Universität des Saarlandes
Fachrichtung 6.2 – Informatik

Diplomarbeit

Linearisation, Minimisation and Transformation of Data Graphs with Transients

Guido Tack

Mai 2003

Betreut von
Prof. Dr. Gert Smolka
und
Dipl.-Inform. Leif Kornstaedt

Erklärung

Hiermit erkläre ich, Guido Tack, an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Saarbrücken, den 16. Mai 2003

Abstract

This thesis introduces data graphs as a formal model for the objects in a programming system's memory, and describes three services on such data graphs: linearisation, minimisation, and transformation.

The SEAM system offers an *abstract store* that provides a programming language's implementor with a platform- and language-independent abstraction layer, hiding the complex issues of memory management. This thesis aims at giving a formal description of this store and of the services mentioned above.

Data graphs are presented here as a formal model for the objects that reside in such a store. Starting from this model, an abstract store can be described by an abstract data type (ADT) that implements data graphs as imperative objects.

The *linearisation* service (also known as "pickling") translates a data graph into a linear, external, platform-independent representation (a pickle) from which a copy of the original graph can be reconstructed. Pickles can be written to files for persistence, or distributed over a network, implementing inter-process communication. Linearisation and delinearisation are described formally in terms of the data graph, and the SEAM implementation of pickling and unpickling is discussed.

Minimisation applies graph minimisation techniques to data graphs, yielding a store service that eliminates redundancy in the graph. The formal background as well as implementation issues of this service are presented, and its applicability to data graphs is evaluated.

Finally, the store is extended by transients, a mechanism essential for an efficient implementation of futures, logic variables and lazy evaluation. Transients are applied to both minimisation and linearisation; for the latter, they allow for the implementation of a powerful *transformation* mechanism that translates between an internal and an external representation of data graphs during pickling and unpickling.

Acknowledgements

I would like to thank Gert Smolka and Leif Kornstaedt for supervising this project. Most of the formal work is based on long discussions with Gert Smolka; I am grateful for the time he invested and the way he helped me think in new directions. Leif Kornstaedt was responsible for keeping this work grounded, he helped me synchronising the formal models with the SEAM architecture. Both my supervisors read drafts of this thesis, their comments improved my understanding of scientific writing quite a lot.

Leif Kornstaedt, Thorsten Brunklaus and Andreas Rossberg answered all my questions concerning Alice and the SEAM architecture; without their help I would not have been able to implement the algorithms presented in this thesis.

I am grateful to Marco Kuhlmann, mainly for being a friend, but also for many detailed comments on drafts of this thesis, and for sharing his LaTeX experience.

I have to thank everyone at Programming Systems Lab for creating an atmosphere that has made me enjoy working here for the last two and a half years.

Finally, I would like to thank my family for the support through all my life, and Monika Schwarz for being there.

Contents

1 Introduction

Computer memory can be regarded at different levels of abstraction: On the physical hardware level, there are chips and transistors. The operating system provides the programmer with a “logical memory” that hides these low-level issues and looks like a continuous array of words, indexed also by words.

Programming languages all have their own memory model. C and C++, for example, adhere to the array view of memory, but provide syntactic support for objects of more than one word in size (and the corresponding address arithmetics). In other programming languages like Java or SML, the underlying memory architecture is completely abstracted away, the programmer only knows how objects can be created and manipulated, rather than how they are represented in memory.

1.1 Overview

In this thesis, I present an *abstract store*, an interface between low-level and high-level memory. This store interface is independent of platform and programming language and hides the complex low-level issues connected with memory management. On top of this interface, high-level programming languages can be implemented. Figure 1.1 illustrates these levels of abstraction.

Java / SML / Alice
Abstract Store / SEAM
C, C++
Array of Words
Hardware Memory

Figure 1.1: Computer memory, levels of abstraction

The motivation for this thesis was to put the implementation of a virtual machine for the Alice project [2] on solid formal ground. I have implemented the algorithms described here for the SEAM [5] system (Simple Extensible Abstract Machine – the virtual machine implemented for Alice), and the formal models closely fit its store architecture.

Data Graphs and Abstract Stores

In the context of garbage collection, the state of the memory is often described by a *data graph* [19]. I extend the concept of a data graph to a formal model for the objects in an abstract store. Chapter 2 presents this model.

In Chapter 3, I develop an abstract data type (ADT) that implements imperative data graphs. This ADT is the abstract store mentioned above, and its interface resembles the SEAM store interface. The abstract store provides several services, the most well-known is probably garbage collection. I investigate three other services in this thesis, the linearisation, minimisation and transformation of data graphs.

Both data graph model and abstract store are language-independent. They provide a generic abstraction that can be used by *language layers* to implement a store that provides the abstractions needed by the specific programming language.

As one application of the data graph model, I describe how the equality type of some value (whether it can be compared using structural or token equality) can be annotated directly on the data graph level. For these annotated data graphs, I define an equivalence relation expressing semantic equality, which is needed for the minimisation service.

Linearisation

One of the abstract store services I present is *pickling*, a means of linearising parts of the data graph. A data graph in linearised form (a pickle) can be used to reconstruct (unpickle) an isomorphic copy of this data graph, possibly in a different abstract store.

The pickle contains an external, platform-independent representation of the data graph. It can be written to a file from where the graph can be reconstructed at a later point in time; this is often called *persistence*. As another application of pickling, the pickled graph can be interchanged between two processes, for example via a network connection. That way, inter-process communication can be realised.

This thesis discusses what pickling means formally in Chapter 4, describing it in terms of the data graph and relating it to standard graph algorithms. Furthermore, I discuss the details needed to implement pickling in the SEAM system. Three other programming systems that also provide pickling – Java, SML/NJ and Mozart – are compared to the solution I present here.

Minimisation

Another service that is investigated is the *minimisation* of data graphs. Minimisation in this context means to maximise the sharing of equivalent subgraphs. This makes minimisation a form of garbage collection, as it reduces the "structural garbage" in an abstract store.

It is known that graph minimisation is closely related to automaton minimisation. So far, this technique has not been applied to data graphs. In Chapter 5 of this thesis, I present a minimisation algorithm in the context of data graphs with annotated equality types and discuss the main issues of an efficient implementation. For one sample application, the minimisation of pickles, I give concrete benchmark results that suggest that pickles offer a significant potential for minimisation while the overhead of minimising the data graph is reasonably small.

Transients

A transient is a store node that can be dynamically made equal to another store node; all of the node's incoming edges are redirected to the other node. Based on this mechanism, lazy evaluation, futures and logic variables can be explained.

This thesis presents transients in the data graph formalism and the modifications to the abstract store that are necessary to support transients efficiently. Furthermore, minimisation and pickling are reconsidered: Both services can benefit from transients. The minimisation algorithm can delete more redundant nodes if it is implemented using transients. In the case of pickling, transients lead to a powerful transformation mechanism. All these issues are discussed in Chapter 6.

Transformation

The transformation mechanism allows to define internal and external representations for certain nodes in the data graph. That means that the abstract store contains a data graph in its internal representation, but if this data graph is pickled, it will be automatically converted into an external format. Unpickling applies the reverse transformation to yield again an internal data graph.

The most important application of the transformation mechanism is the pickling of code: A pickle has to store code in a platform-independent way. On unpickling, a transformation can be applied that compiles the abstract, external code for example to native code for the current platform. Pickling the code must again produce the external, platform-independent version.

1.2 Related Work

A model similar to data graphs is often used in the context of garbage collection, for example by Jones and Lins [19]. Scheidhauer [34] introduces a similar model for the data structures in the store of the Mozart virtual machine.

A lot of high-level programming languages offer some kind of linearisation mechanism, beginning with CLU [16] over Modula-3 [27], Java [18], and Microsoft .NET [24] to SML/NJ [3], Mozart [9] or Alice [2].

Work on minimisation has first been done in the context of finite automata (for example by Hopcroft [17]), and was later extended to graphs by Cardon and Crochemore [7].

Transients can be found in many languages: Smalltalk's [11] `become` primitive is based on similar techiques, logic programming languages like Prolog [1] or Mozart use transients to implement logic variables, and functional programming languages implement lazy evaluation (for example in Haskell [20]) or futures (as in Multilisp [14] or Alice [35]) using transients.

Transformations similar to the ones I present were already investigated in CLU, where they were implemented on the level of the language itself. In Java, the user can customise what fields of an object will be serialised, this is a weak form of transformation. Transformation as I describe it is based on work by Brunklaus and Kornstaedt [5] in the SEAM context.

1.3 Contributions of this Thesis

Data graphs in conjunction with the abstract store are powerful models: I describe all services in this thesis in terms of data graphs, and the correspondence between data graph and abstract store leads to direct implementations of all algorithms.

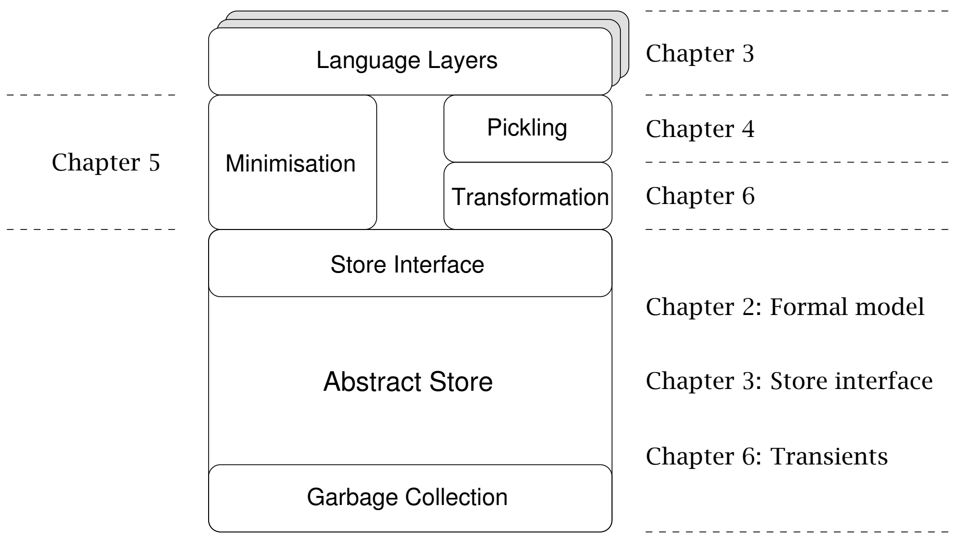
I present a careful analysis of the pickling service, relating it to standard algorithms for graphs and trees. The bottom-up approach I follow in this thesis seems to be natural, but unique: Most other picklers (CLU, Modula-3, Java, Microsoft .NET, Python, Mozart) use a top-down pickling strategy.

This thesis presents a completely new store service, minimisation. Although the techniques are several decades old, graph minimisation has not yet been applied to data graphs in such a generic, platform- and language-independent way. The key to minimisation is the annotation of the equality type on the node level.

Brunklaus and Kornstaedt already described a transformation mechanism for Alice/SEAM. This was however tightly coupled with the pickler and unpickler. Understanding pickling better made it easy to separate it from the transformations. I can now describe the transformation mechanism as just another layer of abstraction between abstract store and pickler/unpickler.

1.4 Organisation of this Thesis

This figure shows the layers of the abstract store and the services built on top of it. It also illustrates how the remaining chapters of this thesis are organised.



2 The Data Graph

This chapter introduces the formal notion of a *data graph*. It is the model used throughout the remaining chapters of the thesis.

Data graphs are models for the objects that a program builds in memory while it is running. The definition given here is abstract enough to be independent of a specific platform or programming language. Data graphs are suited well as a model for the data structures that are needed to implement a programming system, and to describe platform- and language-independent algorithms that implement generic services of programming systems.

The services discussed in this thesis (pickling, minimisation, and transformation) are all explained in terms of data graphs. While this chapter deals with the formal model, the next chapter introduces an abstract data type (called abstract store) that implements data graphs, yielding a direct correspondence between formalisation and implementation.

2.1 Definition of a Data Graph

A data graph is basically a *multi-graph with ordered edges*, or simply an *ordered graph*.

Definition 2.1 (Ordered graph)
An ordered graph is a function g such that

$$Ran(g) \subseteq Dom(g)^*$$

If the function g is finite, the graph is called finite, too.

An element of $Dom(g)$ is called a *node*, and a pair of a node v and a natural number n is an *edge* if the nth component of $g(v)$ is defined. An ordered graph is thus a function mapping nodes to a tuple of their successors. Figure 2.1 illustrates this

v	$g(v)$
0	$\langle 1, 2\rangle$
1	$\langle 2\rangle$
2	$\langle 3\rangle$
3	$\langle 1\rangle$

$$\begin{tikzcd}
& 0 \arrow[dl] \arrow[dr] & \\
1 \arrow[rr] & & 2 \arrow[dl] \\
& 3 \arrow[ul] &
\end{tikzcd}$$

Figure 2.1: An ordered graph

with an example where $Dom(g) = \{0, 1, 2, 3\}$. The edges are implicitly numbered from left to right.

For a data graph, some more structure is needed. Given a set Lab of so called *labels* and a set Str of so called *strings*, a data graph can be defined:

Definition 2.2 (Data graph)
A data graph is a finite function g such that

$$Ran(g) \subseteq \mathsf{Lab} \times (\mathsf{Str} \uplus Dom(g)^*)$$

The strings reflect the fact that scalar data is usually treated differently from the other nodes.

A node v' is a *successor* (the ith successor) of a node v in a data graph g, written $v \to_g v'$ ($v \to_{i,g} v'$), if and only if there exists some $i \in \mathbb{N}$ such that $g(v) = (l, \langle \dots, v_i, \dots \rangle)$ and $v_i = v'$. If $v \to_g v'$, v is called a *predecessor* of v'. The set of (ith) successors of a node u is defined as $\mathsf{succ}(u) = \{v | u \to_g v\}$ ($\mathsf{succ}_i(u) = \{v | u \to_{i,g} v\}$).

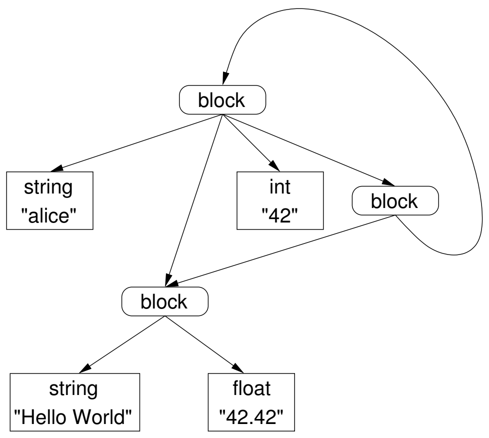
A node v' is *reachable* from a node v in g if and only if there exist nodes $v_1, \dots, v_n$ such that $v = v_1 \to_g v_2 \to_g \cdots \to_g v_n = v'$. In this case, v is an *ancestor* of v'.

Figure 2.2(a) shows an example of a data graph. Here, the set Lab is assumed to be $\mathsf{Lab} = \{\mathsf{int}, \mathsf{string}, \mathsf{float}, \mathsf{block}\}$.

A data graph can be seen as a mapping from addresses to nodes, which are labelled objects with outgoing edges. Such a mapping is just what one would expect an abstract store to provide: A means to access the content (in this case the label and string) and successors of a node through its address.

Figure 2.2(b) shows the data graph from Figure 2.2(a) seen as a store.

The algorithms given in Chapters 4 and 5 operate on *subgraphs* of the data graph:



(a) A data graph

address	*label*	*contents*
0	block	∣ 3 ∣ 4 ∣ 5 ∣ 6 ∣
1	string	"Hello World"
2	float	"42.42"
3	string	"alice"
4	block	∣ 1 ∣ 2 ∣
5	int	"42"
6	block	∣ 4 ∣ 0 ∣

(b) ... seen as a store

Figure 2.2: Data graphs

Definition 2.3 (Subgraph)
A data graph g' is a subgraph of a data graph g if and only if

$$g' = g|_A \textit{ for some } A \subseteq \mathit{Dom}(g)$$

As this definition requires g' to be a data graph as well, the subgraph is *closed*, meaning that every node occuring as a successor of a node in $\mathit{Dom}(g)$ is itself in $\mathit{Dom}(g)$; g' is the subgraph that is reachable from the nodes in A.

Until now, the domain of g was not restricted in any way. As a model for objects in an abstract store, however, it seems natural to assume that $\mathit{Dom}(g)$ is a set of addresses (a subset of the natural numbers).

Rooted Data Graphs

Based on Definition 2.2, a more specialised form of data graphs can be defined:

Definition 2.4 (Rooted data graph)
A data graph g is called rooted *if and only if there is at least one node $r \in \mathit{Dom}(g)$ such that for all nodes $v \in \mathit{Dom}(g)$, v is reachable from r. Consequently, every such r is called a* root *of the data graph.*

Rooted data graphs play an important rôle, because every program that traverses the graph will start at one single node and traverse the subgraph reachable from this node. Examples for such programs are the store services garbage collection, pickling and minimisation, which are introduced in the following chapters.

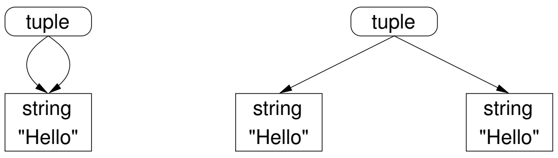


Figure 2.3: Two structurally equivalent data graphs

Such graphs with one explicit root node can be defined as a pair (g, r) of a data graph g and a distinguished root node $r \in Dom(g)$.

2.2 Structural and Token Equality

This section investigates how much of the data graph's structure a program can witness. As an example, consider these two SML expressions:

```
val x1 = let
           val t = "Hello"
         in
           (t,t)
         end

val x2 = let
           val t1 = "Hello"
           val t2 = "Hello"
         in
           (t1,t2)
         end
```

These two values can be represented by the data graphs in Figure 2.3. In SML, tuples and strings are defined with *structural equality*, meaning that two values have to be considered equal if their structures are isomorphic. This is the case for x1 and x2, although, of course, the structures of the underlying data graphs are not isomorphic.

The other type of equality is called token equality (or sometimes referential equality). It means that two values are only considered the same if they are in fact represented by the same node in the data graph. The next three SML expressions make clear why this is necessary:

```
val y1 = let
           val t = ref "Hello"
         in
           (t,t)
         end
```

```
val y2 = let
           val t1 = ref "Hello"
           val t2 = ref "Hello"
          in
           (t1, t2)
          end

val z = let
           val (t1, t2) = y2
         in
           t2 := "World";
           y2
         end
```

After evaluation of the first two expressions (y1 and y2), the data graphs describing their values look similar to those in Figure 2.3. If structural equality was allowed for references, too, a program accessing them in a read-only fashion could not distinguish between y1 and y2. After the evaluation of z, however, it is clear that y1 and y2 differ. So it is essential that y1 and y2 are not defined to be equal in the first place. Two references can only be equal if they are in fact the same reference, if they are represented by the same node in a data graph.

In order to annotate the equality type of a node directly on the data graph level, the set Lab can be partitioned:

$$\mathsf{Lab} = \mathsf{Lab}_{\mathsf{structural}} \uplus \mathsf{Lab}_{\mathsf{token}}$$

This yields data graphs with slightly more structure. The minimisation algorithm presented in Chapter 5 makes use of this additional structure.

Equality between nodes in a data graph can be described by *equality relations* $\sim \in$ *Dom*$(g) \times$ *Dom*(g) satisfying the following conditions:

1. If $v \sim v'$, $g(v) = (l, \_)$ and $l \in \mathsf{Lab}_{\mathsf{token}}$, then
   - $v = v'$
2. If $v \sim v'$, $g(v) = (l, s)$ and $s \in \mathsf{Str}$, then
   - $g(v') = g(v)$
3. 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
   - $\forall i \in \{0, \ldots, n\} : v_i \sim v'_i$

It is easy to see that the union of two equality relations is again an equality relation. It is also clear that the relation $\sim_=$, which is defined as $v \sim_= v' :\Leftrightarrow v = v'$, is an equality relation. These two observations justify the definition of $\sim_S$ as the (non-empty) union of all equality relations:

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

As equality relations are closed under union, $\sim_S$ is also an equality relation and hence the unique largest equality relation. It exactly expresses *semantic equality* of two nodes in a data graph, meaning that these two nodes are semantically indistinguishable by any client. It is necessary to consider the largest such relation, because any smaller relation does not yield the required equivalences for cyclic graphs.

The definition of semantic equality is similar to Scheidhauer's [34]. The difference is that each node explicitly carries its equality type, and that the relation $\sim_S$ is defined for both nodes with structural and with token equality type.

2.3 Levels of Structure

The same data can be described at different levels of structure:

In a data base, the full structure of all objects is known – their types and relations. The same is the case for data in a strictly typed programming language. At the other extreme is the memory of a computer at the hardware level. Here, only words of a certain width exist at addresses that are themselves words interpreted as integers. Pointers cannot be distinguished from scalar data.

The level of structure of the data graph as defined above lies in between. Only the connections established by edges between nodes can be seen, while nothing is known about the semantics, function, or type of a node.

This resembles the structure found in XML documents, which define the semantics of an individual node by a *tag*, while only the connection structure is known directly. XML documents are often said to be *semi-structured* [6], and this term seems to be appropriate for data graphs as well.

Different services may require different amounts of information about the graph. The services presented in this thesis require only little more information than provided by Definition 2.2. Minimisation, for example, needs to talk about semantic equality of nodes as introduced in the section above.

2.4 Related Work

There are other abstract models for the store of a programming system. Reynolds [31], for example, uses such a model to get an extended version of Hoare logic for low-level imperative programs.

On the other hand, the Java language specification [12] only specifies what the store (called “memory”) looks like on a high level. This allows to explain the effects of concurrency.

For Mozart, there is both a high-level and a low-level description. Scheidhauer [34] models the data graph of the virtual machine for Oz in a similar way as it is done in this thesis. The language Oz itself is defined in terms of a *constraint store* [37, 38], an abstract store model talking only about constraints, variables and records.

3 An Abstract Store

This chapter presents an interface to an abstract store. The abstract store is an abstract data type (ADT) that implements imperative graphs – nodes can be created, and edges can be redirected. Data graphs as introduced in the preceding chapter are the natural formal model for the objects that reside in the abstract store. Of course data graphs can only model the static aspects of abstract store graphs, they model the state of the store at one moment in time.

The programs that use the abstract store will be called *clients*. In the context of garbage collection, they are often called mutators, but in this thesis clients seems more appropriate because programs acting in a read-only fashion are interesting as well.

The abstract store developed in this thesis is supposed to hold *all* data structures that a programming system uses at run-time; it is not restricted to heap allocated data. That way, the abstract store gains full control over every cell of memory that a client program uses, so that store services like garbage collection can be implemented in a generic way.

3.1 Design Decisions

Implementing an abstract store requires a lot of design decisions concerning the internal representation of the high-level objects that the store provides.

This thesis cannot go into much detail, but for the store interface one has to consider the following points:

- Some data structures need a special representation. Numbers, for example, should be stored in the computer's native format to make computations reasonably fast.

- For each node, its arity and label have to be stored. Restrictions have to be placed on arity and number of different labels so that operations can be performed fast and still the space overhead is reasonable for nodes with small arity.

3.2 The Store Interface

The interface to the abstract store is described by means of an SML signature.

The store contains *nodes* that it stores at *addresses*. The nodes can be either integers, blocks or chunks. Each block or chunk node has a label, realised as an integer.

```
type addr
datatype node_t = INT | BLOCK | CHUNK
type label = int
type size = int
```

A block of size n corresponds to an n-ary data graph node, and a data graph node with a string stands for a chunk. Chunks are sequences of bytes, so this fixes the set Str to the strings over bytes.

The store as introduced below is a little less general than the definition of data graphs in that it knows of integer nodes. In the data graph, they will appear as nodes with the label int and a string with a platform-independent representation of the number. The abstract store can take advantage of a clever encoding of integers into addresses. That way, no store allocation is necessary for integers, and they can be used without the overhead of extracting them from a string.

The following functions allocate nodes:

```
val intToAddr : int -> addr
val allocBlock : label * size -> addr
val allocChunk : label * size -> addr
```

The type of a node can be inspected:

```
val typeOfNode : addr -> node_t
```

And of course it is possible to access the nodes' contents:

```
val addrToInt : addr -> int
val getBlockArg : addr * int -> addr (* Subscript *)
val getChunkArg : addr * int -> byte (* Subscript *)
```

As the abstract store provides an imperative data structure, blocks and chunks can be updated at a given index:

```
val setBlockArg : addr * int * addr -> unit (* Subscript *)
val setChunkArg : addr * int * byte -> unit (* Subscript *)
```

The *(* Subscript *)*-comment means that an error can occur if an element at an invalid index is accessed.

3.3 Language Layers

The store model given above is general enough to support data structures of arbitrary programming languages. The following examples show how some general-purpose data structures, as well as the run-time data structures of a typical abstract machine can be mapped to this model.

3.3.1 General Purpose Data Structures

Strings, arrays, and structured data types like `structs` in C, `records` in Pascal, or ML datatypes, are important data structures. All of them have straightforward implementations using the abstract store model.

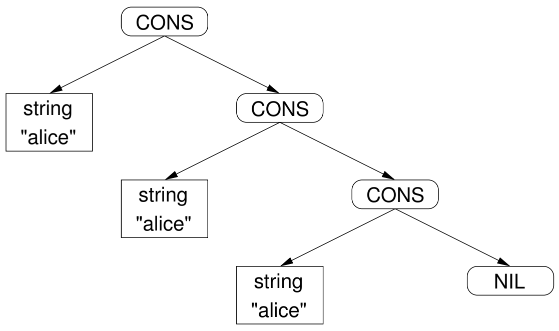
Strings

Strings are mapped one-to-one to a chunk with label string. Depending on the semantics of the programming language, they can be given structural or token equality type.

Arrays

An array of size n can be implemented as a block of size n. It is identified by a label array and given token equality type. A non-mutable, single assignment array (like vectors in the ML standard library) can of course implement structural equality.

There is another way of representing arrays of fixed-length scalar data: n items of size m each can be stored in a chunk of length $n \times m$.



CONS("alice",CONS("alice",CONS("alice", NIL)))

Figure 3.1: A data graph of a list

Structured Data

Every kind of structured data can be seen as an array of its contents. Constructor data types in ML for example can be mapped to blocks with the constructor's name as label and its arity as size. Lists, for instance, are realised as the data type

```
datatype 'a list =
     CONS of 'a * 'a list
   | NIL
```

Figure 3.1 illustrates what a list looks like as a data graph. The equality type of CONS nodes has to be chosen according to whether these lists are mutable or not.

3.3.2 Abstract Machine Data Structures

Most abstract machines (for example those discussed by Wilhelm [41] or Scheidhauer [34]) use a scheme of three different types of memory areas:

The Program Store

The program store is the container for the program's instructions. Its implementation depends on the representation of code: Byte code instructions can be stored in a scalar array, more complex instructions in an array or a list. Functional programming languages need a representation of first class procedures as heap objects, often called *closures*. A closure must contain a pointer to the code implementing the procedure, realised as a pointer to and an index into the program store.

As code can only be referenced through closures, it can become garbage when there is no more live closure referencing it. To enable automatic garbage collection of code, the program store can be split into several smaller parts, down to one program store per procedure definition.

The Stack

Stacks can be implemented as usual, based on arrays or lists. If direct access to stack cells other than the topmost one is desired, the array approach fits best. Often, several stacks are used, for example one for each thread of computation in concurrent languages, or a call stack and a trail for logic programming languages.

The Heap

Objects in the heap only require a suitable encoding into data graphs. In most abstract machines, heap objects are tagged, allowing for efficient case distinctions. Labelled store nodes are the natural implementation of tagged objects.

3.4 Garbage Collection

The abstract store interface as defined above conceals the fact that there is only a finite amount of memory available. Internally, the store must provide a *garbage collection* [19] mechanism that reclaims memory that is not used any longer.

3.4.1 Liveness

For garbage collection, it is essential to know which nodes in the store are still *alive*, meaning that there is still a client that has references to these nodes. In order to scan all the clients' references, these usually have to be put into a *root set*. The garbage collector then computes the set of live nodes as the set of nodes that are reachable from the root set.

As the abstract store is intended to hold *all* client data, it seems natural to represent the root set inside the store itself. This way, the store has one root node (the "entry point" of the root set) from which every live store node is reachable. The SEAM abstract store implements a root set in such a way.

3.4.2 Garbage Collection in the Data Graph Model

The live part of a store with a root node is described exactly by a rooted data graph (Definition 2.4). If a data graph g describes the state of an abstract store while the live part of the same store yields the rooted data graph g_{live}, the task of garbage collection is to “delete” all nodes in $Dom(g) \setminus Dom(g_{\text{live}})$. This means that after garbage collection, any data graph describing the state of the store is rooted.

3.4.3 Implementation of Garbage Collection

To implement garbage collection efficiently, a lot of knowledge about store internals is needed. That is why garbage collection is a service that cannot be implemented on top of the store but is at its very base.

Copying Garbage Collection

The most widely used implementations of garbage collection are so called *copying garbage collectors* as introduced by Minsky [25], often in their improved variant that needs only constant stack space, developed by Cheney [8]. Since these algorithms copy the live objects into a fresh memory region, the objects’ addresses change. Garbage collection in the abstract store can be regarded as a function that takes the root node of the store and returns the new root node. After a garbage collection, the returned root node is the only valid address. Hence, every client of the abstract store can only access the data in the store relative to the root node. It must not store addresses somewhere outside the store and use them at a later point in time, unless it can be sure that no garbage collection happened in the meantime.

Generational Garbage Collection

A more advanced garbage collection method divides the store into several *generations*, accounting for the fact that long-lived objects should be collected less often than short-lived ones [40, 21].

A drawback of generational garbage collection is that so called *inter-generational edges* have to be tracked, edges from nodes in older generations to nodes in younger generations. This can be realised by *write barriers*, imposing the need for a slightly modified store interface with an additional function:

```
val initBlockArg : addr * int * addr -> unit (* Subscript *)
```

`initBlockArg` may only be used if it is guaranteed that since allocation of the block no garbage collection happened. That way, `initBlockArg` never creates inter-generational edges. In all other cases, `setBlockArg` has to be used, hence creating an inter-generational edge.

4 Pickles

This chapter deals with the construction of a linear, external, platform-independent representation of a data graph, called *pickle*. A pickle is a string that contains enough information to reconstruct the original data graph, an operation known as *unpickling*. The pickle can be written to a file or passed to another process, allowing for persistent storage of the data graph as well as inter-process communication.

The algorithms in this chapter are described in terms of the data graph, and they can be implemented as a service that is based on the abstract store interface. This makes pickling and unpickling language- and platform-independent.

The term “linearisation” comes of course from the data graph perspective; internally, the abstract store has to represent data graphs in a linear way, so that linearising a data graph is in fact a conversion from one linear format into another.

In practice, only rooted subgraphs of a data graph will be pickled. This is due to the fact that the pickling algorithm is applied to one node of the data graph and returns a pickle of the subgraph reachable from that node. The data graphs in this chapter are always assumed to be rooted.

4.1 Related Work

Pickling is a quite common mechanism, implemented in a number of different programming languages and systems. The term “pickling” was coined in a paper about small data bases [4].

An early publication [16] presents a generic mechanism for encoding objects in CLU [22] (a language derived from Pascal) into messages and transferring them over the network. The Modula-3 language [27] inherited some of the concepts from CLU and also comes with a pickling mechanism.

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* [18]; there also is a compact introduction to the internals [32], revealing the roots of Java serialisation in CLU and Modula-3. Section 4.6.1 compares Java's flexible serialisation mechanism with the algorithms investigated in this chapter.

Microsoft's .NET Framework [24] offers a library module that provides a serialisation mechanism similar to Java's. In contrast to Java, the .NET specification only talks about the interface, not about the mechanism itself. The .NET system offers serialisation to different formats: an unspecified binary format, the "Simple Object Access Protocol" (SOAP, an XML format used for inter-process communication [10]), and plain XML.

Some other languages like Ruby [33] or Python [30] also implement methods of serialisation, but in a rather ad-hoc way: In Python, there is one mechanism that 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. Ruby specifies nothing at all about the mechanism or the pickle format. Both languages provide Java-like custom serialisation, which allows objects to provide their own string representation.

SML/NJ uses pickling to generate binary modules for separate compilation [3]. Pickling is done with the help of the garbage collector; this idea again comes from the Modula-3 world [27]. Section 4.6.2 discusses its advantages and disadvantages.

OCaml [28] offers a library module `Marshal` which allows for persistence and distribution of values. As for Python and Ruby, no internals are specified, except that it is not possible to pickle code (which is no surprise in a language that can be compiled to native code).

Mozart/Oz uses pickling for persistence and distribution of arbitrary data, including the so called functors, which implement the Oz module system. This is probably the most extensively used application of pickling. Alice and SEAM inherited many ideas from Mozart; Section 4.6.3 compares Mozart's pickling with the algorithms described in this chapter.

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 customisation.

4.2 Constructing Data Graphs

Pickles can be regarded as sequences of instructions for an interpreter that reconstructs the corresponding data graph in the abstract store. This section explores the basic functionality such an interpreter must provide, beginning with the reconstruction of trees and then generalising to acyclic and finally cyclic graphs.

4.2.1 Constructing Trees

Often trees are described as constructor terms, and a natural way of building them is the bottom-up approach: Beginning with the leaves, always build the children of a node and then the node itself, up to the root node.

The usual way of implementing bottom-up construction of terms is a stack-based interpreter. Its commands are pairs of a node's label and its arity: A command (l, n) results in popping n nodes off the stack and constructing a new node with label l and these nodes as its children. The new node is then placed on top of the stack. When the interpreter finishes, the stack contains only the root node of the constructed tree.

The data graph model provides nodes that contain strings as a special node type for scalar data. These nodes can be constructed just as easily with the above interpreter, they are built by instructions (l, s), where l is again the label and s is the string.

A pickle is a sequence of instructions for the interpreter, which is therefore called *pickle interpreter*. The language that it accepts is the *pickle language*.

Implementing a pickle interpreter on top of the abstract store interface is straight-forward: A (l, n) instruction creates the corresponding node via a call to `allocBlock`, then pops the addresses of the children from the stack and fills them in using `setBlockArg`. Finally, the address of the newly created node is placed on top of the stack. For (l, s) nodes, the `allocChunk` and `setChunkArg` store operations are used. The stack can of course be itself realised as a data structure in the abstract store.

Figure 4.1(a) shows a tree and the corresponding sequence of instructions, the pickle from which it can be built.

4.2.2 Constructing Acyclic Graphs

The bottom-up approach also works for directed, acyclic graphs (DAGs), but the interpreter has to be extended by a set of *registers*: Each node that has more than one predecessor – these nodes are called *shared nodes* – has to be placed in a register when it is first built, and loaded when it is needed again. Therefore the pickle language has to be extended by STORE i and LOAD i instructions, where i is the register where the node is stored into or loaded from.

Again, the implementation on top of the abstract store is easy: STORE i stores the address on top of the stack into register i (without popping the stack), and LOAD i fetches the address from register i and pushes it on the stack. The registers are realised as an array in the abstract store.

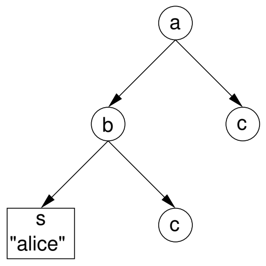
A DAG with its corresponding pickle can be found in Figure 4.1(b).

4.2.3 Constructing Arbitrary Graphs

The pickle interpreter up to this point cannot construct cyclic data structures, because it can only build a node when all its children are already constructed. Hence, cycles must be made explicit: A PROMISE instruction creates a temporary node that promises to once become one node on the cycle. That way, all the other nodes on the cycle can be built before the promise is fulfilled (by a corresponding FULFIL instruction).

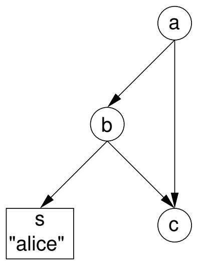
The pickle interpreter implements the PROMISE and FULFIL instructions by a technique called *back-patching*: A PROMISE for a node with label l and arity n constructs a node with that label and arity but arbitrary children (for example a 0 for every child). Fulfilling this promise then simply means replacing the "dummy" children by the real children. The interpreter needs both label and arity when promising a node, so the PROMISE instructions gets these as arguments. For fulfilling, it needs to find the previously created promise, so both instructions need a register as an argument.

Figure 4.1(c) shows an example of a cyclic graph, and the pickle containing the instructions to construct that graph.



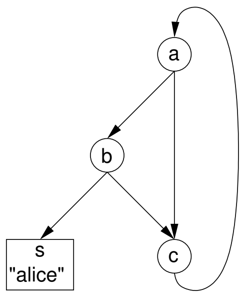
Instruction	*Arity*
s ”alice”	0
c	0
b	2
c	0
a	2

(a) A Tree



Instruction	*Arity*
s ”alice”	0
c	0
STORE 0	–
b	2
LOAD 0	–
a	2

(b) An acyclic graph



Instruction	*Arity*
s ”alice”	0
PROMISE 0 a	2
c	1
STORE 1	–
b	2
LOAD 1	–
FULFIL 0	2

(c) A cyclic graph

Figure 4.1: Data graphs and their pickles

4.3 Pickling Data Graphs

The remaining problem is how to construct, given a data graph, a corresponding pickle. The observation that leads to a solution is the following:

Linear data structures are sequences of nodes. In a data graph, however, nodes contain only part of the information; the *relations* between nodes are located in the *edges*. The first step towards a linearised data graph is to shift some information from edges to nodes.

This can be achieved by converting the graph into a tree. A tree has the property that every node has exactly one predecessor, except the root node, which has none. Tree edges therefore carry less structural information than graph edges.

4.3.1 Pickle Trees

A *pickle tree* of a rooted data graph (g, r) is supposed to contain all the structural information that the graph contains. Hence, it has to express sharing of nodes explicitly. A data graph node $v \in \mathit{Dom}(g)$ is called a *shared node* if it has more than one predecessor, or if it is the root r of the graph and has at least one predecessor.

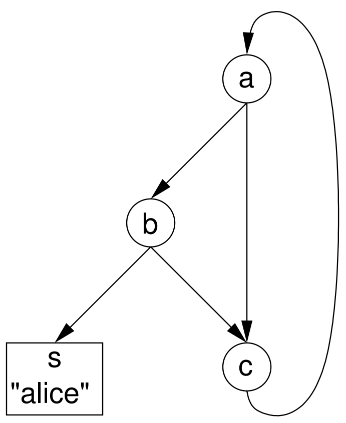
A pickle tree of g has as many edges as g, but possibly more nodes: For any shared node $v \in \mathit{Dom}(g)$ with $n+1$ predecessors, the pickle tree of g contains a node for v plus n leaf nodes. If $\{v_0, \ldots, v_n\}$ is the set of shared nodes, the node for v_i is labelled with $i : l$ (if l is its label) and called *indexed node*. The new leaf nodes carry the label $\rightarrow i$ and are called *reference nodes*.

The pickle trees of one data graph can differ in their structure, but they all have the same set of nodes (modulo renaming of the sharing indices).

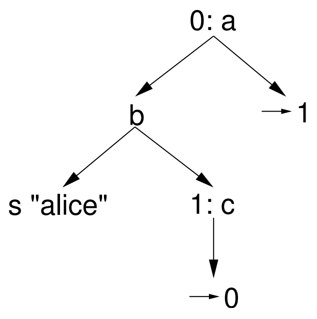
Figure 4.2(a) shows a data graph, Figure 4.2(b) one of its pickle trees. The data graph nodes labelled a and c are shared nodes, the pickle tree nodes with the labels 0:a and 1:c their respective indexed nodes, and the $\rightarrow 0$ and $\rightarrow 1$ nodes are reference nodes.

4.3.2 Shared Nodes

It is crucial for pickles to retain the sharing information expressed through indexed and reference nodes:



(a) A data graph



(b) A pickle tree

Node	*Arity*
s "alice"	0
→ 0	–
1:c	1
b	2
→ 1	–
0:a	2

(c) Postorder linearisation

Instruction	*Arity*
s "alice"	–
PROMISE 0 a	2
c	1
STORE 1	–
b	2
LOAD 1	–
FULFIL 0	2

(d) Resulting pickle

Figure 4.2: From data graph to pickle

- For nodes with token equality type, sharing is semantically visible (as discussed in Section 2.2).
- 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 result in an infinite loop.

4.3.3 Linearisation of Pickle Trees

The classical algorithms for linearising a tree are *preorder* and *postorder* traversal of the tree (as described by Sedgewick [36]). For both, the only structural information that has to be added in order to fully encode the tree is the number of successors of each node (called its *arity*). Representing edges implicitly in the order of the nodes makes the linear version of the tree compact.

Preorder and postorder tree traversal usually traverse a node's children from left to right (or from child 0 to child n), but the reverse order may be just as suitable for pickling.

Figure 4.2(c) shows a postorder linearisation of the pickle tree from Figure 4.2(b). This linearisation corresponds directly to a pickle that the pickle interpreter understands (as illustrated in Figure 4.2(d)). Only the reference and indexed nodes have to be adjusted:

Given an indexed node $i : l$ with arity n and a set of corresponding reference nodes $\{(\to i)_1, \dots, (\to i)_m\}$, the following conversions have to be done:

- If the linearisation places any of the reference nodes before its indexed node, the first such reference node is replaced by
  PROMISE $i\ l\ n$
  and the indexed node by
  FULFIL $i\ n$.
  Every other reference node becomes a
  LOAD i.
- Otherwise, the indexed node becomes
  $l\ n$
  STORE i
  and the reference nodes become
  LOAD i
  instructions.

4.3.4 Preorder Linearisation and Top-down Construction

The previous section showed that a postorder traversal of a pickle tree yields a pickle that can be used to reconstruct the data graph in a bottom-up way. A preorder traversal implies just the dual approach, a top-down reconstruction.

A pickle interpreter for a preorder or top-down pickle language has to construct a node before its children are constructed; this looks similar to the PROMISE/FULFIL mechanism for bottom-up pickles, and the same back-patching technique can be applied.

For each data graph, one can find a pickle tree that has a preorder linearisation in which no reference node comes before its corresponding indexed node. This means that a top-down unpickler only needs stack and registers, but no PROMISE and FULFIL instructions.

4.3.5 Depth First Search

Depth first search (DFS) is a classical algorithm for graph traversal. R. Tarjan invented it in 1972 [39], and it is discussed in detail in most textbooks on algorithms (for example by Sedgewick [36]).

When applied to trees, DFS does just a preorder or postorder traversal. In a graph, it detects sharing and cycles, building the DFS tree on its way. A DFS tree of a data graph is very similar to a pickle tree. Using Sedgewick's terminology, DFS classifies a graph's edges into *tree* edges, *down* edges, *cross* edges and *back* edges.

DFS can build a pickle tree by creating ordinary tree nodes for each graph node with only one predecessor, indexed nodes for each shared node, and reference nodes for each down, cross and back edge.

Building the pickle tree and linearising it can thus both be performed by DFS. This makes DFS the natural choice for a pickling algorithm.

4.4 Resources

Resources are objects in an abstract store for which no meaningful representation exists outside this store. The classical example is operating system file descriptors. They could be stored in a pickle, as they are merely integer numbers, but they make no sense at all to any other process.

Generally, every store node that in some way refers to something outside the abstract store but local to the current process is a resource. This includes pointers to native code outside the abstract store – for example library functions implemented in a language like C.

The pickling algorithm should be able to recognise resources and reject to pickle any graph containing resources. As for the equality type of a node, resources can be marked by partitioning the set of labels:

$$\mathsf{Lab} = \mathsf{Lab}_{\mathsf{resource}} \uplus \mathsf{Lab}_{\mathsf{non-resource}}$$

Section 6.4 presents a mechanism that can automatically convert resources into non-resource descriptions upon pickling. The unpickler can then reconstruct a new resource from this description.

4.5 Implementation Details

The pickler and unpickler implemented for SEAM are basically straightforward implementations of the algorithms sketched above: The pickler uses a variant of DFS to traverse the data graph, and the unpickler interprets the pickle as a stack-based programming language.

In the following, some optimisations and facts to consider are discussed which are deemed essential for an efficient implementation.

4.5.1 Marking Nodes

DFS has to mark nodes as visited in order to detect sharing and cycles. There are two ways of marking a node in the abstract store: Either the store reserves a special mark bit in the node itself, or it provides hash tables that can have store nodes as keys (an indirect way of marking a node).

Mark bits have three major disadvantages: They require space (which is usually quite expensive, because there already has to be quite a lot of information stored in the node header), and they have to be cleared after the algorithm has finished. But the biggest problem is that mark bits cannot be used in a concurrent setting: If two pickler processes are working on the same nodes, marking is ambiguous and therefore useless.

Hash tables with nodes as keys require active support from the store: The only information that can be used as a hash key for a node is its address, which may change through garbage collection. The garbage collector must be aware of node hash tables and adjust them appropriately after garbage collection has finished.

Hashing has another advantage: Arbitrary data can be associated with a node. The pickling algorithm can use this to store bookkeeping information (like the DFS preorder and postorder numbers) with each node.

4.5.2 Pre-Computing the Maximum Stack Height

The maximum height of the stack that the unpickler needs for interpreting a certain pickle can already be computed at pickling time. The function that computes the stack height of a node v in a pickle tree looks like this:

$$height(v) = \begin{cases} 1 \\ \qquad \text{if } v \text{ is a leaf node} \\ max\{height(v_0) + 0, height(v_1) + 1, \ldots, height(v_n) + n\} \\ \qquad \text{if } v \text{ has children } \langle v_0, \ldots, v_n \rangle \end{cases}$$

This function assumes that the pickle-tree is linearised in a left-to-right way.

If the maximum stack height is stored at the beginning of the pickle, the unpickler can allocate the stack once for the whole unpickling operation. Otherwise, it would need to use a dynamic stack implementation.

4.5.3 The SEAM Pickle Language

The pickle language used in the concrete implementation on SEAM is shown in Figure 4.3 in BNF form. `<uint>` stands for an unsigned integer. The CHUNK and BLOCK instructions create the corresponding store nodes, and STORE, LOAD, PROMISE and FULFIL work exactly as described above. The INIT instruction specifies how many registers the pickle uses, and how much stack the pickler needs. The unpickler is realised as an interpreter of this stack-based language.

An example of a SEAM pickle can be found in Figure 4.4. It represents the graph from Figure 4.2(a) (for readability the labels have not been coded into integers).

pickle	::=	init instrs ENDOFSTREAM
init	::=	INIT stackSize noOfRegisters
stackSize	::=	<uint>
noOfRegisters	::=	<uint>
instrs	::=	instr instrs
	|	ϵ
instr	::=	simpleInstr
	|	complexInstr
simpleInstr	::=	STORE register
	|	LOAD register
	|	CHUNK label size <byte>*size
complexInstr	::=	PROMISE register complexInstr’
	|	FULFIL register size
	|	complexInstr’
complexInstr’	::=	BLOCK label arity
register	::=	<uint>
size	::=	<uint>
arity	::=	<uint>
label	::=	<uint>

Figure 4.3: The SEAM pickle language

```
INIT 2 2
CHUNK s "alice"
PROMISE 0 BLOCK a 2
BLOCK c 1
STORE 1
BLOCK b 2
LOAD 1
FULFIL 0 2
ENDOFSTREAM
```

Figure 4.4: Example of a SEAM pickle

4.5.4 Exploring Depth-First Rightmost

In functional programming languages like ML, run time data structures tend to be either balanced or recursive to the right. Lists in ML, for instance, are represented as "degenerate trees".

A leftmost exploration strategy is not optimal for this kind of data: On unpickling, a stack is needed that is at least as large as the length of – for example – the list. For the ML implementation Alice it proved useful to pickle depth-first rightmost, decreasing the average stack height by one order of magnitude. This is especially due to the fact that the representation of code is right recursive.

4.5.5 Two-Pass Pickling

The pickler as implemented for SEAM builds the pickle tree, linearises it, and computes the stack height in a first pass, using one depth-first traversal. This traversal fills a buffer containing the resulting pickle *except for the* STORE *instructions.* They have to be placed after shared nodes, but the sharing can only be detected later during DFS. STORE instructions therefore have to be remembered during DFS and inserted afterwards, in a second pass.

4.5.6 Some Numbers

The pickler/unpickler implementation is based on the previous version by Brunklaus and Kornstaedt. This older version used a preorder (and thus top-down) pickle format without explicit LOAD/STORE operations. It therefore needed dynamic data structures during unpickling and a register for every node, which made it less efficient. The pickle format, however, was a bit more compact than the one suggested in this thesis and implemented in the new (un)pickler.

Both pickler and unpickler are written in C++, making up around 1500 and 1200 lines of code, respectively. The next table shows some benchmarks comparing old and new implementation:

Task	*No. of pickles*	*Old (un)pickler*	*New (un)pickler*	*gain*
Alice bootstrap	106994	32m19s	29m30s	9%
Bootstrap, only pickling	572	24.5s	25.0s	−0.2%
Bootstrap, only unpickling	106422	320.5s	263.5s	18%
Invoking Alice toplevel	77	3.45s	2.73s	21%

The interactive Alice toplevel depends on 77 components (and thus pickles) to be loaded on startup, so the improvement here is only due to the faster unpickler. The bootstrap process is faster by 2 minutes and 49 seconds, but the benchmarks counting only pickling and unpickling can explain only 57 seconds. The remaining acceleration is reached because the new unpickler use less memory, resulting in less garbage collections.

All benchmark times were measured on an Intel Pentium III with 1400 MHz and 1 Gigabyte of RAM.

4.6 Comparison with other Pickling Mechanisms

This section compares the pickler presented in this chapter with the Java, SML/NJ and Mozart picklers. Java was chosen because it is the most widely used system that has a well-defined pickling mechanism. SML/NJ employs an interesting approach in that it reuses the garbage collector for pickling. The comparison with the Mozart pickler is interesting because Alice and SEAM inherited a lot of ideas from their predecessor Mozart.

4.6.1 Java Serialization

The Java pickling mechanism [32, 18] combines a lot of high-level and low-level aspects. It uses Java objects as its central metaphor and is always applied to the object graph that is reachable from some root object.

Low-Level

On the low-level side, the exact byte stream format is specified. A pickle contains the objects' representations in depth-first preorder. Each object is assigned a unique *handle* that makes it possible to reference the object later on in the stream for sharing preservation and creation of cycles (similar to indexed and reference nodes).

The low-level realisation is not too different from the one given in this thesis. A suitable coding of Java objects into abstract store data structures would directly produce an alternative way of serialising Java data structures.

One advantage of Java's commitment to objects as the low-level metaphor is however a certain degree of inherent type-safety. Every object that can be reconstructed from a serialised stream obeys the type constraints of its class. The SEAM unpickler can only check that the reconstructed graph is in fact a data graph; it is impossible to check that this data graph satisfies the type constraints imposed by a language layer.

High-Level

The high-level serialisation interface offers a lot of flexibility:

- Objects can implement their own serialisation method, giving them the opportunity to define their external format.
- There are techniques for resolving versioning conflicts, making it possible to load serialised object streams by a version of a class different from the one they were written by.
- Certain object fields can be marked “transient”. These fields are not serialised, so this is a mechanism for hiding parts of the object.

The last point can be modelled easily with the pickling algorithm presented in this thesis: The object graph would have to be traversed and copied, doing all the necessary transformations before the transformed copy is pickled. The Java approach however has the advantage that no copy has to be created and no additional graph traversal is necessary: Transformation and serialisation are intertwined. Chapter 6.4 introduces a mechanism that allows for the transformation of data graphs during pickling and unpickling.

In Java, code cannot be pickled. The only way to make code persistent is to compile it to a class file. This of course means that the objects in a pickle are separated from the definition of their classes and thus from the implementation of the methods that are applied to them.

4.6.2 Reusing the Garbage Collector for Pickling

The idea to reuse the garbage collection infrastructure for pickling was first formulated in the context of Modula-3 [27]. It is based on the observation that the garbage collector does a job quite similar to pickling: It has to traverse the data graph.

One of the most well-known garbage collection algorithms is Cheney’s copying garbage collector [8]. Appel and MacQueen [3] give a hint how it can be used for pickling: Make a copying garbage collection of the subgraph you want to pickle into a fresh area of memory (usually called semi-space in this context), applying some sort of address translation. Then simply write the contents of this semi-space to a file. Unpickling involves another address translation, it becomes a process like linking, but for data instead of code.

The problem is that the pickling format has to be exactly the internal format of objects in the store. This internal format is most probably platform-dependent, so that the pickle format cannot be used for platform-independent communication.

If no other transformation than an address translation is done, the pickle contains all edges between nodes explicitly (as in the store). The pickle does not take advantage of the fact that the nodes are ordered and will thus yield a less compact format than the one described in this chapter.

4.6.3 Mozart Pickles

Mozart [9] provides a generic traversal engine for the data graph, which employs a depth-first strategy. There are 22 different node types in the Mozart store, and a client of the graph traversal engine has to provide one method for each type of node.

The pickler acts as such a graph traversal client, and it traverses the graph twice. On the first traversal, the pickler gathers information about resources and sharing of nodes. On the second traversal, it generates the pickle.

The Mozart pickler uses both left-to-right and right-to-left exploration, depending on the type of the node, for efficiency reasons. The serialisation of the nodes happens in DFS preorder, implying a top-down construction on unpickling.

As Mozart is a language with advanced concurrency features, pickling can be done in a concurrent setting. Chapter 6 shows how this fits into the abstract store model given in this thesis.

Procedures are first-class citizens in Mozart, and it is possible to pickle values containing procedures (and thus containing code). This is a unique feature of Mozart, and Mozart's module system makes heavy use of pickling. Code is, however, not represented as ordinary store data structures, so that it requires special treatment from the pickler on a low level.

Alice inherited the idea of a pickle-based module system. For Alice/SEAM, code is nothing but another data structure, so no special treatment as in Mozart is necessary.

5 Minimisation

Minimisation in the context of data graphs means to transform one graph into an equivalent one with a minimal number of nodes. This is defined formally in this chapter, and an algorithm is given that can do this transformation efficiently.

One straightforward application of the minimisation mechanism is to minimise a data graph before it is pickled. This has proved to have a positive impact on both pickle size and the time needed for unpickling. This chapter presents this and other areas of application for a generic minimisation mechanism.

5.1 Definition of Minimal Graphs

Minimality of a data graph means, in short, that token equality and semantic equality of nodes coincide:

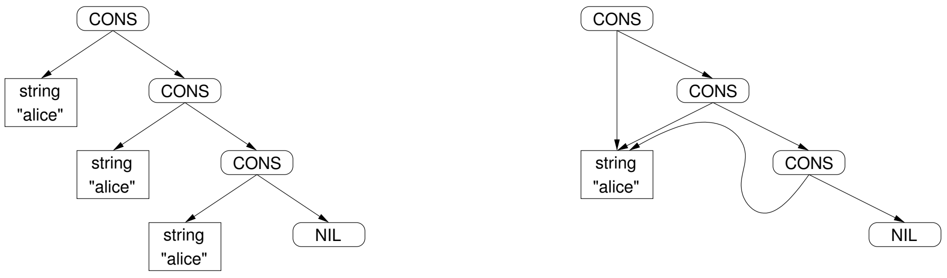
Definition 5.1 (Minimal data graphs)
A data graph is called minimal *if and only if for any two nodes v and v' the following holds:*

$$v \sim_S v' \Longrightarrow v = v'$$

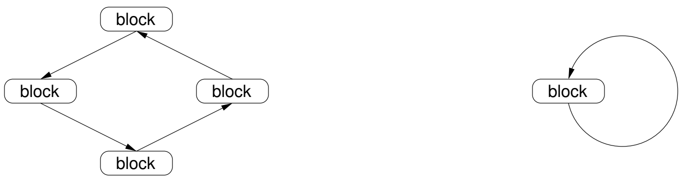
Each data graph g is equivalent to the graph $g_{/\sim_S}$, the minimal data graph that has as nodes the equivalence classes of the nodes in g.

Figure 5.1 shows some data graphs and their minimal equivalents. Nodes with labels that are prefixed with "t:" have token equality type.

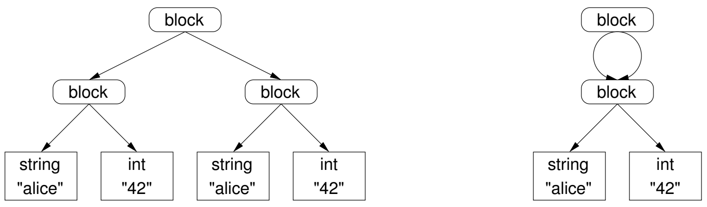
The goal of the minimisation algorithm is to compute $\sim_S$ for a graph g and transform g into $g_{/\sim_S}$ by collapsing equivalent nodes into one representative per equivalence class.



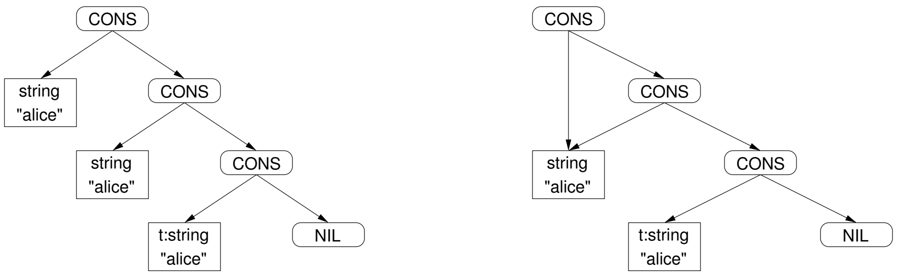
(a) Sharing of equivalent leaves



(b) Shrinking cycles



(c) Sharing of equivalent subgraphs



(d) Sharing in the presence of token equality nodes

Figure 5.1: Some data graphs on the left and their minimal equivalents on the right

5.2 The Minimisation Algorithm

Minimisation was first investigated in the context of finite automata. Algorithms for minimising the number of states in a finite automaton can be found in textbooks from the sixties (for example Harrison [15]), and in 1971 Hopcroft gave an algorithm with $n \log n$ worst case time complexity [17]. Cardon and Crochemore [7] generalised Hopcroft's algorithm to the minimisation of arbitrary graphs. This section presents an application of Cardon's and Crochemore's results to data graphs.

5.2.1 Partitioning a Graph

Cardon and Crochemore describe an algorithm that can partition a graph, meaning that it computes an equivalence relation on the nodes of the graph. Formally, they describe their algorithm in terms of regular congruences and refinement of equivalence relations. These notions can be analogously defined for data graphs:

A *congruence* $C \in \mathit{Dom}(g) \times \mathit{Dom}(g)$ on a data graph g is an equivalence relation that commutes with the successor relation of the graph:

$$\begin{aligned}&\forall u, v, w \in \mathit{Dom}(g)\ \forall i \in \mathbb{N}:\\ &\qquad \Big[(u,v) \in C \wedge v \rightarrow_{i,g} w \Rightarrow \exists w' \in \mathit{Dom}(g) : u \rightarrow_{i,g} w' \wedge (w', w) \in C\Big]\end{aligned}$$

A congruence C is called *regular* if and only if for every equivalence class $\mathcal{D}$ of C, the following holds:

$$\forall u, v \in \mathit{Dom}(g)\ \forall i \in \mathbb{N} : (u,v) \in C \Rightarrow |\mathsf{succ}_i(u) \cap \mathcal{D}| = |\mathsf{succ}_i(v) \cap \mathcal{D}|$$

An equivalence relation $\mathcal{G}$ is a *refinement* of an equivalence relation $\mathcal{F}$ if and only if

$$\forall u, v \in \mathit{Dom}(g) : (u,v) \in \mathcal{G} \Rightarrow (u,v) \in \mathcal{F}$$

If $\mathcal{G}$ is a refinement of $\mathcal{F}$, $\mathcal{F}$ is also called *coarser* than $\mathcal{G}$.

Cardon's and Crochemore's algorithm takes an initial equivalence relation $\mathcal{F}$ and refines it. Each step of refinement produces a new equivalence relation Π_i. When the algorithm finishes after some N steps, Π_N is the coarsest regular congruence that is a refinement of $\mathcal{F}$.

5.2.2 Computing $\sim_S$

The graph minimisation algorithm can be used to compute $\sim_S$ on a data graph, because there is an equivalence relation $\mathcal{F}$ such that $\sim_S$ is the coarsest regular congruence that is a refinement of $\mathcal{F}$.

The following arguments prove this:

Proposition: $\sim_S$ is a regular congruence.

For two nodes $u \sim_S v$, their successors have to be pairwise $\sim_S$-equivalent. This makes $\sim_S$ a regular congruence.

Proposition: $\sim_S$ is a refinement of some $\mathcal{F}$.

Let $\mathcal{F} \in Dom(g) \times Dom(g)$ be defined as follows:

For any two nodes $v, v' \in Dom(g)$, $(v, v') \in \mathcal{F}$ if and only if

- $v = v'$ or
- $g(v) = (l, \langle v_0, \ldots, v_n \rangle)$ and
  $g(v') = (l, \langle v'_0, \ldots, v'_n \rangle)$ and
  $l \in \mathsf{Lab}_{\mathsf{structural}}$
  or
- $g(v) = (l, s)$ and
  $g(v') = (l, s)$ and
  $s \in \mathsf{Str}$ and
  $l \in \mathsf{Lab}_{\mathsf{structural}}$

It follows from the definition of equality relations that every equality relation, and hence $\sim_S$, is a refinement of $\mathcal{F}$.

Proposition: Π_N, which refines $\mathcal{F}$, is an equality relation.

The proof of this proposition follows the definition of equality relations:

1. If $(v, v') \in \Pi_N$, $g(v) = (l, \_)$ and $l \in \mathsf{Lab}_{\mathsf{token}}$, then
   - $v = v'$ (Π_N refines $\mathcal{F}$)
2. If $(v, v') \in \Pi_N$, $g(v) = (l, s)$ and $s \in \mathsf{Str}$, then
   - $g(v') = g(v)$ (Π_N refines $\mathcal{F}$)

3. If $(v, v') \in \Pi_N$ and $g(v) = (l, \langle v_0, \dots, v_n \rangle)$, then
   - $g(v') = (l, \langle v'_0, \dots, v'_n \rangle)$ (Π_N refines $\mathcal{F}$)
   - $\forall i \in \{0, \dots, n\} : (v_i, v'_i) \in \Pi_N$ (Π_N is a regular congruence)

Corollary: $\sim_S$ is the coarsest regular congruence that is a refinement of $\mathcal{F}$.

As Π_N is an equality relation, it follows from the definition of $\sim_S$ that $\Pi_N \subseteq \sim_S$. As $\sim_S$ is a regular congruence and a refinement of $\mathcal{F}$, it follows from the fact that Π_N is the coarsest such congruence, that $\sim_S \subseteq \Pi_N$. The conclusion is that $\sim_S = \Pi_N$.

5.3 Areas of Application

Especially in functional languages, where a lot of data structures can be implemented so that they support structural equality, the potential for minimisation seems promising. This section investigates general approaches of applying the minimisation algorithm.

5.3.1 A High-Level Interface to the Minimiser

The minimisation functionality can be made available at language level. In Alice, for example, it can be accessed through a structure with the following signature:

```
signature Minimiser =
   sig
     val minimise : 'a -> unit
   end
```

The function `minimise` takes an arbitrary value and minimises the subgraph that is reachable from the node that represents this value. This interface allows user programs to make use of the minimiser in a type safe way.

5.3.2 Minimised Pickles

One application that quickly comes to mind is to minimise a data graph before it is pickled. Pickling seems to be the right time to do minimisation because of three reasons:

	Normal size	*Minimal size*	*% gain*
Alice run-time	9316	6608	29%
Code pickle	3359	2009	40%
Signature pickle	78	63	19%

Table 5.1: Pickle sizes with and without minimisation

1. Most pickles are written once and loaded often so that the cost of minimising can be compensated.
2. The pickle becomes more compact, resulting in a faster unpickling.
3. Pickling takes some time anyway, so if one is willing to spend time on pickling, minimisation can be done with reasonable overhead.

Alice makes heavy use of pickling, because its entire component system is based on it. Alice components are pickles that contain – besides other things – both the values (including functions and thus code) that make up the component and the type (the signature) of the component.

The part of the Alice system that is written in Alice itself has been compiled with and without minimisation to investigate how much a real world application can profit from pickle minimisation. Table 5.1 shows the results, as well as the sizes of one test pickle containing mostly code, and one containing mostly signature information. The code pickle contains a linked version of the Alice compiler, with all signature information stripped. The signature pickle imports a large part of the Alice library (and hence has big import and export signatures) but contains only little code. All sizes are given in kilobytes.

5.3.3 Minimisation as a Garbage Collection Stage

The redundant information deleted by minimisation can be regarded as a form of “structural garbage”. In a generational garbage collection framework, minimisation could be used as an additional stage, applied before a “major collection” (a collection of the oldest generation).

5.4 Incremental Minimisation

Mauborgne [23] describes a modification of Hopcroft’s algorithm allowing for *incremental* minimisation of a graph – in this context, incremental means that a larger graph is kept minimal and it is possible to add new graphs to it without minimising the whole resulting graph again.

Mauborgne's paper deals with sets of regular trees represented as graphs. The algorithm he presents can incrementally add new regular trees to a given set of minimal regular trees in time $O(n \log n)$, where n is the size of the *added* regular tree.

Woop and Horbach [42] give an improved description of the algorithm including proofs of correctness and asymptotic complexity.

The drawback of incremental minimisation is that a lot of additional bookkeeping information is needed: The graph has to be analysed for its strongly connected components (SCCs); a hash table is needed that maps each node to its SCC. This information has to be kept between invocations of the minimiser, so there probably is a large memory overhead. It has to be investigated whether there are applications where the incremental algorithm can outperform the non-incremental one.

5.5 Implementation Details

A naïve implementation of the algorithm, especially of its bookkeeping data structures, would result in poor performance. This section discusses some key issues that are crucial for efficiency.

A modification of Hopcroft's algorithm (as described and implemented by Horbach and Woop [42]) serves as the base for the SEAM implementation. This algorithm computes the same congruence as Cardon's and Crochemore's (Horbach and Woop proved this) and can be realised with efficient data structures as discussed below.

5.5.1 Partitions

The main data structure that the minimiser uses has to represent equivalence classes and provide an operation that splits a class into two. This scheme is known as *partition refinement*, and Habib et al. [13] develop a generic approach to it. Their implementation uses an array and a list of pointers into the array as the partition data structure. Figure 5.2 illustrates the idea.

With this data structure, the operation that removes n arbitrary members of a class and puts them into a new class has an asymptotic complexity of $O(n)$. Additionally, the initial partition can be computed by sorting the array with respect to a suitable order.

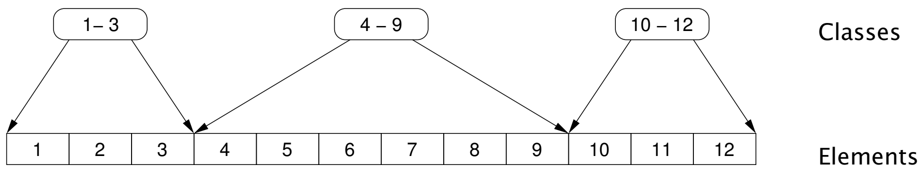


Figure 5.2: Implementation of partitions

This technique can be directly applied to Hopcroft's minimisation algorithm, whereas Cardon's and Crochemore's approach needs different (heavily list-based) data structures. Hopcroft's algorithm can be easily adjusted to deal with data graphs; the SEAM minimiser is a modified version of Hopcroft's algorithm combined with Habib's ideas.

5.5.2 Filling the Data Structures

As its first step, the algorithm has to create the partition data structure and fill it with the nodes of the subgraph that will be minimised. In addition, a table has to be filled that contains an inverse version of the edges: For each node, the list of its predecessors is stored, together with some more bookkeeping information. This involves a traversal of the graph, similar to the pickling algorithm; again, a depth-first traversal is the natural choice.

5.5.3 Collapsing of Nodes

When $\sim_S$ has been computed, all the nodes in one equivalence class have to be replaced by one representative of that class: The edges of all direct predecessors of a node have to be redirected to the chosen representative.

This can easily be done, because Hopcroft's algorithm needs a backwards version of every edge anyway, making it possible to access a node's predecessors.

An alternative way of replacing nodes will be presented in the next chapter.

5.5.4 Implementation on SEAM

The basis for the implementation of the minimisation algorithm was the SML version by Horbach and Woop [42], combined with Habib's partition refinement techniques.

The original algorithm was tuned for clarity and correctness of the worst case runtime analysis. For the actual implementation on SEAM, some compromises were made:

- As the average in-degree of nodes is small, a list of incoming edges is used instead of an array of lists that is indexed by the incoming edges' numbers.
- The initial partition is created using a quicksort algorithm (see Section 5.5.1). This has complexity $O(n \log n)$ instead of $O(n)$ for the original implementation that used hashing. In practice, this does not change the overall complexity but instead improves the performance.
- The order used to create the initial partition is extended such that two block nodes are only in the same class if all their successors containing integers are equal. This makes the comparison during quicksort a bit more complex (scanning all the successors of a node), but that way no integer nodes have to be processed afterwards.

The minimiser comprises around 1700 lines of C++ code. In addition, some parts of the Alice library have to be changed to make use of it.

Section 5.3.2 already gave some numbers indicating that pickles can be shrinked considerably using minimisation. The following table shows the time needed to bootstrap the Alice system with and without minimisation. The figures suggest that the performance overhead is rather small. The minimised pickles even reduce the start-up time of the interactive toplevel.

Task	*No. of pickles*	*w/o minimisation*	*w/ minimisation*	*gain*
Alice bootstrap	106944	29m24s	30m34s	−4%
Invoking Alice toplevel	77	2.73s	2.27s	17%

6 Transients

In this chapter, an extension to the data graph and the abstract store is investigated: Transients are nodes that can become other nodes, a powerful concept with a lot of applications.

A similar mechanism can be found in several programming systems: Prolog and Mozart use transient-like objects to implement logic variables. They can be created without being bound to a value, and later, when they are bound, disappear. In Prolog, such a binding can be undone for backtracking.

In lazy functional programming languages, the evaluation of expressions is suspended until their value is needed. This is achieved by creating suspensions (often also called closures) containing the code that evaluates the expression, and after evaluation replacing the suspension with the resulting value. Peyton Jones [20] describes this in the context of Haskell.

Smalltalk provides a primitive *become* operation that exchanges the identity of two objects [11]. It is, for example, used to implement dynamically growing collections of objects: When the collection gets too small, a new collection is allocated, the objects from the old collection are put into the new one, and the identities of old and new collection are swapped. That way, every subsequent access to the collection will be performed on the new one. The next garbage collection removes the old collection.

6.1 The "become" Operation

The store has to provide only one additional operation in order to support transients: `become(v1,v2)` substitutes the node `v2` for the node `v1`.

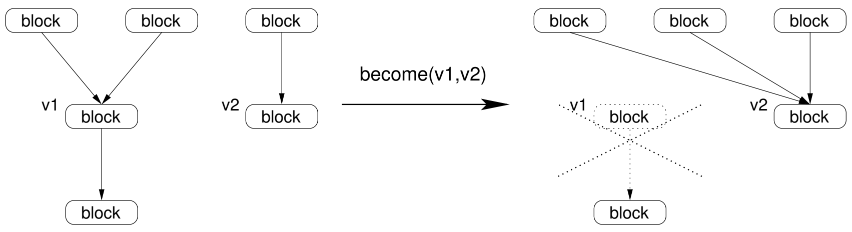


Figure 6.1: The *become* operation

6.1.1 Intended Semantics

Figure 6.1 shows two data graphs, the left one before and the right one after an application of the `become(v1,v2)` operation: All the incoming edges of `v1` are redirected to point to `v2` instead.

The node `v1` is not connected to the data graph any longer, as *all* its incoming edges have been redirected. The next garbage collection will thus delete the node. Note, however, that its *successors* could stay alive, as they may be referenced by other nodes.

The *become* operation is the only way of explicitly deleting a node in the store, and it deletes nodes in a safe way: No dangling pointers are created, because all pointers to the deleted node are redirected to another node. These nodes that are replaced by other nodes are called *transients*.

6.1.2 Implementation Using Redirection Nodes

The operation `become(v1,v2)` cannot be implemented efficiently using the normal abstract store interface, as this would imply a traversal of the whole data graph in order to find all predecessors of `v1`.

An efficient solution for this problem is to introduce a special internal node type called a *redirection node*. Then `become(v1,v2)` transforms `v1` into a redirection node with an edge to `v2` (shown in Figure 6.2). The store operations transparently follow chains of redirection nodes, so that client programs do not witness these internal structures.

Garbage collection eliminates redirection chains. That way, scanning for `v1`'s predecessors is postponed to a time when a scan of the data graph has to be done anyway.

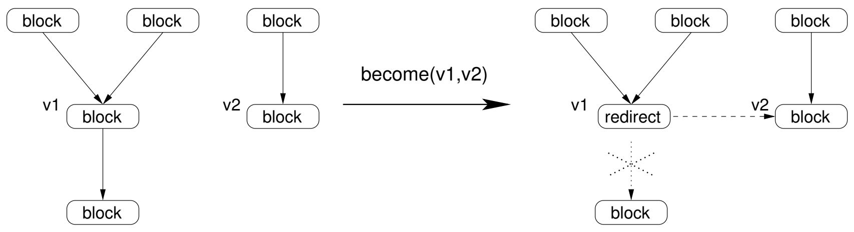


Figure 6.2: Redirection nodes

Redirection nodes are a well-known technique already used in the context of the Warren Abstract Machine [1] (where they are called “REF cells”).

How Redirection Nodes can be Realised

Basically, a redirection node is an ordinary store node with a special label and one child – the target node. Redirecting a node `v1` to a node `v2` means changing `v1`’s label to the redirection label and setting its first child to `v2`. This reveals a drawback: Only nodes with outgoing edges can be turned into redirection nodes.

This limitation can be worked around by not allowing 0-ary nodes in the first place – a node without children can always be represented by an integer node. Redirection of integer nodes is not necessary, because they are not allocated but represented directly as an address (as discussed in Section 3.2), and thus are always shared anyway.

6.1.3 Optimisations

The implementation suggested above implies that every store operation must test for redirection nodes. This can become a performance issue, because redirection nodes will probably be rare.

As an optimisation, the store interface is extended by the following functions:

```
val directTypeOfNode : addr -> node_t

val directAddrToInt : addr -> int
val directGetBlockArg : addr * int -> addr (* Subscript *)
val directGetChunkArg : addr * int -> byte (* Subscript *)

val directSetBlockArg : addr * int * addr -> unit (* Subscript *)
val directSetChunkArg : addr * int * byte -> unit (* Subscript *)
```

The “direct” functions behave exactly like their counterparts but assume that the addresses do not contain redirection nodes. They can thus omit the test, which makes them more efficient but less safe.

6.2 Concurrency: Futures and Promises

This section discusses futures and promises, which are the main applications of transients in Alice/SEAM.

6.2.1 Futures and Promises

Futures were first introduced in Multilisp (as described by Halstead [14]). They can be used to model synchronisation in a concurrent programming language and provide a means of introducing lazy evaluation into otherwise strict functional languages. These concepts were discussed formally for Alice by Schwinghammer [35].

In the abstract store, futures are transient nodes. Binding a future `f` to the node representing its value `v` simply means invoking `become(f,v)`. The future disappears, and subsequent operations accessing the former future will get the value instead.

In Multilisp, bound futures are called *determined*. They are also realised as chains of redirection nodes, but there is no abstraction layer that dereferences those chains automatically. The garbage collector shortens chains of determined futures, just as described above.

6.2.2 Synchronisation of Concurrent Threads

Before the future gets bound, it contains a list of threads that are waiting for the future, and – in case of a *byneed future* – a pointer to a procedure that can compute the future’s value.

Triggering the evaluation of a byneed future and waking the waiting threads when a future gets bound are complex operations involving the thread scheduler. Brunklaus and Kornstaedt [5] describe this in detail.

6.2.3 Concurrent Pickling and Minimisation

Concurrency in combination with destructive update always requires special care: Programs have to be implemented in a *thread-safe* way, meaning that no race conditions occur when two threads access the same objects.

This is especially true for pickler and minimiser: Their results are not predictable if the subgraph they are working on is changed during pickling or minimisation. The only way to make both operations thread-safe is to make them atomic, not allowing any other thread to become runnable before they are finished.

This implies a simple strategy for dealing with futures: As they refer to a suspended computation, they are resources and as such cannot be pickled. The minimiser has to treat them as ordinary nodes with token equality type but may not wait for them, as that would make minimisation a non-atomic operation again.

In order to ensure that some data graph does not contain futures, an operation

```
val awaitAll : addr -> unit
```

can be implemented that triggers evaluation of all byneed futures in the subgraph reachable from the given address and blocks on all futures. When a future gets determined, the whole process has to be started again, because the thread that computed the value of the future can have modified parts of the data graph and that way introduced new futures.

The `awaitAll` operation can be called before pickling or minimising a data graph. It must be ensured, however, that between leaving `awaitAll` and entering the pickler, no other thread is run.

6.3 Minimisation

The minimisation algorithm was always applied to a rooted subgraph of the data graph. If the equivalent nodes are collapsed as suggested in section 5.5.3, a situation like the one shown in Figure 6.3 may arise: The subgraph reachable from `v` is minimised, but yet a non-minimal copy of it has to be kept because it was referenced from some node `v'`.

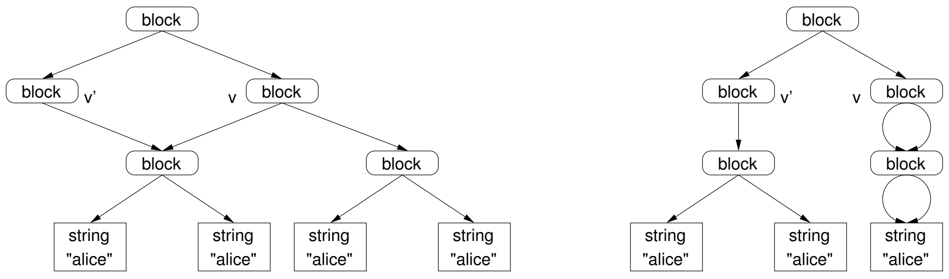


Figure 6.3: Minimisation of subgraphs

6.3.1 Collapsing using Redirection Nodes

The use of transients can cure this: Collapsing of nodes `v` and `v'` is done by calling `become(v,v')`. This may be dangerous if the optimised "direct" store operations are used, because this `become` operation can arbitrarily introduce redirection chains.

In the current Alice implementation, only data structures built by user programs are minimised, and the "direct" operations are only used for internal data structures of the system. Such design guidelines have to be followed to make minimisation a safe operation.

6.3.2 A Minimal Store Area

The use of redirection nodes for collapsing allows for another application of minimisation: An abstract store containing an area which is always kept minimal. Without redirection nodes, adding of a new node to the minimal store could end up in a situation like the one in Figure 6.3.

A minimal store area offers an extremely cheap equality test: Two values that are both in the minimal store are semantically equal exactly if they are represented by the same node.

Figure 6.4 shows a simple implementation of a minimal store area in Alice, realised as an abstract type. It makes use of the minimisation interface from Section 5.3.1. The `addVal` function takes an arbitrary value, adds it to the minimal store (represented as a list) and minimises it. A value of an abstract type is returned so that only minimised values can be tested for equality using the `equal` function. The original value can be accessed through the `getVal` function.

```
signature MIN_STORE =
    sig
        type a
        type t
        val addVal : a -> t
        val getVal : t -> a
        val equal : t * t -> bool
    end

functor MkMinStore(Arg:sig type a end) :> MIN_STORE
    where type a = Arg.a =
    struct
        type a = Arg.a
        type t = Arg.a
        val addVal =
            let
                val minstore = ref nil
            in
                fn a =>
                (minstore := a::(!minstore);
                 Minimizer.minimize (!minstore);
                 a)
            end
        fun getVal a = a
        val equal = UnsafeValue.same
    end
```

Figure 6.4: Implementation of a simple minimal store area

This kind of implementation reveals a problem: The root nodes of the graphs contained in the minimal store area are kept alive as long as the minimal store is alive. They are prevented from becoming garbage, possibly yielding a memory leak. There are two ways of solving this problem: Either there is an operation that deletes a node from the minimal store area, or the store itself provides a mechanism called *weak tables*. A weak table is a table of store nodes that is not scanned during garbage collection but cleaned after a garbage collection so that each dead node is removed.

6.4 Pickling and Unpickling with Transformations

This section presents a generic framework that can transform data graphs when they are pickled and unpickled. Herlihy and Liskov already discuss this idea in the context of CLU [16], where these transformations are called encoding and decoding. This thesis uses the terms *abstraction* (for the transformation done on pickling) and *instantiation* (for the unpickling transformations), following the terminology of Brunklaus and Kornstaedt [5].

6.4.1 Abstraction and Instantiation

Abstraction and instantiation are functions conforming to the following signature:

```
val abstract : addr -> addr
val instantiate : addr -> addr
```

Both functions can do arbitrary computations involving the whole subgraph reachable from their argument. They return a store node representing an abstract, external or instantiated, internal version of their argument, respectively.

Abstraction Model

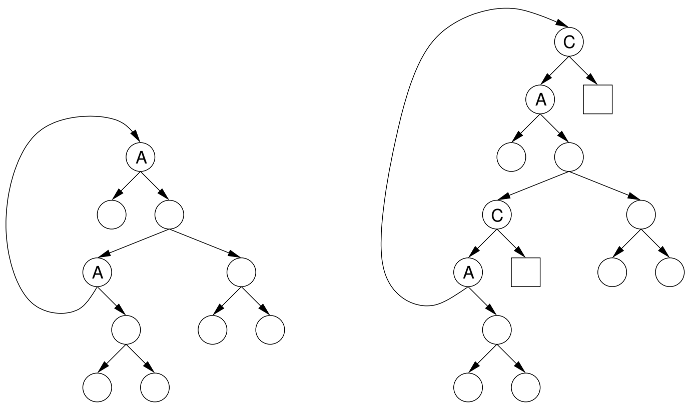
Abstraction is done top-down, node-wise, and with memorisation of the already abstracted nodes. In other words, abstraction uses a depth-first traversal of the data graph, and the `abstract` function is applied to each child of a node before the traversal descends into that child.

Instantiation Model

Nodes are instantiated in the depth-first postorder, thus in a bottom-up way. Sharing as well as cycles have to be treated specially: Any edge leading to a node `v` must be redirected to the node `instantiate(v)`.

Example

The following figure illustrates the ideas behind abstraction and instantiation. In this example, instantiation only changes nodes with label `A` (for abstract) and creates a new node with label `C` (for concrete). This conrete node has the abstract node as its first and a chunk as its second child (depicted by a box). Abstraction of such an instantiated, concrete node is just a projection to its first child (hence recovering the original abstract node).



Abstract data graph Instantiated data graph

A model similar to this is used in SEAM for transforming code during pickling and unpickling – this is discussed below as a potential application.

Reasoning about Transformations

The transformation model introduced above is quite similar to the one investigated by Herlihy and Liskov [16]. They reason about soundness of those transformations by stating invariants that abstraction and instantiation (which they call encoding and decoding) have to obey. Finally, they discuss problems connected with cyclic data structures: Using the terminology introduced above, this means that an instantiation operation may not use a node's children – for example by descending further – unless it can be sure that they have already been instantiated (which cannot be guaranteed generally for cyclic data graphs).

6.4.2 Modifications to Pickler and Unpickler

Applying transformations during pickling is easy: The `abstract` function has to be applied during the depth first traversal before descending into a node's children, thus descending into the abstracted children of the node instead. This can be achieved by providing a modified version of `getBlockArg` that the pickler uses to access nodes:

```
val getBlockArg' = abstract o getBlockArg
```

In addition, the pickler must be invoked with `abstract(r)` instead of `r` as the root node.

For the implementation of instantiation, transients are essential: The unpickler cannot predict the result of the instantiation when it encounters a PROMISE, so it cannot create a node with the right label and arity that is later back-patched when its children are available. A promise realised as a future, though, can be created and bound to the result of the instantiation afterwards. Implementing promises this way also respects the instantiation model as described above, because all edges originally pointing to the promise will be redirected to its instantiated version when the future is bound.

The implementation of the unpickler remains the same as before, but a new layer of abstraction is put between store and unpickler. This layer provides the following operations:

```
val allocBlock' : label * addr list -> addr
val promiseBlock : unit -> addr
val fulfilBlock : addr * label * addr list -> addr
```

The lists of addresses are the children of the block that is created. `promiseBlock` returns an `addr` that is a future, and `fulfilBlock` takes this future, allocates the final block and binds the future to it.

Assuming that there is a function `allocFuture` returning a fresh future, the abstraction layer can be implemented as follows:

```
fun allocBlock' (l, addrs) =
   let
      val block = allocBlock(l, List.length addrs)
   in
      List.appi (fn (i, n) => setBlockArg(block, i, n)) addrs;
      instantiate block
   end

fun promiseBlock () = allocFuture()

fun fulfilBlock (a, l, addrs) =
   let
      val block = allocBlock'(l, addrs)
   in
      become(a, block);
      block
   end
```

With this layer between abstract store and pickler/unpickler, neither abstraction nor instantiation requires to build the complete transformed graph. Both operations can be performed on-the-fly during pickling and unpickling.

6.4.3 Possible Applications

The main application for this transformation mechanism is the representation of code in pickles. The `instantiate` function can take code in an external, platform-independent representation and generate an internal, possibly platform-dependent or otherwise optimised representation. On pickling, the `abstract` function creates an external representation again. This external representation can either be recomputed, or just returned if the `instantiate` function memorised it as part of the internal representation.

Transformations offer a more dynamic way of coping with resources. Certain objects may be resources internally, but one can find an external, non-resource description for them. The mechanism presented here can automatically do the necessary transformations during pickling and unpickling. Numbers (integers and floating point numbers) are candidates for such a conversion. Resources like operating system file descriptors could be abstracted into an external representation and bound to a corresponding resource on unpickling again – this is of course only possible for resources that are present on every system, like for example `stdin`. This technique is called *resource re-binding*.

Programming languages implemented on top of the abstract store could make the transformation mechanism available on the language level; one interesting application could be a Java-like, object based transformation mechanism as discussed in 4.6.1. It is unknown, however, how this can be done in a type-safe way in a strictly typed functional language like Alice.

6.4.4 Concrete Implementation

The transformation mechanism was already present in the former SEAM pickler implementation by Brunklaus and Kornstaedt. It was however not well understood: The transformations are inherently bottom-up (as a subgraph can only be transformed after it has been completely built), but the pickler embarked on a top-down strategy. The result was a mixed top-down/bottom-up unpickler.

In SEAM, transformation is mainly used for transforming code, but also resources are realised this way: The transformation function raises an exception when applied to a resource.

The transformation function itself is not hard-wired into the system but can be augmented with user-defined functions. The identification is not only done by the label:

External nodes have the label TRANSFORM and as their first child a string with an identifier. The corresponding transformation function is looked up in a table under that identifier and then applied.

Internal nodes are identified by the label CONCRETE and directly reference their transformation function via the first child.

7 Discussion and Outlook

This chapter concludes the thesis with a discussion of the concepts that were introduced and an outlook onto future work.

7.1 Summary

This thesis introduced, as its basic notion, the formal model of data graphs. Based on this, an architecture of a language-independent abstract store was developed, together with some examples how high-level data structures can be encoded into data graphs.

Pickling, encoding a data graph into an external, linear representation, was the main abstract store service introduced in this thesis. Its application areas are persistent storage and communication (distribution) of dynamically created data structures over a network. The algorithms presented here offer an efficient, generic, platform- and language-independent way of storing and distributing data structures.

The data graph model distinguishes between nodes with structural equality and those with token equality. This information is needed for minimisation, which was described as an algorithm computing a partition of the data graph. Every class in this partition contains only semantically equal nodes, so that redundant nodes can be deleted. Some technical details were discussed that are crucial for an efficient implementation of the minimisation algorithm. Benchmark figures suggested that if a data graph is minimised before it is pickled, the pickle sizes shrink considerably, while the cost of pickling is increased only moderately.

As an extension of the data graph and abstract store models, transients were introduced. They are nodes which can later be merged with other nodes. This concept has interesting applications, as it can model logic variables and futures. The abstract store had to be extended by just one operation, `become`, to efficiently support transients. Internally, binding a transient is done by introducing redirection nodes that are hidden behind the store interface and eliminated by the garbage collector.

One interesting application made possible by transients is the transformation of a data graph while it is being unpickled. Neither unpickler nor abstract store interface have to be modified, only an additional layer of abstraction is needed between them.

7.2 Outlook

This section poses questions that were raised or remained open while I wrote this thesis. All of them seem to be interesting starting points for future research.

7.2.1 Abstract Store

A question touching the base of the design is whether the abstract store's level of abstraction, its level of structure, is chosen well. The claim is that the data structures of arbitrary languages can be encoded into store data structures. Some more research has to be done whether this encoding is in fact efficient; the Java implementation on SEAM [5] should be evaluated further.

Having an architecture that supports multiple languages, inter-language operability suggests itself as a research topic. Especially the similarities between the abstract store as a semi-structured data model and XML (which was originally designed with language independence in mind) should be investigated, maybe some techniques used in the context of XML can be reused for data graphs.

The store interface from Section 3.2 uses a general purpose type `addr` for nodes in the store. The programmer always has to either test for the node type or be sure by construction of the data structures that all operations are applied to the correct node types. This is a potential source for type errors that cannot be discovered statically. It is not clear but interesting how this interface can be realised in a statically type-safe way.

7.2.2 Minimisation

The minimisation algorithm has been implemented successfully for SEAM. So far, its only application is the minimisation of pickles. Neither of the two applications suggested in Section 5.3.3 and Section 6.3.2 have been implemented yet.

Especially the minimal store area is a unique, new concept that deserves attention. On the implementation side, it should be investigated whether such a concept can be integrated directly into the store: a separate memory area, such that it is always known from a node's address whether it is already minimised or not. This is a low-level issue – the abstract store and the garbage collector have to treat this area specially.

On a higher level, it is interesting to explore applications for such a minimal store; they should take advantage of the fast equality test that is possible for minimised data. Originally, the intention was to minimise the representation of run-time types (RTTs) in Alice, because they seem to consume a lot of memory in the system. The problem is that the current data type for RTTs is not suited for minimisation; it is not obvious what a better representation may look like. Furthermore, the algorithms that handle RTTs should take advantage of the unique representation and use the cheap equality test. It should be a worthwhile topic for future research to develop a RTT representation and algorithms with minimisation in mind.

7.2.3 Pickling

The implementation of the pickler in SEAM is stable, fast and usable. Yet there are some issues that deserve a closer look (ordered from low-level to high-level):

Robustness of the Pickle Format

The current SEAM unpickler can only check implicit consistency of pickles: whether there is no stack overflow, whether each register is assigned only once (which is a property of the current pickler implementation), whether each LOAD instruction has a corresponding STORE or PROMISE, and whether each PROMISE is fulfilled.

Transmission errors like swapped or missing bytes may remain undetected if they occur inside scalar data or labels. As a solution, an explicit consistency check like the Cyclic Redundancy Check (CRC) [29] should be integrated.

Of course this does not reduce the risk of forgery: Still, a malicious party can forge pickles and tag them with a correct checksum.

Modal Pickling

Different contexts may impose the need for a different behaviour of the pickler. Persistence and distribution, for example, could implement different strategies for dealing with resources: Persistence should fail, while distribution could create a "proxy", a remote handle for the resource. Java implements modal pickling [32] and considers it important.

Microsoft .NET also provides a sort of modal pickling, the three pickler back-ends (for a binary format, SOAP, and XML) show differing behaviour concerning which fields of an object get pickled. It is not clear whether this serves any purpose; if so, it should be investigated whether the SEAM pickler can benefit from a similar mechanism.

The question in SEAM is always whether a feature belongs to the abstract machine level or to a language level. This has to be answered for modal forms of pickling.

Language-independent Security Features

Two features that could make the use of pickles more secure are authentication and encryption.

Authentication means that the unpickler can verify that the pickle was created by someone who is trusted. There are standard cryptographic tools to achieve this.

Encryption using strong cryptography yields a secure channel between the creator of the pickle and its receiver. If the pickle is not directly written to a file but to some kind of "stream", this stream can be encrypted. Java follows this approach [18]. For network transmission, standard techniques like for example SSL can be applied.

For both authentication and encryption, some form of certificate or key management is needed; the cryptographic public keys have to be stored, managed and checked. This cannot be done without user interaction.

As these cryptographic methods do not rely on properties of the programming language, it should be possible to implement them in a language-independent way on the level of the pickler.

Language-dependent Security Features

The most important security feature currently implemented in Alice is the dynamic type check at run-time. Unfortunately, this check only tests for the consistency at the module level. It is not possible to check whether the actual data in a pickle really is a representation of a value of that type. It is not obvious how this can be guaranteed or tested.

Similar ideas exist for code: Java does a "byte-code verification", testing for some invariants that the code has to obey. Thinking out this concept leads to an interesting area of research: proof-carrying code [26]. Perhaps these techniques can be implemented for Alice/SEAM and coupled with the code transformation mechanism.

Type-Safe Transformations

As stated in Section 6.4, some work should be invested into making the transformation mechanism available in Alice. It is not obvious at all how this can be done in a type-safe way: The pickling and the unpickling process have to agree upon the type of the pickled data, and furthermore, the transformation functions have to be compatible. Herlihy and Liskov [16] discuss such transformations for abstract types in CLU, and it should be investigated in how far their results can be used for Alice.

8 Bibliography

[1] H. Ait-Kaci. *Warren's Abstract Machine*. Logic Programming. MIT Press, Cambridge, Massachusetts;London, England, 1991.

[2] The Alice Project. Available from `http://www.ps.uni-sb.de/alice`, 2003. Homepage at the Programming Systems Lab, Universität des Saarlandes, Saarbrücken.

[3] A. W. Appel and D. B. MacQueen. Separate compilation for Standard ML. In *Proceedings of the ACM SIGPLAN '94 conference on Programming language design and implementation*, pages 13–23. ACM Press, 1994.

[4] A. Birrell, 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*, pages 149–154. ACM Press, 1987.

[5] Th. Brunklaus and L. Kornstaedt. A virtual machine for multi-language execution. Technical report, Programming Systems Lab, Universität des Saarlandes, Saarbrücken, November 2002. Available from `http://www.ps.uni-sb.de/Papers/abstracts/multivm.html`.

[6] P. Buneman. Semistructured data. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 12-14, 1997, Tucson, Arizona*, pages 117–121. ACM Press, 1997. Invited Tutorial.

[7] A. Cardon and M. Crochemore. Partitioning a graph in $O(|A| \log_2 |V|)$. *Theoretical Computer Science*, 19(1):85–98, July 1982.

[8] C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–8, November 1970.

[9] The Mozart Consortium. The Mozart programming system. Available from `http://www.mozart-oz.org`, 2003.

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

[11] A. Goldberg and D. Robsen. *Smalltalk-80. The Language*. Addison-Wesley Series in Computer Science. Addison-Wesley, 1989.

[12] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, 1996.

[13] M. Habib, Ch. Paul, and L. Viennot. Partition refinement techniques: An interesting algorithmic tool kit. *International Journal of Foundations of Computer Science*, 10(2):147–170, 1999.

[14] R. H. Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(4):501–538, 1985.

[15] M. A. Harrison. *Introduction to Switching and Automata Theory*. McGraw-Hill Series in Systems Science. McGraw-Hill, New York;St. Louis;San Francisco, 1965.

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

[17] J. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In Z. Kohavi and A. Paz, editors, *Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.

[18] Java Object Serialization Specification. Available from `http://java.sun.com/j2se/1.4/docs/guide/serialization/`, 2001.

[19] R. E. Jones and R. D. Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, New York, 1996.

[20] S. L. Peyton Jones and J. Salkild. The spineless tagless g-machine. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 184–201, London, UK, September 1989. ACM Press.

[21] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, 1983.

[22] B. Liskov and St. Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN symposium on Very high level languages*, pages 50–59. ACM Press, 1974.

[23] L. Mauborgne. An incremental unique representation for regular trees. *Nordic Journal of Computing*, 7(4):290–311, 2000.

[24] Microsoft. Microsoft .NET. Available from `http://www.microsoft.com/net`, 2003.

[25] M. L. Minsky. A Lisp garbage collector algorithm using serial secondary storage. Technical Report Memo 58 (rev.), Project MAC, MIT, Cambridge, MA, December 1963.

[26] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, January 1997. ACM Press.

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

[28] The OCaml programming system. Available from `http://www.ocaml.org`, 2003.

[29] W. H. Press, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C++: The Art of Scientific Computing*. Cambridge Univ. Press, Cambridge, second edition, 2002.

[30] The Python programming language. Available from `http://www.pyhton.org`, 2003.

[31] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS '02; 17th Annual IEEE Symposium on Logic in Computer Science; July 22-25, 2002, Copenhagen, Denmark; Proceedings*, Washington;Brussels;Tokyo, 2002. IEEE.

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

[33] The Ruby programming language.
Available from `http://www.ruby-lang.org`, 2003.

[34] R. Scheidhauer. *Design, Implementierung und Evaluierung einer virtuellen Maschine für Oz*. Dissertation, Universität des Saarlandes, Fachbereich Informatik, Saarbrücken, December 1998.

[35] J. Schwinghammer. *A Concurrent Lambda-Calculus with Promises and Futures*. Diploma thesis, Programming Systems Lab, Universität des Saarlandes, Saarbrücken, February 2002. Available from
`http://www.cogs.susx.ac.uk/users/js35/publications/da.pdf`.

[36] R. Sedgewick. *Algorithms in C++*. Addison-Wesley, third edition, 2002.

[37] G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 324–343. Springer-Verlag, Berlin, 1995.

[38] G. Smolka and R. Treinen. Records for logic programming. *Journal of Logic Programming*, 18(3):229–258, April 1994.

[39] R. E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[40] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 157–167. ACM Press, 1984.

[41] R. Wilhelm and D. Maurer. *Compiler Design*. International Computer Science Series. Addison-Wesley, 1995.

[42] S. Woop and M. Horbach. Incremental algorithms and a minimal graph representation for regular trees.
Available from `http://www.ps.uni-sb.de/~horbach/fopra.html`, 2002.