Open Programming Services for Virtual Machines

The Design of Mozart and SEAM

Thorsten Brunklaus
Programming Systems Lab
Saarland University
Postfach 15 11 50
66041 Saarbrücken, Germany
brunklaus@ps.uni-sb.de

Leif Kornstaedt
Programming Systems Lab
Saarland University
Postfach 15 11 50
66041 Saarbrücken, Germany
kornstaedt@ps.uni-sb.de

ABSTRACT

This paper discusses designs for integrating services in general and open programming services in particular into virtual machines. We draw on our experience with two systems. The first is Mozart, a programming system implementing the language Oz. Mozart's virtual machine provides a rich set of services for open programming, such as concurrency, persistence of data and code, components with dynamic linking, and distribution. The second system is Alice, which is at the same time a statically-typed variant of Oz, and an extension of Standard ML for open programming. Our design proposals for open programming services culminate in the definition of a virtual machine called SEAM, which we claim to be simple, efficient, and extensible. We substantiate these claims with preliminary performance results.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Design Studies; D.2.11 [**Software Engineering**]: Software Architecture

General Terms

Design, Languages

Keywords

Virtual Machines, Experience Report, Lessons Learned

1. INTRODUCTION

Mozart [14] is an open programming system implementing the programming language Oz. Oz is a rich multi-paradigm language supporting the idioms of logic, concurrent, functional, and object-oriented programming. Mozart is based on a virtual machine that provides services required for *open programming*. By this, we mean features that make an application able to interact in non-trivial ways with other applications or application fragments that were unknown at the time of implementing the application. Open programming features include persistence, components with lazy dynamic linking, and distribution.

This paper presents our insights to the design of services and their integration into virtual machines in general, and of services for open programming in particular. This research was motivated by experience gained in two projects. The first is the implementation of the Mozart virtual machine [12, 9]. It started as a modification of the Warren Abstract Machine [1], but its first approach not even foresaw garbage collection. While early versions were founded on a clean model [10], the virtual machine wildly grew as it tried to keep pace with quickly changing requirements. We and our fellow implementors often needed to re-think design decisions in order to accomodate new features, and occasionally faced design problems that could not be solved satisfyingly in the existing system.

The other project is the design and implementation of a new language called Alice [2]. Conceptually, Alice is at the same time a statically-typed variant of Oz, and an extension of Standard ML [11] incorporating open programming features à la Oz. The first Alice prototype was implemented by translation to Mozart bytecode [7], with the additional goal that both languages interoperate gracefully. In the course of building the prototype, we were able to take a new viewpoint on the Mozart virtual machine. This raised the wish for an improved virtual machine, with a more rational approach to services, and more generally suited to support other programming languages than Oz.

In this paper, we assess the implementation of the Mozart virtual machine, focusing on the domain of open programming. We identify which design decisions have proven themselves and which turned out to be hard to maintain or hard to extend. We complement this critique by presenting a revised design that addresses Mozart's shortcomings and emphasizes simplicity, efficiency, genericity, and extensibility. We have implemented this design in a virtual machine called SEAM [4]. We have a running prototype of SEAM that can execute full Alice. Performance of Alice on SEAM is roughly comparable to that of Alice on Mozart.

This paper is structured as follows. In Section 2, we describe which language features we consider essential to enable open

programming and why. Section 3 gives a high-level overview of programming systems and virtual machines, and presents principles for service design. We take a closer look at the core components of the Mozart virtual machine in Section 4, to provide a good infrastructure for services. Section 5 examines our open programming services. We describe what from our experience has proven itself and what has not, and from this experience present revised design decisions where appropriate. Section 6 summarizes the current status of SEAM and presents a promising evaluation of our prototype. Section 7 concludes the paper.

2. LANGUAGE FEATURES FOR OPEN PROGRAMMING

The emphasis in open programming lies on two things: interaction, and being dynamic. For example, a Web browser supporting plug-ins to extend its functionality (for instance to handle new content types) at run-time is open, a Web server allowing new handlers for specific resources to be added and replaced dynamically is open, and a compute server that allows clients to submit arbitrary program fragments as compute requests is open.

The following language and system features that we consider essential for open programming have emerged from Oz and Mozart. Alice implements these same features, although in the context of a statically-typed programming language.

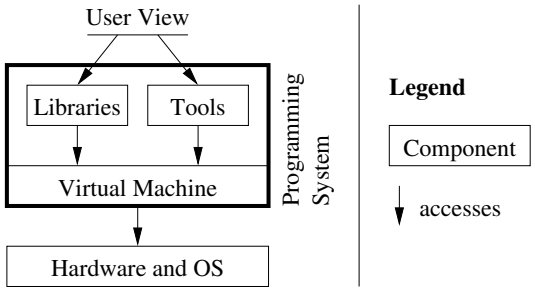
Concurrency dramatically simplifies the handling of multiple simultaneous connections, such as to files, graphical windows, clients, servers, or peers. To make it possible for a number of concurrent threads to cooperate, programming systems have to provide synchronization primitives.

Dynamic linking is the composition at run-time of program fragments known as *components*, and provides for configurability and extensibility. In Oz and Alice, dynamic linking is performed by module managers (resp. component managers). A single running system can have arbitrarily many module managers—in other words, it can link components in separate, configurable namespaces. This provides for sandboxing. Component names are represented as Uniform Resource Identifiers (URIs) [3], are available as first-class entities and can be computed at run-time. This enables, for instance, the realization of plug-ins and late registration of handlers. Components can automatically be downloaded from Web servers via HTTP.

Persistence (or pickling, or serialization) of language entities provides a means to define file types (file formats) and communication protocols by means of expressive language-level data structures, instead of just bits and bytes. In particular, Oz and Alice offer persistence of graph-structured language data, maintaining coreferences and cycles.

Mobile code denotes the ability to transfer first-class procedures, that is, closures together with their code, to other processes. A receiving process neither needs to know the code beforehand, nor does it need to be able to locate it upon reception of the closure. Mobile code makes it possible to define higher-order communication protocols and enables implementation of expressive mobile agents. In Oz and Alice, mobile code is obtained simply by defining pickling to operate on arbitrary higher-order data structures (data structures containing first-class procedures).

Figure 1: Programming System Architecture



Network-transparent distribution makes it irrelevant to inter-connected computations whether they operate on local or remote data. A distribution subsystem manages automatic establishment of connections and exchange of language data structures. The previously named language features already allow the implementation of simple distribution support at the language level. If we require network transparency on more data types than are supported by pickling, or other distribution behaviours than cloning (for example, stationary procedures with a remote procedure call), then this requires additional support of the virtual machine. We do not in this paper consider Mozart's distribution layer in detail.

Linguistic reflection gives a programming system the ability to generate new program fragments and incorporate them into the ongoing computation [13]. This requires a language-level interface to the compiler. Linguistic reflection serves to accomodate user interaction and configuration using the high-level language. Examples are evaluation of queries in a command shell, an interactive top-level, or a source-level debugger, or implementation of PHP- or ASP-like Web server "pages" with dynamic recompilation.

3. ARCHITECTURE OF PROGRAMMING SYSTEMS

A programming system for a (high-level) language comprises, besides a compiler and a run-time system for the language, libraries and tools for developing applications, as depicted in Figure 1. One central feature of open programming is dynamic exchange of data and code. For this reason, the user of the high-level language has no way to access the features of the underlying hardware and operating system but through abstractions. In other words, the programming system defines a *virtual machine* that provides a system-independent view of the concrete machine, and programs targeting the virtual machine can run on any concrete machine for which an implementation of the virtual machine exists.

The virtual machine needs to be implemented in a language that is close to the underlying concrete machine in order to implement the abstract interface. Using C or C++ allows to easily port the virtual machine to a variety of architectures. In the following, we will speak of the *low-level language* when

we mean the language in which the virtual machine is implemented.

Since the user-visible tools and libraries are often complex, they are usually implemented in part in the low-level language and in part in the high-level language. The interface between the high-level and low-level parts consists of a number of primitives. We call a *service* of the virtual machine the set of primitives required to implement a language-level feature.

Only the virtual machine and its services need to be ported when targeting a new platform. It is therefore desirable to keep these as small as possible. Typically, only when a feature is well-understood can one define a good virtual machine service for it—one that is small while allowing an efficient implementation of the feature.

In Section 3.1, we describe the structure of the core virtual machine. Section 3.2 describes different ways how services can be added to the core virtual machine, and their drawbacks and advantages.

3.1 The Core Virtual Machine

The core virtual machine consists of a number of components. The *store* provides a model of data representation and memory management. The *scheduler* coordinates the execution of concurrent threads. The *execution unit* actually executes code. The *I/O subsystem* abstracts away the interface to the operating system's input and output channels.

Store. The data structures used by computations reside in the store. Conceptually, the store represents a graph of data nodes; the implementation of the store manages allocation of these nodes and their layout in computer memory. During program execution, a number of these nodes is directly referenced from the program's environment. The set of these nodes called the *root set*. Since nodes in the store need not be explicitly deallocated, memory needs to be reclaimed periodically according to a given policy, by a process called *garbage collection*. Garbage collections can take place at specified points during program execution called *synchronization points*. The memory occupied by all nodes not reachable directly or indirectly through edges of the store graph is made available for allocation again.

Scheduler. Concurrency is supported through interleaved execution of several *threads*, each of which maintains its own task stack. The *scheduler* maintains a queue of threads which are passed to the execution unit for execution in a round-robin fashion. The threads in the queue are said to be *runnable*; while a thread is being executed by the execution unit, we say it is *running*. At each synchronization point, the execution unit will preempt execution of the current thread if a flag in the *status register* becomes set. The status register is a vector of flags, each of which signals an asynchronously raised condition which requires synchronous handling, that is, while no thread is being executed. One of the status register flags is periodically set by a timer to achieve time-slicing for fair preemptive scheduling of threads. Another status register flag is set by the store to signal the need for a garbage collection.

Execution Unit. Programs are compiled to bytecode and are executed by an *interpreter*. A procedure call creates an activation record for the called procedure. Activation records are managed in a *task stack*.

Concurrent I/O and Synchronization. An *input/output subsystem* abstracts the details of the handling of communication channels with the environment of the virtual machine process, which may be specific to the operating system. When a thread waits for an input/output channel to become ready, it is said to be *blocked*. Runnable threads continue to be executed while other threads are blocked. When input/output channels become ready, the threads waiting for them become runnable again, that is, they are enqueued in the scheduler's thread queue again.

Communication between concurrent threads requires synchronization, and thus can also lead to blocking of threads. Specifically, Mozart provides for logic variables and futures for synchronization.

3.2 Architecture of Services

As outlined above, the virtual machine provides a number of services on top of its core components. We want to distinguish between two design principles for services that differ in how a service relates to the core.

We say a service is a *stand-alone service* if its realization is independent of the core virtual machine, and it provides its own infrastructure for memory management and execution control. Computation within the service is atomic from the scheduler's point of view. Only the computation's results are communicated to the store.

In contrast, an *integrated service* reuses the core virtual machine as its infrastructure. In particular, the service allocates its data structures in the store and its computations execute under fine-grained control of the scheduler.

The following paragraphs discuss advantages and disadvantages of using stand-alone vs. integrated services.

Stand-alone Services. Designing a service to be stand-alone offers several advantages. The designer of a stand-alone service can use the data representation best suited for the service, instead of being constrained by the virtual machine's store. This allows for optimal efficiency and expressivity. Stand-alone services can use any external libraries, reducing design and implementation effort. Finally, it is easy to add new or experimental services to the virtual machine as stand-alone services, because they can be implemented independently (even when they are not yet deeply understood).

On the other hand, it is difficult for stand-alone services to interoperate with other services on the virtual machine in a fine-grained way. For instance, stand-alone services are atomic, that is, they do not interoperate with the core virtual machine's concurrency features. One way out is to split the service into a number of (atomic) sub-services, so that each has a short runtime and does not interfere noticeably with preemptive scheduling. Such a partition may be hard to find, or increase complexity.

Another disadvantage of stand-alone services is that they may increase total size of low-level code, because each carries a full implementation of its own infrastructure. This may result in maintenance and consistency problems.

Integrated Services. Reusing the core infrastructure obviates the need of infrastructure duplication for each service. Most importantly, automatic memory management comes for free with reuse of the store. Integrated services interoperate with the core virtual machine's concurrency model. The resulting implementation may be smaller (if the service had good potential for reuse) and may be perceived as more elegant, because it focuses on implementing the service itself and not some infrastructure.

The difficulty with designing an integrated service is that the service's data and control structures need to be modeled in terms of what the core virtual machine offers. For instance, data needs to be allocated as store nodes, possibly resulting in less efficient representations because of the overhead inherent to store nodes. Resources need to be wrapped into store nodes and handled by finalization (we call any entity that is not fully managed by the store a *resource*, for example an area of heap-allocated memory outside the store). In particular, the implementor may suffer from wrapping overhead with any external libraries he may want to use, since any data managed by the external library are resources. Some libraries may even not be usable at all in the implementation of an integrated service, because they may be fundamentally incompatible with the core virtual machine (for instance, if a library function can block the process, then it is not compatible with the virtual machine's idea of concurrency).

4. DESIGN OF CORE COMPONENTS

Development of the Mozart virtual machine started shortly after the design of the Oz language. Oz then had none of the open programming features except for (at the time, implicit and fine-grained) concurrency. The virtual machine implementation tried to keep pace with the fast development of Oz and many services were at first only experimental. For these reasons, most services are stand-alone. Now that the open programming features are well-understood, we want to investigate how they can be redesigned as integrated services. Efficiency and genericity of the core components is especially important for integrated services. This section takes a closer look at some of the core components to make them easier to *use and reuse* by services.

4.1 Store and Data Representation

Mozart's Approach. Since Oz is a dynamically-typed language, run-time type tests are frequent and need to be efficient. Mozart's store therefore uses a two-level tagging scheme. Pointers are tagged with three- to four-bit *primary* tags that allow the distinction of the data types occurring most frequently, such as logic variables, list components, atoms, or records. A designated primary tag delegates node type representation to a 16-bit secondary tag stored in a header word in the heap-allocated node. A designated secondary tag stands for an *extension* node, a wrapper for custom data types managed by means of user-defined virtual functions.

The store is managed by a stop-and-copy garbage collector. Each node type is defined by a C++ class with (non-virtual) methods for copying nodes, marking them, storing forward pointers, and recursing on the contained pointers to other nodes.

Experience. In Mozart, the only design criteria for data structures were compactness and efficiency. Thus, from the perspective of memory management, the nodes have an ad-hoc definition. As a consequence, that allowed only for a simple, recursively-defined garbage collector that consumes C++ stack space. Classic techniques such as Cheney-style scanning [15] were not possible because nodes are not self-describing with respect to embedded pointers. Oz classes are Oz data structures, and therefore suffer from the lack of generational garbage collection. To compensate, class creation was optimized to trade time for space efficiency, and the system libraries minimize use of classes or precompute them.

Due to the ad-hoc definition of node types, every modification to the representation of data structures or introduction of a new node type requires re-thinking and adapting memory management. Even users defining extension nodes are burdened with memory management issues.

The complex tagging scheme, whose two-level organization is visible to all clients using store, is hard to modify. Overhauling it to extend the address space from 512 MB (full of platform dependencies) to the full range of 32-bit addresses has been a daunting task; now Mozart pays the penalty of aligning all store nodes to double-words. More by luck than by design, the most frequent of the nodes identified by primary tags fit into multiples of two words. A port to 64-bit architectures is still outstanding.

Revised Design. SEAM aims for a reusable store. Its uniform nodes à la ZINC [8] are generic, thus easy to use: Each node is either tagged as an integer or as a heap block. The header word present in all heap blocks specifies size and a second-level tag, and all component words are tagged. Thus, heap blocks are self-describing and can be treated uniformly, allowing for a simple, Cheney-style garbage collector, and making the implementation of techniques such as generational garbage collection maintainable. In summary, this makes reuse of the store attractive. New data types simply identify themselves using the second-level tag, thus they are cheap, easy to introduce, and do not need to care about memory management: This is essential for successful reuse. Porting this scheme to 64-bit architectures is straightforward. There is a single *but*: The additional header word imposes an overhead on small blocks, for instance for list components.

4.2 Code Representation

Mozart's Approach. Code in the Mozart virtual machine is a bytecode[1] with a register-based instruction set strongly inspired by that of the Warren Abstract Machine (WAM) [1]. Self-modifying instructions perform specialization with respect to actual values of constants unknown at compile time.

[1]Conforming to common use, we use the term *bytecode* to stand for the representation of intermediate languages used in platform-independent object files, even if opcodes and operands are not encoded as sequences of bytes.

Until Version 2, Oz had implicit concurrency—this made environment trimming as performed in the WAM impractical, which is why register liveness information is needed for accurate garbage collection. Liveness information is not represented explicitly, however: It is computed at run-time as required and cached.

Experience. The register-based bytecode, together with a language that implicitly initializes program variables with unbound logic variables, led to a number of hard-to-find compiler bugs: Incorrect allocation of registers almost always led to deadlocks that manifest themselves arbitrarily late. A stack-based bytecode would have made the compiler (and maybe the interpreter) significantly simpler; especially since we never intended to have native code compilation, where it is argued that a stack-based bytecode is at a disadvantage.

Although the compiler knows accurate liveness information for the code it generates, it just throws it away: To compensate, during garbage collection, liveness analysis must again be performed for procedures that have a live activation record. Mozart uses a non-linear liveness analysis algorithm that only computes an approximation and not accurate data. This precludes compiler optimizations such as dead code elimination after calls to built-ins statically known never to return (such as for raising exceptions). Since liveness information is performed during garbage collection, no Oz data structures may be used (the store is not in a state that would allow it). Instead, analysis data is managed manually using C++ heap allocation.

Revised Design. SEAM's bytecode is designed for efficient run-time compilation. This provides a reusable infrastructure for optimizations that may be required by services, as can be seen later. The bytecode is in static single assignment (SSA) form, which can easily be mapped to either stack-based or register-based architectures, and is the basis for many optimization techniques. Instead of the program counter progressing in sequence, each instruction carries an explicit reference to its successor. In other words, instructions span a directed acyclic graph—acyclic because the compiler transforms loops into tail-recursive procedures, and procedure calls are always made through identifiers. It is therefore not necessary to reconstruct an intermediate graph representation for run-time code generation.

4.3 Concurrency

Mozart's Approach. Mozart implements threads in the virtual machine instead of using native threads managed by the operating system, and minimizes them in terms of space (especially concerning their stack). New threads are created by a built-in procedure that expects as a nullary first-class procedure the computation to execute. Furthermore, the implementation guarantees fairness, which is typically not offered by operating system threads.

Experience. Light-weight threads were a success, because they made heavily concurrent designs possible: Instead of restricting concurrency to where it is absolutely needed, concurrency can be employed as a full-fledged design mechanism.

Defining thread creation on first-class procedures introduced a subtle form of memory leak. The instruction set manages its environment in three register banks, namely temporary "X" registers, local "Y" registers, and the closure (G registers). Liveness analysis is only performed on X registers. In particular, G registers are always kept live, because it must be assumed that the closure can be applied again and again (which, in thread creations, is usually not the case). A common open programming idiom is to create a stream of values (that is, a non-terminated list that grows concurrently) and to spawn a separate thread to iterate over them. This often leads to situations where the head of the list is kept live in a Y or G register, and therefore, no part of the list ever becomes garbage.

Revised Design. Threads with strong guarantees, namely their light weight and fairness, may provide a useful infrastructure for services, and are therefore kept in SEAM. The bytecode's SSA form with accurate liveness information for the local environment avoids one kind of memory leak. Having an explicit thread creation instruction in the bytecode eliminates the other kind of memory leak, since no closure needs to be constructed.

5. SUPPORT FOR OPEN PROGRAMMING

This section is structured by language features. For each language feature except for concurrency, which is provided as a core component, we describe the supporting virtual machine service offered by Mozart, and revise its design.

5.1 Dynamic Linking

To discuss dynamic linking, we first define what a component is and how components are linked. We then examine how components are located at run-time, followed by optimizations to make component-based programming efficient.

5.1.1 Components and Linking

Mozart's Approach. In Mozart, a component is an Oz record consisting of an import specification, an export specification, and its body [6]. Linking consists of applying the unary procedure to a record of the imported modules. The body actually evaluates the component and returns the exported module. Static and (lazy) dynamic linking use identical principles. Linking is performed by a *component manager* implemented in Oz [5].

Experience. Representing components as values of the high-level language had major advantages. The tools operating on components are simple to implement in the high-level language; besides the component manager, this includes a static linker. The virtual machine needs no support for components except for loading a component. Built-in operations of the virtual machine are nicely integrated into this model as built-in components. Also, booting the virtual machine becomes simple: Mozart boots from a component that has no imports except for a procedure that provides access to the built-in components. The boot component initializes the full-fledged component manager, and starts an application (also a component). Typical programming systems for high-level languages, in contrast, boot from a heap image. The heap image needs to be prepared by some magic outside the high-level language. Heap images do not provide a basis for the construction of a component system.

Revised Design. The design above has proven itself, and it already maximizes reuse of the core, so we do not see a need to revise it.

5.1.2 Locating Components

Mozart's Approach. In Mozart, components and other resources are denoted by URIs [3]. At run-time, a resource URI needs to be *localized*, for example, via a HTTP download, to an actual file from which the resource contents can be obtained. Localization consists of rewriting the abstract URI to a concrete URL (Uniform Resource Locator), and accessing the resulting URL. Rewriting is implemented in Oz, but downloading in C++, so that it can also be used for the boot component.

Experience. The implementation of downloading in the low-level language, as opposed to the high-level language, is complex (it needs to resort to processes or system threads) and hard to extend. FTP and HTTP download duplicates input/output handling and, to be compatible with concurrency, requires a separate process (under Unix) resp. system thread (under Windows). The feature that required a low-level implementation, namely downloading of the boot component, never actually proved useful.

Revised Design. In our revised design, we restrict the URI of the boot component to directly point to a local file. We therefore need no low-level implementation of downloading. Instead, everything is designed for an implementation in the high-level language and uses the standard input/output primitives—in other words, is completely based on reuse of the core.

5.1.3 Optimizations

Mozart's Approach. Components were introduced into Oz after the virtual machine's bytecode was already defined and optimized. Many of the optimizations, such as first-order procedure application and the above-mentioned self-modifying instructions, only apply at the top-level of a program and not within procedures. To compensate for the performance loss induced because component bodies *are* procedures, component bodies are marked as procedures that require *instantiation* at the time of application. Instantiation consists of copying the code of the procedure, replacing placeholders for the values they define (closures and names) by fresh values. This re-enables optimizations, for instance, higher-order applications are turned into first-order applications again.

Experience. Code instantiation optimizes intra-module calls, therefore produces good results for coarse-grained components. Any optimization to recover has to be explicitly foreseen by the instruction set, and the choice of optimizations covered (actually, only two) was ad-hoc. For instance, inter-module calls (as are frequent in the presence of fine-grained modularization) are not accounted for.

Revised Design. Mozart's optimization correctly identifies specialization as a (subordinate) service. In Seam, which by design features run-time compilation, specialization is integrated with the run-time compiler, and can perform optimizations more general than what the instruction set can express. Self-modifying instructions as in Mozart become unnecessary. Currently, the decision of when to specialize is made by the offline compiler and encoded as an alternative closure creation instruction.

5.2 Persistence

Pickling consists of performing a depth-first traversal of a graph in the store, and generating a linear representation from it (a *pickle*). *Unpickling* is the process of recreating a graph from the linear representation. The components discussed in Section 5.1 are stored on files using pickling.

Mozart's Approach. The pickler performs two traversals of the graph. First, all nodes are recorded in a hash table to detect the sharing present in the graph, and it is tested whether the graph contains any nodes not allowed in pickles. If it does, an exception is raised. The second traversal performs the actual pickling. The hash table used for sharing detection hashes nodes on their address. After a garbage collection (which modifies node addresses), the table needs to be re-hashed. Graph traversal is iterative, managing its own stack of yet-to-be-visited nodes. The unpickler consists of a *builder* featuring operations for the top-down construction of data structures, and of an interpreter reading the linear representation and invoking builder methods accordingly. The builder internally maintains a stack of partially initialized nodes, which are filled up using a similar technique to S-pointers in the WAM.

Experience. Mozart provides an abstract graph traversal algorithm as a C++ class. The algorithm is instantiated by overriding 22 type-specific processing methods. Because Mozart nodes are not uniform, every node type has its own traversal routine, and these do not use a consistent policy: The children of some compound nodes are traversed left-to-right, others right-to-left. Sharing detection is left to the subclasses (because, for instance, performance is better when the two-pass pickler only actually tests for sharing in its first pass); accordingly, macros for handling sharing are expanded on average 13 times per pass. To make matters worse, there are two types of sharing detection: The first pass performs sharing detection on Oz nodes (labeling only shared nodes), the second pass performs sharing detection on manually managed C++ objects (all labeled).

For top-down construction, all picklable node types require extra constructors and accessors to make late initialization possible. Since constructors have not been designed consistently for this purpose, the builder implements a mixture of top-down and bottom-up construction to compensate. S-pointers are not sufficient to model late initialization in all cases, therefore the builder's stack can hold 40 types of task. (Some tasks are duplicated because sharing is not represented generically in pickles—instead, we have tags for labeled and unlabeled nodes).

The traversals and the builder are reused by marshaling and unmarshaling respectively, used in Mozart's distribution protocols. Their designs have not been separated thoroughly enough though: Pickling handles logic variables (they are disallowed) and futures (they cause blocking) inconsistently, and two kinds of unpicklable nodes are artificially distinguished in exceptions (irrevelant and confusing to the user). On the plus side, all the infrastructure to make pickling and

unpickling concurrent is there, because marshaling and unmarshaling needed to be made suspendable. Pickling and unpickling just do not use this infrastructure—they are always atomic.

Revised Design. SEAM actually implements pickling and unpickling as embedded virtual machines. Many of Mozart's problems are simply absent because SEAM's store has uniform node types. SEAM uses an instruction set in the pickle that constructs graphs bottom-up, which in practice degenerates less (for instance, lists do not need special optimization).

5.3 Mobile Code

Mozart's Approach. Mozart code is made mobile simply by allowing first-class procedures in pickles. Mozart therefore has to distinguish between an internal one and an external code representation, and pickling/unpickling convert between them. For instance, external code uses integer opcodes, while internally we use threaded code (a pointer to the instruction's implementation replaces the integer opcode). Mobile code requires garbage collection of code to prevent space leaks. Mozart code is allocated in the C++ heap in so-called "code areas". Closures on the Oz heap carry a reference to a code area, and code areas can reference values on the Oz heap as immediate operands. Mozart therefore has one garbage collection algorithm for each kind of heap. Code garbage collection is non-copying and uses explicit deallocation. The policy is to couple every fifth Oz heap garbage collection with a code garbage collection.

Experience. The unit of allocation is the code area. Every unpickled procedure occupies a code area. Because nested procedures are contained within the same code area, this means that often, Mozart cannot deallocate the whole code area, rendering code garbage collection useless. It does work if mobile code is not used to transfer components but just small computations.

Bidirectional links between data and code require complex handling in the pickler and unpickler. Also, the garbage collector has to be careful to remove immediate operands from the root set when deallocating a code area.

Mozart's member selection instruction makes the accessed module explicit, meaning that the whole module is reachable (and therefore pickled or transferred) instead of just those of its members that are actually used. Mozart's system components therefore often begin with a sequence of shortcut declarations of the form `A = M.a`.

Revised Design. SEAM represents code as a regular language data structure, rendering immediate arguments and code garbage collection trivial. Conversion to internal representation is rationalized by the run-time compiler, but keeps the external representation around for eventual pickling. Note that also the internal code must be aware of possibly being moved by the garbage collector. The member selection problem is rather an Oz problem—Alice fixes this by making all module accesses lazy. They can thus be hoisted to the top-level without changing the semantics.

5.4 Linguistic Reflection

Table 1: Component Size Comparison (LOC)

Component	Mozart	SEAM
Store/Data Representation	18050	3500
Pickling	11250	1650
Execution Unit	6100	3400
Assembler	1600	10

Mozart's Approach. Mozart's compiler is implemented in Oz and is available as an Oz component. Its interface allows to set a first-class environment and to compile code, represented as a string or an abstract syntax tree, relative to this environment. The compiler generates a list of Oz records, each record's label representing an instruction's opcode and the subtrees specifying the operands. A built-in assembler implemented in C++ traverses the instruction list, and fills a new code area with the corresponding bytecode, returning a nullary procedure. Application of the nullary procedure causes the code to be executed.

Experience. We pay the price of having one more code representation in the system, that of an Oz list of instructions, and thus one more place to adapt when the instruction set is modified (which has happened often). The alternative would have been for the compiler to produce a pickle instead of assembling into the heap. However, since not all data structures can be pickled, this would have restricted the values that could be contained in the compilation environment.

Revised Design. We keep the property that the compiler is available as a standard component and generates code into the heap. Since SEAM has an external code representation in the form of a high-level language data structure anyway, assembling becomes trivial in the revised design: It is reduced to creation of a closure from first-class code.

6. IMPLEMENTATION STATUS OF SEAM

We have a running prototype of SEAM which implements full Alice, which roughly can be regarded as Oz with Types. SEAM validates the consistency of the revised design from the previous sections. We find evidence that the goals of simplicity and compactness are fulfilled, by comparing the respective sizes of Mozart and SEAM components. Table 1 presents sizes measured in lines of code (LOC).

Table 2 shows the results of some standard benchmarks from [12] run on SEAM and Mozart. We report the minimum time obtained from eight runs on a Sony GRX316MP with a 1600MHz Pentium 4 processor and 512MB main memory, running Windows XP. Benchmarks cover symbolic computation involving recursion and simple integer arithmetic (*naïve reverse*, *deriv*), creation and termination of 100,000 threads (*mkthread*), and concurrent computation (*concfib*). SEAM lies within 20% of Mozart's performance for sequential computations—very promising for a prototype, if we keep in mind that a lot of tuning has been done for Mozart.

A second benchmark suite evaluates pickling and unpickling efficiency of long integer lists (*list*), large components containing mostly code and little sharing (*component*), and signature representations with many coreferences (*types*). As

Table 2: Benchmark Results (in milliseconds)

	naïve reverse	*deriv*	*mkthread*	*concfib*
Alice-on-Mozart	1341	390	451	180
Alice-on-SEAM	1652	551	250	60

Table 3: Pickling Results (in milliseconds)

	list		*component*		*types*	
	save	load	save	load	save	load
Alice-on-Mozart	220	30	1692	110	1191	60
Alice-on-SEAM	200	20	1151	70	331	30

shown in Table 3, SEAM beats Mozart on all benchmarks.

More details can be found in [4]. The current prototype implementation is available on request.

7. CONCLUSIONS

This paper has examined the architecture and implementation of open programming systems. We identified a core virtual machine encapsulating platform-specific aspects, and a clearly separated set of services. Services can either be implemented stand-alone, or integrated into the core virtual machine. We examined the realization of core and services in Mozart and found that Mozart's services are mostly stand-alone. Based on our experience with Mozart's approach, we proposed revised designs in the form of a virtual machine called SEAM that clearly favors integrated services. Comparisons indicate that the SEAM design is superior to that of Mozart with respect to both implementation effort and performance.

8. ACKNOWLEDGMENTS

We thank all Mozart implementors for their good work. Our criticism may sound harsh, nonetheless Mozart is an efficient, stable system used in real projects. The covered material greatly benefited from discussions with Christian Schulte, who also provided significant initial input for the abstract machine's factorization. Ulrike Becker-Kornstaedt's feedback greatly improved the readability of this paper. The SEAM pickler includes improvements by Guido Tack.

9. REFERENCES

[1] H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. The MIT Press, 1991.

[2] The Alice programming system, version 0.9.1. Web Site at the Programming Systems Lab, Universität des Saarlandes, 2003. `http://www.ps.uni-sb.de/alice/`.

[3] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifiers (URI): Generic syntax. Request for Comments 2396, Network Working Group, 1998.

[4] T. Brunklaus and L. Kornstaeadt. A virtual machine for multi-language execution. Technical Report, Nov. 2002. `http://www.ps.uni-sb.de/Papers/abstracts/multivm.html`.

[5] D. Duchier, L. Kornstaedt, C. Schulte, and G. Smolka. A higher-order module discipline with separate compilation, dynamic linking, and pickling. Technical report, Programming Systems Lab, DFKI and Universität des Saarlandes, 1998. `http://www.ps.uni-sb.de/Papers/abstracts/modules-98.html`.

[6] M. Henz and L. Kornstaedt. *The Oz Notation*. The Mozart Consortium, Feb. 2000. `http://www.mozart-oz.org/documentation/notation/`.

[7] L. Kornstaedt. Alice in the land of Oz—an interoperability-based implementation of a functional language on top of a relational language. In *Electronic Notes in Theoretical Computer Science*, volume 59.1, pages 18–33. Elsevier Science Publishers, 2001.

[8] X. Leroy. The ZINC experiment: An economical implementation of the ML language. Technical Report RT-0117, INRIA, Feb. 1990.

[9] M. Mehl. *The Oz Virtual Machine—Records, Transients, and Deep Guards*. Doctoral dissertation, Technische Fakultät der Universität des Saarlandes, 1999.

[10] M. Mehl, R. Scheidhauer, and C. Schulte. An abstract machine for Oz. In *Proceedings of PLILP'95, LNCS*, Utrecht, The Netherlands, 1995. Springer-Verlag.

[11] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[12] R. Scheidhauer. *Design, Implementierung und Evaluierung einer virtuellen Maschine für Oz*. Doctoral dissertation, Fachbereich Informatik, Universität des Saarlandes, Dec. 1998.

[13] D. Stemple, L. Fegaras, R. B. Stanton, T. Sheard, P. Philbrow, R. L. Cooper, M. P. Atkinson, R. Morrison, G. N. C. Kirby, R. C. H. Connor, and S. Alagic. Type-safe linguistic reflection: A generator technology. In *Fully Integrated Data Environments*, pages 158–188. Springer, 1999.

[14] The Mozart Consortium. Mozart Oz 1.2.5. Web Site, Feb. 2003. `http://www.mozart-oz.org/`.

[15] P. R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, Saint-Malo, France, Sept. 1992. Springer-Verlag.