

Bibliotheca Opalica

A Document on Structured Use And Abuse

The OPAL Group
written by Klaus Didrich

Copyright © 1994, 1995, 1996 The OPAL Group

1 Introduction

A library is one of the integral components of modern languages. Its function is to relieve the user of the task of constantly redefining standard structures. This is especially true for OPAL, since only two structures, `BOOL` and `DENOTATION` are predefined. A library for OPAL, containing most data types and functions needed for everyday use, has been developed in recent years.

These structures do belong to the OPAL release, but they are not part of the language. Most of them are written in OPAL and only some are substituted by handwritten code (see Section 4.1 [Handcoded], page 14) for reasons of efficiency (both space and time). This difference is not apparent to the OPAL user.

This paper offers you an overview of BIBLIOTHECA OPALICA, so that you will be able to use (and abuse) the structures in the library. That is, you will not find explanations of the functions, but you will learn where these explanations are located. If you need information about a particular function, please have a look at the signature parts.

It is not the purpose of this paper to give you an introduction to OPAL; for this you are referred to *The Programming Language OPAL — The Implementation Language* and *The OPAL Tutorial*. Neither does the paper serve as an introduction to the compilation system, which is explained in *A User's Guide to the OPAL Compilation System*.

We first outline the structuring principles which were used in the design of the library. Then, we proceed through the library structures explaining what types are defined and what additional structures are available. Last, we turn to some particularities of BIBLIOTHECA OPALICA. The appendix contains, among other topics, a list of the currently known bugs and an aid to converting programs from the old library to BIBLIOTHECA OPALICA.

The OPAL library has matured in recent years and summarizes the experience of several years of functional programming. We would like to also include *your* experience in BIBLIOTHECA OPALICA. So, if you have any suggestions or contributions to make, please do not hesitate to drop us a line or send us a piece of code which may appear in the next release of BIBLIOTHECA OPALICA.

The OPAL Group is reachable via email: `opal-users@projects.uebb.tu-berlin.de`.

The OPAL distribution is available via WWW at the URL `http://projects.uebb.tu-berlin.de/opal/`

2 Structuring Principles

BIBLIOTHECA OPALICA consists of more than 100 different structures. Lest the user get lost, this multitude of structures clearly requires a structure of its own.

The first principle of structuring is of course the division of structures according to their kind, i.e. predefined structures, parameterized structures, environment access structures and the like. This is detailed in Chapter 3 [Survey], page 5.

There are, however, other structuring principles which are valid across these divisions. One of them is the naming of structures which follows certain principles and allows you to conclude some information about a structure from its name.

The other is the principle that similar functions have the same name in OPAL, even if they work on different data types. Typical examples of such functions are functions for converting data types to textual representation.

2.1 What's in a Name? — Naming of Structures

Roses smell as sweet by any other name, and likewise the semantics of a structure is not affected by its name. To the user, however, the name of a structure provides valuable clues about the content. By adhering to naming principles, the task of remembering the existing structures is also made easier.

In BIBLIOTHECA OPALICA the following principles are used:

Type This is always the basic structure for the mentioned *type*. It contains the type definition, if possible with a free type declaration, and the most basic functions that are needed for using this *type*. Note that types always start with lowercase (e.g. `nat`), while structure names are capitalized (e.g. `Nat`).

TypeConv These structures (e.g. `CharConv`, `SetConv`) contain functions that convert from data type *type* into another type, while preserving information; i.e. they are (in principle) invertible. Typical examples are conversions to text or converting different types of numbers.

This also works the other way round: If you are looking for a function which converts members of type *type*, it is usually found in Structure *TypeConv*. There are, however, exceptions to this rule:

There is no Structure `DenotationConv`. Conversions from `denotation` to *type* are considered to be a kind of "pseudo-constructor" and are therefore placed in Structure *Type*.

Conversions from sequences to another aggregate type (see Section 3.4 [AggregateTypes], page 7) are likewise not included in `SeqConv` but in *TypeConv*.

TypeFilter

Functions in these kind of structures (like `SetFilter`) take a bool-valued function and an aggregate type and remove those elements which — as required — either fulfill or do not fulfill the given predicate.

TypeFold

TypeReduce

These structures (**SeqFold**, **SeqReduce**, and so on) contain functions which connect every element of an aggregate type with a given function. Reducing requires a start value while folding does not, but the latter is more restrictive, e.g. it is not defined on empty objects.

TypeMap In structures like **SeqMap**, **SetMap**, and their counterparts for other aggregate types, we find functions for applying one or several functions to every element of the aggregate type.

NameByAttr

These structures give hints about different kinds of implementation of *Name*. This applies to data types (e.g. sets: **SetByPred**) or different methods for defining certain functions (e.g. orderings: **OrderingByInjection**).

Other kinds of structures are still in the development stage and will be added in later releases.

2.2 Orthogonality

Orthogonality means that functions which perform similar tasks are named similarly. The principles of structuring structures obviously support sticking to the "Principle of Least Astonishment".

Also note that differently implemented types have the same name, and can therefore easily be replaced by another implementation: simply change the imported structure and recompile the code. So for sets of natural numbers you can choose whether to use the ordinary implementation or the bitstring-oriented implementation by switching from `IMPORT Set[nat] ONLY set {} + in ...` to `IMPORT Bitset ONLY set {} + in ...`.

The reader should particularly note the following:

- There is always a function `‘ : type -> denotation`. If the type is parameterized, the functionality is augmented by functions for converting the parameter type, e.g. sequences: `‘ : (data -> denotation) -> seq -> denotation`. The only exception is the type `denotation` itself. Even for types like `set’SetByPred` (see Section 3.4 [AggregateTypes], page 7) which do not provide a sensible textual representation, a function which returns a fixed text describing the type is supplied.

The function `‘` is located in Structure *TypeConv*.

- If a conversion from `denotation` to `type` is given, this function is called `!` and located in Structure *Type*.
- Functions which convert one type into another type are called `asType`, e.g. `asNat` for a function which converts type `int` to type `nat`. The exceptions are the functions `‘` and `!` (see above).
- Aggregate types (see Section 3.4 [AggregateTypes], page 7) have many similarities which are exploited in the naming of functions.

`!` Selection of one of the components.

`#` The size of the aggregate object.

| | |
|-----------------------------|--|
| <code>%</code> | Combining up to eight elements to produce an aggregate. |
| <code>.. iter</code> | Generating elements with the help of a function. |
| <code>*</code> | The mapping operator(s). |
| <code>/ (\)</code> | Reducing and folding. If order of elements matters, then both functions (left and right reduce or fold) are given. |
| <code> </code> | Filtering out elements with a given predicate. |
| <code>exist? forall?</code> | Applying predicates to all elements of an aggregate. |

3 Survey of Structures

We will now take a look at the structures in BIBLIOTHECA OPALICA. We will not examine each and every structure, rather will provide explanations concerning each group of structures. There will be no further comment on structures whose purpose is explained by the naming principles (see Section 2.1 [Naming of Structures], page 2).

The structures of BIBLIOTHECA OPALICA have been grouped into five subsystems, some of which have been further divided into subsubsystems. The user need not be aware of the structure of BIBLIOTHECA OPALICA for using OPAL and the OPAL Compilation System. Searching for information about the library is made easier, though, if you know about the structure.

Internal The subsystem **Internal** contains structures which are dependent on the actual implementation.

BasicTypes

In this subsystem the "bread-and-butter" structures are compiled. This includes structures for handling booleans, denotations (see Section 4.3 [Text], page 14), characters and numbers (see Section 4.2 [Numbers], page 14).

Functions

This subsystem contains structures for creating functions from other functions.

AggregateTypes

Aggregates allow you to group elements together and treat them as a single entity. There are heterogeneous aggregates, such as Cartesian products and disjoint union, as well as homogeneous aggregates, such as sequences (a.k.a. lists), binary trees, sets, mappings and the like.

System Structures in the Subsystem **System** provide access to the environment. This encompasses structures for debugging, access to the facilities of the operating system and, last but not least, structures for input and output.

3.1 Internal

The subsystem **Internal** is divided into further subsystems. The user sees only the subsubsystem **Strange**.

Strange

As the name suggests, these structures are not for everyday use. They do have their purpose, but think twice before you use them. They are not semantically safe and, in most cases, you will be able to do very well without them.

CAST gives you the possibility of lying about the type of an expression. There is no guarantee about any properties of the result. In particular, you may not assume that `cast[a,b] (cast[b,a] (x))` yields `x` again, and be assured that casting, say `pair[nat, char]` to `seq[option[string]]`, will cause considerable confusion.

EQUALITY provides a generic equality on data types which do not contain functions. Note that this constraint is not checked by the compiler.

INLINE provides an interface to usage of C macros. This is only useful in connection with handcoding (see Section 4.1 [Handcoded], page 14).

3.2 BasicTypes

Basic Types are finitely or infinitely enumerable types. Currently, the following types are members of this subsystem:

bool As noted before, **bool** is a built-in sort. The predefined functions can be looked up in structure **BOOL**. Another structure allows for converting **bool** to **denotation**.

char Apart from the basic operations, there is a structure **PrintableChar** which contains definitions for the ASCII character set.

denotation The type **denotation** is predefined. The predefined signature is called **DENOTATION**. The structure **Denotation** supplies additional functions which make usage of denotations possible.

nat, int, real These structures deliver the usual functions on the different kinds of numbers. Note that identifiers like **64** are not special to OPAL — they are like any other identifier. Hence you have to define other numbers for yourself. The advice is to define a constant with an appropriate name, e.g. `DEF hashtablesize == "497"!'`.
The functions **!** are very strict and do not accept anything besides the indicated characters ("anything" includes whitespace!). This is to prevent you from typing mistakes¹.

rel, subrel These structures contain types which are used to indicate a result of a comparison with functions found in a *TypeCompare* structure.

void This is the simplest data type. It consists of exactly one constant constructor **nil**.

ABORT is sort of an annotated "undefined". Use this structure to give the user of your OPAL programs hints as to what might have gone wrong. It is a good idea to add the source, that is, name and origin of the function, to the message. See Section 4.4 [ABORT], page 15.

3.3 Functions

The **Functions** subsystem is divided into three parts: **General** for composition, iteration and the like, **Orderings** for constructing ordering relations, and **Special** for structures which need special care in their usage.

¹ The old library was less strict about this, which sometimes led to confusing results.

General

funct This structure allows a function type to be "packed" in a data type.

Compose, ComposePar

These structures define functions for composing two functions either sequentially or concurrently.

Control This structure offers you basic control structures for iterating functions.

Predicate

This structure allows you to combine bool-valued functions almost the same way boolean values can be combined. Of course, no equality on functions can be defined in OPAL.

Orderings

When defining an (enumerated) type, defining the ordering relations by yourself can often be both tedious and lead to errors. The structures in this subsystem facilitate the task:

OrderingByInjection

This structure is to be instantiated with an injection into the natural numbers and then yields the usual ordering relations.

OrderingByLess

Use of this structure requires still to define the "less" relation, but relieves you of defining the other operations.

InducedRel

This structure is more general than the previous two. You may translate any relation to another type by supplying an appropriate translation function.

Special

Functions provided by structures of this subsystem generally come with the warnings "handle with care" and "use only, if you know what you are doing".

AcceleratorC, AcceleratorF

The purpose of these structures is a cost-effective evaluation of binary functions, which is often available if both arguments are known to be equal. Sometimes it is possible to apply a quick (but somewhat dirty) test for equality, which may fail even if both arguments are equal, but which never yields true if they are not.

3.4 AggregateTypes

The concept of aggregating elements to a single entity is one of the most important in functional programming. BIBLIOTHECA OPALICA offers you many different ways to do this:

- Product types correspond to Cartesian products of different data types.

- Union types allow for the disjoint union of data types.
- Sequence-like types comprehend elements of a single data type while preserving order.
- Tree-like data types structure data in the form of binary trees.
- Set-like data types differ from sequence-like data types in that order does not matter.
- Map-like data types contain pairs of data types which constitute finite mappings.

ProductLike

This subsystem contains structures for the use of Cartesian products of types. Our experience is that more than four sorts are usually not needed.

Type `void` may be considered as the empty product (see Section 3.2 [BasicTypes], page 6).

`pair`, `triple`, `quad`

For these structures functions are provided to build and to unbuild objects, to select components, to map a function onto the components. Additionally, `pair` can be compared. Note, that type `quad` is defined in the Structure `Quadruple`.

`AnonPair`, `AnonTriple`, `AnonQuadruple`

These "anonymous" structures are like their named counterparts, but they refer to the built-in tupling of data types in OPAL which are denoted by the keyword `**`. Hence, only the selectors are defined.

UnionLike

This subsystem contains structures for disjoint union of data types.

Type `void` may be considered as the empty union (see Section 3.2 [BasicTypes], page 6).

`option` This is the most important type in this subsystem. It is the disjoint union of type `void` (see Section 3.2 [BasicTypes], page 6) with an arbitrary other data type.

This type is useful for fusing two functions like `test? : some -> bool` and `select : some -> data` (where `select` is undefined when `test?` fails) into a single function `testAndSelect : some -> option[data]`.

`union` This type comes in three flavours: as union of two, three or four data types.

SeqLike

Here you will find structures for dealing with aggregate types where number and ordering of elements matters.

`seq` By far the most frequently used type in functional programming is the sequence (or list) type. Consequently, this type is well supported by many structures. Besides the structures which are explained by their names (see Section 2.1 [Naming of Structures], page 2), there are structures for indexing sequences (`SeqIndex`), special variants of mapping (`SeqMapEnv`), structures for dealing with nested sequences (`SeqOfSeq`), for sorting sequences (`SeqSort`) and for combining sequences of equal length (`SeqZip`).

string The type **string** can essentially be treated as an instantiation of type **seq** with type **char**. Some additional functions, which ensure backwards compatibility with the old library, are also available. There is no structure for nested strings or for combining two strings of the same length; the comparison operators are defined in the base structure.

For a discussion about the difference to type **denotation**, see Section 4.3 [Text], page 14.

union This type is defined in structure **BTUnion** and is a disjoint union of basic types, i.e. **bool**, **nat**, **int**, **real**, **char**, **string** and **denotation**. It is used in the structures for formatted input and output.

StringFormat, StringScan

These structures contain functions for formatted input and output of the types which are combined in the type **union**'**BTUnion**. The usage is similar to that of the functions **printf** and **scanf** from **C**, so that usage of the functions from **BIBLIOTHECA OPALICA** should be no more difficult than use of the functions from **stdio** of ANSI-C.

TreeLike

Tree-like aggregates store the data in binary trees. We have general binary trees (in the following often simply called *trees*), heaps and balanced search trees.

tree Structures of this family implement functions for binary trees. **TreeCompare** compares trees of different types. The other members of this family are standard (see Section 2.1 [Naming of Structures], page 2). Note that there are more variants of mapping and reducing functions than e.g. on sequences.

IndexingOfTrees

This structure contains functions which handle indices of trees and are independent of the element type. Trees are indexed by the following scheme: The root node is assigned index 0, and if any node has index i , the left child has index $2i+1$, the right child has index $2i+2$. (This differs from other indexing schemes, since we assign 0 to the root node.)

heap Heaps are trees whose root value is less or equal than either one of the children's values. In addition to trees, we can combine heaps and extract the minimum, so you may use this data type for a priority queue.

bstree'**BSTree**

Balanced search trees are trees whose root value is greater than the value of the left child and is less than the value of the right child. The trees are balanced, that is, sizes of both subtrees must be similar. (Actually, the criterion is $1/w \leq \text{size}(\text{left}) / \text{size}(\text{right}) \leq w$, where w is a small positive natural number.)

SetLike

Set-like aggregates are similar to sequences, but order of elements does not matter, and in most cases it also does not matter how often an element is added to an aggregate.

There are five different implementations of sets. This plentifulness shows that there is no ideal implementation for every purpose. The advantages and disadvantages of the additional implementations are given below.

set'Set This family of structures defines functions for finite sets. **SetConstr** comprises functions for Cartesian product and disjoint union of sets, the other structures adhere to the naming principles (see Section 2.1 [Naming of Structures], page 2). All structures take as an additional parameter an ordering predicate, otherwise functions like **subset** or **in** cannot be implemented².

set'SetByInj These structures provide a slightly different interface. Instead of an ordering predicate you must supply an injection into the natural numbers.

set'SetByPred These sets are defined by a describing predicate. This is useful if you want to perform operations on big sets with short descriptive predicates, and it is a device for dealing with infinite sets in OPAL. The disadvantage is that many functions on sets (e.g. equality, subset) are not computable for these sets and can therefore not be implemented.

set'SetByBST These sets are represented internally by balanced search trees and are more efficient in including, excluding and searching single elements than the standard implementation.

set'Bitset The naming of this family of structures is really an anachronism — a better name would be **SetOfNat**. This type is handcoded and provides an efficient (with respect to time and space) device for dealing with sets of small natural numbers.

bag Bags (sometimes called multi-sets) are similar to sets, but the frequency of elements matters. So some additional functions on bags are available; otherwise they are treated in the same way as the standard implementation of sets (**set'Set**).

MapLike

Here you can find structures which implement finite mappings.

map This is the type for general finite mappings. For an efficient implementation, an ordering relation on the domain type must be provided. Reducing and mapping is defined on the codomain, filtering checks both components of a pair. Additional functions are provided for inverting a map (**MapInvert**) and for composing two maps (**MapCompose**).

² An equality predicate would have been sufficient too, but an ordering results in a more efficient implementation.

array This is a handcoded mapping from natural numbers to an arbitrary data type. Indices start at 0. The order on natural numbers induces an order of elements, so both variations of fold and reduce are available

3.5 System

This subsystem subsumes all structures which provide functions for communicating in some manner with the environment. It is in turn divided into four subsystems: **Debugging** contains structures for support in debugging OPAL programs (surprise, surprise!), **Commands** provides the basic declarations for the OPAL Command Script I/O, the structures in **Streams** define a simple interface for accessing files, and **Unix** contains structures tailored for OPAL programs which run under **Unix**.

Debugging

This subsystem contains just one structure, which in our opinion could not be assigned to any of the other subsystems.

DEBUG This structure contains functions which aid in debugging by means of side-effect output of messages. This constitutes dirty programming style and should not be used in your ready-to-be-shipped program, but sometimes debugging is quicker when you don't quite stick to the rules.

Commands

This subsystem comprises structures which define functions on *commands* which are the base type for the OPAL Command Script I/O, and some structures for communicating with the environment which are independent from the underlying operating system.

com The command type **com** is essential for I/O in OPAL. It is always parameterized with the type which is to be passed to the next command. If there is no such type, you have to use type **void** (see Section 3.2 [BasicTypes], page 6). **ComAction** defines abbreviations for the instance **com[void]**. Means for composing commands are given in **ComCompose** and **ComCheck**; **ComSeqReduce** reduces a sequence to a single command. **ComChoice** finally provides external choice between two commands.

agent'ComAgent[result]

An agent executes a monadic command concurrently with other agents. Communication between agents is supported by synchronization with the termination of an agent and by client/server oriented communication via service access points (see structure **ComService**). A special command for delaying an agent is provided in the structure **ComTimeout**.

sap'ComService[in, out]

This structure provides a model for agent communication based on the client / server approach. Client and servers communicate via so-called service access points (SAPs).

- Env** This structure provides access to arguments with which the program was called, and to the environment variables.
- Random** This structure implements a pseudo random number generator.

Streams

Streams are a simple and abstract means for reading or writing data from or to a file.

- Stream** This is the basic structure and contains functions for text input and output.

BinStream

This structure contains functions for reading and writing arbitrary data types. Since the files contain no information about the type which was written into them, strong typing is violated. You can use a tag to identify the correct file type for yourself. Note that this tag is internally augmented to ensure that the binary format used by internal routines is the same.

Unix

The structures described here are tailored especially for the Unix operating system. If OPAL is ever run under another operating system, these structures will most likely be missing.

For the current release, several structures have been added, which provide Posix conformant access to system calls. See below under **ProcessCtrl**, **Signal**, **UserAndGroup** and **Wait**.

- file** The structures **File** and **BinFile** correspond roughly to **Stream** and **BinStream** (see [Streams], page 12). They provide functions for input and output to and from files but are more flexible than the structures in the subsystem **Stream**.

FileSystem

This structure provides access to the Unix file system. This structure contains functions for accessing Unix file attributes like file types, file modes, inodes, devices, atime, ctime, mtime; working with (hard and symbolic) links, renaming files, changing access permissions, working on directories and creating named pipes.

FileSystemFun contains some convenient functions for dealing with file modes.

ProcessCtrl

This structure is the successor of **Process** which is no longer supported but still contained. It provides functions for creating, terminating, mutate processes, getting information about one's own process, and changing some of the attributes of a Unix process (group ids, working directory and the like).

- Signal** This structure enables the user to deal with signals from an OPAL program. There exist functions for masking signals, sending signals, waiting for signals and there is also a limited capability for handling signals.

- time** These structures allow you to access system time from within OPAL.

UserAndGroup

This structure provides functions for managing information about Unix users, Unix groups and their ids.

UnixFailures

This structure contains definitions for constants which describe all possible causes of failure.

Wait

In this structure you will find functions which enable you to wait for the children of the current process.

4 Special Features in BIBLIOTHECA OPALICA

4.1 Handcoded Structures

Some structures in BIBLIOTHECA OPALICA are not implemented in OPAL, rather are handcoded and implemented in the target language of the OPAL compiler. This is done for two different reasons.

One reason is that access to the environment must be provided by some kind of run-time system. This runtime system is not one monolithic block of code with which every OPAL program is burdened. Instead, each structure which accesses different parts of the environment adds only the necessary code.

The other reason is that some basic data types are already supported by the target language. So, for reasons of efficiency, some data types have been handcoded. In contrast to the structures which provide access to the environment, these structures could have been implemented in OPAL.

If, for either of these reasons, you wish to add some handcoded structures to your system, you are kindly referred to the *Handcoder's Guide* which is included in the OPAL distribution.

4.2 Numbers

Numbers belong to the family of handcoded structures (see Section 4.1 [Handcoded], page 14). You should never make assumptions about possible sizes of numbers, since these may change in later releases of BIBLIOTHECA OPALICA. Use the constants `min` and `max` wherever possible.

The following (in)equalities are guaranteed to hold in every implementation of OPAL:

```
min'Nat = 0
max'Nat >= 1073741822

min'Int <= -536870911
max'Int >= 536870910

min'Real <= -1e+37
max'Real >= 1e+37
```

The current implementation exceeds these bounds.

4.3 String vs. Denotation

In BIBLIOTHECA OPALICA two types designed for textual representation are available namely `denotation` and `string`. Both are implemented differently.

The type `denotation` should be used for text which does not change very often, since insertions and deletions are relatively expensive. Space requirements, on the other hand, are low. This type is best used for messages and constant strings.

The type `string` has a user interface which is almost identical to that of sequences. In particular, all higher-order functions are available for strings, so operations on texts can

now directly be defined on type **string**. Note that strings are implemented differently than sequences, so operations on strings are faster than operations on sequences of character, and space consumption is lower.

4.4 ABORT

ABORT is a special function. It is essentially a function which yields the undefined value of the parameter type. Hence, you can use this function to replace the automatically inserted error messages with your own messages, which may provide more information about the real cause.

The most interesting feature about **ABORT** is that tests for **ABORT** are eliminated completely if the optimization switch **-op** is set. Thus, you can develop your program with informative error messages, then turn on the compiler switch and eliminate all the (expensive) checks for definedness.

Appendix A Acknowledgement

BIBLIOTHECA OPALICA is only the latest development of the OPAL library.

The predecessor of BIBLIOTHECA OPALICA was designed by *Wolfram Schulte*, together with *Andreas Fett* and *Gottfried Egger*; the structures which are now found in subsystem *Commands* were designed by *Gottfried Egger* and *Wolfram Schulte*.

The restructuring of the library is the result of fruitful discussions with *Wolfgang Grieskamp*, *Joachim Faulhaber*, *Mario Südholt* and *Sabine Dick*.

The library has profited from many suggestions from (in alphabetical order) *Olaf Brandes*, *Christoph Breitkopf*, *Gottfried Egger*, *Sebastian Erdmann*, *Andreas Fett*, *Christian Maeder* and *Burkhart Wolff*.

The text was carefully proofread by *Niamh Warde*.

Appendix B Bugs And Flaws

There are currently no known bugs or flaws in BIBLIOTHECA OPALICA.

Appendix C Changes in OPAL programs necessitated by BIBLIOTHECA OPALICA

When a new library for OPAL was announced, some people feared they would have to forget all they knew about the old version, and learn the whole new library from scratch. This is not the case.

The most important change is the finer structuring. The old library was divided into three subsystems, two of which were of no interest for ordinary users. Now, BIBLIOTHECA OPALICA is divided into many sub- and subsystems. The structuring has no significance for the user as far as the OPAL Compilation System is concerned.

The other important change is the treatment of text, which involves the types **denotation** and **string**. This is detailed in Section 4.3 [Text], page 14.

BIBLIOTHECA OPALICA is highly, but not fully compatible with the former library. Code which was written using the old library has to be adapted. We distinguish two kinds of adaptations:

- Relocations involve only **IMPORT** statements. You will have to add some **IMPORT** statements which reflect the finer structuring of BIBLIOTHECA OPALICA.
- Incompatibilities require change of code in function declarations and definitions.

We will first specify the incompatibilities and then list old library structures with the changes they each require. The treatment of text is considered in a separate section.

C.1 Incompatibilities

C.1.1 Bitset

The name of the sort has changed from **bitset** to **set**. The function **maxBit** no longer exists, since these sets are no longer bounded.

C.1.2 Conversion

The **format** and **scan** functions from **Conversion** (which have been relocated) formerly interpreted the backslash combinations (**\n**, **\x41**, etc.) themselves, even in strings. Since denotation constants are interpreted differently now, this feature has been removed.

The functions **! : basictype -> string** (where *basictype* is one of **bool**, **char**, **nat**, **int**, **real**) have been replaced by **‘ : basictype -> denotation** in structure *BasictypeConv* (see Section 2.1 [Naming of Structures], page 2). So you need to change all applications of these functions, e.g. **5!** to **!(5‘)** (where the first **!** has origin **String**) in order to get an equivalent expression. For treatment of text, see also Section C.3 [Treatment of Text], page 20.

C.1.3 Constants of type denotation

Denotation constants are now interpreted in a C-like way, i.e. combinations like **\n**, **\x41**, etc. are now interpreted as a single character. The only exception is ****, which is interpreted as end of text. To denote a single ****, write ****.

C.1.4 Natural Numbers

The constant `max'Nat` was raised and is no longer convertible to an object of type `int`. Strictly speaking, this is not an incompatibility, since this was never guaranteed, but since quite a few people seem to have relied on this fact we record it here.

If you assumed `asInt(max'Nat)` to be defined, you will get a runtime error message which says `asInt'NatConv: natural too large`. A quick remedy is to replace occurrences of `max'Nat` by `asNat(max'Int)`, but this is a little bit awkward. A better (but more time consuming) solution is to check whether your program needs either objects of type `int` or of type `nat` and to change it as required.

C.1.5 Section

The structure `Section` no longer exists. Use lambda expressions instead of the functions which were supplied by `Section`.

C.1.6 Suspend

The structure `Suspend` no longer exists. Use lambda expressions instead of the functions which were supplied by `Suspend`.

C.2 List of Old Structures

This section contains a list of those structures from the old library which were modified, and thus necessitate changes in your program.

Structures which are "split into" no longer exist. Structures which are marked "split off" still exist, but some functions now have to be imported from different structures.

Functions which are marked "old fashioned" in BIBLIOTHECA OPALICA are marked with (+) in the following list.

ArrayMapReduce

split into `ArrayMap` with function `*`, and `ArrayReduce` with function `/`.

Bag split off `BagFilter` with function `|`, and `BagConv` with functions `explode(+)`, `implode(+)`, `explodeCt(+)`, `implodeCt(+)`.

Bitset sort `bitset` renamed `set`, function `maxBit` no longer exists.

Char split off `CharConv` with function `ord(+)`, and `NatConv` with function `chr(+)`.

ComData renamed `BinStream`.

ComEnv structure is renamed `Env`, function `random` relocated to `Random`, function `localtime` will be relocated to structure `Time`.

ComProcess

has been reimplemented differently. See structures `Process` and `Pipe` for adapting your code.

ComSocket

will be renamed `Socket`.

| | |
|-----------------------|---|
| ComStream | renamed Stream . |
| Conversion | split into NatConv with function <code>! : nat -> string</code> renamed <code>'NatConv: nat -> denotation</code> , (similar for type <code>int</code> , <code>bool</code> , <code>real</code> and <code>char</code>); into BTUnion with type <code>union</code> and function <code>u</code> ; into StringFormat with function <code>format</code> and StringScan with function <code>scan</code> . |
| Int | split off IntConv with function <code>asNat</code> , and NatConv with function <code>asInt</code> . |
| Map | split off MapConv with function <code>explode(+)</code> and <code>implode(+)</code> . |
| MapMapReduce | split into MapMap with function <code>*</code> , and MapReduce with function <code>/</code> . |
| PrintableChar | function <code>_</code> renamed <code>underscore</code> . |
| Real | split off RealConv with function <code>trunc(+)</code> , and NatConv with function <code>asReal</code> . |
| Section | no longer exists. |
| Seq | split off SeqFilter with functions <code> </code> , <code>partition</code> , <code>take</code> , <code>drop</code> , <code>split</code> , and SeqIndex with functions <code>!</code> , <code>slice</code> . |
| SeqFun | split into SeqCompare with type <code>rel</code> and functions <code><</code> , <code>=</code> , <code>></code> , <code><?</code> , <code>=?</code> , <code>>?</code> , <code>cmp</code> and <code>eq?</code> , into SeqOfSeq with function <code>flat</code> , and SeqSort with functions <code>msort</code> and <code>merge</code> . |
| SeqMapReduce | split into SeqMap with function <code>*</code> , and SeqReduce with functions <code>/</code> , <code>\</code> . |
| Set | split off SetConv with functions <code>explode(+)</code> and <code>implode(+)</code> , and SetFilter with functions <code> </code> and <code>partition</code> . |
| SocketInterNet | will be renamed Internet . |
| String | split off StringConv with functions <code>implode(+)</code> and <code>explode(+)</code> , and StringIndex with functions <code>!</code> , <code>slice</code> , <code>insert</code> , <code>delete</code> and <code>:=</code> . |
| Suspend | no longer exists. |

C.3 Treatment of Text

One of the novelties in BIBLIOTHECA OPALICA is the different treatment of the types `string` and `denotation`. We will discuss here why we made this change and advise which type to choose in which circumstances.

The situation in the old library was unsatisfactory. First you had the type `denotation`, which was almost unusable, since no functions operated on denotations (well, one did). Then there was the `string` type which, while it provided a usable interface for simple treatment of text, was not equipped with a free type. So whenever you wrote an "interesting" function on text, you first had to convert the string into a sequence of characters, and, after performing

two expensive conversions, you finally had access to the nice higher-order functions like `map`, `reduce` and the like.

This has changed in BIBLIOTHECA OPALICA.

In principle, the only change you need to make is to adapt the `IMPORT` lines, since all functions on strings have been retained in BIBLIOTHECA OPALICA.

However, you should examine your code, and depending on the usage of text, choose one of the following approaches:

If you have strings that are not changed very often you should consider changing type `string` to type `denotation` and the `IMPORT` line from `IMPORT String ONLY ...` to `IMPORT Denotation ONLY ...`.

The other possibility is that you are working on the text, and will therefore perform an `explode` on the text before you really start applying functions. In this case you should replace type `seq[char]` with type `string` and remove the `explode` and `implode` function calls. You will now have to import the functions which you used on sequences from the corresponding string structures.

Appendix D Designing Structures For Your Own Data Types

When designing structures for your own data types, you should follow the principles used for structuring the library itself (see Section 2.1 [Naming of Structures], page 2).

First have a look at the library and search for the most similar type already available. Perhaps you are missing another arithmetic type, or your new type can be considered as an aggregate type, or you want to give a different implementation for a data type which already exists.

If you find a similar data type, try to copy the interfaces of the corresponding structures as far as reasonable. If you give a different implementation for an existing data type, keep the name of the type.

If you want to define a totally new data type, you should nevertheless try to use names from existing structures which perform a similar task. If you somehow "select" elements, call that function `!`. The concatenation is called `++`, composition is named `o` and no doubt you will find other functions which resemble functions on the new data type.

Always define a function `' : type -> denotation` (appropriately augmented if the type is parameterized). There are first experimental tools, which exploit the fact that for every type `type` a conversion function `'` exists in `TypeConv`.

Try to keep interfaces small. Give a base structure which contains the most necessary function declarations. Group additional functions into separate structures.

And do not forget to send us the product of your efforts if you think it fills a gap in BIBLIOTHECA OPALICA.

Appendix E Efficiency: What "Not For User Purpose" Really Means

There are some functions in BIBLIOTHECA OPALICA which are marked "not for user purpose". The functions in these sections provide access to internal representations; these are needed for the efficient implementation of other functions on the same data type, when the data type is located in another structure.

In principle, you may use these functions as you do other functions and perhaps you gain some additional performance in doing so. You do this at your own risk, however. We do not guarantee anything about the behaviour of functions which are marked "not for user purpose"; moreover these functions are subject to change in later releases of BIBLIOTHECA OPALICA without further notice.

Index of Types

A

agent'ComAgent[result] 11
array[data] 11

B

bag[data, <] 10
bool 6
bstree'BSTree[data, <] 9

C

char 6
childstat'Wait 13
com[data] 11

D

denotation 6
device'FileSystem 12

F

file 12
filemode'FileSystem 12
filestat'FileSystem 12
filetype'FileSystem 12
fission'ProcessCtrl 12
funct[from, to] 7

G

group'UserAndGroup 13
groupid'UserAndGroup 13

H

heap[data, <] 9

I

inode'FileSystem 12
int 6

M

map[dom, <, codom] 10

N

nat 6

O

option[data] 8

P

pair[data1, data2] 8
permission'FileSystem 12
process'ProcessCtrl 12
procstat'ProcessCtrl 12

Q

quad'Quadruple[data1, data2, data3, data4]
..... 8

R

real 6
rel 6

S

sap'ComService[in, out] 11
seq[data] 8
set'Bitset 10
set'Set[data, <] 10
set'SetByBST[data, <] 10
set'SetByInj[data, #] 10
set'SetByPred[data] 10
sigaction'Signal 12
sighandler'Signal 12
sigmask'Signal 12
signal 12
string 9
subrel 6

T

time 12
tree[data] 9
triple[data1, data2, data3] 8

U

union'BTUnion 9
union'Union2[data1, data2] 8
union'Union3[data1, data2, data3] 8
union'Union4[data1, data2, data3, data4] ... 8
user'UserAndGroup 13
userid'UserAndGroup 13

V

void 6

W

wday'Time 12

Table of Contents

| | | |
|-------------------|--|-----------|
| 1 | Introduction | 1 |
| 2 | Structuring Principles | 2 |
| 2.1 | What's in a Name? — Naming of Structures | 2 |
| 2.2 | Orthogonality | 3 |
| 3 | Survey of Structures | 5 |
| 3.1 | Internal | 5 |
| | Strange | 5 |
| 3.2 | BasicTypes | 6 |
| 3.3 | Functions | 6 |
| | General | 7 |
| | Orderings | 7 |
| | Special | 7 |
| 3.4 | AggregateTypes | 7 |
| | ProductLike | 8 |
| | UnionLike | 8 |
| | SeqLike | 8 |
| | TreeLike | 9 |
| | SetLike | 9 |
| | MapLike | 10 |
| 3.5 | System | 11 |
| | Debugging | 11 |
| | Commands | 11 |
| | Streams | 12 |
| | Unix | 12 |
| 4 | Special Features in BIBLIOTHECA OPALICA | 14 |
| 4.1 | Handcoded Structures | 14 |
| 4.2 | Numbers | 14 |
| 4.3 | String vs. Denotation | 14 |
| 4.4 | ABORT | 15 |
| Appendix A | Acknowledgement | 16 |
| Appendix B | Bugs And Flaws | 17 |

| | | |
|-------------------|---|-----------|
| Appendix C | Changes in OPAL programs necessitated by BIBLIOTHECA OPALICA | 18 |
| C.1 | Incompatibilities | 18 |
| C.1.1 | Bitset | 18 |
| C.1.2 | Conversion | 18 |
| C.1.3 | Constants of type denotation | 18 |
| C.1.4 | Natural Numbers | 19 |
| C.1.5 | Section | 19 |
| C.1.6 | Suspend | 19 |
| C.2 | List of Old Structures | 19 |
| C.3 | Treatment of Text | 20 |
| Appendix D | Designing Structures For Your Own Data Types | 22 |
| Appendix E | Efficiency: What "Not For User Purpose" Really Means | 23 |
| | Index of Types | 24 |