Project MI 6:

NEP:
Statically Typed Programming Environment for Concurrent Constraints

2.1 Overview

Funding period:	01. 01. 2002 – 31. 12. 2004
Period covered by report:	01. 02. 2001 – 31. 01. 2004
Projekt director:	Prof. Dr. Gert Smolka
Work address:	Fachrichtung Informatik Universität des Saarlandes Postfach 15 11 50 66041 Saarbrücken
Phone number:	(0681) 302 – 5311
Fax number:	(0681) 302 – 5341
Email address:	smolka@ps.uni-sb.de
Researchers: *(UF: University funding)*	Thorsten Brunklaus, Dipl.-Inform. (temporarily UF) Leif Kornstaedt, Dipl.-Inform. (until 2002/12/31) Didier Le Botlan, DEA (UF, since 2004/01/01) Andreas Rossberg, Dipl.-Inform. Christian Schulte, Dr. (UF, until 2002/03/31) Gábor Szokoli, MSc (UF, since 2003/10/15) Guido Tack, Dipl.-Inform. (UF, since 2003/06/01)

The goal of the project is developing Alice into an application-strength programming system. Alice makes available functionality of the Mozart Programming System (futures, constraints, concurrency, distribution, persistence) in simplified and improved form. The most significant enhancement is the integration of that functionality into a statically typed context. This provides for the automatic verification of the consistent use of interfaces and abstractions during program development. Such consistency checks reduce the cost of developing and extending software. Alice has been designed as an extension of the functional programming language Standard ML.

The current phase of the project has a largely practical orientation. The previous phase has layed the necessary foundations for the design and implementation of Alice and delivered a prototype. The main goal of the current phase is developing an application-strength imple-

mentation of the system. A first version has been released in December 2002. During the course of development, the language design has been refined and aspects of dynamic typing and concurrency have been investigated theoretically. A novel virtual machine architecture has been developed. The constraint functionality has been completely redesigned into a more principled and more efficient library framework.

2.2 State of the Art at the Beginning of the Funding Period and Original Research Issues

The starting point for NEP was the concurrent programming model OPM (Smolka, 1995b, 1995a) and its practical implementation in form of the programming language Oz and the programming system Mozart (Mozart Consortium, 2003). OPM combines a concurrent model of computation with high-level linguistic primitives for the construction of inference components using the constraint programming paradigm (Schulte, 2002; Van Hentenryck, Saraswat, et al., 1997). Since 1996, the SFB contributes significantly to the development of Oz and Mozart. Constraint techniques are particularly suited for the syntactical and semantical processing of natural language, which makes them interesting for the projects CHORUS (MI 4) and NEGRA (EM 6). A second aim of Mozart is providing high-level linguistic support for persistence and distributed programming in the context of the Internet. Here, programming-in-the-large techniques play an important role. This side of Mozart is interesting for OMEGA (MI 4). As partners within the Mozart Consortium, two other research groups (Louvain and SICS) participate in the development of Mozart.

Alice is an attempt to provide Mozart's strengths on top of a statically typed functional language. As its basis we chose Standard ML (SML), for its prominent position in programming language research and teaching. It is a functional language with an expressive type and module system. Since significant parts of the Mozart functionality have never been incorporated into a typeful framework, Alice raised a number of interesting research issues.

By the end of the previous period a preliminary design and a prototype of Alice had been developed that already provided some essential elements of Mozart's functionality. The focus of the current period is to complete the design of Alice and turn it into a practical implementation that can be used by other projects in the SFB. The proposal identified two main questions:

1. How can constraint programming and Internet programming be modelled elegantly in a typed context? Which additional typing concepts are necessary?

2. How can the implementation of a programming system for Alice be decomposed into simple, generic, reusable and efficient components? What are the prinicple concepts for a platform-independent virtual machine?

These primary questions have been decomposed into six working packages, which will be addressed in Section 2.4.

2.3 Methods Applied

Investigation of programming language semantics for the Alice design relies on programming language theory, in particular type theory and operational semantics. Constraint technology is based on computational logic, operations research, artificial intelligence and algorithms. The implementation of Alice requires methods from software engineering, compiler construction, and operating systems.

2.4 Results and their Implications

The main goals formulated in the NEP project proposal have been achieved.

Type-safe open programming. Importing dynamic objects at runtime requires a form of dynamic typing to verify their validity. Dynamic typing has been incorporated into the statically typed context of Alice by providing a special type of values carrying modules with dynamic type information. These so-called *packages* generalise previous approaches known from literature in a flexible and convenient way. Alice is the first statically typed programming system that provides a comparable feature.

Complementing a static type system with forms of dynamic typing requires special care: it should not compromise safety and abstraction guarantees provided by the static type system. We have developed a new formalisation of type abstraction (encapsulation based on types) that is fully compatible with dynamic typing and forms the semantic basis for modules in Alice.

Apart from the type system, the concurrent semantics of Alice have been reformulated as a lambda calculus with futures. The practical suitability of the Alice language design has been verified in several case studies. Moreover, most parts of the Alice system have been implemented within the language itself.

Implementation. Two implementations of Alice have been developed: the current release version is based on the Mozart system and targets interoperability with existing Oz software. A second, improved implementation addresses the issues identified in the proposal. It is based on a new modular virtual machine architecture called SEAM (*simple efficient abstract machine*) that provides a clearly defined set of services, including memory management, task management, and portable object input/output. All services are defined in a platform-independent and language-independent manner. Different high-level languages can be implemented on top of them (e.g. Alice, Java). Using just-in-time native code compilation, Alice on SEAM performs up to 4 times faster than Alice on Mozart.

Constraint programming has been decoupled completely from the language and virtual machine. A newly designed, generic constraint programming library called GECODE (*generic constraint development environment*) provides an efficient constraint framework with support for arbitrary kinds of constraint variables (e.g. finite integers, finite integer sets). Preliminary numbers show a more than one order of magnitude performance increase for certain constraint problems, compared to Mozart. From Alice, GECODE is accessible as a library provided for SEAM.

Release. The Alice System was first released for public use in December 2002. Since then there have been frequent updates. The CHORUS project already started migrating from Oz to Alice. The current Alice release includes a comprehensive set of libraries and tools, but is still based on the Mozart implementation. An application-strength system based on SEAM and GECODE will be publicly available by the end of the funding period.

The following sections describe the results in more detail, in accordance to the six work packages identified in the project proposal.

2.4.1 WP1: Language, Typing, Case Studies

Alice has been conceived as a conservative extension of SML. The main extensions inherited from Oz are high-level language support for concurrency, constraints, persistence, distribution, and dynamic modules. The Alice ML language and its extensions with respect to SML are described in detail in the Alice manual (Alice Team, 2004). As a test-bed for preliminary language design studies we developed a small reference implementation of Standard ML, which has been released separately (Rossberg, 2002). That phase uncovered some minor inconsistencies in the formal language specification of SML (Rossberg, 2001).

Futures. Concurrency and laziness in Alice are uniformingly based on *futures*, a refinement of logic variables. Futures are place holders for values not yet computed and allow for implicit dataflow synchronisation. They enable expressing a variety of high-level concurrency constructs, including channels, ports, and semaphores (Smolka, 1998, 1999). Furthermore, *failed futures* provide for clean interaction of futures and concurrency with exceptions. We have formalised the semantics of futures and their interaction with mutable state in an extension of the lambda calculus. We have proved safety of this calculus on basis of a linear type system (Niehren, Schwinghammer, & Smolka, 2003; Schwinghammer, 2002).

Constraints. The constraint system underwent a complete redesign. It has been decoupled from the language proper and moved to a library (see Section 2.4.5). Typing of constraints turned out to be mostly straightforward. The type system distinguishes strictly between constraint variables and integer values, hence avoiding a common pitfall in Oz programs. Feature constraints have been omitted from Alice, since they are not needed for most applications.

Dynamic Typing. The conventional programming model offered by SML requires that a program is produced from a closed set of components. In contrast, one of the central features of Alice is its *open* programming model. An Alice process can import objects computed by other processes. These objects can contain program fragments in the form of procedures. To ensure that such objects are compatible with the type assumptions made by the importing process, dynamic type checks are necessary.

Alice incorporates dynamic typing through *packages*. Packages are values encapsulating a possibly higher-order module and a dynamic representation of its signature. Packages are hence a generalisation of the concept of *dynamics* (Abadi, Cardelli, Pierce, & Rémy, 1995), which can carry only simple values. They also represent a dynamically typed variant of reified modules (modules as 1st-class values, Dreyer, Crary, & Harper, 2003). A package is created by simple injection of a corresponding module/signature pair. The inverse projection dynamically checks the package's signature against an expected signature, using the subtype relation as specified by the SML module typing rules. Unlike dynamics, packages hence require no complex dynamic type analysis construct in the language, while still being sufficiently flexible. Although package projection is the only operation performing a dynamic type check, packages are expressive enough to type the Alice library interfaces for type-safe input/output, inter-process communication, and the component management system.

Another advantage of packages over dynamics is that only module-level computations are type-directed. Core language polymorphism remains fully parametric (Reynolds, 1983). Para-

metricity provides a strong basis for reasoning about typed programs (Wadler, 1989) and enables an efficient type-erasing compilation scheme for polymorphic expressions (Pierce, 2002). However, the loss of parametricity for modules has an impact on the theoretical interpretation of type abstraction. Type abstraction is a key concept for modular software construction: it provides encapsulation by hiding the implementation of user-defined types and allowing access only through a set of well-defined interface operations. The standard existential type model for type abstraction (Mitchell & Plotkin, 1988) crucially relies on parametricity. To define the semantics of packages, a new formalisation of type abstraction was hence necessary that models type abstraction by dynamic type generation and explicit coercions between an abstract type and its representation (Rossberg, 2003).

Components. A component is the unit of compilation and deployment. It usually depends on a number of *imports* – other components residing at arbitrary locations in the Internet. Components are loaded on demand by the runtime system (see Section 2.4.4). Files containing *pickles* arise as a special case of components that do not have any imports and are in value form. Unlike Oz, Alice hence makes pickles and components freely interchangable.

Components carry interface information in the form of a (higher-order) ML signature. When a component is loaded, its signature is matched against the respective import signatures. Components are hence very close to the concept of packages. The investigation of the precise relation is currently under research and a formal model of components based on packages will be developed in the remaining time of the period.

Type Classes and Overloading. Overloading can be formalised by an instantiation of the generic HM(X) type inference framework (Sulzmann, Odersky, & Wehr, 1999) with constraint handling rules (CHRs), a declarative language for describing incremental constraint solvers (Frühwirth, 1995). Under some sufficient conditions, CHRs can precisely describe the relationship between overloaded identifiers and achieve decidable type inference (Stuckey & Sulzmann, 2002). We provide a general coherence result under such conditions. The coherence property is lost in the case of a strict language, but can be recovered by imposing a condition similar to the value-restriction found in ML (Sulzmann & Rossberg, 2002). Furthermore, we developed an encoding of most of Haskell's type class mechanism (Wadler & Blott, 1989; Peyton Jones, Jones, & Meijer, 1997) into a simple language based on the generic CHR-based overloading mechanism (Rossberg & Sulzmann, 2002).

Case studies. The design of Alice and its implementation has been evaluated with several case studies:

- *Distributed search engine:* We have implemented a framework for distributing the computation of a search problem over an arbitrary number of worker sites (Schulte, 2002). The study asserted that the distribution interface based on packages is practical. It also revealed that the language could profit from the addition of statically typed reified modules, as a complement to the dynamically typed packages.
- *Distributed multi-user game:* A multi-user version of the *Snake* game makes heavy use of inter-process communication and graphics. It showed that the HTTP-based distribution protocol in Alice was too heavy-weight and had to be replaced by a binary one. The abstractions utitilized in the implementation made this easy.
- *Alice server pages:* This application allows Alice to be used as a web scripting language by embedding it into HTML pages, e.g. for implementing web forms. It relies on a preliminary runtime interface to the Alice compiler for executing the embedded code. That interface will be available in future versions of Alice.

Apart from that, the compiler (Section 2.4.2) and most of the runtime system (Section 2.4.4) and tools (Section 2.4.6) have also been implemented in Alice.

2.4.2 WP2: Compiler

The Alice compiler has been written in Alice and is able to bootstrap. A runtime interface that provides dynamic access to compilation is currently under development.

Internal Type Checking. Most compiler stages use a typed intermediate representation of the program. We have developed an internal type checker for the typed representation and integrated it into the compiler. It proved to be helpful with discovering and avoiding bugs during subsequent extensions to the compiler.

Dynamic Types. The compiler uses a standard type-erasing approach to compilation (Pierce, 2002). In order to implement the type-dependent semantics of Alice, dynamic types have to be represented via reification into terms. The type representation is abstracted as part of the runtime library (see Section 2.4.4). The compiler performs the respective program transformation, which particularly includes module types.

Interoperability. Particular care has been taken to design the compilation process to the Mozart Virtual Machine in a way that allows interoperation between components written in Alice and Oz (Kornstaedt, 2001).

2.4.3 WP3: Virtual Machine

SEAM (simple extensible abstract machine) is a virtual machine intended for the efficient execution of Alice and other languages. SEAMis designed to be simple and language-independent, and based on few principled services. SEAM does not know about Alice (runtime) types, nor does it implement any constraint functionality. The constraint system is obtained from the GECODE library (see Section 2.4.5).

Uniform data representation and memory management. All data structures used to represent computations, including code and threads, reside in an abstract *store*, which represents an abstract graph of data nodes. Language specific data structures are modelled on top of the language-independent store structures. The store manages allocation of nodes and their efficient layout in memory. Unused memory is reclaimed using a generational garbage collector.

Platform independent external representation. Store values are converted to a portable *pickle* representation during export (*pickling*), and converted back during import (*unpickling*). Pickling and unpickling has been formalised as generation, respectively execution, of programs in a simple byte code language (Tack, 2003). A language-specific *transfer language* is defined to describe values independent from platform. It also defines an abstract instruction set to represent code. Unpickling operates with respect to a language-dependent transformation. For example, abstract code can be instantiated either to byte code or to native code.

Generic computation model. Computations are defined by a generic interface, which is instantiated by *interpreters*. This allows for different code interpreters to easily be used at the same time and interact freely. The current implementation of SEAM uses one external code representation (the one defined in the transfer language) together with three different interpreters: a classical interpreter, a debug interpreter used for debugging Alice programs, and a native code interpreter providing for native code execution.

We have a running implementation of SEAM which covers most of Alice. It will be ready for release until the end of the project. The design goals have been met successfully:

Simplicity. In total, the size of the SEAM core components is around 25 percent the size of the respective Mozart components (Brunklaus & Kornstaedt, 2003).

Genericity. SEAM is a suitable platform for programming language research. A prototype of the Java Virtual Machine (Lindholm & Yellin, 1999) has been implemented using SEAM with less than three person weeks effort (Brunklaus & Kornstaedt, 2002).

Efficiency. Standard benchmarks from Scheidhauer (1998) show that Alice on SEAM is up to 4 times faster than Oz on Mozart. SEAM's pickling and unpickling performance is also superior to Mozart on all benchmarks (Brunklaus & Kornstaedt, 2003).

2.4.4 WP4: Runtime System

The runtime system comprises the non-primitive portions of the virtual machine environment and has been written in Alice itself. It consists of three main parts.

Runtime Type Library. Implements the runtime type representation and respective operations for reifying the type language into terms. It is also used by the component manager. The current implementation is complete but performance can degrade for large signatures. A thorough investigation of an efficient representation and its algorithms is an open problem.

Component Manager. Performs localisation, loading, type-checking and evaluation of imported components. While implemented in Alice, it uses unsafe reflection at designated points.

Distribution System. The distribution model is vastly simplified in Alice, compared to Oz. In particular, it abstains from supporting mobility for futures and state. This had the desired effect of reducing the size of the implementation to 2% of Mozart's distribution subsystem.

2.4.5 WP5: Libraries

The Alice system includes a comprehensive library, providing generic data structures, database access, network programming functionality, and access to Alice-specific services like concurrency, distribution, pickling and component management. Alice allows programming of user interfaces and graphics through a wrapper for the wide-spread Gtk library. It has been generated semi-automatically by a generic foreign function interface tool (Section 2.4.6).

GECODE. The GECODE (*generic constraint development environment*) library is a modular, language-independent framework for constraint programming. It provides generic services for propagation and search, and interfaces to plug in specific types of constraint variables and propagators. Modules for finite integers and finite integer sets are included, others can be added easily.

The functionality of GECODE subsumes most of Mozart. State-of-the-art techniques like prioritized propagators, iterator-based variable interfaces, and batch recomputation (Choi, Henz,

& Ng, 2001) result in an order of magnitude performance increase for certain problems. Finite integer constraints can use either domain- or bounds-consistent propagators, allowing optimised modelling of constraint problems (Schulte & Stuckey, 2001).

GECODE is available in Alice as a set of library components for SEAM. Spaces and different types of constraint variables are available as first-class values with abstract types. The standard search engines (including the *Explorer*, see Section 2.4.6) are implemented in Alice itself.

2.4.6 WP6: Tools

The Alice System contains a rich set of tools.

Compiler and Static Linker. The compiler architecture allows multiple backends and currently supports code generation for Mozart, SEAM, the Java Virtual Machine, and the Microsoft .NET Common Language Runtime.

Interactive Toplevel ("Interpreter"). Every individual input is treated as a component source, compiled, and then loaded as a dynamic component. Only few modifications to the compiler were necessary for supporting auto loading of import signatures. An Alice mode for integrating the toplevel into Emacs is available.

Inspector and Explorer. The graphical tools for browsing of arbitrary data structures and for browsing search trees as known from Mozart have been reimplemented using the Gtk library. The difficulties in reimplementing the Inspector expected in the proposal due to the lack of object-oriented programming features in Alice did not materialise. Unlike runtime values in Oz however, values in an Alice process are not self-describing. The Alice Inspector hence relies on the dynamic type information provided by the dynamic typing facilities for displaying values.

FFI Generator. For SEAM, an FFI (*foreign function interface*) generation tool has been developed. It allows semi-automatic generation of Alice components interfacing C libraries.

Debugger. The SEAM version of Alice will feature a source-level debugger. The debugger makes essential use of SEAM's generic architecture that allows multiple code interpreters. Like the Inspector, it also relies on dynamic type information to print program values.

Lexer and Parser. Yacc-style lexer and parser generator pre-processors for Alice will be part of the next major release.

2.4.7 Cooperation within the Collaborative Research Center

The Alice system has reached a sufficient state of maturity to make it useful for other projects. CHORUS (MI 2) started migrating parts of its Oz based software to Alice in 2003 and intends to continue on that path in the next funding period. Some new CHORUS applications are already developed with SEAM and GECODE. CHORUS makes extensive use of finite set constraints and hence pushes the integration of these into GECODE. The performance increase for constraint programs will make Alice on SEAM a similarly attractive target for other projects computing large constraint problems.

2.5 Comparison with Research outside of the Collaborative Research Center

Distribution and Dynamic Typing. Sewell has presented a system that models distributed components exchanging data of abstract type (Sewell, 2001). However, unlike ours, his system is not dynamically abstraction-safe. Recent work refines the abstraction mechanism by identifying abstract types by a hash over their implementation (Leifer, Peskine, Sewell, & Wansbrough, 2003). While this provides a more flexible approach to abstract types, it still sacrifices abstraction safety in the presence of state. Weirich and others have investigated extensions of intensional type analysis, of which the unpack construct for packages can be seen as a variant, to higher-order (Weirich, 2002) and issues of compilation (Weirich, 2001; Crary, Weirich, & Morrisett, 2002).

Modules and Sealing. (Dreyer et al., 2003) have developed a type system for modules that unifies previous notions of static sealing (type abstraction). Followup work by Dreyer (2004) addresses the problem of recursive modules. Neither addresses dynamic sealing. Sumii and Pierce have developed an untyped calculus and proof techniques for dealing with dynamic sealing (Sumii & Pierce, 2003, 2004).

Virtual Machines. The Virtual Virtual Machine (Folliot et al., 2002) provides a core machine that defines a simple intrinsic object and memory model extensible with environment specifications called *VMlets*. Unlike SEAM, it also defines a RISC-like instruction set. The Microsoft Common Language Runtime supports different programming languages (Hamilton, 2003). Unlike SEAM, it defines an object-centered, typed intermediate language. Metadata makes programs self-describing, allowing language-independent tools. VMGen (Ertl, Gregg, Krall, & Paysan, 2002) is designed to automatically generate efficient bytecode interpeters

from a given description. It incorporates many state-of-the-art interpretation techniques and could be used to generate interpreters for SEAM.

2.6 Open Issues

Given the wide range of questions that arise in the context of a new programming language and system, some of the more interesting questions that could not be addressed within the NEP project are the following.

Type language. The Alice type language is a higher-order recursive lambda calculus with ad-hoc rules to ensure termination. A more thorough formal treatment is needed.

First-class modules and type classes. Case studies revealed the need for having statically typed first-class modules. They might also be a basis for integrating type classes seamlessly.

Distribution. The distribution model is not formalised. Pickling causes runtime exceptions on any attempt to export a value containing local resources. It is not obvious how the type system could be extended to prevent this sort of failure.

Optimisation of futures. The necessity to dynamically test for futures (*touch*, Flanagan & Felleisen, 1999) everywhere imposes a significant performance overhead. Optimising these tests based on data-flow analysis is an open problem.

Open VM specification. The component format is not documented. A format open for third parties requires a typeful and verifiable specification of component consistency. The untyped nature of the lower-level component format is currently incompatible with this.

Type representation. The runtime type representation is expensive and does not maintain minimality, which is difficult due to the complex structure and equivalence rules of the type language. A minimal representation based on Mauborgne (2000) and optimised algorithms are desirable, but require a non-standard reformulation of the type language.

Dynamic recompilation and caching. Runtime compilation alone cannot eliminate the overhead of lazy linking and futures. Dynamic recompilation is used as a solution for Java (e.g. Arnold, Hind, & Ryder, 2002). Since runtime compilation can be costly, caching of compiled components is also desirable (Serrano, Bordawekar, Midkiff, & Gupta, 2000). It remains an open question how either technique can be applied to Alice.

Memory management and concurrency for native bindings. The current support for native library bindings ensures mutual exclusion and assumes that a native call does return after a short period of time. It is not clear how these restrictions could be lifted.

References

Abadi, M., Cardelli, L., Pierce, B. C., & Rémy, D. (1995). Dynamic typing in polymorphic languages. *Journal of Functional Programming, 5(1)*, 111–130.

Alice Team. (2004). *The Alice System.* http://www.ps.un-sb.de/alice/.

Arnold, M., Hind, M., & Ryder, B. G. (2002). Online-feedback directed optimization of Java. In *Conference on Object-oriented Programming Systems, Languages and Applications* (pp. 111–129). Seattle, USA.

Brunklaus, T., & Kornstaedt, L. (2002). *A virtual machine for multi-language execution* (Tech. Rep.). Programming Systems Lab, Universität des Saarlandes. (http://www.ps.uni-sb.de/Papers/abstracts/multivm.html)

Brunklaus, T., & Kornstaedt, L. (2003). *Open programming services for virtual machines* (Tech. Rep.). Programming Systems Lab, Universität des Saarlandes. (http://www.ps.uni-sb.de/Papers/abstracts/vmservices.html)

Choi, C. W., Henz, M., & Ng, K. B. (2001). Components for state restoration in tree search. In T. Walsh (Ed.), *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming* (Lecture Notes in Computer Science, vol. 2239). Springer Verlag.

Crary, K., Weirich, S., & Morrisett, G. (2002). Intensional polymorphism in type erasure semantics. *Journal of Functional Programming, 12*(6), 567–600.

Dreyer, D. (2004). A type system for well-founded recursion. In *31st Symposium on Principles of Programming Languages.* Venice, Italy.

Dreyer, D., Crary, K., & Harper, R. (2003). A type system for higher-order modules. In *30th Symposium on Principles of Programming Languages.* New Orleans, USA.

Ertl, M. A., Gregg, D., Krall, A., & Paysan, B. (2002). Vmgen – a generator of efficient virtual machine interpreters. *Software – Practice and Experience, 32(3)*, 265–294.

Flanagan, C., & Felleisen, M. (1999). The semantics of future and an application. *Journal of Functional Programming, 9(1)*, 1–31.

Folliot, B., Piumarta, I., Seinturier, L., Baillarguet, C., Khoury, C., Leger, A., & Ogel, F. (2002). Beyond flexibilty and reflection: The VVM approach. In *Advanced Environments, Tools, and Applications for Cluster Computing* (Lecture Notes in Computer Science Vol. 2326). Mangalia, Romania: Springer-Verlag.

Frühwirth, T. (1995). Constraint handling rules. In *Constraint programming: Basics and trends* (Lecture Notes in Computer Science Vol. 910). Springer-Verlag.

Hamilton, J. (2003). Language integration in the common language runtime. *ACM SIGPLAN Notices, 38(2)*, 19–28.

Kornstaedt, L. (2001). Alice in the land of Oz – an interoperability-based implementation of a functional language on top of a relational language. In *First Workshop on Multi-language Infrastructure and Interoperability (BABEL'01)* (Electronic Notes in Computer Science Vol. 59). Florence, Italy: Elsevier Science Publishers.

Leifer, J., Peskine, G., Sewell, P., & Wansbrough, K. (2003). Global abstraction-safe marshalling with hash types. In *Eighth International Conference on Functional Programming.* Uppsala, Sweden.

Lindholm, T., & Yellin, F. (1999). *The Java virtual machine specification* (2nd ed.). Addison Wesley.

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

Mitchell, J., & Plotkin, G. (1988). Abstract types have existential type. *Transactions on Programming Languages and Systems, 10*(3), 470–502.

Mozart Consortium. (2003). *The Mozart programming system.* (http://www.mozart-oz.org/)

Niehren, J., Schwinghammer, J., & Smolka, G. (2003). *A concurrent lambda calculus with futures* (Tech. Rep.). Programming Systems Lab, Universität des Saarlandes.

Peyton Jones, S., Jones, M. P., & Meijer, E. (1997). Type classes: an exploration of the design space. In *Proceedings of the Haskell Workshop* (Lecture Notes in Computer Science Vol. 788).

Pierce, B. (2002). *Types and programming languages.* The MIT Press.

Reynolds, J. (1983). Types, abstraction and parametric polymorphism. In *Information Processing* (pp. 513–523). Amsterdam: North Holland.

Rossberg, A. (2001). *Defects in the Revised Definition of Standard ML* (Tech. Rep.). Saarbrücken, Germany: Saarland University.

Rossberg, A. (2002). *Hamlet – To be or not to be Standard ML.* http://www.ps.uni-sb.de/hamlet/.

Rossberg, A. (2003). Generativity and dynamic opacity for abstract types. In *Fifth International Conference on Principles and Practice of Declarative Programming.* Uppsala, Sweden.

Rossberg, A., & Sulzmann, M. (2002). *Beyond type classes* (Tech. Rep.). Saarbrücken, Germany: Universität des Saarlandes.

Scheidhauer, R. (1998). *Design, Implementierung und Evaluierung einer virtuellen Maschine für Oz.* Doctoral Dissertation, Universität des Saarlandes.

Schulte, C. (2002). *Programming constraint services* (Lecture Notes in Artificial Intelligence Vol. 2302). Berlin, Germany: Springer-Verlag.

Schulte, C., & Stuckey, P. J. (2001). When do bounds and domain propagation lead to the same search space. In *Third International Conference on Principles and Practice of Declarative Programming.* Florence, Italy.

Schwinghammer, J. (2002). *A concurrent lambda calculus with promises and futures.* Diplomarbeit, Saarbrücken, Germany. (http://www.ps.uni-sb.de/Papers)

Serrano, M., Bordawekar, R., Midkiff, S., & Gupta, M. (2000). Quicksilver: A quasi-static compiler for Java. In *Conference on Object-oriented Programming Systems, Languages and Applications.* Minneapolis, USA.

Sewell, P. (2001). Modules, abstract types, and distributed versioning. In *28th Symposium on Principles of Programming Languages.* London, UK.

Smolka, G. (1995a). The definition of Kernel Oz. In A. Podelski (Ed.), *Constraint programming: Basics and trends* (Lecture Notes in Computer Science Vol. 910). Springer-Verlag.

Smolka, G. (1995b). The Oz programming model. In J. van Leeuwen (Ed.), *Computer science today* (Lecture Notes in Computer Science Vol. 1000, pp. 324–343). Berlin: Springer-Verlag.

Smolka, G. (1998). Concurrent constraint programming based on functional programming. In *Programming Languages and Systems* (Lecture Notes in Computer Science Vol. 1381, pp. 1–11). Lisbon, Portugal: Springer-Verlag.

Smolka, G. (1999). *From concurrent constraint programming to concurrent functional programming with transients* [Slides]. Summer School on Constraints in Computational Logics, Gif-sur-Yvette, France. (http://www.ps.uni-sb.de/ smolka/ccl99.ps)

Stuckey, P. J., & Sulzmann, M. (2002). A theory of overloading. In *Seventh International Conference on Functional Programming.* Pittsburgh, USA.

Sulzmann, M., Odersky, M., & Wehr, M. (1999). Type inference with constrained types. *Theory and Practice of Object Systems, 5(1).*

Sulzmann, M., & Rossberg, A. (2002). *A theory of overloading Part II: Semantics and coherence* (Tech. Rep.). University of Melbourne. (http://www.cs.mu.oz.au/tr_submit/test/cover_db/mu_TR_2002_1.html)

Sumii, E., & Pierce, B. (2003). Logical relations for encryption. *Journal of Computer Security, 11*(4), 521–554.

Sumii, E., & Pierce, B. (2004). A bisimulation for dynamic sealing. In *31st Symposium on Principles of Programming Languages.* Venice, Italy.

Tack, G. (2003). *Linearisation, minimisation and transformation of data graphs with transients.* Diploma thesis, Universität des Saarlandes, Saarbrücken.

Van Hentenryck, P., Saraswat, V., et al.. (1997). Strategic directions in constraint programming. *ACM Computing Surveys, 28*(4), 701–726.

Wadler, P. (1989). Theorems for free! In *4th International Conference on Functional Programming and Computer Architecture.* London, UK.

Wadler, P., & Blott, S. (1989). How to make ad-hoc polymorphism less ad-hoc. In *16th ACM Symposium on Principles of Programming Languages* (pp. 60–76).

Weirich, S. (2001). Encoding intensional type analysis. In *10th European Symposium on Programming* (Lecture Notes in Computer Science Vol. 2028). Genova, Italy: Springer-Verlag.

Weirich, S. (2002). Higher-order intensional type analysis. In *11th European Symposium on Programming.* Grenoble, France.