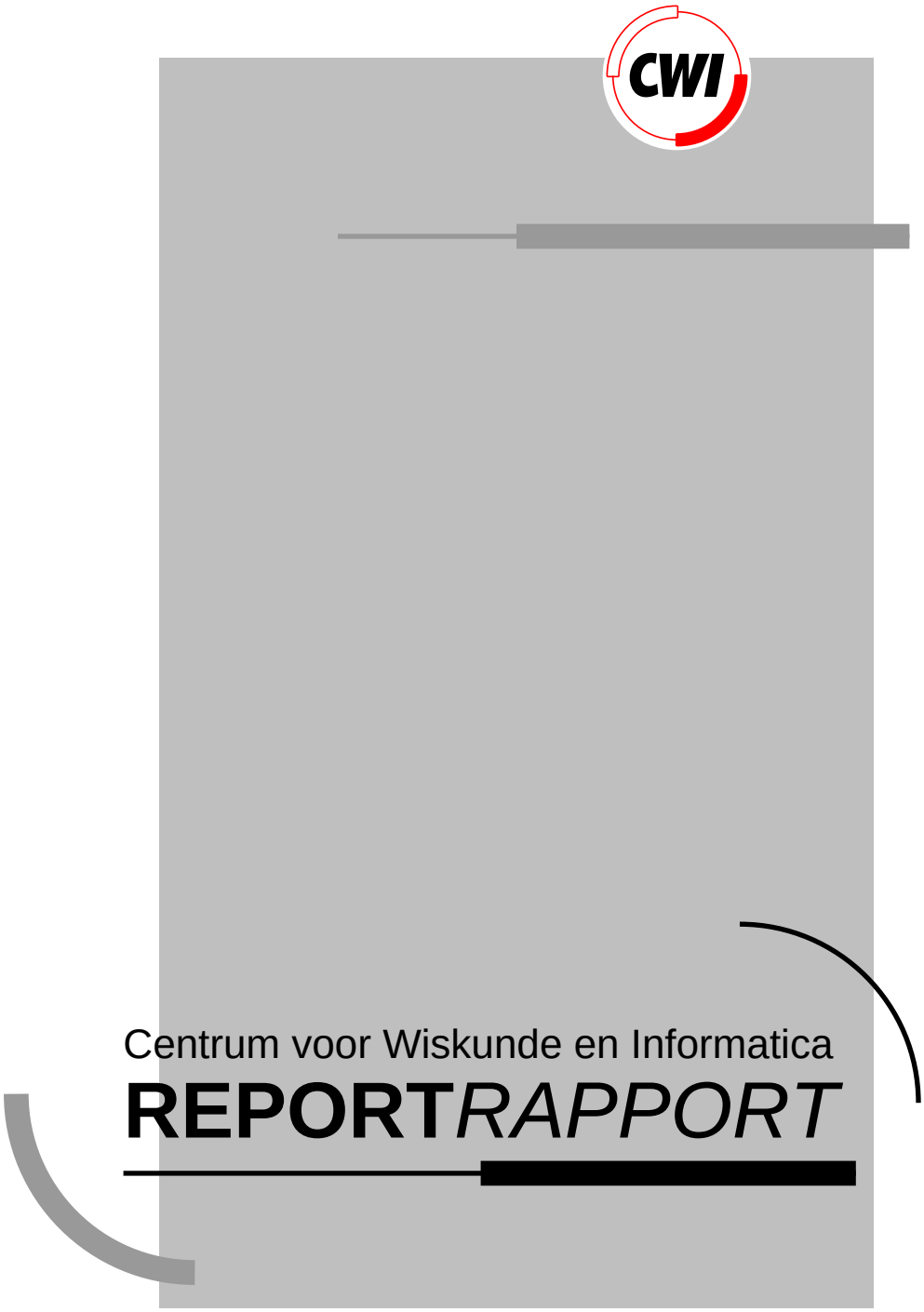

Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

A trying C++ experience (why compare dropped C++)

T.B. Dinesh

Computer Science/Department of Software Technology

CS-R9457 1994

A trying C++ experience
(Why COMPARE dropped C++)

T.B. Dinesh
dinesh@cwi.nl

CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Abstract

COMPARE is an acronym for "Compiler generation for parallel machines". Its main objectives are to build compiler components in such a manner that they can be mixed and matched to develop compilers for new languages and new machines.

It was decided to use C++ in this project, in preference to C, for most of the major components that are to be integrated together. This report summarizes the various opinions *for*-and-*against* favoring C++, the technical problems, and the subsequent standardization problems which eventually resulted in dropping C++ in favor of C. In order to make a reasonably coherent document, we concentrate on issues with respect to mapping the intermediate representation language (fSDL) to C++.

AMS Subject Classification (1991): 68-01, 68N20

CR Subject Classification (1991): D.1.5, D.2.0, D.2.5, D.2.6, D.3.4

Keywords & Phrases: C, C++, Large projects, Compiler generation, Intermediate representation, CoSy, Multiple inheritance.

Note: Support has been received from the European Communities, ESPRIT project 5399: Compiler Generation for Parallel Machines (COMPARE)

Acknowledgements: This document is a consequence of input from members of the COMPARE consortium who were directly or indirectly involved with the C++ issue. Special thanks are due to: Jasper Kamperman and Pum Walters for developing the mapping from fSDL to C++ classes; Helmut Emmelman, Martijn de Lange, Georg Sander and Laurent Voisin for lots of input on their opinion on the merits of C or C++; Tamas Gaal and Georg Sander for providing detailed information on the status of C++ compilers at their sites; Marcel Beemster for investigating the efficiency issue of virtual inheritance; other participants in the discussions on the subject of C++; and for those who contributed by their efforts, in development time spent, using C++.

Table of Contents

1. Introduction

The COMPARE consortium consists of ACE, CWI, Harlequin, INRIA, Steria, UKA, USaar and GMD (till end of 1993). COMPARE is an acronym for "Compiler generation for parallel machines". Its main objectives are to build compiler components in such a manner that they can be mixed and matched to develop compilers for new languages and new machines.

Initially, it was decided to use C++ in this project, in preference to C, for most of the major components that are to be integrated together. This report summarizes the reasons for favoring C++, the technical problems, and the subsequent standardization problems which eventually resulted in dropping C++ in favor of C. This report documents

- a scenario in a large project where many partners, with various interests, have to work together;
- a (recurring) situation of having to choose between C and C++ in the context of large projects;
- use of C++ in practice requires a deep understanding of the various constructs in the language.

In order to make a reasonably coherent document, we concentrate on issues with respect to the development of the fSDL compiler.

2. fSDL and DMCP

fSDL is a language for describing the IR structures that are used in the COMPARE project and is described in the documents [CWI93, WKD94]. See Appendix 1 for an example fSDL description.

fSDL is a language which allows flexible definitions and manipulations of data structures in terms of a calculus of domains. The calculus provides constructive and restrictive operators over domains for their construction and refinement. The process of construction and refinement of domains results in an implicit set of data structures, and domains serve as a definition of a fine grain view of these data structures. fSDL can thus be used to flexibly define a group of data structures and various views of them. When compared to object oriented languages, an fSDL domain could represent a group of *classes*, where arbitrary attributes of them are hidden.

A higher level language like fSDL for defining data structures helps generate utilities, which might be application specific, at a target language level. These may vary from type-checking functionality to generic readers, writers, memory management and debugging routines. In a multi user context, users can extend and/or restrict views of common data structures. Incremental modification of views or data structures by one user need not affect other users' views.

fSDL is designed in the context of the CoSy [AAvS94] compiler model which provides a framework for flexibly combining and embedding compiler phases to facilitate the construction of parallelizing and optimizing compilers. The flat form, ffSDL, can be used by analysis tools in conjunction with engine descriptions and configuration descriptions. A data manipulation and control package (DMCP) specific to a target language is generated which can be used by the engine writers for accessing the pool of data (Common data pool or CDP).

2.1 The compiler

The fSDL compiler, fSDC (short for 'full Structure Definition Compiler'), consists of two major phases, Flatten and Codegen. Flatten transforms the input specification into so-called *flat* form, a sub-language of fSDL that contains only constructive definitions. From the flat form, Codegen produces the actual code for the DMCP.

The flat form provides an interface for the cooperation with other compiler-generation tools. In a number of iterations, these tools may transform the *flat* fSDL specification by adding or deleting domains, operators or fields, and inserting functor applications. Flatten is used to produce a new flat form as result of every iteration.

From the flat form, code is generated to allow the actual use of the DMCP. The current implementation of fSDC only generates C code. Earlier in the project, C++ was chosen as the implementation language. From the point of view of fSDL, C++ has advantages as well as disadvantages when compared to C.

3. Choosing between C and C++

Steria strongly favored C++ as the implementation language for the DMCP. GMD was explicitly in support. ACE liked the technical aspects of C++, but doubted C++'s compliance in practice. Most other partners were not very biased towards one or the other. The reasons stated for certain biases are recorded in this section.

3.1 Arguments for C++: Technical issues

1. **Static type-checking** For a hand-written engine, C++ is the good choice since one wants to have strong static type-checking in order to gain productivity by detecting the most likely errors at compile-time, instead of run-time.

   For a generated engine, the generator has already done all the type-checking before generation, and therefore need not do the type-checking at compile-time. Well, in fact when testing the output of the generator, it can be useful to have strong type-checking, this saves time for hand-written engines. But, once the generator has been stabilised and tested, the type-checking of the implementation language is redundant.

2. **Naming conventions** Using member functions and overloading name spaces become smaller and hence names can become shorter. For example all the member functions of an object have their own name space. In C the names must be globally unique within the whole C program. When implementing with macros, they have to be unique within one compilation unit.

3. **Exploiting Type Information** There are 3 major areas in the DMCP interface where type checking is useful:

   - During access of a field of a node it has to be checked if it is valid. This test has to be done usually dynamically because the operator of a node, an engine variable points to, is often statically not known. However there are many cases where it is indeed statically known, but this is difficult to express.
   - During tree construction it has to be checked if the tree which is constructed is well formed according to the fSDL description, e.g. the condition of a WHILE tree node must be an Expression.

- When calling procedures, or other engines it should be checked that the right kinds of trees are passed, e.g. a procedure calculating the type of an expression expects a tree representing an Expression.

Static type information in the trees can be exploited in three ways:

- Only certain pointers, those of a certain type, in the tree are "virtual" pointers. This makes the use of virtual pointers feasible. They naturally have a certain overhead, but typing allows to restrict this overhead only to fewer pointers.
- Overloading can be exploited. This allows the compiler to decide statically which procedure to call depending on the type of the operands. When carefully used the naming scheme of procedures can be made much easier.
- Automatic type transformations can be specified by the user. A procedure is provided which performs this transformation. This feature can be exploited to perform dynamic type checks in C++, by providing a procedure which does a type cast which in-turn performs the type check.

In fact the C++ language provides some useful implicit type conversions e.g, if we have:

```
class A {...};
class B: public A {...};
```

then a pointer to an object of class B can be implicitly converted to a pointer to an object of class A. But, on the other hand, if we want to consider a pointer to an object of class A as a pointer to an object of class B, then we must use an explicit type cast – as an object of class B is supposed to contain more information than an object of class A.

4. **Hiding the actual implementation** The C++ language, as it is more or less object-oriented, provides means to hide the implementation of a type. Therefore, it ensures more than in the C language, that the code written in the engines will be independent of the actual implementation of the CDP (common data pool) on the host machine.

5. **Explicit or implicit pointers** Explicit pointers mean that the user (the person who writes an engine) knows that the type "Tree" is a pointer to a struct or an object and thus uses the dereferencing operator.

   Having explicit pointers makes it impossible to use (at some time in the future) virtual pointers which, when dereferenced, get the object from another processor or from disk or trigger the calculation of the object. A virtual pointer could be implemented e.g., by a struct, so it should be possible to recompile the engines after redefining the type Tree from a pointer into a struct.

6. **Type conversions** We have three options:

   - Use the C language for implementation, in which case we do not do type conversions (notion of types not very strong).

- Use C++ language with hidden pointers and then implement all the type conversions that are provided by the language for pointers. Note that this approach has two major drawbacks: increased compilation time and code.
- Use the C++ language as it was intended to, that is with explicit pointers.

Therefore, having pointers hidden or not is a strong commitment to make, if we want to use the C++ language. It seems more appropriate to use the pointers as they are provided by the implementation language. *Note: This contradicts earlier item 5.*

7. **Size of code** When using the CDP, there are two approaches for each fSDL view – independent of the implementation language:

   - When debugging or experimenting, we would like to have a rapid compilation process, explicit calls to functions and dynamic type-checking, therefore the code will be large and the executable slow. The compilation can be speeded up by providing only the classes and types of member functions in the .h file, while their actual implementation is hidden in the .C file
   - Once an engine has been tested and stabilised, we want efficient code, and therefore in-lining of functions (or macros in C), so that almost everything is in the .h file.

   The DMCP generator should allow the compiler writer to choose, for each view, which possibility should be followed. For instance, for the CLaX[1] demo, if we use in-line functions, disable the dynamic type checking and use the `-O` option of compiler, the stripped code is only 1.34 Mb.

   So the choice of C++ as the implementation language does not mean necessarily more code.

3.2 Arguments against C++: Managerial issues

From a technical point of view C++ is likable. If one did not have the C-history, the followers syndrome, the market-trends to watch, and some money to make; one would prefer a much cleaner and rather more orthogonal and mature concept.

There are several considerations that keep marketing people busy and insecure about C++:

1. There does not seem to be a buyers market for C++. The amount of requests is limited and the type of prospects are the ones that:

   want to receive your documentation up front

   then, want to come to your offices for two days to do compilation try-outs

   then, want to discuss special features they are used to and would like to see build in (for free)

   then, want to have a three month evaluation copy for free

   then, want a single CPU license even though they have a network of 50 workstations (8.000 ECU each) telling you that 2.000 ECU per CPU is extremely expensive

   then, would like a discount on this copy

   then, tell you that they will only have budget next year so could they have the copy now and pay later

[1] CLaX: COMPARE language example, is a Pascal like language.

then, tell you they have decided to do their own port of GNU compilers because they come for free.

This GNU-effect has killed such compiler manufacturers as Greenhills and is why one should decide to stay away from these markets: such customers are simply too expensive.

2. C++ is neither frozen nor standardized. It is getting close to one specification now but it is not yet there. Therefore, it is not feasible to extend our C front-ends with a *standard* C++ implementation.
   - g++ is not of any commercial help or value
   - cfront 1.2 we have in source but is outdated and not royalty free
   - cfront 2.0 is reasonable but should be purchased again and is not royalty free
   - cfront 3.0 is close to what you want but should be purchased and is not royalty free
3. From a commercial point of view the thing most attractive is to implement what appears to emerge as the standard (almost cfront 3.0) and have a royalty/export free proprietary product, provided there will be a buyers market (which we do expect to grow).
4. Marketing people expect a mature market as of mid 1993 (following year), so the topic is commercially hot today.

Still C++ is a good choice for its engineering aspects and do endorse its usage in COMPARE, but we have to have clear plans on which dialect to use and which compiler to rely on in COMPARE. It might be the case that a C++ front-end engine needs to be developed for the purpose of the COMPARE developments and in order to make the final COMPARE compilers a complete and viable family. In a way we would encourage this development but unfortunately we are afraid that we will not have the resources in the project to achieve this.

Some **Questions** regarding this:

Was a rough estimate ever made of the thousands (millions) of man-hours (ECUs) that are spent world-wide on in-stable (GNU) compiler products? Imagine that these do-it-yourself people would buy commercial products and do the work they really need to do! This would be a significant financial injection for the systems software industry while developers would make significant progress and savings on their real work and for their organization.

What exactly could be used as a core definition of C++, so that one is sure to be independent of actual compiler products and *F*ree *S*oftware *F*oundation *features*? Is there any writeup on this topic from someone with experience in porting software in C++ from one (e.g. cfront) to another (e.g. g++ older versions). It is very important to achieve such C++ *subset* so we keep our options open.

3.3 Arguments against C++: Technical issues

For rapid prototyping, C is preferable as there are more tools and libraries for C than for C++. Sometimes, using C++ results in a slow down of a factor 3 of the programming because one has to reinvent the wheel a second time. C++ programming style is only a subset of full C++ because of personal taste. Normally one should avoid overloading, because programs which heavily use this feature are not readable nor understandable. Class objects are used for event oriented programming, and normal C-style is used for flow oriented programming,

because that is the way the C++ features are intended to be used. Normally only simple class hierarchies are used for flow oriented algorithms, because one cannot concentrate on large complex classes. Type security of C++ is no important reason. If one wants to have more security in C, one can use more checking tools, e.g. lint, ccheck, clash, printfck, or even hand made tools. Think of C++ as a different language, and not as a C with an advanced type system.

- **Explicit or implicit pointers** In C, implicit pointers are implemented by a .h-file like:

```
typedef struct bla_struct { int x; int y; } *bla;

#define BLA_X(p)  ((p)->x)
#define BLA_Y(p)  ((p)->y)
```

  In the program text, always `BLA_X` and `BLA_Y` should be used. If this style of programming is always used, there is no problem to change later to virtual pointers.

  In C++, some pointers are more implicit than others (see Orwell). For 'explicit pointers', the method above for C could be used. For the 'most implicit pointers', this looks like the following:

```
struct bla_struct { int x; int y; };

class bla {
private:
   bla_struct *p;
public:
   int BLA_X();
   int BLA_Y();
   int set_BLA_X();
   int set_BLA_Y();
};
```

  This implies that except from the viewpoint of type checking, there is no advantage or disadvantage concerning C and C++.

- **Exploiting Type Information** The functional language *Miranda* had a very obscure mechanism to transform types automatically. These types were called 'types with algebraic laws'. A type is declared together with laws that transform it into normal form. These laws are automatically applied when possible.

  E.g. rational numbers (not quite Miranda syntax):

```
type rational_number = ( int , int );
   law  (0,b)     =>  (0,0)
   law  (a,b)     =>  let x = gcd(a,b) in (a/x, b/x)
```

Rational numbers are represented by tuples (a,b) and always reduced such that a and b have no common factors.

This feature is now removed from Miranda because of type-theoretical problems. The existence of a normal form cannot be checked statically and is totally unclear when the transformation occurs.

This seems to be similar to the self-programmed type conversions of C++. If a complex algorithm is needed to transform a value from one form into another form, this should be explicit in the program. Otherwise one cannot check the correctness of the program in a simple way, thus it is more a disadvantage than an advantage.

Two points for exploiting type information (in general):

A) Detecting of programming errors.

B) Increasing the efficiency of parameter passing.

If C++ features are reasonably used, C++ is better with respect to A). Reasonable means: use friend classes very seldomly, construct your type hierarchy such that casts are seldom, avoid everything which makes the program unreadable, e.g. overloading or too many implicit algorithms. With respect to B), there are no differences between C and AT&T C++. The compiler simply ignores the additional type information during code generation.

- **Naming conventions** Understandable names are usually not very short. Names need to be unique for one compilation unit, and not just for too small a part, because then one need not concentrate on the context where the name is used, when reading a program. *Okay, i,j,k are always the local induction variables of loops, and h, hh, hhh are always the local help variables. That's it.*

  If one uses overloading and the very small contexts of member functions to produce a lot of identifiers with the same name, then understanding the algorithm is seriously hampered.

- **Size of code** There is a small difference between the type checking of fSDL and the type checking of C++. Of course, C++ is fully type checked in C++.

  The type system of fSDL has two dimensions:

  a) The type hierarchy of the objects and access functions. It is to check if an access function is possible and that the parameters have the correct type.

  b) The view concept. It is to check which nodes and access functions are visible.

  The natural 'view concept' of C and C++ is the use of header-files which specify which functions are available. The disadvantage of member functions is, that they always have to be included with class declarations. Thus, the 'view concept' is orthogonal to the 'class concept'. In the class concept, it is not easily possible to specify which functions of a class are visible and which are not.

  If the 'view concept' is implemented using the 'class concept', this yields duplication of code. This duplication of code can be avoided if an additional layer is introduced with its own naming scheme, i.e. classes are always included with all member functions, but the names of member functions are (artificially) invisible and are made visible by a preprocessor.

- **Type checking** This situation cannot be improved by selecting another language, because of type-theoretical reasons. So far, C++ does what is possible, while C needs dynamic type checking more often.

  If the fSDL specification is in a way that we have many levels of hierarchy, we also need many dynamic checks (this happened in some engines of CLaX). Then the level of static type security is low.

  Doing type checking by a separate type checker running on the C files or running the code through a preprocessor would be easily possible for C, if we knew the rules which to check, i.e. the type calculus. The implementation is absolutely no problem. A simple C-parser already exists, there is enough experience with implementing such type systems. (Theoretically, it is also possible for C++; but currently no split C++ parser). Thus, if someone is able to design a reasonable type calculus which can be applied to C programs and is parameterized with a fSDL specification, this solution would be strongly preferred.

- **Type casts** It is well known that one can simulate dynamic type checking casts by using cast-functions instead of cast-operators in C. This is not a problem. It is also possible to use polymorphic-like functions in C, and this style is more often used (at least in K&R C) than in C++. This is also no problem. But of course, in this case, there is no static type checking anymore. However, the amount of code is reduced.

- **Quotation**

  > C++ makes it more difficult to shoot yourself in the foot, but if you do it, you'll probably loose your whole leg.
  > – Bjarne Stroustrup (author of C++)

4. Technical issues

In this section, we concentrate mainly on the multiple inheritance mechanism of C++.

4.1 Multiple inheritance

Before developing the C++ model of section 5.2, the following multiple inheritance cases were considered as alternatives for generating a C++ model of a given fSDL description of structure relations. Each of these options turned out to be problematic or compromised efficiency too much.

Normal multiple inheritance Normal in the sense that it is not virtual inheritance (also see section 4.1).

This setting (figure 0.1 (a)) makes it hard to cast a D to an A, since it is ambiguous as to which A it should cast in C++. That is, casting up (or widening) is hard. Also, this setting is not desirable since C++ keeps 2 copies of A instance variables in a D object.

Virtual multiple inheritance This setting (figure 0.1 (b)) makes it hard to cast an A to a D since a virtual declaration of A, which makes this setting possible, has implementation bearings that makes casting down (narrowing) hard. This provided the desired model for fSDL to C++ mapping (Section 5.2), not without costing too many dereferences (Section 5.2).

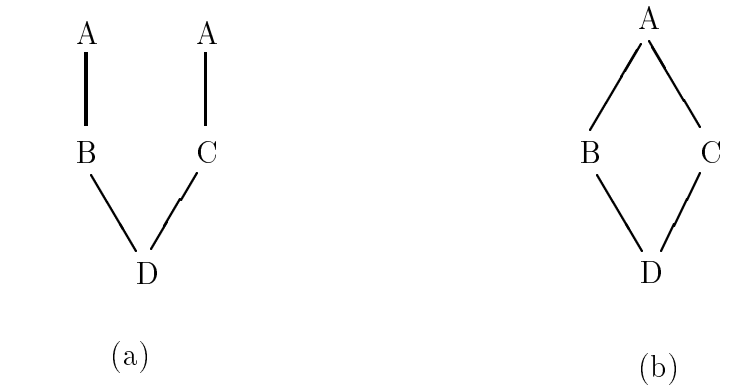

Figure 0.1: (a) Normal inheritance, (b) Virtual inheritance

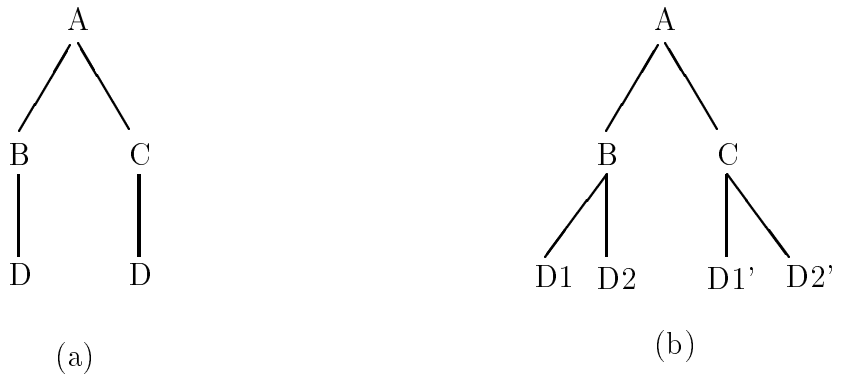

Figure 0.2: (a) Single inheritance, (b) Views with single inheritance

Single inheritance In this setting (figure 0.2 (a)) D level is only operators while A is the sort. The role of sort is that it contains all the shared fields (in any sub domain) while the operators contain only the operator specific fields.

With this respect consider Figure 0.2 (b).

It represents the various "views" of the same operator – D1 and D1' are both potentially the same operators but instead of D1 multiply inheriting from B and C, D1 and D1' now separately inherit from B and C.

This means however that D1 could be eventually be passed (by casting or proper conversion) to where D1' is expected – a conversion which does this is not desirable for efficiency reasons and a cast, if it works, is desired. So for the casts to work the strategy used is that none other than the operators and sorts (bottom layer and top layer – in a many layered tree) will contain instance variables. The reasoning behind this is: The raw pointer casting provided by C++, to move up and down an inheritance chain, is not just a char* (or void*) style conversion but changes the pointer by a delta up or down respectively. If there are no instance variables in a class then the delta is zero. So if a D1 is casted up to B (a delta is lost) and when B is casted to A (delta is zero) implying that when a B is cast to C, still the pointer to raw data

is valid. Now if a C is cast to D1' the same delta (that was lost) is recovered, since D1' and D1 have the same instance variables *in the same order*. However a casting of D1 to D2 or D2', even indirectly is a runtime error which is checked by the conversion operators.

4.2 Problems with method over-loading

Different classes having methods of the same name is not the problem. When the same name is used for two different methods, there are subtle flexibility issues that might come up as a restriction someday.

Let us assume that class A has get_ops() and set_ops(v) both represented by the same name ops. Thus a.ops() gets value and a.ops(v) sets the value to v.

If B inherits from A and B wants to hide the set ops(v).

```
Class B : public A {        or      class B : private A {
private:                            private:  // ...
  ops(V);                           public:
public: //...                         ops() ;
}                                   }
```

will do. But these require new definitions. One way to avoid new definitions is by using A::op in public/private instead. This however will not distinguish the two ops, and thus either hide both or show both but not hide only one.

This is also related, in more subtle, public/private making issues. Therefore, it is better to avoid this.

5. Implementation models

Several implementation models were considered in order to optimize efficiency, static type checking and accurately reflect the input hierarchy. As was envisaged, conversion to a 'full' C++ model proved to be a substantial effort.

Beginning of 1993, the model was announced. The model presented in the section 5.2, had to be reworked to that of section 5.3 since it was potentially inefficient (see 5.2).

The reading of this section requires some knowledge of fSDL, to be able to understand the details of the various models proposed. A reader without the necessary fSDL knowledge can still get an idea of the time and effort spent in developing a suitable mapping of fSDL to C++.

5.1 Mapping fSDL to C++ classes

Initially, a three level hierarchy with dmcp_nodes, sorts and operators was considered, by providing all domain coersions through implicit casts operations.

This was, in general, not received very well. Some wanted the C++ classes to correspond analogously to fSDL domains. This way maximal static type checking would result and the implementation model would correspond directly to domain descriptions. Also, it was required that data fields should be hidden from the user. Thus the access should be done through a hidden pointer. It was agreed that time spent traversing this pointer is time well spent, since the advantages of having this additional dereference included memory management issues as well.

In response to this, the use of PTR class (smart pointer template) to hide the data description was suggested. The general idea was:

```
template<class N> class PTR {
   N* rep;
   /* GC stuff, RTTI stuff, Conversion stuff, etc */
public:
   PTR() { rep = new N; } ;
   PTR(N* r) : rep(r) { } ;
   PTR(N r) { rep = &r; } ;
   ~PTR()  ;

   N* operator->() { return rep; } ;
} ;
```

Although templates could not used for this idea, mostly due to available C++ compiler deficiencies, the general idea was utilized in the final model (Sections 5.2 and 5.3).

5.2 Sort and field hierarchies

An attractive C++ model allows static type checking where possible and inserts dynamic checks where needed. There are no explicit casts needed as long as one stays with in the realm of proper domain calculus. Code duplication is avoided as much as possible.

The model It has taken quite some time to find a satisfying mapping of domains to a C++ inheritance hierarchy. One of the reasons is the limited power of C++. For example, when multiple inheritance is used, we want to use virtual base classes in order to prevent multiple copies of the 'same' instance variable. But then casting to a derived class becomes impossible (see Section 4.1). A second problem is the impossibility to do selective hiding. That is, to hide features at some place, but to have them visible elsewhere, is complicated and ugly.

But even when these problems are worked around, complications remain. The main problem lies in the difference in inheritance of operators and inheritance of fields. Considering shared fields (instance variables of objects), a domain expression

```
A: B + C
```

means that the domain A has the shared fields of both B and C. Naturally, this leads to an inheritance picture as in figure 0.3 (a) (where super classes appear above subclasses).

Figure 0.3: Shared field versus Operator inheritance

But when considering operators, the same domain expression means that an operator of B is also an operator of A, and that an operator of C is also an operator of A. Naturally, this leads to an inheritance picture as in figure 0.3 (b).

Note that the direction is reversed. Consequently, a simple inheritance hierarchy in C++ cannot directly model an fSDL specification.

Therefore the following model:

- For every operator, there is a separate class with instance variables for all private fields of the operator. These classes are not directly accessible to the user.
- For every visible domain D, there is a class Df defining exactly the shared and private fields introduced in D, the latter being delegated to the operator class described above, protected by a dynamic check on the operator. Other accessible fields are inherited from the class where they have been introduced. All these classes form an ordinary C++ (multiple inheritance) hierarchy. At the top of the hierarchy, there is a class Operator, containing the opcode and an indirection to an object containing the private fields. At the bottom are the classes corresponding to sorts. Figure 0.4 describes this pictorially. These classes are also inaccessible to the user.

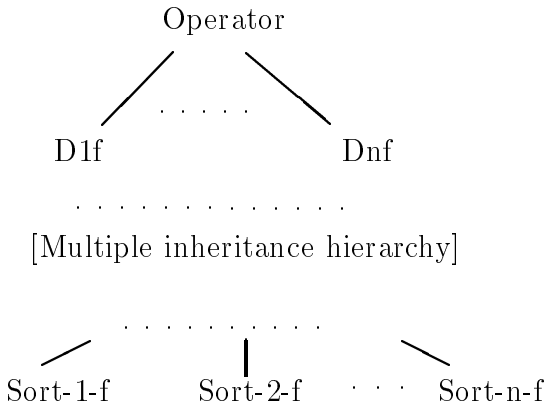

Figure 0.4: Hierarchy diagram

- For every domain D, there is a class D of smart pointers (See section 5.1). The class D is the only class that is known by the user. Every smart pointer is in fact a pointer to an object of Sort-i-f, but the C++ dereference operator `(->())` casts this pointer, to a pointer to an object of Df.

  Using a smart pointer with its dereference operator, a user exactly has access to the fields accessible in the domain D. Whenever two domains have operators in common (this happens only within a sort), an implicit conversion function is generated. When needed, this function does a dynamic check on the opcode.

Design Remarks The private fields of operators are in separate objects reached by an indirection. Removing the indirection and introducing a union will produce faster code, but also more memory consumption.

Life would be much easier if it were possible to maintain a pointer to an object of class Operator in the hierarchy mentioned in figure 0.4, and then cast this pointer down to a pointer to an object of Df. But multiple inheritance forces us to use virtual base classes, and C++ doesn't allow casting down from a virtual base class.

When the operators in D1 are a subset of the operators in D2, the implicit conversion function can be forced into existence by adding an inheritance relation between D1 and D2. In the example, this is done in the relevant places. Using this, and view information, the total number of conversion functions can be kept in check.

Throwing exceptions when a dynamic type check fails would be preferable. Regretfully, in none of our C++ compilers, the exception mechanism is implemented.

Virtual inheritance is expensive The problem arises from the use of virtual classes. Virtual classes cause sharing of inherited classes in a class hierarchy while normal (non-virtual) class inheritance leads to duplication. The problem is that this does not come for free. For an extensive discussion see [ES91, pages 217-237]. The bottom line is that everywhere you write "virtual", a pointer is introduced.

Compiling an example, with AT&T 2.1 C++, indicates that the following code is generated for the class Bf, which seems to be no more than a harmless type declaration:

```
class Bf: public virtual Operator,
          public virtual B1f, public virtual Cf {};
```

But the code that is generated from it looks like this:

```
struct Bf {   /* sizeof Bf == 44 */
struct Cf *PCf;
struct B1f *PB1f;
struct Operator *POperator;
struct Cf OCf;
struct B1f OB1f;
};
```

The size of 44 bytes is not alarming as it is caused mainly by the fields OCf and OB1f. However, the three pointers PCf, PB1f and POperator are alarming. They are introduced by the C++ translator in order to make a type-cast possible from Bf to one of its *virtual* base classes. Had the classes not be virtual, then a static offset computation would suffice, instead of a dynamic pointer lookup as in this case. Note that the need for these pointers seems unavoidable. It is unrealistic to hope that g++ or any other compiler can avoid them.

As a result, every instance of a Bf needs to have these pointers and initialise them. This introduces both space and time overhead.

In order to avoid this unacceptable overhead, it was recommended to avoid the use of virtual classes (in fact virtual anythings, virtual functions have the same problem) in the generated C++ code.

5.3 Fixing sort and field hierarchies

It has become clear that multiple inheritance is virtually unusable in C++. Even multiple inheritance of classes that only define functions, but no instance variables, leads to unexpected overhead (because any class object must have non-zero size).

It should be pointed out that multiple (virtual) inheritance arises as a natural consequence of the request of some partners for a C++ model which reflects the fSDL hierarchy more closely. All occurrences of 'virtual' and all multiple inheritance is removed from the C++ implementation model (for accounting Section 5.2). Thus, meeting the (runtime) time and

space efficiency requirements. Also, the indirection to the operator is removed, which has a beneficial effect on time efficiency. However, the code size has increased and the complexity of the generator has suffered severely. This is due to the fact that in many places C++ inheritance has been removed. In the generator, the inheritance has to be mimicked.

The re-model In the new model, there are three kinds of classes; data classes, access classes and user classes.

1. **Data classes** Data classes are classes where fields are defined. In the inheritance hierarchy for the data classes, there is a class Operator containing only an opcode. For every sort S, class Ss defines all shared fields in the sort. For every operator o there is a class o, that inherits from the class corresponding to its sort (note that the indirection to the operator has become superfluous). Pictorial description is in figure 0.5.

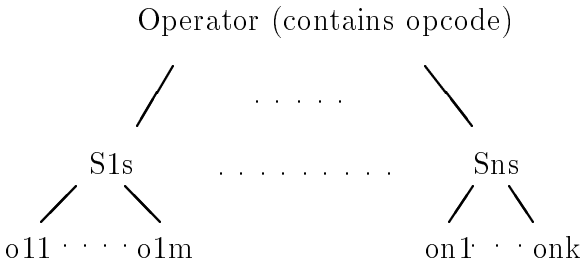

Figure 0.5: Data Classes

2. **The access classes** The access classes are classes used to control accessibility. There is a class REP that contains a pointer to an Operator, and defines a function to obtain the opcode. For each domain, it also defines functions to check if an operator belongs to the domain. For every visible and invisible domain D, there is an 'access' class Df, and a smart pointer class D. An access class defines the access functions available in D. Every access class inherits (directly or indirectly) from REP. The access functions first dereference the pointer in REP, and then access the fields in Ss or oij.

   To reduce superfluous code, an access class may inherit from other access classes but this is limited to single inheritance. For the fSDL example at the end of this text, a possible picture of the access classes is given below. All inheritance relations indicated with dotted lines, in figure 0.6, are implemented by 'sharing' the code in the superclass (that is, each function which the subclass should inherit is implemented by explicitly calling the function in the superclass). Note that fields shared between more than one sort (such as the field `c` introduced in domain C) are simply duplicated. See figure 0.6

   It is unclear whether an algorithm can be found to produce an optimal inheritance hierarchy.

3. **The user classes**

   The user classes are the only classes available to the user (e.g. class D for the domain D). A user class inherits from the corresponding access class (e.g. Df), and it defines automatic conversion functions to all user classes that have operators in common. When

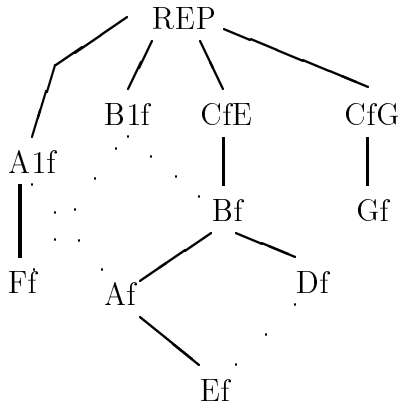

Figure 0.6: Access classes

needed, these functions are dynamically type checked (All dynamic type checking can be turned off in production versions).

In the new model, the automatic conversion functions between user classes are all defined by the generator, and cannot be forced into existence by inheritance.

5.4 Towards implementation of views

It is annoying that we cannot conform to the C++ rule that a class may only be defined in one .h file. Even if this rule is not stated very clearly (formal status of files in C++ is not impressive – `#include` means textual substitution and nothing more), it must be granted that it is bad to have several versions of the 'same' class hanging around.

Due to the fact that our domain classes (with one 'shadow' per view) define only one data member, the current scheme works. It is also hard to imagine why it shouldn't work for future C++ compilers (if linking doesn't change significantly compared to now). However, views whose restrictions necessitate a different implementation of 'fake inheritance' might become quite difficult to implement.

A better model for the implementation of views could be:

- A view is a class, which has local class definitions for all the domains that are visible in the view. Because the domain classes are local to the view class, there is no C++ related objection to have several definitions of a class that implements a domain.
- In this model, an engine class is implemented as a C++ class, that inherits from it's view. The engine functionality must then be defined as member functions of the engine class. These member functions will see the local class definitions for the domains in their view.

This model is much better than the current implementation model, because it adheres to the rules of C++, and it matches the terminology in the CoSy document (where the term 'engine class' is introduced) much more closely than the previous model.

The consequences of this model for the implementation plans of the CoSy group and restrictions in C++ that might hinder implementation of this idea has to be studied.

Below, is a small example of what such views could look like. Please note that there are more possibilities.

```
#include <stream.h>

//this is superclass of everything
class DMCP_node {public: int code; DMCP_node() {code = 0;};};

//A is supposed to be a domain that is in every view
//declaration imported in .h file for every view
class A: public DMCP_node {public:  int a1; int a2;};

//domain B is going to be restricted in View1,
//declaration not imported in .h file for View1
class B: public DMCP_node {public: int b1; int b2;};

//A view is a class with local class definition for domain B
class View1 {
  public:
    //it is no problem to inherit from a class outside View1
    class B: public DMCP_node
       {public: int b1; B() {b1 = 1;};};
    int tryb1() {B myb; return(myb.b1);};
    int tryb2() {B myb; // ``myb.b2'' would give an error
                 return 2; //return a dummy value };
};

//An engine class inherits from its view(s)
class Engine: public View1 {
  public: void dothething() {
            cout << tryb1() << "\n";
            cout << tryb2() << "\n"; };
          Engine() {}; //constructor prevents warning
};

main()
{ Engine myengine;
  myengine.dothething();
}
```

5.5 An expensive bug

Here is a small piece of code that exemplifies the nature of the bug in cfront (AT&T 3.0.1) that dodged successful compilation (a work around) for over 3 months.

```
class Parent {public: Parent() {}};

class Child: public  Parent {
```

```
  public: Child(const Child&){}
};

void f(Child aChild)
{Parent aParent = aChild;}

main(){}
```

6. A typical C++ compiler requirement

As the problems of having incompatible compilers tended to remain unsolvable, partners were asked as to what their requirements would be. This section illustrates what a partner asked oneself in order to provide a suitable requirement.

a) What do we expect from the work of partners wrt C++?

- Anything which works in our environment is okay. Currently we have the following compilers installed: gcc/++ 1.42.3 gcc/++ 2.2.2 gcc/++ 2.3.3 AT&T USL C++ <3.0> AT&T USL C++ <3.0.1>

  If it is C:
  it should be ANSI C. For ANSI C, it is not important whether gcc 1.xxx or gcc 2.xxx or what else is used, because both gcc are quiet stable (for C) and ANSI C is standardized.

  If it is C++:
  g++ 1.42.3 is sufficiently stable, but implements a subset of C++ (e.g. no templates) with a superset of additional features which is not documented anywhere.
  g++ 2.xxx less than 2.2 are not stable enough.
  g++ 2.2.2 still has problems with templates, but everything else is stable. g++ 2.2.2 is okay if templates are not used.
  g++ 2.3.3 is okay.
  AT&T compilers depend on the underlying C compiler. Both of Our versions of AT&T C++ are installed to use Sun cc which is a disadvantage because in-lined C code cannot be ANSI C. We recognized, for AT&T C++, small problems with in-line functions and templates, which are correctly handled by g++ 2.3.3. AT&T C++ <3.0> has some well-known serious bugs that are solved in <3.0.1>.

  We recommend (if it is C++):
  If AT&T 2.1 then it should compile with at least one other C++ compiler that is available for us.
  If g++ 1.42.3 or earlier, or AT&T C++ <3.0> then it should compile with at least one other C++ compiler too, because these compilers may produce executable code (without warning) that is not according to the language C++.
  g++ 2.2.2 is appropriate if no templates are used.
  g++ 2.3.3 or AT&T C++ <3.0.1> are okay in general. Both have small advantages and disadvantages. (One general rule: never use templates and in-line together!)
  If AT&T C++ <3.0> or g++ < 2.3.3>, then the implementor has to be careful to bypass bugs in the compiler.

  It is in principle possible to write portable C++ software, as you may see in the package LEDA. This works with AT&T 2.0, 2.1 AT&T 3.0.1, g++ 1.37, 1.40, 2.1, 2.2.2, 2.3.3. There exists a version using templates and a version without templates.

b) What can be expected from our work wrt C++?

- Our work is based on ANSI C and a very conservative view of C++. K&R C is not used directly, but we can transfer ANSI C automatically to K&R C. Currently, the program analyzer generator is the only tool that produces C++. This will compile with g++ 2.3.3 and AT&T 3.0.1 (templates are used). A switch to ANSI C is possible in principle, but currently not done. The Cosy prototype implementation is in ANSI C completely, and will compile with g++ 2.3.3, too. If it is compiled for X11, it works with ANSI C. The DDA tool, vectorizer, tree parser generator, type check generator, produce real ANSI C that will compile with g++ 2.3.3, too.

7. Calling of C

Several discussions start on various ways to overcome the problems with respect to C++ compilers. Suggestions are made for a task force to find a common COMPARE C++ subset. An inventory is made of what effort is already spent on C++ that would be wasted, on the effort that would be spent in future on the learning aspects of the language C++ and the idiosyncrasy of C++ compilers.

A call is made for votes, on a decision, during a meeting of the technical managers. Portability/maintenance is an issue for the commercial partners, shaky tooling is a problem for all, learning curve is an issue for the commercial partners. An additional consideration is the very low level of inter-operability of C++. Where C functions can quite easily be accessed from almost any other language (including C++), C++ functions have their names mangled and are difficult to access from other languages.

Scenarios formulated were:

1. **C only**
   - – migration
   - \+ portability
   - \+ well-known
   - – C++ engines can interface, but C++ classes cannot be taken as types of attributes.
2. **C++ only**
   - – development/debugging
   - \+ libraries, reuse
   - \+ Type clashing
3. **Map Layer in C++/ Kernel in C** (in addition to Item 1)
   - – expensive now (long term option)
   - – feasibility has to be checked
   - \+ mixed systems C and C++ engines
   - – loss of in-lining
4. **First use C; later convert to C++** (long term option)
   - – 1 language only

– huge effort to convert

Effort estimated for option 1 is an additional 2 man months (when time debugging and design aspects of C++ is deducted) for all the partners to whom this causes a rework to C.

For option 1: 2 abstain, rest in favor. Other options are not seriously considered (during the voting). Partners in favor of C++ request everyone to try to keep their C code compatible with C++.

8. Conclusions

The most notable advantage of C++ is the stronger type system. Apart from stronger type-*checking*, the type system of C++ permits *overloading* of functions, which leads to significantly shorter names, especially for functions generated by functor applications.

Another advantage of C++ is the use of inheritance to specify functionality common to all or a large subset of all domains. In the C variant, this functionality has to be repeated many times with only minor differences. (This is not a strong consideration for generators).

An important disadvantage of C++ is a lack of power in its concepts of inheritance. Especially the use of multiple inheritance, where a subclass inherits from several super classes, can lead to many problems. A natural implementation of multiply inheriting f SDL domain hierarchy requires the use of 'virtual base classes'. However, this leads to implementations of classes which are, unexpectedly, many times larger than what was intended.

Another weak point is the lack of possibilities to selectively hide functions from inheritance. These problems led to an implementation that was effectively much more complicated than an implementation in C.

The most important point, however, is the current state of C++ compiler technology with respect to inter-operability, availability and reliability. The support for overloading seriously hampers inter-operability; at the moment, it is very well possible to incorporate C functionality into a C++ program, but the other way around is near impossible. The differences between different compilers for C++ (AT&T 2.0, gnu 1.40.3, gnu 2.2.2) are rather shocking (even for porting existing C-code). Code is not automatically portable between these compilers (by far). Availability and reliability differ from company to company, but are too low to warrant a dependency of the COMPARE project. Therefore, COMPARE has abandoned C++ in favor of ANSI C.

References

[AAvS94] Martin Alt, Uwe Aßmann, and Hans van Someren. Cosy Compiler Phase Embedding with the CoSy Compiler Model. In Peter A. Fritzson, editor, *Compiler construction, 5th international conference, CC '94*, number 786 in Lecture notes in Computer Science, pages 278–293, Edinburgh, U.K.,, April 1994. Springer-Verlag.

[CWI93] CWI, Amsterdam. *The fSDL user manual*, 1993.

[ES91] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1991.

[WKD94] H.R. Walters, J. Th. Kamperman, and T.B. Dinesh. An extensible language for the generation of parallel data manipulation and control packages. In *Proceedings of the Poster Session of Compiler Construction '94*, April 1994. Appeared as technical report LiTH-IDA-R-94-11, university of Linköping.

1. fSDL: The full structure definition language

The utility of fSDL and brief description of its purpose is given in Section 2.

fSDL can be used to flexibly define a group of data structures (domains) and various views of them. The bare domain calculus is extended with *opaque* and *functor* domains, for the purpose of making the calculus freely extensible to a particular target language. Opaques free fSDL from language specific pre-defined types, whereas functors free fSDL from all pre-defined abstract data types (e.g., lists, graphs, sets, locks) as well as facilitate polymorphic library functionality for a given target language.

An fSDL description of a tiny language Femto [WKD94]:

```
//Stat  ::= asgn Id Exp |  out Exp |  while Exp Stat |
//          if Exp Stat Stat | begin StatList end

domain Stat : { seq <stats: LIST(Stat)>,
                asgn < id: Id, exp: Exp >,
                while < cond: Exp, body: Stat >,
                ifst < cond: Exp, thenp: Stat, elsep: Stat >,
                out < exp: Exp > };

//Exp  ::= Id | Int |
//         plus Exp Exp | minus Exp Exp |  mul Exp Exp

domain Exp : Un + (BinOp-<*>);

//Various intermediate domains are defined for specific uses.
domain BinOp : { plus, minus, mul };
domain BinFld : < right: Exp, left: Exp >;
domain Bin : BinOp + BinFld;
domain Un : { id < name: Id > }
          + { const < val: Int > };

//lexical syntax:      Int ::= [0-9]+
//                     Id  ::= [a-zA-Z]+
opaque Int: h[| typedef int Int; ...  |]
opaque Id:  h[| typedef char* Id; ... |]

// Functor description of the LIST data type used
functor LIST(S)
  begin
    domain LIST: {nil,cons<hd:S,tl:LIST(S)>}
                 + h[| ... |]
  end

domain Symtab : {nil,
                 varval<var:Id, val:Int, next:Symtab>};
```