Electronic Notes in Theoretical Computer Science 59 No. 1 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume59.html 16 pages

Alice in the Land of Oz

An Interoperability-based Implementation of a Functional Language on Top of a Relational Language

Leif Kornstaedt

kornstae@ps.uni-sb.de
Programming Systems Lab
Universität des Saarlandes
Saarbrücken, Germany

Abstract

This paper reports practical experience in implementing Alice, an extension of Standard ML, on top of an existing implementation of Oz. This approach yields a high-quality implementation with little effort. The combination is an advanced programming system for both Oz and Alice, which offers more than either language on its own.

1 Introduction

The language Oz and its implementation Mozart [19] constitute an advanced programming system, targeted at concurrent constraint programming and open distributed computing. It is used in areas as diverse as combinatorial problem solving and scheduling, natural language processing (parsing, semantics representation, and inference), multi-agent systems, and collaborative tools.

However, Oz has the reputation of being an exotic language, preventing it from being more widely used. In part, it owes this reputation to its syntax. To make the achievements of Oz available to a larger community, we are integrating its key features into a wide-spread functional language: In a first step, we have extended Standard ML by data-flow-driven concurrency and components with lazy dynamic linking. We call this language *Alice*.

This paper describes an implementation of Alice on top of Oz. This approach maximizes reuse of the existing implementation technology developed for Oz. By making Alice-Oz-interoperability a major goal, we immediately obtain the following benefits:

- It becomes straightforward to implement Alice's primitives in Oz.

- Since Alice can import components implemented in Oz, all features of Oz' runtime system are immediately available to Alice, such as constraints [15], computation spaces for speculative computation [18], persistence, and distribution [7].
- Since Oz can import components implemented in Alice, users can evaluate Alice while keeping their existing Oz codebase, implementing some component or other in Alice, before maybe migrating entire projects.
- Since Alice is backward-compatible with SML, existing SML libraries can be made directly accessible to Oz programmers, by compiling them with the Alice compiler.

In short, the resulting system offers benefits to both the existing Oz community and to existing SML programmers.

This paper is structured as follows: Section 2 presents the languages Oz and Alice. Section 3 introduces some terminology and motivates our approach to interoperability, before Section 4 gives a detailed account of how we put this into practice. The implementation is described in Section 5. We discuss the pros and cons of our approach in Section 6 and conclude the paper with an outlook on future work in Section 7. Appendix A provides example applications of the interoperability interface.

2 The Cast: Oz and Alice

This section introduces the protagonists of our interoperability story: the language Oz, the programming system Mozart which implements Oz, and the language Alice.

2.1 The Language Oz

The language Oz is defined in terms of a small sub-language, Core Oz, and a number of syntactic extensions which are defined by translation to Core Oz [9]. Like Oz, Core Oz is a dynamically typed, concurrent, non-backtracking relational language. Its syntax is summarized in Fig. 1, using the symbol i for integers, a for atoms, and x, y, and z for variables.[1]

Core Oz supports the following data types:

Transients are placeholders for unknown values. Transients provide for data-flow synchronization: Threads block when they need the values of transients for computations. Binding a transient replaces the transient by its value, resuming all threads suspending on it.

One kind of transient is the *logic variable*, similar to Prolog's. A read-only view of a logic variable is a *future*. A *by-need* future carries a nullary procedure which is applied when the future's value is required, with the

[1] We omit the constraint extensions for the purpose of this paper.

Fig. 1. Syntax of Core Oz.

Statements			
s	::=	$s_1\ s_2$	sequential composition
	|	**local** x **in** s **end**	local declaration
	|	$x = !!y$	future extraction
	|	$x = y$	unification
	|	$x = t$	tell value
	|	$x = y.z$	record selection
	|	{NewName x}	name creation
	|	**case** x **of** t **then** s_1 **else** s_2 **end**	conditional
	|	**proc** $\{x\ y_1 \ldots y_n\}$ s **end**	abstraction ($n \geq 0$)
	|	$\{x\ y_1 \ldots y_n\}$	application ($n \geq 0$)
	|	**thread** s **end**	thread creation
	|	{NewCell $x\ y$}	cell creation
	|	{Exchange $x\ y\ z$}	cell exchange
	|	{Raise x}	exception raising
	|	**try** s_1 **catch** x **then** s_2 **end**	exception handling
Terms			
t	::=	i	integers
	|	$l(f_1 : x_1 \ldots f_n : x_n)$	records
l	::=	$x \mid a \mid$ **true** $\mid$ **false** $\mid$ **unit**	literals
f	::=	$l \mid i$	features

effect of replacing the future by a new value computed by the procedure. By-need futures are used to model lazy computations. [2]

Atoms are internalized strings, like Lisp symbols.

Names represent structureless identities. An infinite number of new names can be generated. The boolean values **true** and **false** are predefined names, as is the value **unit** used for synchronization. Atoms and names are collectively referred to as *literals*.

Records are structured values consisting of a *label* l and any number of *subtrees* x_i, accessible through the unique *features* f_i of the record. The label l serves to model algebraic data types. If y is a record of the form $l(f_1 : x_1 \ldots f_n : x_n)$ and $z = f_i$, then the record selection statement $x = y.z$ unifies x with x_i. Records with integer features starting from 1 are called *tuples*. Lists are defined as either the atom 'nil' or a binary tuple with label '|', whose second element is a list.

Procedures take any number of arguments. All arguments are input arguments; output arguments are modelled by supplying a logic variable as input argument and binding it in the procedure's body. Procedures are first-class data structures.

[2] By-need futures are created by a primitive operation not depicted in the syntax.

Fig. 2. (a) A sample Oz component 'Test'. (b) A sample Oz application using 'Test'.

(a)

```
functor
export fak: Fak
define
   fun {Fak N}
      if N == 0 then 1
      else N * {Fak N - 1}
      end
   end
end
```

(b)

```
functor
import
   Test at 'Test'
   System at 'x-oz://system/System'
   Application at 'x-oz://system/Application'
define
   {System.show {Test.fak 7}}
   {Application.exit 0}
end
```

Cells are the only stateful entities in Core Oz. A cell holds a reference to an Oz value and supports an atomic exchange operation.

While Core Oz is a relational language, meaning that '=' is unification, not assignment, full Oz supports the functional and object-oriented programming paradigms with dedicated syntax. This syntax is translated to Core Oz: Functions, for instance, are procedures with an additional argument for output (unification allows for bidirectional data flow). Due to the translation, functions can be called as procedures and vice-versa.

Oz uses records as modules, without any dedicated syntax: Modules are just a convention for structuring applications. Oz applications are deployed as a number of components. A component is defined using the **functor**[3] construct (see Fig. 2(a) for an example). A component consists of an import declaration, specifying the components to link to as well as their locations, in the form of URLs; an export declaration, naming the values to export (as a record); and a sequence of declarations and statements to execute in order to build the export module. Components are translated to Core Oz data structures; the exact translation is beyond the scope of this paper.

2.2 The Programming System Mozart

The Mozart System is an implementation of the language Oz. It is based on a virtual machine, which in its original conception owes much to the Warren Abstract Machine [1] designed for Prolog. The Mozart Virtual Machine has specifically been designed for Oz [11,10], and as such provides most of the features required by Oz directly: The garbage-collected memory store is able to represent all Oz data types directly, in particular including logic variables, futures, and constraints. Data-flow synchronization is implicitly taken care of by the instructions. The runtime knows how to make data structures and code persistent, and how to marshal them to other sites in a distributed

[3] Oz calls components *functors*; to avoid confusion with SML functors, we will only be using the term *component* in this paper.

environment.

The runtime is started up with the URL of the application component to execute on the command line. The machine first loads an initialization component, which may only import built-in components. The initialization component instantiates a *component manager*, which is responsible for lazy loading and linking of components. When a component from a specific URL is first imported, a by-need future is created that will eventually cause the loading and execution of the corresponding component. Executing a component causes it to compute and return a module. The mapping from URLs to by-need futures resp. modules is maintained in a table internal to the component manager. Next, the initialization component dynamically imports and requests the application component. The actual application consists of the side-effects caused by executing the application component (see Fig. 2(b) for an example).

Mozart components are platform-independent: Currently, Mozart is available for a wide range of platforms, including Windows, Linux, and MacOS X.

Oz' component system and component managers are described in [6].

2.3 The Languages Standard ML and Alice

Standard ML (*SML*) is a strict functional programming language. As such it is based on the notion of evaluating expressions instead of executing sequences of statements, uses call-by-value for function applications, and supports functions as first-class values. SML is specifically designed for safety: Data structures are classified into immutable and stateful data structures. Error handling is facilitated by an exception handling mechanism. Run-time type errors are prevented by its static polymorphic type system.

SML is factored into the core language and the module language. The core language supports type inference, allowing concise programs with minimal notational overhead for type annotations. SML has an advanced module system with type abstraction and parameterized modules. Both the static and dynamic semantics of SML are formally defined [14].

One peculiarity of SML is that all constructors and functions are unary, as opposed to taking an arbitrary number of arguments. The common idiom is to model multiple arguments by tupling, although a concise syntax for defining curried functions is available.

Alice [2] is an extension of Standard ML targeted specifically at open programming. We chose SML as the base language for Alice for the following reasons:

- Due to its clean design, it is relatively easy to devise extensions that do not break the soundness of the language. Its formal definition will eventually allow us to model the extensions formally and prove their soundness.
- Standard ML is well-known and has a large user community. Programmers fluent in SML will be able to learn Alice quickly.

Fig. 3. (a) A sample Alice component 'Test'. (b) A sample Alice application using 'Test'.

(a)

```
structure Test = struct
  fun fak 0 = 1
    | fak n = n * fak (n - 1)
end
```

(b)

```
import structure Test from "Test"
import structure TextIO
  from "x-alice:/lib/system/TextIO"
import structure OS
  from "x-alice:/lib/system/OS"
val _ = TextIO.print (Test.fak 7)
val _ = OS.Process.exit 0
```

- A lot of teaching material is available for SML. This material can serve as a starting point to teach resp. learn Alice.

To compensate for the areas in which we find SML lacking, we integrated extensions inspired by Oz:

- Concurrency comes naturally to Open Programming (modelling servers; responding to input in graphical user interfaces). Alice extends SML by data-flow driven concurrency. Logic variables (which are better termed *holes* in the context of functional languages) are powerful but error-prone: For increased safety, Alice only provides futures and *promises*, which are non-transparent logic variables.
- Dynamically linked components are mandatory for a truly open programming system. SML's language definition completely omits even the practical issues of how programs are to be organized into source files and how to support separate compilation. The existing SML implementations address these issues each in their own way, for instance Standard ML of New Jersey [3] has its *Compilation Manager* [5]. None of these approaches provide dynamic linking in a satisfactory way. Alice extends Standard ML by platform-independent components à la Oz. Fig. 3 shows Alice versions of the Oz components depicted in Fig. 2. When the compiler encounters an import announcement, it loads the referenced component to obtain its export signature. At run-time, the expected signature is checked against the actual export signature upon dynamic linking, raising an exception upon mismatch.

3 Interoperability: Terminology and Approach

The design of an interoperability interface has to take two levels into account: the language level and the implementation level. At the language level, we can distinguish between transparent interoperability (programmers do not see whether the procedures or data they use are foreign) and non-transparent interoperability (programmers will need to use different operations when dealing with native vs. foreign procedures and data). At the implementation level, we

find direct interoperability (both languages operate on the same representations of procedures and data) and marshaling-based interoperability (each language uses its own representations for procedures and data, which have to be converted at the interoperation boundaries).

In practice we find all possible combinations:

Transparent/direct is how C# [12] and Visual Basic 7 interoperate in Microsoft's .NET Common Language Runtime [13].

Transparent/marshaling-based is used by the .NET Common Language Runtime to access platform-specific dynamically linked libraries (P/Invoke).

Non-transparent/direct interoperation takes place when using Java objects from MLj [4].

Non-transparent/marshaling-based is found in Mozart's C/C++ foreign function interface.

In the following, we will consider the case where one programming language (termed the *guest language*) is to be implemented on top of an existing implementation of another language (the *host language*), sharing the host's runtime system, with bidirectional interoperability. Let's consider the advantages and drawbacks of the approaches outlined above.

Transparent vs. non-transparent. When interoperating non-transparently, special language extensions or libraries are required for dealing with foreign procedures and data. Transparent interoperability makes the strong presupposition of a mapping (ideally bijective, if bidirectional interoperation is intended) which maps the natural way to express something in the guest language to the natural way to express that something in the host language.

Marshaling vs. direct. Marshaling-based interoperability allows each language to use the most efficient representation for its data and procedures, but causes performance loss at the interoperation boundaries due to conversion and copying of data and adaption of calling conventions. Direct interoperability does not impose this overhead, but again is much harder to design: When there is no equivalent for some data structure, then it has to be *modelled* using the data structures of the host language, that is, expressed indirectly.

3.1 Our Approach

In our case, the host language is Oz with its implementation Mozart, and the guest language is Alice. The following considerations determined the design of the interoperation interface:

- Since both Oz and Alice support higher-order data structures, it is most natural to aim for bidirectional interoperability.

- Futures must be anticipated virtually anywhere in Alice data structures.[4] Thus, representation analysis techniques for SML are not applicable. This excludes elaborate mappings, only leaving us with a very direct approach—with the advantage of becoming compositional: This means that the direct mapping from Alice to Oz will immediately imply how to call from Oz to Alice.

Note that if Alice was the host programming system and Oz the guest language, the situation would be different: Directly mapping a dynamically typed language to a statically typed language is impossible in general—a modelling approach would be mandatory.

4 Mapping Alice to Oz

We will now present the translation of the Alice core, module, and component languages to Oz.

4.1 Core Language

Primitive data types. For many Alice core data types, there are Oz equivalents: Alice integers and characters are mapped onto Oz (infinite-precision) integers. Alice reals become Oz floats. Alice strings are represented as Oz byte strings. Oz has been extended by a basic data type 'word' (with bitwise operations), provided in C++ using Mozart's foreign function interface.

Records. Tuples and records in Alice are very similar to one another: Tuples are just syntactic convenience for records whose labels only consist of digits, which when interpreted as integers are consecutive and start at one. This maps nicely to Oz: Oz record features can be either integers or literals (atoms or names). Similar to Alice, when a record's features are consecutive integers starting from one, the record is a tuple. Our mapping preserves this similarity by mapping labels consisting only of digits to integers and all other labels to the corresponding atoms. In contrast to Alice, Oz records are equipped with an additional tag, called the *label*: We chose the label '#', which is used by the syntactic sugar for mixfix tuples in Oz. Thus, the Alice tuple '$(1, 4, 7)$' has the same representation as the corresponding Oz tuple '1#4#7'. The single exception to this is for the record that has no fields (that is, the Alice tuple '()' resp. record '{}'): The common idiom in Alice is to use this value as argument of nullary functions, or as result of functions that only perform a side-effect and do not return any interesting value. It is the only value of the so-called unit type. Since we want to map idioms, it seems most natural to represent this as the Oz name **unit** (which incidentally also is an Oz record without fields).

[4] Note that we cannot do whole-program analysis due to component-based programming.

Constructors. Alice constructors are in general mapped to Oz atoms corresponding to the constructor's identifier, except for exception constructors, which have to be mapped to Oz names because they are generative. Some constructors have a special translation: For instance, the Alice constructors true and false are mapped to the Oz names **true** and **false**, to make boolean expressions interoperate.

Constructed Values. Because conceptually all Alice constructors are unary, we have introduced the notion of *syntactic arity*. When a constructor is syntactically declared to take a record (or a tuple) as argument, we consider this constructor to be n-ary, n being the number of fields in the record. We map constructed values of n-ary constructors to Oz records, using the mapped constructor as label instead of '#'. It follows that algebraic data types are mapped to exactly their usual representation in Oz. Because we specially translate the constructor '::' to the atom '|', for instance, the Alice list '1::2::nil' is mapped to the Oz list '1|2|nil'. For syntactic arity to work, we needed to introduce a similar ML restriction as OCaml [16] does: Signature matching is not allowed to change the syntactic arity of constructors.

State. One kind of constructor has a different translation, namely the 'ref' constructor which introduces state. This is translated to Oz cells, which support equivalent operations. In Alice, it can be statically decided whether a constructor is 'ref'.

Functions. Like constructors, Alice functions are unary. We also introduce the concept of a syntactic arity for functions: Function definitions use pattern matching on their arguments. If any pattern explicitly deconstructs the argument as a tuple, and no pattern binds the whole tuple to an identifier, then the function has the tuple's arity.[5] It is not possible to preserve syntactic arity as for constructors: This is why the conversion of calling convention (unary vs. n-ary) has to be performed dynamically. n-ary functions are mapped to $n+1$-ary Oz procedures: The additional argument is used for returning the result.

Note that due to the special handling of the empty tuple (unit), we have a certain irregularity here: A nullary Alice function is translated to a binary Oz procedure expecting the *unit* value as first argument, instead of a unary Oz procedure taking only one output argument for the function's result.

Futures. Promises, futures and by-needs are implemented as builtins on top of the functionality of Oz and require no special care.

In terms of representation analysis, arity raising of n-ary constructors and functions are the only optimizations performed. Note that in the present case, our primary motivation for this is interoperability, not efficiency.

The type language part of the core language is subject to type erasure,

[5] Note that we cannot use the function's type to determine the arity, because this could change the semantics when the argument is a by-need future: It would be requested sooner than necessary.

meaning that its operational semantics are defined for programs stripped of all type annotations [14, Section 6.1].

4.2 Module Language

Oz modules are represented as records. To interoperate smoothly, Alice structures are thus translated to records with atom features. Values are stored under their identifier. Constructors that take arguments are represented as two fields: The constructor's identifier designates the constructor function, expecting the constructor's argument and returning the constructed value. This reflects the fact that a constructor used as expression is the constructor function. Its identifier prefixed with a quote (which is not a valid value identifier in Alice) designates the constructor itself and is used to translate pattern matching.

Structures reside in their own namespace, which is distinct from the namespace for values and constructors. To avoid capturing, their identifier is prefixed with a dollar sign, again yielding an invalid Alice identifier.

Functors (functions computing modules from modules) are unary; multiple arguments are passed as a structure containing structures. Functors are translated into binary procedures taking a translated structure as input and returning a new structure.

Like types, signatures are subject to type erasure.

4.3 Components

An Alice component can be considered as a functor, with the extension that its argument and result may contain nested signatures and functors.[6] The component functor explicitly identifies its argument modules by URLs to instruct the component manager on how it is to be applied.

Because components are inspired by Oz, Alice components and Oz components are so similar to one another that translation becomes trivial, with some additional considerations about component signatures. Alice components extend Oz components by signatures for imports and export. To import an Alice component into Oz, no mechanism besides ignoring these signatures is necessary.

Alice, being statically typed, can import a dynamically-typed Oz component only if its export signature is known. When the Alice compiler encounters an import directive, it loads the export signature from the referenced compiled component. In the case of an Oz component, such an export signature will be absent. Instead, the programmer can provide a signature in a file with the same base name as the referenced component and the special extension '.asig'. At run-time, this file is not needed.

[6] In contrast to SML, Alice supports higher-order functors.

We extended the Oz component manager to be aware of component signatures: When an Alice component imports another Alice component, it will at link-time verify that their signatures match, as required by the Alice language definition. In all other cases, signatures are ignored.

Under this translation, the example components in Fig. 2 and 3 become interchangeable. Appendix A gives more involved examples of interoperation.

4.4 Libraries

Alice supports the Standard ML Basis Library [17] for compatibility with SML.[7] Extensions adhere to the style of the Standard Basis. This makes them feel more natural to SML programmers.

The extensions include in particular libraries for constraint programming, providing finite domain constraints, finite set constraints, and encapsulated search. Since these were already available in Oz through libraries (Oz' constraint syntax is only syntactic sugar [9]), it was only a matter of recasting these in a statically typed formulation [2].

5 Implementation

Both the implementation of the Alice compiler and the Alice runtime make reuse of existing Mozart technology. This is detailed below.

5.1 Mozart Extensions for Alice

The Alice runtime is the Mozart System, with minimal extensions. These are now part of the standard Mozart distribution.

Tupling and Detupling. For efficiency, two instructions have been added to the Mozart Virtual Machine for run-time calling convention conversions. The first applies a procedure with $n > 1$ arguments. A run-time check tests whether the arity of the procedure and the call match, in which case the call proceeds normally. Else, the procedure must be binary (that is, a unary function): A tuple is constructed from the first $n - 1$ arguments and passed as a single argument. The nth argument becomes the second argument and is used to return the function's result.

The second instruction works the other way round: If a procedure expecting more than two arguments is called with only two arguments, the first must be a tuple which is deconstructed to obtain the actual $n - 1$ input arguments. The second argument becomes the nth argument and is used for output.

In principle, calling convention conversions could have been emulated in Oz, which supports inspection of a procedure's arity as well as an 'apply'

[7] Only parts of the library have been implemented yet, due to a lack of development resources.

procedure taking a procedure and a list of arguments. However, this would have been significantly more expensive.

Typed Components. Oz components already carry type fields for the export signature of components and the expected import signatures of referenced components. This type representation has been extended to represent Alice signatures, which are much more expressive than the simple Oz types intended for type falsification. [8]

Link-time Type-checking. The Mozart component manager and static linker are now parameterized over a type-checking procedure, to accomodate for Alice link-time type checking.

5.2 The Alice Compiler

The Alice compiler generates Mozart components from Alice sources. To reuse the bytecode generator from Mozart's Oz compiler, we used an intermediate representation of the latter as target for the Alice compiler. This intermediate representation basically corresponds to alpha-renamed Core Oz programs in which optimizations have been made explicit. This allowed us to directly plug in the existing backend, which performs liveness analysis, instruction selection, and register allocation.

Thus, the Alice compiler's frontend is implemented in Alice, while its backend is implemented in Oz. The bootstrapped compiler integrates these two using the Oz-Alice-interoperability described above. However, for bootstrapping the compiler, this option is not available. We use Standard ML of New Jersey [3] to compile the frontend, which is implemented using only the SML subset of Alice. A dummy backend writes the abstract syntax tree to an Oz pickle (a persistent representation of a data structure). An independent Mozart process, in turn, reads this pickle and executes the real backend. The bootstrapped version of Alice does not require Standard ML of New Jersey to run: it is a pure Mozart application.

6 Discussion

The motivation behind our work was to provide an implementation of the programming language Alice, but it can also be argued that with only minimal extensions to Standard ML's language definition and the Mozart runtime, we made SML interoperate smoothly with Oz.

Our approach has some limitations however regarding the use of the guest language Alice from the host language Oz:

- Alice uses the type-based mechanism of signature abstraction to implement abstract data types. Whether some data is an instance of a concrete or an

[8] Oz being a dynamically typed language, type verification is impossible. Type falsification allows for early error diagnosis.

abstract type does not influence its representation. As a consequence, Oz programs, being dynamically typed, will be able to break these abstraction barriers: Alice abstract types are concrete as seen from Oz.

Curing this problem is not trivial due to the fundamentally different approach to data abstraction in both languages. Oz provides a special data type called a *chunk*: A chunk is like a record which does not permit first-class inspection of its features. Thus, data can be made inaccessible by storing it in a chunk under a name hidden by lexical scoping. [9]

- The Oz compiler does not implement dynamic calling convention conversion, which would actually alter the defined semantics of Oz. The consequence is that Oz programs have to use the exact arity chosen by the Alice compiler when calling Alice functions. The arity can change as the implementation of an Alice function changes, without even changing its signature. Oz programmers can account for this situation by performing calling convention conversion at the language level, by explicitly testing the procedure's arity.

  A solution to this problem would consist of extending signature coercion by the operational effect of always arity raising the functions contained in the argument structure. Then, representations would always be predictable at module interfaces.
- Alice and Oz currently use different exceptions for signaling similar error conditions. In particular, it is impossible to catch the exceptions raised by Oz primitives in Alice. We are thinking about adding an Alice exception constructor for representing reflected Oz exceptions.

None of these limitations has ever been a serious problem in practice, although the interoperation interface has already been used extensively.

In terms of sheer performance, the present implementation of Alice nearly attains that of Oz, but of course cannot compete with high-performance ML implementations. Nevertheless, we believe that in the application domains in which Oz is used heavily, Alice has a definite advantage over SML.

7 Outlook

We intend to formalize the Alice extensions in the style of the Definition of Standard ML. Work has started on the formalization of futures and laziness.

Already we have several projects running that build on the work in this paper to exploit the potential of the combination of Oz and Alice. Oz libraries are being reshaped as Alice libraries, to give Alice programmers access to Oz' constraint systems and constraint solving facilities. Alice already benefits from the development tools developed for Oz: A customized version of the Oz Inspector, intended for interactive value inspection, uses Alice syntax. The Oz Explorer, used for visualizing search trees, is available for Alice

[9] Typically, Oz programs use classes instead, which build on this same mechanism [8].

constraint problems. The Oz Profiler collects statistics about Alice functions. The usefulness of the Oz debugger for Alice programs is being investigated. In the case of a new GTk-based graphical user interface library, development for Alice has even overtaken that for Oz.

The ongoing work of extending Alice by run-time type representations has enabled us to enrich the language with dynamically typed packages and type-safe pickling. Also, the first type-safe distributed applications are now running.

Acknowledgement

Many thanks go to Ulrike Becker-Kornstaedt, Thorsten Brunklaus, Tobias Müller, and Christian Schulte for their comments on a previous version of this paper. For the numerous discussions regarding the details of the translation scheme, thanks go to Andreas Rossberg, implementor of the Alice compiler frontend. Finally, I'd like to thank the anonymous referees for their comments.

References

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

[2] *The Alice Project*, Web Site at the Programming Systems Lab, Universität des Saarlandes (2001).
URL `http://www.ps.uni-sb.de/alice/`

[3] Appel, A. and D. MacQueen, *Standard ML of New Jersey*, 3rd International Symposium on Programming Language Implementation and Logic Programming (1991), pp. 1–13.

[4] Benton, N. and A. Kennedy, *Interlanguage working without tears: Blending SML with Java*, in: *Proceedings of the fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, Paris, France, 1999, pp. 126–137.

[5] Blume, M. and A. Appel, *Hierarchical modularity*, ACM Transactions on Programming Languages and Systems **21** (1999), pp. 813–847.

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

[7] Haridi, S., *Efficient logic variables for distribution*, ACM Transactions on Programming Languages and Systems **21** (1999), pp. 569–626.

[8] Henz, M., "Objects in Oz," Ph.D. thesis, Universität des Saarlandes, Fachbereich Informatik, Saarbrücken, Germany (1997).

[9] Henz, M. and L. Kornstaedt, "The Oz Notation," The Mozart Consortium (2000).
URL `http://www.mozart-oz.org/documentation/notation/`

[10] Mehl, M., "The Oz Virtual Machine—Records, Transients, and Deep Guards," Ph.D. thesis, Technische Fakultät der Universität des Saarlandes (1999).

[11] Mehl, M., R. Scheidhauer and C. Schulte, *An Abstract Machine for Oz*, in: *Proceedings of PLILP'95, LNCS, Springer-Verlag*, Utrecht, The Netherlands, 1995.

[12] Microsoft, Hewlett-Packard and Intel Corporation, *C# language specification (ECMA TC39/TG2)*, Proposed draft standard (2000).
URL `http://msdn.microsoft.com/net/ecma/`

[13] Microsoft, Hewlett-Packard and Intel Corporation, *Common Language Infrastructure (ECMA TC39/TG3)*, Proposed draft standard (2000).
URL `http://msdn.microsoft.com/net/ecma/`

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

[15] Müller, T., "Constraint Propagation in Mozart," Doctoral dissertation, Universität des Saarlandes, Saarbrücken, Germany (2001), in preparation.

[16] Projet Cristal, *The Caml language*, Web Site at INRIA, Paris, France.
URL `http://caml.inria.fr/`

[17] Reppy, J., "The Standard ML Basis Library," Bell Labs, Lucent Technologies (1997).
URL `http://cm.bell-labs.com/cm/cs/what/smlnj/doc/basis/`

[18] Schulte, C., "Programming Constraint Services," Doctoral dissertation, Universität des Saarlandes, Saarbrücken, Germany (2000).

[19] The Mozart Consortium, *Mozart Oz 1.2.0*, Web Site (2001).
URL `http://www.mozart-oz.org/`

A Examples

This section gives example uses of the interoperability interfaces. These are taken from the implementation of parts of the Standard ML Basis Library in Oz.

Figure A.1 shows an implementation 'Vector.tabulate'. Alice vectors are represented as Oz tuples with the label '#[]' (just to make them distinguishable from Alice tuples). It implements the Alice function by means of an Oz procedure and illustrates that the argument function can be called from Oz,

Fig. A.1. An implementation of 'Vector.tabulate' in Oz.

```
val tabulate: int × (int → 'a) → 'a vector
```

```
proc {VectorTabulate N F V}
   try
      V = {Tuple.make '#[]' N}
   catch _ then
      {Exception.raiseError alice(GeneralSize)}
   end
   {Record.forAllInd V fun {$ I} {F (I − 1)} end}
end
```

Fig. A.2. An Oz Implementation of the CommandLine module from the Standard Basis Library.

```
structure CommandLine:
   sig
      val name: unit → string
      val arguments: unit → string list
   end
```

```
functor
import
   Property(get)
   Application(getArgs)
export
   'CommandLine$': CommandLine
define
   CommandLine =
   '#(
      'name':
         fun {$ unit}
            {ByteString.make {Property.get 'root.url'}}
         end
      'arguments':
         fun {$ unit}
            {Map {Application.getArgs plain} ByteString.make}
         end
   )
end
```

and how to raise Alice exceptions. (The variable 'GeneralSize' is assumed to be bound to the exception constructor available as 'General.Size'.)

Figure A.2 implements the 'CommandLine' structure. The shown component exports a structure, makes use of the fact that Oz lists are Alice lists, and ensures that all strings (which can have various representations in Oz) are represented uniformly as byte strings before passing them to Alice.