THE STANDARD ML BASIS LIBRARY

EDITED BY

Emden R. Gansner and John H. Reppy

CAMBRIDGE www.cambridge.org/9780521791427

The Standard ML Basis Manual

This book provides a description of the Standard ML (SML) Basis Library, the standard library for the SML language. For programmers using SML, it provides a complete description of the modules, types, and functions comprising the library, which is supported by all conforming implementations of the language. The book serves as a programmer's reference, providing manual pages with concise descriptions. In addition, it presents the principles and rationales used in designing the library and relates these to idioms and examples for using the library. A particular emphasis of the library is to encourage the use of SML in serious system programming. Major features of the library include I/O, a large collection of primitive types, support for internationalization, and a portable operating system interface.

This manual will be an indispensable reference for students, professional programmers, and language designers.

Emden R. Gansner is a Principal Technical Staff Member at AT&T Laboratories. Having taught at several prestigious universities, he is currently an adjunct Professor of Computer Science at Stevens Institute of Technology. He has published articles in numerous journals, such as the *Journal of Combinatorial Theory*, *Discrete Mathematics*, and *SIAM Journal of Algorithms and Discrete Methods*. He also jointly received a patent on a technique for drawing directed graphs.

John H. Reppy is an Associate Professor of Computer Science at the University of Chicago. He recently served as Associate Editor of *ACM TOPLAS* and is the author of *Concurrent Programming in ML*, also published by Cambridge University Press.

The Standard ML Basis Manual

Edited by

Emden R. Gansner
AT&T Laboratories

John H. Reppy
University of Chicago

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore, São Paulo

Cambridge University Press
The Edinburgh Building, Cambridge CB2 2RU, UK

Published in the United States of America by Cambridge University Press, New York

www.cambridge.org
Information on this title: www.cambridge.org/9780521791427

First published in print format 2004

ISBN-13 978-0-511-19570-9 eBook (NetLibrary)
ISBN-10 0-511-19570-2 eBook (NetLibrary)

ISBN-13 978-0-521-79142-7 hardback
ISBN-10 0-521-79142-1 hardback

ISBN-13 978-0-521-79478-7 paperback
ISBN-10 0-521-79478-1 paperback

Contents

Foreword

Of all modern programming languages, Standard ML has ascribed perhaps the highest priority to rigorous semantic definition. It is therefore the preferred language for many applications where rigor is important; this is notably true of tools for formal program analysis. It has also gained users who value its high degree of portability, a direct consequence of the unambiguity of its definition.

Now Emden Gansner and John Reppy have equipped SML with another essential ingredient: a library of signatures, structures, and functors which will greatly ease the programmer's task. The SML Basis Library has been long in gestation, but this has ensured that it contains the right things. Only by close cooperation with users, over a considerable period of time, can one be sure of consistency and balance in defining a library. We can therefore be confident that the Basis Library will bring SML into still wider use, and we owe warm thanks to its creators for undertaking an arduous task with skill, care, and dedication.

Robin Milner
Cambridge, July 2003

Preface

One essential for the success of a general-purpose language is an accompanying standard library that is rich enough and efficient enough to support the basic, day-to-day tasks common to all programming. Libraries provide the vocabulary with which a language can be used to say something about something. Without a broad common vocabulary, a language community cannot prosper as it might.

This document presents a standard basis library for SML. It is a basis library in the sense that it concerns itself with the fundamentals: primitive types such as integers and floating-point numbers, operations requiring runtime system or compiler support, such as I/O and arrays; and ubiquitous utility types such as booleans and lists. The SML Basis Library purposefully does not cover higher-level types, such as collection types, or application-oriented APIs, such as regular expression matching. The primary reason for limiting the scope in this way is that the design space for these interfaces is large (e.g. choosing between functors and polymorphism as a parameterization mechanism) and, unlike the case with lists and arrays, we do not have many years of common practice to guide the design. It is also the case that the SML Basis Library specification is a substantial document and expanding its scope would make it unwieldy.

The primary purpose of this book is to serve as a reference manual for the Basis Library, describing as clearly and completely as possible the types, values, and modules making up the Library. This specification is designed to serve both implementors of the SML Basis Library and users. While the specification is not formal, we have tried to make it precise and complete enough to guarantee a high degree of portability between implementations.

It is sometimes difficult to program from a reference manual; all the pieces are there but it is not clear how they fit together. For the working programmer who wants to use the Library, the book also discusses how the functions were meant to be used alone and together. Although not a tutorial, the book should assist the programmer in understanding and using the Library, clarifying when and how various structures should be used, and making the apparent arcana accessible.

There are certain roles the book does not attempt. As we've already noted, it is not a textbook, for either the Library or SML. There are already many fine books and papers teaching the joys of writing in SML, some of which address this Library as well. When dealing with the Library's interface to external software such as Unix or Windows, it assumes the reader already knows how to use them or has access to sources providing that information.

The Library is certainly not complete; there are some glaring omissions, such as a module for handling regular expressions or guidelines for internalization. It is assumed that, as needs are identified and consensus is reached on the design of a structure, new modules will be added to the Library or be standardized as a separate library. The evolution of the Library will be reflected in the online version of this document, the latest version of which can be found at

`http://standardml.org/Basis`

Overview of the book

The book is organized in three main parts: an overview of the Library, its structure and conventions; a tour of the main areas covered by the Library, providing programming tips, idioms, and examples aimed at Library users; and a set of manual pages defining the signatures and structures composing the Library.

The first three chapters form the first part. Chapter 1 presents the philosophy, principles, and rules concerning the design of the Library. It also notes the conventions used in documenting the Library. The second chapter lists all of the signatures, structures, and functions in the Library, noting their connections and whether they are optional. Chapter 3 considers those parts of the Library that are available at the top level, outside of any structure.

The following chapters describe some of the component areas, such as I/O and text handling, in more depth. These chapters discuss the common themes connecting the modules of a component, and note related assumptions and restrictions. The Library includes some elegant solutions to certain programming tasks, but these are not necessarily obvious from a bare presentation of the signatures. Thus, many of these chapters include short tutorial sections that discuss how various types and functions were intended to be used, including examples of idiomatic use.

Chapter 11 is the meat of the book, containing manual pages describing the signatures, structures, and functors specified by the Library, and their semantics. The modules are presented in alphabetical order. Generic modules, those with multiple possible implementations, are gathered under their defining signature. Thus, the `Char` structure is discussed in the `CHAR` section. Each non-generic module, those with a unique implementation, such as `Timer`, heads its own section shared with its signature. Significant substructures, for example, `Posix.IO` also rate their own sections.

During the design of the Library, the authors of the SML language have revised its definition [MTHM97], partly in response to the needs of the Library. The appendix describes some of the changes that have taken place in the SML language, especially in relation to the Library, and also notes where the Library differs from the initial basis described in the original SML definition. The back matter also provides an index of the exceptions defined in the Library.

Contributors

The main architects of the SML Basis Library are Andrew Appel, Dave Berry, Emden Gansner, John Reppy, and Peter Sestoft. In addition, the following people contributed to the design discussions and writing: Nick Barnes, Lal George, Lorenz Huelsbergen, David MacQueen, Dave Matthews, Carsten Müller, Larry Paulson, Riccardo Pucella, and Jon Thackray. This document is edited and maintained by Emden Gansner and John Reppy.

Acknowledgments

As usual in a work like this, many people have been involved in the process. We had helpful comments on the Library or this document from Peter Michael Bertelsen, Matthias Blume, Jeremy Dawson, Matthew Fluet, Elsa Gunter, Ken Larsen, Peter Lee, Neophytos Michael, Kevin Mitchell, Brian Monahan, Stefan Monnier, Chris Okasaki, Andreas Rossberg, Jon Thackray, Mads Tofte, Dan Wang, and Stephen Weeks. David Gay and Serban Jora helped with insights and pointers concerning IEEE floating-point numbers and the Windows operating system, respectively. We would especially like to thank our editor at Cambridge University Press, Lauren Cowles, whose patience has been unbounded.

This document was written using the *ML-Doc* toolkit, which is an SGML-based system for documenting SML interfaces. More information about ML-Doc can be found at

`http://people.cs.uchicago.edu/~jhr/tools/ml-doc.html`

1
Introduction

This document describes the Standard ML Basis Library. The Library provides an extensive collection of basic types and functions for the Standard ML (SML) language, as described by the Definition of Standard ML (Revised) [MTHM97]. The goals of the Basis library are to:

- serve as the basic toolkit for the SML programmer, whether novice or professional;
- focus attention on the attractiveness of SML as a language for programming in a wide variety of domains, e.g., systems programming;
- replace the many incompatible general-purpose libraries currently available.

The original definition of the Standard ML language [MTH90] was published in 1990, for which reason we refer to it as SML'90. The Definition specified an *initial basis*, i.e., a set of primitive types such as `int` and `string` along with some related operations, which was used to define various derived forms and special constants. Though adequate for the purpose of language specification, it was too limited for programming applications. In response, most implementations of the language extended the basis with large collections of generic libraries. With the libraries coming from different sources, they tended to be incompatible, even when implementing the same abstract types and functions. The result was that, despite the standardization of the language, any significant SML program could be compiled on multiple implementations only if the programmer were willing to provide portable libraries that relied only on the initial basis.

The SML Basis Library is a rich collection of general-purpose modules, which can serve as the foundation for applications programming or for more domain-specific libraries. It provides most of the basic types and operations expected by a working programmer and specifies that anyone using SML can expect to find them in any implementation.

Some goals in designing the Library worked toward its expansion. One, suggested above, was the desire for the Library to be "complete enough." If using a type provided by the Library, the programmer should be able to look in the defining structure and find the right function or, at least, the functions needed to build the desired function easily. In addition, the Library attempts to provide similar functions in similar contexts. Thus, the traditional `app` function for lists, which applies a function to each member of a list, has also been provided for arrays and vectors.

An opposite design force has been the desire to keep the Basis library small. In general, a function has been included only if it has clear or proven utility, with additional emphasis on those that are complicated to implement, require compiler or runtime system support, or are more concise or efficient than an equivalent combination of other functions. Some exceptions were made for historical reasons or for perceived user convenience.

The SML language has the rare property of being a practical, general-purpose programming language possessing a well-defined, indeed formal, semantics. Following in this spirit, some SML-based libraries, e.g., CML [Rep99], build on this precision by supplying their own formal semantics. Although we viewed this goal as beyond what we could provide for the Basis library, we still felt very strongly that the functions included here should be defined as precisely and clearly as possible. In some cases, we have defined the meaning of basis functions via reference implementations. We want SML programs to be *deterministic* (aside from their interaction with the external world), and so we specify the traversal order for higher-level functions such as `List.map`. The description of a function provides the dynamic constraints on the arguments, such as that an integer index into an array must be less than the length of the array, and relates what happens when a function invocation violates these constraints, typically the raising of a particular exception. We have tried to stipulate completely the format of return values, so that, when a type's representation is visible, the programmer will know what to expect concretely, not just abstractly. We have avoided unspecified or implementation-dependent results whenever possible. Some functions were excluded from the Library because we could not provide a clean specification for the function's behavior.

1.1 Design rules and conventions

In designing the Library, we have tried to follow a set of stylistic rules to make library usage consistent and predictable, and to preclude certain errors. These rules are not meant to be prescriptive for the programmer using or extending the Library. On the other hand, although the Library itself flouts the conventions on occasion, we feel the rules are reasonable and helpful and would encourage their use.

1.1.1 Orthographic conventions

We use the following set of spelling and capitalization conventions. Some of these conventions, e.g., the capitalization of value constructors, seem to be widely accepted in the user community. Other decisions were based less on a dominant style or a compelling reason than on compromise and the need for consistency and some sense of good taste.

The conventions we use are

- Alphanumeric value identifiers are in mixed-case, with a leading lowercase letter; e.g., `map` and `openIn`.
- Type identifiers are all lowercase, with words separated by underscores; e.g., `word` and `file_desc`.
- Signature identifiers are in all capitals, with words separated by underscores; e.g., `PACK_WORD` and `OS_PATH`. We refer to this convention as the *signature* convention.
- Structure and functor identifiers are in mixed-case, with initial letters of words capitalized; e.g., `General` and `WideChar`. We refer to this convention as the *structure* convention.
- Alphanumeric datatype constructors follow the signature convention; e.g., `SOME`, `A_READ`, and `FOLLOW_ALL`. In certain cases, where external usage or aesthetics dictates otherwise, the structure convention can be used. Within the Basis library, the only use of the latter convention occurs with the months and weekdays in `Date`, e.g., `Jan` and `Mon`. The only exceptions to these rules are the identifiers `nil`, `true`, and `false`, where we bow to tradition.
- Exception identifiers follow the structure convention; e.g., `Domain` and `SysErr`.

These conventions concerning variable and constructor names, if followed consistently, can be used by a compiler to aid in detecting the subtle error in which a constructor is misspelled in a pattern-match and is thus treated as a variable binding. Some implementations may provide the option of enforcing these conventions by generating warning messages.

1.1.2 Naming

Similar values should have similar names, with similar type shapes, following the conventions outlined above. For example, the function `Array.app` has the type:

```
val app : ('a -> unit) -> 'a array -> unit
```