Compilation of LOTOS Abstract Data Types

Hubert GARAVEL*

Laboratoire de Génie Informatique
Institut I.M.A.G.
GRENOBLE FRANCE

This article describes an experiment with the compilation of the data part of LOTOS. *Using a pattern-matching compiling algorithm described in [Sch88], a tool named* CÆSAR.ADT *was developed. It enables* LOTOS *abstract types to be translated automatically into corresponding concrete types, implemented in the C language.*

This paper intends to give a fair idea of the usefulness of this algorithm, when it is combined with appropriate data representations. On several case studies taken from OSI descriptions, this paper explains the basic principles of the translation and shows the great quality of the generated code, which is likely to have better performances than other existing approaches.

Introduction

The ISO specification language LOTOS [ISO87] [BB88] has two distinct parts: a control part, based on a process algebra, and a data part, involving algebraic abstract data types [Gut77] [EM85]. Since each part is independent from the other, it is possible to handle them separately.

LOTOS data structures are described by *sorts*, which represent value domains, and *operations*, which are mathematical functions defined on these domains by algebraic *equations*. Value expressions are terms built from variables and operations. Sorts, operations, and equations are grouped in modules called *types*, which can be combined together using multiple inheritance, renaming, parametrization, and actualization.

Most of the tools available for LOTOS handle the data part by dynamically rewriting value expressions according to the equations. The rewriting motor can be either independent from the equations (as it is in HIPPO [Tre87] and SPIDER [Joh88]) or specifically tailored to take advantage of the properties of the equations (for instance in the "LOTOS to C compiler" [ndM88]).

Other implementation techniques are based on code generation for abstract machines. Originally intended for functional languages with pattern-matching features (like HOPE, ML, and Lazy ML) they can also be applied to LOTOS [WB89].

*This work was also supported by I.N.R.I.A. and VERILOG. Author's current address is:
VERILOG Rhône-Alpes
Centre HITELLA
46 avenue Félix Viallet
38031 GRENOBLE cedex
FRANCE
e-mail: hubert@imag.imag.fr | hubert@imag.uucp
tel: +(33) 76 57 45 87 fax: +(33) 76 57 46 95

The algorithm described in [Sch88] allows a completely different approach. Algebraic specifications can be statically compiled into data structures and routines of an algorithmic language, provided some restrictions. The code produced is deterministic: rewrite rules application is fully determined at compile-time. There is neither unification nor backtrack at run-time. The algorithm is general enough to deal with many-sorted algebras and conditional equations.

A tool was built, named CÆSAR.ADT [Bar88], which implements this algorithm. The C language was chosen as the target language for its machine efficiency, wide availability on existing hardware architectures, and standardized interface conventions. Moreover, by using such a well-known language, one can easily estimate the quality of code generation. The code produced by CÆSAR.ADT may serve to test the correctness of type specifications. It may also be integrated into more sophisticated applications, for example simulation and verification tools, such as CÆSAR [Gar89].

This paper is organized as follows. Section 1 defines the theoretical framework of the approach: assumptions and constraints are carefully stated before the principles of the pattern-matching compiling algorithm are explained. Section 2 is more pragmatic and insists rather on practical issues the implementor is confronted with. Remaining sections deal with the problem of data representation through concrete applications. Section 3 presents the general-case solution while sections 4 to 6 investigate three special-case situations that are likely to be implemented with optimal efficiency: enumerations, numerals and tuples.

Examples are taken either from the standard library of LOTOS [ISO87, annex A] or from "real" specifications, e.g., draft descriptions in LOTOS of the OSI transport service [ISO89a] and transport protocol [ISO89b] [LS88]. To understand this article, a minimal knowledge of both LOTOS and C languages is required.

1 Theoretical framework

LOTOS equational specifications are obviously too expressive and powerful to allow efficient compilation: practical restrictions have to be made. The proposed constraints seem reasonable with respect to LOTOS specifiers' actual needs. Furthermore, the user can get rid of certain limitations if the compiler performs appropriate transformations on LOTOS descriptions.

1.1 Orienting the equations

Equations occurring in a LOTOS source text have the following syntactic form (brackets denote optional elements):

$$[C_1, ..., C_m \texttt{ =>}]\ V_1 \texttt{ = } V_2$$

where V_1 and V_2 are value expressions, and $C_1, ..., C_m$ are optional premisses, each premiss C_i being either a boolean expression V_i' or a simple equation of the form $V_i' \texttt{ = } V_i''$.

For compilation purpose, LOTOS data specifications are considered as term rewriting systems instead of equational systems. This means that equations are supposed to be *oriented*: provided that all premisses C_i are satisfied, the term V_1 can be rewritten into V_2. Conversely V_2 can not be rewritten into V_1 unless explicitly specified by another equation.

Attempts to translate automatically systems of equations into systems of rewrite rules would involve the Knuth-Bendix completion algorithm [KB70].

It is necessary that any variable X occurring in V_2, C_1, ..., or C_m also occurs in V_1. When specifiers keep rewrite semantics in mind, this constraint is always fulfilled.

Note that although LOTOS equations may contain universally quantified variables (i.e., declared in a

"**forall**" clause), terms to be rewritten at run-time may not: their variables are always bound to known values.

1.2 Assuming termination

The rewriting system satisfies the *termination* (or *Noetherian*) property if no term can be rewritten indefinitely.

CÆSAR.ADT does not try to verify whether LOTOS algebraic specifications terminate or not. A survey of general sufficient conditions ensuring termination can be found in [Der87].

1.3 Assuming confluence

The rewriting system satisfies the *confluence* (or *Church-Rosser*) property if, when a term V_1 can be rewritten into V_2 and V_3, there exists a term V_4 such that V_2 and V_3 can be rewritten into V_4.

Since CÆSAR.ADT aims at compiling, not verifying, it does not check for confluence (most tools follow the same policy). Surveys of general techniques for proving this property are given in [HO80] and [Klo87].

1.4 Assuming "call-by-value + priority" rewrite strategy

When rewriting a term, several possibilities often exist: *rewrite strategies* are heuristic criteria to remove (or limit) such non-determinism. A number of strategies have been proposed: *call-by-need* [HL79], *lazy evaluation* [PJ87], ... The strategy used in translating LOTOS to C is "*call-by-value+priority*". It is defined by the following rules:

- when several subterms can be rewritten simultaneously, innermost ones are rewritten first.
- when, for a given subterm, several equations declared in the same type apply simultaneously, the first equation (in the LOTOS source text) is chosen.

This strategy is not always sufficient for reducing multiple possibilities to one single selection. In such cases, the compiler will choose one solution which satisfies both constraints (the user should not expect leftmost terms to be rewritten first). Such non-determinism is compatible with traditional rules for evaluating expressions in algorithmic languages; the compiler may also take advantage of it to optimize code generation.

Call-by-value is well-suited for LOTOS application field because data described in OSI-like specifications do not involve infinite objects that would require ad hoc strategies (such as lazy evaluation). Moreover specifiers often have an operational (rather than declarative) view of LOTOS data types; they expect expressions to be evaluated as they are in algorithmic languages: this is call-by-value semantics.

The code produced by Schnoebelen's algorithm is correct with respect to call-by-value+priority strategy. The remaining issue is the conformance of this strategy to the declarative semantics of LOTOS, as it is defined in the standard [ISO87] (keeping in mind that equations are oriented).

If the system satisfies both termination and confluence, all strategies will rewrite a given term into the same form: conjunction of Church-Rosser and Noetherian properties is a sufficient condition for correctness.

If the rewriting of a term (according to call-by-value+priority strategy) never terminates, the corresponding code will enter into infinite recursive calls.

Assuming a decreasing priority between equations may be found questionable. This does not violate the declarative semantics of LOTOS if the rewrite system is confluent. Many tools, nevertheless, assume such a *de facto* order on equations, even if they do not verify confluence; specifiers are often aware of this feature, and make use of it by writing intentionally non-confluent descriptions, such as:

$$X \texttt{ eq } X \texttt{ = true}$$
$$X \texttt{ eq } Y \texttt{ = false}$$

which defines the equality operator "`eq`" as the syntactic identity of algebraic terms. To prevent this, a compiler could very well apply a random permutation on the equation set. However specifiers may have strong reasons for relying on priorities:

- it allows concise (therefore safe) and generic operator definitions, whereas standard LOTOS style is often heavier (compare the above definition of "`eq`" with those given for "`eq`" in types "`TPDUSubsort`" and "`TransportAddress`", sections 4 and 5 respectively).
- it generally leads to more efficient implementations, because equations are simpler. For instance the previous description of "`eq`" compiles in a single equality test, whereas more complicated definitions of "`eq`" produce more complicated code.
- such specifications belong to a more general framework than confluent rewrite systems. The operational semantics remains simple and compatible with the user's intuitive point of view (although declarative semantics for rewrite systems with priorities may be more difficult to define formally).

When specifiers comply to standard LOTOS style, it is quite certain that non-confluent descriptions are nothing but mistakes.

1.5 Linearizing the equations

The pattern-matching compiling algorithm described in [Sch88] requires all equations to be *left-linear* which means that, for each equation:

$$[C_1, ..., C_m \texttt{ =>}]\ V_1 \texttt{ = } V_2$$

the left-side term V_1 does not contain several occurrences of the same variable (this constraint stands only for V_1 and does not apply to premisses C_i even if they have the form $V_i' \texttt{ = } V_i''$).

Most tools require left-linearity when they do not verify termination. Fortunately, it is possible to get rid of this constraint automatically by performing syntactic transformations on equations before applying Schnoebelen's algorithm. The basic idea is to replace non-linear equations by conditional ones. If some variable X has at least two occurrences in V_1, then a new variable X' is created (whose sort is that of X), one of the occurrences of X in V_1 is replaced by X' and the premiss $X\texttt{=}X'$ is added to the equation. For instance:

$$C_1, ..., C_m \texttt{ => } F\ \texttt{(}X, X\texttt{) = } V_0$$

is replaced by:

$$C_1, ..., C_m, X \texttt{ = } X' \texttt{ => } F\ \texttt{(}X, X'\texttt{) = } V_0$$

This must be repeated until all variables occurring in V_1 are pairwise different.

The transformation is correct under previous assumptions. Because of confluence and termination assumptions, any term can be rewritten into a unique *normal form*, i.e., a term (without any variable) that can not be rewritten. In call-by-value strategy context, the comparison of two terms X and X' (as well as "$X \texttt{ = } X'$", where "`=`" is the congruence symbol of LOTOS) can be implemented by syntactic identity of the normal forms of X and X'. Note that this transformation is not necessarily compatible with other rewrite strategies (call-by-need, for example).

1.6 Identifying constructors

The translation algorithm must be successively applied to each (non-formal) sort of a LOTOS specification. For each sort S the set of operations whose result has S for sort is considered. This set must be divided in two classes:

constructors of S: an operation F is a *constructor* if there exists a normal form term without any variable and containing F. This means that some occurrences of F can not be reduced because the semantics of F is not completely defined by the equations.

non-constructors of S: an operation F is a *non-constructor* if all occurrences of F in terms having no variable can be eliminated by rewriting.

For instance, the boolean sort "`Bool`" defined in LOTOS standard library [ISO87, annex A.4] has only "`true`" and "`false`" as constructors. The integer sort "`Nat`" [ISO87, annex A.6.1.1] has "`0`" and "`Succ`" as constructors.

The constructor concept is important for code generation because any expression can be evaluated (i.e., rewritten) to a normal form term containing only constructor operations. It follows that constructors are the basis for data structure representation and implementation. On the other hand, non-constructors are auxiliary operations defined in terms of constructors: they can be implemented as functions operating on data.

Some compiling algorithms [WB89] treat all operations as constructors. Conversely Schnoebelen's algorithm takes as input a set of operations that are supposed to be constructors. This set does not need to be exact, although it must contain all the constructors. The smaller this set is, the better is the generated code.

The current version of the tool CÆSAR.ADT makes no attempt to determine whether an operation is constructor or not. The user has to indicate constructors by setting a *special comment* `(*! constructor *)` in the LOTOS source text, immediately after each constructor declaration place. This constraint is not too severe because the user usually knows (or should know!) the set of constructors. With constructor indication, LOTOS type specifications can also be read and understood more easily. One may wonder what happens if the specifier does not supply CÆSAR.ADT with the right set of constructors:

- if a non-constructor operation F is declared as "`constructor`" in a special comment, then normal form terms resulting from evaluation will contain occurrences of F. The user has to fix the mistake if this result is not expected.
- if a constructor operation F is not declared as "`constructor`", the definition of F can not be total and a run-time error will occur while executing the code generated for F.

A more sophisticated approach would involve automatic constructor characterization, derived from in-depth analysis of the equations. This is theoretically possible [CR87] [Com88]. The general algorithm is probably difficult to implement, but in the particular case where equations are left-linear, the solution can be computed quite simply.

1.7 Removing equations between constructors

An algebraic data type specification can be compiled with the restricted form of Schnoebelen's algorithm [Sch88, § 3] when its equations have the following form:

$$[C_1, ..., C_m \texttt{ =>}]\ F\ [(V_1, ..., V_n)]\ \texttt{=}\ V_0$$

where F is a non-constructor operation (possibly infixed in LOTOS) and $V_1, ..., V_n$ are value expressions containing only constructor operations and variables (occurrences of non-constructors in $V_1, ..., V_n$ are

forbidden). These constraints prevents constructors from being rewritten. In such case they are called *free constructors* and it is said that the specification has no *equation between constructors*.

The following example, taken from the transport service formal description [ISO89a, § 12.3.2], contains equations between constructors. Two sorts are defined: "`TSP`" represents transport requests, and "`TReqHistory`" describes data structures which store histories of requests. For clarity, definition of requests is shortened to the strict minimum:

```
type TransportServicePrimitive is Boolean
   sorts
      TSP
   opns ...
      IsTReq : TSP -> Bool
      _eq_ : TSP, TSP -> Bool
   eqns ...
endtype

type TransportServiceBasicTSPRequestHistory is TransportServicePrimitive, Boolean
   sorts
      TReqHistory
   opns
      NoTReqs : -> TReqHistory
      Append : TSP, TReqHistory -> TReqHistory
      Empty, NonEmpty : TReqHistory -> Bool
      _eq_, _ne_ : TReqHistory, TReqHistory -> Bool
   eqns
      forall t, t1, t2 : TSP, h, h1, h2 : TReqHistory
      ofsort TReqHistory
         not (IsTReq (t))  =>  Append (t, h)  =  h
      ofsort Bool
         Empty (NoTReqs)  =  true;
         Empty (Append (t, h))  =  Empty (h) and not (IsTReq (t));
         NonEmpty (h)  =  not (Empty (h));
         IsTReq (t) => NoTReqs eq Append (t, h)  =  false;
         IsTReq (t) => Append (t, h) eq NoTReqs  =  false;
         IsTReq (t1), IsTReq (t2) =>
            Append (t1, h1) eq Append (t2, h2)  =  (t1 eq t2) and (h1 eq h2);
         h1 ne h2  =  not (h1 eq h2);
endtype
```

Obviously, operations "`NoTReqs`" and "`Append`" are constructors of sort "`TReqHistory`" because the terms "`NoTReqs`" and "`Append (t, NoTReqs)`", when "`IsTReq (t)`" is true, can not be rewritten. But "`Append`" is not a free constructor since it appears on the left part of the first equation. Intuitively, this means that "`Append`" serves two aims: it plays both a data representation role (request histories are isomorphic to LISP structures, given that "`NoTReqs`" and "`Append`" behave as "`nil`" and "`cons`") and an operational role of selective insertion (transport primitives which do not verify "`IsTReq`" are discarded and kept away from the history).

It is possible to transform a type specification in order to remove all equations between constructors [Tho86, § 3]. Given a non-free constructor F, the basic idea is to dissociate both roles of F by using two operators instead of one:

- a new free constructor operation F' is introduced, which is intended to play the data representation role of F.
- F becomes a non-constructor operation, which will only be in charge of the operational role.
- it is also necessary to modify equations, in order to define each non-constructor operation (in-

cluding F) in terms of F' instead of F. More precisely, for each equation of the form:

$$[C_1, ..., C_m \texttt{ =>}]\ F_0\ [(V_1, ..., V_n)]\ \texttt{=}\ V_0$$

where F_0 is an operation that can be equal or not to F, all occurrences of F in value expressions $V_1, ..., V_n$ must be replaced by F'.

- eventually, a new equation must be added, which indicates that F has to be replaced by F' in all cases where its semantics is left undefined by the original equations. This is achieved by introducing the following equation:

  $$F\ [(X_1, ..., X_n)]\ \texttt{=}\ F'\ [(X_1, ..., X_n)]$$

  (where $X_1, ..., X_n$ are free variables declared by a "**forall**" quantifier) which must be put after all equations of the form:

  $$[C_1, ..., C_m \texttt{ =>}]\ F\ [(V_1, ..., V_n)]\ \texttt{=}\ V_0$$

  according to decreasing priority between equations. It could be possible (but rather complex) to avoid the use of priority between equations by adding adequate premisses to the new equation.

This transformation is correct with respect to orientation, confluence, termination, and call-by-value+priority assumptions.

Applied to the former example, it leads to the dichotomy of the non-free constructor "`Append`" into a free constructor "`App`" and a non-constructor "`Append`". The resulting type specification is:

```
type TransportServiceBasicTSPRequestHistory is TransportServicePrimitive, Boolean
   sorts
      TReqHistory
   opns
      NoTReqs : -> TReqHistory
      App : TSP, TReqHistory -> TReqHistory
      Append : TSP, TReqHistory -> TReqHistory
      Empty, NonEmpty : TReqHistory -> Bool
      _eq_, _ne_ : TReqHistory, TReqHistory -> Bool
   eqns
      forall t, t1, t2 : TSP, h, h1, h2 : TReqHistory
      ofsort TReqHistory
         not (IsTReq (t))  =>  Append (t, h)  =  h;
         Append (t, h)  =  App (t, h);
      ofsort Bool
         Empty (NoTReqs)  =  true;
         Empty (App (t, h))  =  Empty (h) and not (IsTReq (t));
         NonEmpty (h)  =  not (Empty (h));
         IsTReq (t) => NoTReqs eq App (t, h)  =  false;
         IsTReq (t) => App (t, h) eq NoTReqs  =  false;
         IsTReq (t1), IsTReq (t2) =>
            App (t1, h1) eq App (t2, h2)  =  (t1 eq t2) and (h1 eq h2);
         h1 ne h2  =  not (h1 eq h2);
endtype
```

The current version of CÆSAR.ADT does not allow non-free constructors, since it only implements the restricted form of the pattern-matching compiling algorithm [Sch88, § 3]. The user has to remove equations between constructors by hand or, better, not write them.

Equations between constructors could be handled automatically, either by applying the above transformation before the restricted form of the algorithm (another technique for removing constructors can also be found in [Com89]), or by implementing the enhanced version of the algorithm [Sch88, § 7]. As a matter of fact, both methods give identical results.

Whether non-free constructors should be allowed or not is still an open debate:

- detractors argue that equations between constructors are not necessary and should be avoided since they bring confusion between data and operations. They prone *constructor discipline* [GH78] which is, in fact, an "imperative" approach to specification: data structures are defined first by the means of constructors; then other operations are defined by induction on the set of constructors. It is true that this methodology provides suitable guidelines that, in most cases, prevent the specifier from non-confluence, non-termination and non-completeness. Furthermore it allows automatic verification (see in [GH78], for instance, a sufficient condition ensuring that non-constructor operations are totally defined).

- conversely, partisans of equations between constructors praise their expressiveness (they are needed for specifying assertions on data structures, for example). They also emphasize the fact that non-free constructors are better understood than in the past, due to recent advances in algebraic specification theory [Com88] [Com89].

2 Implementation issues

While trying to implement Schnoebelen's algorithm for generating C programs from LOTOS specifications, several problems arise, caused by the existing differences between source and target languages. These issues and the design choices of the tool CÆSAR.ADT are presented here.

2.1 Flattening types

Compilation basically aims at translating each LOTOS sort into a corresponding C type, and each LOTOS operation into a corresponding C function.

When compiling the data part of a LOTOS specification, all process definitions are skipped and the standard *flattening* transformation [ISO87, § 7.3.4] is applied. In fact, things are not so simple; although the modular organization of types should not interfere with the compilation of sorts and operations, it does so, mainly because of C language conventions:

- LOTOS allows the specifier to freely organize type definitions whereas C requires objects to be declared before being used. For this reason LOTOS types must be compiled in "reverse inheritance order" i.e., in a topological order satisfying type dependence constraints. If T_1 and T_2 are LOTOS types such that T_1 inherits from T_2, the C code defining objects of T_2 should precede that of T_1.

- moreover, the translation of LOTOS sorts often results in recursive type declarations. The same problem occurs with operations, leading to mutually recursive functions. To enforce C rules, appropriate mechanisms must be used (CÆSAR.ADT automatically supplies them): forward type and function declarations, use of generic (i.e., "`char *`") pointers and related type conversions.

With respect to these constraints, each sort can be handled in turn by Schnoebelen's algorithm. In fact, the current version of CÆSAR.ADT forbids inheritance of constructor operations: the constructors of a sort have to be declared in the same type as the sort.

No code is generated for sorts and operations defined by renaming: they share the implementation of the sorts and operations they rename.

No attempt is made to deal with types parametrized by formal sorts and operations: to be compiled, types must be fully actualized. This approach leads to larger but more specialized (therefore efficient) code. Production of polymorphic C code is possible, yet difficult: the corresponding amount of work was considered too important for the CÆSAR.ADT project.

2.2 Generating C libraries

The C code produced from a LOTOS description is not a complete program that can be executed by itself. It is rather a library of types and functions, to be used in application programs (probably also written in C). For this reason CÆSAR.ADT generates two source files according to C separate compilation rules:

- **a ".h" file:** this file is an interface which contains types, macro-definitions, and function headers. It has to be included by application programs.
- **a ".c" file:** this file contains function definitions which implement function headers declared in the interface file. It is compiled separately and the resulting object module is linked with object modules generated by separate compilation of application programs.

To improve clarity of examples given below, both "`.h`" and "`.c`" files are merged into a single one.

2.3 Mapping LOTOS identifiers to C ones

To make use, in applications programs, of the libraries produced by CÆSAR.ADT, the user must be aware of the correspondence between LOTOS sort and operation identifiers, and related C type and function identifiers.

It is not always possible to give a C object the same name as the LOTOS object it implements, because LOTOS and C identifiers follow different lexical conventions (LOTOS allows operation identifiers such as "`#`", "`**`", or "`<=>`" that could not be used in C) and because LOTOS permits operation overloading whereas C does not.

To overcome this problem, CÆSAR.ADT requires the user to specify the C name under which a given LOTOS object is to be implemented. This is achieved through the use of special comments (also called *qualifying comments* or *annotations*) which are a meaningful subset of LOTOS comments. When no C name is given, CÆSAR.ADT generates automatically a unique C identifier, which is not necessarily user-friendly.

Each declaration of a LOTOS operation F can be followed by a special comment having the form:

`(*!` **implementedby** N_0 [**constructor**] `*)`

which indicates that, F will be implemented by a C function whose name is N_0. The keyword "**constructor**", when present, states that F is a constructor.

Each declaration of a LOTOS sort S can be followed by a special comment having the form:

`(*!` **implementedby** N_1 **comparedby** N_2 **enumeratedby** N_3 **printedby** N_4`*)`

which means that:

- S will be implemented by a C type whose name is N_1.
- the syntactic equivalence on value expressions of sort S (i.e., the relation denoted by LOTOS keyword "`=`", which may be different from the "`eq`" operation whenever it is defined for S) will be implemented by a C function whose name is N_2.
- CÆSAR.ADT has to generate a C macro-definition called N_3 which allows a C variable of type N_1 to enumerate all values of S exhaustively. This feature is needed for executing LOTOS constructs such as "**choice** $X:S$", "`?`$X:S$", and "**any** S". Of course the domain of S must be finite: this feature is only available for enumerations and (bounded subsets of) numerals. Practically, it is implemented in C as a "**for**" loop. Although the order in which values of S are visited, is determined, application programs must not rely on it.

- CÆSAR.ADT must produce a C function called N_4 that displays value expressions of sort S on a file, in some readable ASCII format.

All C names given in special comments must be pairwise different and also different from every C reserved keyword (including "**main**").

In all examples below, the following naming conventions are used:

- the name of the C object implementing a LOTOS object is the capitalized LOTOS identifier, unless overloading problems occur (for example with "`eq`"), in which case the capitalized name of the sort is appended: "`EQ_BOOL`", "`EQ_TSP`", ... (this will be sufficient here to distinguish overloaded forms of an operator)
- the names used for implementing syntactic equivalence for sorts "`Bool`", "`Nat`", ... are "`CMP_BOOL`", "`CMP_NAT`", ... respectively.
- the names used for implementing enumeration macros for sorts "`Bool`", "`Nat`", ... are "`ENUM_BOOL`", "`ENUM_NAT`", ... respectively.
- the names used for implementing printing functions for sorts "`Bool`", "`Nat`", ... are "`PRINT_BOOL`", "`PRINT_NAT`", ... respectively.

2.4 Choosing between functions and macros

When efficiency is necessary, it can prove useful to translate LOTOS operations into macro-definitions instead of functions. Whether to use macros rather than functions is an implementation choice which is fully independent from pattern-matching compiling algorithm.

Macros are likely to be faster because function call overhead is avoided. But their expansion often leads to larger object code. Furthermore, using macros can also have negative effects on speed, if the arguments are to be evaluated more than once (see the code generated for operators "`xor`" and "`iff`" in section 4); it can even become incorrect if macros are used (in hand-written application programs) with arguments that are not free from side effects.

The current version of CÆSAR.ADT attempts to take the right solution. Nevertheless the tradeoff is not obvious and the user should perhaps be allowed to decide, for instance by using a special comment "`(*! inline *)`".

For conciseness, functions whose bodies contain nothing but a "**return**" statement are replaced by equivalent macro-definitions in examples below.

2.5 Allocating and reclaiming memory

In most cases, concrete implementations of abstract data types make use of dynamic memory. Storage management may be a major issue for large-size applications. Representation of values is often expensive and lots of memory cells created for intermediate results become unreferenced when computations finish.

To overcome this problem, garbage collecting (for memory cells of various sizes) should be used. Several techniques exist [PJ87]: *reference counting* [Knu73, p. 412–413] [Bob80] is quite obsolete; *generation scavenging* [Lie83] [Ung84] seems to be an efficient solution.

Following examples use a rudimentary allocation scheme (using the C function "`malloc`") which does not check for memory exhaustion and does not involve garbage collecting.

3 Compilation of "ordinary" sorts

The first step, when translating LOTOS abstract data types in C, is the implementation of LOTOS value expressions. This part of the problem is not covered in [Sch88], since it is highly dependent of the target language. The solution retained in CÆSAR.ADT is presented here.

The domain of a sort S is the set of normal form terms of sort S. If S has n constructor operations $F_1, ..., F_n$ and if $m_1, ..., m_n$ are the respective arities of these operators, the domain of S is contained in the set Σ of terms V defined by the following syntactical rule:

$$V \;=\; F_1\ [(V_1^1, ..., V_1^{m_1})]\ |\ ...\ |\ F_n\ [(V_n^1, ..., V_n^{m_n})]$$

(where symbol "|" denotes alternative choice). Note that values $V_1^1, ..., V_n^{m_n}$ are not necessarily of sort S, since LOTOS allows many-sorted algebras. The domain of S is equal to Σ if there are neither equations between constructors of S, nor conditional equations. Otherwise the inclusion is strict: Σ contains terms that can be rewritten.

Data representation derives from this property. Terms are implemented by syntax trees, using C structures (i.e., records), unions (i.e., discriminated records) and pointers. A value of sort S is represented by a pointer to a so-called *S-node* which is a union with n fields and a discriminant with n tags. The i^{th} tag is associated with constructor F_i. The i^{th} field of the union is a structure having m_i sub-fields, corresponding to the operands $V_i^1, ..., V_i^{m_i}$ of F_i . The j^{th} sub-field is a pointer to the S_i^j-node implementing V_i^j, where S_i^j is the sort of V_i^j.

This schema could be improved by two optimizations:

- for each F_i such that $m_i = 1$, the i^{th} field of the union is a structure with a single sub-field; it can be replaced by this sub-field.
- for each F_i such that $m_i = 0$, the i^{th} field is an empty structure which can be removed. But a more subtle enhancement also allows the i^{th} tag to be removed: the term F_i can be represented by an "exotic" pointer value (for example "`NULL`", or "`NULL+1`", or "`NULL+2`", ... with appropriate type conversions) different from all pointer values referencing S-nodes. This technique ensures faster data access and manipulation.

Once data representation is determined, the next step is the translation of LOTOS operations into C functions (or macros). Constructors and non-constructors are handled differently.

Each constructor operation of S is implemented by a C function which allocates a new S-node in the heap memory, initializes its tag, fields and sub-fields, and returns a pointer to the node.

Compilation of non-constructors is achieved by Schnoebelen's pattern-matching compiling algorithm. Note that this algorithm is fully independent of the way values are implemented. For each non-constructor F, all equations of the form:

$$[C_1, ..., C_m\ \texttt{=>}]\ F\ [(V_1, ..., V_n)]\ \texttt{=}\ V_0$$

are considered. The body of the C function corresponding to F is recursively generated by induction on the set of these equations defining F. It looks like a decision tree, depending on the values of the formal parameters of F. Whenever equations do not define completely the meaning of an operation, the generated code contains a call to function "`ERROR`" which issues a warning message and terminates the execution.

The following example shows (a simplified version of) the code generated by CÆSAR.ADT for the description given in section 1.7 of request histories (after removal of equations between constructors). The example focuses on sort "`TReqHistory`", the constructors of which are "`NoTReqs`" and "`App`". The implementation of sorts "`Bool`" and "`TSP`" and related operations is not shown here.

```
typedef struct STRUCT {
     enum { DISCR_NOTREQS, DISCR_APP } DISCR;
     union {
          struct {
               TSP  T;
               struct STRUCT *H;
          }    APP;
     }    UNION;
}    *TREQHISTORY;

#define ALLOC_TREQHISTORY(X) (X) = (TREQHISTORY) malloc (sizeof (TREQHISTORY *))

TREQHISTORY NOTREQS () {
     TREQHISTORY  X1;
     ALLOC_TREQHISTORY (X1);
     X1->DISCR = DISCR_NOTREQS;
     return X1;
}

TREQHISTORY APP (X1, X2) TSP  X1; TREQHISTORY  X2; {
     TREQHISTORY  X3;
     ALLOC_TREQHISTORY (X3);
     X3->DISCR = DISCR_APP;
     X3->UNION.APP.T = X1;
     X3->UNION.APP.H = X2;
     return X3;
}

TREQHISTORY APPEND (X1, X2) TSP  X1; TREQHISTORY  X2; {
     if (CMP_BOOL (NOT (ISTREQ (X1)), TRUE)) return X2;
     else return APP (X1, X2);
}

BOOL EMPTY (X1) TREQHISTORY  X1; {
     if (X1->DISCR == DISCR_NOTREQS) return TRUE;
     else return AND (EMPTY (X1->UNION.APP.H), NOT (ISTREQ (X1->UNION.APP.T)));
}

#define NONEMPTY(X) (NOT (EMPTY (X1)))

BOOL EQ_TREQHISTORY (X1, X2) TREQHISTORY X1, X2; {
     if (X1->DISCR == DISCR_NOTREQS)
          if (X2->DISCR == DISCR_APP)
               if (CMP_BOOL (ISTREQ (X2->UNION.APP.T), TRUE)) return FALSE;
               else ERROR ();
          else ERROR ();
     else if (X2->DISCR == DISCR_NOTREQS)
          if (CMP_BOOL (ISTREQ (X1->UNION.APP.T), TRUE)) return FALSE;
          else ERROR ();
     else if (CMP_BOOL (ISTREQ (X1->UNION.APP.T), TRUE) &&
              CMP_BOOL (ISTREQ (X2->UNION.APP.T), TRUE))
               return AND (EQ_TSP (X1->UNION.APP.T, X2->UNION.APP.T),
                           EQ_TREQHISTORY (X1->UNION.APP.T, X2->UNION.APP.T));
          else ERROR ();
}

#define NE_TREQHISTORY(X1,X2) (NOT (EQ_TREQHISTORY (X1, X2)))

int CMP_TREQHISTORY (X1, X2) TREQHISTORY  X1, X2; {
     if (X1->DISCR != X2->DISCR) return 0;
     switch (X1->DISCR) {
     case DISCR_NOTREQS :
          return 1;
```

```
     case DISCR_APP :
          return CMP_TSP (X1->UNION.APP.T, X2->UNION.APP.T) &&
                 CMP_TREQHISTORY (X1->UNION.APP.H, X2->UNION.APP.H);
     }
}

void PRINT_TREQHISTORY (F, X) FILE *F; TREQHISTORY  X; {
     switch (X->DISCR) {
     case DISCR_NOTREQS :
          fprintf (F, "NOTREQS");
          break;
     case DISCR_APP :
          fprintf (F, "APP (");
          PRINT_TSP (F, X->UNION.APP.T);
          fprintf (F, ", ");
          PRINT_TREQHISTORY (F, X->UNION.APP.H);
          fprintf (F, ")");
          break;
     }
}
```

4 Compilation of enumerations

A sort S is an *enumeration* when its constructors $F_1, ..., F_n$ have no operand, i.e., have the following profile:

$$F_1 \texttt{ : -> } S \qquad ... \qquad F_n \texttt{ : -> } S$$

The set of normal form terms of sort S is syntactically characterized by:

$$V \;=\; F_1 \mid ... \mid F_n$$

Consequently S can be translated into an enumerated C type. Each constructor operation of S is implemented by a corresponding enumerated value.

Booleans are the most simple case of enumeration. From the definition of type "`Boolean`" given in the standard library of LOTOS [ISO87, annex A.4] (and annotated by special comments stating that "`true`" and "`false`" are constructors), CÆSAR.ADT produces the C code below:

```
typedef enum { TRUE, FALSE } BOOL;

BOOL NOT (X) BOOL X; {
     if (X == TRUE) return FALSE; else return TRUE;
}

BOOL AND (X1, X2) BOOL X1, X2; {
     if (X2 == TRUE) return X1; else return FALSE;
}

BOOL OR (X1, X2) BOOL X1, X2; {
     if (X2 == TRUE) return TRUE; else return X1;
}

#define XOR(X1,X2) (OR (AND (X1, NOT (X2)), AND (X2, NOT (X1))))

#define IMPLIES(X1,X2) (OR (X2, NOT (X1)))

#define IFF(X1,X2) (AND (IMPLIES (X1, X2), IMPLIES (X2, X1)))
```

```
#define EQ_BOOL(X1,X2) (IFF (X1, X2))

#define NE_BOOL(X1,X2) (XOR (X1, X2))

#define CMP_BOOL(X1,X2) ((X1) == (X2))

#define ENUM_BOOL(X) for((X) = TRUE; (int)(X) <= (int)FALSE; (X) = (BOOL)((int)(X) + 1))

void PRINT_BOOL (F, X) FILE *F; BOOL X; {
     switch (X) {
     case TRUE: fprintf (F, "TRUE"); break;
     case FALSE: fprintf (F, "FALSE"); break;
     }
}
```

Enumerations are not limited to booleans. The following example, taken from the transport protocol description [ISO89a, § 12.3.3.3, p. 10] (see also [LS88, p. 253]), proves that they can be compiled with optimal efficiency. It describes a list of enumerated values which are used to establish a basic classification of TPDUs.

```
type TPDUSubsort is NaturalNumber
   sorts
      TPDUSubsort (*! implementedby TPDUSUBSORT *)
   opns
      CR   (*! implementedby CR constructor *),
      CC   (*! implementedby CC constructor *),
      DR   (*! implementedby DR constructor *),
      DC   (*! implementedby DC constructor *),
      DT   (*! implementedby DT constructor *),
      AK   (*! implementedby AK constructor *),
      ED   (*! implementedby ED constructor *),
      EA   (*! implementedby EA constructor *),
      RJ   (*! implementedby RJ constructor *),
      ER   (*! implementedby ER constructor *) : -> TPDUSubsort
      h    (*! implementedby H *) : TPDUSubsort -> Nat
      _eq_ (*! implementedby EQ_TPDUSUBSORT *),
      _ne_ (*! implementedby NE_TPDUSUBSORT *) : TPDUSubsort, TPDUSubsort -> Bool
   eqns
      forall s, s1 : TPDUSubsort
      ofsort Nat
         h (CR)   =  0;
         h (CC)   =  succ (h (CR));
         h (DR)   =  succ (h (CC));
         h (DC)   =  succ (h (DR));
         h (DT)   =  succ (h (DC));
         h (ED)   =  succ (h (DT));
         h (AK)   =  succ (h (ED));
         h (EA)   =  succ (h (AK));
         h (RJ)   =  succ (h (EA));
         h (ER)   =  succ (h (RJ));
      ofsort Bool
         s eq s1  =  h (s) eq h (s1);
         s ne s1  =  not (s eq s1);
endtype
```

From this type specification, CÆSAR.ADT generates the following C code (the definitions of "CMP_TPDUSUBSORT", "ENUM_TPDUSUBSORT" and "PRINT_TPDUSUBSORT" are not shown, since they are almost the same as for booleans):

```
typedef enum { CR, CC, DR, DC, DT, AK, ED, EA, RJ, ER } TPDUSUBSORT;

NAT H (X) TPDUSUBSORT X; {
     if (X == CR) return ZERO;
     else if (X == CC) return SUCC (H (CR));
     else if (X == DR) return SUCC (H (CC));
     else if (X == DC) return SUCC (H (DR));
     else if (X == DT) return SUCC (H (DC));
     else if (X == ED) return SUCC (H (DT));
     else if (X == AK) return SUCC (H (ED));
     else if (X == EA) return SUCC (H (AK));
     else if (X == RJ) return SUCC (H (EA));
     else return SUCC (H (RJ));
}

#define EQ_TPDUSUBSORT(X1,X2) (EQ_NAT (H (X1), H (X2)))

#define NE_TPDUSUBSORT(X1,X2) (NOT (EQ_TPDUSUBSORT (X1, X2)))
```

5 Compilation of numerals

A sort S is a *numeral* if it only has two constructors F_1 and F_2 the profiles of which are:

$$F_1 \texttt{ : -> } S \qquad F_2 \texttt{ : } S \texttt{ -> } S$$

The set of normal form terms V is defined by the following syntactic rule:

$$V \;=\; F_1 \mid F_2 \texttt{ (} V \texttt{)}$$

It is therefore clear that values of sort S are isomorphic to natural numbers (coded in base 1). They can be represented by unsigned integers, while F_1 and F_2 are respectively implemented by 0 and the function $x \mapsto x + 1$.

This special case obviously applies to natural numbers defined in the standard library of LOTOS [ISO87, annex A.6.1.1]. The type "`BasicNaturalNumber`" has to be completed with special comments telling CÆSAR.ADT that "`0`" and "`Succ`" are constructors and that "`+`", "`*`" and "`**`" must be implemented by C functions called "`PLUS`", "`MULT`" and "`POWER`" respectively.

The bodies of these functions reflect Schnoebelen's algorithm behavior. For each formal parameter y, tests are made to decide whether the form of y is "`0`" or "`Succ (`x`)`", in which case the value of x is obtained as $y - 1$. For obvious reasons, iteration over the domain of "`Nat`" is restricted to the interval 0..255, the bounds of which can easily be modified by the user. Natural numbers are printed using decimal notation, instead of algebraic terms with "`0`" and "`Succ`".

```
typedef unsigned int NAT;

#define ZERO 0

#define SUCC(X) ((X) + 1)

NAT PLUS (X1, X2) NAT X1, X2; {
     if (X2 == ZERO) return X1;
     else return PLUS (SUCC (X1), X2 - 1);
}
```

```
NAT MULT (X1, X2) NAT X1, X2; {
     if (X2 == ZERO) return ZERO;
     else return PLUS (X1, MULT (X1, X2 - 1));
}

NAT POWER (X1, X2) NAT X1, X2; {
     if (X2 == ZERO) return SUCC (ZERO);
     else return MULT (X1, POWER (X1, X2 - 1));
}

#define CMP_NAT(X1,X2) ((X1) == (X2))

#define ENUM_NAT(X) for (X = 0; X <= 255; X++)

#define PRINT_NAT(F,X) fprintf (F, "%u", X)
```

Numerals different from sort "Nat" can also be found in LOTOS specifications. The following example appears in the description of the transport service [ISO89a, § 8.2] [LS88, p. 251]:

```
type TransportAddress is Boolean
   sorts
      TAddress (*! implementedby TADDRESS *)
   opns
      SomeTAddress    (*! implementedby SOMETADDRESS constructor *) : -> TAddress
      AnotherTAddress (*! implementedby ANOTHERTADDRESS constructor *) : TAddress -> TAddress
      _eq_            (*! implementedby EQ_TADDRESS *),
      _ne_            (*! implementedby NE_TADDRESS *) : TAddress, TAddress -> Bool
   eqns
      forall a, a1 : TAddress
      ofsort Bool
         SomeTAddress eq SomeTAddress =  true;
         SomeTAddress eq AnotherTAddress (A)  =  false;
         AnotherTAddress (A) eq SomeTAddress  =  false;
         AnotherTAddress (A) eq AnotherTAddress (A1)  = A eq A1;
         A ne A1  =  not (A eq A1);
endtype
```

The corresponding C code produced by CÆSAR.ADT is:

```
typedef unsigned int TADDRESS;

#define SOMETADDRESS 0

#define ANOTHERTADDRESS(X) ((X) + 1)

BOOL EQ_TADDRESS (X1, X2) TADDRESS X1, X2; {
     if (X1 == SOMETADDRESS)
          if (X2 == SOMETADDRESS) return TRUE;
          else return FALSE;
     else
          if (X2 == SOMETADDRESS) return FALSE;
          else return EQ_TADDRESS (X1 - 1, X2 - 1);
}

#define NE_TADDRESS(X1,X2) (NOT (EQ_TADDRESS (X1, X2)))
```

6 Compilation of tuples

A sort S is a *tuple* when it has only one constructor F, whose profile is:

$$F \ : \ S_1, ..., S_n \ \texttt{->} \ S$$

The arity n should be greater than zero (otherwise S is an enumeration with a single value) and sorts $S_1, ..., S_n$ should be different from S (otherwise the domain of S would be empty since it would be impossible to write well-typed value expressions of sort S).

A tuple value is implemented by a pointer to structure having n fields, one per formal parameter of F. This can be seen as a degenerate form S-node with no discriminant and a single field. Use of pointers could be avoided: this is a well-known debate arising in most programming languages. The constructor is translated into a C function that dynamically allocates a structure, fills in the fields and returns a pointer to it.

Both optimizations described in section 3 also apply to tuples, especially the second one, which enables sorts with several constructors, but a single constructor of arity greater than zero, to be implemented like tuples. Lists, trees, natural numbers with an additional "undefined" value, ... fall into that range.

A general example of tuple appears in the transport protocol description [ISO89b, § 37.3]. It defines a sort "`ThreeTuple`" which is a tuple composed of three sorts, respectively "`Element1`", "`Element2`" and "`Element3`". These sorts are generic: no assumption can be made about their structure, except the fact that the operators "`eq`" and "`ne`" are defined for them. Three selector operations "`First`", "`Second`" and "`Third`" are also defined to extract the tuple components.

```
type ThreeTuple is Element1, Element2, Element3, Boolean
   sorts
      ThreeTuple (*! implementedby THREETUPLE comparedby CMP_THREETUPLE printedby PRINT_THREETUPLE *)
   opns
      Tuple  (*! implementedby TUPLE constructor *) : Element1, Element2, Element3 -> ThreeTuple
      First  (*! implementedby FIRST *) : ThreeTuple -> Element1
      Second (*! implementedby SECOND *) : ThreeTuple -> Element2
      Third  (*! implementedby THIRD *) : ThreeTuple -> Element3
      _eq_   (*! implementedby EQ_TUPLE *),
      _ne_   (*! implementedby NE_TUPLE *) : ThreeTuple, ThreeTuple -> Bool
   eqns
      forall x1, y1 : Element1, x2, y2 : Element2, x3, y3 : Element3, t1, t2 : ThreeTuple
      ofsort Element1
         First (Tuple (x1, x2, x3)) = x1;
      ofsort Element2
         Second (Tuple (x1, x2, x3)) = x2;
      ofsort Element3
         Third (Tuple (x1, x2, x3)) = x3;
      ofsort Bool
         Tuple (x1, x2, x3) eq Tuple (x1, x2, x3)  =  true;
         x1 ne y1  =>  Tuple (x1, x2, x3) eq Tuple (y1, y2, y3)  =  false;
         x2 ne y2  =>  Tuple (x1, x2, x3) eq Tuple (y1, y2, y3)  =  false;
         x3 ne y3  =>  Tuple (x1, x2, x3) eq Tuple (y1, y2, y3)  =  false;
         t1 ne t2  =  not (t1 eq t2);
endtype
```

The last equation was added since [ISO89b] does not provide any equation to define operation "`ne`" (this minor omission was detected by CÆSAR.ADT). This type definition can be implemented as follows:

```
typedef struct {
     ELEMENT1 E1;
     ELEMENT2 E2;
     ELEMENT3 E3;
}    *THREETUPLE;

#define ALLOC_THREETUPLE(X) (X) = (THREETUPLE) malloc (sizeof (THREETUPLE *))

THREETUPLE TUPLE (X1, X2, X3) ELEMENT1 X1; ELEMENT2 X2; ELEMENT3 X3; {
     THREETUPLE X4;
     ALLOC_THREETUPLE (X4);
     X4->E1 = X1;
     X4->E2 = X2;
     X4->E3 = X3;
     return X4;
}

#define FIRST(X) ((X)->E1)

#define SECOND(X) ((X)->E2)

#define THIRD(X) ((X)->E3)

BOOL EQ_TUPLE (X1, X2) THREETUPLE X1, X2; {
     if (CMP_ELEMENT3 (X1->E3, X2->E3) &&
         CMP_ELEMENT2 (X1->E2, X2->E2) &&
         CMP_ELEMENT1 (X1->E1, X2->E1)) return TRUE;
     else if (CMP_BOOL (NE_ELEMENT1 (X1->E1, X2->E1), TRUE)) return FALSE;
     else if (CMP_BOOL (NE_ELEMENT2 (X1->E2, X2->E2), TRUE)) return FALSE;
     else if (CMP_BOOL (NE_ELEMENT3 (X1->E3, X2->E3), TRUE)) return FALSE;
     else ERROR ();
}

#define NE_TUPLE(X1,X2) (NOT (EQ_TUPLE (X1, X2)))

#define CMP_THREETUPLE(X1,X2) (CMP_ELEMENT1 ((X1)->E1, (X2)->E1) && \
                               CMP_ELEMENT2 ((X1)->E2, (X2)->E2) && \
                               CMP_ELEMENT3 ((X1)->E3, (X2)->E3))

void PRINT_THREETUPLE (F, X) FILE *F; THREETUPLE X; {
     fprintf (F, "TUPLE (");
     PRINT_ELEMENT1 (F, X->E1);
     fprintf (F, ", ");
     PRINT_ELEMENT2 (F, X->E2);
     fprintf (F, ", ");
     PRINT_ELEMENT3 (F, X->E3);
     fprintf (F, ")");
}
```

Conclusion

A realistic compilation technique was presented for translating a wide class of LOTOS data descriptions into libraries of C types and functions. It is based on the pattern-matching compiling algorithm described in [Sch88], completed by efficient value representations. It generates good-quality code at high speed (all examples shown in this paper are processed in a few seconds).

The general translation scheme can be refined to handle, with optimal efficiency, some special cases of current use, that can be recognized from a limited analysis of type signatures. Experience confirms this

approach; the following statistics give an idea of the intensive use of enumerations, numerals and tuples in OSI specifications (sorts defined by renaming or actualization are not taken into account since they reflect the sorts they derive from):

	transport service [ISO89a]	transport protocol [ISO89b]
enumerations	4	7
numerals	2	1
tuples	9	3
"ordinary" types	1	4

A prototype tool, CÆSAR.ADT, was developed according to these principles. To simplify the implementation, some pragmatic restrictions were made: the user must indicate constructor operations to the compiler and constructor discipline is enforced. Although the current version of the tool is far from being perfect, it gives a precise idea of what can be achieved.

With respect to other FDTs, the proposed compiling technique obviously applies to SDL whose abstract data types are close to those of LOTOS. It also makes possible automatic translation of LOTOS data specifications into ASN.1 type definitions (in addition to C ones). Only sorts and constructors would be considered in the translation process because ASN.1 describes data but not operations; non-constructors could be dropped since they have no equivalent in ASN.1. Moreover, C functions could be generated automatically to interface internal (i.e., C) and external (i.e., ASN.1) data representations.

Acknowledgements

The work described is this paper was carried out by several persons. The pattern-matching compiling algorithm was initially developed for translating FP2 into LISP [Sch86]. Saddek Bensalem and Jacques Voiron had the idea of applying it to LOTOS. Hubert Garavel designed the general specifications of CÆSAR.ADT and the data representation schemes. Christian Bard implemented the compilation algorithm and developed the tool. Jean-Michel Houdouin brought him help for reusing the front-end part of CÆSAR.

Acknowledgements are due to Christian Bard and to Philippe Schnoebelen for his major contribution to this paper. The author is also grateful to Pippo Scollo who provided him with transport service and protocol descriptions, Daniel Pilaud and Claire, Philippe Leblanc, Xavier Nicollin, Florence Maraninchi, Carlos Rodriguez and the anonymous referees for their judicious comments.

References

[Bar88] Christian Bard. *CÆSAR.ADT 1.0 Reference Manual*. Laboratoire de Génie Informatique — Institut IMAG, Grenoble, August 1988.

[BB88] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–29, January 1988.

[Bob80] Daniel G. Bobrow. Managing Reentrant Structures Using Reference Counts. *ACM Transactions on Programming Languages and Systems*, 2(3):269–273, July 1980.

[Com88] Hubert Comon. *An Effective Method for Handling Initial Algebras*. In J. Grabowski, P. Lescanne, and W. Wechler, editors, *Proceedings of the 1st International Conference on Algebraic and Logic Programming*, volume 343 of *Lecture Notes in Computer Science*, pages 108–118. Springer-Verlag, Berlin, November 1988.

[Com89] Hubert Comon. *Inductive Proofs by Specifications Transformation*. In N. Dershowitz, editor, *Proceedings of the 3rd International Conference on Rewriting Techniques and Applications*, volume 355 of *Lecture Notes in Computer Science*, pages 76–91. Springer-Verlag, Berlin, April 1989.

[CR87] H. Comon and J.L. Remy. How to Characterize the Language of Ground Normal Forms. Rapport de recherche 676, I.N.R.I.A., Roquencourt, June 1987.

[Der87] N. Dershowitz. Termination of Rewriting. *Journal of Symbolic Computation*, 3(1):69–115, February 1987.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1 — Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1985.

[Gar89] Hubert Garavel. *Compilation et vérification de programmes LOTOS*. Thèse de Doctorat, Université Joseph Fourier (Grenoble), November 1989.

[GH78] J. V. Guttag and J. J. Horning. The Algebraic Specification of Abstract Data Types. *Acta Informatica*, 10:27–52, 1978.

[Gut77] J. Guttag. Abstract Data Types and the Development of Data Structures. *Communications of the ACM*, 20(6):396–404, June 1977.

[HL79] G. Huet and J. J. Lévy. Call by Need Computations in Non-Ambiguous Linear Term Rewriting Systems. Rapport de recherche 359, I.N.R.I.A., Rocquencourt, 1979.

[HO80] G. Huet and D. C. Oppen. *Equations and Rewrite Rules: a Survey*. In R. Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 349–405. Academic Press, 1980.

[ISO87] ISO. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. Draft International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Geneva, July 1987.

[ISO89a] ISO/IEC JTC 1/SC 6/WG 4N. Formal Description of ISO/IS 8072 in LOTOS (Transport Service Definition). PTDR Draft Revised Text 10023, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, March 1989.

[ISO89b] ISO/IEC JTC 1/SC 6/WG 4N. Formal Description of ISO/IS 8073 in LOTOS (Connection Oriented Transport Protocol Specification). Working draft, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, March 1989.

[Joh88] Stuart G. Johnston. SPIDER — Service and Protocole Interactive Development Environment. In Kenneth J. Turner, editor, *Proceedings of the 1st International Conference on Formal Description Techniques FORTE'88*, pages 67–71, Amsterdam, September 1988. University of Stirling, Scotland, Elsevier Science Publishers B. V.

[KB70] D. Knuth and P. Bendix. *Simple Word Problem in Universal Algebras*. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, Oxford, 1970.

[Klo87] J. W. Klop. Term Rewriting Systems: a Tutorial. *Bulletin of EATCS*, 32:143–182, June 1987.

[Knu73] Donald E. Knuth. *The Art of Computer Programming — Volume I : Fundamental Algorithms*. Computer Science and Information Processing. Addison-Wesley, Reading, Massachusetts, 1973.

[Lie83] H. Lieberman. A Real-Time Garbage Collector Based on the Lifetimes of Objects. *Communications of the ACM*, June 1983.

[LS88] Jeroen van de Lagemaat and Giuseppe Scollo. On the Use of LOTOS for the Formal Description of a Transport Protocol. In Kenneth J. Turner, editor, *Proceedings of the 1st International Conference on Formal Description Techniques FORTE'88*, pages 247–261, Amsterdam, September 1988. University of Stirling, Scotland, North-Holland.

[ndM88] J. A. Ma nas and T. de Miguel. From LOTOS to C. In Kenneth J. Turner, editor, *Proceedings of the 1st International Conference on Formal Description Techniques FORTE'88*, pages 79–84, Amsterdam, September 1988. University of Stirling, Scotland, North-Holland.

[PJ87] S. L. Peyton-Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, Englewood Cliffs, New Jersey, 1987.

[Sch86] Philippe Schnoebelen. About the Implementation of FP2. Rapport de recherche L.I.F.I.A. 42/I.M.A.G. 574, L.I.F.I.A., Grenoble, January 1986.

[Sch88] Philippe Schnoebelen. Refined Compilation of Pattern-Matching for Functional Languages. *Science of Computer Programming*, 11:133–159, 1988.

[Tho86] Simon J. Thomson. Laws in Miranda. In *Proceedings of the 86 ACM Conference on LISP and Functional Programming, Cambridge, Massachusetts*, 1986.

[Tre87] Jan Tretmans. HIPPO Handout. In *ESPRIT/SEDOS/C3/WP/54/T — Third year project report*, Enschede, December 1987. Universiteit Twente.

[Ung84] D. Ungar. Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. In *SIGSOFT/SIGPLAN Proceedings*, April 1984.

[WB89] Dietmar Wolz and Paul Boehm. Compilation of LOTOS Data Type Specification. In Giuseppe Scollo Ed Brinksma and Chris Vissers, editors, *Proceedings of the 9th IFIP WG 6.1 Meeting on Protocol Specification, Testing and Verification (University of Twente, The Netherlands)*. North Holland, June 1989.