Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 148 (2006) 79–103
www.elsevier.com/locate/entcs

Generic Pickling and Minimization

Guido Tack[1] Leif Kornstaedt[1] Gert Smolka[1]

Programming Systems Lab
Saarland University
Saarbrücken, Germany

Abstract

This paper presents generic pickling and minimization mechanisms that are provided as services similar to garbage collection. Pickling is used to externalize and internalize data. Minimization means to maximize the sharing in arbitrary data structures. The paper introduces the notion of an abstract store as a formal basis for the algorithms, and analyzes design decisions for the implementation aspects of pickling and minimization. The mechanisms presented here are fully implemented in the Alice programming system.

Keywords: pickling, marshalling, serialization, graph minimization, persistence, distributed programming

1 Introduction

Pickling, also known as marshalling or serialization, means to externalize data at runtime. This mechanism is found in many programming languages, from Java, over .NET and Python to Mozart, Caml, and Alice, to name only a few. This paper presents a language-independent architecture for pickling.

The Alice ML language is a conservative extension of Standard ML, enriching the language with support for concurrency, lazy evaluation, and a rich component system. In higher-order languages such as Alice, both the component system and network distribution can be based on pickling. We therefore investigate pickling as a fundamental mechanism of a programming system.

[1] Email: {tack,kornstaedt,smolka}@ps.uni-sb.de

doi:10.1016/j.entcs.2005.11.041

The basis for the discussion of pickling is the *abstract store*, a language-neutral memory model that can be used as the foundation of a virtual machine design. The pickler is, like a garbage collector, introduced as a generic service of the abstract store. All data, including code, is represented in the abstract store and can hence be pickled.

The abstract store provides an interface to a graph of primitive objects like integers, strings, or structured nodes connected to other objects. Apart from pickling, we introduce graph minimization as another service of the store. This is particularly well suited for a functional programming language, as many objects in the store are purely functional and benefit from a larger amount of sharing.

The Alice system [2] implements Alice ML in a virtual machine that is based on an abstract store with garbage collection, pickling, and minimization services.

Contributions of the paper

The abstract store presented in this paper is a model for the heap of a programming system that is both abstract enough for formal reasoning and concrete enough for efficient implementation. Based on this model, we present a detailed description of a generic, platform- and language-independent pickling service. We define its pickle language, develop algorithms for pickling and unpickling, and show how pickling integrates with ML-like languages. We further discuss what it needs to make pickling well-defined in a concurrent language such as Alice. The paper introduces general graph minimization as a novel service of abstract stores. We give formal as well as implementation details and show that such a service is both efficient and worthwhile.

Organization of the paper

In the next section, we give an overview of how pickling integrates into the Alice language and programming system. Section 3 introduces a formal model for the store, upon which the pickling service is built in Sections 4 and 5. Section 6 develops the minimization service. After that, in Section 7, we extend the model and algorithms with support for concurrency as present in Alice. Finally, we discuss related work in Section 8 and conclude and present ideas for future work in Section 9.

2 Pickling in Alice

The Alice ML language extends Standard ML in a conservative way to support *typed open programming* [24]: programs are built from components that can be

dynamically created and dynamically loaded, from files or over the network. Components can be created statically by the compiler as well as dynamically, in the language. This requires three main features:

(i) Components must be able to contain arbitrary values (including functions and hence code).

(ii) As they are acquired from statically unknown sources, components must be dynamically type checked.

(iii) Dynamic creation of components calls for a mechanism to externalize components into a format that is suitable for persistent storage or network transmission.

The first two points are addressed by an extension of the SML module system and dynamic type checks. The third point describes what is known as pickling, marshalling, or serialization.

2.1 Modules and packages

A key feature of Alice is that the components comprising a program are loaded and linked dynamically. Alice components are based on a rich module system and support for dynamic type checks to ensure integrity of the runtime system.

A well-known way to introduce dynamic type checks into an otherwise statically typed language are *dynamics* [1]. Arbitrary values can be injected into a universal type `dyn`. Values of this type carry enough runtime type information for a projecting type-case operation.

For Alice, we modified this concept slightly: *packages* (as dynamics are called in Alice) contain modules. The Alice module system extends Standard ML modules with support for higher-order functors and local modules, as well as nested and abstract signatures. Runtime type-checking amounts to matching the *package signature* against a static one. As packages are proper Alice values, they introduce first-class modules into the language. The semantics of packages is formally specified in [23].

A package is basically a pair of a module and its signature. It is created by injecting a module, expressed by a structure expression *strexp* in SML:

```
pack strexp : sigexp
```

The signature expression *sigexp* defines the package signature. The inverse operation, projection, is provided by the module expression

```
unpack exp : sigexp
```

This takes a package and extracts the contained module, provided that the package signature matches the target signature denoted by *sigexp*. Otherwise, an exception is thrown.

2.2 Pickling and unpickling

Packages are used primarily as the basis of the Alice component system. Alice incorporates a notion of component that is a refinement of the system found in the Oz language [8]. A component is an externalized package that can be stored in a file or transferred over a network. This externalization of packages is called *pickling*, the reverse operation *unpickling*.

Alice provides pickling to files through the library structure `Pickle`:

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

The `save` operation writes a package to a file with the given name. The semantics of pickling requires that a pickle comprises the whole transitive closure of a package:

```
val p = let
           fun f x = List.append (x,x)
           structure S = struct val b = (f,"b") end
        in
           save ("pickle.alc",
                 pack S : sig val b : (α list → α list) × string end)
        end
```

The resulting pickle contains the complete closure of `S`, including the code for `f` and the `List.append` function, but not the complete `List` structure. The package can be unpickled from a file using `load`:

```
structure SCopy = unpack Pickle.load "pickle.alc" :
                  sig val b : (α → α) × string end
```

Pickling and unpickling preserve the structure of the pickled value, in particular sharing and cycles. When the unpickled package is unpacked, the usual dynamic type check is performed. If type checking succeeds, all operations on the unpickled and unpacked module are *statically* type safe again.

3 An Abstract Store

All data that an Alice ML program uses and produces at runtime is represented in the *abstract store* (or store, for short). In this section, we formally define the abstract store and exemplify how Alice data is represented.

The abstract store is a general model of a store for a high-level programming language. We use Alice as an example, because the store was originally designed with Alice in mind. All abstractions however readily carry over to other programming languages.

3.1 A formal model of the store

Formally, an abstract store is an ordered, labelled graph with two types of nodes: scalars and structured nodes.

Definition 3.1 Given a set of addresses Adr, a set of labels Lab and the set of byte sequences BSeq, an *abstract store* is a finite function

$$g \in \mathrm{Adr} \rightharpoonup \mathrm{Lab} \times (\mathrm{BSeq} \uplus \mathrm{Adr}^*)$$

Intuitively, a store maps addresses to labelled nodes, which can either be scalars or structured nodes with a sequence of edges to other nodes. We call scalar store nodes *chunks*, and structured nodes *blocks*. The set of all stores is denoted with *Store*.

For simplicity, we do not consider an optimized representation for certain scalar types such as integers. This definition of a store captures only the static part – the graph represents a snapshot of the store at some point in time. Programs of course modify the store dynamically, by creating new nodes and redirecting edges of existing nodes.

3.2 Representation of Alice data in the store

All data like integers, strings, or tuples can be mapped upon objects in the abstract store in a straight-forward way.

Integers are chunks with label int.

Strings are chunks with label string.

Tuples are blocks with label tuple and edges to the tuple's components. Tuples are also used to represent values of algebraic datatypes.

Code is data. This is the fundamental assumption that makes the store uniform and allows us to present pickling and minimization in terms of nothing but the store. Possible representations include chunks with byte code, or nested tuples for abstract syntax trees.

3.3 Stateful data

Stateful data such as references or arrays is implemented in the same way as stateless data. However, equality on stateful data is defined in terms of *token equality*: two references are equal if they actually denote the same memory cell, whereas two tuples are equal if their structure is the same and their components are equal (structural equality). To support this, we partition the set of labels: $\mathrm{Lab} = \mathrm{Lab}_{\mathrm{structural}} \uplus \mathrm{Lab}_{\mathrm{token}}$. This will allow us to treat stateful data differently during pickling and minimization.

3.4 Garbage collection

Automatic garbage collection means, in terms of the abstract store, to remove nodes from the graph that are unreachable from a distinguished *root node*. The root node can for example be the stack, from which all live data should be referred to in some way. Formally, garbage collection is a function

$$gc \in \mathit{Store} \times \mathrm{Adr} \to \mathit{Store} \times \mathrm{Adr}$$

such that $gc(s, \mathit{root}) = (s', \mathit{root}')$ and s' is isomorphic to the subgraph of s reachable from root, and root' is the root node in s'. This formal model resembles copying garbage collection.

3.5 Implementation

The Alice system is based on Seam, the simple extensible abstract machine library [6]. Seam provides an abstract store that closely implements the abstractions introduced in this section. For efficiency reasons, integers are optimized, they do not actually require a node that is allocated in the store. Seam features a generational garbage collector and has built-in support for pickling and minimization, the store services that are discussed in the following sections.

4 Basic Pickling and Unpickling

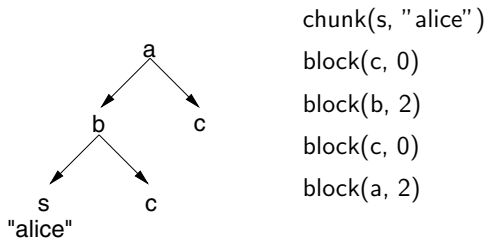
We can now define what pickling and unpickling means in the context of abstract stores: to transform a subgraph of a store into a platform-independent, external format, and to construct a subgraph that is equivalent to the original in a possibly different store.

We first develop the unpickler together with a suitable pickle language. In a second step, we show how a pickle of a subgraph of an abstract store can be generated.

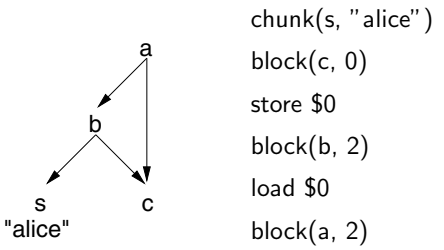
4.1 The pickle language

We regard a pickle as a sequence of instructions for an interpreter that constructs the corresponding graph in the store. The language will be presented as a bnf grammar, but it should be clear that this language is simple enough to correspond directly to a sequence of bytes. In the actual implementation, both pickler and unpickler do not produce or consume an intermediate, datatype-like term, but binary data.

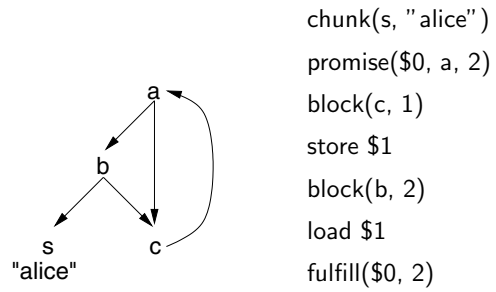
We will require the pickle language to be able to express sharing in the store as well as cycles. Cycles are ubiquitous in typical Alice data structures.



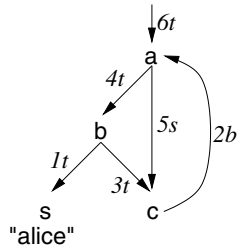
(a) A Tree



(b) An acyclic graph



(c) A cyclic graph



(d) A DFS-annotated store

Fig. 1. Graphs in an abstract store and the corresponding pickles

Note that, as opposed to classical ML implementations, Alice data can contain stateless cycles that are not established through ref cells.

4.2 Constructing graphs in the store: unpickling

For the unpickler, we begin with a pickle language for trees and then generalize to acyclic and finally cyclic graphs.

The usual way to construct trees is in a bottom-up fashion using a stack machine. A pickle language for constructing trees in an abstract store needs the following instructions:

$$\begin{array}{lcl} \langle \mathit{pickle} \rangle & ::= & \langle \mathit{instr} \rangle\ \langle \mathit{pickle} \rangle \mid \epsilon \\ \langle \mathit{instr} \rangle & ::= & \mathsf{block}(\mathit{label}, \mathit{arity}) \\ & \mid & \mathsf{chunk}(\mathit{label}, \mathit{string}) \end{array}$$

An instruction $\mathsf{block}(l, n)$ pops n nodes off the stack, constructs a new node with label l and these n children, and pushes the new node on the stack. Scalar nodes are constructed by the chunk instruction. After successful construction of the whole tree, the root node is the only item on the stack. Figure 1(a) shows a tree in an abstract store and the pickle from which it can be built.

Bottom-up construction also works for directed, acyclic graphs (DAGs), but the interpreter must be extended by an unbounded set of *registers*: Each

node that has more than one predecessor – these nodes are called *shared nodes* – must be placed in a register when it is first built, and loaded when it is needed again. We extend the pickle language with the corresponding instructions:

⟨*instr*⟩ += store *register*
| load *register*

You can find a DAG with its corresponding pickle in Figure 1(b). For clarity, registers are written with a leading $.

We cannot construct cyclic data yet, because constructing a node requires all its children to be already constructed. Hence, cycles must be made explicit:

⟨*instr*⟩ += promise (*register*, *label*, *arity*)
| fulfill (*register*, *arity*)

A promise instruction creates a node that promises to become one node on the cycle eventually. Using a promise as a placeholder, all the other nodes on the cycle can be built before the promise is *fulfilled* by a corresponding fulfill instruction. Promises may only be fulfilled once. A cyclic graph with the pickle constructing it is shown in Figure 1(c).

The unpickler implements the promise and fulfill instructions through *backpatching*: A promise for a node with label l and arity n constructs a node with that label and arity but arbitrary "dummy" children. Fulfilling this promise then simply means replacing the dummy children by the real children. The unpickler needs both label and arity when promising a node, so the promise instruction gets these as arguments. For fulfilling, it needs to find the previously created promise, so both instructions require a register.

We omit the details of the low-level pickle format, as it is a straightforward mapping from instruction sequences to byte sequences.

4.3 Optimized pickle language

The pickle format can be optimized for the resulting pickle size, for efficient unpickling, or for efficient pickling. If pickles are used mainly for persistent storage, pickle size and efficient unpickling are important, because pickles are written once but read often. For network transfer however, pickling can become the bottleneck.

To optimize the pickle size, the store and block/chunk instructions can be combined into store block and store chunk instructions.

The unpickler can benefit from knowing the maximum stack height it needs for unpickling as well as the number of registers used in a pickle, because then the stack and register bank do not have to be able to grow dynamically.

One could employ liveness analysis and a simple register allocator to minimize the number of needed registers. We have not yet further investigated

⟨*pickle*⟩	::=	⟨*instr*⟩ ⟨*pickle*⟩ | ϵ
⟨*instr*⟩	::=	⟨*store*⟩ block(*label*, *arity*)
	|	⟨*store*⟩ chunk(*label*, *string*)
	|	load *register*
	|	promise (*register*, *label*, *arity*)
	|	fulfill (*register*, *arity*)
⟨*store*⟩	::=	store *register* | ϵ

Table 1
The pickle language

this, as we think that the benefit of slightly fewer registers does not justify the overhead in the pickler.

The pickle format we present must make a compromise between genericity and compactness of the pickle language. A less generic system may use type information to make pickles more compact and provide specialized external representations for common language entities.

To summarize, Table 1 shows the full pickle language.

4.4 Pickling subgraphs of a store

The remaining problem is to generate a pickle of the above format from a subgraph of the store. If unpickling means *constructing* a graph according to given instructions, pickling means *traversing* the graph in the same order and generating these instructions. The pickle can be seen at the same time as instructions for the unpickler, and as a protocol of the graph traversal.

4.4.1 Sharing and cycles

Maintaining sharing and cycles is a vital property of the pickler we want to develop. Although detecting sharing and cycles causes an overhead during pickling, it is essential for making pickling efficient: pickles become bigger without sharing of nodes, an exponential blowup is possible. Cyclic data structures cannot be represented without sharing, and trying to pickle them would diverge.

4.4.2 Depth first search

A standard algorithm for graph traversal is *depth first search* (DFS), as invented by Tarjan in 1972 [27]. We will now show that the pickle format introduced above is similar to the protocol of a postorder, depth-first left-to-right traversal of a graph.

Trees are easy to pickle: their pickles correspond exactly to the sequence

of nodes in a postorder left-to-right DFS traversal. For DAGs and arbitrary graphs, the pickle corresponds to a postorder sequence of the *edges*. Depth first search can order and classify the edges in a graph:

Tree edges are those edges that DFS traverses when first visiting a node.

Back edges are edges that lead from a node to one of its ancestors in the DFS tree.

Sharing edges are the remaining edges.

The edges are ordered according to the time they are visited (preorder) and the time DFS finishes the edge (postorder). Nodes are marked as unshared if they have only one predecessor, promised if they have at least one incoming back edge, and shared otherwise. The following definition extends abstract stores with DFS annotations on the edges:

Definition 4.1 A *DFS-annotated* abstract store is a finite function

$$g \in \mathrm{Adr} \rightharpoonup \mathrm{Lab} \times \{\mathsf{unshared},\mathsf{shared},\mathsf{promised}\} \times (\mathrm{BSeq} \uplus (\mathrm{Adr} \times \mathbb{N} \times \{\mathsf{tree},\mathsf{back},\mathsf{sharing}\})^*)$$

Figure 1(d) shows a postorder-DFS-annotated abstract store. Tree, back, and sharing edges are abbreviated with t, b, and s. Node annotations are not shown because they are clear from the incoming edges. For regularity, the root node has an additional incoming edge. It is a tree edge and gets the highest postorder number.

In a DFS-annotated store, each edge corresponds to one pickle instruction. Pickling amounts to reading off these instructions in the postorder of the edges.

A tree edge to an unshared node v corresponds to a block or chunk instruction, depending on v.

A tree edge to a shared node v needs a store in addition to the block or chunk.

A tree edge to a promised node v becomes a fulfill. Note that the type of v is block, as only blocks can lie on a circle.

A back edge to a node v corresponds to a promise if it is the back edge to v with the lowest postorder number, otherwise to a load.

A sharing edge corresponds to a load instruction.

4.4.3 Interleaving DFS annotation and linearization

DFS can be used to generate the sequence of pickle instructions directly. As a consequence, the DFS-annotated store is never constructed in memory.

The only difficulty is to determine whether a tree edge points to an un-

shared or a shared node. This information is not available when DFS assigns the postorder number to a tree edge. All other edges are not critical: sharing edges are visited after the tree edge pointing to the same node, resulting in a load after the corresponding store. For back edges, DFS can determine whether it is the first back edge to that node, and hence whether a promise or load must be generated. Tree edges to promised nodes are visited after the promise was generated and can hence produce a fulfill instruction.

To solve the problem with tree edges to not promised nodes, we apply a two-phase approach: the first phase creates the instruction sequence without any store instruction, and at the same time maintains an ordered list of pickle program offsets where store instructions must be inserted afterwards. The second phase then creates the actual pickle by copying the pickle program and inserting the store instructions where necessary.

An alternative approach would be to insert store instructions for every tree edge. With the combined store block instructions, this does not increase the pickle size. However, many more registers are needed in the unpickling interpreter. For data structures with little sharing (such as lists), the overhead is linear in the size of the pickle.

4.4.4 Pickling has the desired semantics

Pickling and unpickling produce copies of a subgraph of the store, much like copying garbage collection does, or on a lower level the operating system's virtual memory system. And just as copying garbage collection does not change the semantics of the copied data, pickling followed by unpickling is – in the simplest case – an identity function. If there are no stateful nodes and no nodes with transform labels, the result of pickling a subgraph and unpickling it is indistinguishable from the original.

With stateful data, the difference between original and unpickled copy can be observed. In Alice, this is how the semantics of pickling is currently defined. If a different semantics is required, stateful nodes must not be pickled.

The second source of difference between original and unpickled copy is the transformation mechanism. Here it is the responsibility of the implementor of abstraction and instantiation functions to make sure that the behavior of these functions fits the language semantics.

4.4.5 Preorder pickling, top-down unpickling

Pickling in postorder is non-standard. Besides Alice, only Python [21] seems to use postorder linearization and bottom-up construction. The dual approach to postorder pickling and bottom-up unpickling is to use a preorder linearization of the edges, which then corresponds to a top-down construction of the graph.

A preorder linearization of the edges requires a slightly different pickle language and interpreter: the promise and fulfill instructions are not necessary, because every node is constructed *before* its children. The interpreter needs to back-patch the children into the parent node once their are constructed.

The difference between the two approaches is the kind of recursion that is needed for pickling and unpickling. Bottom-up unpickling needs a simple recursion, constructed nodes are pushed on the stack and remain there until their parent is constructed. Top-down unpickling needs a more complex stack, because a node is pushed before its children are constructed, and must be popped after the last child is constructed. The situation for pickling is dual: postorder pickling requires a more complex stack than preorder pickling. This means that the two approaches are equally efficient only if each pickle is read exactly once. As soon as unpickling is more frequent than pickling, the bottom-up approach is superior.

4.4.6 Right-to-left traversal

Depth-first search as presented here usually traverses the children of a node from left to right (from the first child to the last). Some data structures, for example lists in ML, are right-recursive. Left-to-right construction as assumed here requires stack in the size of the list, whereas right-to-left construction could manage with constant stack space. It may therefore make sense to use right-to-left traversal and construction for certain nodes (depending on the label). For Alice, experiments have shown that generally using a right-to-left strategy can significantly reduce unpickling time.

4.4.7 Breadth-first search

Breadth-first search (BFS) is the algorithm of choice for the classic abstract store service, copying garbage collection. The Cheney algorithm [7] uses the store area to which it copies the live nodes as a queue, the control data structure for BFS. The garbage collector does not need any additional memory.

Pickling and unpickling can be based on a variation of Cheney's algorithm. The pickler copies the nodes of a subgraph using a Cheney scan, replacing the real addresses in the abstract store with abstract addresses, counting from the start of the memory the graph is copied to. The unpickler can then use another Cheney scan to recreate a copy of the graph, mapping the abstract addresses back to real addresses.

For pickling, the advantage of not using any additional memory disappears: Cheney's algorithm needs *forwarding pointers* to mark already copied nodes. A pickler cannot mark the nodes in place, as they do not become garbage after pickling. It needs additional memory to store that information. The unpickler

cannot mark the nodes directly in the pickle either. Still, no additional control data structure (stack or queue) is needed.

A complication is that during BFS, it is hard to control when exactly the construction of a subgraph of the pickle is complete. We will need this information to implement transformations as introduced in the next section.

5 Extensions to Pickling and Unpickling

In this section, we extend pickling to deal with objects that should not be pickled and to apply transformations during pickling and unpickling.

5.1 Resources

A resource is a node in the store that does not have a well-defined meaning outside that store. The classical examples are file descriptors or window handles. Resources must not be pickled. This is easily achieved by partitioning the set of labels into labels for resources and labels for picklable data. The pickler throws an exception if it finds a resource. The pickler does not try to rebind resources (as discussed in [9,16]). Rebinding could however be realized with the transformation mechanism introduced next.

5.2 Transformation

The important design goal for pickling is platform independence. On the other hand, the important design goal for the implementation of an abstract store must be efficiency. The pickler and unpickler so far generate exact copies of subgraphs of the store. This means that the store may not optimize the representation of data in a platform-dependent way.

An optimized internal representation is however desirable; for instance floating point numbers have different efficient representations on big- and little-endian platforms. The solution is a generic transformation mechanism built into the pickler and unpickler.

To this end we introduce special transform store labels and assume that for each such label lt, there is a pair of functions $\text{abstract}_{lt} \in \text{Adr} \to \text{Adr}$ and $\text{instantiate}_{lt} \in \text{Adr} \to \text{Adr}$ [2].

The abstract_{lt} function is applied to a node v with label lt when the pickler descends into it. The function then computes the abstract, external representation v' of v, and the pickler descends into v'. The unpickler applies

[2] In a realistic setting, these functions can of course construct new nodes in the store.

instantiate$_{lt}$ after constructing a node v with label lt. Instantiation produces a node v' that the unpickler then uses for further construction instead of v.

Note that both abstraction and instantiation suit the postorder pickling and bottom-up unpickling approach well: abstraction can happen before visiting a node for the first time, and instantiation after completing its construction. In a preorder and top-down scenario, the unpickler must introduce a placeholder for each node with a transform label, unpickle the node and its subtree, and then fill the placeholder with the result of the instantiation. This emulates a bottom-up construction using placeholders.

5.2.1 Just-in-time compilation

An advanced application of the transformation mechanism is just-in-time compilation: an external, platform-independent code representation is compiled to an efficient, internal representation like optimized byte code or native machine code.

In the current implementation, the Alice compiler generates a component containing code in an intermediate format called Alice Abstract Code. Code is tagged with a transform store label, and on unpickling, the abstract code is compiled to byte code or native machine code. For true just-in-time compilation, code is not compiled when loaded but when first executed. In Alice, we can use lazy futures (see Section 7) to accomplish this in a straightforward way: instead of instantiating the unpickled value, a lazy suspension is created that will perform the necessary transformation when the value is first accessed. This has the additional benefit that the actual transformation can be fully concurrent and interrupted by garbage collection, whereas instantiation in the unpickler must be atomic.

Instantiation often cannot be undone easily. It is for example impossible to recover Alice Abstract Code from machine code. In this case, instantiation must keep a copy of the external representation that abstraction can then simply return.

6 Minimization

A pickle as introduced previously represents exactly the nodes of the subgraph of the abstract store that was pickled. One could say that a pickle is automatically garbage collected, as only reachable nodes are pickled. To further reduce the pickle size, *redundant* nodes can be removed. This section defines formally which nodes are redundant, relates it to graph minimization algorithms, and presents further application areas apart from pickling.

6.1 Redundancy in the abstract store

A node in the abstract store is redundant if it is indistinguishable in the language from some other node in the store. Consider the example of balanced binary trees:

```
datatype t = N | T of t*t
val a = T( T (N,N), T (N,N) )
val b = let val c = T(N,N) in T(c,c) end
```

The two trees are semantically indistinguishable. However, one of the children of the root of `a` is redundant, as the tree `b` shows. The tree `b` shares the two subtrees of the root node. For trees of depth d constructed in this manner, not sharing anything results in $2^d - 1$ nodes being allocated, while complete sharing only requires d nodes. One will hence always try to design data structures with minimal redundancy.

This is of course hard to achieve, especially from a more global point of view. Values are typically constructed independently, often in different components of the system, and there is no central authority to control sharing. Another limitation, for which there is no workaround as it is immanent in the language, is that values of different types can never be subject to sharing, although they may be *represented* internally in the same way.

6.2 Minimizing acyclic graphs: hash-consing

If the data structures under consideration do not contain cycles, or if all cycles are established through stateful nodes (as for instance in SML), one can employ a technique called *hash-consing* to maximize sharing in the store. Upon construction of a node v in the store, a hash value of v is computed and v is stored in a table. If a node w with the same hash value is constructed, a reference to v is used instead of actually allocating w. The cost of maximizing sharing is hence amortized during the construction of nodes.

Hash-consing was first introduced in LISP systems. Appel and Gonçalves discuss how to integrate hash-consing with the SML/NJ garbage collector [3]. Their system only hashes nodes that have survived at least one garbage collection, such that the cost of hashing is only paid for longer-lived nodes. Another application area of hash-consing is the efficient representation of type information in intermediate languages [25,26].

6.3 General graph minimization

In the following, we develop a general minimization service on the level of the abstract store. The algorithms are based on automaton and graph minimization and remove all redundant nodes in a given subgraph of the abstract store.

For Alice, it is important that minimization can handle cyclic data, because stateless cycles are ubiquitous: the representation of runtime types, the code representation as well as function closures may contain such cycles.

The definition of minimization relies on a definition of equivalence of nodes in an abstract store. This is captured by an *equality relation* $\sim \in \mathrm{Adr} \times \mathrm{Adr}$ that is defined to satisfy the following conditions:

(i) If $v \sim v'$, $g(v) = (l, \_)$ and $l \in \mathrm{Lab}_{\mathrm{token}}$, then
- $v = v'$

(ii) If $v \sim v'$, $g(v) = (l, s)$ and $s \in \mathrm{Str}$, then
- $g(v') = g(v)$

(iii) If $v \sim v'$ and $g(v) = (l, \langle v_0, \ldots, v_n \rangle)$, then
- $g(v') = (l, \langle v'_0, \ldots, v'_n \rangle)$ and
- $\forall i \in \{0, \ldots, n\} : v_i \sim v'_i$

The union of two equality relations is again an equality relation. The relation $\sim_=$, which is defined as $v \sim_= v' :\Leftrightarrow v = v'$, is an equality relation. Let Eq be the set of all equality relations. Then $\sim_S$ is defined as the (non-empty) union of all equality relations:

$$\sim_S = \bigcup_{\sim \in Eq} \sim$$

As equality relations are closed under union, $\sim_S$ is the unique largest equality relation. It exactly expresses *semantic equality* in an abstract store: two nodes cannot be distinguished except for their address. It is necessary to consider the largest such relation, because any smaller relation does not yield the required equivalences for cyclic graphs.

Semantic equality in the store as defined by $\sim_S$ is strictly stronger than semantic equality in Alice:

- Values of non-eqtypes can be considered equal. In particular, values of functional type can be considered equal.
- Values of different types can be considered equal, as long as their representations allow this.
- Cyclic values can be considered equal, although the equality operator applied to them would diverge.

Given a subgraph g, minimization computes the equivalence classes of g with respect to $\sim_S$. We can then use this information to replace each node in g by a unique representative of its equivalence class. The resulting graph g' is minimal in the sense that for two nodes w and w' in g', $w \sim_S s'$ implies $w = w'$.

Graph minimization is a well-know problem: first algorithms for minimizing finite state automata can be found in textbooks from the sixties. In 1971 Hopcroft gave an algorithm with $O(n \log n)$ worst case time complexity [13]. His algorithm easily generalizes to ordered graphs.

The main idea is to refine an initial equivalence relation such that it is compatible with the graph structure: if $v \sim v'$, then for the successor w_i of v at edge number i, v' has a successor w_i' at edge i such that $w_i \sim w_i'$. Such an equivalence relation is called a *congruence*.

If we start with an equivalence that makes exactly the nodes with different labels different, it is easy to see that graph minimization will yield a congruence that describes exactly semantic equivalence of nodes.

To actually minimize the graph, a representative r of each congruence class c must be chosen, all edges to the other members of c redirected to r, and all other members of c deleted.

Our implementation of the minimizer is based on the generic approach of partition refinement as described by Habib et al. [10]. A combination of Hopcroft's algorithm with these ideas can be easily adjusted to the abstract store model. It yields an efficient general purpose minimization service.

6.4 Stateful data

In order to be used in non-pure languages, the minimizer must be able to deal with stateful data. Stateful data, that is store nodes whose edges can be redirected, must remain unique. This follows directly from the fact that in the following example, a and b are distinguishable in the language, while a' and b' are not:

```
val a  = ref 3
val b  = ref 3
val a' = SOME 3
val b' = SOME 3
```

Graph minimization only *refines* the initial equivalence relation, it therefore suffices to make all stateful nodes distinct in the initial partition, and they will remain distinct after minimization.

The minimizer is available in the Alice standard library as a function `minimize : 'a -> unit`. It takes an arbitrary Alice value and minimizes its representation in the store.

6.5 Areas of application

Minimization can be seen as a form of *semantic garbage collection*: it does not reclaim nodes that are unreachable, but nodes that are redundant. As for any

form of garbage collection, one has to trade off the runtime it requires against the benefits of a smaller memory footprint.

It is imaginable to integrate the minimizer with the garbage collector, much like Appel and Gonçalves did [3]. The minimizer can be applied to only certain areas of the store, or only at a major collection. We have not yet implemented this scenario, and the benchmarks provided by Appel and Gonçalves suggest that the space savings were "not very impressive". The situation may be different for Alice, though, because the data that is minimized is rather different from the data in SML: not only runtime data, but all components of the system are kept in the heap, including their code representation. This may provide more potential for the minimizer.

The most straightforward application is to always minimize before pickling. A pickled value is supposed to be long-lived, so investing into minimization may pay off. The Alice compiler and static linker provide options for minimizing the compiled components before pickling them, and this has proved to be feasible as well as worth the effort (see the benchmarks in the next section).

Minimization together with a test on token equality provide a structural equality test for cyclic data:

```
fun cyclic_equal(x,y) = (minimize (x,y); token_eq(x,y))
```

Other applications are more problem-specific. We are currently investigating how the runtime representation of dynamic types (as required for packages) can benefit from minimization. What is especially interesting here is the fast structural equality test: in a minimized representation, two types are equal if and only if they are represented by the same node.

6.6 Benchmarks

As a benchmark, we have compiled the Alice library, consisting of 818 components, with and without minimization:

	component size	*pickle size*	*compilation time*
not minimized	104.29	12.58	22m36s
minimized	71.80 (69%)	12.51 (99%)	22m26s (99%)

The component size is the size (in megabytes) the components take up in the store. The minimizer can shrink the components by 30%. Pickle size (also in megabytes) measures the file size of the pickles. The pickler uses gzip compression to further shrink pickle file sizes, and in compressed format, minimized pickles are only slightly smaller than ordinary pickles. This is due to the fact that gzip compression also eliminates redundancy, and there is less redundancy to eliminate in minimized pickles.

The last column is the time needed to compile all components. It shows

that minimization does not incur any runtime overhead, on the one hand because it is fast, on the other hand because the compiler benefits from loading smaller components for type checking.

All Alice programs benefit from the minimized library, because of both reduced startup times and a smaller memory footprint.

The benchmarks were made on a AMD 64 3000+ with 512MB RAM, running Debian Linux for AMD64. The exact numbers for the component sizes will differ on a 32bit platform, but the relative compression will remain the same.

7 Concurrency and Laziness

Alice is a concurrent language, and the fundamental concept that concurrency is based on is that of a *future*. A future is a placeholder for a value that is not yet known, such as the result of a concurrent computation or a lazy evaluation. Futures provide data-flow synchronization: a thread that needs to access a future is implicitly blocked until the future is determined. Futures were first introduced in Multilisp [11]. A formal semantics of futures can be found in [19].

As an example, consider a function `f : int -> int` that performs some complex computation on integers. We can evaluate it in a new thread with the expression `val a = spawn f 3`. This immediately returns a future, and `f 3` is evaluated concurrently. If we now access `a`, as for instance in the expression `val b = a + 3`, this thread will block until the result of the evaluation of `f 3` is available. The result will replace the future, and the thread evaluating `b` automatically resumes.

Laziness is similar to concurrent evaluation: the expression `val a = lazy f 3` immediately evaluates to a *lazy* future. Only when the value is actually needed, for example in the expression `val b = a + 4`, the application `f 3` is evaluated and the future is bound to the result.

7.1 Futures in the abstract store

The future semantics requires that a future is *replaced* with a value when it is bound. To make this replacement efficient, futures are built into the abstract store in the form of nodes with special labels: Lab $\supseteq$ {fut,lazy}.

When a concurrent future is created, a block of arity 1 with label fut is constructed. Replacing a future f with a value v amounts to a become operation on the store: the label of the node representing f is changed to the internal label fwd (a forwarding node), and the edge is redirected to point to

the node representing v. All other operations on the store follow chains of fwd nodes transparently, and garbage collection performs a path compression of these chains.

7.2 Concurrent pickling and unpickling

Pickling and unpickling as described so far work well for sequential languages. In the concurrent setting that we have in Alice, however, we must take care of futures.

Unpickling is not critical: the root of the constructed subgraph becomes available only after the graph has been fully constructed.

For pickling, however, we require a *snapshot semantics*: the pickle shall represent the subgraph at one particular moment in time. The simplest solution is to regard futures as resources and just not pickle them. In a language like Alice, where futures are ubiquitous due to lazy linking, this would however render pickling considerably less useful.

For lazy futures, there are two possible approaches: force their evaluation, or pickle the complete lazy suspension (which is also represented in the store). Pickling concurrent futures would amount to pickling threads, which is more involved and will not be discussed here. We assume that the evaluation of concurrent futures must be forced during pickling.

If we consider only stateless graphs, futures guarantee that the graph grows monotonically. Forcing their evaluation during pickling will hence result in a snapshot of the graph after pickling.

Once we have a cloning semantics for pickling stateful data, a concurrent pickler cannot make sure that the graph does not change during pickling. Pickling itself hence must be atomic. Evaluation of futures can still be forced: the pickler leaves atomic mode to wait for the future to be bound, and later starts over the whole atomic pickling process. This can of course imply that pickling does not terminate.

The Alice pickler is eager, it forces evaluation of lazy futures and waits for all futures to be bound. Because of lazy linking, a lot of futures whose suspensions contain resources are in the system, and components could not easily be pickled. This has the trade-off, however, that lazily constructed infinite data structures cannot be pickled. Thus, it would be desirable to be able to choose the pickling mode per data structure.

7.3 Concurrent minimization

Like the pickler, the minimizer must be able to deal with futures. It has two options to do so: ignore futures (treating them like stateful data) or force their

evaluation in the same way pickling does (such that minimization is atomic and always minimizes a snapshot).

Our implementation currently copies our pickler's behavior. As for the pickler, one could provide a non-forcing mode for the minimizer, too.

In conjunction with pickling, requesting the futures is necessary. Otherwise, pickling itself will request the futures, and the pickle may not be minimal. It is however not clear if this is always the desired semantics: a data structure that would benefit from minimization but at the same time contains a lot of lazy futures may actually get *bigger* if the minimizer requests all futures.

8 Related work

Birrell et al. [5] coined the term pickling in a paper about small databases. Pickling mechanisms in one form or the other can be found in many programming systems. The first published pickling mechanism for a programming language is that of CLU by Herlihy and Liskov [12]. It already contains most of the essential ideas, like customization of the pickling process and a discussion of safety properties. The Modula-3 language [18] inherited some of the concepts from CLU and also comes with a pickling mechanism.

Mozart/Oz[28] uses pickling for persistence and distribution of arbitrary data, including the so called functors, which implement the Oz module system. Mozart pickles can hence contain code. Alice inherits many ideas from Mozart.

SML/NJ uses pickling to generate binary modules for separate compilation [4]. Pickling is provided through the `Unsafe` library structure. As the name suggests, unpickling performs no type checks, the behavior is undefined if types do not match. A look at the source code reveals that the pickler uses the garbage collector's Cheney scan to copy a subgraph to a fresh memory area, and then adjusts pointers and optimizes the representation of some objects (as was discussed in the section on BFS in this paper).

Objective Caml [20] offers a library module `Marshal` which allows for persistence and distribution of all values except objects. Code is not pickled directly but only as a pointer into the address space. Pickling is thus not platform independent (as the Caml compiler produces native machine code); in fact a pickle containing function pointers is only valid if loaded from the same binary that produced it. The pickler has support for detecting resources and rejects to pickle them. Pickling is unsafe, as unpickling performs no dynamic type check.

Kennedy presents a library of pickler combinators [15], with implementations in Haskell and SML. It is implemented in the language itself, and for each type that is supposed to be pickled, a specific pair of pickling and unpickling

functions must be implemented. This approach only works for ground values, functions cannot be pickled. Cyclic data can be dealt with, if the cycle is established through a ref cell. Kennedy uses an idea similar to hash-consing to deal with sharing: the algorithm maintains a dictionary of the byte sequences of values already pickled, and if a byte sequence appears again, a reference to the original sequence is pickled instead. The effect is a minimization of the pickle.

The most prominent implementation of pickling can be found in Java, where pickling is called *serialization*. It is described in detail in the *Java Object Serialization Specification* [14]. There also is a compact introduction to the internals [22], revealing the roots of Java serialization in CLU and Modula-3. Java serialization differs from pickling as we present it: only the state of the object graph is serialized, not the object definitions (the classes, and hence the code). Pickling can be customized in the language, one can cut off branches or define the external (binary) representation of an object.

Microsoft's .NET Framework [17] offers a library module that provides a serialization mechanism similar to Java's. In contrast to Java, it clearly separates customization of the pickling process from the resulting pickle format, and it provides different pickle formats (binary, XML, and SOAP [29]).

The object-oriented scripting language Python [21] implements two distinct methods of serialization: one mechanism can pickle code, but no cyclic data structures and does not preserve sharing between objects. Another mechanism can cope with cycles and sharing, but not with code. The Python pickler is the only system besides Alice that is known to employ a postorder/bottom-up strategy.

This list is not exhaustive, because most high-level programming languages provide some implementation of persistence or distribution. Most of them do not specify any details though and concentrate on the high-level features like type-safety and customization.

There has been some work on maximizing sharing of ML runtime data structures. Most efforts are based on a variation of hash-consing. This solves a problem that is different from what our minimizer does, as only acyclic data can be shared. This is feasible in a setting where all cycles go through ref cells which may not be shared anyway. Appel and Gonçalves describe how to integrate hash-consing with the garbage collector [3].

The MLTon system features a library function `share : 'a -> unit`, very similar to the `minimize` function introduced here. The mechanism is implemented as a mark-compact garbage collector extended with hash-consing and cannot minimize cycles. Only fixed-size objects are hashed.

9 Conclusion and Future Work

We presented a generic pickling and minimization mechanism. We showed how Alice, as a conservative extension of Standard ML, uses pickling in a type-safe way for its component system. To build a formal base for the algorithms, we introduced abstract stores as a universal memory model. Unpickling and pickling are based on this model, allowing us to analyze and evaluate our design decisions such as bottom-up versus top-down unpickling and right-to-left versus left-to-right traversal. Minimization can be used to decrease the size of pickled data. However, the general mechanism presented here seems suitable for other applications such as efficient representation of runtime types. Finally, we extended the system with support for concurrency as present in Alice. We analyzed how pickler and minimizer must behave in such a concurrent setting.

9.1 Conclusions

The abstract store presented in this paper is a powerful layer of abstraction, both for implementation as for formalizing aspects of a programming system. It is the basis of advanced store services, pickling and minimization. Pickling as presented here is especially well suited for functional programming languages, where the closure semantics and the ability to pickle code is essential. We have shown that minimization, as a store service, helps reduce pickle sizes considerably.

All the concepts introduced here were implemented in the Alice programming system, proving that the presented design decisions lead to an efficient implementation.

9.2 Future work

Future work on the pickling mechanism must address the lower-level safety issues: how can a pickle be verified on the byte level? The directions to explore include cryptographic methods and advanced techniques such as typed pickles, typed abstract stores, and proof carrying code.

The minimizer suggests to explore more data structures that would benefit from automatic minimization. Our exploration of the representation of dynamic type information will give more insight into this. Coupling the minimizer with the garbage collector may be worth a try – especially if this is done in the presence of special store areas that are always kept minimal.

Acknowledgements

We would like to thank Andreas Rossberg for helpful comments on a draft of this paper, and Marco Kuhlmann for fruitful discussions about the presentation of the pickling algorithm. The main ideas of the paper were developed for Guido Tack's Diploma Thesis that was supervised by his co-authors. We thank the anonymous reviewers for their constructive comments.

References

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

[2] *The Alice Project*, Available from http://www.ps.uni-sb.de/alice (2005), homepage at the Programming Systems Lab, Universität des Saarlandes, Saarbrücken.

[3] Appel, A. W. and M. J. R. Gonçalves, *Hash-consing garbage collection*, Technical Report CS-TR-412-93, Princeton University (1993).

[4] Appel, A. W. and D. B. MacQueen, *Separate compilation for Standard ML*, in: *Proceedings of the ACM SIGPLAN '94 conference on Programming language design and implementation* (1994), pp. 13–23.

[5] Birrell, A., M. Jones and E. Wobber, *A simple and efficient implementation of a small database*, in: *Proceedings of the eleventh ACM Symposium on Operating systems principles* (1987), pp. 149–154.

[6] Brunklaus, T. and L. Kornstaedt, *A virtual machine for multi-language execution*, Technical report, Programming Systems Lab, Universität des Saarlandes, Saarbrücken (2002), available from http://www.ps.uni-sb.de/Papers/abstracts/multivm.html.

[7] Cheney, C. J., *A nonrecursive list compacting algorithm*, Communications of the ACM **13** (1970), pp. 677–678.

[8] Duchier, D., L. Kornstaedt, C. Schulte and G. Smolka, *A higher-order module discipline with separate compilation, dynamic linking, and pickling*, Technical report, Programming Systems Lab, DFKI, and Universität des Saarlandes, Saarbrücken, Germany (1998), Available from http://www.ps.uni-sb.de/Papers/.

[9] Fuggetta, A., G. P. Picco and G. Vigna, *Understanding code mobility*, IEEE Transactions on Software Engineering **24** (1998), pp. 342–361.

[10] Habib, M., C. Paul and L. Viennot, *Partition refinement techniques: An interesting algorithmic tool kit*, International Journal of Foundations of Computer Science **10** (1999), pp. 147–170.

[11] Halstead, R. H., *Multilisp: a language for concurrent symbolic computation*, ACM Transactions on Programming Languages and Systems (TOPLAS) **7** (1985), pp. 501–538.

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

[13] Hopcroft, J., *An $n \log n$ algorithm for minimizing states in a finite automaton*, in: Z. Kohavi and A. Paz, editors, *Theory of Machines and Computations* (1971), pp. 189–196.

[14] *Java Object Serialization Specification*, Available from http://java.sun.com/j2se/1.4/docs/guide/serialization (2001).

[15] Kennedy, A., *Pickler combinators.*, Journal of Functional Programming **14** (2004), pp. 727–739.

[16] Leifer, J. J., G. Peskine, P. Sewell and K. Wansbrough, *Global abstraction-safe marshalling with hash types.*, in: C. Runciman and O. Shivers, editors, *ICFP* (2003), pp. 87–98.

[17] Microsoft, *Microsoft .NET*, Available from http://www.microsoft.com/net (2003).

[18] Nelson, G., editor, "Systems Programming with Modula-3," Prentice Hall Series in Innovative Technology, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[19] Niehren, J., J. Schwinghammer and G. Smolka, *A concurrent lambda calculus with futures*, in: B. Gramlich, editor, *5th International Workshop on Frontiers in Combining Systems*, Lecture Notes in Computer Science **3717** (2005), pp. 248–263.

[20] *The OCaml programming system*, Available from http://www.ocaml.org (2005).

[21] *The Python programming language*, Available from http://www.pyhton.org (2005).

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

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

[24] Rossberg, A., 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* (2005).

[25] Shao, Z. and A. W. Appel, *A type-based compiler for Standard ML*, in: *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation* (1995), pp. 116–129.

[26] Shao, Z., C. League and S. Monnier, *Implementing typed intermediate languages*, in: *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming* (1998), pp. 313–323.

[27] Tarjan, R. E., *Depth first search and linear graph algorithms*, SIAM Journal on Computing **1** (1972), pp. 146–160.

[28] The Mozart Consortium, *The Mozart programming system*, Available from http://www.mozart-oz.org (2005).

[29] World Wide Web Consortium, *Simple Object Access Protocol (SOAP)*, Available from http://www.w3.org/TR/SOAP (2000).