the coercion can be performed statically. Projection then amounts to simple indexing. Unfortunately, the situation is different with packages: unpack employs subtyping *dynamically*, such that only the super signature is known statically. To maintain the representational invariants of a tuple representation, unpack would have to perform dynamic narrowing on a module of statically unknown signature. Although possible, it would require a more complex and expensive form of access to the original structure, e.g. using intensional signature analysis. Since dynamic linking heavily relies on packages, we believe that such an approach would be more costly.

7.2 Translation

In the same manner as for types, Figure 4 defines the representation of modules and packages as a source-to-source translation to an untyped core SML. In particular, record expressions do not need to have a statically fixed type in the target language, and projection $\#lab$ is "polymorphic". Again, we assume an injective embedding $\bullet\$$ of module identifiers. Note that structure bindings subsume functors in Alice ML's higher-order module language.

The translation uses a few auxiliary meta-level operators:

- $\textsc{binds}(dec)$ expands to a sequence of bindings $id{=}id$ for all identifiers id bound in dec (including types and signatures).
- $\textsc{narrow}(strid, p, sigexp)$ constructs a representation of the module $strid.p$ narrowed to signature $sigexp$.
- $\textsc{seal}(strid, p, sigexp)$ constructs a representation of the module $strid.p$ sealed by the signature $sigexp$.
- $\textsc{newrea}(p, sigexp)$ is a realisation mapping every abstract type in $sigexp$ to a fresh type constructor, under path p.
- $\textsc{selfrea}(strid, p, sigexp)$ is a realisation mapping every abstract type in $sigexp$ to the representation from $strid.p$.

The latter operators are defined in Figure 5, assuming previous expansion of signature identifiers with their respective definition. They are all defined by induction over a path p of nested structures.

Selfification is necessary for package signatures to avoid anomalies in the semantics of unpack, as described in [25]. The construction-time selfification given here optimises the deconstruction-time selfification given in the formal semantics of packages in that paper.

7.3 Structure access

As a final translation step of the module translation, all *long identifiers* appearing in expressions are translated to sequences of record selections, as shown in Figure 4. But there is an additional twist not shown in the figure. To keep pickles containing code as small as possible, it is important to tune the semantics of pickling such that a long identifier M.x appearing in some function body only represents a reference to the individual field x of the module M, *not* a reference to the entire module!

For example, the closure representing the function

fun last xs = List.hd(List.rev xs)

$$
\begin{array}{ll}
[\![\textsf{structure}\ strid{=}strexp]\!] & = \textsf{val}\ strid\$\ =\ [\![strexp]\!] \\
\\
[\![longstrid]\!] & = longstrid\$ \\
[\![\textsf{struct}\ dec\ \textsf{end}]\!] & = \textsf{let}\ [\![dec]\!]\ \textsf{in}\ \{\textsc{binds}(dec)\}\ \textsf{end} \\
[\![strexp : sigexp]\!] & = \textsf{let val}\ strid\$ = [\![strexp]\!] \\
 & \quad \textsf{in}\ \textsc{narrow}(strid, \epsilon, sigexp)\ \textsf{end} \\
[\![strexp\ :\!>\ sigexp]\!] & = \textsf{let val}\ strid\$ = [\![strexp]\!] \\
 & \quad\ \ \textsf{val}\ \mathsf{rea} = \textsc{newrea}(\epsilon, sigexp) \\
 & \quad\ \ \textsf{val}\ sigid\$ = \\
 & \quad\quad \mathsf{Sig.realise}([\![sigexp]\!], \mathsf{rea}) \\
 & \quad \textsf{in}\ \textsc{seal}(strid, sigid, \epsilon, sigexp)\ \textsf{end} \\
[\![\textsf{fct}\ strid{:}sigexp \Rightarrow strexp]\!] & = \textsf{fn}\ strid\$ \Rightarrow [\![strexp]\!] \\
[\![strexp_1\ strexp_2]\!] & = [\![strexp_1]\!]\ [\![strexp_2]\!] \\
[\![\textsf{let}\ dec\ \textsf{in}\ strexp\ \textsf{end}]\!] & = \textsf{let}\ [\![dec]\!]\ \textsf{in}\ [\![strexp]\!]\ \textsf{end} \\
[\![\textsf{unpack}\ exp : sigexp]\!] & = \textsf{let val}\ (strid\$, sigid\$) = [\![exp]\!] \\
 & \quad \textsf{in if}\ \mathsf{Sig.matches}(sigid\$, [\![sigexp]\!]) \\
 & \quad\quad \textsf{then}\ strid\$\ \textsf{else raise}\ \mathsf{Unpack} \\
 & \quad \textsf{end} \\
\\
[\![\textsf{pack}\ strexp : sigexp]\!] & = \textsf{let val}\ strid\$ = [\![strexp]\!] \\
 & \quad\ \ \textsf{val}\ \mathsf{rea} = \textsc{selfrea}(strid\$, \epsilon, sigexp) \\
 & \quad \textsf{in}\ (strid\$, \mathsf{Sig.realise}([\![sigexp]\!],\ \mathsf{rea}))\ \textsf{end} \\
[\![strid_1.\cdots.strid_n.vid]\!] & = \#vid(\#strid_n\$(\cdots(strid_1\$)\cdots))
\end{array}
$$

Figure 4. Module and package translation

should only contain the individual functions hd and rev, but none of the other definitions from the List module (and other modules transitively referenced by these!). Semantically, this can be modelled by trimming closure environments during pickling appropriately.

To achieve the same effect in the implementation, the selections resulting from the translation of long identifiers are *hoisted* to the binding point of the structure itself. For the above example, this would yield

```
val List_hd = #hd List$
val List_rev = #rev List$
...
fun last xs = List_hd(List_rev xs)
```

with the auxiliary declarations appearing at the binding point of the List structure.

8. Components

Components form the topmost language layer in Alice ML [25], being the unit of compilation, linking and deployment. In a nutshell, a component represents an "unlinked, unevaluated module", which may contain free references to modules from other components via *import* declarations. Components can be linked dynamically and acquired from remote locations denoted by a URL.

Components are first-class, they can be dynamically computed and exported from a running process. In its most general form, a component is a value of abstract type component created by the expression form

comp $import^*$ **in** $spec$ **with** dec **end**

where $import^*$ is a sequence of import declarations of the form

import $spec$ **from** url

As explained in [25], components can be defined as a derived concept within the language itself. They can be represented as simple higher-order functions over packages:[3]

[3] For historical reasons, the actual representation in the Alice system is slightly different, but with mostly equivalent effect.

$$
\begin{array}{ll}
\textsc{narrow}(X, p, \textsf{sig}\ spec\ \textsf{end}) & = \{\textsc{narrow}(X, p, spec)\} \\
\textsc{narrow}(X, p, \textsf{fct}\ Y{:}sig_1 \rightarrow sig_2) & = \textsf{fn}\ Y\$ \Rightarrow \\
 & \quad \textsf{let val}\ Z\$ = X.p\$(Y\$) \\
 & \quad \textsf{in}\ \textsc{narrow}(Z, \epsilon, sig_2)\ \textsf{end} \\
\textsc{narrow}(X, p, \textsf{val}\ x : ty) & = x{=}X.p.x \\
\textsc{narrow}(X, p, \textsf{type}\ \overline{\alpha}\ t\ \langle{=}\ ty\rangle) & = t\${=}X.p.t\$ \\
\textsc{narrow}(X, p, \textsf{structure}\ Y : sig) & = Y\${=}\textsc{narrow}(X, p.Y, sig) \\
\textsc{narrow}(X, p, \textsf{signature}\ S = sig) & = S\${=}X.p.S\$ \\
\textsc{narrow}(X, p, spec_1{;}spec_2) & = \textsc{narrow}(X, p, spec_1), \\
 & \quad \textsc{narrow}(X, p, spec_2) \\
\\
\textsc{seal}(X, S, p, \textsf{sig}\ spec\ \textsf{end}) & = \{\textsc{seal}(X, S, p, spec)\} \\
\textsc{seal}(X, S, p, \textsf{fct}\ Y{:}sig_1 \rightarrow sig_2) & = \textsf{fn}\ Y\$ \Rightarrow \\
 & \quad \textsf{let val}\ Z\$ = X.p\$(Y\$) \\
 & \quad \textsf{in}\ \textsc{narrow}(Z, \epsilon, sig_2)\ \textsf{end} \\
\textsc{seal}(X, S, p, \textsf{val}\ x : ty) & = x{=}X.p.x \\
\textsc{seal}(X, S, p, \textsf{type}\ \overline{\alpha}\ t\ \langle{=}\ ty\rangle) & = t\${=}\mathsf{valOf} \\
 & \quad (\mathsf{Sig.findTyp}(S\$, \textsc{str}(p.t))) \\
\textsc{seal}(X, S, p, \textsf{structure}\ Y : sig) & = Y\${=} \\
 & \quad \textsf{let val}\ T\$ = \\
 & \quad\quad \mathsf{Sig.findStr}(S\$, \textsc{str}(p.Y)) \\
 & \quad \textsf{in}\ \textsc{seal}(X, T, p.Y, sig)\ \textsf{end} \\
\textsc{seal}(X, S, p, \textsf{signature}\ T = sig) & = S\${=}\mathsf{Sig.findSig}(S\$, \textsc{str}(p.T)) \\
\textsc{seal}(X, S, p, spec_1{;}spec_2) & = \textsc{seal}(X, S, p, spec_1), \\
 & \quad \textsc{seal}(X, S, p, spec_2) \\
\\
\textsc{newrea}(p, \textsf{sig}\ spec\ \textsf{end}) & = \textsc{newrea}(p, spec) \\
\textsc{newrea}(p, \textsf{fct}\ Y{:}sig_1 \rightarrow sig_2) & = [] \\
\textsc{newrea}(p, \textsf{val}\ x : ty) & = [] \\
\textsc{newrea}(p, \textsf{type}\ \overline{\alpha}\ t = ty) & = [] \\
\textsc{newrea}(p, \textsf{type}\ \overline{\alpha}\ t) & = \textsf{let val}\ t\$ = \mathsf{Type.new}\ |\overline{\alpha}| \\
 & \quad \textsf{in}\ [(\textsc{str}(p.t), \mathsf{Type.tycon}\ t\$)]\ \textsf{end} \\
\textsc{newrea}(p, \textsf{structure}\ Y : sig) & = \textsc{newrea}(p.Y, sig) \\
\textsc{newrea}(p, \textsf{signature}\ S = sig) & = [] \\
\textsc{newrea}(p, spec_1{;}spec_2) & = \textsc{newrea}(p, spec_1) \\
 & \quad @\ \textsc{newrea}(p, spec_2) \\
\\
\textsc{selfrea}(X, p, \textsf{sig}\ spec\ \textsf{end}) & = \textsc{selfrea}(X, p, spec) \\
\textsc{selfrea}(X, p, \textsf{fct}\ Y{:}sig_1 \rightarrow sig_2) & = [] \\
\textsc{selfrea}(X, p, \textsf{val}\ x : ty) & = [] \\
\textsc{selfrea}(X, p, \textsf{type}\ \overline{\alpha}\ t = ty) & = [] \\
\textsc{selfrea}(X, p, \textsf{type}\ \overline{\alpha}\ t) & = [(\textsc{str}(p.t), X.p.t\$)] \\
\textsc{selfrea}(X, p, \textsf{structure}\ Y : sig) & = \textsc{selfrea}(X, p.Y, sig) \\
\textsc{selfrea}(X, p, \textsf{signature}\ S = sig) & = [] \\
\textsc{selfrea}(X, p, spec_1{;}spec_2) & = \textsc{selfrea}(X, p, spec_1) \\
 & \quad @\ \textsc{selfrea}(X, p, spec_2) \\
\\
\textsc{str}(x_1.\cdots.x_n) & = ["x_1", \cdots, "x_n"]
\end{array}
$$

Figure 5. Auxiliary definitions for translation

type component = (url → package) → package

Figure 6 recaps the respective decomposition of the component syntax. Given that decomposition, all functionality on components can be implemented in a library. In particular, *component managers*, responsible for controlling dynamic linking and realising sandboxing, can fully be programmed in the source language. More details of this can be found elsewhere [25, 27].

The net effect is that components require basically no extra support from the runtime system. A component is simply represented by its underlying function representation. A component file produced by the compiler simply is a pickle of that function. All the VM must be capable of is loading an initial pickle as a *boot* component and applying it to a rudimentary component manager.

```
comp import* in spec with dec end    ⇝
     fn link ⇒ let import* in pack (dec) : (spec) end

import spec from url    ⇝
     open (unpack link url : sig spec end)
```

Figure 6. Components decomposed

8.1 Persistence

In fact, the inverse is also true: all Alice ML pickle files are actually full-fledged components, interchangeable with other components. That is, the core library primitives for accessing pickles are

```
Component.save : string × component → unit
Component.load : string → component
```

The pickling functions previously presented in Section 7 are merely implemented as

```
fun save(f, p) = Component.save(f, Component.fromPackage p)
fun load f = DummyComponentManager.link(Component.load f)
```

where the fromPackage function has the straightforward definition

```
fun fromPackage p = fn _ ⇒ p
```

according to the above decomposition of components. The module DummyComponentManager simply is a component manager that allows no imports, i.e. where link is defined as follows:

```
fun link c = c (fn _ ⇒ raise Error)
```

This prevents accidental linking of untrusted components in an insecure manner.

9. Compilation

Since a component file is simply a pickle of a value of type component, it follows that code generation reduces to dynamically creating an appropriate closure corresponding to this type. To this end, the compiler must have access to the lower level abstraction, such that it can create the respective data structures representing the code of the function.

9.1 Dynamic Separate Compilation

On the outside, the compiler is just a function

```
val compile : string → component
```

This function is provided as part of the library, thus giving the programmer a first-class interface to compilation.

On top of this function, functionality similar to Lisp's eval can easily be realised. The Alice ML library does so in form of a family of functions, whose simplest member is

```
val eval : string → package = ComponentManager.link o compile
```

Here, ComponentManager is the explicit interface to the "current" component manager, which is responsible for linking [25].

More elaborate variants of the compile and eval functions allow threading an environment. For example:

```
val evalWith : env × string → env × package
```

The abstract type env encapsulates a compilation environment consisting of a static part mapping identifiers to types, kinds, and signatures, and a dynamic part mapping the same identifiers to actual values, runtime types, and modules, respectively. Through consecutive uses of evalWith, incremental compilation and evaluation can be performed. The environment sensitive compilation function,

```
val compileWith : env × string → component
```

can be used with environments resulting from previous calls to evalWith. Where the compiled program refers to the environment, the respective *dynamic* values will simply be embedded in the produced code as immediates (Section 5.4). This implies that the produced component can still directly be pickled as a self-contained stand-alone object, despite its dependency on dynamic values. To our knowledge, such a dynamic form of compilation into separate binary components is not available in any other system except Oz/Mozart [11], where it is untyped.

9.2 Program Transformations

The semantics of pickling is very sensitive to code transformations. Consider the following program:

```
fun f x = (print; x)
fun g x = if true then x else print "dead"
```

In conventional terms, the uses of print in these definitions are redundant or dead code, and the compiler would be free to remove them. However, print is a sited value, it cannot be pickled. Hence, the functions f and g, referencing this value, should become sited as well. Partial evaluation, dead code elimination and related transformations change this semantics.

In other words, through pickling, the extent of a transitive closure of a value becomes observable where it contains sited values (which cause failure) or futures (which cause blocking, and can trigger other computations). To maintain precise semantics, the compiler would have to be very careful and restrictive about desirable optimisations. Obviously, this would be unfortunate.[4]

The alternative is to consciously under-specify the language semantics in this respect. That is, allow any semantics that is *conservative* with respect to the canonical one, in the sense that it can only make more pickling operations succeed. This is the choice we adopted for Alice ML.[5] For cases where the programmer wants to deliberately enforce sitedness – e.g. to protect certain abstractions – the Alice ML library offers an abstract type α sited that can be wrapped around respective data structures.

10. Distribution

One important aspect of open programming is the ability to communicate over a network. To this end, Alice ML offers a high-level approach based on pickling and dynamic components, which we will briefly describe in this section. More details can be found in other sources [27, 26].

10.1 Proxies

Alice ML adopts *remote function calls* as the most natural way to generalise a functional language to a distributed language. The Alice library provides a single primitive for that:

```
proxy : (α → β) → (α → β)
```

This function takes a local function and creates a *proxy* for it. The proxy is a mobile wrapper for the local, stationary functions. Proxies can be pickled and thus passed to other processes. When applied remotely, the call is forwarded to the original site by pickling argument and result.

In a sense, proxies represent (logical) connections between processes. By representing connections as functions, inter-process communication directly inherits a number of useful properties, in particular:

1. Connections are first-class and mobile (i.e., can be pickled).

[4] Note that tracking sitedness in types would avoid this problem.

[5] There is no complete operational semantics of Alice ML, but an "obvious intuition" can be synthesised from given partial formalisations [25, 19, 24].

2. Communication is statically typed.
3. Communication is two-way and synchronous.

Using proxies first-class, arbitrary communication patterns can be formed by passing proxies back and forth as higher-order arguments to other proxies. Note that asynchronous communication can be achieved trivially but orthogonally by wrapping remote calls into futures [27]. Futures also enable programming of time-outs.

Under the hood, proxies can easily be implemented as closures that transfer pickles over some TCP connection. The most interesting aspect here is that a proxy is a *typed* connection, i.e. communication is well-typed by construction. Consequently, raw pickles are sufficient for passing arguments and results, they do not need to carry any runtime type information information themselves (unless verification is desired, cf. Section 12).

10.2 Establishing connections

With proxies representing connections between processes, there is the question of how such connections are *established*, i.e. how proxies are initially transferred between processes. The Alice ML library offers two ways, corresponding to two different scenarios:

- *Client/server*: a server process can *offer* a component on a network. The library function offer : component → url generates a temporary URL under which a client process can retrieve the component as a pickle.
- *Master/slave*: a master process can spawn computations by creating slave processes on accessible machines using the function run : host × component → package.

In both scenarios, a dynamically computed component can be passed between the processes, achieving four goals:

1. Arbitrary functionality can be transmitted between the processes.
2. Dynamic data from the original process can be captured, especially proxies.
3. Functionality available on the target site can be abstracted over as imports, especially resources.
4. Dynamic linking will check all type assumptions, especially of contained proxy connections.

The implementation of this functionality again is using pickling. It largely consists of straightforward utilisation of appropriate operating system services, such as ports, sockets, and SSH tunnels. By building on pickling and dynamic components, distribution hence is made simple and type-safe, yet expressive. Note again that dynamic type checking is only necessary for *establishing* a connection, later *use* of it for communication is statically type-safe.

Note also that due to pickling – which means copying – no distributed garbage collection is necessary. The only inter-process references are proxies. Currently, the Alice system does keep functions for which proxies have been created live for the rest of the process's life time. In principle, this can lead to space leaks when dynamic communication patterns are implemented. It remains to be seen whether this is a problem in practice.

10.3 Example

As a simple example for a client/server architecture with bi-directional communication, consider a minimal chat room consisting of a chat server to which multiple clients can connect using the generated URL. For simplicity, we omit error handling.

This example does not require the full power of dynamic components, because it does little more than communication. Thus the connection is established by offering merely a package instead of a full component – no imports are required on the target site. Both sides need to agree on a signature for the exchanged package. Basically, it describes the server interface:

```
signature SERVER =
sig
  val register :  (string → unit) → unit
  val broadcast : string → unit
end
```

Clients that register with the server will receive all messages sent by other clients, and they can broadcast messages themselves.

Here is the full code for the server:

```
val clients = ref nil
fun register client = clients := client :: !clients
fun broadcast message =
    List.app (fn receive ⇒ spawn receive message) (!clients)

structure Server = (val register = proxy (mutex () register)
                    val broadcast = proxy broadcast)
val url = offer (pack Server : SERVER)
do TextIO.print (url ^ "\n")
```

The server simply keeps a list of registered clients (represented by their receive functions); broadcasting iterates over this list and forwards the message to each. In order to avoid having to wait for each client in turn to receive the message, forwarding happens asynchronously, using a future created with spawn. Moreover, since the client list is stateful, we have to avoid race conditions when several clients try to register at the same time. The exported register function is hence synchronised on a fresh mutex lock.

The code for a client is even simpler:

```
val [url, name] = CommandLine.arguments ()

structure Server = unpack take url : SERVER
do Server.register (proxy TextIO.print)

fun loop () = case TextIO.inputLine TextIO.stdIn of
                | NONE ⇒ OS.Process.exit OS.Process.success
                | SOME line ⇒
                    (Server.broadcast (name ^ ": " ^ line); loop ())
do loop ()
```

It expects a server URL and a user name on the command line, registers with the server, and simply forwards everything typed by the user to the server. Note that the call to register is a proxy call, passing another proxy as argument, thereby establishing the bi-directional connection.

11. Futures and Concurrency

An important feature of Alice ML we have ignored so far are *futures* [27, 19]. Futures are place-holders for undetermined values. A thread encountering a future blocks implicitly until the value is available. This is known as *data flow synchronisation*, and requires that futures are *transparent*, meaning that they do not form a special type, and any value can potentially be a future. This gives orthogonal support for asynchronicity and enables elegant formulation of a wide range of concurrency abstractions.

Efficiently integrating futures requires extension of the abstract store model. We have to throw in one additional kind of node:

- *Future*, represents a future in the data graph. Depending on the kind of future [27], future nodes may have at most one child node, e.g. a closure in the case of a lazy future.

A future node will *morph* into another kind when the value is determined. It may also morph into another future, enabling chaining. Internal to the abstract store, morphing is realised by marking the future node as a *forward* to the actual node, similar to techniques found in the implementation of logic languages [2]. Forwards are

transparently followed by the store abstraction, and garbage collection performs path compression by removing forwards.

In order to maintain the closedness property – and avoid the significant complications of distributed state at system level – futures are never pickled. Instead, whenever futures are encountered, the pickler *touches* them, i.e., synchronises and does not proceed before all futures have been determined.

Apart from the store level, no fundamental changes are necessary to the described system architecture to reconcile pickling with futures. However, to maximise module-level laziness – and specifically, to support lazy linking – we have to insert lazy suspensions in three places of the previously described design:

- *Type representations* are computed lazily, by wrapping every reified declaration into a lazy future. That is, we change the translation scheme from Figure 3 to

$$\begin{array}{ll} [\![\textbf{type}\ \overline{\alpha}\ tycon = ty]\!] & = \textbf{val}\ tycon\$\ =\ \textbf{lazy}\ [\![\lambda\overline{\alpha}.ty]\!] \\ [\![\textbf{signature}\ sigid = sigexp]\!] & = \textbf{val}\ sigid\$\ =\ \textbf{lazy}\ [\![sigexp]\!] \end{array}$$

- *Structure access* (cf. Section 7.3) have to be treated as lazy projections, i.e., an identifier A.B.C maps to (a hoisted instance of) **lazy** #C(#B A). An open declaration is considered sugar for a sequence of projecting declarations, and thus likewise is lazy.
- *Imports* should be performed lazily, which is achieved by wrapping the unpack into a lazy future:

import *spec* **from** url $\rightsquigarrow$
open (**lazy unpack** link url : **sig** *spec* **end**)

Care also has to be taken to make all layers thread-safe. In particular, a stateful representation of runtime types requires caution: because types are implemented in ML land, atomicity cannot be assumed for type checks, and at the same time, the type graphs can potentially be shared between different ML threads. Consequently, algorithms that mutate parts of a type graph for optimisation purposes have to make sure that no other thread can ever see it in inconsistent state. In the Alice system, we strategically insert temporary futures in the graph to achieve fine-grained locking of subgraphs during mutation.

11.1 Modules and Types

Futures in Alice ML may not only occur on the level of terms, but also on the level of modules. Since runtime types may be computed from types stemming from modules that are futures, this also induces a notion of *type futures*. In particular, Alice ML's lazy linking strategy gives rise to a notion of *lazy types* [18] as a pervasive phenomenon. Lazy types are touched only by explicit or implicit uses of unpack or by pickling, but may, through modules, trigger arbitrary term-level computations.

Here is a simple example of a lazy computation that is triggered through a dynamic type check:

```
structure M = lazy struct do print "Now." type t = int end
val p = pack (val it = 5) : (val it : int)
structure X = unpack p : (val it : M.t)
```

The structure M is constructed lazily. However, the dynamic type check performed by unpack has to match int against M.t, which requires the representation of M.t. The type checker hence touches the lazy future representing M, thereby forcing evaluation. In practice, such cases primarily arise in conjunction with lazy linking (as defined by the modified import expansion above), where type checking one import may require loading of a seemingly unrelated one, because some type information is imported transitively.

Since pickling forces futures, performing it can be another cause for triggering lazy computations through type futures. The following variation of the above example demonstrates this:

```
structure M = lazy struct do print "Now." type t = int end
val p = pack (val it = 5) : (val it : M.t)
do Pickle.save("five", p)
```

Pickling the package p involves pickling the type M.t from its signature, and will hence trigger M.

Thanks to the implementation of runtime types as regular ML values, realising lazy type semantics requires no extra effort at all – it is simply reflected by the appearance of term-level futures in the type data structures.

11.2 Thread Collection and Thunkification

Naturally, every thread possesses its own stack. Like everything else in the Alice runtime, stacks and thread objects are allocated in the store. This allows the Alice VM to garbage collect threads that block on futures that are not reachable anymore and thus can never be determined.

In principle, this set-up would also allow us to support thread *thunkification*. To do so, it would suffice to provide an operation that delivers the current continuation of a thread, i.e. callcc. The representation of a captured continuation would contain a reference to the stack, and consequently, pickling it would amount to thunkifying the thread and enable passing it to another process. However, we have not pursued this possibility, because we are concerned about the risk of (silently) breaking stateful abstractions with such a feature, especially locks.

12. Pickle Verification

We argue that the design we have presented so far ensures type safety for pickles in the sense that there is no way to corrupt the runtime system from *within* the language: for any pickle written by the system, unpickling is always safe, regardless of type mismatches. However, there remains the obvious possibility that pickles are forged *outside* the control of the system (this may include the use of the library's low-level I/O subsystem). For those cases, the mechanisms presented so far are insufficient to maintain type safety. In other words, we protect against accident, but not against malice. To distinguish these cases, we also speak of *internal* vs. *external* type safety.

To achieve external safety, we not only have to check type-correct *usage* of pickles, we also have to check their inherent *consistency*. In analogy to Java bytecode verification, we call such a check pickle *verification*.

So far, the Alice system implements only a very limited amount of verification: the raw unpickler checks that the pickle code is a valid description of a store data graph. This is sufficient to capture most practical cases of erroneous pickles, but obviously does not protect against malicious attackers who craft bogus pickles that form a valid store graph description, but where that graph does not comply to the higher-level type system.

Verification of Alice ML pickles is non-trivial. In particular, it is more difficult than the byte code verification performed in languages like Java for at least two reasons:

1. Java semantics includes dynamic checks at individual method calls, so that much pressure is taken from verification and instead shifted to runtime errors. For Alice ML, full static type safety would have to be established.
2. Pickles do not only include compiler generated code but also dynamically computed data. In effect, verification must hence be able to type-check arbitrary portions of the heap.

Despite these difficulties, we believe that the architecture described in this paper can be reconciled with verification. To see why, let us first make three observations:

1. The raw pickling service ensures that a pickle can be transformed into a valid data graph. Hence, verification can be performed by a higher layer inspecting the resulting graph.
2. Verification amounts to type *checking* a data graph; the outermost type to check against is always known, hence no *reconstruction* is necessary.
3. Because pickles are closed, type checking always is performed against an empty environment.

These points imply that checking can be performed by a directed algorithm that propagates type information inwards in order to check subgraphs. Consequently, in most cases, no additional type annotations are needed in the data graph itself. We identify the following exceptions:

- *Functions.* Obviously, type checking would require a typed code format and thus *typed compilation* (i.e., the translations given in Sections 6–7 had to be refined). The internal type system would have to be sufficiently expressive to embrace our compilation of modules. While not straightforward, we see no principal problem in achieving this. The code typing also gives the types of respective closure environments.
- *Abstract Types.* To check values of abstract type, type names must be mapped to their representation types. This requires maintaining a type heap [23], either explicitly, or more easily, implicitly by embedding it pointwise into the representation of abstract type names with constructor $\mathsf{new} : \mathsf{int} \times \mathsf{ty} \to \mathsf{tycon}$.
- *Exception Constructors.* Similarly to abstract types, the constructor heap has to be represented in typed form, in order to derive the argument type of constructed values. Again, the most obvious way to do this is pointwise, i.e. by hooking the type information into the representation of individual constructors.

Note that *packages* already contain the necessary type information for the embedded module.

Looking at the list, we are positive that verification would not require substantial changes to our set-up. The only major change is the requirement for a typed code format. The remaining bits of additional runtime type information seem easy and cheap to get. We leave exploration of these ideas for future work.

13. Related Work

Many of the individual techniques we build upon are influenced by previous work. We only mention the most prominent systems here, more comprehensive comparisons of existing mechanisms for pickling and distribution can be found in [14] and [26].Unfortunately, surprisingly little has been published on the actual implementation of pickling services, so that most of the discussion is limited to what can be gathered from system documentation and experimentation.

CLU, Modula-3 and Java. The first pickling mechansim in a programming system was developed in the context of CLU [12]. Only "transmissible" types could be pickled and no type information was included. Programmers had to provide transformation functions to make abstract types transmissible. CLU later inspired similar mechanisms for the object-oriented languages Modula-3 [7] and Java [22]. Neither of these mechanisms meets all the requirements stated in Section 1.2. In particular, they remain limited with respect to higher-orderness, type safety and portability. Only Java ensures the latter, and it performs verification on class files. Class files can be transmitted separately to simulate higher-orderness, as done by Java's remote method invocation (RMI) [33], but that is fragile and significantly weakens static guarantees.

Oz/Mozart. The closest relative to Alice ML with respect to its focus on pickling is the Oz/Mozart system [11], also featuring higher-order, platform-independent pickles, a component system, and a first-class compiler. As Oz is a dynamically checked language however, no type safety can be guaranteed. No verification is performed either, such that bogus pickles can crash the system despite dynamic checking. On the other hand, pickling in Mozart is more expressive than in Alice ML, providing distributed futures. The price is a significantly more complex semantics and language implementation with multiple modes of pickling and a strong need for distributed garbage collection, which we wanted to avoid.

Erlang. Erlang [5] is a dynamically checked distributed language for embedded telecommunications systems. Processes can be spawned on different nodes in a network and communicate through channels, using copying like proxies. Although Erlang is a higher-order language, functions cannot directly be communicated. Erlang primarily targets embedded systems, consequently it is not concerned with security or inhomogeneous networks.

ML. In the world of typed functional programming, both Standard ML of New Jersey as well as Objective Caml feature pickling mechanisms. In SML/NJ, pickling is used in separate compilation [4] and reuses the garbage collection infrastructure. Objective Caml provides a library module `Marshal` [15], which allows pickling of arbitrary values (except objects). In neither system pickling is safe, portable, or higher-order, although Objective Caml enables pickling of functions as pointers into the address space of the program, which limits portability to the exact same program.

HashCaml [6] wraps Ocaml's marshaller with abstraction-safe runtime type checking. Unlike our design, it requires type passing polymorphism. Types are represented as simple hashes externally, hence subtyping is not supported.

ML derivatives. Acute [29] is an ML-based language for distributed programming that is closest to Alice ML and provides a similar generic pickling mechanism. Unlike in Alice ML, pickling is not separated from dynamic typing, and all inter-process communication hence dynamically typed. Also, Acute supports implicit rebinding of resources, which we exclude for security reasons. As in the current Alice system, no pickle verification is possible.

Facile [32] extended Standard ML with facilities for concurrency and distributed programming inspired by the π-calculus. To achieve dynamic connectivity, Facile requires taking an indirection through a central *structure server*, which allows making ML structures persistent. A structure is retrieved from the server by requesting a module with a suitable signature, which naturally implies a form of dynamic signature check. If several structures match a given signature, the last one stored is returned.

JoCaml [9] takes a similar stake as Facile, but extending Objective Caml and with concurrency being based on the richer Join Calculus. Communication is type-safe but limited to monomorphic values and no longer higher-order in the most recent version.

Ohori developed a typed translation of high-level inter-process communication operations into low-level primitives in an ML-like language [20].

Clean. The only functional language with a type-safe form of persistence is Clean, which features high-level I/O based on dynamics [21]. However, pickles are neither higher-order nor portable.

Pickler combinators. Kennedy implements a limited form of pickling in form of a combinator library written in ML [13]. Although surprisingly flexible, a library approach is bound to fail on most of our requirements: combinators are neither universal (especially, they cannot support higher-orderness), nor can they guarantee properties like transparency or closedness. Efficiency also is a major concern: maintaining sharing requires extra effort and can only be achieved for a statically bounded number of types, because it needs an explicit environment per type.

14. Conclusion

Alice ML and the Alice Programming System have been designed with a universal pickling service in mind, from ground up. Pickling in this system is higher-order and typed, it embraces persistence, code mobility and dynamic modularity in a type-safe manner.

We have shown how such a pickling service can be implemented in a modular architecture based on layered abstractions. The layering enables separation of concerns, each layer is small and comparably easy to implement. The main innovations lie in (1) the structure of code with function granularity and embedded higher-order values, which dynamic separate compilation takes advantage of, (2) the higher-order representation of components, and (3) the avoidance of type passing polymorphism. The approach has been proved practical in the existing implementation of the Alice system, which has been ported to x86, AMD-64 and PowerPC architectures, with full interoperability.

Both language and implementation considerably benefit from the expressiveness of pickling and the modular architecture underlying it. Alice ML provides a level of dynamicity that – to the best of our knowledge – is not available in any other compiled, statically typed programming language system, including those with much more lax type systems.

The main direction for future work is the integration of full-scale pickle verification. While verification may not be required in most cases of *local* distributed programming, it seems a necessity if we want to truly embrace the idea of *open* programming, where we have to deal with untrusted principals.

Another interesting direction would be to extend the pickling service to support *incremental* or *lazy* pickling and unpickling, so that it could be used to realise large structured persistent databases with incremental updates to selected subgraphs. However, it is far from obvious how this could be achieved.

References

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *16th Symposium on Principles of Programming Languages*, pages 213–227, Austin, USA, Jan. 1989.

[2] H. Ait-Kaci. *Warren's Abstract Machine*. Logic Programming. MIT Press, 1991.

[3] Alice Team. *The Alice System*. Programming Systems Lab, Saarland University, `http://www.ps.un-sb.de/alice/`, 2003.

[4] A. Appel and D. MacQueen. Separate compilation for Standard ML. In *Conference on Programming Language Design and Implementation*, pages 13–23. ACM Press, June 1994.

[5] J. Armstrong. *Making Reliable Distributed Systems in the Presence of Software Errors*. Doctoral dissertation, Royal Institute of Technology, Stockholm, Sweden, Dec. 2003.

[6] J. Billings, P. Sewell, M. Shinwell, and R. Strniša. Type-safe distributed programming for OCaml. In *Workshop on ML*, Portland, Oregon, USA, Sept. 2006.

[7] A. Birrell, G. Nelson, S. Owicki, and E. Wobbler. Network objects. *Software – Practice and Experience*, 25(S4), Dec. 1995.

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

[9] S. Conchon and F. Le Fessant. Jocaml: mobile agents for objective-caml. In *First International Symposium on Agent Systems and Applications and Third International Symposium on Mobile Agents*, pages 22–29, Palm Springs, USA, Oct. 1999.

[10] K. Crary, S. Weirich, and G. Morrisett. Intensional polymorphism in type erasure semantics. *Journal of Functional Programming*, 12(6):567–600, Nov. 2002.

[11] D. Duchier, L. Kornstaedt, C. Schulte, and G. Smolka. A higher-order module discipline with separate compilation, dynamic linking, and pickling. Technical report, Programming Systems Lab, Universität des Saarlandes, 1998. Draft, `http://www.ps.uni-sb.de/papers`.

[12] M. Herlihy and B. Liskov. A value transmission method for abstract data types. *Transactions on Programming Languages and Systems*, 4(4):527–551, Oct. 1982.

[13] A. Kennedy. Pickler combinators. *Journal of Functional Programming*, 14(6):727–739, Nov. 2004.

[14] L. Kornstaedt. *Design and Implementation of a Programmable Middleware*. Doctoral dissertation, Saarland University, 2006.

[15] X. Leroy. *The Objective Caml System*. INRIA, 2003. `http://pauillac.inria.fr/ocaml/htmlman/`.

[16] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[17] C. Müller. Run-time byte code compilation, optimization, and interpretation for Alice. Diploma thesis, Saarland University, 2006.

[18] G. Neis. A semantics for lazy types. Bachelor's thesis, Universität des Saarlandes, Saarbrücken, Germany, Sept. 2006.

[19] J. Niehren, J. Schwinghammer, and G. Smolka. A concurrent lambda calculus with futures. *Theoretical Computer Science*, 364(3):338–356, Nov. 2006.

[20] A. Ohori and K. Kato. Semantics for communication primitives in a polymorphic language. In *20th Symposium on Principles of Programming Languages*, pages 99–112. ACM Press, 1993.

[21] M. Pil. First class file I/O. In W. Kluge, editor, *8th International Workshop on Implementation of Functional Languages*, volume 1268 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[22] R. Riggs, J. Waldo, A. Wollrath, and K. Bharat. Pickling state in the java system. *Computing Systems*, 9(4):291–312, 1996.

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

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

[25] A. Rossberg. The missing link – dynamic components for ML. In *11th International Conference on Functional Programming*, Portland, Oregon, USA, Sept. 2006. ACM Press.

[26] A. Rossberg. *Typed Open Programming – A higher-order, typed approach to dynamic modularity and distribution*. Doctoral dissertation, Universität des Saarlandes, Saarbrücken, 2007.

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

[28] SEAM Team. Simple extensible abstract machine, 2004. `http://www.ps.uni-sb.de/seam/`.

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

[30] Z. Shao, C. League, and S. Monnier. Implementing typed intermediate languages. In *3rd International Conference on Functional Programming*, pages 313–323, Baltimore, USA, Sept. 1998.

[31] G. Tack, L. Kornstaedt, and G. Smolka. Generic pickling and minimization. In *Workshop on ML*, volume 148(2) of *Electronic Notes in Theoretical Computer Science*. Elsevier, Mar. 2006.

[32] B. Thomsen, L. Leth, and T.-M. Kuo. A Facile tutorial. In *7th International Conference on Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*, pages 278–298. Springer, 1996.

[33] A. Wollrath, R. Riggs, and J. Waldo. A distributed object model for the Java system. In *2nd Conference on Object-Oriented Technologies & Systems*, pages 219–232. USENIX Association, 1996.