Design and Implementation of a Programmable Middleware

Leif Tobias Kornstädt

Dissertation

zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

Saarbrücken
eingereicht am 26. April 2006

Leif Tobias Kornstädt
6033 150th Ct NE
Redmond, WA 98052
USA

kornstaedt@hotmail.com

Abstract

This thesis presents a language-based, safely programmable middleware for the simple, high-level, and expressive construction of composable open systems. The middleware provides services for pickling, components, and distribution. All are based on a minimal set of primitives and syntax extensions, such that they otherwise can be completely implemented and customized in a high-level language with automatic memory management, exception handling, higher-order functions, futures, and dynamic types. Using this approach, it becomes possible to describe the *complete* architecture of the middleware system, and to leverage the language's safety features in the middleware itself.

Zusammenfassung

Die vorliegende Arbeit beschreibt eine Programmiersprachen-basierte programmierbare Middleware, die eine einfache Konstruktion offener Systeme auf hoher Ebene ermöglicht. Die Middleware bietet Dienste für Pickling, Komponenten und Verteilung an, die allesamt auf einem minimalen Satz an Primitiven und Syntaxerweiterungen beruhen. Der Hauptteil der Dienste kann so in einer höheren Programmiersprache mit automatischer Speicherverwaltung, Ausnahmebehandlung, Prozeduren höherer Ordnung, Futures und dynamischen Typen realisiert werden. Dies ermöglicht es, die Architektur der Middleware *vollständig* zu beschreiben, sowie die Sicherheitsgarantien der höheren Programmiersprache in der Implementierung der Middleware selbst zu nutzen.

Extended Abstract

This thesis presents a language-based, safely programmable middleware for the simple, high-level, and expressive construction of composable open systems.

An *open system* is a software system that can non-trivially interact with independent software components and other software systems, which may possibly not have existed at the time the system was deployed. Since the advent of the World Wide Web, most modern software systems are open systems.

The implementation of open systems can be greatly simplified by the use of a powerful, generic middleware. A *middleware* is a layer in a software architecture that mediates between an application and its heterogeneous networked computing environment.

This thesis presents the design and implementation of a middleware whose services fall in the areas of pickling, components, and distribution. Based on a minimal set of primitives and syntax extensions, the middleware is otherwise implemented completely in a high-level language. This work assumes any programming language with automatic memory management, exception handling, higher-order functions, futures, and dynamic types.

There are a number of ramifications to this approach. It becomes possible to describe the *complete* architecture of the middleware system. All language safety features directly carry over to the middleware. Since the primitives are safe, they can be made accessible to programmers. This makes all middleware services based on the primitives programmable: application programmers can devise new services and design patterns as they need them.

The underlying primitives provide generic services. *Pickling* provides a unified model for data exchange, messages in distributed communication, and the representation of components. *First-class components*, with operations to reflect metadata, link components, and dynamically create components at run time, allow to program powerful component services, namely lazy dynamic linking; sandboxing; exporting dynamically-created components; symbolic, structured names for components and customizable name lookup; and hiding at the level of components. A *distribution* layer includ-

ing remote invocation is built on top of components; all that requires in terms of primitives is a dynamic call mechanism.

The approach of this thesis is to devise a minimal set of safe primitives for each service, and to demonstrate how the service can be implemented in the high-level language on top of those primitives. The work examines the properties of the resulting system to discuss alternatives. This leads both to a thorough understanding and discussion of middlewares, and to a standard library of middleware services in generally useful incarnations.

All concepts have been implemented in the context of two systems: Mozart, a production quality system implementing Oz, which pioneered most concepts; and Alice, whose design was founded on the presented concepts. In fact, as the base language and generic runtime system are incrementally extended, the system built in the thesis more and more resembles full Alice.

Ausführliche Zusammenfassung

Die vorliegende Arbeit beschreibt eine Programmiersprachen-basierte programmierbare Middleware, die eine einfache Konstruktion offener Systeme auf hoher Ebene ermöglicht.

Ein *offenes System* ist ein Softwaresystem, das in nichttrivialer Weise mit unabhängigen Softwarekomponenten und anderen Softwaresystemen interagieren kann, die womöglich zu dem Zeitpunkt, da das System in Betrieb genommen wurde, noch nicht existiert haben. Seit der Einführung des World Wide Web sind die meisten modernen Softwaresysteme offene Systeme.

Offene Systeme können durch den Einsatz ausdrucksstarker Middleware bedeutend einfacher implementiert werden. *Middleware* ist die Schicht in einer Softwarearchitektur, die zwischen einer Anwendung und ihrer heterogenen, vernetzten Umgebung vermittelt.

Diese Arbeit stellt Entwurf und Implementierung einer Middleware vor, deren Dienste in die Bereiche Pickling, Komponenten und Verteilung fallen. Die Middleware basiert auf einem minimalen Satz an Primitiven und Syntaxerweiterungen und ist andernweitig vollständig in einer höheren Programmiersprache realisiert. Dabei wird eine beliebige höhere Programmiersprache mit automatischer Speicherverwaltung, Ausnahmebehandlung, Prozeduren höherer Ordnung, Futures und dynamischen Typen angenommen.

Der Ansatz bringt eine Reihe von Vorteilen mit sich. Es wird möglich, die *vollständige* Architektur der Middleware zu beschreiben. Die Sicherheitsgarantien der Programmiersprache übertragen sich auf die Middleware. Da die Primitive sicher sind in dem Sinne, dass sie nicht verwendet werden können in einer Art, die die Garantien der Sprache verletze würde, können sie dem Programmierer direkt zugänglich gemacht werden. Hiermit sind alle Dienste der Middleware, die auf diesen Primitiven basieren, programmierbar: Anwendungsprogrammierer können ihre eigenen Dienste und Entwurfsmuster ersinnen, sobald sie solche benötigen.

Die Dienste, die die Primitive erbringen, sind generisch. *Pickling* bietet ein vereinheitlichtes Modell für den Datenaustausch, für Nachrichten in der Kommunikation zwischen verteilten Systemkomponenten und für die Darstellung von Softwarekomponenten. *Komponenten erster Klasse* bieten Operationen zur Projektion von Metadaten, zum Binden von Komponenten und

zur dynamischen Erzeugung. Dies erlaubt die Programmierung mächtiger Komponentendienste, etwa verzögertes dynamisches Laden und Binden, Ausführung von Programmen in abgesicherter Umgebung (*sandboxing*), Export dynamisch erzeugter Komponenten, anpassbare Auflösung strukturierter symbolischer Bezeichner für Komponenten, sowie Verdeckung auf der Ebene von Komponenten. Eine *Verteilungsschicht* mit ferngesteuerter Aktivierung wird auf Komponenten aufgebaut; hierzu wird lediglich ein weiteres Primitiv für dynamisches Binden benötigt.

Die Arbeit geht folgendermaßen vor. Für jeden Dienst wird ein minimaler Satz von Primitiven bestimmt und es wird aufgezeigt, wie der Dienst mit Hilfe dieser Primitive in der höheren Programmiersprache realisiert werden kann. Die Eigenschaften des resultierenden Systems werden untersucht und mit Alternativen verglichen. Das führt zu einem guten Verständnis der Middleware, sowie zu einer Standardbibliothek von Middlewarediensten in allgemein nützlicher Form.

Sämtliche Konzepte wurden in zwei Systemen implementiert: Mozart, einem serienreifen System, das die Sprache Oz implementiert und die Konzepte als erstes System anbot, sowie Alice, dessen Entwurf auf eben den vorgestellten Konzepten basiert. Im Zuge der inkrementellen Erweiterungen der Basissprache und der generischen Laufzeitumgebung wird das vorgestellte System tatsächlich Alice immer ähnlicher.

Contents

Chapter 1

Introduction

This thesis presents a language-based, safely programmable middleware for the simple, high-level, and expressive construction of composable open systems.

1.1 Middleware

Today, software developers need to build extensible, heterogeneous, and dynamic systems, also called *open systems*. In most general terms, an open system is a software system that can interact in any non-trivial ways with independent software components and other software systems, which may possibly not have existed at the time the system was deployed.

The World Wide Web plays host to plenty of examples of open systems: Web servers such as Apache [Apa03] are run-time configurable to accommodate new methods for generating dynamic content; Web browsers such as the Microsoft Internet Explorer [Mic02] download new ActiveX controls [Cha96] on-the-fly to enrich user experience; and servers dynamically exchange information by means of Web Services [BHM+04] discoverable at run time. Another example for an open system is the extensible architecture of the Eclipse development environment, whose core, the *plug-in engine* [Bir05], manages installation, loading, configuring, *et cetera*, of plug-ins—in other words, it is a middleware.

The implementation of open systems can be greatly simplified by the use of a powerful, generic middleware. A *middleware* is a layer in a software architecture that mediates between an application and its heterogeneous networked computing environment, to enable the application to participate in an open system. The ideas of middlewares have been around since the late 1970's, but the term has only been coined in the early 1990's [CMHC03]. This work focuses on how middleware automates data exchange, discovery of services, and their execution both in-process through dynamic code loading, and out-of-process through remote procedure calls.

Data Exchange. To participate in an open system, an application has to be able to communicate with its peers. This makes data exchange a fundamental feature of a middleware.

Packaging Services. Services provided by software can be packaged as self-describing and independently usable program fragments, called *components* [Szy02]. Components are self-describing in that they explicitly specify dependencies on other components and the types of the objects they define. Systems are built out of sets of components by means of *linking.* The middleware defines a component model and operations to link components.

In-process Execution. Linking can be dynamic, which means that it enriches a running application with the objects defined by a new component. Components can be discovered at run time. Configurable environments for dynamic linking provide for access control and *sandboxing*: application fragments can be executed in a "sandbox", meaning that they only have restricted capabilities so that they can cause no harm. The middleware provides the services of run-time discovery and dynamic linking, and enforces execution policies.

Out-of-process Execution. In addition to running in-process, services can be hosted in independent processes, called *servers*, be it for security reasons, distribution, or longevity. The middleware must provide mechanisms for starting server processes and for establishing connections to existing servers possibly running on remote machines.

1.2 Motivation

The design and implementation of middleware is a hot research topic. This thesis addresses the following aspects for which, I believe, as yet no satisfactory solutions exist.

Simplicity and Understandability. Many middlewares are large and complex, making their design and implementation prone to error and hard to communicate. This work addresses the issue by designing a middleware that is as simple as possible and requires only minimal low-level support, making it possible to describe its *complete* architecture.

Safety and Security. Designing and implementing open applications is inherently difficult. The number of security bulletins [DFN03] and functional patches paints a desolate picture of the state of the art of application vulnerability and reliability.

The conjunctions in the preceding sentence indicate that there are two dimensions to this problem—safety and security. A *safe* programming environment *protects programmers* from making mistakes. In contrast, a *secure* programming environment *protects itself* against attacks from third parties.

It is a fact of life that an application built on an insecure middleware cannot be secure. But building an application on a secure middleware does not, by itself, make the application secure, either. Still, a middleware can be a key player in increasing both safety and security: A secure *and safe* middleware is required for application developers to build secure applications. The middleware has to provide expressive, high-level abstractions that shield applications from requiring low-level or unsafe code. These abstractions then need to be implemented and secured just once, and all applications benefit from it.

On the other hand, designing and correctly implementing a middleware is not easy, either: consider, for instance, the problems initially present in Java and the Java Virtual Machine [DFW96]. This thesis explores a simple idea: Just as secure applications can more easily be built on top of a safe middleware, a secure middleware can more easily be built on top of a safe programming language.

Ease of Use and Expressivity. Mechanisms such as mobile code [FPV98] can enrich the toolset of the application developer immensely. Yet, unless they are provided by the underlying platform, they are often too complex to afford.

A middleware warrants the investment of developing and securing expressive mechanisms. This work performs this for several mechanisms, among them mobile code. The mechanisms must add as little complexity as possible to the middleware itself, and must not make the platform harder to master. For greatest flexibility, the mechanisms must be arbitrarily composable.

Programmability. The patterns that application programmers can express their design with are limited by what the middleware offers. In many cases, there is no “one size fits all”. The goal of this work is to lift limitations by making the middleware highly programmable: only a small number of primitives are fixed, and the actual services are provided in the high-level language. Services become accessible to and modifiable by application programmers: they can devise new design patterns and corresponding services as needed.

Unified Models. Externalization of data in the form of *pickling* (also called *serialization*) on the one hand, and components on the other hand are two fundamental services provided by a middleware. In all existing approaches, these two are distinct, making mobile code a non-trivial extension. There is no intrinsic reason for not using pickling as the unified file format for data, communication messages, and components.

1.3 Approach

This thesis takes the following route to address the shortcomings of existing approaches mentioned above.

A Language-based Middleware. With a minimal set of primitives and syntax extensions, I am able implement the middleware in the language itself. Modern high-level languages provide for strong typing, automatic memory management, parameterization and abstraction, and exception handling. With the present approach, all of these directly carry over to the middleware. Since the primitives are safe, they can be made accessible to programmers, enabling them to implement custom middleware services—in other words, the middleware becomes programmable.

Safety and Security. As motivated above, this work's main focus is on safety. In fact, the only specific security feature addressed is sandboxing, under an appropriate assumption (pickles need to be verifiable—which is outside the scope of this work). No other material in this thesis fundamentally depends on this assumption.

The Language. A safe programming environment must be based on strong typing. The focus of this work is first on dynamically-typed languages, which enforce strong typing by raising exceptions at run time on type mismatches. A secondary concern is how much of the approach carries over to statically-typed languages, where this work identifies a number of limitations.

At the outset, my solution assumes any programming language with support for higher-order functions, futures, and dynamic types. Specifically, I use a variant of Standard ML extended with futures and packages, which I call L. As I extend the base language, I obtain OpenL, in which lie the contributions of my work.

Complete Architecture. Starting from L and a generic runtime, this work develops the *complete* architecture of a middleware, from the underlying runtime to primitive operations to the actual middleware services. This makes this work unique: other work usually only takes a spot-light approach to a specific area, and gives other areas an ad-hoc treatment.

Pickling. At the outset, pickling is just a mechanism to externalize and internalize graphs of programming language data. In my approach, it additionally serves as the foundation for component deployment and distributed communication. As such, it merits an in-depth treatment: I analyze the design space of pickling and survey existing mechanisms, to make educated decisions on a suitable pickling mechanism for my approach.

Components. Starting from a very simple component model for modular programming, I incrementally devise extensions to provide powerful component services, namely lazy dynamic linking; sandboxing; dynamic creation of first-class components; exporting dynamically created components to other processes; symbolic, structured names for components and customizable name lookup; and hiding at the level of components. As an application of pickling and components, I demonstrate how to provide basic distribution services.

Implementation. For each middleware service, I devise a minimal set of safe primitives and demonstrate how the service can be implemented in the high-level language on top of those primitives. The implementation descriptions are complemented with a qualitative look at incurred costs, and proposals for optimizations to address these costs. Furthermore, they are backed by actual implementations in the context of two systems: Mozart, a production quality system implementing Oz, which pioneered most concepts; and Alice, whose original design was founded on the presented concepts.

1.4 Contributions

This work makes the following contributions:

- This work presents a classification for pickling mechanisms, and surveys a large number of different pickling mechanisms according to the classification.
- This work develops a principled approach to bottom-up pickling and unpickling, and relates the problem to existing algorithms in Computer Science.
- To my knowledge, this work is the first to discuss problems in the interaction of concurrent pickling with state and futures, and to propose solutions.
- A mechanism for customizing pickling at the level of the virtual machine is presented that allows to uniformly capture pickling behaviors of resources, first-class functions, and primitives.
- This work identifies a tiny set of low-level mechanisms sufficient to integrate component programming with first-class components and lazy dynamic linking into a programming language. The mechanisms can be safely exposed to programmers as primitive operations.
- Except for the most basic virtual machine services, the runtime system including the lazy dynamic linker can be built out of components. For instance, the system startup procedure is itself defined by a component.

- When instantiated for a dynamically-typed programming language, the presented approach to linking can be defined on the level of the language, and implemented by language operations.
- This work proposes a simple and systematic way to propagate runtime errors originating from the programming infrastructure, such as dynamic linking errors, by reducing them to the single generic concept of failed values.
- The presented approach is the first, to my knowledge, that represents components as pickles, and reduces loading of components to unpickling. At the same time, this allows to persist dynamically-created components.
- The component model unifies many representations for executable code, which traditionally are static libraries, dynamic libraries, and executables.
- What is traditionally serialized data, is unified with components. A component can contain code, data, or a mixture of these.
- From a client's perspective, even servers running on the network are integrated with the component concept. Servers are simple applications implemented in the high-level programming language.

1.5 Context of this Work

The results presented in this thesis have impacted real programming systems and development projects. This section sketches the context in which I performed my research, namely Mozart and Alice.

1.5.1 Oz and Mozart

Oz is a concurrent, dynamically-typed, data-flow-driven programming language. It has its roots in logic programming, redefined concurrent constraint programming, and evolved into a multi-paradigm language integrating the relational, functional, and object-oriented paradigms. Oz is defined by a small core language that is founded on a well-defined programming model [Smo95]. High-level features are provided by powerful abstractions and are reduced to this core language: for instance, *computation spaces* are abstractions for programming constraint services at a high level [Sch02a], and the object system is reduced to the core language [Hen97].

Development of Oz and its implementation DFKI Oz started at the German Research Center for Artificial Intelligence (DFKI). Researchers from the Programming Systems Lab at Saarland University, the Swedish Institute for Computer Science in Stockholm, and the Université Catholique de Louvain, Belgium, cooperated to release the Mozart Programming System [Moz04] in 1998, with state-of-the-art performance [Sch98].

The pickling-based component services presented in this thesis were pioneered by Mozart [DKSS98, DKS04a]. Mozart and its middleware have been used successfully for many projects both in research and the industry. The system is freely available for download [Moz04], including its source code, and runs on a large number of platforms including Microsoft Windows and Unixes.

1.5.2 Alice ML and the Alice System

To explore how to integrate the middleware services into a statically-typed language, we defined the programming language Alice ML [Smo98, RLT$^+$04, Pro05]. Alice ML is both an extension of Standard ML [MTHM97] that incorporates essential features of Oz and Mozart, and a statically-typed variant of Oz. The system that is incrementally built in this thesis in the end very closely resembles Alice.

Mozart/Oz taught many lessons about what to improve, why to improve it, and how to improve it. Its main deficiencies are missing regularity, orthogonality, simplicity, and genericity [BK03]. The design of the Alice system puts simplicity and modularity first, and addresses efficiency by analyzing the bottlenecks and optimizing where needed. The intent was to make Alice into a flexible research vehicle—among other goals, to support the claims made in this thesis. Alice features its own virtual machine, called Seam [BK02], and a rationalized store and pickling mechanism [Tac03].

Alice is platform-independent; Seam runs on Unix and Windows. Both are available, including source code, for download [Pro05].

1.6 Source Material

The pickling algorithms and data structures from Chapter 5 have previously appeared in Guido Tack's Diploma Thesis, supervised by Gert Smolka and me.

- Guido Tack. Linearisation, minimization and transformation of data graphs with transients. Diplomarbeit, Naturwissenschaftlich-Technische Fakultät I, Fachrichtung Informatik, Universität des Saarlandes, Saarbrücken, Germany, May 2003. [Tac03]

- Guido Tack, Leif Kornstaedt, and Gert Smolka. Generic pickling and minimization. Proceedings of the ACM SIGPLAN Workshop on ML, Tallinn, Estonia, September 2005. [TKS05]

Some of the material about pickling-based first-class components with lazy dynamic linking from Chapter 8 and Sections 9.1 to 9.3 and 10.1 has previously appeared as a technical report. The initial implementation was released with Mozart 1.0.

- Denys Duchier, Leif Kornstaedt, Christian Schulte, and Gert Smolka. A higher-order module discipline with separate compilation, dynamic linking, and pickling. Technical report, Programming Systems Lab, DFKI, and Universität des Saarlandes, Saarbrücken, Germany. September 1998. [DKSS98]
- Denys Duchier, Leif Kornstaedt, and Christian Schulte. Application programming. Tutorial in the Mozart online documentation. 1998–2004. [DKS04a]

Localizers described in Section 10.2 are initially due to Denys Duchier and first appeared in Mozart 1.0.

- Denys Duchier, Leif Kornstaedt, Martin Homik, Tobias Müller, Christian Schulte, and Peter Van Roy. System Modules. Reference manual in the Mozart online documentation. 1998–2004. [DKH+04]

The first implementation of bundling from Section 10.3 was the Oz linker `ozl`, originally due to Christian Schulte.

- Denys Duchier, Leif Kornstaedt, and Christian Schulte. Oz Shell Utilities. Reference manual in the Mozart online documentation. 1998–2004. [DKS04b]

The distribution services developed in Chapter 11 were greatly inspired by Mozart's distribution layer (although diminished in functionality), regarding the classification of distribution behaviors and application programmer's interface.

- Seif Haridi, Peter Van Roy, Per Brand, and Christian Schulte. Programming languages for distributed applications. *New Generation Computing* 16(3):223–251, 1998. Ohmsha, Ltd. and Springer-Verlag. [HVBS98]

1.7 Outline

The body of this thesis is structured as depicted in Figure 1.1. Arrows indicate direct dependencies between sections.

Context. The first part of the thesis provides context for the design developed in this thesis, based on my two main research goals. The first of these is language integration of services—accordingly, the target high-level language is presented in Chapter 2. A clear definition of required low-level primitives is the second research goal; Chapter 3 provides them with the description of a runtime environment to live in.

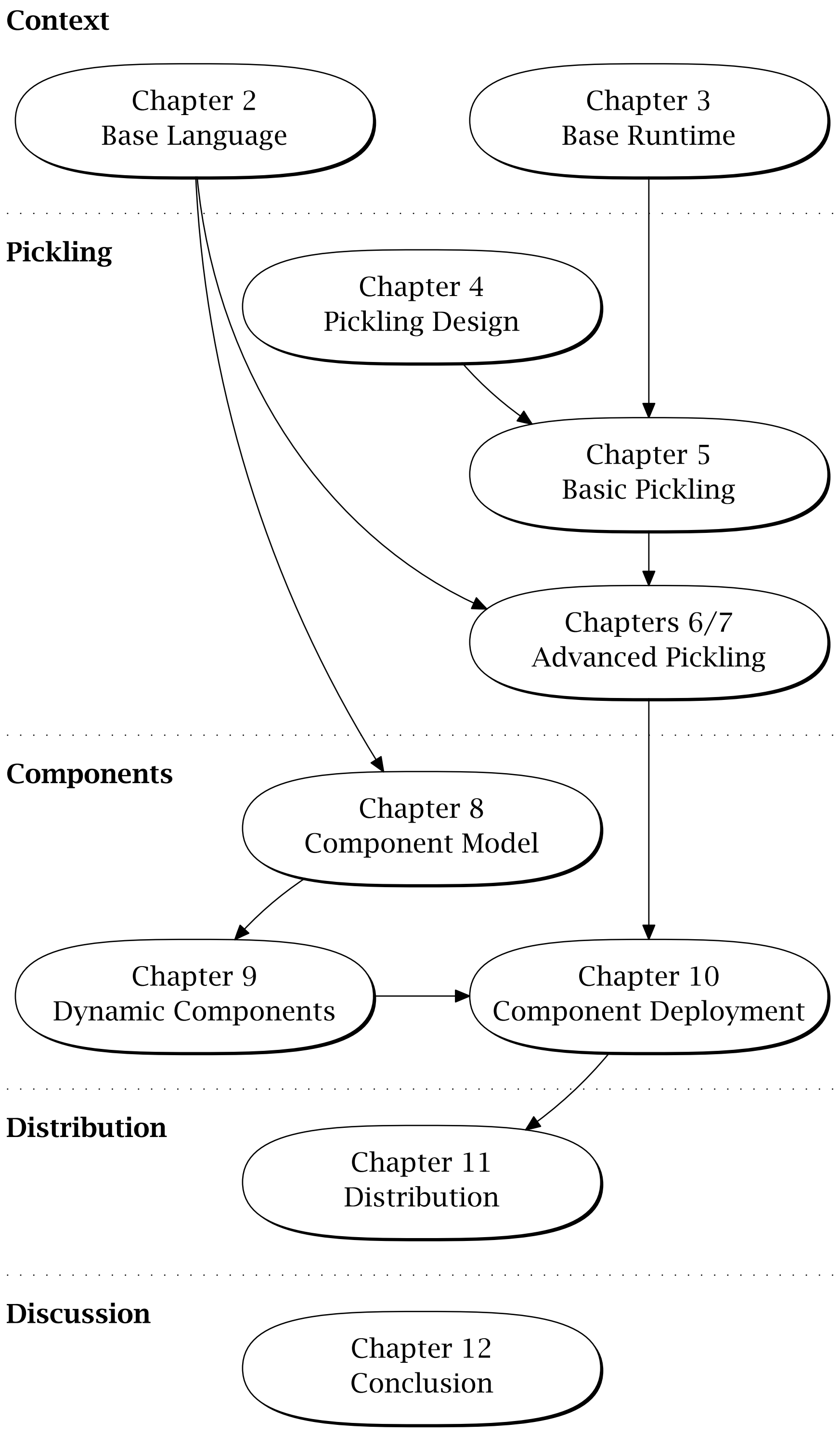


Figure 1.1: Structure of the Thesis: Dependencies between Chapters.

Pickling. Pickling is the basis for the other services, and as such merits a thorough discussion: Chapter 4 explores the design space of pickling. Chapter 5 develops the basic pickling algorithm for terms with sharing, which is generic to be useful for a wide range of programming languages. Chapter 6 analyzes the interaction of concurrent pickling with state and futures. Chapter 7 covers pickling of non-term nodes.

Components. Chapter 8 introduces the component model. The model is extended to cover programmable, lazy dynamic linking in Chapter 9. The deployment of components builds on pickling and is analyzed in Chapter 10. Chapter 11 puts pickling and components into practice to provide support for distributed systems.

Discussion. Chapter 12 concludes with an overview of the complete architecture, summarizes the main contributions, and provides directions for future research.

Chapter 2

The Base Language

This chapter introduces the language that serves as a starting point for the rest of the work. This language is called L, to distinguish it from Alice ML. Extensions of L in later chapters will we called OpenL, and OpenL will become increasingly similar to Alice ML as L is extended.

L's ancestry includes the following languages (summarized in Figure 2.1), as well as existing research:

Standard ML. Like Standard ML [MTHM97], L is a statically-typed non-pure functional language. The design of Standard ML emphasizes safety. In particular, it provides powerful abstraction mechanisms enforced by the static type system. Errors do not crash programs, but lead to *exceptions* that programs can handle. Automatic memory management and the absence of explicit pointers eliminate illegal memory accesses, dangling pointers, and memory leaks. L supports a core language and module system in the spirit of Standard ML's. L uses the syntax of Standard ML.

Oz. From Oz [Smo95], L inherits concurrency and data-flow synchronization. Subsequent chapters extend L with more of the features of Oz, such as pickling (Chapter 5) and components in the style of Oz functors (Chapter 8).

Futures. L provides concurrency and data flow synchronization through *futures*, which are initially due to Multilisp [Hal85]. The model presented here derives from a proposal for Oz by Mehl et al. [MSS98], and was proposed for Alice by Smolka [Smo98]. It has been formalized by Niehren et al. [Sch02b, NSS05]. This model has also been proposed for Java, with the language called Flow Java [DSBH03].

Dynamics. L supports dynamically-typed values in the form of *packages*, inspired by dynamics [ACPP91, ACPR95]. Packages as present in L are due to Rossberg [Ros03, Ros05], who provides a formalization of packages and describes their implementation.

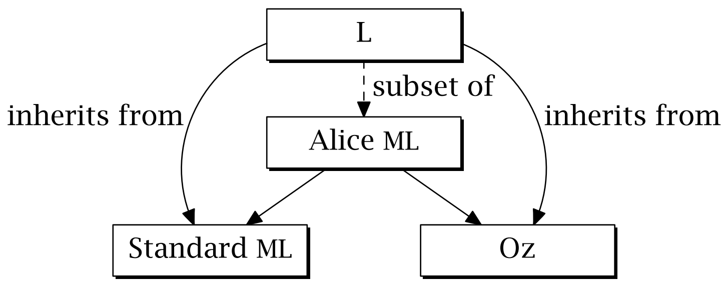


Figure 2.1: Relations between Languages.

This work assumes the reader is somewhat familiar with Standard ML, but does not require the reader to know Oz.

Structure of this Chapter. This chapter presents L's *core language* in Section 2.1, and its *module language* in Section 2.2. In contrast to Standard ML, L has a construct to inject modules into core language values, namely *packages*, presented in Section 2.3. Section 2.4 concludes with some examples of L programs that are not possible in Standard ML.

2.1 The Core Language

Basic data structures, functions, and exception handling in L are borrowed from Standard ML, and are described very succinctly (refer to the definition of Standard ML for details [MTHM97]). Promises and futures, which are not present in Standard ML, are described in more detail.

2.1.1 Data Structures

L programs operate on data in an store that can automatically manage deallocation of data structures. The basic data types are integers and characters. The immutable data structures are records (subsuming tuples), constructed values (annotated with a constructor defined by an algebraic data type), and vectors (cross-products of an arbitrary number of homogeneous elements). In contrast to Standard ML, algebraic data types in L have structural equality—values of two algebraic data types are compatible if and only if the types have the same set of constructors and corresponding constructors' argument types are equal. Some commonly used types are defined in terms of the types above: strings are vectors of characters; booleans and lists are defined as algebraic data types.

L cleanly separates immutable and mutable data structures. The basic stateful data type is the *reference cell*, a container holding a single value that can be updated to hold a different value (of the same type). Reference cells are modeled as constructed values using a designated `ref` constructor. Other stateful data types are conceptually defined in terms of reference cells and immutable data structures. An *array*, for instance, conceptually is a vector of reference cells holding the elements.

L supports Standard ML's structured pattern matching for discriminating on constants and constructors of algebraic data types, and for decomposing structured data.

2.1.2 Functions

Computations can be encapsulated through the definition of *functions*. A function takes a single argument and computes a single value, which is the result of evaluating an application of the function. L is a higher-order language, that is, functions can themselves be used as values (functions are *first-class citizens*). L is lexically scoped: Evaluation of a function definition creates a *closure*, which pairs the function definition with the environment binding its free variables to values.

Functions taking multiple arguments can be expressed either as *Cartesian* functions (taking a record of arguments) or as *curried* functions (functions returning functions). Multiple return values are expressed as records. Zero arguments or no return value are represented as the empty record, the only element of the so-called *unit* type.

The term *function* is commonly both used to mean a closure and a function definition. The present work uses the term function only to denote a closure.

2.1.3 Primitive Operations

The operations on built-in data types are implemented by *primitives*. L programs are evaluated in an *initial environment* that is empty except for bindings of identifiers to primitives. For example, integer addition and polymorphic comparison are bound to the symbolic identifiers + and =.

2.1.4 Exceptions

When an error occurs, L primitives and library functions raise *exceptions*. The exception itself is a value of the predefined type `exn` that describes the exceptional condition. An expression e_1 **`handle`** x => e_2 evaluates to the result of e_1, unless that results in an exception, in which case the exception is bound to x and e_2 is evaluated. Exception handlers are dynamically scoped: When an exception is raised, control is passed to the current computation's innermost installed handler.

2.1.5 Promises and Futures

A *future* is a placeholder in a data structure. The first kind of future in L is the *promised future*. Every promised future has an associated *promise*. A promise can be *fulfilled* with a value at most once, which causes the associated promised future to be replaced by the value. The effect of separating promises and futures is the following: any computation that holds a reference to a future has the permission to read its value, but only a computation that holds a reference to the promise has the permission to define the value. By construction, the computation that creates a future controls access to the promise.

Operations. The type t `promise` is the type of promises for values of type t, and supports the following primitive operations:

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

A promise is created by the primitive function `promise`. The primitive function `future` returns the promise's associated future. Note that the presence of promises in values are type-explicit, while the presence of futures is *not* visible in types. The `fulfill` primitive, given a promise and a value, replaces the future associated with the promise by the value. Any attempt to fulfill a promise that has already been fulfilled results in a `Promise` exception.

Example. The following expression is an example showing how a cyclic list can be constructed using promises and futures—something that is impossible in Standard ML:

```
let
    val tail = promise ()
    val list = 1::future tail
in
    fulfill (tail, list); list
end
```

Promises versus Reference Cells. Promises may be thought of as analogous to reference cells. The `promise` primitive corresponds to the `ref` constructor, except that it does not take an initial value—the initial value conceptually is the implicitly created promised future. `future` is analogous to reference cell access. `fulfill` corresponds to reference cell assignment, but for the single-assignment requirement for promises.

2.1.6 Concurrency and Laziness

L supports concurrency through the `concur` primitive:

```
val concur : (unit -> 'a) -> 'a
```

Given a nullary function f, an application `concur` f causes creation of a new *thread* to evaluate f `()`. Threads execute concurrently and are fairly and preemptively scheduled. `concur` f immediately returns a *concurrent future* x. When the concurrent thread terminates, x is replaced with the result of the evaluation of f `()`.

Data-flow Synchronization. If some computation requires the value of x while it is still a future (it *requests* the value of x), the computation automatically *blocks* until the future is bound. In particular, matching a value against a pattern requests the value, and applying a function requests the function. Additionally, a value of any type can explicitly be requested using the primitive `await`:

```
val await : 'a -> 'a
```

Blocking on a promised future deadlocks if no concurrent thread ever fulfills the corresponding promise.

Observing Failure. Given an application `concur` f, it can happen that the application f `()` in the new thread fails with an uncaught exception. In this case, the concurrent future returned by `concur` f is replaced by a *failed value* storing the exception. Every time a thread requests a failed value, the exception stored in the failed value is raised in the thread (without any effect on the failed value itself). Thus, all clients equally observe a thread's failure and can react upon it, making it possible to program for robustness.

One can define an operation `failedValue` to explicitly create a failed value:

```
val failedValue : exn -> 'a
```

This operation can be expressed in terms of `concur`:

```
fun failedValue e =
    let
        val x = concur (fn _ => raise e)
    in
        await x handle _ => x
    end
```

Laziness. A computation can be made to evaluate lazily, that is, only once its result is needed by some thread, by means of the `byneed` primitive:

```
val byneed : (unit -> 'a) -> 'a
```

`byneed` takes a function f and returns a *lazy future* x. The first thread to block on x causes x to be replaced by the result of `concur` f, which creates

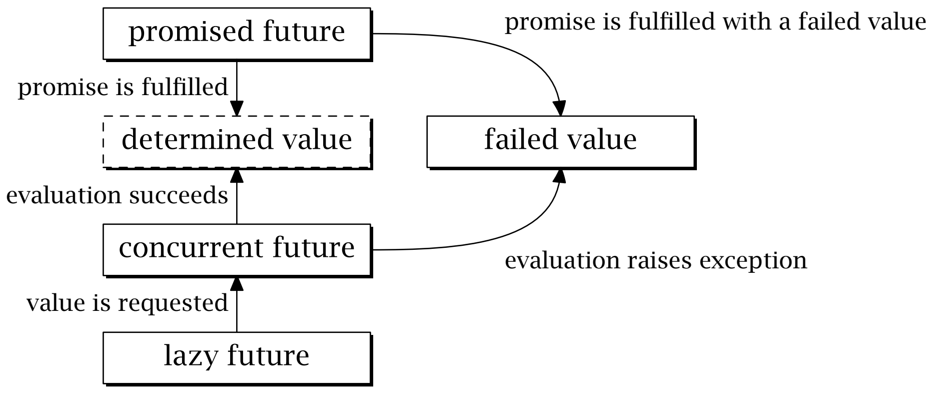


Figure 2.2: Transitions between Futures.

a new thread to evaluate f `()`. This means that even multiple threads racing to request x cause a single evaluation of f `()`. In essence, `byneed` and `concur` differ only in that `byneed` defers the creation of the thread until its result is first needed.

As syntactic sugar, L supports **`lazy`** *e* to mean `byneed (`**`fn`**` _ =>` *e*`)`.

Value Status. If a value is neither a future nor a failed value, we say it is *determined*. The status of a value x—whether x is a promised, concurrent, or lazy future; a failed value; or determined—can be observed with the `status` primitive:

```
datatype status = FUTURE | FAILED | DETERMINED
val status : 'a -> status
```

Transitions. Figure 2.2 summarizes possible transitions between the various kinds of future.

2.1.7 Atomic Exchange

In contrast to Standard ML, L features an `exchange` operation on reference cells:

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

`exchange` atomically combines access and assignment. In combination with promises and futures, atomic exchange forms the basis for thread-safe programming. This is illustrated by examples in Section 2.4.

2.2 The Module Language

L inherits Standard ML's expressive module system: A *structure* is a collection of named values and types—the module language equivalent of core language records. *Signatures* are types of structures. Every structure has an implicit *principal* signature. Structures can be *coerced* to an explicitly annotated signature; the annotated signature must *match* the structure's principal signature. The annotated signature may be more specific, which allows to hide structure members, and, more importantly, allows for *type abstraction*: While the principal signature states the representations of all the types defined in the structure, the annotated signature can hide the actual definition of a type. The definition of abstract types in L relies entirely on this mechanism.

Structures can be computed by *functors*, functions taking a structure as argument and returning a new structure. In contrast to Standard ML, L allows for functors to be part of structures.

Laziness in the Module Language. Similarly to how the core syntax `lazy` e denotes lazy evaluation of an expression e, the module language has syntax for lazy functor application: given a functor F and a structure X, `lazy` $F\ X$ returns a lazy structure that will be evaluated only once one of its items is requested.

2.3 Packages

The ultimate goal of L is to support dynamic exchange of data in connected systems. This requires, by nature, dynamic typing, since the underlying media (disk files and network channels) are untyped[1]. This is why L has support for *packages* (also known as *dynamics*).

A package is a pair of a structure and (the run-time representation of) its signature. Packages are created and examined by special language constructs. A **pack** expression takes a structure and a signature, and returns a package as an instance of the abstract type `package`:

pack ⟨*structure expression*⟩ **:** ⟨*signature expression*⟩

Correspondingly, the **unpack** structure expression allows to access a packaged structure:

unpack ⟨*expression*⟩ **:** ⟨*signature expression*⟩

`unpack` takes a core expression, which must evaluate to a `package`, and a signature. It checks whether the given signature matches the signature obtained from the package, and if so, extracts and returns the structure. If the signatures do not match, a `Mismatch` exception is raised.

[1]To be exact, both are singly-typed—as byte arrays and byte vectors, respectively.

2.4 Examples

This section illustrates some aspects of how to program in L.[2] The examples consist of several implementations of non-thread reentrant locks.

A first possible signature is as follows:

```
signature LOCK =
sig
    type t
    val lock : unit -> t
    val sync : t -> ('a -> 'b) -> 'a -> 'b
end
```

Type t is an abstract type. `lock` creates a new lock that is initially available. `sync` takes a lock l, a function f, and an argument x. It acquires the lock l (waiting until it becomes free), applies $f\ x$ and frees the lock, even if f raises an exception. If f does not raise an exception, the result of $f\ x$ is returned as the result of `sync`.

The first implementation of this signature represents locks by a reference cell containing a boolean, which is `true` if and only if the lock is free. If the lock is not free, it performs a busy waiting loop.

```
structure BusyWaitingLock :> LOCK =
struct
    type t = bool ref
    fun lock () = ref true
    fun sync l f x =
        if exchange (l, false) then
            (* lock was free and is now acquired *)
            (f x handle e =>
                     (l := true; raise e)
             before l := true)
        else sync l f x
end
```

The second implementation of this signature improves on the first by using promises and futures to block the thread (obviating the need for busy waiting) until the lock becomes free:

```
structure Lock :> LOCK =
struct
    type t = unit ref
    fun lock () = ref ()
    fun sync l f x =
        let
```

[2]Note that additionally, many of the examples from the Alice Tour [Ros04] are valid for L as well.

```
                val p = promise ()
            in
                await (exchange (l, future p));
                f x handle e =>
                        (fulfill (p, ()); raise e)
                before fulfill (p, ())
            end
    end
```

Note that type t is now a reference cell storing the empty record, instead of a boolean value. This type would be completely uninteresting in Standard ML (if one ignores the fact that reference cells provide for token equality), but in L, this reference cell is actually used to store one of *two* values: either the empty record, or a promised future.

A different signature is conceivable for locks that act as a container protected by a lock, whose contents can only be accessed after acquiring the lock:

```
signature LOCKER =
sig
    type 'a t
    val lock : 'a -> 'a t
    val sync : 'a t -> ('a -> 'a * 'b) -> 'b
end
```

The abstract type 'a t represents a locked container, whose contents is of type 'a. lock constructs a container given an initial contents. sync, applied to a locked container l and a function f, acquires the lock, after blocking until it is free, and applies f to the contents x. If f raises an exception, the lock is freed and the contents is not modified (except for side-effects f may have had on x). Otherwise, f returns a new contents x' and a result y. The contents is set to x', the lock is freed, and y is returned as the result of the application of sync.

The implementation looks similar to Lock, but introduces an extra constructor C instead of using the contents itself as a synchronization value. Waiting for the lock to be free is implicitly performed by the **val** C x =... construct. If the contents x itself was used as synchronization value, acquiring the lock would cause x to be requested, with unexpected semantics if x happened to be a future. Using C x as the synchronization value circumvents this problem.

```
structure Locker :> LOCKER =
struct
    datatype 'a contents = C of 'a
    type 'a t = 'a contents ref
    fun lock x = ref (C x)
    fun sync l f =
        let
```

```
                val p = promise ()
                val C x = exchange (l, future p)
                val (x', result) =
                    f x handle e =>
                            (fulfill (p, C x); raise e)
            in
                fulfill (p, C x'); result
            end
    end
```

Chapter 3

The Base Runtime System

This chapter introduces a runtime system for executing L programs. This serves both to define terminology and to provide a basis for subsequent chapters to extend. As presented, the runtime system consists of well-known technology only. Novel features are provided only by the extensions.

Overview. The runtime system for L is based on a *virtual machine*. A virtual machine defines a computing platform for the execution of compiled programs. It abstracts from the underlying hardware and operating system to provide a standard interface. The following sections describe the four main constituents of a virtual machine, namely an abstract memory model (Section 3.1), an execution unit (Section 3.2), an input/output unit (Section 3.3), and a model for synchronization of concurrent threads (Section 3.4). These constituents are depicted in Figure 3.1.

The chapter concludes with how programs are compiled and how to run

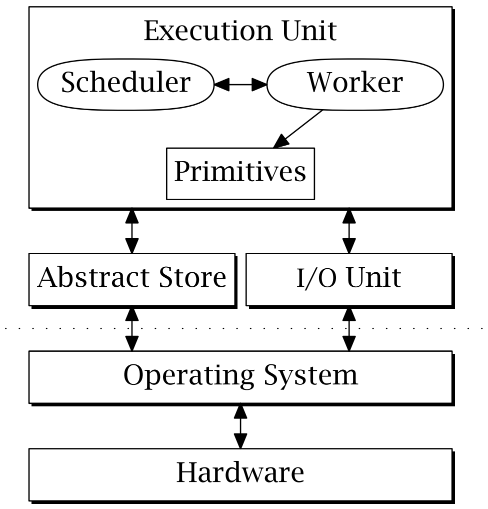


Figure 3.1: Structure of the Closed Programming System.

them with the virtual machine (Section 3.5).

3.1 Memory Model

Conventional hardware computer memory represents an unstructured array of binary words, and not even distinguishes between addresses and integer data. In contrast, the virtual machine's memory model manages semi-structured data in the form of *data graphs* in what is called the *abstract store*.

Data Graphs. Semi-structured data is arranged as a graph of nodes, called a *data graph*. Every node is annotated with a symbolic label, represented as an integer. A node is either a value node, whose value is a sequence of integers, or a compound node with a finite number of ordered directed edges to other nodes. Data graphs explicitly represent sharing (multiple edges to a single node) and can contain cycles. Tack [Tac03] provides design and implementation considerations for data graphs.

Data graphs abstract both from the underlying hardware and from the data structures of a particular programming language. This fact makes data graphs an ideal concept on which to base this work, to make it generally applicable. High-level programming languages, too, define data graph representations for their data structures. In the case of the programming language Oz, for instance, there are two distinct definitions, a high-level one based on constraints [Smo95], and a lower-level one based on semi-structured data [Sch98]. Language-level data maps nicely to the constraint-based representation, which can easily be mapped to the semi-structured data representation—which in turn can be expressed as the data graphs introduced above.

Managing Data Graphs in the Abstract Store. The abstract store represents, at any point in time, a data graph, which evolves as execution of a program progresses. In this sense, a snapshot of the state of an abstract store is a data graph. A running program has an environment consisting of references to nodes in the abstract store. The set of these nodes is called the *root set*. Nodes which are no longer reachable from the root set can safely be removed from the graph. This operation is called *garbage collection*. The abstract store defines a simple interface for operating on nodes in the graph. More precisely, it provides operations for creating nodes, establishing and redirecting edges, querying whether a node is a leaf or a compound, accessing node values, following edges, and performing garbage collection (given some root set).

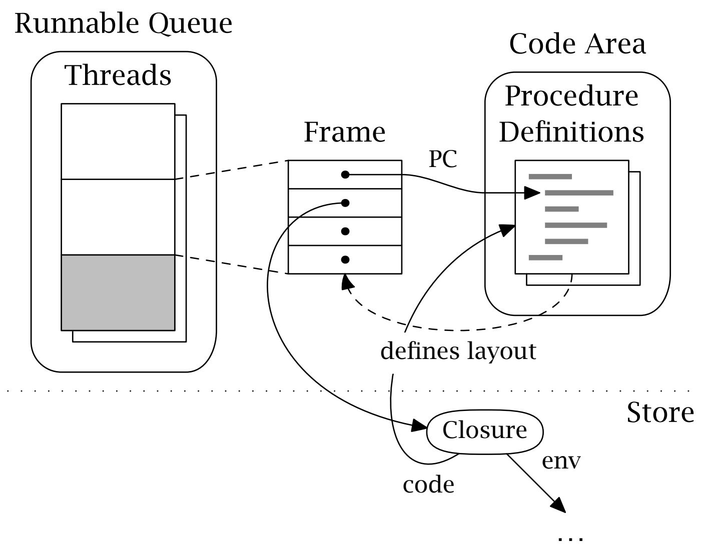


Figure 3.2: Data Structures for Execution.

3.2 Execution Model

The operational representation of computations and their execution in a virtual machine makes use of a number of concepts, summarized in this section. Computations are represented as *code*, describing data flow and control flow. Nonlocal control flow is enabled by application of *functions*, and by raising and catching *exceptions*. Computations are executed concurrently in a number of *threads*, managed by a *scheduler*. Figure 3.2 depicts the data structures used for execution that are described in the following.

Code and Code Execution. Code is stored as a sequence of instructions[1] in a static memory area called the *code area*, typically separate from the store. Code is executed by a number of concurrent *threads*. A thread maintains a stack of *activation records* (also known as *frames*), each storing an environment of references to store nodes, and a pointer to an instruction in the code area, the *program counter* (PC).

User-defined Functions vs. Primitives. New activation records are created by applying a *function*, which is the encapsulation unit for code. *First-class* functions are supported by *closure* nodes in the store. A closure holds a pair of the code and its environment. Besides user-defined functions, there is a set of functions called *primitives*. Primitives implement the built-in operations of the language that cannot directly be expressed by the code's instructions, such as operations on built-in data types and calls to the operating system.

[1]The specific representation of instructions, for instance as bytecode or native machine code, is of no concern here.

Exceptions. The virtual machine supports nonlocal control flow in the form of *exception handling*. Code can contain instructions to install an *exception handler*. Exception handlers are nested. Each exception handler is associated with the activation record current at the time it is created, and consists of a program counter pointing to its exception handling code. Primitives can *raise* exceptions. When an exception is raised, the worker discards activation records up to, but not including the activation record associated with the innermost exception handler. Execution of instructions is then resumed from the handler's program counter.

Thread Scheduling. Threads are managed by the *scheduler*. The scheduler maintains a pool of threads. When the scheduler becomes active, it fetches a thread from its pool and passes it to the *worker*. The worker obtains the program counter from the topmost activation record and starts to fetch instructions sequentially from the code area, to decode and execute them, which can cause activation records to be created (pushed) or discarded (popped), or threads to be created and entered into the scheduler's thread pool.

Status Register. Once a *preemption condition* is signaled, the worker must return control to the scheduler. The worker periodically checks for preemption conditions by testing the *status register*, which is a vector of flags. When a processing unit of the virtual machine needs processing time, it sets its associated flag in the status register. Each time the scheduler re-obtains control from the worker, it dispatches to the querying unit(s). For example, the store can signal that it needs a garbage collection lest available memory not become low; this is called *synchronous garbage collection* (as opposed to *asynchronous garbage collection*) and simplifies implementation as the root set needs only to be known at synchronization points defined by the worker. Another flag in the status register is periodically set by a *timer* to implement fair preemptive scheduling—thread execution is interrupted after a given *time slice*.

3.3 Input/Output Model

The input/output subsystem provides for interaction with files, pipes, and sockets. Reading from and writing to pipes and sockets may not always be performed immediately due to data or reader unavailability. Typically, operating systems provide for *non-blocking* input/output operations: Instead of waiting until a read or write operation can actually be performed, a process can inquire whether such an operation would immediately succeed. Concurrent virtual machines make use of this facility to avoid blocking the whole virtual machine just because a single thread has to wait for input/output: During this time, other runnable threads can be executed. The scheduler periodically polls pipes and sockets that threads block on, to see whether they are available and to eventually make threads runnable again.

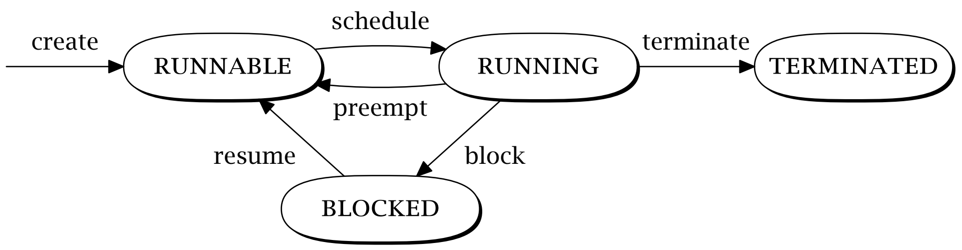


Figure 3.3: Thread States and Transitions.

Thread States. At this point, all states a thread can be in have been discussed. They are depicted in Figure 3.3: A thread is *runnable* if it resides in the scheduler's thread pool. The thread that is currently being executed by the worker is *running*. A thread that is waiting until an input/output operation can be performed is *blocked*. When the worker has popped the last activation record from a thread's stack, the thread is *terminated*.

3.4 Concurrent Synchronization

Concurrent threads need to communicate and synchronize. All threads access the same store and thus can trivially share data. For the purpose of synchronization, the store has a built-in notion of *future nodes*. An implementation of future nodes has been described by Mehl [Meh99].

Future nodes are placeholders for not-yet-computed values. To *eliminate* a future node means to replace it by its value. Conceptually, the future node is removed from the graph; in the implementation, the future node becomes transparent by having it point to the value it has been replaced with: All operations on a future node are instead performed on the value it has been replaced with.

Types of Futures. There are three kinds of future nodes. (1) A *promised future node* carries a queue of threads. When a thread requires the value of a promised future node, the worker enqueues the thread and returns control to the scheduler. A special primitive exists to explicitly eliminate a promised future node; in this case, the enqueued threads are appended (for fairness) to the scheduler's thread pool. (2) A *failed value* represents an unusable value and contains an exception. When a thread requires the value of a failed value, the contained exception is immediately raised in that thread. (3) A *lazy future node* holds a reference to a closure. When a thread requires the value of a lazy future node, the lazy future node becomes a promised future node whose queue contains the current thread, and a new thread is created with two activation records. The topmost activation record runs the closure, while the deeper activation record eliminates the new promised future node. On success, the promised future node is replaced by the value returned by the closure; on an exception, the promised future node is replaced by a failed value.

3.5 Compiling and Running Code

The virtual machine cannot execute L programs directly. Instead, an *offline compiler* translates L programs into an untyped intermediate language, called L_I in the following. Compilation proceeds as follows:

- Type definitions are translated into value declarations that bind type identifiers to term representations of the type. This makes it possible to perform dynamic type-checks requires by packages.
- Other than that, types are discarded after elaboration. Like Standard ML, L has the property that (apart from creation and inspection of packages) types are unnecessary for execution; this is also called *type erasure semantics.*
- Structures are translated to records: Identifiers bound in structures become record labels, and the value each identifier is bound to becomes the value of the record field.
- Signature coercion prunes record fields that are not part of the signature.[2]
- Functor definitions are translated to function definitions. Functor argument structures become function argument records, and the structure computed by the functor becomes the return record value of the function.
- Records are translated to tuples: First, records are normalized by sorting fields according to a lexicographic ordering of the labels, then labels are discarded. This is possible because for every record selection, the set of labels of the selected record is fully statically known. Thus, the field can be selected by its integer index.
- Function definitions are translated to an instruction sequence that creates closure nodes.
- Tuples and vectors are represented as compound nodes in the abstract store. Constructed values are represented as pairs of the constructor index and the constructor argument.

L_I is very similar to the intermediate language defined for Alice [Ros02]. Compilation of L into L_I is very similar to the usual operation of compilers of Standard ML.

[2]Note that functor arguments and result structures are also subject to signature coercion. In other words, signature coercion performs a recursive representation change.

Execution. A compiled L_I program can be placed in a file that consists of an instruction stream, in which references to primitives are symbolic. The virtual machine is started with an argument giving the name of a file containing a compiled program. The code area is initialized with the instructions from the file, resolving references to primitives. The virtual machine creates a single thread with a single activation record, whose program counter is the address of the first instruction in the code area. The thread is added to the thread pool and control is delegated to the scheduler, thereby executing the program.

Termination. The virtual machine terminates in either of the following cases:

- An exception is raised in a thread that has no exception handler installed. The virtual machine reports the exception and terminates with an error code.
- No thread is runnable and no thread is blocked on input or output. (Note that threads blocked on futures are not counted.) In this case, execution is stuck: nothing could make a thread runnable again. The virtual machine terminates with a success code.

Chapter 4

The Design Space of Pickling

Pickling is an operation that takes a root node of a data graph (as defined in Section 3.1) as argument, and produces a linearized representation from it, called a *pickle*, which is just a byte sequence. *Unpickling* is the converse operation: From a pickle, unpickling recreates in an abstract store (possibly in a different process, on a different machine, on a different computing platform) a clone of the original data graph. A *clone* of a data graph G is a data graph G' such that G and G' are isomorphic and have no nodes in common.

Pickling enables a computation process to effortlessly store a clone of a live data structure out-of-process, or to transport it to a computation in another process. A pickle both describes a value and specifies its type. This means that pickles allow to define file formats and messages in a communication protocol by programming language types. If pickles can contain first-class functions, then even mobile code and dynamic libraries can be stored as pickles. This is heavily made use of by component deployment in Chapter 10 and distribution in Chapter 11.

Goals. This chapter spans the design space for pickling and derives a novel classification for pickling mechanisms. Ultimately, the goal is to design a pickling mechanism suitable as a foundation for open programming in L, with the following properties:

A Principled Approach. The literature widely seems to regard pickling as a simple mechanism. A closer look, however, reveals a number of interesting issues, such as bottom-up versus top-down pickling and unpickling. This work takes a principled approach to identify and discuss these issues.

Platform-independence. If general data exchange is to be based on pickling, then pickles need to be platform-independent. In an abstract store, in contrast, data representation emphasizes efficiency. This leads to the distinction between the *external* and *internal* representations of data structures, and raises the question of how to convert between them.

Resource Access Security. Data structures in the store can contain references to resources pertinent to the current process. This raises two issues: Resource references could possibly not be interpreted meaningfully in other processes; and implicit creation of resource references through unpickling may introduce security problems. Resources need to be recognized and disallowed in pickles.

Pickling of Code. L is a higher-order programming language, which allow functions as first-class values in data structures. A natural consequence is that pickles need to be able to describe data structures with first-class functions, including their code (the function definition itself, a description of the computation).

Support for Futures. L programs compute with futures (see Sections 2.1.5 and 2.1.6). As described in Section 3.4, futures are represented in the store by special types of nodes. The pickler must be able to cope with these nodes.

Concurrent Pickling. Concurrent systems with pickling present the potential pitfall that pickling may be traversing a data structure that is being mutated. To avoid producing unusable pickles, a requirement is that a pickle must always correspond to a consistent snapshot of the store at some point in time.

This work is the first to address all of these problems.

Overview. Many programming systems offer some form of pickling, which also goes by the names *linearization*, *serialization*, *marshaling*, and *flattening*. Section 4.1 summarizes related work with respect to the presented goals. The remainder of the chapter derives the possible design choices that can be made when designing a new pickling mechanism, resulting in a novel classification for pickling mechanisms. Section 4.2 presents an overview of the classification. The subsequent sections then discuss the classification's dimensions in detail. This culminates in founded design decisions, presented in Section 4.11, for a pickling mechanism for L that is suited to fulfill the above goals.

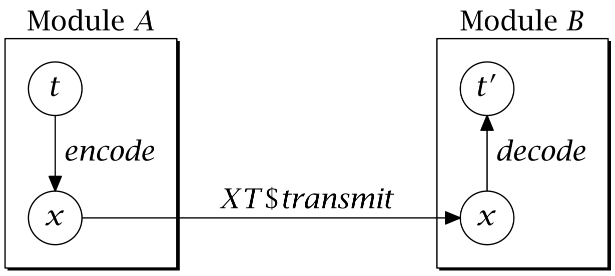
4.1 Related Work

This section reviews existing approaches to pickling. This section starts out with a presentation of the first systems that feature pickling, roughly in chronological order. The first approach to integrate pickling into programming systems [HL82], developed in the context of CLU, already contains most of the essential ideas. Subsequent publications on pickling contribute comparatively small insights on pickling. Therefore, after the pioneering approaches, only the ideas subsequent approaches contribute are discussed.

The term *pickling* was coined by Birrell et al. [BJW87] in the context of efficient small databases.

4.1.1 Value Transmission in CLU

Herlihy and Liskov were the first to publish a pickling mechanism for a programming language [HL82]. The context of their work is CLU [LZ74], a procedural language derived from Pascal that emphasizes the definition of abstract data types. They define a *transmissibility* property on types: All the primitive types are transmissible, and abstract data types are transmissible if they define operations for converting between the internal and external representations of values of the type as follows:



Converting a value of an abstract data type to an external representation means reducing it to a representation using only transmissible types. This representation will be marked as being an instance of a specific abstract data type in the external representation. Therewith Herlihy and Liskov propose the first customization mechanism for pickling, and hint at correctness criteria for pairs of externalization and internalization functions. Externalization functions explicitly decide which values shall be tested for sharing. The linearized form they describe amounts to a top-down description of the data structure.

4.1.2 Modula-3 Pickling

Modula-3 [DEC99], an object-oriented language descended from Modula, features a pickling mechanism [BNOW95] clearly inspired from CLU's, but retargeted to an object-oriented language. Its design aims for simplicity. Sharing preservation and cyclic data structures are supported. Source documentation [DEC94] claims that pickles are platform-independent, but this does not mean that a pickle can be read back by any Modula-3 application: Pickles are parametric in the bit sizes of primitives types, which are not prescribed by the pickle format. An attempt to read a pickle into a process whose ordinal types have a smaller size than that required by the pickle causes an exception.

The behavior of pickling can be influenced in several ways. By default, all data structures (in particular including object state) are cloned. Values of *untraced references*, pointers to data allocated outside the Modula-3 heap, are not included in pickles; they receive the value **nil** upon unpickling. The pickling and unpickling classes allow their methods for reading and writing objects to be overridden. Additionally, handlers may be registered for specific types. While the CLU approach completely separates the issues of customization and pickle format, Modula-3 entangles the two.

Types are represented in the pickle as *fingerprints*, that is, hash values computed over the type definition. The language's definition of type equivalence carries over to fingerprints. Types of unpickled values must be defined in the running system in order for unpickling to succeed.

Procedures are stored in pickles as pairs of name and type. In other words, they are pickled by reference and not by value. Upon unpickling, a separate mechanism is used to locate the implementations of referenced procedures, via the given name and type.

According to Nelson [Nel91, Page 80], who describes an older version of Modula-3's pickling module, the pickler performs a depth-first search to collect the information to store in a pickle. Pickles as described by Nelson depend on the machine architecture and use the same layout as the garbage collector. Therefore, unpickling amounts to reading the pickle as an image and adjusting the pointers contained therein.

4.1.3 Java Object Serialization

Java [GJS00] is an object-oriented language inspired from C++ [Str00], but fully type-safe and with automatic memory management. Java's Object Serialization [RWW96, Sun01] is a direct descendant of Modula-3 pickling and adds only few ideas to it. In particular, its designers valued type-safety more than efficiency, making use of the fact that store nodes are fully self-describing. This resulted in a body of work trying to improve efficiency [PH00, BP01]. Java pickling is implemented in Java, with unsafe reflection primitives to allow access to private object fields and methods.

Java separates serializing of object state (pickling) from the transmission of stateless computations (class files): to interpret a given pickle, the virtual machine has to obtain the referenced classes out of band.

4.1.4 .NET Serialization

A similar approach to Java's object serialization is taken in Microsoft's .NET Framework [Mic03a]. The .NET runtime, called the Common Language Infrastructure [TC301b], is not tied to a single language, but commits to an abstract store that represents objects as instances of fully-known classes.

.NET's pickling [Mic03d] is superior to Java's in that it factors customization and pickle format, up to providing a set of pickle formats (binary, XML, and SOAP [GHM+03]) and allowing new formats to be defined by users. Customization methods for pickling return a description of the object they are called on as semi-structured data (as a pair of a class name and a set of name-value-pairs). Unfortunately, the pickling and unpickling processes are not specified in detail; for instance, it is unclear in what order user-defined delayed object initialization callbacks are run after unpickling.

Pickling and unpickling are, like in Java, implemented in a high-level language (in this case, C# [TC301a]), and use reflection to access private object state. Language-level security in the sense of maintaining encapsulation of private state is achieved by introducing an explicit serialization permission: Code that calls a pickling or unpickling operation must have the serialization permission.

4.1.5 Standard ML of New Jersey

Standard ML of New Jersey (SML/NJ) [SML02] is a native-code implementation of the non-pure functional language Standard ML [MTHM97]. Its structure `Unsafe` provides operations `blastWrite` and `blastRead` that respectively perform pickling and unpickling, with next to no documentation. As the structure name suggests, these operations are unsafe in that they do not perform type-checking upon unpickling: behavior is undefined if the types do not match. The operations clone state and maintain sharing and cycles. An attempt to pickle a first-class function or a resource (such as a reference to an open file) leads to termination of the process.

Pickling has originally been added to SML/NJ for the sake of storing static environments in compilation images [AM94, Section 4]. From the start, pickling was implemented on top of the garbage collector in order to reuse existing traversal algorithms. As a consequence, nodes are enumerated in breadth-first order. The idea is to start a garbage collection with a root set consisting only of the root node, copying the data structure in a fresh memory area (or semi-space), adjusting pointers to be relative offsets to the start of the memory area, and writing out the memory area to disk.

4.1.6 Objective Caml

The functional and object-oriented language Objective Caml [INR02a] (also known as O'Caml) from the ML family of languages comes with a pickling mechanism in its standard library [INR02c], called module `Marshal`. The only source of information about its properties are the sparse online documentation and the freely available source. The pickler supports all basic and constructed types except for objects, which cannot be pickled. Code is pickled as a pointer into the address space of the pickling process. This means that pickling of code is not platform-independent; in fact, it does not even allow for recompilation on the same platform. Code consistency is checked by means of an MD5 digest [Riv92] computed over the entire code area. Lazy suspensions are pickled by pickling the unevaluated expression itself. The pickler recognizes *abstract values* from their memory representation and disallows them in pickles. Abstract values are used for instance for handles to dynamically linked C primitives; in other words, pickling is well-defined in this case. Customization is only supported at a low level: types defined using the C foreign function interface carry user-defined pickling and unpickling operations. Values of a user-defined type are identified in pickles by their type name. The type of the pickled value is not represented in the pickle; unpickling is unsafe in that it just assumes that the types match.

4.1.7 G'Caml

G'Caml [INR02b] extends O'Caml with extensional polymorphism. Furuse and Weis [FW00] describes a pickling mechanism for G'Caml. The main concern is type safety of representation types: Types are reduced to MD5 fingerprints computed over the normalized type. Normalization means that sum and product types (that is, algebraic data types and records) are sorted by their labels, and the names and labels are removed. The intent is to be robust against modifications of type definitions. For instance, the following types are equivalent after normalization:

```
type 'a tree =                type 'a arbre =
| Leaf of 'a                  | Feuille of 'a
| Branch of 'a tree list      | Branche of 'a arbre list
```

Type abstraction is not supported. Type fingerprints are used exclusively to verify upon unpickling that the pickled value has the same representation as the type used to read it back. Type fingerprint and value are stored sequentially in pickles; their consistency is assumed. Pickling of functions is not supported. The pickle format is not described, but an example [FW00] suggests that it is based on a depth-first search with a preorder enumeration of nodes.

4.1.8 Python Pickling and Marshaling

The object-oriented scripting language Python [Pyt05] has three picklers, called `pickle`, `cPickle`, and `marshal`, unpublished but for the online documentation and freely available source. `pickle` and `cPickle` use the same pickle format, support sharing detection and cyclic data structures, and pickle code by reference as pairs of module name and symbolic class name. `pickle` and `cPickle` differ in that `pickle` is implemented in Python and is subclassable to allow for customization, and `cPickle` is implemented in C (hence the name) and does not support customization, but is claimed to be three orders of magnitude faster. These picklers are the only picklers known to be based upon a depth-first-search with postorder enumeration of nodes. Pickles are bytecoded programs and unpickling amounts to interpreting the program.

The `marshal` pickler is completely separate from the first two. It is used to pickle compiled Python programs, which are represented as Python objects. `marshal` does not maintain sharing nor support cyclic data structures as these do not occur in Python code. The pickles produced by `marshal` are not portable (that is, platform-independent). Different implementations of Python have different code representations.

4.1.9 Clean Dynamic Input/Output

Clean [Sof04] is a lazy purely functional programming language with dynamics. A dynamic is a pair of a value and its type, and can be deconstructed by a **`typecase`** construct. Clean features file input/output operations for dynamics [VP02]. Any dynamic can be written to a file, including higher-order functions. Lazy expressions are stored as unevaluated expressions. Resources cannot be pickled since they are based on *unique types*, and unique types are disallowed in dynamics.

Type definitions are not stored in the file with the dynamic, but are stored as a reference into a global repository. An early account mentions that higher-order pickling was planned for Clean [Pil96], but this was later solved by representing function definitions as references into the repository [VP02]: The repository contains native code images, so dynamics on files are not platform-independent. Repositories are managed using special commands that exist outside the programming language.

Loading of dynamics is lazy, and is separated into on-demand loading of the type (for matching in the **`typecase`** construct) and the value, given that the types match. Nested dynamics in dynamics that already reside on files are stored as references to the other files, which greatly increases complexity of the design and implementation. The paper does not give the rationale for this approach, and it remains unclear what practical problems it is supposed to solve.

4.2 Overview of the Classification

The discussion of related work in the previous section alludes to the broad design space that exists for pickling. Understanding the design space is an essential precondition for founded design decisions when introducing pickling into a programming language and system. This section therefore tries

- Dimensions
  - High Level: Design
    - Types
      - Used by the Mechanism
      - Used for Type-Checking
    - Portability and Platform-independence
    - Treatment of Specific Data Structures
    - Customization
  - Low Level: Implementation
    - Level of Implementation
    - Pickling and Unpickling Metaphors
    - Representing Sharing in Pickles

Figure 4.1: Dimensions of the Pickling Design Space.

to capture this design space in the form of a novel classification for pickling mechanisms. While the preceding presentation of related work is organized by system, this section takes the complementary view of examining features of pickling mechanisms and is structured by feature. The following shall present an overview of the dimensions the classification consists of, depicted in Figure 4.1.

Types Used by the Mechanism (Section 4.3). This item is concerned with the type information that the mechanism uses for the directing traversal of a graph in the store, and for interpreting the contents of a pickle.

Types Used for Type-Checking (Section 4.4). Type-checking upon unpickling can verify that the pickle contents matches expectations.

Portability and Platform-independence (Section 4.5). This item describes to what degree pickles are portable. At the same time, this raises the question whether there the syntax and semantics of pickles are defined by an open specification.

Treatment of Non-term Data (Section 4.6). Nodes representing functions, resources, or threads have to be treated specially. They may or may not be picklable, and if they are, there are several possibilities for how they are pickled.

Customization (Section 4.7). The next question is whether and how the pickling process can be customized to implement user-defined behavior for specific data structures.

Level of Implementation (Section 4.8). One question to ask is at what level to implement the pickling operations themselves: either in the high-level language itself, or as a service provided at a lower level by the runtime.

Pickling and Unpickling Metaphors (Section 4.9). There is a choice of the mechanism to use to perform the traversal. This is similar to the choice of an algorithm for a garbage collector: It either operates as a depth-first search or as a breadth-first search.

Sharing in Pickles (Section 4.10). This concerns the representation, if any, of sharing and cyclic data structures in pickles.

Related work is classified in tabular fashion according to the above criteria. The tables also contain entries for Mozart and Alice, which form the context of this work. The pickling mechanism for L has the same properties as Alice, as will be discussed in Section 4.11.

4.3 Types Used by the Mechanism

Pickling has to interpret every node in the graph to be pickled, and has to be able to descend into child nodes. This requires some type information for each node. The first question therefore is how type information is made available.

Type-directed Pickling. One possibility is to define the pickling operation as a function of two arguments, a (pointer to a) node and (a representation of) its type. The node is interpreted according to the type, and the type is deconstructed in parallel with the node to recursively invoke the pickling operation with a constituent node and the corresponding constituent type. An approach similar to this is for instance taken by CLU [HL82].

Limitations. Several language features may make this approach difficult, impractical, or even impossible. First, in a language with polymorphism, the types given as arguments to the pickling operation must be instantiated types, that is, they must not reference any (free) type variables. One solution to this problem is to execute programs under *intensional polymorphism* [HM95], which means that instance types become explicit additional arguments to polymorphic functions. Another problem arises in languages with first-class functions: Nodes referenced through closures cannot be traversed, because the type of the closure itself cannot in general describe the types of the nodes in the closure. Yet another problem arises in languages with generative abstract types: Such types do not describe their implementation, thus cannot be used for traversal—instead, the corresponding representation type would be needed. Implementations can alleviate the latter problem by representing abstract types as pairs of the generated name and the (hidden) representation type.

Reflection-based Pickling. An alternative, used in most pickling implementations, is to define the pickling operation to have only the node as its argument. The node's type is discovered by inspecting the node using *reflection* primitives. In languages with automatic memory management, these primitives are typically easy to provide, as they return the same information as the garbage collector needs. The limitations of type-directed pickling therefore do not apply to reflection-based pickling.

Because the pickling and unpickling operations must be the inverse of one another (pickling an unpickled data structure should produce the same pickle, modulo renamings), all types represented in store nodes must be represented in the pickle and vice-versa. Besides making it possible to interpret the contents of a pickle and to reconstruct the value in an abstract store, types in pickles can serve other purposes:

Integrity Verification of the Pickle. The more type information nodes contain, the more invariants can be checked at run time. At one extreme, types may be used to verify the internal consistency of a pickle.

Debugging. It may be useful to represent more type information in the pickle for the purpose of debugging. In other words, more detailed type information can make a pickle easier to interpret by humans.

Type System Used. The type system used by the mechanism needs not be the type system used at the language level. Frequently, they are the same. For instance, in implementations of object-oriented languages with automatic memory management, every object carries a reference to the corresponding class definition, which fully describes the object's representation. This is the case for the Java Virtual Machine [LY99] and the object serialization implemented on top of it [RWW96, Sun01]. Also, dynamically-typed languages such as Oz have to represent full language types at run time, since they require run-time type checks.

Other systems represent just enough information in the store and in pickles to make it possible to implement services in the runtime, such as garbage collection. The graph in the store then needs not be related to the language-level types at all, only to their representation in the actual implementation. This kind of store is typically used in implementations of statically-typed functional languages such as those in the ML family, for instance in SML/NJ [SML02] or O'Caml [Ler90].

Classification. Table 4.1(a) classifies approaches by whether the traversal is type-directed or reflection-based, and whether the type system used by the mechanism is the language's or a lower-level type system.

4.4 Types Used for Type-Checking

After unpickling, the resulting data structure will be bound to a program variable. At this point, the type with which the variable is declared (the

Approach	**(a) Mechanism**		**(b) Type-checking**
	Traversal	**Node Types**	
CLU	type-directed	language	explicit/full
Modula-3	reflective	language	implicit
Obliq	reflective	language	dynamic types
Java	reflective	language	implicit
.NET SOAP	reflective	language	implicit
.NET Binary/XML	reflective	language	implicit
O'Caml	reflective	low-level	none
G'Caml	reflective	low-level	explicit/fingerprints
SML/NJ	reflective	low-level	none
Python `pickle`	reflective	language	dynamic types
Python `cPickle`	reflective	language	dynamic types
Python `marshal`	reflective	language	none
Clean	unspecified	language	dynamics/repository
Mozart	reflective	language	dynamic types
Alice	reflective	low-level	dynamics

Table 4.1: Classification of Pickling Mechanisms: Use of Types (a) by the mechanism, (b) for type-checking.

expected type) and the type of the data structure (the *actual* type) must match. There are several approaches to type-checking.

None. The weakest option is to omit type-checking entirely and assume that the types match. The burden of ensuring type consistency is left to the programmer. The pickling mechanism from O'Caml's standard library takes this approach, with undefined results if the types do not match.

Dynamic Types. In dynamically-typed languages, variables are not statically restricted to hold values of given types only. If a variable holds a value of the wrong type, this will sooner or later result in a run-time type error (typically an exception or program abort). Here, the burden of ensuring type consistency is also left to the programmer, but at least the system provides some help. Python takes this approach.

Explicit Types. If types have a run-time representation (for example, as terms or fingerprints), the actual type can be explicitly stored in the pickle—in other words, the pickled value is, at least conceptually, a pair of a value and its type. The unpickling operation takes the expected type as an additional argument and performs type-checking prior to allowing access to the value. An example for this is G'Caml.

Dynamics. Some statically-typed languages provide pairs of a value and its type as first-class values, called *dynamics* [ACPP91, ACPR95]. Inspection of a dynamic is performed using a **`typecase`** construct. In other words, type-checking is decoupled from unpickling. Dynamic file I/O in Clean takes this approach, by only allowing values of type dynamic to be pickled.

Implicit Types. The information present in pickle node types may be made expressive enough to be sufficient for language-level type-checking. Types can then be checked not only for the root node of the pickled value, but for all its constituents. This is the case for Java Object Serialization, which needs to perform subtyping tests for assigning values to object fields, since the class definitions used for pickling need not be the same as those present during unpickling. (Note that with respect to the root node, Java uses dynamics: The unpickled value has the type `java.lang.Object`, which is the top type of all objects. Before such an object can actually be used, it must be cast down. Down-casts are checked, that is, they raise an exception if types are incompatible.)

Representing Types. In the case of explicit types or dynamics, the type stored with the value can either be physically part of the pickle or it can be a symbolic name resolved by means of a global type definition repository. The actual representation of the type can either be a full representation of a type, or it can be a fingerprint. In either case, the type can either be the language type including eventual type abstractions or it can be some representation type.

Table 4.1(b) classifies how existing approaches perform type-checking.

4.5 Portability and Platform-independence

Many approaches in the literature claim to produce *portable* or *platform-independent* pickles. Different authors however use different underlying definitions for platform-independence. One can identify at least the following levels of platform-independence, in increasing order:

None. At the lowest level, pickles are readable only by processes executing the same compilation image as the process that produced them. This is, for example, the case for the pickler in the O'Caml standard library, which includes addresses in the pickle that are valid for the compilation image only (see the discussion on functions).

Cross-image. At this level, pickles abstract from the actual compilation image used, but contain hardware or operating system-specific data (such as exact sizes of integers). Such pickles lose validity when read on systems with different hardware or operating system properties. This can be relaxed somewhat:

Cross-architecture. In addition, pickles at this level abstract from hardware and operating system specifics. They remain tied to a specific implementation of the programming system however. For example, they may make assumptions on the layout of data structures that could be different on other implementations (such as byte order or representations of floating point numbers).

Some systems make some effort to achieve cross-platform portability, with semantically observable limitations. For instance, Modula-3 includes the assumed integer sizes in the pickle's header and allows them to be read on systems with larger integer sizes. Python reads back 64-bit integers as standard integer objects on 64-bit architectures, but types them as long integers (as opposed to standard integers) on 32-bit architectures.

Cross-implementation. Here, the specification of the pickling mechanism includes the full pickle format and semantics, so that other implementations can actually be developed. Of course, pickles must contain sufficient information so they can be interpreted independently of implementation considerations.

Java Object Serialization [Sun01] and XDR [Sun87] attain this level. The .NET/XML and .NET/Binary picklers do not fall into this category because of the lack of an open specification.

Cross-version. At the highest level, pickles declare the version of the specification they adhere to. Implementations may be able to read pickles conforming to several versions of the specification for backward compatibility. This is the case for .NET/SOAP.

An orthogonal issue in portability is whether pickles are self-contained.

Self-contained Pickles. Moving the pickle amounts to a simple data transfer (file copy or download) to another system, where it can be interpreted.

Global Repository. Pickles can only be interpreted with respect to a given environment. This environment is typically stored in a global repository. Clean for example looks up type and function definitions in a repository. Moving a pickle to a different system requires to move (parts of) the environment with it and integrate it with the environment of the target system. Clean provides special commands external to the programming language to manage pickles and repositories.

Classification. Table 4.2 summarizes the levels of portability provided by existing approaches, and whether they provide an open specification or not.

4.6 Treatment of Non-term Data

Data structures representing control structures or operating system resources typically cannot be pickled by simple cloning. This section examines design possibilities for functions (also called procedures, depending

Approach	**Portability**	**Open Specification**
CLU	unspecified	no
Modula-3	cross-architecture	no
Obliq	unspecified	no
Java	cross-version	yes
.NET SOAP	cross-version	yes
.NET Binary/XML	cross-architecture	no
O'Caml	none	no
G'Caml	unspecified	no
SML/NJ	unspecified	no
Python `pickle`	cross-version	no
Python `cPickle`	cross-version	no
Python `marshal`	cross-architecture	no
Clean	cross-image	no
Mozart	cross-architecture	no
Alice	cross-version	planned

Table 4.2: Classification of Pickling Mechanisms.

on the language), resources, and threads. Table 4.3 summarizes the design decisions taken by existing approaches.

4.6.1 Functions

Most language support some form of first-class function, be it through full-fledged closures, delegates (pairs of object and method pointer), or function pointers. Because of the wide range of representations of functions and function definitions (called a function's *code*), a number of possible design choices with respect to pickling have to be considered:

Disallow. The simplest, but least expressive solution is to disallow functions in pickles, in which case the representation of code becomes irrelevant. This could be enforced statically by the type system. For instance, Standard ML's type system provides for *equality types*, which do not contain the function types, such that the pickling operation could require its argument to be of an equality type. Alternatively, it could be enforced dynamically by having the pickler recognize and reject nodes that represent functions.

Pickle by Reference. One could allow references to functions to be contained in pickles, but not their code. The function would be represented as a symbolic reference (which Java does) or an address in the code address space (as is done in O'Caml). The unpickler would be required to resolve the reference, either by requiring a function with the same symbolic name to be present in the system, or by dynamically locating and linking a component that defines it. O'Caml makes its symbolic representation safe by including an MD5 digest of the code of *all* functions in the system.

Approach	Procedures	Resources	Threads
CLU	?	disallowed	N/A
Modula-3	by-reference	disallowed	disallowed
Obliq	by-value	disallowed	disallowed
Java	by-reference	disallowed	disallowed
.NET SOAP	disallowed	disallowed	disallowed
.NET Binary/XML	by-reference	disallowed	disallowed
O'Caml	by-reference	disallowed	disallowed
G'Caml	disallowed	unspecified	unspecified
SML/NJ	disallowed	disallowed	disallowed
Python `pickle`	disallowed	disallowed	disallowed
Python `cPickle`	disallowed	disallowed	disallowed
Python `marshal`	by-value	disallowed	disallowed
Clean	via repository	disallowed	N/A
Mozart	by-value	disallowed	disallowed
Alice	by-value	disallowed	disallowed

Table 4.3: Classification of Pickling Mechanisms: Supported Data Structures.

Pickle by Value. Another option is to include in the pickle the function complete with its code. Obliq [Car95] stores functions in source form; other options are abstract syntax trees, compiled bytecode, or compiled native code.

Link to a Code Repository. Procedures can also be translated to references into a global code repository upon pickling. This is Clean's approach. The unpickling operation is able to dynamically load, if needed, the code from the repository to resolve the reference.

4.6.2 Resources

Values representing resources of the current process lose their meaning when interpreted in other processes. For this reason, it makes no sense in general to include them in pickles. Resources include, for instance, handles to open files or graphical windows, pointers to C functions obtained through the foreign function interface, or local runtime structures such as threads. There are several approaches to deal with this problem:

Undefined Behavior. The weakest option is not to handle resources specially at all. Unpickling a value representing a resource then has undefined behavior. This happens, for example, when loading a heap image in SML/NJ.

Disallow. Values representing resources can be specially marked. If the pickler encounters a resource, it fails (for example, with an exception). This approach is taken, for example, by O'Caml for foreign functions. Java provides for a language mechanism to declare data structures serializable on a per-class basis, and the class library uses this mechanism to make all classes representing resources non-picklable.

Re-binding. Some values represent resources that exist in every process. A mechanism called *re-binding* replaces such resources upon unpickling by the local resource of the unpickling process [FPV98].

Using the Type System. At the language level, an effect system or a monadic type system can be used to determine statically whether a value may contain a resource. The pickling operation would restrict its argument to be of resource-free type. Clean takes this approach by means of its *unique types*.

4.6.3 Threads

Some research has gone into allowing to pickle thread state or first-class threads [BH00, FPV98]. Unpickling a thread causes creation of a new thread in the unpickling system, whose state is a clone of the original thread. Applications include checkpointing, server replication, and thread mobility.

4.7 Customization

Customization is the user-level ability to influence pickling behavior of specific values. Several levels of support can be identified:

None. It is simplest to not provide any customization mechanism at all. Implementors of abstract data types are then required to choose a representation that implicitly produces the required pickling behavior. Section 6.4 presents ways to deal with the issue.

At the Level of Extensions. Many systems allow users to define their own low-level data types (typically called *extensions*) using a foreign function interface. O'Caml, for instance, allows users to supply pickling and unpickling operations for extensions implemented in C.

At the Language Level. The most expressive is to enable customization at the language level. In CLU, implementors of abstract data types can define operations `encode` and `decode` that perform translations between internal and external representations. Java and .NET programmers can choose to disallow serialization of their classes entirely, they can specify that specific fields of the object state are left out upon pickling, and they can supply replacement objects during pickling (that even may be of a type unrelated to the original object) that are translated back during unpickling. Of course, the latter possibility implies that type compatibility must be explicitly checked again at unpickling time.

Approach	**(a) Customization**	**(b) Impl. Level**
CLU	high-level	language/safe
Modula-3	high-level	language/unsafe
Obliq	none	runtime service
Java	high-level	language/unsafe
.NET SOAP	high-level	language/unsafe
.NET Binary/XML	high-level	language/unsafe
O'Caml	low-level	runtime service
G'Caml	none	runtime service
SML/NJ	none	runtime service
Python `pickle`	high-level	language/safe
Python `cPickle`	none	runtime service
Python `marshal`	none	runtime service
Clean	none	unspecified
Mozart	low-level	low-level
Alice	low-level	low-level

Table 4.4: Classification of Pickling Mechanisms: (a) Customization and (b) Level of Implementation.

Separation of Customization and Pickle Syntax. It is possible either to couple user-defined translation functions with pickle syntax or to decouple them. Modula-3 and Java couple the two: translation functions convert between the internal representation and a byte sequence. In contrast, the two issues are separated in CLU and in .NET. In CLU, translation functions are free to choose any external representation, defined in terms of language data structures (that are more primitive than the type to translate). .NET decouples the translation from the low-level pickling format by defining external representations to be pairs of a type name and a finite mapping from names to values. User-defined classes can specify the translation between their state and this type of external representation, which is then taken to actual pickle syntax by a class-library provided lower layer (which, by the way, can also be replaced by the user).

Modal Pickling. As an extension, customization can be made to depend upon the pickling context. This is called *modal pickling*. Java, for instance, builds its Remote Method Invocation [WRW96] mechanism on pickling. Object references may however differ in their behavior depending upon whether they are serialized to a file or whether they are transferred as an argument to a remote method. One special application of this is to omit type information from pickles representing remote arguments for efficiency [BP01]. .NET adopts the general solution to allow a *context* parameter to be given to the pickling operation, which is then passed to user-defined translation functions.

Table 4.4(a) summarizes customization facilities as offered by existing approaches.

4.8 Level of Implementation

A fundamental decision to make when implementing a pickling mechanism is at what level in the programming system it operates.

High-level Language Implementation. Pickling can be implemented in the high-level language itself. This is the case for Modula-3 [BNOW95], Java [RWW96], .NET [Mic03a], and Python [Pyt05]. This typically requires unsafe inspection and deconstruction primitives for pickling, and unsafe construction primitives for unpickling. A noteworthy exception is the dynamically-typed language Python, which defines everything, including the class of an object, as objects that can be inspected in a systematic way.

This approach is problematic in languages with concurrency when the pickler is defined to clone stateful data structures: With such a pickler, the snapshot property does not come "for free", meaning, without programmer intervention (see Section 6.1). Existing approaches all seem to disregard the issue, placing the burden on the programmer.

Virtual Machine Service Implementation. The alternative is to implement the pickler as a virtual machine service. The virtual machine obviously has access to all representation details of values. This approach is deemed to have better efficiency, as is illustrated by the fact that low-level pickling implementations have been proposed for Java [BP01] to improve its performance, or that Python has a module `cPickle` that is a low-level implementation (in C, hence the name) of the module `pickle`.

Table 4.4(b) gives an overview of what implementation level is chosen by which existing approach.

4.9 Pickling and Unpickling Metaphors

Two distinct traversal mechanisms are found in the literature.

Depth-first Search. Implementations of depth-first traversal require either recursion or an explicitly maintained stack. There are two possibilities to enumerate nodes: preorder enumeration, which is the approach taken by most existing mechanisms, or postorder enumeration, which only seems to be used in Python. Depth-first search is the mechanism used in most approaches.

Garbage-collector Based. Pickling can use standard garbage-collection algorithms, based on the observation that both perform a traversal of a graph. This is done, for example, in SML/NJ [AM94], where pickling amounts to a breadth-first search of the graph using Cheney's algorithm [Che70, Wil92]. In their case, pickling becomes iterative, with a constant-size runtime stack. In contrast, depth-first search needs a stack linear in (some measure of) the depth of the graph. Conceptually, unpickling amounts to loading a memory segment and patching pointers (although it may technically be realized differently), directly exploiting the fact that abstract stores internally also use a linearized representation for graphs.

Drawbacks. Using the garbage collector for pickling, or using the memory layout as pickle format makes design decisions of the implementation of the store shine through. Not abstracting from memory layout is likely to limit portability of pickles. In contrast, a graph-based traversal can be implemented on top of the interface of the abstract store, and allows for a layered design.

Dual to node enumeration order during pickling is what concept the unpickler builds on.

Term Interpretation. The pickle conceptually represents a term built from a set of constructors (plus some representation for sharing). Unpickling amounts to evaluating the term. This is the canonical approach if preorder enumeration is used for pickling.

Bytecode Evaluation. The pickle conceptually represents a bytecoded program, with concepts such as opcodes, operands, a stack, and local variable slots. Unpickling amounts to executing the program. This is the canonical approach if postorder enumeration is used for pickling.

Heap Loading. The pickle uses the same representation as objects in the heap. Unpickling amounts to loading the pickle contents into a fresh memory area and relocating the pointers it contains. This is the canonical approach if breadth-first enumeration is used for pickling, but is also chosen in an early version of Modula-3 [Nel91], which uses preorder enumeration for pickling.

The traversal mechanism and unpickling metaphor used by existing approaches are summarized in Table 4.5(a).

4.10 Representing Sharing in Pickles

It may be important to maintain sharing in pickles for several reasons. Most importantly, there is a semantic issue: If sharing is not maintained, this may become visible for nodes that have token equality. When not handling sharing, cyclic data structures cannot be pickled. Furthermore, there is the issue of efficiency: In the worst case, pickles can suffer an exponential growth.

This leads to the question of how to represent sharing in pickles. The usual way is to logically bind subgraphs to identifiers, and allow references to

Approach	**(a) Metaphors**		**(b) Sharing/Announced**
	Pickling	**Pickles**	
CLU	preorder	term	all nodes
Modula-3	preorder	heap/term	all nodes
Obliq	preorder	term	all nodes
Java	preorder	term	all nodes
.NET SOAP	preorder	term	all nodes
.NET Binary/XML	preorder	term	all nodes
O'Caml stdlib	preorder	term	all nodes
G'Caml Weis	preorder	term	all nodes
SML/NJ	breadth-first	heap	all nodes
Python `pickle`	postorder	bytecode	all nodes
Python `cPickle`	postorder	bytecode	all nodes
Python `marshal`	preorder	term	none
Clean	unspecified	unspecified	? + external links
Mozart	preorder	term	only shared/no
Alice	postorder	bytecode	only shared/yes

Table 4.5: Classification of Pickling Mechanisms: (a) Metaphors and (b) Sharing.

identifiers instead of node constructors in other places in the pickle.

All Nodes Are Marked. All nodes are considered potentially shared and are assigned identifiers. The advantage is that pickling can be performed in a single pass. Single-pass pickling however has the disadvantage that if the pickle is being sent out to a network at the same time it is generated, an explicit abort must be sent in the protocol if an unpicklable data structure is found [BP01]. Furthermore, with single-pass pickling the mapping from identifiers to nodes typically must be maintained in a dynamic (enlargeable) array or a hash table in the unpickler.

Only Shared Nodes Are Marked. This induces the cost of two-pass pickling; the first pass determines which nodes are shared, and the second inserts appropriate identifier assignments and identifier references into the pickle. If the pickle header explicitly announces how many identifiers are contained in the pickle, a static array can be used at unpickling time to reconstruct sharing. Performance is potentially superior to dynamic arrays.

No Nodes are Marked. The simplest approach is to not deal with sharing at all, as does Python's `marhsal` module. Cyclic data structures then lead to non-termination (or an abort when memory is exhausted). That sharing is not maintained is semantically observable in mutable data structures.

Table 4.5(b) summarizes how existing approaches handle sharing.

4.11 Summary: Towards a Pickling Mechanism for L

This section decides on the requirements for a pickling mechanism for L, suited to fulfill the goals defined in the beginning of this chapter. Chapters 5 to 7 develop a design according to these requirements.

Types Used by the Mechanism (Section 4.3). Pickling for L has to be reflection-based: L is a polymorphic higher-order programming language with expressive type abstraction facilities, which precludes type-directed pickling. For general applicability, L is based on a traditional store as present in many implementations of ML languages (see Section 3.1). As a consequence, the mechanism shall be based on a lower-level type system.

Types Used for Type-Checking (Section 4.4). L features packages (Section 2.3), which are dynamics at the level of the module language. Pickling can therefore be based on packages (the only data type that can be pickled or unpickled is the package), and thereby decouple type-checking from pickling. Packages can therefore be designed orthogonally. Types can easily be made picklable, even in an abstraction-safe way, by hashing [LPSW03] or by choosing a term representation. L will assume a term representation, which is outside the scope of this work.

Portability and Platform-independence (Section 4.5). A system based on pickling can only be open if pickles are portable across architectures. For the language itself to be open, pickles would have to be portable across implementations. This is a non-goal in the present work.

Treatment of Non-term Data (Section 4.6). Users of a higher-order language, in which functions are fully emancipated, would expect functions to be just as picklable as terms. When pickles are used as messages, they should be self-contained, which implies that functions should be pickled by value. Chapters 10 and 11 make heavy use of this property.

Resources are disallowed in pickles. This is an essential prerequisite for safety and security. Threads are, for simplicity, not supported; *captured components* as introduced in Section 9.2.3 provide a high-level solution to deal with migration and replication of computations.

Customization (Section 4.7). For simplicity, and because of the otherwise non-trivial implications for type-checking in the light of the foregoing design decisions, pickling for L shall support customization only at the level of data structures defined through the foreign function interface. The foreign function interface being unsafe anyway, this introduces no new security implications. Modal pickling as provided by Java is not needed for L, as discussed in Chapter 11.

Level of Implementation (Section 4.8). L is concurrent, and pickling is defined to clone state. Therefore, a high-level language implementation would not have the snapshot property (see Section 6.1). The pickling mechanism is therefore implemented as a service provided by the runtime. Since loading of components and distributed communication are to be based on pickling, pickling needs to have the efficiency only achievable by a low-level implementation.

Pickling and Unpickling Metaphors (Section 4.9). Pickling in L enumerates nodes in a depth-first postorder traversal. The metaphor for unpickling is bytecode execution. This approach has the best tradeoff between pickle sizes as well as pickling and unpickling efficiency [Tac03].

Representing Sharing in Pickles (Section 4.10). Pickles in L only mark those nodes as shared that actually *are* shared. Shared nodes are assigned consecutive integer identifiers starting from zero, and the pickle header announces the number of assigned identifiers. This makes it possible to use a static array to reconstruct sharing during unpickling, for improved efficiency. The extra cost incurred in pickling is small [Tac03].

Chapter 5

Pickling and Unpickling of Terms with Sharing

Pickling a data graph requires a traversal of the data graph, to produce a representation of it as a sequence of bytes. This chapter develops a generic pickling algorithm, restricted to data graphs that represent terms, that treats all nodes in the same way, and that maintains sharing. This makes the algorithm very general and applicable to a wide range of programming systems. In particular, the presentation does not assume any specific programming language.

With only few exceptions that are only published as source code, existing approaches linearize graphs in a top-down fashion. Tack [Tac03] provides empirical evidence that bottom-up unpickling is superior to top-down unpickling in terms of run time efficiency, with an insignificant increase in pickle size. The presented approach uses the following metaphors to reduce the design of a bottom-up mechanism to well-known problems in Computer Science:

- Pickles are bytecoded programs.
- Unpickling amounts to execution of a pickle program.
- Pickling is an application of depth-first search.

Overview. Section 5.1 introduces the data graphs on which the algorithms operate. Since unpickling can be useful without pickling, whereas the converse is not true, unpickling is developed first, in Section 5.2. A by-product is the definition of the format of pickles. A pickling algorithm to produce pickles in this format is then developed in Section 5.3. Section 5.4 concludes with a summary and description of validating implementations.

5.1 Data Graphs

As described in Section 3.1, a data graph is composed of nodes, annotated with labels, and carrying either a value or having a finite number of ordered directed edges to other nodes. Formally, this leads to the following definition [Tac03].

Definition 5.1 *Given a finite set of constructors con and a set of strings str, a finite function g is a* data graph *if and only if:*

$$\mathrm{Ran}(g) \subseteq con \times (str \uplus \mathrm{Dom}(g)^*)$$

The elements of $\mathrm{Dom}(g)$ are the nodes. Scalar data is represented by nodes carrying a string value.

Logical View. Logically, edges emanating from nodes can be seen as arguments to the constructor associated with the node. A data graph therefore corresponds to a (possibly infinite) term. A `let rec` construct can be added for the sake of having finite terms represent finite data graphs.

At this level, scalar data needs not be treated specially: Instead, new constructors can be used to represent (encode) the string values. For this reason and for a more compact presentation, the following discussion does not treat scalar nodes in data graphs specially. The developed algorithms and pickle language could, however, trivially be extended to treat scalar data more directly and, possibly, more efficiently.

Implementation View. A programming system implements an instantiations of general data graphs. This means that it specifies the sets *con* (corresponding to the labels used by the abstract store, which are typically a finite subset of the integers) and *str* (typically as sequences of bytes). A data graph can be seen as a mapping from addresses to nodes, with $n \in \mathrm{Dom}(g)$ the addresses. This is the view taken by the interface of the abstract store. Depending on the constructor of scalar data, store implementations can choose optimized representations, such as an unboxed representation.

Non-term Nodes. The data graphs considered in this chapter describe terms with sharing. For now, assume that the pickler raises an exception when it encounters a non-term node. The pickler could recognize non-term nodes by designated label.

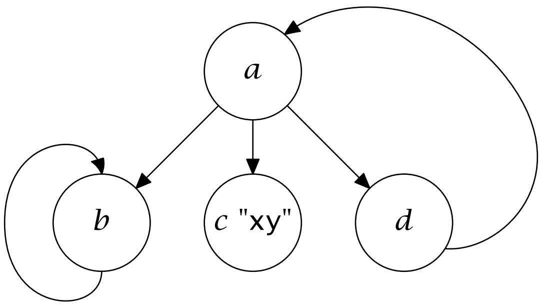
Example. Figure 5.1 illustrates the various views of graphs by an example. Here, $con = \{a, b, c, d\}$ and $str = \{\mathtt{a}, \ldots, \mathtt{z}\}^*$. (The notation $'\mathtt{x}', '\mathtt{y}'$ is used to stand for a sequence of integer codes representing the characters x and y in some character set.)

$$\begin{aligned} g = \{ n_0 &\mapsto (a, \langle n_1, n_2, n_3 \rangle), \\ n_1 &\mapsto (b, \langle n_1 \rangle), \\ n_2 &\mapsto (c, \text{"xy"}), \\ n_3 &\mapsto (d, \langle n_0 \rangle) \} \end{aligned}$$

(a) Data Graph.

$$\begin{aligned} &\text{let rec} \\ &\quad x_1 = a(x_2, c(\text{'x'}, \text{'y'}), d(x_1)) \\ &\quad x_2 = b(x_2) \\ &\text{in } x_1 \end{aligned}$$

(b) Logical View.



(c) Graphical Representation.

#0:	a	3
#1:	#4	
#2:	#6	
#3:	#8	
#4:	b	1
#5:	#4	
#6:	c	
#7:	"xy"	
#8:	d	1
#9:	#0	

(d) Store Representation.

Figure 5.1: Example: Views of a Data Graph.

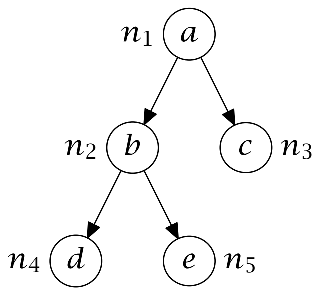
5.2 Unpickling

The following sections present algorithms to solve increasingly harder unpickling problems: First, the simple case of constructing trees is handled. This is then extended to cover directed acyclic graphs, in other words, to be able to establish sharing. The last extension adds support for general directed graphs, which may contain cycles.

Every step defines a *pickle language*. Programs in the pickle language, so-called *pickle programs*, consist of a sequence of instructions, whose execution by a pickle interpreter causes construction of a graph.

5.2.1 Constructing Trees

Logically, a tree is just a term constructed recursively by constructor applications. Evaluation of the term is similar to the evaluation of arithmetic expressions, for which a well-known compilation technique is to target an abstract stack machine [ASU86, Section 2.8]. Abstract stack machines are composed of a value stack (hence the name) and an instruction sequence



(a) A Tree.

```
CONS d 0
CONS e 0
CONS b 2
CONS c 0
CONS a 2
EOS
```

(b) Pickle Program.

CONS d 0	CONS e 0	CONS b 2	CONS c 0	CONS a 2
	n_5		n_3	
n_4	n_4	n_2	n_2	n_1

(c) Execution States.

Figure 5.2: Example: Constructing a Tree.

to evaluate; instructions consume values from the stack and push values to the stack.

Tree Pickle Language. Assuming a set ⟨*constructor*⟩ of valid constructors, pickle programs for constructing trees on an abstract stack machine can thus be defined as follows, in BNF notation:

```
⟨pickle⟩ ::= ⟨instrs⟩ EOS
⟨instrs⟩ ::= ⟨instr⟩ ⟨instrs⟩
           | ϵ
 ⟨instr⟩ ::= CONS ⟨constructor⟩ ⟨size⟩
  ⟨size⟩ ::= ⟨int⟩
```

The initial state is the empty stack of nodes. The CONS instruction takes as operands a constructor and an integer. It pops as many nodes from the stack, applies the constructor to them, and pushes the resulting node onto the stack. This amounts to a bottom-up construction of the tree. The EOS instruction marks the end of the program. It expects a single value on the stack, which is the root node of the final tree.

Example. Figure 5.2 shows an example tree, a pickle program that constructs it, and the sequence of states produced by execution of the pickle program. Above each state the instruction whose execution produced the state is given.

5.2.2 Constructing Acyclic Graphs

In contrast to a tree, a directed acyclic graph, or DAG, can express sharing of subtrees. Pickle programs must be able to construct sharing; unfolding a DAG into a tree may be semantically visible (if shared nodes are stateful), and it may be inefficient (it may cause an exponential blowup).

DAG Pickle Language. The pickle language needs to be extended by a mechanism that allows to again push nodes onto the stack that have already been built. Assuming a set ⟨*var*⟩ of variables, the pickle language is extended as follows (+= is used here to stand for addition of new production rules to an existing nonterminal):

$$\begin{array}{lll} \langle \mathit{instr} \rangle & += & \textsc{store}\ \langle \mathit{var} \rangle \\ & \mid & \textsc{load}\ \langle \mathit{var} \rangle \end{array}$$

The abstract machine is extended by a set of registers, designated by the variables in ⟨*var*⟩. The STORE instruction takes a variable and stores the topmost node from the stack in the register designated by the variable. The stack is not affected, but a subsequent LOAD instruction can now retrieve the node from the register and push it once more onto the stack.

Example. Figure 5.3 shows an example of a DAG, along with a pickle program that constructs it and the execution states produced by the program. Boxes in the state represent, from left to right, the registers associated with variables x_0 and x_1.

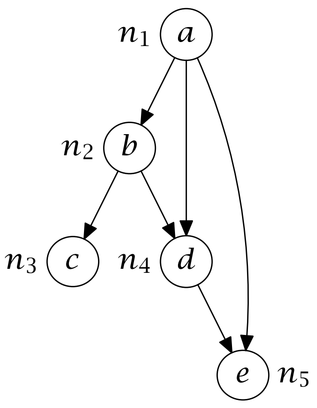
5.2.3 Constructing Cyclic Graphs

Unpickling must able to construct general directed graphs. What is still missing is the possibility to construct graphs with cycles. So far, nodes can only be constructed once all their children are known—which is not the case for any node that is part of a cycle. In other words, unpickling needs to first use a placeholder for some node on the cycle, construct the remainder of the nodes, then finally replace the placeholder by the actual node.

Pickle Language. Creation and replacement of placeholders is made possible by addition of two instructions to the pickle language:

$$\begin{array}{lll} \langle \mathit{instr} \rangle & += & \textsc{prom}\ \langle \mathit{constructor} \rangle\ \langle \mathit{size} \rangle\ \langle \mathit{var} \rangle \\ & \mid & \textsc{fulfill}\ \langle \mathit{var} \rangle\ \langle \mathit{size} \rangle \end{array}$$

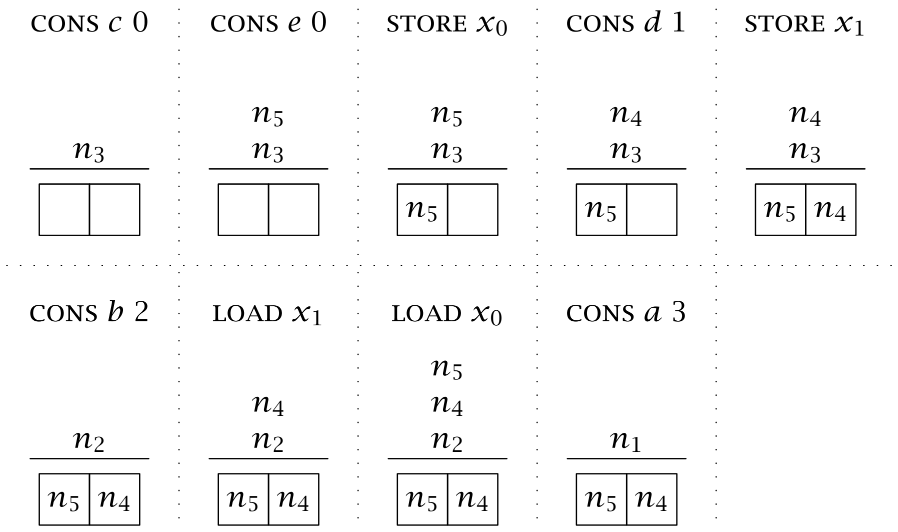
The instruction PROM takes a constructor, a size, and a variable. It introduces a placeholder for a node with the given constructor and size, pushes it to the stack, and stores a reference to it in the register denoted by the variable. In other words, it *promises* that the placeholder will once become a node with the given constructor and size. On the level of the abstract store, a node with the given constructor and of the given size is allocated (which is what the constructor and size operands are needed for), but not initialized—what is logically the placeholder is therefore, on the level of the store, an incompletely constructed node.



(a) An Acyclic Graph.

```
CONS  c  0
CONS  e  0
STORE x0
CONS  d  1
STORE x1
CONS  b  2
LOAD  x1
LOAD  x0
CONS  a  3
EOS
```

(b) Pickle Program.

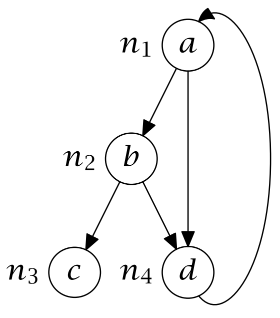


(c) Execution States.

Figure 5.3: Example: Constructing an Acyclic Graph.

Correspondingly, the instruction FULFILL takes a variable and a node count, retrieves the (uninitialized) node from the register denoted by the variable, and initializes its child edges by consuming as many nodes from the stack. (If the store supports a reflective operation that, given a node, returns its exact number of children, then the node count operand is not needed.)

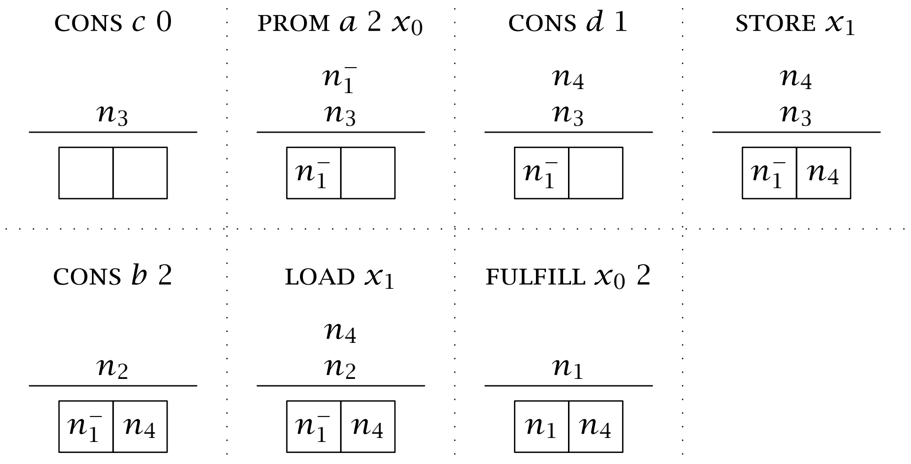
Example. An example graph containing a cycle is shown in Figure 5.4, along with a pickle program that constructs it and the execution states the program produces. Within execution states, the notation n^- is used to denote a node n that has already been allocated, but not yet initialized.



(a) A Cyclic Graph.

CONS	c	0
PROM	a	$2\ x_0$
CONS	d	1
STORE	x_1	
CONS	b	2
LOAD	x_1	
FULFILL	x_0	2
EOS		

(b) Pickle Program.



(c) Execution States.

Figure 5.4: Example: Constructing a Cyclic Graph.

5.2.4 Optimizations

As it stands, the pickle interpreter has to manage dynamic data structures during unpickling: a stack that grows dynamically, and a mapping from variables to registers that is extended incrementally. With the following small extension to pickle programs, these data structures can be allocated statically.

Maximum Stack Height. Abstract interpretation of a pickle program can determine the maximum height of the stack used during execution. If this number is announced in the pickle header, the unpickler can allocate a fixed array to hold the node stack.

Variables and Registers. The variables occurring in pickles can bijectively be mapped to consecutive integers starting from zero. Obtaining the register denoted by a variable then corresponds to indexing an array. If the number of variables is announced in in the pickle header, the unpickler can allocate a fixed array to hold the register bank.

Combined Instructions. Pickle programs can be transformed such that every STORE instruction is preceded by a CONS instruction. To reduce the size of pickles, these can be combined in a single instruction. Conceptually, STORE becomes be a prefix instruction modifying the subsequent CONS instruction. For symmetry, PROM can also become a prefix instruction.

Pickle Language. To summarize, here is the full version of the pickle language, including support for general directed graphs and the above optimizations:

$$\begin{array}{rcl}
\langle \mathit{pickle} \rangle & ::= & \langle \mathit{header} \rangle\ \langle \mathit{instrs} \rangle\ \textsc{eos} \\
\langle \mathit{header} \rangle & ::= & \textsc{init}\ \langle \mathit{nstack} \rangle\ \langle \mathit{nvars} \rangle \\
\langle \mathit{nstack} \rangle & ::= & \langle \mathit{int} \rangle \\
\langle \mathit{nvars} \rangle & ::= & \langle \mathit{int} \rangle \\
\langle \mathit{instrs} \rangle & ::= & \langle \mathit{instr} \rangle\ \langle \mathit{instrs} \rangle \\
 & | & \epsilon \\
\langle \mathit{instr} \rangle & ::= & \langle \mathit{cons} \rangle \\
 & | & \textsc{store}\ \langle \mathit{var} \rangle\ \langle \mathit{cons} \rangle \\
 & | & \textsc{load}\ \langle \mathit{var} \rangle \\
 & | & \textsc{prom}\ \langle \mathit{var} \rangle\ \langle \mathit{cons} \rangle \\
 & | & \textsc{fulfill}\ \langle \mathit{var} \rangle\ \langle \mathit{size} \rangle \\
\langle \mathit{cons} \rangle & | & \textsc{cons}\ \langle \mathit{constructor} \rangle\ \langle \mathit{size} \rangle \\
\langle \mathit{size} \rangle & ::= & \langle \mathit{int} \rangle \\
\langle \mathit{var} \rangle & ::= & \langle \mathit{int} \rangle
\end{array}$$

5.3 Pickling

The previous section has solved the problem of constructing general directed graphs and has defined a pickle language able to describe such constructions. This section now addresses the problem of generating a pickle program from a data graph. The problem is addressed in several steps. First, the easier problem of producing a pickle program from a tree-shaped data graph is tackled. The construction of pickle programs from general graphs is then reduced to (an extension of) the simpler approach.

5.3.1 From Trees to Pickle Programs

The construction of a pickle program from a tree is very similar to the compilation of an expression tree to a program for an abstract stack machine [ASU86, Section 2.8]. A data tree is like an operator tree, where the operators may be n-ary and correspond to the nodes' constructors. Compilation consists of performing a depth-first traversal of the tree and enumerating nodes in postorder as CONS instructions.

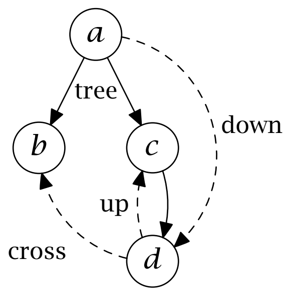


Figure 5.5: Example: A Graph with All Kinds of Edges.

5.3.2 From Graphs to Pickle Trees

The algorithm for pickling graphs is based on a classical depth-first search for graph traversal [Tar72]. The idea is to construct a corresponding tree during a depth-first search of the graph, and to introduce special nodes to represent the structural information that would not be present in the tree. Sedgewick classifies graph edges according to how they are visited in a depth-first search—into *tree* edges, *up* edges (from nodes to ancestors), *down* edges (from nodes to indirect descendants), and *cross* edges (from nodes to nodes that are neither ancestors nor descendants) [Sed83]. The various kinds of edges are illustrated in Figure 5.5. According to this terminology, construction of a pickle tree introduces special nodes exactly to represent the non-tree edges. In other words, sharing is made explicit.

Pickle Trees. A node $n \in \mathrm{Dom}(g)$ is called a *shared node* if it has more than one predecessor, or if it is the root node and has a predecessor. Let *var* be a set of variables, of which one is assigned to each shared node. Again ignoring scalar nodes for the reasons stated above, pickle trees correspond to the tree-shaped graphs g' with the following definition:

$$\mathrm{Ran}(g') \subseteq \mathit{con} \times \mathrm{Dom}(g')^* \uplus \mathit{var} \times (\mathit{con} \times \mathrm{Dom}(g')^*) \uplus \mathit{var}$$

Expressed informally, pickle trees can contain two kinds of nodes in addition to the nodes n representable in a data graph: *labeled* nodes (x, n) and *reference* nodes x for $x \in \mathit{var}$.

Constructing Pickle Trees. A pickle tree is obtained from a data graph by transforming all edges leading to shared nodes as follows. Let $n \in \mathrm{Dom}(g)$ be a shared node. Assign variable $x \in \mathit{var}$ to n. Replace all edges but one that lead to n by edges that lead to new reference nodes x, and replace the remaining edge by an edge leading to the new labeled node (x, n). Iterate until the graph contains no more shared nodes. The algorithm is shown in Figure 5.6.

Note that a single graph can have many pickle trees with distinct structure, but they all have the same set of nodes, modulo variable renaming.

```
input
    $g$, a data graph
    $var$, a set of variables
effect
    modifies $g$ to become a pickle tree
begin
    while there is some shared node $n$ in $g$:
        let $x$ be a variable from $var$;
        let $var := var \setminus \{x\}$;
        let $n'$ be a new labeled node $(x, n)$;
        let $n''$ be a node from $g$ whose edge $i$ leads to $n$;
        replace edge $i$ in $n''$ by an edge to $n'$;
        while $g$ contains a node $n'' \neq n'$ with an edge $i$ that leads to $n$:
            let $n'''$ be a new reference node $x$;
            replace edge $i$ in $n''$ by an edge to $n'''$
end
```

Figure 5.6: Algorithm for Constructing a Pickle Tree.

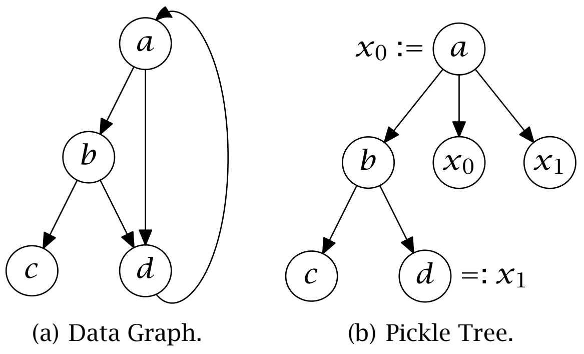


Figure 5.7: Example: Constructing a Pickle Tree.

Figure 5.7 shows a data graph and a pickle tree constructed from it. The graphical representation uses $x := \cdot$ to indicate labeled nodes, and x for reference nodes.

5.3.3 From Pickle Trees to Pickle Programs

Given a pickle tree, the next task is to construct a pickle program from it. This solution is again reduced to the compilation of expressions for a stack machine, as was the case for ordinary trees. The difference lies in the pickle program instructions generated for the newly introduced labeled and reference nodes.

Compilation of a pickle tree into a pickle program thus proceeds as follows. First, a linearized form as a sequence of nodes is produced by a postorder traversal of the pickle tree. Then, the pickler iterates through the nodes in the linearized form, generating code for each node according to the following cases:

Data Nodes. If the current node is a data node n, emit a CONS f m instruction, where f is the constructor of n and m is the number of children of n.

Labeled Nodes. If the current node is a labeled node (x, n):

Store. If this is the first occurrence of x in the linearized form, emit a STORE x prefix followed by the code for n.

Fulfill. If this is not the first occurrence of x in the linearized form, emit a FULFILL x m instruction, where m is the number of children of n.

Reference Nodes. If the current node is a reference node x, and the corresponding labeled node is (x, n):

Promise. If this is the first occurrence of x in the linearized form, emit a PROM x prefix followed by the code for n.

Load. If this is not the first occurrence of x in the linearized form, emit a LOAD x instruction.

5.3.4 Efficient Implementation

The preceding presentation gave a high-level description of the transformations performed by the pickler. This section discusses how to obtain an efficient algorithm for pickling.

Detecting Sharing. To construct a pickle tree, an algorithm must detect shared nodes. There are two fundamentally different approaches: maintaining a set of visited nodes, or marking visited nodes directly in their internal representation. In a garbage-collected store, marking of nodes can be implemented with no overhead, since the marking mechanisms used by the garbage collector can be reused (of course, after pickling is finished, marking has to be undone). The disadvantage of marking is that it prohibits concurrent pickling. This is not a problem when maintaining a set of visited nodes. Actually, the set approach has another advantage: Additional bookkeeping information can be associated with visited nodes. In other words, an incrementally extended mapping from nodes to bookkeeping information is maintained instead of a set of nodes. Such a mapping can easily be implemented as a hash table, using the address of the node in the store as hash function.

Interleaving Pickle Tree Construction and Linearization. The most efficient solution to constructing a pickle tree is a depth-first search of the graph. By the observation that a depth-first search of a pickle tree can be used to produce a postorder enumeration of nodes, linearization can be interleaved with pickle tree construction. As a consequence, the pickle tree never needs actually be constructed in memory. The added difficulty is that nodes are visited and linearized before their sharing status has been determined. The solution is to record bookkeeping information about visited nodes: When a node is visited for the first time, the offset in the linearized form is recorded. If a node is encountered for the second time, a variable is assigned to the node and recorded in the bookkeeping information, the linearized form is updated to replace the node by a labeled node, and a reference node is appended to the linearized form. Every subsequent visit of the node results in a new reference node being produced.

Interleaving Pickle Program Generation and Linearization. Explicit construction of the linearized form can be done away with by directly generating the pickle program instructions into a buffer instead. The problem is that nodes cannot be backpatched into labeled nodes (this would require moving parts of the buffer to accommodate the extra prefix instruction, in the worst case resulting in quadratic cost). The solution is to maintain an ordered list of pickle program offsets where a STORE instruction prefix has to be inserted. Only when writing out the pickle program buffer are the STORE instruction prefixes actually inserted.

Adding the Pickle Program Header. When the buffer is written out, the INIT instruction has to be prepended. At this point, the number of variables used in the pickle is known. What still needs to be computed is the stack height required for unpickling. This can be computed from the pickle tree: If n is a pickle tree node, the required stack height is given by the function

$$
\mathit{height}(n) = \begin{cases} 1 \\ \qquad \text{if } n \text{ is a leaf node} \\ \max_{i=0}^{m} (\mathit{height}(n_i) + i) \\ \qquad \text{if } n \text{ has children } \langle n_0, \ldots, n_m \rangle \end{cases}
$$

Right-to-left Versus Left-to-Right Linearization. Some languages feature right-recursive data structures, such as Standard ML lists. The preceding discussion assumed left-to-right construction of data structures. According to the above stack height function, right-recursive data structures require a larger stack. It may therefore make sense to partition the set of constructors into two sets, one with the constructors that tend to be used for left-recursive or balanced data structures, and another with the constructors that tend to be used for right-recursive data structures. For the second set of constructors, right-to-left construction could then be used.

5.4 Summary

The pickling mechanism presented above is implemented in the SEAM virtual machine for Alice. Because of its regularity and genericity, it requires only 1650 lines of C++ code. A first version of the pickler was top-down instead of bottom-up. It was roughly equivalent in size, but the efficiency of bottom-up pickling proved to be superior [Tac03].

An ancestor of SEAM's pickling is Mozart's pickler, now in its third generation, which is complex in comparison: Pickling always performs two traversals, implemented as instances of an abstract graph traversal algorithm that requires overriding 22 type-specific processing methods. Each node type has its own traversal routine, of which some iterate over their children left-to-right, others right-to-left. The macros for handling sharing need to be expanded on average 13 times per traversal implementation. There are two types of sharing detection: The first pass performs sharing detection on nodes in the Oz heap (labeling only shared nodes), the second pass performs sharing detection on manually managed C++ objects (all labeled). Unpickling is separated into a *builder* abstraction and the actual unpickler that parses the pickle. To accommodate two styles of C++ constructors for heap nodes, the builder implements a mixture of top-down and bottom-up construction (internally using a stack that can hold 40 types of task, of which some are reminiscent of S-pointers à la Warren Abstract Machine [AK91]). Mozart's pickling is expressed as 11250 lines of C++ code, and is outperformed by SEAM's pickler.

Chapter 6

Concurrent Pickling with State and Futures

This chapter analyzes the problems that state and futures cause in conjunction with pickling and concurrency, and proposes solutions to these problems.

The algorithm presented in the previous chapter did not make a distinction between immutable and mutable nodes, also called *stateful* nodes. This means that if no further provisions are made, pickling would cause stateful nodes to be cloned. Several scenarios could actually benefit from this behavior:

Checkpointing. To enable recovery from failures (induced by crashes or system power-down), long-running processes can regularly write their current state to disk, called a *checkpoint*. The checkpoint can then be used to restart the application. This amounts to cloning the application's state.

Server Replication. A server application could replicate itself onto another machine, for example, to react to increasing request rates or to accommodate for system maintenance. Such a replication could be implemented by cloning the server's state.

Problem: Preservation of Invariants. If pickling always blindly clones state, this can lead to semantic problems: In general, a stateful data structure has to fulfill representation invariants. Usually, these are guaranteed by encapsulating state in abstract data types. Every public operation supported by an abstract data types restores the invariants before it returns.

To illustrate a problem that cloning can introduce even without concurrency, consider an abstract data type with an iteration operation that takes a first-class function as argument. Assume that the iteration needs to call the first-class function at a point where the invariants temporarily do not hold. While the abstract data type can protect itself against the first-class function invoking other operations on the instance (using timestamping or a flag, for example), it cannot prevent the first-class function from pickling the instance—in which case the resulting pickle would describe an unusable instance. When pickling occurs on one thread while another thread mutates the data graph being pickled, the problem becomes apparent in many more scenarios. With future nodes, which can cause pickling to block indefinitely, the likelihood of the problem appearing is increased still further.

It seems that the implications of cloning state in the presence of concurrent pickling have never been discussed previously. This work identifies the problems and proposes solutions.

Overview. Section 6.1 proposes that a pickle should always describe a snapshot of the data graph in the store, even in the presence of concurrent mutators. This is a first prerequisite for tackling the problems introduced by concurrent pickling. Section 6.2 describes a possible implementation of a pickler that has the snapshot property. This solution does not interact nicely with futures, as developers can easily make mistakes that lead to deadlocks. This problem is mitigated by an extension described in Section 6.3. Finally, mechanisms alone always will be insufficient to protect developers from hard-to-find design errors, which is why Section 6.4 proposes design patterns that make it easier for developers to build pickling-safe abstract data types. Section 6.5 concludes with a summary.

6.1 Concurrent Pickling

A pickler is said to be *concurrent* if other threads that operate on the abstract store are allowed to execute in-between the points in time when a pickling operation starts and when it ends. If a pickler is not concurrent, in other words, it prevents any other threads from executing while it is running, then pickling becomes a bottleneck in designs that rely on frequent use of pickling.

Concurrent Pickling and Automatic Memory Management. One difficulty in implementing a concurrent pickler lies in enabling the pickler to run concurrently with the garbage collector. This requires that the garbage collector includes all bookkeeping information used by the pickler in its root set. The simplest approach is to represent all bookkeeping information by abstract store nodes. The algorithm presented in the previous chapter requires hash tables whose keys are store nodes. Such hash tables can be implemented by using node addresses as hash values; note, though, that a compacting garbage collector would have to be aware of such hash tables and re-hash entries for nodes that have been relocated to new addresses.

Concurrent Pickling and Mutators. The bigger problem is that concurrent threads can mutate the value being pickled before pickling is complete, in which case the resulting value may be inconsistent or break representation invariants. One answer to mitigate this problem is to require that the pickler has the *snapshot property*, defined as follows:

Definition 6.1 *Let $G(n, t)$ be the function that describes the subgraph in an abstract store that is reachable from a root node n at time t. Assume a concurrent pickler that pickles the graph reachable from n, and that the pickler runs between time points t_1 and t_2, producing a pickle representing the graph g. We say that g is a* snapshot *of n if and only if there is some $t \in [t_1, t_2]$ such that $G(n, t) = g$.*

We say a pickler has the snapshot property *if and only if it only produces snapshots of values in the abstract store.*

There are trivial ways to make a pickler have the snapshot property:

Forbidding Pickling of State. The problems only appear when pickling can clone state. The simplest possible answer, then, is to forbid in pickles nodes that represent state. This can easily be achieved by partitioning the set of constructors into $con_{\text{immutable}}$, which are the constructors for immutable nodes, and con_{stateful}, the constructors for stateful nodes.

Stop-the-world Pickling. If pickling is atomic, in other words, $G(V, t)$ is constant on $[t_1, t_2]$, then the pickler trivially has the snapshot property. This can be achieved by suspending all other threads when a pickling operation starts, and resuming then when it ended.

Neither of these solutions is satisfying, as either the expressivity of pickling is severely limited, or pickling can become a bottleneck in a system.

6.2 Pickling with Write Barriers

This section sketches a pickler that achieves the snapshot property by protecting all nodes it has visited against concurrent modification.

Assume the virtual machine supports a new kind of node called a *write barrier.* (The term is inspired by the write barriers employed in generational garbage collection.) Let a write barrier be a new kind of future node, as introduced in Section 3.4. The state of a write barrier is a pair of values (n, Ts), where n is the node protected by the write barrier and Ts is a queue of threads, called the write barrier's *suspension queue.*

When the pickler encounters a stateful node n, it constructs a shallow clone n' of n and replaces n in-place by a write barrier with state (n', nil). Additionally, the pickler records the new write barrier in a set called the pickler's *trail.*

Operations on stateful nodes must account for write barriers. Read operations (including picklers) ignore the write barrier and perform transparently on the value it protects. (In particular, two concurrent picklers cannot deadlock each other.) In contrast, write operations that encounter a write barrier record the current thread in the corresponding suspension queue and block the thread.

When the pickler has finished traversing the graph, all nodes are protected, therefore the produced pickle trivially is a snapshot of the graph. To undo node protection, the pickler iterates through its trail and binds each write barrier to the value it protected, thereby resuming all threads in the corresponding suspension queues.

Performance Overhead. The presented scheme provides for gracefully degrading performance with increasing use of pickling. Assuming that all operations perform a test for future nodes anyway, programs that do not use pickling suffer next to no performance hit: Only in the (infrequent) case that a future is encountered there is an additional test whether the future actually is a write barrier. When pickling is used, and thus new write barriers are created, threads concurrently operating on the data graph being pickled have to block. Conversely, this means that threads independent of that data graph continue to run at full speed.

The drawback is that the assumption above severely restricts opportunities to optimize away future tests through static analysis: Every stateful node can potentially become a future node.

6.3 Futures

Section 3.4 introduced future nodes. Not all of these should be treated as normal term nodes:

Promised Futures. At the language level, a promised future can only be bound by some computation that holds a reference to the corresponding promise. Assume that pickling could clone promised futures. Unpickling would therefore produce a new promised future, but there would be no computation to bind it—rendering the promised future useless. Instead, the pickler could block on promised futures or disallow them.

Lazy Futures. Lazy futures carry with them a computation that can produce the value they will be bound to. Cloning a lazy future can therefore make sense (provided the computation can be cloned, too). This approach is actually taken by Clean [VP02]. Alternatively, the pickler could evaluate the computation and pickle the computed value instead of the lazy future. The following trade-offs need to be considered:

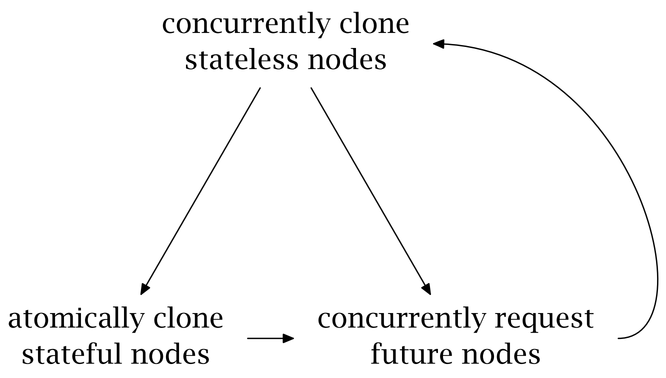

concurrently clone stateless nodes
atomically clone stateful nodes
concurrently request future nodes

Figure 6.1: Pickler Operation Modes.

- If the computation is cloned, it may have to be carried out multiple times—once for every clone.
- If, on the other hand, the computation is carried out by the pickler, to pickle the computed value instead of the computation, it could yield an infinite data graph (meaning that is does not terminate)—-whereas computations always have a finite representation.
- The computation might not be picklable because it references non-picklable nodes—but the value it produces could be picklable. (Also the converse could be true.)

Failed Values. Cloning a failed value produces, upon unpickling, a value that causes an exception to be raised upon access. There are pros and cons to this behavior. Assume that pickling was used to transmit the result of a remote function application—then, it would make sense to pickle a failed value to communicate failure. In persistence scenarios, on the other hand, it could mean that a failure is only discovered considerably later, at a point where it may have become impossible to retry the computation. In this case, it might be desirable to request the failed value at pickling time, causing an exception to be raised.

In summary, there are situations where pickling should block. Unfortunately, in this case, both atomic pickling and pickling with write barriers can cause a deadlock. This section proposes a way to recover the snapshot property.

Concurrent Pickling with State and Futures. Imagine a restartable pickler that can run in one of a number of modes, depicted with their transitions in Figure 6.1.

- At the outset, the pickler shall be concurrent, under the optimistic assumption that the data graph consists of immutable nodes only and does not contain futures. If this assumption holds, pickling is a concurrent activity with the snapshot property.

- Once the pickler encounters the first stateful node, it changes its mode of operation to become atomic (either by suspending all other threads, or by using write barriers). As long as the pickler encounters no future, it trivially has the snapshot property.
- Once the pickler encounters the first future node that it has to block on, the pickler discards the output buffer constructed so far (and unmarks any write barriers, if applicable), becomes concurrent again, and traverses the data graph solely to request all reachable futures. (Since the pickler does not produce a pickle in this mode, it does not lose the snapshot property.) Thereafter, the pickler restarts from the root node as a concurrent pickler.

In any mode, a non-picklable node causes the pickler to abort with an exception.

Properties of the Pickler. Because the pickler chooses its mode of operation depending on the nodes it encounters, its performance gracefully degrades as state and futures occur in data graphs. There are some drawbacks to restarting the pickler after requesting futures, however, since the pickler potentially traverses a data graph that has been concurrently modified in the mean time:

- The modified data graph can contain new futures. Examples can be constructed where pickling would never terminate.
- The set of requested futures can depend on timing, and pickling can request futures that were not actually referenced from the nodes that are, in the end, pickled.
- Pickling can fail due to a non-picklable node temporarily featuring in the data graph, even though another pass on the data graph could have been able to produce a pickle.

These properties should be kept in mind by users of the language, especially library designers. This is one more motivation for introducing design patterns for pickling-safe abstract data types.

6.4 Design Patterns for Pickling-safe Data Types

Introducing pickling into a programming language is a more significant change than is first apparent. The following two problems have been identified above: (1) Pickling of stateful nodes means that invariants on data structures can, in general, not be guaranteed. (2) Futures can cause the pickler to not terminate, deadlock, or behave non-deterministically (depend on timing).

To deal with similar semantic issues, concurrent programming has introduced a property of abstract data types called *thread-safety*. A thread-safe abstract data type has the property that concurrent invocations of operations on instances of that data type can never break invariants (that is, an operation can never see an inconsistent instance). Similarly, when programming with pickling, abstract data types may or may not be *pickling-safe*, which means that pickling will never produce an inconsistent instance.

Examples. Consider a thread-safe abstract data type *t* that synchronizes all operations on all of its instances using a single global lock (as defined in Section 2.4):

```
type t = { a : int ref, b : int ref }
(* invariant: for x : t, !(#a x) = !(#b x) *)
fun new () = { a = ref 0, b = ref 0 }
val inc =
    let
        val lock = Lock.lock ()
    in
        Lock.sync lock
                  (fn x => (#a x := #a x + 1;
                            (* (+) *)
                            #b x := #b x + 1))
    end
```

The data type may be thread-safe, but it has several issues with pickling. (1) If one thread invokes `inc` on an instance and another thread pickles the same instance, it may happen that the pickling thread gets to run while the incrementing thread is suspended at program point `(+)`. The clone breaks the invariants since the pickling thread ignores the lock. (2) Assume that the procedure `inc` was pickled. Two instances of the global lock could coexist in the same process, since the global lock is part of `inc`'s closure.

Solution: Design Patterns. One solution to mitigate these problems is to introduce *design patterns* [GHJV95, BMR$^+$96], similar to those employed in concurrent programming (see [Lea99], for example). Library designers have to define, document, and implement a pickling behavior for the abstract types they define. The purpose of design patterns is to help library designers choose and communicate specific behaviors.

As a starting point, the following paragraphs sketch some possible design patterns for various pickling behaviors:

Non-picklable Instance. Sometimes, picklability is either not possible or not desirable. Examples are lazily constructed infinite data structures (where pickling would lead to an infinite loop or memory exhaustion), or concurrently constructed infinite data structures (such as streams terminated in a future, where pickling would necessarily deadlock), or singletons (as in Problem 2 above).

Instances of such a data type can simply be made non-picklable, causing an exception if encountered by the pickler. The pattern is to embed an artificial reference to a resource into the type's representation (say, by pairing). Such "non-picklability marking" could be provided as an abstract data type in a library, allowing for a more direct implementation and making its use clearer, since the resource is likely never to be accessed.

Single Stateless Node. Instead of a record with one reference cell per mutable field, use a single reference cell containing a record with immutable fields only. For instance, to solve Problem 1 above, change the definition of `t` to read `{ a : int, b : int } ref`. Good candidates for stateless representations with efficient algorithms can, for example, be found in purely functional data structures [Oka98].

Pickling-safe Lock. It is possible to protect a data structure by a lock. Both pairing the data with a `Lock` instance (where the lock is the first component of the pair) or embedding the data within a `Locker` instance work fine: These locks are implemented by reference cells, so once the pickler reaches the lock, it becomes atomic. If the lock is currently not taken, atomic pickling will create a snapshot. If the lock is currently taken, the pickler will find a future within the reference cell, causing it to block and restart—until it manages to "obtain" the lock. Note that this pattern would not work with the `BusyWaitingLock` implementation from Section 2.4, as that lock is clonable, including the lock's state.

Cached Optimized Representation. This pattern is best motivated by an example. Assume an abstract data type representing Standard ML signatures, to be used for separate compilation. The items comprising the signature are best stored as a list, because this allows to express both the correct scoping and hiding. Lookup is presumed to be frequent, so the items should also be stored in a hash table. The pickle should contain only the list representation for compactness, so the hash table should be reconstructed upon unpickling. The hash table is a *cached optimized representation.*

The general pattern is as follows: Define an abstract type t. Define a type `cloned` to hold the cloned part and `cached` to hold the cached part. Generate a unique identifier of some type `id` upon instantiation of the data type, and define t = `id` $\times$ `cloned`. Define a global variable to hold a weak mapping from `id` to `cached`. In each operation on instances of `t` that requires access to the cached part, do the following under mutual exclusion: Check whether the domain of the mapping contains t's identifier; if yes, retrieve the associated value, otherwise, compute the cached part from the cloned part and extend the mapping. All mutation operations must maintain consistency between cloned and cached parts.

Resource Rebinding. A variant of *Cached Optimized Representation* allows to implement resource rebinding [FPV98]: Let the cloned part be a symbolic representation of a resource and the cached part be the (non-picklable) resource itself.

6.5 Summary

This work has described a concurrent pickling mechanism that interacts nicely with state and futures, with gracefully degrading performance as stateful nodes and future nodes occur in the traversed data graph. The described pickler with three modes of operation has been implemented in Seam. Because pickling using write barriers severely restricts potential for optimizing away future tests through static analysis, Seam does not use write barriers to achieve atomicity; instead, it makes the pickling thread non-preemptable (this effectively suspends all other threads since Seam only implements cooperative multithreading).

Future Work. Additional work is needed to identify more design patterns for pickling-safe abstract data types. In particular, a pattern to express high-level customization would be useful. While this seems feasible for dynamically-typed languages, for statically-typed languages, no type-safe solution is known.

Chapter 7

Pickling of Non-term Nodes

This chapter extends the pickling mechanism from the previous sections with provisions for dealing with non-term nodes, in particular, resources and functions.

Overview. Chapter 5 developed a pickling mechanism for terms, which meant that all nodes were treated in the same way. This assumption breaks down for nodes that do not represent terms. Specifically, this chapter examines resources (Section 7.1) and functions (Section 7.2). Section 7.3 presents a customization facility to the pickling mechanism, which can be used to implement the pickling behavior of arbitrary non-term nodes. Section 7.4 concludes with a summary.

7.1 Resources

Some of the abstract types that libraries define represent *resources*. A resource is data that pertains to the process in which it lives, and that should not or cannot be meaningfully interpreted within other processes. Examples are operating system handles.

At the implementation level, resources are typically embedded into the language graph represented in the store by representing as integers. These integers are interpreted by the built-in operations on the resource nodes: For instance, the Unix operating system represents handles to open files, pipes, or sockets (*file descriptors*) as small integers [Ste97]—which can directly be embedded in the language graph. In Microsoft's Windows operating system, handles are pointers into the address space of the creating process [Ric97]. These pointers can be treated as integers and embedded in the language graph.

If resources were represented as integers, the default pickling behavior on them would be that they are cloned. This can cause a number of issues:

Semantics. When interpreted in the context of another process, the integer is likely to denote an arbitrary object, if it denotes a valid object at all. In other words, behavior is, in general, undefined.

Security. The programming language's safety depends on lexical scoping. For unpickling not to violate lexical scoping, it must create a self-contained data graph in the store. In particular, it must not cause references to resource nodes to be created out of thin air.

In short, the pickler must not clone resources. Instead, it must fail when it encounters a resource. Therefore, a resource node cannot simply be an integer, but must be marked so that its resource-ness becomes apparent. To this aim, the set of constructors can be partitioned into a set of term constructors and a set of resource constructors:

$$con = con_{\mathrm{Term}} \uplus con_{\mathrm{Resource}}$$

7.2 Functions

In languages with first-class functions, nodes in the abstract store can represent functions. For functions to really be emancipated data structures, they need to be picklable, and for pickles to be self-contained, they need to be pickled by value. Concretely, this means that the definition of the function (the description of the computation carried out by the function, also called the function's *code*) should be contained in the produced pickle.

As described in Section 3.2, functions come in two flavors: primitives and closures (user-defined functions).

Primitives. Primitives are used to implement operations on built-in types of the programming language, and calls to the operating system. These two are markedly distinct for the purpose of pickling: There is no reason not to make built-in operations picklable, but if system operations were picklable, an unseeming first-class function obtained from a pickle (which was supposed to be self-contained) could, when applied, obtain references to resources.

Different designated constructors can be used to distinguish built-in primitives from system primitives. In the abstract store, primitives can be represented as pointers to a native function provided by the virtual machine; on a pickle, primitives need to have a symbolic representation such that the unpickling process can substitute the corresponding implementation.

Closures. A closure of a user-defined function is a pair of an environment and the function's code. The closure and environment can easily be represented as a store node, and be treated as a normal term node. For code, the following two approaches are conceivable:

Separate Code Area. The virtual machine described in Section 3.2 stores code in a separate code area. A store node would reference code in the code area by storing a pointer into the code area as an integer. In this case, the pickler has to recognize this as a reference into the code area, so that it can extend pickling to include a section of the code area (possibly with relocation information). Depending on the structure of the code area (imagine embedded immediate operands, termed "V registers" by Scheidhauer [Sch98]), a more or less complex second pickler has to be designed.

Code Embedded in the Data Graph. As an alternative, code could be represented within the data graph itself—as store data structures. For example, byte code (or even position-independent native code) can be stored in string nodes, or abstract syntax trees can be stored as graphs of nodes. In this case, it may even be sufficient to just clone the code data structure.

7.3 Customization

A comparison of the two approaches to pickling code from the previous section sparks the following idea: Reduce pickling of non-term nodes (possibly including data types defined via a foreign function interface) to pickling of non-term nodes, and use a single pickler for both. This section develops a customization mechanism based on this idea.

Non-term nodes have an internal representation, which can, in particular, be a pointer into the unmanaged heap (effectively enabling any virtual machine implementor to arbitrarily choose an internal representation). Also, non-term nodes may or may not have an external representation. If they do, the expressive language of data graphs can be used to describe it.

To realize this idea, the following extensions are sufficient:

- The pickle language needs an instruction to convert the data graph that is currently on top of the stack into an internal representation. The instruction takes as operand an identifier denoting the node's type (for example, "built-in primitive" or "code").
- The unpickler looks up an implementation-dependent function for the identifier that creates an internal representation, when given the data graph from the stack as argument.
- The abstract store has to make internal representations apparent. The mechanism to do this is up to the implementation; for example, a designated node label could indicate that a given node is an internal representation. The node's first edge could give an identifier denoting the node's type, and the second edge the actual internal representation (for example, a native pointer encoded as an integer).

- The pickler uses the type identifier to look up and invoke a function. The function creates and returns a data graph that describes the internal representation.

Example: Resources and System Primitives. Resource nodes and system primitives would, in this model, both have identifiers with which no externalization function would be associated, causing the pickler to raise an exception.

Example: Built-in Primitives. The externalization function of a built-in primitive would return a string denoting the given primitive. The internalization function would look up the implementation of the primitive, given the string.

Example: Code. An external representation of code must be portable, easy to explain and process, and perhaps include typing information to make it possible to check well-formedness. Possible choices are bytecode instructions, an abstract syntax tree representation, or even source code. Code in an internal representation must be efficient to execute. For instance, instruction opcodes may be replaced by addresses to the machine code that implements them (so-called *direct threaded code* [Bel73]), branch targets may be specified as absolute addresses, or the code could even be represented as a native code translation. A conversion from the external to the internal representation can be performed by run-time compilation (also called just-in-time compilation, or *JITting*). Unless decompiling JITted code is easy, one would retain the data graph representation even after JITting to enable externalizing the code again.

7.4 Summary

This chapter proposed a simple mechanism that allows to customize pickling at the level of the virtual machine. Still, it is sufficiently general to forbid resources, externalize primitives symbolically, and clone definitions of user-defined functions. Outside the scope of this work is verifiability of pickles. In the presence of customization, and of user-defined functions in particular, verifiability becomes an interesting problem. For dynamically-typed languages, “some” verifiable representation would probably easy to devise: Since all operations are preceded by runtime type-checks, term data (and the environment of closures) does not need to be verified (beyond the pickle being well-formed); for code, the simplest approach would be to store it in source form. For statically-typed languages, this simple solution is ruled out.

Validating Implementations. The sketched customization mechanism has been implemented in Alice-on-SEAM [BK02, BK03].

Chapter 8

Components for Modular Programming

This chapter extends L with a powerful mechanism for modular programming with separate compilation. This defines the first version of OpenL. The realization of the mechanism requires only a minimal extension to the syntax of the source language introduced in Chapter 2, and a modification of the startup procedure of the virtual machine introduced in Chapter 3. The mechanism establishes the basis of OpenL's component system, as presented in Chapters 9 and 10.

Motivation. This chapter defines a component model targeted specifically at fulfilling the following business requirements:

Components for System Composition. Customers need software systems that adapt as their needs change. In other words, program composition, also called *configuration*, is performed on the customer's machine, not the vendor's machine.

Binary Components. Component vendors ship libraries, user interface elements (*controls*), and applications as (sets of) *binary* components. The benefit to customers is performance: no compilation is necessary on their machine. To component vendors, binary components mean protection of the intellectual property that would be apparent in source code.

Interchangeability of Components. The customer, not the application vendor, can choose which component provides a given service—for instance, a spell-checking service for use by applications supporting text input.

These requirements translate into the following technical requirements:

- Components are typed by *interfaces*[1]. Many different components can

[1]Be it either through typeful programming [Car91], or through static typing.

provide implementations for the same interface. Components must be self-describing in the sense that they package their interface with their implementation in a single unit.

- Component systems must enable *separate compilation*. This is to enable compiling single components, as opposed to compiling whole applications, or compiling components only when the implementations of components they depend on are known.
- Component-based applications are assembled at run time, not at compile time. This assembly operation is called *linking*.

Overview. This chapter presents the foundation for a component system, which fulfills the technological requirements outlined above. Section 8.1 introduces OpenL as a minimal extension of the syntax of L to enable modular programming, initially under the assumption of dynamic typing. An OpenL source file defines a component, in contrast to L, where a source file defines an application. Section 8.2 defines the semantics of OpenL programs composed of a set of components, from which Section 8.3 derives a model for separate compilation and linking. Section 8.4 modifies the virtual machine to run componentized programs by linking graphs of components at boot time.

Under the assumption of dynamic typing, OpenL components can be directly translated to L programs. Section 8.5 analyzes with what assumptions and limitations the mechanism carries over to statically-typed programming languages. Related work is then surveyed in Section 8.6. Finally, Section 8.7 summarizes how the concepts presented in this chapter are realized and validated in two programming systems, namely Mozart and Alice.

8.1 Componentizing Programs

OpenL is a strict extension of L, which differs from L in that every source file defines a *component* instead of a monolithic application. Every component defines a *namespace*, akin to an L structure (see Section 2.2). Syntactically, an OpenL component consists of a possibly empty sequence of *import announcements*, followed by a sequence of declarations. The import announcements specify which program objects from other namespaces can be referenced by this component, while the declarations define the contents of the namespace defined by this component. The declarations may only reference identifiers bound in the *initial environment*, identical to the initial environment from Section 2.1.3 in which L source files are interpreted, as well as identifiers bound by import announcements.

Every namespace is named; for now, assume the name of a namespace is given by the base name of the source file. Import announcements refer to other namespaces by name. Syntactically, an import announcement consists of the new keyword **import**, an enumeration of identifiers, the new keyword **from**, and a string denoting a namespace name. Since L distinguishes between identifiers for types, values, signatures, and structures and functors, each identifier in an import announcement shall be prefixed with a keyword (**type** or **datatype**, **val**, **signature**, and **structure** and **functor**, respectively).

Examples. The following simple declaration, when placed in a file with base name Fac, defines a component Fac:

```
fun fac n = if n < 2 then 1 else n * fac (n - 1)
```

A component Main that wants to reference fac from component Fac could look as follows:

```
import val fac from "Fac"
val x = fac 7
```

Components vs. Modules. OpenL uses the term *namespace* to mean the same as a *module* in, say, Oberon-2 [MW91]. This is to avoid confusion with L's notion of a *module*, which is a simple structure or functor. An OpenL namespace is, of course, similar to an L structure—this exact similarity between compilation units and structures, and between component interfaces and signatures, has been previously pointed out by Burstall [Bur84].

8.2 Semantics of Componentized Programs

The definition of the syntax above already hinted at the semantics of OpenL components. The goal of this section is to better capture the semantics of an acyclic directed graph of components, also called a *componentized* program.

A Source-level Rewriting Semantics. For an initial definition, the idea is to first translate every OpenL component into an L source fragment, then concatenate these fragments to obtain an equivalent L program:

- Replace every component body by a declaration of a structure, bound to an identifier obtained from the name of the namespace defined by the component. The body of the structure is a **local** declaration whose private part contains one declaration for each imported identifier, and whose public part contains the declarations of the component. The import declarations bind every imported identifier to a selection of the corresponding program object from the namespace structure corresponding to the imported component.
- The import announcements span a dependency graph whose nodes are components. Assuming that this dependency graph is acyclic, the

components can be *topologically ordered*, which means that they are put into a total order in which every component is preceded by all of its dependees. Concatenate all structure declarations obtained as described above in a topological order, obtaining an L program.

Let the semantics of a componentized OpenL program be the semantics of the obtained L program. Note that because there may possibly be more than one topological order for a given dependency graph, and because declarations can have side-effects, this definition is intentionally ambiguous.

Example. The OpenL program consisting of `Fac` and `Main` from the previous section is equivalent to the following L program:

```
(* fragment corresponding to component Fac *)
structure Fac =
struct
    local
        (* no import announcements *)
    in
        fun fac n = if n < 2 then 1
                    else n * fac (n - 1)
    end
end

(* fragment corresponding to component Main *)
structure Main =
struct
    local
        (* from import announcement *)
        val fac = Fac.fac
    in
        val x = fac 7
    end
end
```

8.3 Separate Compilation and Linking

Cardelli calls the above approach to defining the semantics of a componentized program a characterization of “merging of sources” [Car97]. This does not capture the notion of separate compilation. Consider the following two steps to remedy this:

1. To factor out individual components from the generated monolithic L program, perform lambda lifting [Joh85] on each structure. Lambda lifting turns an expression with free variables into application of a function that takes those variables as arguments. Lambda lifting a

structure produces, correspondingly, a functor definition and a functor application.

2. If one ignores the functor declarations, all that remains is a sequence of functor applications, which is exactly the scaffolding necessary to compose components into an executable program. This skeleton will, in the following, be called a *linking program*. Correspondingly, a *linker* is just a program that generates a linking program.

In summary, a compiled component is a file whose base name is the name of the namespace defined by the component, and whose contents is a functor definition and the metadata needed to generate a linking program. To figure out what metadata is needed, consider how a linker operates: A linker needs to perform, starting from a root component, a depth-first search of the component dependency graph, and output functor applications in post-order to evaluate the component graph bottom-up. (The linker would fail if the graph contained cycles.) This means that the only metadata that is needed for a component is a vector of strings, where each string in sequence would give the name of the namespace the corresponding functor argument corresponds to.

Example. Revisiting the above example, a compiled component for `Main` would contain two pieces of information: the metadata `#[]` (an empty vector of strings), and the code for a functor definition such as the following:

```
functor MkFac () =
struct
    fun fac n = if n < 2 then 1 else n * fac (n - 1)
end
```

Similarly, a component for `Fac` would contain the metadata `#["Fac"]`, and the code for a functor definition such as the following (types omitted for now):

```
functor MkMain (structure Fac : ...) =
struct
    local
        val fac = Fac.fac
    in
        val x = fac 7
    end
end
```

A linker would generate the following linking program for `Main`:

```
structure Fac = MkFac ()
structure Main = MkMain (structure Fac = Fac)
```

As another example, to illustrate sharing, assume a componentized program structured as shown in Figure 8.1. The following linking program

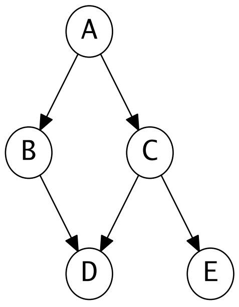


Figure 8.1: Sample Component Dependency Graph.

would perform the linking steps necessary to produce the corresponding namespaces A to E:

```
structure E = MkE ()
structure D = MkD ()
structure C = MkC (structure D = D  structure E = E)
structure B = MkB (structure D = D)
structure A = MkA (structure B = B  structure C = C)
```

Prerequisite: Higher-order Functions. Note that this definition of separate compilation requires higher-order functions: Component declarations typically define functions, which are (after compilation) nested inside a functor. At the level of the intermediate language L_I introduced in Section 3.5, this means that component functors become higher-order functions.

8.4 Execution of Componentized Programs

An OpenL stand-alone program is defined by a component, which is taken to be the root of a component graph. The effect of a stand-alone program is defined to be the sequence of side-effects generated by a linking program for the root component. (Note that the order of side-effects can depend on the topological ordering chosen by the linker, which is consistent with the degree of freedom implied by the rewriting semantics.)

In other words, the virtual machine for OpenL is started with the name of a compiled component as argument. Contrast this to the virtual machine for L, which is started with the name of a self-contained compiled program, as described in Section 3.5. The OpenL virtual machine has a *boot linker* that loads the root component, interprets its metadata, and recursively loads the components directly and indirectly referenced by the root component, incrementally adding code to the code area. Interleaved with the traversal, it interprets a linking program it dynamically generates. To preserve sharing of namespaces, and to evaluate every component only once, the boot linker needs to record the namespace resulting from each component functor application. To this end, it maintains a *boot namespace table* that maps namespace names to namespaces.

8.5 Application to Statically-typed Languages

For simplicity, the above discussion assumed a dynamically-typed language, where the built-in run-time type checks are sufficient to ensure that the component extensions do not compromise type-safety of the underlying language. Statically-typed languages, however, rely on static types to enforce abstractions and hiding, and omit run-time type checks. In the following respects, the semantics of components as introduced so far are lacking:

- Import announcements as described above do not carry enough information to perform compile-time type checking.
- The linking programs generated by the virtual machine are not type-checked, and would be executed even if they were ill-typed.

These shortcomings are addressed in the following two sections. Note that statically-typed languages can no longer express the linking programs, which is why this section descends to the level of L_I.

8.5.1 Component Types and Compilation

At the level of the core language, elaboration of source programs can infer most types automatically. At the level of the module language, however, type inference is not possible. In the case of typed OpenL, this means that the compiler needs to know the types of imported structures. This can be achieved through two mechanisms:

Explicitly-typed Imports. Identifiers in import announcements can be decorated with their full type. This is required, for instance, when compiling a component against a component whose interface has been defined, but for which no implementation is available (or should be assumed).

Load Dependees at Compile Time. Component metadata has to include the full types of all items it exports, for the purpose of link-time type-checking as defined in the next section. This means that if the compiler encounters an identifier in an import announcement that has not been decorated with its type, it can require an implementation of the dependee component to be available at compile time and load the type from its metadata.

When a dependee component is not available at compile time, programmers have to use the first mechanism. If dependee components are available, the second mechanism is, in general preferable: Fully annotated import announcements are large and prone to error, and cross-component type inconsistencies will remain undiscovered at compile time, and only become apparent at link time via the mechanism described next.

8.5.2 Component Types and Run-time Linking

Recall that the virtual machine generates and interprets linking programs at boot time. In a statically-typed setting, this has two implications:

- Linking programs need to be type-checked before they may be executed.
- Linking programs apply functors, and applying a functor requires a signature coercion on all argument structures. Signature coercion may require a representation change (see Section 3.5).

A linking program is type-correct if the actual type of each imported namespace *matches* the type that it was assumed to have at compile time. (Note that a namespace's type can be fully described by a single signature, because namespaces are actually represented as structures.) This means that the linker has to type-check and perform one signature coercion for every edge in the dependency graph. To enable this, the metadata of a compiled component has to specify:

- A signature describing the type of the namespace computed by the component.
- For every imported namespace, a signature describing the type of the namespace that was assumed at compile time.

Implementation. The virtual machine's boot time linker generates linking programs at the level of L_I. Every signature and every namespace has a first-class L_I representation. This means that the linker needs an operation such as the following (L_I is untyped; types are used here solely for documentation):

```
val match : namespace * sign * sign -> namespace
    (* Mismatch *)
```

The first argument to `match` is a namespace, whose type is given by the second argument. The third argument is the type this namespace is expected to have. If the types do not match, `match` raises the `Mismatch` exception from Section 2.3, causing the linker to abort; otherwise, `match` returns the namespace coerced to the expected type.

With this operation, the following is a linking program, expressed in L_I, for the component graph from Figure 8.1. This program assumes that `f`*X* is bound to a function representing the component functor producing namespace *X*, `sign`*X* is bound to a representation of the actual type of namespace *X*, and `sign`*XY* is bound to a representation of the type of namespace *Y* as expected by component *X*:

```
val E = fE #[]
val D = fD #[]
val CD = match (D, signD, signCD)
val CE = match (E, signE, signCE)
val C = fC #[CD, CE]
val BD = match (D, signD, signBD)
val B = fB #[BD]
val AB = match (B, signB, signAB)
val AC = match (C, signC, signAC)
val A = fA #[AB, AC]
```

Limitation. This section has made the assumption that every component is evaluated exactly once, since in modular programming (as opposed to real component programming) the program is structured as a static graph of components. As a consequence of this assumption, the simple approach to type-checking works even for components whose export signature mentions types from one of the import signatures. As more flexible component programming is enabled by Section 9.2.2, the need for more flexible type-checking is called out.

8.6 Related Work

Modular Programming. The idea of an industry that produces software components to increase the productivity of software development has first been published by McIlroy [McI68]. Looking back, the fundamental feature to enable this vision is that of a well-defined module system, as featured in a large number of programming languages designed in the 1970's. Examples of this are Mesa [LS79] developed at Xerox PARC for systems programming, which directly inspired Wirth's Modula [Wir95]; as well as CLU [Lis93], which integrated the ideas of data abstraction and modularity. The key idea was to separate interface from implementation. These languages have in common that the principle of separate compilation and static linking is part of their definition. As was reinforced by Szyperski [Szy02], reuse of software components needs to be based on compiled code as opposed to source code, which prerequires separate compilation.

Theoretical Foundation for Linking. A thorough and founded analysis of linking started only comparatively late, with Cardelli's influential paper [Car97]. According to the terminology introduced by Cardelli, our compiled components are "linksets". Cardelli focuses on theoretical modeling of linking, as opposed to engineering features necessary to enable component programming.

C. The C Programming Language [KR88] supports separate compilation. However, there is no well-defined concept of a module, and linking only has a low-level definition: Every *symbol* (identifier) declared at the top level in a source file can have either *internal* or *external linkage*. The result of compilation is an *object file*, which basically consists of code segments, data segments, a table of defined symbols mapping symbol names to offsets within segments, and a table of undefined symbols listing all references to the symbol from within segments. The linker constructs a single executable or library from a set of object files by merging the segments from all object files and patching cross-object file symbol references to the actual offset in the resulting file. At this point, all symbols with external linkage are part of a single global namespace: no symbol must be defined in more than one object file; and if producing an executable, all symbols must be resolved. The crudeness of this mechanism is due to the fact that historically, every computing platform had a single defined object file format and a single linker, over which compiler writers had no control. The single global namespace precludes instantiating a component more than once in a single program, as pointed out by Flatt and Felleisen [FF98].

Pebble. The idea of regarding components as functions is first found in the work of Burstall [Bur84], who describes an approach that translates programs with modules into typed functional programs, in the context of the functional language *Pebble*. The goal of his work is to study modularity features in programming languages. The present work uses the same foundation, albeit in pursuit of a very different goal.

Standard ML. Standard ML [MTHM97] features a powerful and formally defined module system, which provided a fruitful context for theoretical research. However, Standard ML is only considered with modules as a mechanism for abstraction, not componentization, and does not even provide for separate compilation in its definition. The Standard ML of New Jersey implementation provides a *compilation manager* [BA99] that tackles separate compilation with automatic dependency analysis and cross-module optimizations, but not binary deployment and dynamic linking as required for component programming.

Units. Flatt and Felleisen [FF98] presents Units, which are software components for higher-order typed programming languages. Units represent the approach most similar to the present work. Flatt and Felleisen synthesize their model from the ideas of object files for separate compilation, packages for integrating the module and the core language, and ML-style functors for combining modules into programs and making Units replaceable. Units specify component linkage outside of components (similar to the above linking programs), as opposed to hard-wiring them within components. Units take the Pebble approach to base compilation on function definition and application to model component definition and evaluation. However, Flatt and Felleisen completely disregard loading (an essential part of dynamic linking). They consider the verbosity of their text representation for Units prohibitive for real-world applications: imports need to describe the full types of imported elements. MzScheme [MzS04] implements Units, but the system itself is not composed of Units. Flatt and Felleisen do not define a language-level representation for Units, or language-level linking programs.

8.7 Summary and Validation

This chapter has developed a viable representation for compiled components to solve the classical problem of modular programming languages, albeit with mechanisms different from the traditional model: Compiled components make use of higher-order functions, and componentized applications are linked at startup.

Validation by Implementation. This chapter makes the claim that the component system can be instantiated for any language, statically-typed or dynamically-typed, that supports higher-order functions. This claim is substantiated by two implementations in actual full-fledged programming systems:

Mozart. The first instantiation of the component system took place for the dynamically-typed programming language Oz, in its implementation Mozart. Oz has an open, well-defined language-level representation for components [HK04] and builds compilation and linking on function definition and function application. Oz components (called *functors*) are syntactic sugar only, without requiring any primitives. Mozart has an ad-hoc approach to booting instead of the approach described here.

Alice. The second instantiation of the component system took place for the statically-typed language Alice ML, which is derived from Standard ML. The SEAM virtual machine for Alice performs boot-time linking as described here; however, the boot linker omits link-time type-checking, for a reason: Alice implements type-checking in Alice ML itself (component signatures are represented as Alice ML data structures), and the type-checker is available on *after* the boot linker has completed. This is acceptable, as boot-time linking in Alice is only applied to system components (components deployed with the system), and consistency of system components can be assumed. All subsequently loaded components are linked with type-checking, using the mechanisms described in Chapters 9 and 10.

Chapter 9

Components with Lazy Dynamic Linking

This chapter describes a principled approach to a component system with first-class components, lazy dynamic linking, and sandboxing security. The component system is applicable to programming languages with higher-order functions, lazy futures as described in Section 2.1.6, and packages as described in Section 2.3.

Motivation. The component system presented in the previous section limits its linking to take place eagerly and fully at the time the virtual machine is started up. In this chapter, the limitations are lifted to provide the following features:

- Components are loaded and linked only when they are actually first used. This is commonly known as *lazy linking*.
- Components other than those the root component directly or indirectly depends on can be dynamically discovered and linked at any point while an application runs. This is called *dynamic linking*.
- Assembly of components, also called *system configuration*, can be programmed in the high-level programming language. This provides for maximum flexibility regarding policy and mechanism of linking: application programmers can write their own dynamic linkers to run as substitutes to or side-by-side with the system-provided dynamic linker.
- Components can be evaluated multiple times in a single program, in different contexts. This is also called *instantiating* components.

Overview. The structure of this chapter, depicted in Figure 9.1 (arrows indicating dependencies), is as follows.

First, OpenL's components for modularization and separate compilation from the previous chapter is extended with lazy linking in Section 9.1. The

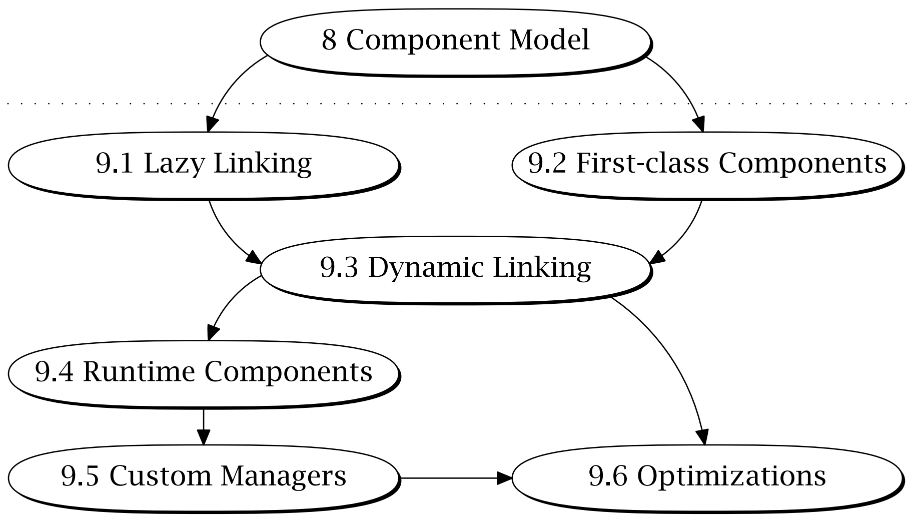


Figure 9.1: Structure of this Chapter.

components from the previous chapter are reified into an abstract data type in the language in Section 9.2, which allows for a simple presentation of a *component manager* for late composition and dynamic linking in Section 9.3. Section 9.4 demonstrates how to integrate the component manager into the execution environment and shows how the component-related parts of the runtime system architecture are composed of components themselves. With this background, Section 9.5 discusses instantiating multiple component managers in a single system for, among other goals, obtaining sandboxing security.

Section 9.6 discusses optimizations for componentized programs as introduced by the combination of all of these extensions. Section 9.7 concludes with a summary and an overview of how the concepts have been validated in real systems.

9.1 Lazy Linking

The previous chapter described *eager* linking, where a component is evaluated after all components it depends on have been evaluated. This section provides for *lazy* linking, that is, linking components *at the latest* before they are actually needed. A major benefit of lazy linking is that execution of a program can start before all components have been loaded and evaluated. Startup time may therefore be reduced, as may the memory footprint of an application. As an example, components required by failure handling, seldom-used code paths, or features not used by many users are only evaluated once they are needed, if at all. As another example, consider a command-line tool invoked to print out its usage, which does not need to link the components that perform its actual business logic.

Another benefit of lazy linking is that it allows for cyclic dependencies, which is not possible with eager linking. Furthermore, in Section 9.3.2, it turns out that lazy linking also allows for a simpler realization of dynamic linking.

9.1.1 The Semantics of Lazy Linking

The semantics presented in Section 8.2 explicitly require that components are evaluated bottom-up with respect to the dependency graph. In the presence of lazy linking, evaluation of components in the component graph is actually triggered in a top-down fashion as components are first needed for computations to proceed. Furthermore, evaluation of one component can actually trigger evaluation of a component it depends on before its own evaluation is completed.

Introducing Laziness into the Semantics. The notion of lazy futures introduced in Section 2.1.6 provides a useful definition of what is means for a value to be "needed" by a computation: Constructs and primitives are *strict* in some of their arguments; using a value as a strict argument constitutes a need. To reuse this definition for the need of components, it is therefore sufficient to simply precede every functor application in linking programs with the **lazy** keyword, as introduced in Section 2.2. For instance, the linking program from Section 8.3 becomes:

```
structure Fac = lazy MkFac ()
structure Main = lazy MkMain (structure Fac = Fac)
val _ = Main.x
```

The last line is required to actually cause evaluation of the root component.

Lazy Structure Selection. Previously, the assumption was that structure selection using *longids* of the form `S.x` was strict: Use of such a longid represents a need of structure S. By the translation of import announcements to value declarations with longids, as specified in Section 8.3, this would actually cause programs to behave exactly the same as under the eager linking discipline, since the first thing evaluation of a component would cause would be evaluation of all components it imports. One solution is to make longids non-strict. Then, dependee components are evaluated only when the value of an imported identifier is first needed, as opposed to when the imported identifier is first referenced.

To accommodate this change, the last line of the linking program above needs to be modified to read:

```
val _ = await Main.x
```

Error Handling. As with eager linking, a component fails to return a namespace when its evaluation raises an exception. In contrast to eager linking, however, a computation may currently be blocking in the middle of execution waiting for the namespace to become bound. Simply halting the program is not acceptable: The computation may need to free resources it has acquired, or recover from the failure for robustness.

It turns out that with lazy linking, a mechanism to handle failure is already built-in, which may not be apparent at first: A lazy longid, when needed, spawns a computation that, in turn, needs the namespace returned by the corresponding lazy functor application, to select a program object from it. Recall from Section 2.1.6 that this exception causes the future that is the placeholder for the result of the functor application to become a *failed value*. Similarly, all lazy longids for that namespace become, when needed, failed values. Effectively, the exception raised by evaluation of a component is delivered to *all* components that wish to use program objects exported by the failed component.

To make exceptions caused during evaluation of a component apparent (after all, in normal operation, they may be unexpected), they are wrapped in an `Eval` exception constructor:

```
exception Eval of exn
```

9.1.2 Implementing Lazy Linking

The previous section has shown what changes are needed in OpenL linking programs. L_I linking programs for statically-typed components, as introduced in Section 8.5.2, require the following additional considerations:

- Type-checking needs to be lazy. Lazy linking is supposed to reduce start-up costs, and type-checking is the most expensive part of linking. To achieve this, add the **`lazy`** keyword to every application of `match`.
- `match` needs to be non-strict in its first argument, which means that is has to perform the representation change caused by coercion lazily. This prevents evaluation of the component in the case of a type mismatch.
- As said above, evaluation of components needs to be lazy. This is achieved by adding the **`lazy`** keyword to every application of a component function.
- As said above, exceptions raised by component evaluation need to be wrapped in an `Eval` constructor and re-raised within the **`lazy`** expression.

Applying these changes to the L_I linking program from Section 8.5.2 results in:

```
val E  = lazy fnE #[]
              handle e => raise Eval e
```

```
val D  = lazy fnD #[]
              handle e => raise Eval e
val CD = lazy match (D, sigD, sigCD)
val CE = lazy match (E, sigE, sigCE)
val C  = lazy fnC #[CD, CE]
              handle e => raise Eval e
val BD = lazy match (D, sigD, sigBD)
val B  = lazy fnB #[BD]
              handle e => raise Eval e
val AB = lazy match (B, sigB, sigAB)
val AC = lazy match (C, sigC, sigAC)
val A  = lazy fnA #[AB, AC]
              handle e => raise Eval e
val _  = await A   (* execute the linked program *)
```

9.1.3 Related Work

Unix Shared Libraries. The Unix shared libraries ELF object format supports *lazy binding* with the goal of reducing startup time [Lev00]. Lazy binding is less general than lazy linking as supported by the above approach: All libraries are actually loaded eagerly at startup, but cross-library procedure references are resolved lazily. The *Procedure Linkage Table* contains a stub for every imported procedure. When a cross-library procedure call is first executed, the stub is executed. The stub invokes the run time linker to resolve the procedure reference and patch the call site to point to the actual procedure. Lazy binding is analogous to OpenL lazy longids—with the difference that lazy binding only applies to procedures. Cross-library data references require an extra indirection through the *Global Offset Table*.

Windows DLLs. Microsoft Windows "delay-loaded" dynamically-linked libraries [Lev00] employ a similar technique to the ELF Procedure Linkage Table. The run time linker invoked by the stubs, however, also handles finding and loading the dynamically-linked library in addition to resolving cross-library references—which is why the actual loading can be lazy, as opposed to ELF. On the other hand, this requires locking in the loader, which frequently leads to deadlocking problems [Bru03].

Java. The definitions of both the Java language [GJS00] and the Java Virtual Machine [LY99] cover lazy dynamic linking, but define it only informally. Like in OpenL, lazy dynamic linking is composed of loading, linking, and evaluation (called *initialization* in Java). In addition, Java distinguishes sub-phases of linking, namely verification, preparation, and resolution. All of these are low-level mechanisms neither expressible in the language itself, nor in the terminology of the language at all. This makes it nontrivial to capture the semantics in a formal model, which is the subject of ongoing research (for instance, [DLE03]). In contrast, the OpenL semantics are actually captured at the level of the language.

Java class loading [LB98] can fail in a number of ways, such as by raising `NoSuchMethodException`, which essentially correspond to type mismatch exceptions in OpenL. At the first use of a class, the class's or its parent's class initializer is run, which may also throw an exception. This exception is re-thrown as a `ExceptionInInitializerError`, which is analogous to OpenL's `Eval` exception. In contrast to OpenL, where all clients of a class are equal in that they see the same exception, subsequent attempts to use a class for which initialization failed result in `NoClassDefFoundError` exceptions. OpenL's solution using failed values is considerably simpler and more uniform.

Units. Units [FF98] do not address the question of lazy linking, but are interesting in this context in their way of dealing with recursive definitions (cyclic dependencies). Compiled Units use state (reference cells) to stand in for the imported and exported values of a component. This is at the same time the reason why the Units linker needs to distinguish between the operations of linking and evaluation of a component. These operations can be merged into a single function application in OpenL, making for a simpler design.

9.2 First-class Components

One goal stated in the introduction is to make it possible to write OpenL programs that acquire components and compose them into runnable systems. This means that OpenL programs must be able to manipulate components as values, to load them, reflect their metadata, and evaluate them. To this aim, OpenL introduces a predefined abstract `component` type for first-class components, with the following operations:

```
structure Component :
sig
    type component

    exception Malformed

    val load : string -> component
        (* IO.Io, Malformed *)

    val imports : component -> (string * sign) vector
    val sign : component -> sign
    val apply : component -> package vector -> package
        (* Mismatch *)
    val capture : package -> component
end
```

These operations are all type-safe and can thus be exposed to the application programmer.

9.2.1 Loading Components

Given the name of a namespace, `load` finds a corresponding compiled component file, adds the contained code to the code area as described in Section 8.4, and constructs and returns a new `component` instance. A component instance is a tuple of metadata and a first-class functor, which can trivially be represented in the abstract store. If `load` cannot find or read a file, it raises `IO.Io`; if a file is found but is not a valid component file, `load` raises `Malformed`. `imports` and `sign` project the metadata defined in Section 8.5.2.

Note that `load` will be made non-primitive in Section 10.1.

9.2.2 Evaluating Components

`apply` takes a component and a vector of packaged namespaces. These namespaces correspond one-to-one to the dependencies mentioned in the `imports` metadata vector. `apply` processes these namespaces, under the expected and actual types, in the same way as `match` is used by L_I linking programs—in particular, type-checking and coercion are lazy. `apply` evaluates the component (which may or may not trigger type-checks, coercions, and component evaluations, depending on the argument component) and returns the resulting namespace, packaged with the component's type.

Generative Types. In contrast to previous linking programs, which evaluated every component at most once, `apply` can be used to evaluate a component more than once. For generative types, type-checking now deserves a closer look. Assume that a given component C defines an abstract type t. When C is evaluated twice, this effectively defines two distinct abstract types t_1 and t_2. (Note that t_1 and t_2 can actually be structurally different if the definition of t depends on a type passed as an argument to `apply`.) So far, no mechanism has been introduced to make t_1 and t_2 incompatible, which they need to be, lest `apply` break the soundness of the type system. The following proposes a simple solution and outlines its limitations.

A Simplistic Solution. During elaboration, the compiler generates a static type name t_s for every definition of an abstract type t. At run time, every evaluation i of the definition of t requires a new abstract type name t_i to be generated (run-time types already need to fulfill this requirement, to make the package concept sound). One solution to the problem outlined above is to have `apply` operate under a substitution from every static type name t_s

in its argument namespaces to the corresponding dynamic type name t_i. A requirement for a type t stated by an import signature of a component is fulfilled by a substitution $t_s \mapsto t_i$; type equivalence constraints of the form $t_s = u_s$ are fulfilled only if, according to the substitution, both t_s and u_s map to the same t_i. Similarly, `apply` needs to extend the substitution for the type names introduced by the namespace it produces.

Substitutions and Package Signatures. For soundness, it must not be possible to dissociate substitutions from namespaces. In other words, a partial substitution must be stored in the signature that is part of every package. Assuming that the **pack** construct similarly includes a partial substitution in the runtime signature stored in the package it produces, packages obtained through `apply` can interoperate with packages obtained through **pack**.

Limitations. One thing that the substitution scheme above cannot deal with is when import signatures have free type variables. Consider the following component:

```
    import type int from "Fundamental"
    import val f : int -> int from "MyComponent"
    ⟨... use f ...⟩
```

In the simplistic solution above, free occurrences of type variables have to be disallowed. This solution is simplistic, because it severely reduces the expressivity of the component system in the presence of static typing with generative types. This can be mitigated, to some extent, by making fewer types generative: (1) Ubiquitous types, like `int` above, can be placed in the initial environment. They would be treated as constants, as opposed to free type variables. (2) Algebraic datatypes can have structural equality, as opposed to token equality.

Rossberg [Ros06b] proposes a component system for statically-typed languages that covers more scenarios.

9.2.3 Capturing Components

A package constructed at run time can be turned into a component using the `capture` injection function. The resulting component has no imports, the same signature as the argument package, and its component functor returns the structure contained in the package verbatim. Note that the exact same structure is returned, as opposed to a copy. In essence, the `capture` operation essentially allows a package to be used in place of any first-class component.

Applications. When Section 10.1 introduces a way to persist first-class components, the `capture` operation unfolds its full potential. As an example, some component may compute data that is static in the sense that it is only ever read after initialization. The computation may be expensive, but the actual representation of the data may be small. By evaluating the component, capturing the result, and persisting it, the data can be made available to other applications without requiring to perform the computation again.

Another application are mobile agents [FPV98, Section 4.1.4], discussed in Section 11.2.

9.2.4 Syntactic Support for First-class Components

OpenL can easily be extended with a syntax to express first-class components by adding the following expression syntax:

comp ⟨*import announcements*⟩ ⟨*declarations*⟩ **end**

A **comp** expression has type `component` and evaluates to a component. With file-level components, it shares the property that it has imports; with captured components, it shares the property that it can contain references to run-time computed data structures, if the **comp** expression has free variables.

Note that the expressivity of the **comp** expression is superior to both file-level components created by the batch compiler, and captured components. One application is remote execution of agents, using the mechanism described in Section 11.2.3.

9.3 Late Composition and Dynamic Linking

All linking programs described so far had the property that they assumed their free variables were already bound to component metadata and component functors. In other words, the set of components that make up an application had to be fully known at the time of constructing the linking program. This section adds the ability to extend a running application by components discovered at run time.

Section 9.3.1 describes the interface to the *component manager*, the entity responsible for coordinating and fulfilling dynamic link requests. The component manager is built on first-class components, with no additional magic, as Section 9.3.2 shows. Finally, Section 9.3.3 generalizes the component manager to allow for multiple evaluation of the same component.

9.3.1 The Component Manager

To dynamically link a component, one needs to know the name of the namespace it defines, as well as its type. Obtaining a name of a namespace at run time is called *discovery*; since namespace names are simple strings, discovery can be implemented in any number of ways. With this said, the simplest conceivable interface to the component manager is the following:

```
structure ComponentManager :
sig
    exception Eval of exn

    val link : string -> package
        (* IO.Io, Mismatch, Eval, Malformed *)
end
```

`link` takes the name of a namespace, loads and links the component if it has not been linked already, and returns the resulting namespace as a package. It is up to the calling program to unpack the package under a valid type.

Semantics of Dynamic Linking. In an application such as `link` *s*, the component designated by *s* may depend on other components, which `link` would need to recursively link. To be well-defined, an application linked by the dynamic linker must have the exact semantics described in Section 9.1.1. This implies that linking has to be lazy, the same amount of sharing has to be maintained, and cyclic dependencies need to be correctly resolved. In particular, every application of `link` to a given string *s* returns the same package.

The Namespace Table. To this aim, the component manager maintains a *namespace table*. Every time the component manager comes across a reference to a namespace name that is not in the namespace table, it adds an entry to the namespace table mapping the namespace name to a lazy future. A request of the value of this lazy future causes the component manager to load and link the corresponding component. The lazy future is then replaced by either the resulting namespace, or, if evaluation of the component raised an exception, a corresponding failed value.

9.3.2 Implementing the Component Manager

The component manager needs to be thread-safe. Its only state is the namespace table. Internally, the namespace table is represented as a mapping from namespace names to packages. The following operations provide a thread-safe namespace table:

```
val init : (string * package) list -> unit
datatype reference =
    EXISTING of package
  | NEW of package promise
val reference : string -> reference
```

`init` initializes the namespace table with the given contents. `reference` is the only accessor to the namespace table; it is atomic. If the component table already contains a package *package* for the given namespace name, `reference` returns a the existing entry as `EXISTING` *package*. If not, `reference` creates a promise *p*, creates an entry in the component table with the value `future` *p*, and returns `NEW` *p*. (In other words, for a given string *s*, the first application of `reference` *s* returns a `NEW` result, and every subsequent application of `reference` *s* returns an `EXISTING` result.) These operations can easily be implemented in plain L and are not shown here for conciseness.

Then, `link` can be defined in OpenL as follows:

```
fun link name =
    case reference name of
        EXISTING package => package
      | NEW p =>
            let
                val package =
                    lazy (eval (Component.load name)
                          handle e => raise Eval e)
            in
                fulfill (p, package); package
            end
and eval component =
    Component.apply component
        (Vector.map (fn (name, sign) => link name)
                    (Component.imports component))
```

Dynamic Typing. For dynamically-typed languages, the above definition is identical but for the simplification that there is no need for packages: all occurrences of packages can be replaced by the corresponding namespaces, and `apply` does not need to perform type-checking.

Properties. The implementation has a number of important properties. It works for cyclic references. It is fully thread-safe. Exceptions thrown by component evaluation result in a failed value due to the definition of **`lazy`** and can therefore be handled by all clients.

9.3.3 Evaluating Components Multiple Times

One requirement stated in the introduction is the possibility of evaluating a component multiple times. This is not possible with the component manager interface presented above, unless the same component exists under multiple namespace names—once a component has been evaluated for a namespace name, subsequent applications of `link` retrieve the namespace from the namespace table instead of re-evaluating the component.

However, the above implementation of `link` already defines an `eval` operation that exactly fulfills the requirement of evaluating a component without entering it into the namespace table. `eval` just needs to be exposed from the `ComponentManager` structure as follows:

```
val eval : component -> package
```

Section 9.5.1 extends the concept to allow for evaluating a component in various contexts.

9.3.4 Related Work

Configuration Languages. The need for making linking of components configurable has been recognized as early as the 1970's. Mesa [LS79] had one of the first configuration languages for defining the linking of separately compiled components. Mesa's configuration language is separate from the programming language.

The approach taken in the context of Pebble [Bur84] defines a single functional programming language that is used to express both the components themselves and the linking of these components. This is similar to what this chapter does with first-class components: OpenL itself becomes the configuration language.

Unix Dynamic Libraries. The Unix `dlopen` library supports dynamic linking. `dlopen` interfaces to `ld.so`, the dynamic linker. There is no easy way for application programmers to substitute their own version of `dlopen` and `ld.so`, since the latter makes use of platform-dependent low-level mechanisms for its operation. In other words, application programmers have little control over the actual linking—in contrast to OpenL, which makes linking entirely programmable at the level of the language.

Module Management as a System Service. Bracha et. al. [BCLO93] propose a system service, running as a separate process, that provides for module management. Their understanding of module management is similar to the goals of this work's component system. The module management service defines an abstract data type for modules, which can be instantiated for many languages. In essence, their system replaces the system's dynamic linker and provides application programmers with more control over system configuration. Their work delivers only a model, not a practical implementation: The authors themselves say, "many [...] assertions remain largely unsubstantiated claims." In contrast, OpenL delivers, in its incarnations in Mozart and Alice, viable implementations proving all of its claims.

Units. Units [FF98], like OpenL components, are first class. They also provide a linking operation to make program composition programmable in the high-level language. As noted in Section 8.6, they do not address the problems of dynamic loading and deployment.

Java Class Loaders. Java's *class loaders* [LB98] allow to load and link components explicitly at run time. Classes obtained in this way are less convenient to use than statically referenced classes: they are returned as instances of the `java.lang.Class` class [Sun03b] and can only be instantiated through an expensive and cumbersome reflection method instead of the built-in **new** operation. Only by casting the instance to a statically-known base class or interface can it be used by built-in method invocation. This is an artifact introduced by Java's type system, and OpenL does not suffer from such a problem.

Classes can use instances of classes defined by other class loaders. Java has a similar typing issue as the one discussed above regarding type compatibility across class loaders. Instead of using substitutions, Java identifies each type by a pair of the class loader identity and the symbolic name of the class. Interestingly, this issue was overlooked in early versions of Java, compromising soundness [DFW96], and needed to be retrofitted into the virtual machine definition.

Type-safety. OpenL's dynamic linking only hints at typing issues with dynamic linking, by delegating them to the largely-unspecified type language. Dean [Dea97] proposes a mechanism to prove type consistency in a system with dynamic linking. Duggan [Dug02] describes how to maintain type abstractions provided by the module language in the context of dynamic linking; Sewell [Sew01] pursues the same goal in the context of distribution. Rossberg [Ros06b] describes Alice's approach to these problems, which are outside the scope of this work.

First-class Environments. In the context of Scheme, there has been some research about how to provide minimal language extensions to support first-class modules [Jag94, QR96], taking the very different approach of *reified environments*. OpenL's syntax for first-class components shares with their approach the goal of allowing parameterization of a program with respect to an unordered set of variables, as opposed to functional abstraction. Their focus is, however, on modules as units of reuse, as opposed to units of compilation and deployment—analogous to how Standard ML modules differ from OpenL components. OpenL packages and captured components have some similarities with such first-class environments.

9.4 Architecting the Runtime from Components

The preceding sections introduced `Component` and `ComponentManager` as predefined structures. This section takes a closer look at how a system architecture can actually make these structures themselves available to the programmer as components.

To this aim, Section 9.4.1 introduces the distinction between built-in primitives, the system namespace, and runtime components. This distinction allows the programming system to be composed from components, and at the same time is an essential precondition for sandboxing security as presented in Section 9.5.3. Section 9.4.2 revises the virtual machine from Section 8.4 to start up a componentized runtime and run user applications from within a component manager.

9.4.1 The System Namespace and Runtime Components

Chapter 3 considered the execution environment to consist of a virtual machine that could interpret a code representation of programming language constructs, and of a number of primitive operations. OpenL distinguishes two groups of primitive operations:

Built-in Primitives support built-in language types; they include, for instance, arithmetic on integer types, vector and array operations, and the operations on futures from Sections 2.1.5 and 2.1.6. The defining property of a built-in primitive is that it has an effect only on the abstract store (see Section 3.1), but no access to or effect on the program's environment (disregarding memory operations performed by the implementation of the abstract store).

System Primitives provide the remaining operations not expressible in the programming language: operations for accessing properties of the current process, such as environment variables and command line parameters; accessing the file system; creating and communicating with other processes; and performing network operations.

Built-in primitives are made available by the compiler through the initial environment as described in Section 8.1. The virtual machine supports a special instruction to apply a built-in primitive; in other words, references to the initial environment at the source level translate to direct references to the built-in primitives from generated code.

In contrast, system primitives cannot be directly referenced from code. Instead, the virtual machine places all system primitives into the so-called *system namespace*, which is constructed at startup, and for which there exists no persisted representation. The boot linker's namespace table (see Section 8.4) initially consists solely of the system namespace, registered under a designated namespace name.

The *runtime components* are predefined components that are part of the system and that can be implemented in OpenL itself. Runtime components import the system namespace and provide a type-safe high-level interface to system resources.

The constituents of the architecture are depicted in Figure 9.2. Above the dashed line is shown a user component that makes use of runtime components and built-in primitives. Below the dashed line are the constituents that are part of the programming system.

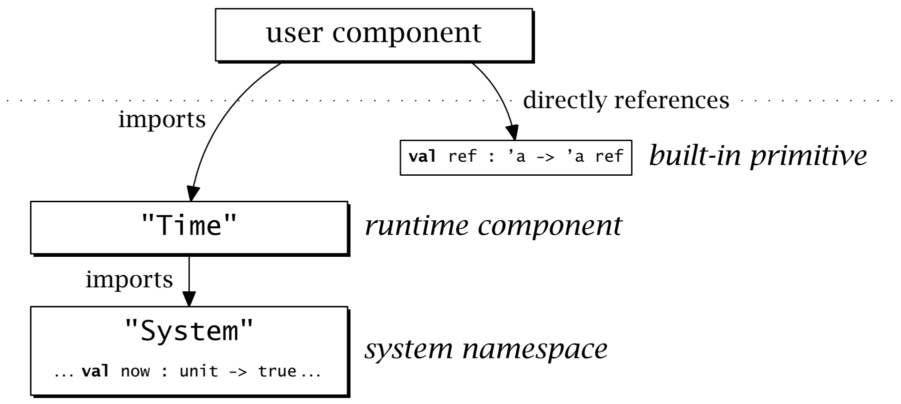


Figure 9.2: Componentized Runtime Architecture.

9.4.2 Booting the Virtual Machine

The OpenL virtual machine already supports eager linking on booting (see Section 8.4). This mechanism can be used to link a component graph below a *boot component* that creates a component manager and uses it to link a user application's root component.

In contrast to the root component, which is executed just by evaluating it, the boot component has to be passed parameters from the boot linker—namely the boot namespace table, lest the components linked by the boot linker be evaluated more than once. After evaluating the boot component, the boot linker therefore expects the resulting namespace to define a function `boot`, which it applies to the boot component table:

```
val boot : (string * package) list -> package
```

`boot` creates the component manager and populates its namespace table according to its argument. It then links the root component using the component manager's `link` operation, and returns the resulting package. The virtual machine then requests the namespace contained in the package—which is not in general expressible in OpenL itself: Requesting just the package does not, because of laziness, evaluate the component.

Accessing the Component Manager. The component manager itself must be made accessible to the application so that the latter can issue a dynamic link request. To this aim, `boot` needs to create an entry in the namespace table for a namespace containing the component manager.

Implementation. The following is an implementation sketch of one way to define a boot component:

```
import structure Component from "Component"
⟨definitions of init and reference⟩
fun boot bootNamespaceTable =
    let
```

```
        structure ComponentManager =
            struct
                exception Eval of exn
                fun link name =
                    ⟨definition of link⟩
                and eval component =
                    ⟨definition of eval⟩
            end
        val package =
            pack ComponentManager : ⟨...⟩
        val rootComponent =
            List.hd (CommandLine.arguments ())
    in
        init (("ComponentManager", package)::
              bootNamespaceTable);
        ComponentManager.link rootComponent
    end
```

9.4.3 Related Work

Heap Images. Many systems, in particular interactive Lisp, Scheme, and ML systems, boot their runtime from *heap images*. When the system is built, ad-hoc bootstrapping mechanisms are used to start a system with an empty heap and an initial environment consisting of only built-in and system primitives. This system is then enriched with the runtime components by compiling and evaluating source files. At the end of this initialization procedure, the contents of the heap is dumped to a file called a *heap image*. Thereafter, every time the user starts the programming system, a heap image is read back into memory at boot time, whereby the runtime environment is made available. Often, this mechanism is also the only possibility for application programmers to produce executable applications—after compiling the application code into an existing heap image, application programmers produce a new heap image that they ship to customers.

Heap images do not provide flexible customization, beyond providing multiple heap images at system build time. Multiple heap images also cannot share any data—any data contained in multiple images must exist multiple times on disk. Finally, each process can only load a single heap image, determined at startup time. While it may be possible to dump a subset of a running system's heap, there is no possibility of linking this subset into another system's heap. In contrast, OpenL provides for all of these.

Java. Exposing the runtime as language components is a common approach. Java, for instance, exposes all runtime system components as classes. In contrast to OpenL, however, Java requires a number of ad-hoc solutions to deal with chicken-and-egg problems and circular dependencies at startup time—for instance, the `java.lang.Object` and `java.lang.`

`System` classes are required to parse class files (Java's externalized components), but their own definitions reside in class files. Booting such a system is significantly more complex than OpenL's approach, as is exemplified by that fact that the approach taken for the Jalapeño virtual machine [AAB$^+$99] is simple compared to other Java virtual machines, but significantly more complex than OpenL's approach.

9.5 Custom Component Managers

There are a number of scenarios that cannot be covered by a single predefined component manager:

- The namespace table only ever grows, and no namespace is ever removed from it. This precludes unloading of components.
- Components may be obtained from untrusted sources, such as "applets" originating from a Web page and running in a Web browser, or Web server applications uploaded by customers of an Internet Service Provider. To limit the damage these components can cause, they may need to be evaluated in restricted contexts.
- Components imports may be used as a mechanism for parameterization, as opposed to just expressing static dependencies for a modularized program. An application may need to evaluate such a component multiple times, in different contexts, to make use of different instantiations of the service.

Richer Predefined Component Managers. Since component managers are defined by ordinary OpenL functions, application programmers can obviously implement their own customized initial namespace table population, strategies for locating components, and loading policies. Many scenarios can, however, be covered by just a richer definition of the predefined component manager. Consider the following functor:

```
functor MkComponentManager
    (val initialNamespaceTable :
         (string * package) vector) :
sig
    exception Eval of exn
    exception Collision

    val link : string -> package
        (* IO.Io, Mismatch, Eval, Malformed *)
    val eval : Component.component -> package
        (* Mismatch *)
    val enter : string * package -> unit
        (* Collision *)
        val namespaceTable :
            unit -> (string * package) vector
    end
```

This definition differs from the `ComponentManager` structure from Section 9.3.1 as follows:

- The definition is exposed as a functor, and can therefore be applied to obtain multiple component managers with distinct namespace tables.
- The definition is parameterized over the initial namespace table contents.
- `enter` adds a new, possibly dynamically-created namespace to the namespace table even after initialization of the component manager.
- `namespaceTable` returns an existing component manager's namespace table, from which new component managers can be created.

The following sections sketch how to realize the above scenarios with this abstraction.

9.5.1 Component Managers for Multiple Evaluation

Evaluating a single component *C* under multiple contexts becomes possible through the following steps: Retrieve a namespace table from an existing component manager; modify it to establish contexts; create new component managers from the modified contexts; and link *C* in all component managers. The types defined by namespaces obtained from the initial component managers will be compatible across all component managers, while abstract types defined by *C* (and possible dependencies that were not present in the initial namespace table) will be distinct.

9.5.2 Component Managers for Component Unloading

Unloading a component after having made use of its functionality becomes a matter of creating a component manager from the namespace table of an existing component manager, linking a component, and dropping all references to the new component manager. As values from the namespace are no longer used, the resources occupied by the component manager and the namespaces it created can be reclaimed by the garbage collector. This is by virtue of the fact that component managers are ordinary language objects.

No explicit unloading operation is required, which preserves the property that a component manager's namespace table can only ever monotonically grow. To program against this declarative idiom is less prone to error than programming against an operational idiom: If used in the way described above, it is guaranteed by construction that all dynamic type names t_i obtained from a component manager for an abstract type t agree.

9.5.3 Component Managers for Sandboxing Security

The idea of *sandboxing security* is to execute an untrusted application in a so-called *sandbox*, an environment that allows only controlled access to system resources. As described in Section 9.4.1, all operations that provide access to system resources reside in the system namespace, and system primitives cannot directly be referenced from code. Establishing a sandbox therefore means that untrusted applications must not be given references to the system namespace or to runtime components.

Solution. OpenL makes a simple solution possible: One can create a *restricted component manager* by applying the `MkComponentManager` functor to a namespace table with the following properties: (1) There is no entry for the system namespace. Since there exists no persisted representation for the system namespace, the only way to access system primitives is if there is an entry in the namespace table. (2) For every runtime component, there is an entry that maps its namespace name to a restricted implementation that limits or controls access. For instance, a restricted version of a runtime component providing file system access could have all of its operations throw exceptions, or allow access to files under a controlled directory only (also known as "isolated storage").

To leave its sandbox, a component has to gain access to a system primitive. Components evaluated by a restricted component manager can only access system primitives that the restricted runtime components let them access.[1]

Limitation. The initial namespace table of a restricted component manager represents a *blacklist* of the components an untrusted component should not be allowed to use unrestricted. As often with blacklists, this has the potential of introducing a maintenance and versioning problem that can lead to vulnerabilities: Imagine a newer version of the runtime that defines more runtime components. A sandbox implemented against a previous version of the runtime would not restrict access to the new runtime components. Separating logical and physical component names as introduced in Section 10.2 is one solution to this problem, effectively providing a whitelist instead of a blacklist.

[1]Obviously, this relies on the assumption that one can verify that the component, even though it was obtained from an untrusted source, is well-formed and that its metadata is consistent with its code. This lies outside the scope of this work.

9.5.4 Related Work

Capability-based Security. Sandboxing as presented above amounts to a combination of capabilities, first introduced by the Hydra operating system kernel [LCC$^+$75], and namespace management [WBDF97]. A *capability* is a token that gives the possessor permission to access an entity [Lev84]. *Namespace management* stands for providing different component implementations, depending on who wants to access them.

Java. Java pioneered sandboxing in an industrial-strength, general purpose programming language. Sandboxing was first introduced to secure the host system from applets running in a Web browser. Java's sandboxing security is achieved through a combination of class file verification (ensuring that the code cannot forge references) and security managers, which allow or deny requests from class loaders to create specific classes. Application programmers can define custom class loaders and security managers. Reportedly, these mechanisms are difficult to master [DP02].

Microsoft .NET. The Microsoft .NET runtime has a concept of *application domains* [TC301b, Mic03e], originally introduced for ASP.NET to support efficiently running multiple server-side Web applications in a single server process. Application domains provide for isolation with respect to accesses and faults of multiple applications similar to having multiple processes, but without the process boundary. At the same time, they serve to unload unused assemblies (.NET's term for components). In other words, they share some goals with OpenL component managers.

Every type and every object lives in an application domain. Every newly created application domain is initially empty (at least conceptually—for efficiency, every application domain shares the same set of system-defined ubiquitous assemblies). Each Application Domain can resolve the same component name to a different component implementation, based on configuration of the Application Domain. Application domains cannot immediately share references to the same type instances; instead, all cross-application domain invocations have to use *remoting*, which marshals arguments using serialization. Such application domain crossings are subject to the same limitations, and therefore protection, as cross-process accesses. Compatibility of types across application types depends on the user-defined underlying serialization policy.

While application domains aim to solve a larger problem than component managers (for instance, application domain termination causes forced unwinding of all threads running in the application domain—a concept not supported by component managers), the mechanism is significantly more heavy-weight than OpenL's for the common goals.

9.6 Optimizations

In general, many optimizations rely on inferring properties and identities of objects referenced by variable occurrences. For example, if, in a function application, the applied function is known statically, a compiler can compile the application as a first-order call instead of a general higher-order call: Since it can exploit known implementation details of the callee (up to inlining the callee), much more efficient code can be generated.

In a system with dynamically linked components, the actual objects referenced across components only become known at run-time. A naïve implementation therefore loses many opportunities for optimization; more so in applications that have a fine-grained component structure in which cross-component references and calls are frequent. This section proposes ways to recover the most important optimizations.

Overview. Section 9.6.1 describes a simple optimization to minimize the number of futures introduced by lazy longids. Seen on its own, the effect is small, but it plays an enabling rôle with respect to picklability of abstractions, and for the effectiveness of later optimizations. Section 9.6.2 proposes to statically bind to core library components, to recover first-order applications of frequently-used primitive operations. When components can be evaluated multiply in different contexts, this means that functions in the component cannot even fully optimize first-order applications of other functions defined in the same component—this is recovered by a technique proposed in Section 9.6.3. If all else fails, a last chance for optimization is doing late optimizations in a run-time compiler, as discussed in Section 9.6.4.

9.6.1 Hoisting Lazy Structure Selections

A naïve implementation of non-strict longids is to translate every OpenL longid `S.x` to the L_I intermediate language expression **`lazy`** `#x S`, as opposed to `#x S`. In other words, every longid would create a lazy future at run time, increasing generated code size and memory footprint proportional to the number of longids in the program. Instead, the compiler could hoist all these lazy selection expressions to the program point just after where the selected structure is introduced into the scope (be it a structure declaration, a signature coercion, an import announcement, a functor argument, or the result of a functor application). In other words, multiple selections from a single structure then appear in batches, lending themselves to common subexpression elimination—leading to creation of fewer futures in the system, and possibly earlier elimination of those futures.

Semantics. Note that in the presence of pickling, this transformation has an observable effect, since pickling traverses the closures of functions. A function whose body references a longid would, before the transformation, reference the whole structure from its closure, which could have a large pickled representation or even not be picklable at all due to embedded resources. After the transformation, such a function would reference only individual items from the structure, and pickling would not consider parts of the structure that are not actually used from the function.

9.6.2 Compile-time Cross-component Optimizations

Typical programming systems provide as part of their library a number of operations on built-in data types that are so fundamental that they are applied at a high frequency by all but the most contrived programs. Such functions are likely to never see their definition changed, and are therefore termed *ubiquitous* functions in the following. Ubiquitous functions are provided by predefined components. If the compiler treated these like every other component, this would mean that every one of the frequent applications of the ubiquitous functions would be compiled into a higher-order call. The goal of this section is to optimize applications of ubiquitous functions (more generally, uses of arbitrary ubiquitous values).

Solution. To this aim, assume that the compiler be able to *statically bind* not only against built-in primitives, but arbitrary functions, in the sense that the code for these functions would be included in every compiled component, and references to them hard-wired or even inlined.

To provide the optimization, one must identify the ubiquitous functions and define them in a single runtime component. The compiler would, on startup, evaluate this component and use the resulting namespace as the initial environment (remember that until now, the initial environment was assumed to consist of the built-in primitives only). The compiler shall in general statically link against all entities contained in the initial environment.

Trade-offs. Since every ubiquitous function is potentially present in the system many times due to it being part of many components, the memory footprint is likely to grow as more functions are made ubiquitous, possibly outweighing the advantages of static linking. Also, since the code of ubiquitous functions is included in the compiled component, ubiquitous functions may contain no references to system primitives (directly or indirectly). Note also that by design, ubiquitous functions are not versionable and cannot be modified once deployed.

9.6.3 Link-time Intra-component Optimizations

To enable multiple evaluation of components in different contexts, every component body is translated to a functor. This means that every declaration, even though it may syntactically appear at the top level of a component body, is effectively a nested declaration. In other words, even references within a component to objects declared by the component can no longer be translated to first-order references. For example, consider the following component:

```
fun f () = ...
fun g () = ... f () ...
fun h f' = ... f'() ...
```

The application of `f` within `g` would be as expensive as the application of `f'` within `h`, even though much more is known statically about `f`. This is addressed by the next technique.

Solution. The idea is to compile a component as if it was part of a monolithic program, in other words, to assume that all declarations are top-level declarations—with the exception that the compiler would produce a special representation for first-order intra-component references. Additionally, it would mark the produced functor to be *copy-on-apply* (much like a memory management unit's copy-on-write flag on virtual memory pages). `apply` would always duplicate a functor's code before applying it, and hard-wire the first-order references in the copy. The original code can be viewed as a template, and the copies as instantiations. In essence, this means trading cost of component instantiation (both in terms of space and time) against efficiency of executing the code in the component's declarations.

Limitations. Due to the need for a special representation that lends itself to instantiation, the optimization will only apply to an ad-hoc set of entities. In Mozart/Oz, for instance, first-order reference optimizations are only performed for the following entities:

Functions. Scheidhauer [Sch98] has shown that optimization of first-order applications is critical for performance, as the majority of function applications in typical programs is first-order.

Names. Oz names are unstructured values only consisting of their identity, not unlike argument-less exception constructors in OpenL. Oz uses names for protection, in particular of private class members. Calls of statically-bound methods can be optimized only if the name is hard-wired in the code, which necessitates the above technique.

In Mozart, this optimization recovers most of the performance of comparable monolithic programs for programs that are subject to *coarse-grained* componentization.

9.6.4 Link-time Cross-component Optimizations

Even in the presence of copy-on-apply, every application of a function imported from another component remains equivalent to a higher-order application. This section proposes an optimization for cross-component applications, to reduce the performance impact on programs with fine-grained componentization.

Solution. The idea is to exploit the run-time compiler present on many virtual machines. As opposed to a batch compiler, the run-time compiler runs *after* linking, whereby it can actually resolve object references, and generate more first-order references. In other words, the intermediate representation of functions is unchanged, but the code generated at run-time can be specialized with respect to the component's imports.

Limitations. Due to lazy longids, it often happens that the run-time compiler can only resolve object references to lazy futures. The run-time compiler could now actually request the lazy future (as is, in fact, done in Java and Microsoft .NET), with an observable effect, or schedule the function for dynamic recompilation. The latter could, for instance, be achieved by inserting code to this effect right after every construct that requests a lazy future from the function's closure.

9.6.5 Related Work

Most existing work on cross-module optimizations is based on propagating static information across compilation units [Blu97] or across (possibly higher-order) module boundaries [Sha98], or on static interpretation of module constructs [Els99]. By definition, propagating static information establishes strong dependencies between compiled program fragments across which optimizations have been applied, and static interpretation assumes that the linking structure of a set of components is known at compile time. Both of these properties are contrary to the whole idea of components, which should be weakly dependent on one another in the sense that components should be interchangeable, and that the linking structure is determined at run time.

9.7 Summary and Validation

This chapter has presented a principled approach to lazy dynamic linking. The semantics of lazy linking, handling of errors caused by lazy linking, and implementation of the lazy linker are all reduced to the single concept of futures. Simple operations, based on packages to obtain type-safety, are identified for first-class components and allow the definition of dynamic linkers in the programming language itself. This makes linking policies and system configuration completely programmable, which, for instance, allows application programmers to employ components not only for structuring and extensibility, but also as a parameterization mechanism. The additional distinction of built-in versus system primitives allows to express sandboxing security.

Validation. The proposed solutions have been validated in the context of two systems, Oz/Mozart and Alice/Seam.

Lazy Linking is provided by both Mozart and Alice. Non-strictness of accesses to imported components is supported differently by both systems: Alice ML makes all longids lazy as described; Mozart creates lazy futures only for identifiers explicitly enumerated for a namespace name in the import announcement.

Component Managers are defined both in Mozart and Alice. Both allow the definition of restricted component managers for sandboxing security by virtue of distinguishing between system primitives and built-in primitives, but neither provides one as part of its standard libraries.

First-class Components with captured components are both in Mozart and Alice. Oz has syntactic support similar to the one proposed in this chapter; Alice provides only captured components. Oz does not coerce imported namespaces to the given signature (which solely consists of an enumeration of identifiers), but makes this fact transparent to application programmers by disallowing first-class access to namespaces for which a signature is given. Due to its more ambitious type system, Alice cannot use the exact primitives proposed above, and builds its component managers completely on top of unsafe primitives instead.

Component System Optimizations are provided both in Mozart and Alice. In particular, both define ubiquitous functions, called the *base environment* in Oz and the *core library* in Alice. In terms of link-time optimizations, Mozart implements only the proposed copy-on-write technique, while Seam implements optimizations based on the runtime compiler as described.

Chapter 10

Component Deployment

This chapter shows how the OpenL component mechanism can provide in a simple and high-level way important services for component deployment. *Deployment* is the "process whereby software is installed into an operational environment" [Sun03a]. Specifically, this chapter addresses the following requirements:

Programmable Component Acquisition. All services for component acquisition, including resolving component addresses and loading of components, must be expressible in the high-level language, without any specific support from the underlying runtime system. Specifically, it must be possible to internalize components from arbitrary byte streams. This makes component acquisition fully programmable by application developers.

Component Persistence. Whether they have been created statically by the batch compiler or dynamically by capturing, components must be externalizable to arbitrary byte streams, provided they only reference data that has external representations. This allows applications to persist parts of their run-time computed data *and* architecture.

Component Interchange Format. The byte stream representation must be an open format, such that components can be produced by non-OpenL programs. This makes OpenL components be a versatile data interchange format.

Versatile Default Mechanisms. The runtime must provide standard mechanisms for component acquisition that incorporate networked file systems defined by standard Internet protocols such as HTTP [FGM+99].

Hiding. For safety and business reasons, it must be possible to deploy applications with a component granularity different from that used during development. Component boundaries internal to the system being deployed must be hidden from clients of the system.

Overview. Section 10.1 reduces externalization of components to pickling, thereby defining a portable file format for components and enabling data components. Furthermore, it investigates the ramifications of integrating pickles and components. Section 10.2 proposes to use Uniform Resource Identifiers (URI) [BLFM98] to represent component names, allowing for principled component addressing and integrating the networked file system into component acquisition. *Localizers* provide a flexible mechanism for locating components at run time and for decoupling static program structure from link-time binding decisions. Section 10.3 presents *bundling* to enable hiding at the component level, making it possible to force specific linking decisions statically, and allowing for components to be deployed at a more coarse granularity than that used during development. As a side-effect, the overhead incurred by link-time type-checking is reduced and internal components are made inaccessible to external components. Section 10.4 summarizes the results. All presented features have been implemented in Mozart/Oz and Alice, thereby validating the proposed techniques.

10.1 Representing Components as Pickles

The previous chapters defined the syntax and semantics of components, and demonstrated how to implement eager static linking and lazy dynamic linking. Representation of components on disk was not deeply looked at. This section proposes to use pickling to obtain a representation of components on files. Section 10.1.1 tackles the technical challenges of applying pickling to components; Section 10.1.2 describes how to make the format of pickled components open. Sections 10.1.3 and 10.1.4 develop the application programmer's interface to loading and saving components from and to pickles, respectively. The ramifications of unifying pickles and components are discussed in Section 10.1.5.

10.1.1 Making Components Picklable

Pickling is defined on a subset of L_I data structures. To apply pickling to OpenL components, their existing first-class representation must be expressed in this subset. Chapter 8 decomposed OpenL components into the following constituents:

Export Signature. Every component contains metadata describing the signature of the namespace it computes, which is a single structure type instance. Section 8.5.2 already mandated that all run-time types have an L_I representation. Types are terms, and terms in general are representable by the picklable subset of L_I.

Imports. The metadata describing component dependencies is defined to be exactly the value returned by the `imports` operation presented in Section 9.2. This value is a vector of pairs of strings and types, all of which are picklable.

Component Body. The body of a component is translated to a functor, as introduced in Section 8.3. In L_I, functors become functions, and L_I functions are picklable if all data structures in their closure are picklable (remember that code is picklable).

These constituents can be packaged as an L_I tuple. Taken together, this means that:

- all components produced by the batch compiler are picklable, as their closure contains built-in primitives only (see Section 9.6.2), which are picklable by definition; and
- captured components are picklable if and only if the captured value is picklable. Note that Section 9.4.1 requires that system primitives not be picklable.

Note also that Section 9.6.1 described how hoisting of lazy longids ensures that only the used parts of structures are part of the transitive closure of abstractions.

10.1.2 Making the Format of Pickled Components Open

Section 7.3 already discussed how pickles in general can and should be represented in an open format. To make pickled components be an open format, the way how components are represented in L_I must be part of the definition. This means that (1) the type language and operations of types must be defined; and (2) the names and semantics of built-in primitives must be defined.

(1) and (2) together encompass, in particular, the `apply` operation. Note that the actual linkers need not be defined, by virtue of being fully defined in terms of `apply` and the other primitives.

10.1.3 Loading Components from Pickles

The single extension that is needed to obtain a component from its pickled representation is the following `unpickle` primitive:

```
val unpickle : Word8Vector.vector -> component
    (* Malformed *)
```

The `unpickle` primitive verifies that the argument byte vector represents a well-formed pickled component[1], in which case the L_I data structure represented by the pickle is constructed and returned. This operation is type-safe and can, therefore, be made available to the application programmer in the `Component` structure from Section 9.2.

With this operation, the operation `load` from Section 9.3 becomes non-primitive:

```
fun load filename =
    let
        val instream = BinIO.openIn filename
        val vector =
            BinIO.inputAll instream
                handle e =>
                    (BinIO.closeIn instream; raise e)
    in
        BinIO.closeIn instream;
        unpickle vector
    end
```

10.1.4 Persisting Components to Pickles

The operation inverse to `unpickle` is `pickle`, which takes a first-class component and pickles it, returning the result as a byte vector:

```
exception Sited
val pickle : component -> Word8Vector.vector
    (* Sited *)
```

In case the argument component referenced non-picklable data, `pickle` raises a `Sited` exception. `pickle`, too, is type-safe and can be exposed in the `Component` structure.

10.1.5 Pickles Versus Components

Note that until now, the only operation that created a component on disk was the compiler. The `pickle` operation adds to this the possibility to persist captured components—which means that persisted components may contain not only compiled components, but, more generally, computed (higher-order) data.

The `pickle` and `unpickle` operations defined on components, together with captured components, subsume pickling of packages. Therefore, a single application programmer's interface is needed for both pickling and loading components, with the added possibility of saving components to

[1]How the primitive does this is outside the scope of this work.

disk. The interface is type-safe, as it builds on components, which in turn are built on packages. Moreover, pickled data can be imported using import announcements, and pickled data can be bundled with components, as described in Section 10.3 below.

10.1.6 Related Work

Java. The Java virtual machine was designed with a documented file format for compiled components [LY99]. Every class resides on its own *class file*. Class files are platform-independent, self-describing, and verifiable. The concept is proven by the many virtual machines that have been implemented to consume class files, and libraries that produce class files.

Class files are loaded by class loaders, which have an operation that takes a byte array and produces a linked class object from it. Unlike in OpenL, there is no first-class representation for an unlinked class. Also, class files are much more restrictive than OpenL pickles: They can contain only constant data of very basic data types and not arbitrary graphs. Object graphs can only be made persistent through serialization. Serialization and class files are kept separate concepts, which is why each of them is less expressive than OpenL pickles.

First-class Components for Java. Duggan's work on dynamic linking of first-class components [Dug02] mentions that loading of dynamic libraries can be based on Java deserialization. However, it does not consider serialization as a mechanism for making dynamically-created components persistent, or examine the ramifications of combining the concepts of components and pickling. Both of these are featured in OpenL.

Microsoft .NET. In Microsoft .NET, executables (EXEs) and dynamic libraries (DLLs) share the same file format (*assemblies* packaged within *portable executable* files), the difference being that EXEs designate a static method as the entry point at which the application launcher is to start execution. Like in Java, assemblies can only contain very limited static data. .NET goes further than Java in that assemblies can be created dynamically using the classes in the `System.Reflection.Emit` namespace [Mic03b]. Such dynamic assemblies can even be saved to disk. Mechanisms as simple as OpenL capturing, first-class syntax, and pickling are not available.

Units. Flatt and Felleisen [FF98, Footnote 6] disregard loading. Pickling could be employed in their approach in just the same way as in OpenL.

Capabilities in Mobile Code Systems. Wallach et al. [WBDF97] describe the complexities of preventing mobile code systems to perform unmediated transfer of capabilities. Mobile code based on pickled dynamic components mitigates this problem: Capabilities are represented as system primitives, which cannot be part of pickles. Since pickling is defined on closed graphs, an abstraction that possesses a capability is itself not picklable—therefore, capabilities cannot be implicitly transferred as part of mobile code. Mobile code has to be parameterized over the capabilities it requires, enabling mobile code hosts to mediate capability transfer: They need to pass capabilities to downloaded code explicitly, or can choose not to do so.

10.2 Locating Components at Run Time

As described, components are referenced by name in import announcements and metadata, and represented on files as pickles. The preceding sections assumed that component names lived in a flat namespace, and that the base file name was always identical to the component name. These assumptions are unrealistic in the face of the following important scenarios:

Naming Components by Function. At the outset, one goal of components was to make system composition configurable at run time, and accessible to the administrator as opposed to the developer. In order to make components interchangeable that are identical in their programmatic interface, clients should be able to use symbolic component names instead of actual component file names.

Selecting Component Variants. A special case of the preceding scenario is selecting, at run time, variants of a component that are or are not instrumented with assertions, for tracing, for coverage data collection, or for profiling.

Component Interposition. A familiar mechanism in object-orientation is *function interposition* [BCLO93], which allows to rebind all references to a function f to an impostor function. The impostor function can make references to f. This mechanism can be used, for instance, for adapting existing compiled code to new data formats or debugging.

Assigning Security Levels. Section 9.5.3 described how sandboxing security required components to be classified according to the level of trust required from clients. Errors in the classification can lead to security vulnerabilities. System extensions can make this mapping hard to maintain if it cannot easily be derived from component names alone.

Integrating Subsystems without Name Clashes. When subsystems are developed independently of one another, there is a possibility that the names of components internal to each subsystem collide. This can make it difficult to integrate these subsystems.

Network Transparency. In general, for example in the case of *applets*, components can be provided on the local computer, or a different computer on the network. In the latter case, they can also be cached locally. Clients should be able to address a component under a single name, irrespective of the network location.

This section introduces a distinction between *logical* and *physical* component names to break the static connections between components, and to enable external selection of connections. It proposes to structure logical and physical component names by use of URI and URL, respectively. *Localizers* map from logical to physical names. Just as writing functor applications allows to make linking decisions in the module language, defining localizers allows to make linking decisions in the component language.

10.2.1 Logical and Physical Component Names

Uniform Resource Identifiers (URI) [BLFM98] and Uniform Resource Locators (URL) [BLMM94] define a standardized syntax for strings with a hierarchical structure. URI and URL are good candidates for logical and physical component names: URI serve as abstract identities, which require an application-defined mapping to interpret them. In contrast, URL have defined mechanisms for obtaining resources from them (in particular, file contents). A *localizer* shall be a partial function from URI to URL. Note that since the set of URL is a subset of the set of URI, the identity function on URL is a localizer.

Using URI as Logical Component Names. URI have a hierarchical structure. The benefit is that this allows to easily define conventions for identifying sets of related components. In particular, all components below a specific node in the hierarchy could be considered to require higher trust; or organization of sub-hierarchies can serve to heuristically avoid name collisions. For instance, OpenL could use URI starting with `x-openl://system/` for all of its system components: The `x-openl` scheme ensures that there are no collisions with any URL or any user-defined component, and the `system` authority indicates that this component requires a high trust level.

URI can be *absolute* or *relative*. A standardized *resolving* function takes an absolute URI (called the *base URI*) and a relative URI, and returns an absolute URI. Relative URI facilitate relocating sets of interdependent entities as a whole, without needing to adapt references internal to the set.

Using URL Instead of File Names as Physical Component Names. URL can express commonly used file system paths by use of the (implied) `file` scheme. In other words, file names can be used as URL. Beyond that, URL define a networked file system: For instance, the Hypertext Transfer Protocol (HTTP) [FGM+99] is used to access resources named by URL from the `http` scheme.

The networked file system defined by URI is not browsable—and there is no need for it to be, in the context of component naming: Component managers only ever need to open and read files given their name. Dynamic discovery of components amounts to viewing component tables as monotonically growing subsets of the mapping from logical component names to namespaces.

10.2.2 Programming Localizers

From the preceding section, it follows that logical component names and component instances are related by partial functions as follows:

$$\text{absolute URI} \xrightarrow{\texttt{getUri}} \text{absolute URL} \xrightarrow{\texttt{getUrl}} \text{pickle} \xrightarrow{\text{unpickle}} \texttt{component}$$

This section describes how these functions are made available to the OpenL application programmer by library structures. Observe how this functionality is fully implemented in the high-level language.

Acquiring Resources from URL. The `getUrl` operation examines the argument URL and uses, depending on its scheme, network protocols or file system functions to access the resource in binary form, if it exists:

```
type url = string
val getUrl : url -> Word8Vector.vector
    (* IO.Io *)
```

In practice, `getUrl` would be extensible by allowing registration of custom handlers for specific URL schemes.

Localizing a Resource from a URI. A localizer is a simple function. The `getUri` operation is parameterized over a localizer, which it uses to turn a given URI into a URL, and acquire the resource:

```
type uri = string
type localize = uri -> url option
val getUri : localize -> uri ->
             Word8Vector.vector option
```

`getUri` is defined as follows:

```
fun getUri localize uri =
    case localize uri of
        SOME url =>
            (SOME (getUrl url) handle _ => NONE)
      | NONE => NONE
```

In practice, it is useful to be able to specify a list of localizers to try in sequence until one of them returns a URL from which a resource can successfully be acquired. This is analogous to search paths in traditional systems:

```
val getUriN : localize list -> uri ->
              Word8Vector.vector option
```

```
fun getUriN (localize::rest) uri =
    (case getUri localize uri of
         SOME vector => SOME vector
       | NONE => getUriN rest uri)
  | getUriN nil uri = NONE
```

10.2.3 Localizers and Component Managers

To support localizers in component managers, it is sufficient to (1) revise the `Component.load` operation from Section 9.2 to take a localizer as an additional argument; and (2) parameterize `MkComponentManager` from Section 9.5 over a localizer to use.

The `load` operation shall now look as follows:

```
val load : localize list -> uri -> component
    (* IO.Io, Malformed *)

fun load localizers uri =
    case getUriN localizers uri of
        SOME vector => unpickle vector
      | NONE => raise IO.Io {...}
```

Change (2) is accomplished as follows:

```
functor MkComponentManager
    (val initialNamespaceTable :
         (string * package) vector
     val localizers : localize list)
```

Note that the component table operates on (absolute) logical component names only. In other words, if two logical component names map to the same physical component name, the component will be loaded twice.

10.2.4 Configuring the Default Localizer

The virtual machine—and, thereby, the default component manager—needs to use a localizer. It should be possible for the user to specify this localizer at startup. For instance, Mozart and Alice read a localizer specification from an environment variable, whose syntax allows to specify a sequence of localizers from the following vocabulary:

Default denotes the localizer that returns a given URI as-is. This trivial mapping is meaningful since URL are a subset of URI. In practice, most configurations include the default localizer.

Root(*d*) denotes a localizer that resolves relative URI with respect to a given directory name *d* (or any absolute URL, really). This localizer addresses the scenario of starting an application with arguments that are relative URI, but need to be interpreted independently of the directory current when the application is started.

Cache(d) takes URI of the form *scheme*`://`*authority*`/`*path* to URL of the form d'`/`*scheme*`/`*authority*`/`*path*, where d' is the directory name d rewritten to an absolute file URL. This localizer addresses the common scenario of locally caching components from the networked file system. This is also useful for establishing an installation structure for predefined components packaged with the system (both Mozart and Alice use the cache structure for this).

Prefix(s_1, s_2) denotes a localizer that, given a URI starting with string s_1, returns a URL with the prefix s_1 replaced by string s_2. This localizer addresses the scenario where a set of physical locations are given for logical component names *en masse*.

Pattern(p, e) is defined by a pattern p and an expression e and takes a URI that match p to URI computed from e. p can contain *match variables* of the form `?{`x`}`, and e can contain corresponding occurrences of `?{`x`}`. For example, let p be `http://www.?{x}/?{y}` and let e be `ftp://ftp.?{x}/?{y}`. Then `http://www.foo.com/bar` would be mapped to `ftp://ftp.foo.com/bar`. This localizer is a more general form of the prefix localizer.

10.2.5 Related Work

Windows Dynamically Linked Libraries. A Windows dynamic library is referenced by its base file name. The actual library is located by trying to load it from a number of specific directories in sequence, including (details depending on the version of the operating system) the system directory, the directory in which the application resides, and the current directory. This has led to numerous application breakages—applications picking up incompatible versions of libraries; inability of two applications to depend on different versions of the same library; and library name clashes. In general, installation of a new application can break installed applications, a situation generally known as *DLL hell*. The root cause of the problem is that library names have no structure—in particular, they do not include any version information. OpenL circumvents this problem by separating logical from physical component names, and introducing structure on logical component names to allow for encoding of information that differentiates different variants of a single component. Moreover, configurability of localizers allows for special casing in case of version incompatibilities.

Windows COM. The Component Object Model [Rog97] was originally developed as infrastructure for the Object Linking and Embedding (OLE) technology, which allows for documents to embed documents handled by components unknown to the outer document's handler. COM components are implemented by dynamic libraries and live on executable files. A global database (the Windows *registry*, and, in the case of Windows 2000 and successors, the distributed Active Directory database) defines the mapping from human-readable "programmatic IDs" to GUIDs (globally unique identifiers), and from GUIDs to files. The database also supports superseding—a newer version of a component being able to emulate an older version. When a component exists in several versions, their programmatic IDs are distinguished heuristically by appending the version number, while their GUIDs will all be different.

In summary, COM tries to solve goals similar to OpenL's, but the configurability of its logical-to-physical name mapping is limited and global. In contrast, OpenL's mappings are programmable and per-application, or even per-component manager (if an application instantiates multiple component managers).

Java. Application programmers can define custom class loaders. Custom class loaders have full control over the way that classes are located and loaded—OpenL application programmers have the same flexibility. Java's default class loader uses the `CLASSPATH` environment variable, which is akin to the way the OpenL virtual machine parameterizes over a localizer. However, the `CLASSPATH` is just a sequence of directories (and "archives", which emulate directory structure within a file); as such, it is completely subsumed by and much less expressive than localizers.

Microsoft .NET. Microsoft .NET is a successor to COM built on a common runtime for so-called *managed code*. The runtime has control over all aspects of execution, including, in particular, loading and linking of components (called *assemblies*). Assemblies can be addressed by their file name relative to the application's installation location, which means that a componentized application can be installed by just copying a directory tree (termed *XCOPY deployment*, alluding to a DOS recursive file copy command). Assemblies can be addressed by any number of the following constituents: name, locale, processor architecture, and version number. Also, assemblies can be signed by a cryptographic key (called a *strong name*); in this case, the linker will fail unless it finds the exact same assembly at run time. Assemblies can be installed into the *global assembly cache*, or they can be located using a search path. Every *application domain* (similar to component managers, but with strong isolation guarantees) can have its own search path. Application programmers can define custom loaders, but do not have full control: the default assembly loader is always tried first; only if it fails will the user have a chance of delegating to his own loading behavior.

Microsoft .NET is superior to OpenL in that it supports strong names in determining whether an assembly satisfies a given link request. Microsoft .NET predefines more variables in link requests, but OpenL subsumes these by allowing to encode these into URI. Neither system suffers from the "DLL hell" problem.

Function Interposition. Bracha et al. [BCLO93] also address the function interposition problem. They propose a language, called *Jigsaw*, that provides operators on first-class components. This language is used to write what amounts to L_I linking programs, and is expressive enough to solve the function interposition problem.

10.3 Bundling

This section proposes a mechanism called *bundling* that serves to package several components into a single deployment unit.

During development, developers typically structure their application as a number of fine-grained components, for reasons of maintainability and short turn-around times. When it comes to deployment, there are reasons for more coarse-grained componentization:

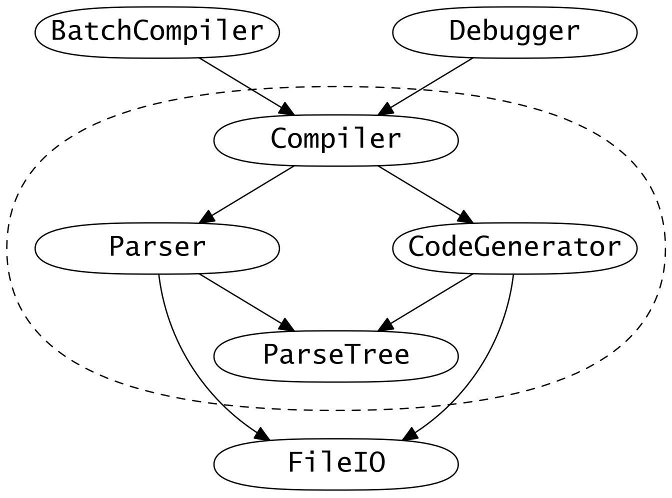
Hiding. Implementation details should not be visible, and unlicensed reuse of internal components should be prevented.

Early Binding. Components can serve to postpone linking decisions until after deployment for flexibility. When componentization decisions were also driven by development considerations, this power is often not needed; in fact, it effectively introduces opportunities for configuration errors on the part of the administrator who is deploying the application.

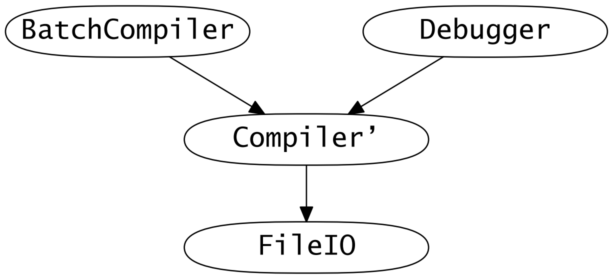
Time Efficiency. It takes longer to load a large number of small components than a small number of large components, especially over the network. In addition, the higher the number of deployed components is, the more edges the dependency graph has, and component types have to be verified on every edge.

Space Efficiency. Type representations are not small and may, in some cases, even dwarf the representation of a component's code and data. Those types that can be fully checked during early binding need not be deployed as part of the application.

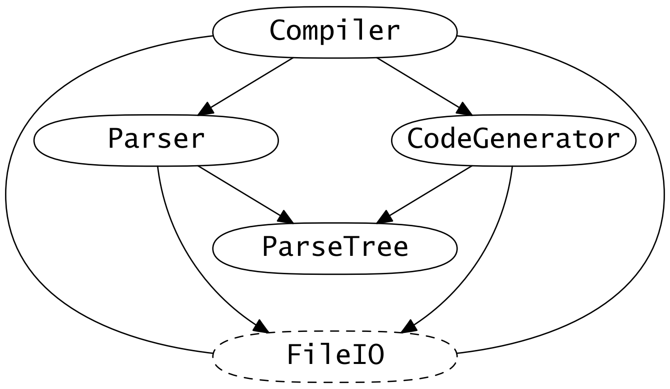
Example. Figure 10.1(a) shows a sample component dependency graph. The `Compiler` component implements a first-class compiler used both by a command-line batch compiler application, and a debugger (say, for evaluating expressions interactively). Several components use a file I/O component from the standard library. Assume that the components should be deployed with a different granularity, namely the components surrounded by the dashed line should become a single *bundle*, resulting in the component graph depicted in Figure 10.1(b). Figure 10.1(c) shows the internal structure of the bundle, where a dashed ellipse represents a dependency of the resulting bundle.



(a) Development Component Graph.



(b) Deployment Component Graph.



(c) The Internal Structure of the `Compiler’` Bundle.

Figure 10.1: An Example of Bundling.

A Pattern for Bundling. The intent is for developers to use bundling as follows. Development produces a set $C \in \text{Name} \longrightarrow_{\text{fin,inj}} \text{Comp}$ of component files. Then, bundling can be applied to a set $N \subseteq \text{Dom}(C)$, which results in a deployment unit $c = \text{bundle}(C, N)$, such that only $C' = C|_{\text{Dom}(C)\setminus N} \cup \{n \mapsto c\}$, $n \in N$, will be deployed. To obtain the final set of deployment units, several bundling steps may be necessary.

Bundling as an Operation on Deployment Units. Looked at from a different angle, at the core of bundling is the following operation: One can package an arbitrary non-closed connected subgraph of the graph spanned by import announcements, starting from a given component, yielding a new deployment unit. (Of course, a single component also forms a deployment unit.) Note that this operation effectively duplicates the bundled components. Unless used under the pattern described above, this can lead to unintended results, as these components may be loaded and evaluated multiple times, each instance having its own state.

Overview of this Section. In a first step, Section 10.3.1 derives the requirements that bundling must fulfill for to address the above scenarios. Section 10.3.2 sketches the OpenL approach and analyzes its properties. The bundle construction algorithm is presented in Section 10.3.3, and implementation issues are discussed in Section 10.3.4. Finally, Section 10.3.5 compares the proposed approach to related work.

10.3.1 Requirements and Non-goals

From the goals outlined in the introduction follow the following requirements:

R1 Bundling must require modifications neither to the source code nor to development-time configuration. This separates deployment from development considerations and decisions.

R2 It must be possible for components in a bundle to depend on components that are not part of the bundle. This allows for libraries to be shared by several bundles, as in the example above.

R3 Other than that, bundles must be self-contained, that is, not reference the components from which they have been constructed. This is so that the constituents components can be physically removed after bundling (when appropriate), to protect against component duplication. This implies that programs that make programmatic use of localization failures at run time may behave differently after bundling.

R4 It should not be trivial to extract constituent components from a bundle. This guards against unlicensed reuse of constituent components.

R5 The linking decisions in-between any two components included in the same bundle must be prescribed by bundling, and no longer be subject to late binding. This eliminates possibility of misconfiguration.

R6 When externalized, the bundle must consist of fewer files than the sum of its constituents. This reduces the number of files that need to be individually loaded.

R7 Imports between any two components that are part of the same bundle must be type-checked at bundling time. This implies that programs that make programmatic use of type mismatches at run time may behave differently after bundling.

R8 Types only needed for type-checking performed at bundling type must not be represented in the resulting bundle.

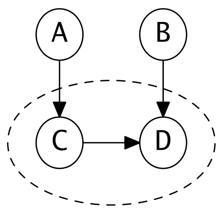
Non-goals. In view of the practical goals the solution has to fulfill, the following are explicitly declared as non-goals:

N1 It is not a goal to make bundling "fool-proof" in that an arbitrary bundling operation applied to a set of components would not modify the components' semantics.

In general, *any* bundling operation modifies semantics—which is really the purpose of bundling: bundling hides components, so that these components cannot be referred to externally any more. In particular, they are not dynamically discoverable any more. This means that any component that is to be dynamically discovered cannot be allowed to exist only in a bundle as a non-root component.

N2 It is not a goal to support references from outside the bundle to more than one component included in the bundle. Such a situation is depicted in Figure 10.2(a). The single designated component to which references are possible will, in the following, be called the bundle's *root* component.

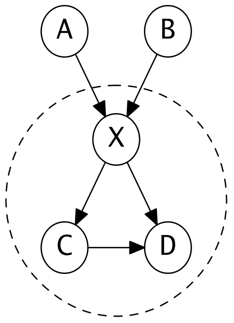
The reason to make this a non-goal is that it significantly reduces the complexity of the problem, and the apparent limitation can be worked around easily. For instance, Figure 10.2(b) defines a component `X` that imports `C` and `D` and reexports the entities they define. (`X` could be auto-generated by a tool.) Now, `A` and `B` have to refer to `X` instead of `C` and `D`. There are several ways to accomplish this: (1) modify the imports of `A` and `B` to refer to `X` directly (see Figure 10.2(c)); or (2) introduce "proxies" for `C` and `D` that import and re-export the corresponding definitions from `X`, and which are deployed under the logical names of `C` and `D`. This is shown in Figures 10.2(d) and 10.2(e).



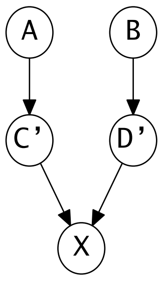
(a) Intended Bundle.

```
import val c from "C"
import val d from "D"
val c = c
val d = d
```

(b) The Merger Component X.



(c) Direct use of X.



(d) Use of X via Proxies.

```
import val c from "X"
val c = c

import val d from "X"
val d = d
```

(e) Components C' and D'.

Figure 10.2: Implementing Bundles with Multiple Roots.

10.3.2 Approach

The central idea of the proposed solution is to generate a single new component to represent a bundle. In other words:

- The imports of the bundle component are those components that are referenced from bundles in the component, but that are excluded from the bundle.
- The signature of the bundle component is the signature of the bundle's root component.
- The component body functor is generated as follows:
  - The functor's arguments are the *namespaces* imported by the bundle.
  - The functor's free variables are the *unevaluated components* included in the bundle.
  - The functor's body is a partial *linking program*.
  - The functor's return value is the namespace of the root component, as computed by the linking program.

This fits into the general strategy as follows. Section 8.3 defined linking programs that could be generated statically for a set of components and resulted in the components being linked and evaluated, and these linking programs were extended to support a lazy linking strategy in Section 9.1.2. Component managers as defined in Section 9.3.2 generate and interpret linking programs on-the-fly. Now, bundling generates a partial linking program for a non-closed connected graphs of components, and packages it into a component.

Fulfilled Requirements. This approach immediately fulfills: Requirement (R1) by virtue of bundling operating on compiled components only; (R2) by virtue of the bundle component supporting imports; (R3) by virtue of pickling transitively including values referenced through the free variables of pickled functions; (R4) by virtue of there being neither operations to obtain a component's body function, nor to obtain that function's free variables; (R5) by virtue of the partial linking program being generated at bundling time, as opposed to on-the-fly by a component manager; (R6) by virtue of a bundle component being picklable onto a single file.

Ramifications. This approach has the following ramifications:

- The fact that bundles are components implies that all operations on components also operate on bundles without modification, for instance, linking, component managers—and bundling. In other words, bundling is compositional: bundles can be part of bundles. This means that bundling has expressive power equivalent to lexical scoping on logical component names of the form **`let`** $n_1 = c_1, \ldots, n_m = c_m$ **`in`** c **end**.
- Since the bundling operation explicitly generates linking programs, the linking strategy used in-between the bundled components is under the control of the bundler. If this strategy is lazy linking, the semantic change induced by bundling is least noticeable—but other strategies are possible.
- Since the only new data and code generated by the bundling operation is metadata composed from the argument components, and a linking program, both of which are picklable, it follows that the resulting component is picklable if and only if all of its constituent components are picklable.
- Due to bundles being intra-linked, bundling provides an opportunity for link-time optimizations that may be too expensive to do at run time; for instance, specialization or inlining. This is conceptually equivalent to partial evaluation of the generated partial linking program.

10.3.3 A Bundling Algorithm

This section presents the actual algorithm for creating a bundle. First of all, the bundler needs the following information to create bundle:

The Root URI. The component denoted by the root URI is always included in the bundle, and the namespace it computes is the namespace exported by the bundle.

A Localizer. When the bundler has determined that the component corresponding to a given URI is to be included in the bundle, it uses the localizer to acquire it.

A Membership Predicate on URI. Assume a component c is included in the bundle. The membership predicate determines, for each of the components imported by c, whether they should also be included in the bundle or excluded from it. To do that, the membership predicate is applied to the imported components' URI as resolved with respect to the URI of c.

A Rewriting Function. Any component excluded from the bundle becomes an import of the resulting component. The bundler knows a unique absolute URI for each of these; however, the resulting component should usually have relative import URI. This is the purpose of the rewriting function. The rewriting function takes absolute URI to absolute or relative URI, and can be used to "undo" resolving—as well as for arbitrary rewriting: For instance, it can serve to realize the scenario depicted in Figure 10.2(c). (Note that the rewriting function needs not be injective.)

Optimizing Exclusions. In general, a component that is imported from bundled components may be referenced under more than one type. For example, in Figure 10.3(a), component `B` imports `D` under type t_1, while `C` imports it under type t_2. If only one of t_1 and t_2 needs to be represented, the bundle's representation becomes smaller and fewer types have to be checked at run time. One can distinguish the following cases:

(1) t_1 and t_2 are equivalent. The two imports can be merged, as shown in Figure 10.3(b).

(2) t_1 is a subtype of t_2 (or the symmetric case). The two imports can be merged into one import under type t_1, as every namespace that matches t_1 will also match t_2.[2] Note that this optimization is not semantically transparent: Assume that at run time, a component `D` is found that matches t_2, but not t_1. In the unbundled system, component `B` will still be allowed to successfully execute against `D`, while in the bundled system, any attempt to use `D` from `B` will fail with a type mismatch.

(3) Neither one of t_1 or t_2 is more specific. This can be further subdivided into the following cases:

[2]In the case that coercion of a namespace to a given type possibly needs a representation change, an adapter component may have to be synthesized into the bundle in-between the remaining import and `B`.

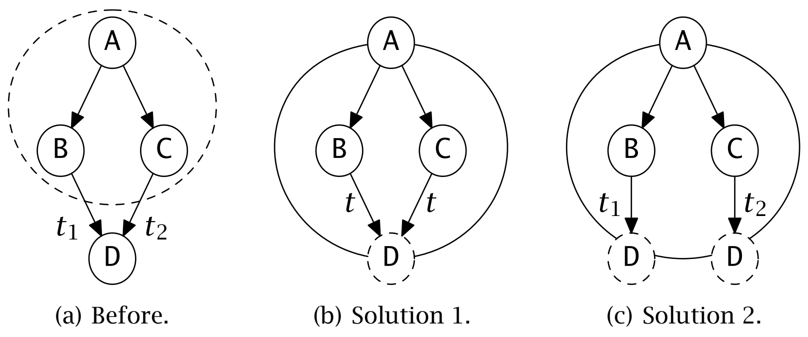


Figure 10.3: Bundles with Multiply Referenced Exclusions.

(3a) The intersection type $t = t_1 \cap t_2$ exists and can be computed. This case can be reduced to case (2).

(3b) The intersection $t_1 \cap t_2 = \emptyset$ is empty, and this fact can be proven. An error should be flagged at bundling time.

(3c) Otherwise, the resulting bundle shall consider the two imports to be unrelated (despite having the same name) and retain both of them, at higher resource cost, as shown in Figure 10.3(c).

Algorithm. Figure 10.4 shows an algorithm that uses the information described above to collect the components to bundle and manage their metadata. In this algorithm, “resolve” stands for the function that resolves a possibly relative URI with respect to another (usually absolute) one. “match” is a predicate that is true if a structure of the type given as first argument matches the type given as second argument. Since the algorithm performs type-checking at time of bundling, it clearly fulfills Requirement (R7). For simplicity, the algorithm assumes that the intersection of two types always exists and can be computed by the “intersect” function. The actual construction of the component bundle is examined in the next section.

10.3.4 Creating the Resulting Bundle Component

Now that the algorithm has determined the components that go in the bundle, the bundler needs on operation to actually construct the bundle. This section presents such an operation, first in a type-safe incarnation that, however, creates unnecessary overhead in the bundle. By descending one level to an unsafe realization, an optimized representation can be obtained.

A Component Construction Primitive. Up to now, the only options to create components at run time were capturing (creating an *evaluated* component from live data) and the first-class component syntax (creating a component from a *static* definition). The bundler, in contrast, creates a *unevaluated, computed* component. This constructor can be exposed as a primitive

```
**function** bundle(*rootUri*, *localizers*, *member*, *rewrite*) =
    **let** *included* be a new map from URI to components;
    **let** *excluded* be a new map from URI to signatures;
    **procedure** depthFirst(*uri*, *sign*) =
        **let** *component* = load(*localizers*, *uri*);
        **if** ¬match(sign(*component*), *sign*):
            **raise** Mismatch;
        *included*[*uri*] := *component*;
        **for each** ($uri'$, $sign'$) ∈ imports(*component*):
            **let** $uri''$ = resolve(*uri*, $uri'$);
            **if** *member*($uri''$):
                **if** $uri'' \notin$ Dom(*included*):
                    depthFirst($uri''$, $sign'$)
            **else if** ¬match(sign(*included*[$uri''$]), $sign'$):
                **raise** Mismatch;
            **else**:
                **let** $uri'''$ = *rewrite*($uri''$);
                **if** $uri''' \in$ Dom(*excluded*):
                    *excluded*[$uri'''$] := intersect(*excluded*[$uri'''$], $sign'$)
                **else**:
                    *excluded*[$uri'''$] := $sign'$;
    **let** *top* be a type matched by all components;
    depthFirst(*rootUri*, *top*);
    **return** a new component with:
        one import for every (*uri*, *sign*) ∈ *excluded*,
        a component body functor that has *included* as a free
            variable and arguments according to *excluded*,
        sign(*included*[*rootUri*]) as export signature.
```

Figure 10.4: Algorithm for Constructing a Bundle.

as follows:[3]

```
val component : { imports : (string * sign) vector,
                  body : package vector -> package,
                  sign : sign } -> component
```

`component` creates a bundle with the given metadata, and whose component body functor is as follows: It expects as arguments structures with the given signatures, packages them, passes the packages as a vector to the given function, unpacks the result, and coerces it to the given export signature.

[3]A convention introduced by the Standard ML Basis library calls for naming the constructor of an abstract type after the type.

Computing the Component Body Function. At this point, all that remains is to compute the body function. Given the result of the argument above, a function of the following type would be appropriate for this:

```
datatype source =
    INCLUDED of int
  | EXCLUDED of int
type script = (component * source vector) vector * int
val createBody : script -> (package vector -> package)
```

`createBody` takes a *script*—a data structure that describes the internal linking structure of a bundle—and returns a body function that takes a vector of argument namespaces. Every element of the script's vector is a tuple of a bundled component and a description of where it gets its arguments from: `INTERNAL` *i* denotes the *i*th element of the script, while `EXTERNAL` *i* denotes the *i*th argument. The integer in a script is the index of the bundle's root. For instance, the structure of the bundle depicted in Figure 10.1(c) could be described by the following script:

```
val script =
    (#[(compiler,      #[INTERNAL 1, INTERNAL 2]),
       (parser,        #[EXTERNAL 0, INTERNAL 3]),
       (codeGenerator, #[INTERNAL 3, EXTERNAL 0]),
       (parseTree,     #[])], 0)
```

A script can easily be computed from the *included* and *excluded* maps computed by Algorithm 10.4. `createBody` can then be implemented thus:

```
fun createBody (script, root) arguments =
    let
        val promises =
            Vector.map (fn _ => promise ()) script
        val futures = Vector.map future promises
        fun link (promise, (component, sources)) =
            let
                fun get (INCLUDED i) =
                    Vector.sub (futures, i)
                  | get (EXCLUDED i) =
                    Vector.sub (arguments, i)
            in
                fulfill (promise,
                         lazy Component.apply component
                              (Vector.map get sources))
            end
    in
        VectorPair.app link (promises, script);
        Vector.sub (internal, root)
    end handle Subscript => raise Mismatch
```

Optimization. In the case of a dynamically-typed language, this solution is complete. For statically-typed languages, although the solution is functional, it does not yet fulfill Requirement (R8), which states that internal types should not be checked at run time, and not even be represented in the bundle.

Note the following observation: By construction, the `apply` operation as used in `createBody` will never raise a `Mismatch` exception for any argument namespace computed from a component that is part of the bundle. Consequently, at the level of L_I, the application of `apply` in `createBody` can be partially evaluated to omit type-checking; and the `component` type can be used transparently to store only the component body function of every bundled component, without its metadata. This exercise is very technical and is, therefore, not explicitly demonstrated here for conciseness.

10.3.5 Related Work

Java. On the level of the Java language, Java *packages* fulfill some of the goals of OpenL bundles: packages have structured names and can be nested, and package boundaries are used to control access to classes and members [GJS00]. On the level of the virtual machine [LY99], which is the level at which code is deployed, packages fail to fulfill these goals: packages can not be closed—after deployment, a new class can be defined in an existing package and will be allowed to access the symbols internal to the package.

Another goal of OpenL bundles—that collections of components can be deployed as a single file—is fulfilled in Java by so-called *Java archives* (JARs). JARs need to be explicitly mentioned in the `CLASSPATH` for the virtual machine to be able to locate at run time the components they contain. In contrast to OpenL bundles, JARs are not intra-linked: Deploying implementations of two components `A` and `B` in a single JAR, where `A` depends on `B`, does not guarantee that `A` will be linked against this implementation of `B`. If another implementation of `B` can be found in a location preceding the JAR in the `CLASSPATH`, that implementation will be used instead. Also, individual components can be extracted from JARs, as opposed to OpenL bundles, which are opaque. In summary, JARs provide only for physical packaging, not hiding.

Microsoft .NET. At the level of the C# and Visual Basic languages, .NET has *namespaces* that are very similar to Java packages. In contrast to Java, however, .NET offer more structure for deployment: While logically types live in namespaces, physically they live in *modules*, many of which can be linked into an *assembly*. The **`internal`** access level means “internal to the assembly”, not “internal to the namespace”. By creating an assembly, all linkage in-between types contained in that assembly is prescribed, and is not subject to late binding any more. In this respect, .NET assemblies are superior to Java packages and JARs, and akin to OpenL bundling. However, in contrast to .NET assembly linking, OpenL bundling is compositional, with the ensuing flexibility.

Units. Flatt and Felleisen's Compound Units [FF98] realize much the same degree of hiding as OpenL bundling, and are intra-linked. Compound Units are defined in terms of source-level rewriting; no mechanism for creating Compound Units from existing (compiled) Units is proposed—in contrast to OpenL, which can even define this operation at the language level.

Jiazzi. The Jiazzi system [MFH01] brings Units to Java. Creating Compound Units in Jiazzi is similar to OpenL bundling—in particular, bundled components are effectively duplicated. Linked Compound Units have multiple export types, which has a higher cost than OpenL's type intersection. Component renaming is performed upon linking, which OpenL performs at run time using localizers, for added flexibility.

10.4 Summary and Validation

This chapter has presented the file format, naming, and packaging of components for deployment. Components, which are first-class in the language, are made *picklable*; thereby, pickles become a single, well-defined interchange format for binary components. Pickles make it possible for externalized components to contain both code *and* data; also, in addition to just loading compiled components from disk, pickling provides for saving of dynamically-created components. Also, pickles can be unified with components, allowing to use pickles and components interchangeably both in language syntax (import announcements) and the programming interface.

This chapter introduced a distinction between *logical* and *physical* component names. This separation enables expressive run time configurability, solving problems such as network transparency, component interposition, and naming components by function. Embodied as Uniform Resource Identifiers, logical component names obtain a well-defined hierarchical structure, and, in conjunction with URI resolving, allow to easily implement sets of related components to be location-independent. *Localizers* are programmable translators from logical to physical component names. By using URL for physical names, the networked, distributed file system defined by standard Internet protocols can be embedded into the component loader.

With *bundling*, this chapter defined a compositional operation to create components from graphs of existing components. Bundling is a form of static linking in the sense of early binding that allows to express hiding. The implementation of bundling (with the exception of one optimization) amounts to generation of a parameterized linking program, which could be realized at the level of the high-level language.

Validation. Both Mozart/Oz and Alice define external component representation in terms of pickling. Also, both systems support a syntax for defining localizers similar to the presented approach.

A tool to perform bundling has been implemented under the name of *static linker* in both Mozart/Oz and Alice. The static linker in Oz impacts semantics to a higher degree, in that it allows for two linking strategies, both of which are different from that of the default component manager used for non-bundled components. The Alice static linker only implements the lazy linking strategy that component managers use.

The Mozart/Oz static linker is fully implemented in Oz itself, which was eased by the fact that Oz is dynamically typed. Alice has a more complex implementation due to Alice ML being statically typed. In fact, for the optimization reasons presented above, the Alice implementation builds heavily on unsafe primitives; the linker as a whole, however, gives the guarantee by construction that the resulting bundle will always be type-safe.

Chapter 11

Distribution

This chapter describes the design and implementation of simple, efficient, and type-safe primitives and abstractions for programming distributed applications in OpenL. The proposed extensions provide for network transparency and network awareness, and are strongly integrated with components. The implementation is compact, due to reusing concepts from pickling and components, and mostly built in OpenL itself. As such, it represents a case study of how to build systems from the services presented in previous chapters.

A *distributed system* is a system that runs on a number of interacting *sites* interconnected by a network. The design and implementation of such systems can be complex; however, programming languages can serve to significantly reduce the complexity if they offer high-level abstractions specifically for exposing services to other sites, invoking services exposed by other sites, and generally transferring data and even computations in-between sites.

This chapter proposes primitives and abstractions for distributed programming in OpenL that fulfill the following requirements:

Network Transparency. A program that is part of a distributed system can hold both *local* and *remote* references, depending on whether the actual data they denote resides on the site on which the program is running, or on a different site. If instances of a language type have both local and remote references, and operations on them are supported with identical behavior on both local and remote references, one speaks of *network transparency* [Car95].[1]

Network transparency has many benefits. Programmers do not need to switch abstraction levels when working with both local and remote references; they can abstract from whether an actual reference is local or remote; and a single-site program can easily be refactored to

[1]Of course, this does not require that all operations be defined on remote references.

work in a distributed setting. Also, consider the case where function application is defined on remote functions: language data can then be communicated between sites as easily as passing them as arguments to functions, without any explicit representation conversion being necessary.

The proposed approach offers network-transparent support for a subset of the language data types that allows to express a number of distributed programming patterns. This support extends across both the definition of interfaces between sites and invocation of services provided by other sites.

Network Awareness. Network transparency does not mean that a system's distribution structure becomes unimportant. On the contrary: for security and efficiency reasons, programmers must understand and control the distributed behavior of data structures.

The proposed approach makes establishing of connections explicit. Distribution behavior is strongly connected to types; programmers can control a system's distribution structure by designing their data representations accordingly.

Type-safety. Typing cross-site communication must be as expressive as typing single-site programs. In particular, operations that are type-safe on local references must be equally type-safe on remote references. The presented approach ascertains type safety when a connection is established. Thereafter, type-checking of communication over that connection is minimized for efficiency.

Security. When a site receives data from another site, it may want to impose a lower level of trust on that data—in particular, if that data contains functions. It must not be possible for a site to forge local references on another site just by sending data to it; the receiving site must explicitly allow access. The proposed approach fulfills this through capability-based security: cross-site transfer of resources and references to resources is disallowed, and received functions need to be explicitly granted access by the receiving site explicitly passing references to resources as arguments.

This chapter defines a middleware for distributed computing in OpenL that is based on pickling and components as introduced in previous chapters. The simplicity of these mechanisms, both in use and implementation, is as much a contribution to the field of distributed middlewares as it is a case study for programming with pickling and components.

11.1 Overview

A discussion of a middleware for distributed computing must happen at two levels. At a high level, *patterns* serve to design distributed systems. At a lower level, the construction of distributed systems relies on the programming system to support implementing these patterns. This section describes some patterns addressed by this work.

11.1.1 Patterns for Establishing Connections

Before a number of initially independent sites can collaborate to form a distributed system, they need to become aware of and connect to one another. Typical patterns include the following:

Client/Server. A *server* is a site that provides one or more services under well-known interfaces. *Clients* are sites that access these services. Clients are short-lived in comparison to servers.

Since servers are usually available over long periods of time, and their address and interface seldom changes, clients typically use early binding to access a server's services.

Peer-to-peer. A *peer-to-peer* relationship is much more flexible. All sites are *peers* in the system. Peers learn of each other's existence at run time. The system as a whole is loosely coupled: peers can dynamically participate in or drop out of the system.

In a sense, every peer can be considered a short-lived variant of a server. Because of system dynamics, peers use late binding to access other peers.

Delegation. Consider an application whose purpose is to perform an expensive computation. For scalability in terms of performance, this application may factor its computation into a number of tasks that can be executed in parallel. In this case, the application may spawn new sites on a number of networked computers, and distribute the sub-tasks across these sites, also resulting in a distributed system.

Section 11.2 shows how these architectures can be expressed in OpenL.

11.1.2 Patterns for Exchanging Data

Sites participating in a distributed system have to exchange data. At the network level, this communication is performed by sending and receiving of messages, which are just sequences of bytes. This is not, however, an adequate level for software designs to model communication. Instead, the distribution middleware needs to create the illusion of a global shared store, and sites communicate via data structures in this store. Programmers think about instances of their data types as, for example, bound to a specific site or stateless and replicable to other sites, or they may want to disallow instances from given types to participate in remote communication. The middleware is in charge of translating remote operations to exchanges of messages.

Section 11.3 describes distributed behaviors of data structures and how they can be expressed in OpenL.

11.2 Connecting Sites

This section looks at how the various patterns for connecting sites can be realized in OpenL. Section 11.2.1 starts with a simple server that serves out a data structure for use by any number of clients. The realization is very simple and requires no extension of OpenL—but it does exhibit a number of interesting properties, discussed in Section 11.2.2. The realization of other connectivity patterns is equally simple, which is illustrated by Section 11.2.3.

11.2.1 A Simple Data Server

As the simplest scenario, consider a simple client/server application, in which the clients can ask the server for a piece of data (say, the current time on the server). It is easy to write an OpenL application to do the following:

- Let x be a reference to the data that is to be exported.
- Capture x into a first-class component X.
- Create a socket, bind it to a port (say, 8001), and listen for incoming connections.
- From every connection, parse incoming data into a HTTP request.[2] If this request is for a specific URI (say, `/x`), pickle X and return it as the body of a HTTP response.

Assuming this application runs on a machine named `leifk`, any HTTP client can retrieve a fresh pickled copy of x from the URL `http://leifk:8001/x`. In particular, a client can use that URL in an import announcement:

```
import val x from "http://leifk:8001/x"
⟨... use x ...⟩
```

[2]HTTP is just one of any number of possible protocols. It serves as an example for the sake of conciseness, as it has a publicly available specification.

This is all there is to it—the first client/server application is running. Every time a client is run, it gets a clone of a current snapshot of *X*.[3]

Error Handling. Since it is based on pickling, the handler on the server can fail if the captured component is sited at the moment the client requests it. (This may be a transient error, as captured components can be stateful.) One solution is to create the handler as follows:

```
fun makeHandler component =
    fn httpReqest =>
       pickle component
       handle Sited => pickle comp raise Sited end
```

Now, if pickling fails, the server still delivers a component to the client. Upon evaluation, the client gets an exception to the effect that the server failed, and can deal with the error appropriately.

Abstractions. The above pattern can be captured by the following abstractions:

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

The `offer` abstraction would perform the server-side operations, and `take` a client-side import. The intuition is that the server offers a data structure, and clients take the offer.

11.2.2 Properties of the Approach

The example in the previous section was short, but it already exhibits a number of interesting aspects:

- All data transfer is reduced to pickling.
- *X* can be any (picklable) component, whether produced by the compiler, by capturing, or by first-class component syntax. In particular, *X* can contain first-class functions.
- Pickles, static components, and dynamic components representing servers in a distributed system are all addressed uniformly.
- Remote references are embedded into the URI namespace of logical component names.
- The client could not be simpler, as it already is integrated with the component manager, by virtue of component loading supporting HTTP URL (see Section 10.2.1).

[3]With a simple modification, the server could take a snapshot once and cache it. What the right thing is depends on the actual application.

- Remote references are URI, in other words, simple strings. Thus, remote references can be transferred over any (external) communication channel, for example, in a telephone conversation or an email message.
- Remote references themselves are untyped, but the components they denote are typed by their metadata. Type-checking occurs at first use of the remote reference.
- If the URL is obtained out-of-band, the URL is passed as argument to `ComponentManager.link`, and both sites have a server running, the client/server model turns into the peer-to-peer model.

11.2.3 Remote Execution

Based on the data server, it becomes simple to write an application that spawns a site and has it execute a given task. Such an application would proceed as follows:

- Create a component X whose evaluation causes the task to be executed.
- Start an in-process HTTP server and register X under URL u.
- Start a new virtual machine (directly on the same machine, or using `rsh` or `ssh` to launch one on a remote machine) and pass u on its command line.

The newly started virtual machine will instantiate a new component manager and use it to load and evaluate the component denoted by u, which is the root URL the virtual machine is started with.

When Section 11.3.4 introduces proxies, this mechanism is extended to provide for bidirectional communication.

11.3 Distribution Behaviors for Data

As detailed above, accessing a remote reference is defined by pickling. Pickling defines a persistence semantics for every type of node, and the persistence semantics of a complex data structure is defined by that of its constituent nodes. This section looks at the distribution behaviors that correspond to the persistence semantics, which come “for free” from the use of pickling, and introduces new node types for additional behaviors that are essential for rich, distributed programming.

11.3.1 Replication

The most straightforward behavior is replication, applicable to stateless nodes. Pickling replicates stateless nodes. A site that is operating on stateless nodes cannot distinguish whether these nodes are stored locally, on a different site, or replicated (copied) from another site. Semantically, replication is just an optimization.

11.3.2 Cloning

Cloning is similar to replication, but applies to stateful nodes. Pickling clones reference cells and arrays. Cloning is not semantically transparent: modifying the state of a clone of a node x does not affect x. Since cloning is already made available to single-site programs via pickling, and exporting and importing data across sites is explicit, extending cloning from pickling to distributed communication does not break applications.

11.3.3 Resources Nodes

Resources are data that have an effect on the environment of a given site. OpenL's capability-based security (see Section 9.5.3 implies that code can only access a given resource if it is explicitly passed a reference to it by the owner of that resource. Extending that model to distributed communication means that a remote reference cannot refer to a resource.

Section 7.1 made the decision to disallow references to resources in pickles. In other words, the default pickling semantics for resource nodes yields exactly the desired distributed behavior.

Note that it still is easy to write mobile agents that request access to given local resources: Components are picklable, and a remote agent that is packaged as a component can mention resources in its imports. It is up to the site that evaluates a component obtained from another site to decide which resources to grant access to, by using a correspondingly restricted component manager.

11.3.4 Stationary Nodes

The communication patterns from the previous section only allow for unidirectional communication. On the one hand, since remote references are URL, a client can use requests of the form

```
    http://leifk:8001/x?a=1&b=2
```

The handler registered on the server obtains these arguments as part of the parsed HTTP request. It could interpret these arguments and dynamically compute a component depending on them. On the other hand, the implementation level is low, and does not provide for network transparency.

Proxy Functions. A network-transparent, language-level equivalent is the *proxy function*. Consider the following operation:

```
functor MkProxy(type a  type b  val f : a -> b) :
    sig
        val f' : a -> b
    end
```

An application of `MkProxy` creates a proxy function f' for a given function f.[4] In a single-site program, an application $f'\ x$ evaluates as follows:

- If x is sited, raise a `Sited` exception. Otherwise, pickle x, then unpickle, resulting in a clone x' of x.
- Evaluate $f\ x'$.
- Case 1: $f\ x'$ succeeds and results in a value y. If y is sited, raise a `Sited` exception. Otherwise, pickle y, then unpickle, resulting in a clone y' of y. Return y' as the result of $f'\ x$.
- Case 2: $f\ x'$ raises an exception e. If e is sited[5], raise a `Sited` exception. Otherwise, pickle e, then unpickle, resulting in a clone e' of e. Raise e' as the result of $f'\ x$.

In essence, in a single-site program, a proxy function is identical to the function except that all data flowing into it and out of it is passed via pickling.

Now comes the time of defining a pickling behavior on the proxy functions *themselves*, and this introduces the new concept of remote invocation: Let the pickled representation of f' not include f, but a remote reference to the instance of f on the site that created f'. Upon unpickling, create a function that establishes a connection to the remote reference, to implement the exact same behavior as described above, with the added site boundary in the middle of every "pickle, then unpickle" pair.

Example. As an example, consider a server that provides a stationary service to convert integers to their string representation:

```
fun f i = Int.toString i
structure Service =
    MkProxy(type a = int  type b = string  val f = f)
val uri = offer (pack Service :
                     (val f' : int -> string))
val _ = TextIO.print (uri ^ "\n")
```

[4]Obviously, in a dynamically-typed language, this operation can be exposed as `val proxy : ('a -> 'b) -> ('a -> 'b)`.

[5]Note that exception constructors are always picklable, but arguments to exception constructors may be sited.

When the server is ready, it prints out the URI under which the service can be accessed, say, `"http://leifk:8001/f"`. A client can then access this service as follows:

```
structure Service =
    unpack (take "http://leifk:8001/x") :
           (val f' : int -> string)
val s = Service.f' 17
```

Note that the client does not need to know, or care, whether the service is stationary or mobile.

By default, an application of the form `f' 17` (where `f'` is a proxy) is synchronous, in other words, blocks the thread until the server responds. Note that any application can be turned into an asynchronous invocation by wrapping it in an application of `concur`.

Bidirectional Communication using Proxies. A simple pattern to establish bidirectional communication between two sites using proxies is as follows: a site A establishes a connection to a site B and retrieves a component X from it that contains proxy functions. One of these could be a proxy function f that takes a component as argument. If A applies $X.f$ to a component Y that contains proxies created by A, then A and B can invoke each other's services via the proxy functions in components X and Y.

Stationary Nodes. The more general distribution behavior that is enabled by proxy functions is the *stationary node.* The function that is being proxied becomes stationary, that is, does not move with the proxy when a remote reference to the proxy is transferred to another site. This is useful in the following scenarios:

Shared State. The default behavior on stateful nodes is cloning. By hiding stateful data behind a proxy function, the stateful data becomes stationary. Different sites effectively share the same state.

Efficiency. It may be more expensive to replicate a large data structure than to repeatedly transfer queries against this data structure and their results. This can be achieved by making the data structure stationary.

Security. If data that represents sensitive information, such as credit card numbers, is replicated across sites, this can lead to information disclosure vulnerabilities. Making sensitive data stationary is a protection against a whole class of attacks.

11.4 Implementation

This section describes the implementation of the distribution layer. The distribution layer is made available as a structure `RemotePort` with the following signature:

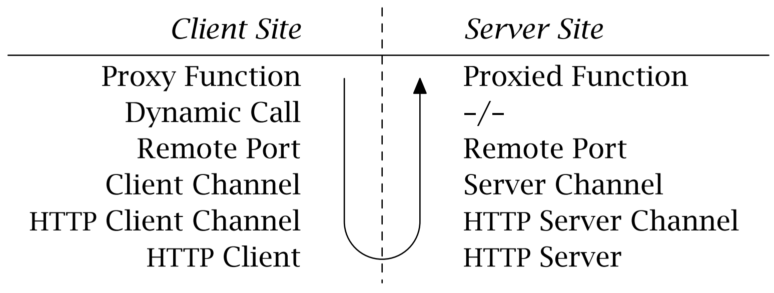


Figure 11.1: Overview of Proxy Invocation.

```
signature REMOTE_PORT =
sig
    val offer : component -> string
        (* Sited *)
    val take : string -> component
        (* IO.Io, Malformed *)

    functor MkProxy(type a  type b  val f : a -> b) :
        sig
            val f' : a -> b
        end
end
```

`offer` registers the component with a server channel and returns the corresponding URI. `take` is a trivial wrapper around `ComponentManager.load`. The interesting part is the implementation of the `MkProxy` functor.

Overview. An invocation of a function through a proxy traverses a number of layers, as depicted in Figure 11.4. The responsibilities of each layer are as follows:

HTTP Client and Server. These implement the HTTP protocol, used in this presentation as the messaging layer on top of TCP/IP.

Client and Server Channels. The actual sending of messages is encapsulated in one abstract channel type for each side. A channel provides functionality for exporting values and proxies, generating an appropriate URI, and for invoking a proxy, given a URI. At this level, value and proxies are just byte vectors and functions from byte vectors to byte vectors, respectively. The HTTP client and server channels implement channels based on HTTP. The abstraction makes it simple to substitute other protocol implementations.[6]

Remote Port. The remote port interfaces between the user-visible signature given above and channels. In other words, it transforms values from

[6]This assumes that the component managers' `load` operation supports registering new client protocols for given URI schemes.

and to byte vectors, and handles exceptions raised by proxied functions. This logic is the same irrespective of the underlying protocol.

Dynamic Call. Proxy functions must be picklable, but the channels they are based on are, by nature, sited. The *dynamic call* mechanism introduces a form of late binding that allows to express an invocation of a sited client channel as a picklable function.

Section 11.4.1 defines abstract channels and describes their incarnation using HTTP. Section 11.4.2 then shows an implementation of the `RemotePort` structure, generated by a functor parameterized over abstract channels.

11.4.1 Server and Client Channels

Server Channels. A server channel is defined by a structure with the following signature:

```
signature SERVER_CHANNEL =
sig
    type vector = Word8Vector.vector
    type handler = vector -> vector
    val registerValue : vector -> string
    val registerProxy : handler -> string
end
```

`registerValue` causes a channel to expose a given value to other sites, and returns a remote identifier for the value in the form of a URI. In a sense, `registerValue` is an untyped variant of `offer`. `registerProxy` creates a remote identifier for a handler that can be activated by other sites, which given a value computes and returns a result value. Behind the scenes, a server channel creates a connection point to which other processes can connect, and a thread that listens for and accepts connections, and processes messages sent over them.

A Server Channel Based on HTTP. One can easily implement a structure `HttpServerChannel` to fulfill the above signature using the HTTP protocol. The first time one of its functions is invoked, it starts a HTTP server on a port number allocated by the operating system. Every registration extends a monotonically growing mapping from URI to values and handlers. The HTTP server processes requests as follows: When a `GET` request is made to a URI that denotes a registered value, it returns a response with status code `200` (success) whose body is that value, verbatim. When a `POST` request is made to a URI that denotes a registered value, it applies that handler to the request's body, and returns a response with status code `200` whose body is the value returned by the handler. All other requests cause a response with an error status code to be returned.

Client Channels. A client channel is defined by a structure with the following signature:

```
signature CLIENT_CHANNEL =
sig
    type vector = Word8Vector.vector
    val invokeProxy : string * vector -> vector
        (* IO.Io *)
end
```

The `invokeProxy` operation is dual to `registerProxy`: it causes a registered handler to be executed on the server. Note that there is no dual operation for `invokeValue`, as this functionality is provided by the component managers' `load` operation.

A Client Channel Based on HTTP. Given the HTTP server channel, implementing `HttpClientChannel` is straightforward. `invokeProxy` expects, as its first argument, a URL with the `http` scheme, parses it to determine the target server, and sends a `POST` request to it. This request carries the second argument as its body, and the body of its response (given it has a success status code) is returned. The component managers' `load` operation acquires a value registered on a server by issuing a trivial `GET` request.

11.4.2 The Remote Port

Now a remote port can be constructed given a server and client channel:

```
functor MkRemotePort(
    structure ServerChannel : SERVER_CHANNEL
    structure ClientChannel : CLIENT_CHANNEL) :
REMOTE_PORT = struct ... end
```

`offer` and `take` are straightforward:

```
fun offer component =
    ServerChannel.registerValue (pickle component)
fun take uri = ComponentManager.load uri
```

For implementing `MkProxy`, two new primitives are needed.

Dynamic Call. As mentioned above, proxy functions need to be picklable, but use sited channels. This contradiction is resolved by the *dynamic call* mechanism:

```
val setTarget : (string * package -> package) -> unit
val dynamicCall : string * package -> package
```

`dynamicCall` is a built-in primitive (in other words, picklable) that simply invokes an arbitrary target that has previously been set (in the same process) by `setTarget`. The type of the target function has been chosen to work on arbitrary types, yet still be type-safe. Since the target operation is not part of the representation of dynamicCall, it is not part either of the proxy function, which makes it picklable. Note that setTarget must be a system primitive (that is, sited; see also Section 9.4.1), as it has a side-effect on the process's environment, which is considered a resource.

The Proxy Target. All proxy functions share a common target for the dynamic call they need to make. The target simply calls the client channel's invokeProxy function, with the necessary conversions from packages to pickles and back:

```
fun target (uri, package) =
    let
        val component = capture package
        val v = pickle component
        val v' = ClientChannel.invokeProxy (uri, v)
        val component' = unpickle v'
    in
        ComponentManager.eval (uri, component')
    end
val _ = setTarget target
```

(Note that for simplicity, this implementation uses the global component manager. For security, this should be a restricted component manager in practice, seeing that invokeProxy is expected to return a component without imports.)

Constructing a Proxy Function. Now that the target is set, the actual proxy function itself can be implemented:

```
functor MkProxy(type a  type b  val f : a -> b) =
struct
    fun handler v = ...
    val uri = ServerChannel.registerProxy handler

    fun f' a =
        let
            val package =
                pack (val x = a) : (val x : a)
            val package' = dynamicCall (uri, package)
            structure B =
                unpack package' : (val x : b)
        in
            await B.x
        end
end
```

The one noteworthy piece is the application of await, which is explained below. Still missing is the definition of the handler that is registered with the server channel:

```
fun handler v =
    let
        val component = unpickle v
        val package =
            ComponentManager.eval ("", component)
        structure A =
            unpack package : (val x : a)
        val component =
            let
                val b = f A.x
                val package =
                    pack (val x = b) : (val x : b)
            in
                capture package
            end handle e => failedValue e
    in
        pickle component
    end handle Sited => pickle (failedValue Sited)
```

The conversions from a pickle to the argument value and from the result value to a pickle are similar to the conversions performed in `target`. More noteworthy is the handling of exceptions, which comes in two forms: (1) The application `f A.x` can raise any exception. If this happens, we replace it with a failed value. Note that failed values are, in general, picklable. (2) It is possible that either the result of `f A.x`, or the exception it raised, are sited. In this case, the application of `pickle` will raise a `Sited` exception. This exception is treated identically to any other exception: it is replaced by a failed value.

Now it becomes obvious why the proxy function `f'` contains an application of `await`: If the proxied function raised an exception, it is transferred back as a failed value. By explicitly requesting the result, the proxy ensures that the client will have to handle this exception.

11.5 Discussion

Optimization: Type-checking. As defined above, every application of a proxy function performs a type-check for the argument on the server, and a type-check for the result on the client. If the server always trusts the integrity of its clients, no type-checks need to occur after initially establishing a connection. This can be achieved by substituting untyped pickling and dynamic call operations for their typed occurrences:

```
val pickle : 'a -> vector
val unpickle : vector -> 'a
```

```
    val setTarget : (string * 'a -> 'b) -> unit
    val dynamicCall : string * 'a -> 'b
```

Then, **pack**/**unpack** (and capture/eval) can be removed, which means that no type-checking occurs.

Optimization: Local Proxy Invocation. If client channels are implemented naïvely, even an invocation of a proxy on the home site of the function it proxies would traverse all transport layers. The layers below pickling (which will still be necessary because proxies by definition have to clone arguments and results) can be optimized away with no change to MkRemotePort: the client channel simply has to test whether the server site is the same as the client site. In this case, is would directly call the registered handler instead of establishing a connection.

Threading Model. In a similar vein, the threading model of proxy invocations is up to the client and server channels, and not MkRemotePort. In general, there is little to be gained by limiting the concurrency of concurrent proxy invocations on the client, but a server could sequentialize all invocations to all or each proxied function. Different channel implementations can be substituted for different synchronization behavior.

Synchronization Nodes. Futures are an expressive construct for concurrent programming. In a sense, distributed programming is an extension of concurrent programming. On the one hand, it could be argued that futures should have distributed semantics as well. On the other hand, synchronization patterns across multi-site systems are usually very different from single-site systems, and cross-site synchronization can often be encapsulated in proxies. (For instance, non-transparent futures can be constructed on top of proxy functions.)

Distributed Garbage Collection. The above design makes the assumption that a value or proxy, once exported, will live as long as its home site: Server channels have no operation to unregister an exported entity. If the design was extended to allow for an unregistering operation, or to include advanced distribution behaviors such as Mozart's mobile objects, the protocols would have to be extended to support distributed garbage collection. This is outside the scope of this work.

11.6 Related Work

There is a huge body of research work on the design and implementation of distributed systems. This section only covers a number of approaches that are similar to the approach taken by OpenL.

CLU. CLU [HL82] featured the first language-level remote procedure call. Every type has a “transmissibility” property that defines if and how it can be transferred to other sites; in other words, CLU defines a pickling mechanism. By default, remote calls all have by-value semantics, but users can provide encode and decode operations for their own types. Messages contain no types: types for remote invocation are checked at compile time only, since types are meant to catch errors, not provide security. OpenL is superior in that it makes by-value and stationary nodes composable, and in that first-class functions are transferred by value with their code.

Modula-3 Network Objects. Birrell et al. [BNOW95] present a distribution layer implemented in and for Modula-3. By-value transfer of arguments and return values are based on pickling; remote object references make objects into stationary nodes. Like in OpenL, a remote reference that an object’s home site passes to a second site remains usable when the second site sends it to a third site (“third-party transfer” of remote references is transparent). Stationary objects are implemented by *stub modules*: stubs need to be generated explicitly by an offline tool, and must be known at the time that clients to the stationary objects are compiled. In contrast, OpenL generates stubs dynamically. Similar to the distinction between the remote port and channels in OpenL, the implementation of Network Objects separates transports (user-definable buffered streams) and protocols that are implemented by stubs. To establish a connection—to initially obtain a reference to a remote object—, a separate registry needs to be set up (called *agent server*). The registry is provided by a well-known object listening on a standard port.

Java RMI. Java Remote Method Invocation [WRW96] is a direct descendant of Modula-3 Network Objects. Java improves over Modula-3 in that clients can automatically obtain and dynamically link the stub classes, albeit the class files need to be transferred out-of-band (for instance, by setting up a Web server on the server machine). This is much simpler in OpenL, since proxies are picklable by-value (they are self-contained and include their code). Java RMI bases transfer of argument and return values on pickling. Efficiency dictates that type information be left out. According to Breg [BP01], this was the principal reason to support modal pickling in Java. The approach presented above shows that this need for modal pickling, which would require support in the low-level mechanism, is mitigated in OpenL.

Obliq. Cardelli [Car95] presents the distributed, dynamically-typed, interpreted scripting language Obliq. Like OpenL functions, Obliq procedures are first-class and transferable by-value over the network. Unlike OpenL, Obliq makes all mutable nodes stationary, in particular objects, and implicitly creates proxies for all locations. Obliq provides a *network copy* operation; in contrast to value transfer over the network, a network copy clones mutable nodes. Many distribution behaviors can be expressed in terms of these operations. All security in Obliq is obtained through lexical scoping. Obliq's implementation is based on Modula-3 Network Objects.

Mozart. Distributed Oz [HVBS98], as implemented by Mozart, provides very elaborated distribution behaviors such as mobile objects [VHB$^+$97] and distributed logic variables [HVB$^+$99], in addition to the behaviors supported by OpenL. In fact, the functionality of the simple distribution layer presented in this chapter was designed as a minimal subset of Mozart. Due to its broader scope, the distribution protocols in Mozart are significantly more complex, but also measurably more efficient. Distribution behaviors are fully implemented at the level of the runtime system, which is fairly intrusive when compared to OpenL's orthogonal extension of the base system.

DSS. The Distribution Subsystem [KEBH03] is a language-independent middleware that provides for distribution behaviors, reference management, and messaging of language entities. Instead of coupling itself to a single programming system, it takes a library approach that programming systems can interface to. As such, by definition, the actual implementation happens at a lower level than the high-level programming language it is used to support, but alleviates that disadvantage by a clean separation of concerns. The DSS is sufficiently expressive to cover the distribution behaviors supported by Mozart.

D'Caml. D'Caml [WAS00] is a distributed version of O'Caml, designed for parallel execution of programs. D'Caml lacks OpenL's dynamic connectivity: all sites are spawned by a single process, called the *host*, and execute the same program; therefore, connection establishment, typing, and security are non-issues. The *distributed shared memory* creates an illusion of a global store: stateless nodes are always replicated on first access, and mutable nodes are all stationary. Whether a node is mutable is always known statically, but any reference to a mutable node can be remote, which requires runtime support and overhead (in contrast to OpenL). Transfer of closures is reduced to the transfer of the environment; since all processes execute the same program, transfer of code is reduced to translating the entry addresses (various sites could run on different hardware and software platforms; the compiler generates one version of the code per platform).

Using URLs to Identify Servers. It is common practice to use URI such as `http://leifk:8001/x` above to identify values exported by other sites. For instance, Java name servers [WRW96] interpret `rmi` URI, Microsoft .NET remoting channels [Mic03c] interpret remote references in URI format, and the offer/take mechanism in Mozart calls its strings *tickets* [HVBS98]. In all these systems, however, these URI can only be used when interpreted by the same class that generated them; in contrast, OpenL generates URL for remote references that can be interpreted by any URL client.

11.7 Summary and Validation

This chapter has presented a simple, yet effective distribution layer for OpenL. Dynamic establishment of connections can be completely expressed without any extension of either the language or the runtime system. The enabling feature for this is the built-in support of network addresses in localizers. Basic transfer of values is directly provided by pickling: stateless nodes are replicated, stateful nodes are cloned, and references to resources are protected.

Additionally, this chapter demonstrated a viable implementation of a transparent remote function application mechanism. *Proxy functions* are the only kind of stationary node. Their realization required no extension of pickling, and the runtime only needed additional non-intrusive primitives to support a *dynamic call*. The dynamic call facility can potentially be more generally useful than just for defining proxy functions.

The implementation cleanly separates the transport layer from the protocol layer. In this fashion, the actual transport protocols become user-definable as so-called *channels*.

Validation. Alice implements its distribution layer practically as described here. The most notable difference is that instead of a type-safe dynamic call, it uses an untyped pickling mechanism, trading safety for efficiency.

Chapter 12

Conclusion

This chapter summarizes the main contributions of this thesis, and gives some concrete ideas for future work in areas which have only been touched on, or which have been out-of-scope for this work.

12.1 Main Contributions

Completeness. I described the *complete* architecture of a programming system that provides a powerful middleware for component-based and distributed programming. Starting from a standard runtime system and a base language with well-known concepts, I defined incremental extensions as shown in Figure 12.1 to cover all layers and aspects that are often treated in an ad-hoc way. All of what this work describes has been validated in real systems, namely Mozart Oz [Moz04] and Alice [Pro05].

Minimal Runtime, Maximal Programmability. I have built the whole system from a very small set of primitives—nine in total, as shown in Figure 12.2. All of these primitives are type-safe and accessible to programmers, making it possible for them to replace all layers above the primitives

Ch.	Language	Primitives	Runtime
5–7			Pickling
8	Component Syntax		Boot Linker
9	Lazy Structure Selection; First-class Component Syntax	Operations on First-class Components	Built-in vs. System Primitives
10		Component I/O and Creation	
11		Dynamic Call	

Figure 12.1: Overview of Extensions.

```
structure Component :
sig
    exception Malformed
    exception Sited

    type component

    val pickle : component -> Word8Vector.vector
        (* Sited *)
    val unpickle : Word8Vector.vector -> component
        (* Malformed *)
    val apply : component -> package vector -> package
        (* Match, Package.Mismatch *)
    val imports : component -> (string * sign) vector
    val sign : component -> sign
    val capture : package -> component
    val component : { imports : (string * sign) vector,
                      body : package vector -> package,
                      sign : sign } -> component
end

structure DynamicCall :
sig
    val setTarget : (string * package -> package) -> unit
    val dynamicCall : string * package -> package
end
```

Figure 12.2: Overview of Primitives.

by customized variants: I illustrated this by presenting a high-level implementation of the dynamic linker (Section 9.3) and the bundler (Section 10.3).

Survey of Pickling Mechanisms. I developed a novel classification for pickling mechanisms and provided the first comprehensive survey of existing mechanisms based on this classification (Chapter 4).

A Principled Approach to Pickling. This work describes a principled approach to bottom-up pickling and unpickling, first published in Tack's thesis [Tac03] (Chapter 5).

Concurrent Pickling and State. This work is the first to discuss the interaction of concurrent pickling with state and futures (Chapter 6) and to propose design patterns for pickling-safe abstract data types.

Principled Lazy Linking. Systems with lazy linking have two things to provide runtime support for: the first access to an object can trigger a linking operation, and any access to an object (for which linking failed) can trigger an exception. Usually, systems deal with these by inserting ad-hoc runtime checks. My approach employs the mechanisms of lazy futures and failed values, which are very generally useful and available at the language level, to implement triggering of linking and error handling in a principled way (Section 9.1).

Component Representation. Since pickling as I define it supports first-class functions, I can represent components as pickles. This means that I need only a single uniform file format. This also means that a single application programmer's interface is needed to deal with both components and pickles—whereby pickles can be referenced in static import announcements, and dynamically-created components can be saved to disk (Section 10.1).

Component Localization. I propose separating logical from physical component names, and using URI and URL for them, respectively. This solves the problems of network transparency (by abstracting from locations, and by integrating the networked file system spanned by HTTP URL into component addressing), component interposition, and naming components by function. The mapping from URI to URL is, in the simplest of cases, trivial; for other practically useful mappings, I presented a simple configuration mechanism (Section 10.2).

Component Hiding. I proposed a *bundling* operation for creating a single component from a graph of existing components. This solves the problem of deploying a set of components as a single file. The operation implements real early binding, as opposed to most commonly-used approaches, allows to express irreversible hiding by preventing unbundling, and is compositional (Section 10.3).

Non-intrusive Distribution Layer. Without any extensions to either the language or the runtime system, I have shown that the system can express dynamic establishment of connections and data exchange (Section 11.2). When I extend the system to support proxy functions for remote invocation, all I need are two very simple primitives for late binding—the rest being fully expressible in the high-level language (Section 11.4).

12.2 Future Directions

Futures. My approach uses transparent futures to express lazy linking, and failed values to report a link error to all clients of the failed component. Transparent futures are, of course, a more generally useful concept, which comes at the cost of an intrinsic overhead on practically every strict operation. In dynamically-typed languages, this cost can be amortized with the cost for dynamic type checks [Sch98]. The cost in statically-typed languages still seems substantial, and there is little research to quantify or even eliminate it. The calculus by Niehren et al. [NSS05] is a good basis for research on static analysis and optimizations.

Link-time Optimizations. This issue is directly linked with the issue of cross-component optimizations. Cross-component optimizations are intrinsically hard, but even more crucial in a statically-typed language with futures, it seems. This work has only hinted at the possibilities of optimization in run-time compilers; this warrants more research. For instance, one could envision that the intermediate language used in pickles to represent code was annotated with strictness constraints. The run-time compiler, which knows the constraints both on the importing and actual components, could solve these constraints; this would tell it at which places a test for a future is really necessary.

Cross-language Pickle Format. This thesis developed a middleware for one specific instance of a programming language. The approach is cross-platform, and concepts carry over to other languages, but the one place where an *actual implementation* would be very tied to one language is the pickling format. The success of Microsoft .NET shows that application programmers want cross-language support from a middleware. To open up the present work for cross-language environments, the pickling format needs to support multiple languages, which is particularly hard for code. Some first attempts have been made to solve this [BK02].

Verifiability of Pickles. The main focus of the present work was on safety. To provide for security, one of the first areas to address would be verification of pickles, which could be stated as: Is there a valid program that could create a data graph that, when pickled, would exactly result in a given pickle? For statically-typed languages with an expressive type system, such as Alice ML, this essentially requires that the heap be type-checkable, including closures of functions. Some work has been done for verifying purely code, such as Java's byte-code verification [LY99]) or proof-carrying code [Nec97]. This does not trivially extend to closure environments. For some scenarios, cryptographic signing is an alternative to verification [PS00, RSA78]. More work is warranted in this area.

More Expressive Static Typing. In the presence of static typing, this work needed to make a number of simplifying assumptions. In particular, signatures in import announcements cannot contain free variables—a severe limitation in the presence of generic abstract types. Rossberg [Ros06b] proposes modifications to the component system to address these limitations.

A Component Calculus. This work presented a component model, one of whose virtues is its simplicity, and actually gave an informal semantics for it. On the other hand, this work—like Alice ML—builds on Standard ML, which is famous for its formal definition. Some more research is needed to bridge this gap: a formal component calculus could serve to prove type soundness of the middleware. Research in this area is ongoing [Ros06a].

References

[AAB$^+$99] Bowen Alpern, C. R. Attanasio, John J. Barton, Anthony Cocchi, Susan Flynn Hummel, Derek Lieber, Ton Ngo, Mark Mergen, Janice C. Shepherd, and Stephen Smith. Implementing Jalapeño in Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Volume 34(10) of *SIGPLAN Notices*, pages 314–324, Denver, Colorado, USA, November 1999. ACM Press.

[ACPP91] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[ACPR95] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, January 1995.

[AK91] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* The MIT Press, 1991.

[AM94] Andrew W. Appel and Dave B. MacQueen. Separate compilation for Standard ML. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Volume 29(6) of *SIGPLAN Notices*, pages 13–23, Orlando, Florida, USA, June 1994. ACM Press.

[Apa03] Apache Software Foundation. *The Apache HTTP Server Project*, 2003. `http://httpd.apache.org/`

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers—Principles, Techniques, and Tools.* Addison Wesley, 1986.

[BA99] Matthias Blume and Andrew W. Appel. Hierarchical modularity. *ACM Transactions on Programming Languages and Systems*, 21(4):813–847, 1999.

[BCLO93] Gilad Bracha, Charles F. Clark, Gary Lindstrom, and Douglas B. Orr. Module management as a system service. In *OOPSLA Workshop on Object-oriented Reflection and Metalevel Architectures*, Washington, D. C., USA, October 1993. Reviewed in OOPSLA'93 Addendum to the Proceedings, page 125.

[Bel73] James R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973.

[BH00] Sara Bouchenak and Daniel Hagimont. Pickling threads state in the Java system. In *33rd International Conference on Technology of Object-Oriented Languages (TOOLS)*, pages 22–32, Saint-Malo, France, June 2000. IEEE Computer Society Press.

[BHM+04] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard, editors. Web services architecture. Working Group Note 11, World Wide Web Consortium (W3C), February 2004. http://www.w3.org/TR/ws-arch/

[Bir05] Dorian Birsan. On plug-ins and extensible architecture. *ACM Queue*, 3(2):32–46, March 2005.

[BJW87] Andrew D. Birrell, Michael B. Jones, and Edward P. Wobber. A simple and efficient implementation for small databases. In *Proceedings of the 11th ACM Symposium on Operating System Principles (SOSP)*, Volume 21(5) of *Operating Systems Review*, pages 149–154, Austin, Texas, USA, November 1987. ACM Press.

[BK02] Thorsten Brunklaus and Leif Kornstaeadt. A virtual machine for multi-language execution. Technical report, Programming Systems Lab, Universität des Saarlandes, November 2002. http://www.ps.uni-sb.de/Papers/abstracts/multivm.html

[BK03] Thorsten Brunklaus and Leif Kornstaeadt. Open programming services for virtual machines: The design of Mozart and SEAM. Technical report, Programming Systems Lab, Universität des Saarlandes, March 2003. http://www.ps.uni-sb.de/Papers/abstracts/vmservices.html

[BLFM98] Tim Berners-Lee, Roy Fielding, and Larry Masinter. Uniform resource identifiers (URI): Generic syntax. Request for Comments 2396, Network Working Group, August 1998.

[BLMM94] Tim Berners-Lee, Larry Masinter, and Mark McCahill. Uniform resource locators (URL). Request for Comments 1738, Network Working Group, December 1994.

[Blu97] Matthias Blume. *Hierarchical Modularity and Intermodule Optimization*. Doctoral dissertation, Princeton University, November 1997. Available as Princeton University Technical Report TR-551-97.

[BMR+96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *A System of Patterns. Pattern-Oriented*

Software Architecture, Volume 1. John Wiley & Sons, August 1996.

[BNOW95] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. *Software—Practice and Experience*, 25(S4):87–130, December 1995. Also appeared as SRC Research Report 115.

[BP01] Fabian Breg and Constantine D. Polychronopoulos. Java Virtual Machine support for Object Serialization. In *Proceedings of the Joint ACM Java Grande – ISCOPE Conference*, pages 173–180, Stanford University, California, USA, June 2001. ACM Press.

[Bru03] Chris Brumme. Startup, shutdown & related matters. WebLog entry, August 2003. `http://blogs.msdn.com/cbrumme/archive/2003/08/20/51504.aspx`

[Bur84] Rod M. Burstall. Programming with modules as typed functional programming. In Institute for New Generation Computer Technology, editor, *Proceedings of the International Conference on 5th Generation Computing Systems (FGCS)*, pages 103–112, Tokyo, Japan, November 1984. Ohmsha, Ltd. and North-Holland.

[Car91] Luca Cardelli. Typeful programming. In Erich J. Neuhold and Manfred Paul, editors, *Formal Description of Programming Concepts*, IFIP State-of-the-Art Reports, pages 431–507. Springer-Verlag, 1991.

[Car95] Luca Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, January 1995.

[Car97] Luca Cardelli. Program fragments, linking, and modularization. In [POP97], pages 266–277.

[Cha96] David Chappell. *Understanding ActiveX and OLE*. Microsoft Press, 1996.

[Che70] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[CMHC03] Hervaldo S. Carvalho, Amy L. Murphy, Wendi B. Heinzelman, and Claudionor J. N. Coelho. Network-based distributed systems middleware. In *Proceedings of the 1st International Workshop on Middleware for Pervasive and Ad-Hoc Computing (MPAC)*, pages 13–20, Rio de Janeiro, Brazil, June 2003. PUC–Rio.

[COO96] *Proceedings of the 2nd USENIX Conference on Object-Oriented Technologies (COOTS)*, Toronto, Canada, June 1996. USENIX Association.

[Dea97] Drew Dean. The security of static typing with dynamic linking. In *Proceedings of the 4th ACM Conference on Computer and*

Communication Security (CCS), pages 18–27, Zurich, Switzerland, April 1997. ACM Press.

[DEC94] Digital Equipment Corporation, Systems Research Center. *Modula-3 Sources: Module `pickle/src/Pickle.i3`*, Release 3.6, 1994. `http://research.compaq.com/SRC/m3sources/html/pickle/src/Pickle.i3.html`

[DEC99] Digital Equipment Corporation, Systems Research Center. *Modula-3*, 1999. `http://research.compaq.com/SRC/modula-3/html/home.html`

[DFN03] DFN-CERT Services GmbH. *Computer Emergency Response Team (CERT) für das Deutsche Forschungsnetz (DFN)*, 2003. `http://www.cert.dfn.de/cert.html`

[DFW96] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java security: From HotJava to Netscape and beyond. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 190–200, Oakland, California, USA, May 1996. IEEE Computer Society Press.

[DKH$^+$04] Denys Duchier, Leif Kornstaedt, Martin Homik, Tobias Müller, Christian Schulte, and Peter Van Roy. *System Modules*. The Mozart Consortium, Version 1.3.1, 2004. `http://www.mozart-oz.org/documentation/system/`

[DKS04a] Denys Duchier, Leif Kornstaedt, and Christian Schulte. *Application Programming*. The Mozart Consortium, Version 1.3.1, 2004. `http://www.mozart-oz.org/documentation/apptut/`

[DKS04b] Denys Duchier, Leif Kornstaedt, and Christian Schulte. *Oz Shell Utilities*. The Mozart Consortium, Version 1.3.1, 2004. `http://www.mozart-oz.org/documentation/tools/`

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

[DLE03] Sophia Drossopoulou, Giovanni Lagorio, and Susan Eisenbach. Flexible models for dynamic linking. In Pierpaolo Degano, editor, *Proceedings of the 12th European Symposium on Programming (ESOP)*, Volume 2618 of *Lecture Notes in Computer Science*, pages 38–53, Warsaw, Poland, April 2003. Springer-Verlag.

[DP02] Márcio Delamaro and Gian Pietro Picco. Mobile code in .NET: A porting experience. In Niranjan Suri, editor, *Proceedings of the 6th International Conference on Mobile Agents (MA)*, Volume 2355 of *Lecture Notes in Computer Science*, pages 16–31, Barcelona, Spain, October 2002. Springer-Verlag.

[DSBH03] Frej Drejhammar, Christian Schulte, Per Brand, and Seif Haridi. Flow Java: Declarative concurrency for Java. In Catuscia Palamidessi, editor, *Proceedings of the 19th International Conference on Logic Programming (ICLP)*, Volume 2916 of *Lecture Notes in Computer Science*, pages 346–360, Mumbai, India, December 2003. Springer-Verlag.

[Dug02] Dominic Duggan. Type-safe linking with recursive DLLs and shared libraries. *ACM Transactions on Programming Languages and Systems*, 24(6):711–804, November 2002.

[Els99] Martin Elsmann. *Program Modules, Separate Compilation, and Intermodule Optimization*. PhD thesis, Department of Computer Science, University of Copenhagen, Copenhagen, Denmark, January 1999.

[FF98] Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Volume 33(5) of *SIGPLAN Notices*, pages 236–248, Montreal, Canada, June 1998. ACM Press.

[FGM$^+$99] Roy Fielding, Jim Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. Hypertext transfer protocol—HTTP/1.1. Request for Comments 2616, Network Working Group, June 1999.

[FPV98] Alfonso Fuggetta, Gian Pietro Picco, and Giovanni Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.

[FW00] Jun Furuse and Pierre Weis. Entrées/sorties de valeurs en Caml. In *Actes des onzièmes Journées Francophones des Langages Applicatifs (JFLA)*, Mont Saint-Michel, France, January 2000. INRIA.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[GHM$^+$03] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen, editors. Simple object access protocol (SOAP), Version 1.2 Part 1: Messaging framework. Recommendation, World Wide Web Consortium (W3C), June 2003. `http://www.w3.org/TR/SOAP/`

[GJS00] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison Wesley, Second Edition, June 2000.

[Hal85] Robert Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[Hen97] Martin Henz. *Objects for Concurrent Constraint Programming*. International Series in Engineering and Computer Science. Kluwer Academic Publishers, Boston, Massachusetts, USA, 1997.

[HK04] Martin Henz and Leif Kornstaedt. *The Oz Notation*. The Mozart Consortium, Version 1.3.0, 2004. `http://www.mozart-oz.org/documentation/notation/`

[HL82] M. Herlihy and B. Liskov. A value transmission method for abstract data types. *ACM Transactions on Programming Languages and Systems*, 4(4):527-551, October 1982.

[HM95] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Conference Record of the 22nd ACM SIGPLAN/SIGACT Symposium on the Principles of Programming Languages (POPL)*, pages 130–141, San Francisco, California, USA, January 1995. ACM Press.

[HVB⁺99] Seif Haridi, Peter Van Roy, Per Brand, Michael Mehl, Ralf Scheidhauer, and Gert Smolka. Efficient logic variables for distributed computing. *ACM Transactions on Programming Languages and Systems*, 21(3):569–626, May 1999.

[HVBS98] Seif Haridi, Peter Van Roy, Per Brand, and Christian Schulte. Programming languages for distributed applications. *New Generation Computing*, 16(3):223–261, 1998.

[INR02a] INRIA Rocquencourt, Projet Cristal. *The Caml Language*, 2002. `http://caml.inria.fr/`

[INR02b] INRIA Rocquencourt, Projet Cristal. *G'Caml—O'Caml with Extensional Polymorphism Extension*, 2002. `http://pauillac.inria.fr/~furuse/generics/`

[INR02c] INRIA Rocquencourt, Projet Cristal. *The Objective Caml Distribution, Version 3.06*, 2002. `http://caml.inria.fr/ocaml/`

[Jag94] Suresh Jagannathan. Dynamic modules in higher-order languages. In Henri E. Bal, editor, *Proceedings of the IEEE International Conference on Computer Languages (ICCL)*, Toulouse, France, May 1994. IEEE Computer Society Press.

[Joh85] Thomas Johnsson. Lambda lifting: Transforming programs to recursive equations. In Jean-Pierre Jouannaud, editor, *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA)*, Volume 201 of *Lecture Notes in Computer Science*, pages 190–203, Nancy, France, September 1985. Springer-Verlag.

[KEBH03] Erik Klintskog, Zacharias El Banna, Per Brand, and Seif Haridi. The design and evaluation of a library of distribution of language entities. In Vijay A. Saraswat, editor, *Proceedings of the*

8th Asian Computing Science Conference (ASIAN), Volume 2896 of *Lecture Notes in Computer Science*, pages 243–259, Mumbai, India, December 2003. Springer-Verlag.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Software Series. Prentice Hall, Second Edition, March 1988.

[LB98] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java Virtual Machine. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Volume 33(10) of *SIGPLAN Notices*, pages 36–44, Vancouver, British Columbia, Canada, October 1998. ACM Press.

[LCC$^+$75] Roy Levin, Ellis S. Cohen, William M. Corwin, Fred J. Pollack, and William A. Wulf. Policy/mechanism separation in Hydra. In *Proceedings of the 5th ACM Symposium on Operating Systems Principles (SOSP)*, Volume 9(5) of *Operating Systems Review*, pages 132–140, Austin, Texas, USA, November 1975. ACM Press.

[Lea99] Doug Lea. *Concurrent Programming in Java. Design Principles and Patterns.* Addison Wesley Professional, 2nd Edition, 1999.

[Ler90] Xavier Leroy. The ZINC experiment: An economical implementation of the ML language. Technical Report 117, INRIA Rocquencourt, February 1990.

[Lev84] Henry M. Levy. *Capability-based Computer Systems.* Digital Press, 1984. `http://www.cs.washington.edu/homes/levy/capabook/`

[Lev00] John R. Levine. *Linkers and Loaders.* Morgan Kaufmann, January 2000. `http://www.iecc.com/linker/linker10.html`

[Lis93] Barbara Liskov. A history of CLU. In *Proceedings of the Second ACM SIGPLAN Conference on History of Programming Languages (HOPL)*, Volume 28(3) of *SIGPLAN Notices*, pages 133–147, Cambridge, Massachusetts, USA, April 1993. ACM Press.

[LPSW03] James J. Leifer, Gilles Peskine, Peter Sewell, and Keith Wansbrough. Global abstraction-safe marshalling with hash types. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Volume 38(9) of *SIGPLAN Notices*, pages 87–98, Uppsala, Sweden, August 2003. ACM Press.

[LS79] High C. Lauer and Edwin H. Satterthwaite. The impact of Mesa on system design. In *Proceedings of the 4th International Conference on Software Engineering (ICSE)*, pages 174–182, Munich, Germany, September 1979. IEEE Computer Society Press.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison Wesley, 2nd edition, April 1999.

[LZ74] Barbara H. Liskov and Stephen N. Zilles. Programming with abstract data types. *SIGPLAN Notices*, 9(4):50–59, 1974.

[McI68] M. Douglas McIlroy. Mass produced software components. In *Software Engineering—Report on a Conference Sponsored by the NATO Science Committee*, pages 138–155, Garmisch, Germany, October 1968.

[Meh99] Michael Mehl. *The Oz Virtual Machine: Records, Transients, and Deep Guards.* Doctoral dissertation, Technische Fakultät, Universität des Saarlandes, Saarbrücken, Germany, 1999.

[MFH01] Sean McDirmid, Matthew Flatt, and Wilson C. Hsieh. Jiazzi: New-age components for old-fashioned Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Volume 36(11) of *SIGPLAN Notices*, pages 211–222, Tampa, Florida, USA, November 2001. ACM Press.

[Mic02] Microsoft Corporation. *Internet Explorer*, Version 6 Service Pack 1, September 2002. `http://www.microsoft.com/ie/`

[Mic03a] Microsoft Corporation. *Microsoft .NET*, Version 1.1, 2003. `http://www.microsoft.com/net/`

[Mic03b] Microsoft Corporation. *Microsoft .NET Framework Class Library: Namespace `System.Reflection.Emit`*, 2003. `http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemreflectionemit.asp`

[Mic03c] Microsoft Corporation. *Microsoft .NET Framework Class Library: Namespace `System.Runtime.Remoting`*, 2003. `http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeremoting.asp`

[Mic03d] Microsoft Corporation. *Microsoft .NET Framework Class Library: Namespace `System.Runtime.Serialization`*, 2003. `http://msdn.microsoft.com/library/en-us/cpref/html/frlrfsystemruntimeserialization.asp`

[Mic03e] Microsoft Corporation. *Microsoft .NET: Programming with Application Domains*, 2003. `http://msdn.microsoft.com/library/en-us/cpguide/html/cpconprogrammingwithapplicationdomains.asp`

[Moz04] The Mozart Consortium. *The Mozart Programming System*, Version 1.3.1, June 2004. `http://www.mozart-oz.org/`

[MSS98] Michael Mehl, Christian Schulte, and Gert Smolka. Futures and by-need synchronization. Technical report, Programming Systems Lab, DFKI, and Universität des Saarlandes, May 1998.

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

[MW91] Hanspeter Mössenböck and Niklaus Wirth. The programming language Oberon-2. *Structured Programming*, 12(4):179–195, 1991.

[MzS04] MzScheme Version 209, December 2004. `http://www.plt-scheme.org/software/mzscheme/`

[Nec97] George C. Necula. Proof-carrying code. In [POP97], pages 106–119.

[Nel91] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall Series in Innovative Technology, 1991.

[NSS05] Joachim Niehren, Jan Schwinghammer, and Gert Smolka. A concurrent lambda calculus with futures. In Bernhard Gramlich, editor, *Proceedings of the 5th International Workshop on Frontiers of Combining Systems (FroCoS)*, Volume 3717 of *Lecture Notes in Computer Science*, pages 248–263, Vienna, Austria, September 2005. Springer-Verlag.

[Oka98] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[PH00] M. Philippsen and B. Haumacher. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7):495–518, May 2000.

[Pil96] Marco Pil. First class file I/O. In Werner E. Kluge, editor, *Proceedings of the 8th International Workshop on the Implementation of Functional Languages (IFL), Selected Papers*, Volume 1268 of *Lecture Notes in Computer Science*, pages 233–246, Bad Godesberg, Gemany, September 1996. Springer-Verlag.

[POP97] *Conference Record of the 24th ACM SIGPLAN/SIGACT Symposium on the Principles of Programming Languages (POPL)*, Paris, France, January 1997. ACM Press.

[Pro05] Programming Systems Lab, Universität des Saarlandes. *The Alice Programming System*, Version 1.1, March 2005. `http://www.ps.uni-sb.de/alice/`

[PS00] Benjamin C. Pierce and Eijiro Sumii. Relating cryptography and polymorphism. Manuscript, July 2000. Substantially revised version to appear in Journal of Computer Security.

[Pyt05] Python Software Foundation. *Python Version 2.3.5*, 2005. `http://www.python.org/2.3.5/`

[QR96] Christian Queinnec and David De Roure. Sharing code through first-class environments. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Volume 31(6) of *SIGPLAN Notices*, pages 251–261, Philadelphia, Pennsylvania, USA, May 1996. ACM Press.

[Ric97] Jeffrey Richter. *Advanced Windows*. Microsoft Press, 1997.

[Riv92] Ronald L. Rivest. The MD5 message digest algorithm. Request for Comments 1321, Network Working Group, April 1992.

[RLT$^+$04] Andreas Rossberg, Didier Le Botlan, Guido Tack, Thorsten Brunklaus, and Gert Smolka. Alice through the looking glass. In *Proceedings of the Symposium on Trends in Functional Programming (TFP)*, Munich, Germany, November 2004. Intellect.

[Rog97] Dale Rogerson. *Inside COM*. Programming Series. Microsoft Press, February 1997.

[Ros02] Andreas Rossberg. *The Stockhausen Intermediate Language*. Programming Systems Lab, Universität des Saarlandes, August 2002. `http://www.ps.uni-sb.de/alice/papers/intermediate.ps`

[Ros03] Andreas Rossberg. Generativity and dynamic opacity for abstract types. In Dale Miller, editor, *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 241–252, Uppsala, Sweden, August 2003. ACM Press.

[Ros04] Andreas Rossberg. *A Tour to Wonderland*. Programming Systems Lab, Universität des Saarlandes, 2004. `http://www.ps.uni-sb.de/alice/manual/tour.html`

[Ros05] Andreas Rossberg. The definition of Standard ML with packages. Technical report, Universität des Saarlandes, Saarbrücken, Germany, April 2005. `http://www.ps.uni-sb.de/Papers/abstracts/sml-with-packages.html`

[Ros06a] Andreas Rossberg. The missing link—dynamic components for ML. Technical report, Universität des Saarlandes, Saarbrücken, Germany, 2006. Submitted.

[Ros06b] Andreas Rossberg. *Typed Open Programming*. Doctoral dissertation, Naturwissenschaftlich-Technische Fakultäten der Universität des Saarlandes, Saarbrücken, Germany, 2006. In preparation.

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[RWW96] Roger Riggs, Jim Waldo, and Ann Wollrath. Pickling state in the Java system. In [COO96].

[Sch98] Ralf Scheidhauer. *Design, Implementierung und Evaluierung einer virtuellen Maschine für Oz*. Doctoral dissertation, Technische Fakultät, Universität des Saarlandes, Saarbrücken, Germany, December 1998.

[Sch02a] Christian Schulte. *Programming Constraint Services*. Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, Germany, 2002.

[Sch02b] Jan Schwinghammer. A concurrent lambda calculus with promises and futures. Diplomarbeit, Naturwissenschaftlich-Technische Fakultät I, Fachrichtung Informatik, Universität des Saarlandes, Saarbrücken, Germany, January 2002.

[Sed83] Robert Sedgewick. *Algorithms*. Addison Wesley, Third Edition, 1983.

[Sew01] Peter Sewell. Modules, abstract types, and distributed versioning. In *Conference Record of the 28th ACM SIGPLAN/SIGACT Symposium on the Principles of Programming Languages (POPL)*, Volume 36(3) of *SIGPLAN Notices*, pages 246–247, London, United Kingdom, January 2001. ACM Press.

[Sha98] Zhong Shao. Typed cross-module compilation. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Volume 34(1) of *SIGPLAN Notices*, pages 141–152, Baltimore, Maryland, September 1998. ACM Press.

[SML02] The SML/NJ Fellowship. *Standard ML of New Jersey (SML/NJ)*, 2002. `http://www.smlnj.org/`

[Smo95] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, Volume 1000 of *Lecture Notes in Computer Science*, pages 324–343. Springer-Verlag, 1995.

[Smo98] Gert Smolka. Concurrent constraint programming based on functional programming. In Chris Hankin, editor, *Proceedings of the 7th European Symposium on Programming (ESOP)*, Volume 1381 of *Lecture Notes in Computer Science*, pages 1–11, Lisbon, Portugal, March 1998. Springer-Verlag.

[Sof04] Software Technology Research Group, Katholieke Universiteit Nijmegen. *Clean Version 2.1*, October 2004. `http://www.cs.kun.nl/~clean/`

[Ste97] W. Richard Stevens. *Unix Network Programming.* Prentice Hall, 2nd Edition, 1997.

[Str00] Bjarne Stroustrup. *The C++ Programming Language.* Addison Wesley, Special 3rd Edition, 2000.

[Sun87] Sun Microsystems, Inc. XDR: External data representation standard. Request for Comments 1014, Network Working Group, June 1987.

[Sun01] Sun Microsystems, Inc. *Java Object Serialization Specification*, 2001. `http://java.sun.com/j2se/1.4/docs/guide/serialization/`

[Sun03a] Sun Microsystems, Inc. *J2EE Glossary*, 2003. Version 1.4. `http://java.sun.com/j2ee/1.4/docs/glossary.html`

[Sun03b] Sun Microsystems, Inc. *Java 2 Platform SE: Class `Class`*, Version 1.4.2, 2003. `http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Class.html`

[Szy02] Clemens Szyperski. *Component Software—Beyond Object-Oriented Programming.* Addison Wesley and ACM Press, Second Edition, 2002.

[Tac03] Guido Tack. Linearisation, minimisation and transformation of data graphs with transients. Diplomarbeit, Naturwissenschaftlich-Technische Fakultät I, Fachrichtung Informatik, Universität des Saarlandes, Saarbrücken, Germany, May 2003.

[Tar72] Robert E. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[TC301a] TC39/TG2. C# language specification. Technical Report 334, ECMA, December 2001. `http://msdn.microsoft.com/net/ecma/`

[TC301b] TC39/TG2. Common Language Infrastructure (CLI). Technical Report 335, ECMA, December 2001. `http://msdn.microsoft.com/net/ecma/`

[TKS05] Guido Tack, Leif Kornstaedt, and Gert Smolka. Generic pickling and minimization. In *Proceedings of the ACM SIGPLAN Workshop on ML*, Volume 148(2) of *Electronic Notes in Theoretical Computer Science*, pages 79–103, Talinn, Estonia, September 2005. Elsevier Science Publishers.

[VHB$^+$97] Peter Van Roy, Seif Haridi, Per Brand, Gert Smolka, Michael Mehl, and Ralf Scheidhauer. Mobile objects in distributed Oz. *ACM Transactions on Programming Languages and Systems*, 19(5):804–851, September 1997.

[VP02] Martijn Vervoort and Rinus Plasmeijer. Lazy dynamic input/ output in the lazy functional language Clean. In Ricardo Pena and Thomas Arts, editors, *Proceedings of the 14th International Workshop on the Implementation of Functional Languages (IFL), Selected Papers*, Volume 2670 of *Lecture Notes in Computer Science*, Madrid, Spain, September 2002. Springer-Verlag.

[WAS00] Ken Wakita, Takashi Asano, and Masataka Sassa. D'Caml: Native support for distributed ML programming in heterogeneous environment. In Arndt Bode, Thomas Ludwig, Wolfgang Karl, and Roland Wismüller, editors, *Proceedings of the 5th International Euro-Par Conference on Parallel Processing*, Volume 1900 of *Lecture Notes in Computer Science*, pages 914–924, Munich, Germany, August 2000. Springer-Verlag.

[WBDF97] Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible security architectures for Java. In *Proceedings of the 16th ACM Symposium on Operating System Principles (SOSP)*, Volume 31(5) of *Operating Systems Review*, pages 116–128, Saint-Malo, Fance, October 1997. ACM Press.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In Yves Bekkers and Jacques Cohen, editors, *Proceedings of the International Workshop on Memory Management (IWMM)*, Volume 637 of *Lecture Notes in Computer Science*, pages 1–42, Saint-Malo, France, September 1992. Springer-Verlag.

[Wir95] Niklaus Wirth. A brief history of Modula and Lilith. *The ModulaTor*, 5(5), January 1995. `http://www.modulaware.com/mdlt52.htm`

[WRW96] Ann Wollrath, Roger Riggs, and Jim Waldo. A distributed object model for the Java System. In [COO96].