Algorithm 871: A C/C++ Precompiler for Autogeneration of Multiprecision Programs

WALTER SCHREPPERS and ANNIE CUYT
University of Antwerp

In the past decade a number of libraries for multiprecision floating-point arithmetic have been developed. We describe an easy to use, generic C/C++ transcription program or precompiler for the conversion of C or C++ source code into new code that uses a C++ multiprecision library of choice. The precompiler can convert any type in the input source code to another type in the output source code. The input source can be either C or C++, while the output code generated by the precompiler and using the new types, is C++. The type conversion is based on a simple XML configuration file which is provided by either the developer of the multiprecision library or by the user of the precompiler. The precompiler can also convert to data types with additional features, which are not supported in the types of the source code. Applicability of the precompiler is shown with the successful conversion of large subsets of the GNU Scientific Library and Numerical Recipes.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications; D.3.2 [**Programming Languages**]; K.2 [**IEEE**]: Floating-Point Standard

General Terms: Algorithms, Design, Languages

Additional Key Words and Phrases: Conversion, precompiler, multiprecision, floating-point

ACM Reference Format:

Schreppers, W. and Cuyt, A. 2008. Algorithm 871: A C/C++ precompiler for autogeneration of multiprecision programs. ACM Trans. Math. Softw. 34, 1, Article 5 (January 2008), 20 pages. DOI = 10.1145/1322436.1322441 http://doi.acm.org/ 10.1145/1322436.1322441

1. INTRODUCTION

It is well known that the conversion of source code by hand is error-prone and very time-consuming. In Bailey [1993b, 1993a] a transcription tool for Fortran programs is described which automates the use of the MPFUN [Bailey 1990] library. Since then several other multiprecision libraries have been developed in C++ and Fortran such as MpIeee [Cuyt 2004], MPFR [Zimmermann and the PolKA Project 2004], ARPREC [Bailey et al. 2002], CLN [Haible 1997], MPFUN90 [Bailey 1995] and FMLIB [Smith 1991].

Authors' address: Universiteit Antwerpen, Middelheimlaan 1, B-2020 Antwerpen, Belgium; email: walter@schreppers.com, Annie.Cuyt@ua.ac.be.

Because the use of types in the existing C++ multiprecision libraries is very unalike, a transcription program written specifically for one multiprecision library cannot be reused for another C++ library. We have therefore developed an easy to use, generic transcription program for the automatic conversion of C or C++ source code into new code that uses data types of a C++ library. This C++ library can be any multiprecision library of choice, among others [Cuyt 2004; Zimmermann and the PolKA Project 2004; Haible 1997; Smith 1991; Bailey et al. 2002]. But it can also be a C++ library with data types for big integer or exact rational arithmetic. Our precompiler can, however, not be used for the transcription of Fortran code.

There are many good reasons for using a precompiler instead of, for instance, a `sed` or `awk` script. A precompiler can do variable scoping, it can exclude the conversion of certain variables or functions and it can automate the conversion of output routines like C `stdio printf` instructions into C++ `iostream cout` instructions. Of course a precompiler is not a true compiler or not even a cross-compiler. In some cases, the precompiler identifies a tricky situation and user interaction is needed, for instance when the C functions `calloc`, `malloc`, `realloc`, `free`, and `sizeof` for memory management are used. A more detailed description on how to resolve these warnings is given in Section 3.

When comparing our precompiler to the Fortran transcription program described in Bailey [1993b, 1993a], or to the Simplified Wrapper Interface Generator SWIG [Beazley 1998] developed at Los Alamos, we point out that our precompiler does not use directives inside the source code, while this is necessary in both Bailey [1993b, 1993a] and the SWIG tool. The precompiler uses an XML conversion configuration file in which the conversions from source data types to destination data types are listed. The actual conversion routines are part of the C++ library implementing the destination types. It should be pointed out that the transcription program described in Bailey [1993b, 1993a] for the MPFUN [Bailey 1990] library, was superseded by the MPFUN90 [Bailey 1995] library, which uses operator overloading and custom data types.

Yet when comparing the functionality of the precompiler with basic operator overloading, the advantages of the precompiler stand out. Clearly, operator overloading is not an option for C source code, while conversion from C to C ++ code is part of the functionality of the precompiler. When the source code is written in C++ or Fortran 90 and operator overloading is possible, it still requires a lot of manual work and insight to get it right, especially if the target class does not contain all operators used by the original hardware type. For C++ libraries this is resolved by the precompiler by mapping missing operators to regular class member calls. In such instances a precompiler or transcription tool is the only way to automate the conversion and avoid the introduction of errors by manual conversion. Finally, since the precompiler is able to convert constants into a string representation, it avoids possible rounding and conversion errors which can be introduced when going from hardware precision to multiprecision.

In the next sections we discuss the following topics:

—The internal working of the precompiler

—How to use the precompiler

—The application of the precompiler to transcribe the GNU Scientific Library (`GSL`) [Galassi et al. 2001] and Numerical Recipes (`NR`) [Press et al. 2002] to multiprecision versions. The transcription is done for use with the `MpIeee` multiradix, multiprecision class library [Cuyt 2004].

We hope this paper might also inspire others to develop precompilers targeted at different programming languages, because the applied conversion techniques such as variable scoping and conversion exclusion, are applicable to most programming languages. Although we we only describe the precompiler for automated conversion into the `MpIeee` type in this paper, our tool can be used for any C++ multiprecision library. The ARPREC [Bailey et al. 2002] library provides C++ wrappers to the MPFUN90 datatypes. The precompiler can also automate the conversion from standard `double precision` code to the new ARPREC datatypes by using a different configuration file, which is given in the algorithm submission [Schreppers 2006].

2. INTERNAL OPERATION OF THE PRECOMPILER

2.1 The XML Conversion Configuration File

To configure the precompiler for transcription to code using a specific multiprecision library, either the developer of the multiprecision library or the user of the precompiler needs to provide a configuration file describing the conversion settings. These settings include: identifying the source types and target types for the conversion, conversion rules between the source and target types, and specifying how assignments to the variables in the source code have to be transcribed in the target code. While writing down these settings may seem cumbersome, in many cases a suitable conversion configuration file already exists or an existing conversion file can easily be modified.

The format we have chosen to write down the configuration settings is XML, and we have developed our own lightweight XML parser to generate a n-tree from the conversion configuration file. This complies with the DOM (Document Object Model) of XML [Harold and Means 2001]. Our lightweight XML parser contains the essentials we need, is 400% faster than the parser in the Qt class framework [Dalheimer 2002], and is easier to use than the alternative C++ XML parsers such as expat [Kim 2001]. Most importantly, by writing our own XML parser, we are not bound by any licenses.

In Figure 1 we give the Document Type Definition (DTD) [Bia et al. 2001] for the conversion configuration files. The longer XML Schema [XML Schema 2001] definitions can be found in the `HTML` files bundled with the algorithm submission [Schreppers 2006] in the `Documentation` directory.[1] There are 9 different tags, most of which are illustrated in the example conversion configuration file in Figure 2.

Based on this conversion configuration file, the precompiler transcribes the source code in Example A.1 for use with the `MpIeee` multiprecision library and generates the output code given in Example A.2.

[1]The site is also online at http://www.win.ua.ac.be/u/wschrep/precompiler/paper.html

```
<!ELEMENT include (#PCDATA)>
<!ATTLIST include global (true|false) #IMPLIED>
<!ELEMENT init (#PCDATA)>
<!ELEMENT keyword (#PCDATA)>
<!ELEMENT token (#PCDATA)>
<!ELEMENT operation (#PCDATA)>
<!ELEMENT source (keyword|token)+>
<!ATTLIST source name CDATA #REQUIRED>
<!ELEMENT rhs (keyword|token)>
<!ATTLIST rhs name CDATA #REQUIRED>
<!ELEMENT target (keyword)>
<!ATTLIST target name CDATA #REQUIRED>
<!ELEMENT convert ((source|rhs|target),target,operation?)>
<!ATTLIST convert name CDATA #IMPLIED>
```

Fig. 1. DTD for the conversion configuration file.

In Figure 2, two `source` tags are given: each tag identifies one or more keywords in the source file which will be converted. Keywords listed under the same source tag, such as `float` and `double`, will all be handled in the same way by the precompiler. One can also define `double` in a different `source` tag and convert it to a different type should one require it. In our example we use the multiprecision type `MpIeee` which can extend both `float` and `double` correctly.

The `target` tags similarly give the target type of a conversion. The first conversion rule states that all variables of a type listed in `sfloat` will be converted to variables of the type given in `tfloat`. In other words, all single- and double-precision variables will be converted to variables of type `MpIeee`. To complete the configuration settings, we need to specify how assignments to single or double precision variables must be transcribed in a semantically correct way. For instance, in the source code in Example A.1, the float variable `b` is assigned the value 0.1. With only the first convert rule (named 'float to mpieee'), the precompiler transcribes this to

```
MpIeee b=0.1;
```

This is not really appropriate because C++ compilers, when parsing constants such as 0.1, perform a conversion from decimal to double precision binary representation, not to a multiprecision representation. In the `MpIeee` library, as in many other multiprecision libraries, assignment of constants to multiprecision variables is done by string initialization. In other words, the precompiler should generate

```
MpIeee b=MpIeee("0.1");
```

This statement avoids conversion from decimal to binary by the C++ compiler. To achieve this transcription, we specify two things in the conversion configuration file: which right-hand sides of an assignment need special conversion and what the appropriate conversion rule is. For the former the `rhs` tag is provided, and for the latter we use a `convert` tag with an optional `operation` specifier. As illustrated in the `rhs` tag `afloat` in Figure 2, special conversion is needed whenever in the right-hand side of an assignment, either an integer token or a decimal token is parsed. The meaning of tokens is fixed by the lexical analyzer,

```
<?xml version="1.0"?>
<document>
  <include> BigInt.hh     </include>
  <include> MpIeee.hh     </include>
  <include> ArithmosIO.hh </include>
  <source name="sfloat">
    <keyword> float </keyword>
    <keyword> double </keyword>
  </source>
  <source name="sint"><keyword> int </keyword></source>
  <target name="tint"><keyword> BigInt </keyword></target>
  <target name="tfloat"><keyword> MpIeee </keyword></target>
  <rhs name="afloat">
    <token> integer </token>
    <token> decimal </token>
  </rhs>
  <rhs name="aint"><token> integer </token></rhs>
  <!---------- the conversion rules ------------>
  <convert name="float to mpieee">
    <source name="sfloat"/> <target name="tfloat"/>
  </convert>
  <convert name="int to bigint">
    <source name="sint"/> <target name="tint"/>
  </convert>
  <convert name="decimal to mpieee">
    <rhs name="afloat"/> <target name="tfloat"/>
    <operation> toStringConstructor </operation>
  </convert>
  <convert name="integer to bigint">
    <rhs name="aint"/> <target name="tint"/>
    <operation> toStringConstructor </operation>
  </convert>
</document>
```

Fig. 2. Conversion file for the `MpIeee` library.

a subset of the precompiler. From the full list of tokens given in Table I, it is clear that the integer and decimal token correspond to our intuition. The third conversion rule in Figure 2 specifies the transcription for the right hand side `afloat`. It very much resembles the first two conversion rules, except that it contains an additional `operation` tag. Without the operation tag, the operation is one of simple replacement. With the operation tag, one can specify the appropriate conversion format. Each conversion operation is implemented by a C++ function with the same name. Each such function is a member function of the `ConvertConfig` class of the precompiler. Clearly, for different multiprecision libraries, different transcription operations may be needed. If this is the case, all the user or developer of the library has to do, is add the appropriate C++ member functions to the `ConvertConfig` class of the precompiler. Examples of such operation functions are given in Section 3 and more details about adding new functions can be found in the algorithm documentation [Schreppers 2006].

```
#include <stdio.h>

float func(float& a){
  float k;
  for(int i=1;i<5;i++)
    for(int j=1;j<3;j++){
      char k=32;
      if(i+j>5) k='y';
      if(k=='y') printf("k=%c, i+j=%i\n",k,i+j);
    }
  k=2*a;
  return 3*a;
}

int main(){
  float b=0.1,c=0;
  double d[3][3]=
    {
      1.0, 2.0, 3,
      4, 5, 6,
      7, 8, 9
    };
  c=3+26;
  b=func(c);
  printf("b=%f\n",b);
  printf("d[1][2]=%f\n",d[1][2]);
  return 0;
}
```

Example A.1. C code example (test.cpp).

The only tag which we have not discussed so far is the `init` tag. When this tag is used, the PCDATA[2] inside it will be inserted in the target code as the first few lines of the `main` function. The same functionality can be achieved with a command line option, as discussed in Section 3.1.

2.2 Variable Scoping in the Precompiler

As mentioned in the introduction, the precompiler handles variable scoping, meaning that variables and functions belong to a certain scope and are removed from the lookup table when the scope is closed. This is necessary, for example, when a double and a char with the same name are created. A simple script without scoping will probably convert assignments to both variables (or none at all). In most cases we only want to convert the integer variables and without scoping such a conversion fails.

We outline below the algorithm on which the precompiler is based and which includes variable scoping. We can separate the algorithm in two parts: handling of `tKnown` and handling of `tOther` variables. The `tKnown` variables have one of the types defined in the conversion file to be converted. The `tOther` variables have a type not given in the conversion file or they are variables that are to

[2]PCDATA refers to the literal text between an open and closing XML tag.

```
#include <iostream>
#include <iomanip>
using namespace std;
#include "BigInt.hh"
#include "MpIeee.hh"
#include "ArithmosIO.hh"
#include <stdio.h>
MpIeee func(MpIeee & a){
  MpIeee k;
  for(BigInt i= BigInt( "1" );i<BigInt( "5" );i++)
    for(BigInt j= BigInt( "1" );j<BigInt( "3" );j++){
      char k=32;
      if(i+j>BigInt( "5" )) k='y';
      if(k=='y') {cout<<"k="<<k<<", i+j="<<i+j<<"\n";}
    }
  k=MpIeee( "2" )*a;
  return MpIeee( "3" )*a;
}
int main(){
  MpIeee::fpEnv.setRadix(2); MpIeee::fpEnv.setPrecision(90);
  MpIeee::fpEnv.setExpRange(-126,127); MpIeee::fpEnv.setRound(FP_RN);
  ArithmosIO::setIoMode(ARITHMOS_IO_MPIEEE_DECIMAL|
                        ARITHMOS_IO_RATIONAL_RATIONAL);
  MpIeee b= MpIeee( "0.1" );MpIeee c= MpIeee( "0" );
  MpIeee d[3][3]=
    {
      MpIeee( "1.0" ), MpIeee( "2.0" ), MpIeee( "3" ),
      MpIeee( "4" ), MpIeee( "5" ), MpIeee( "6" ),
      MpIeee( "7" ), MpIeee( "8" ), MpIeee( "9" )
    };
  c=MpIeee( "3" )+MpIeee( "26" );
  b=func(c);
  {cout<<"b="<<setiosflags((ios::fixed & ios::floatfield))<<b;
   cout.precision(6);cout.fill(' ');cout.width(0);
   cout.setf(ios::dec,ios::basefield);
   cout<<resetiosflags((ios::fixed & ios::floatfield))<<"\n";}
  {cout<<"d[1][2]="<<setiosflags((ios::fixed & ios::floatfield))<<d[1][2];
   cout.precision(6);cout.fill(' ');cout.width(0);
   cout.setf(ios::dec,ios::basefield);
   cout<<resetiosflags((ios::fixed & ios::floatfield))<<"\n";}
  return 0;
}
```

Example A.2. C++ code after conversion of Example A.1

be skipped according to the skip configuration file (these are left unchanged by the precompiler).

(1) Read token `T` from source file using lexer. See Table I for an overview of the tokens.
(2) `T` is a sym_word token

Table I. The Lexer Tokens

Name	Regular Expression or Description
comment	a C/C++ comment (meaning /* . . . */ or //. . .)
directives	a C/C++ directive
quoted_string	a quoted string
sym_word	a C/C++ word (keyword or variable/function name) [a-zA-Z_][a-zA-Z_0-9]*
assignment	[+-*/]?[=]
comparison	[<>][=]?|("! = ")|(" == ")
stream_op	(<< | >>)
mult	*
plus	+
minus	−
div	/
double_colon	::
pointer_op	->
bool_op	("&&")|("|")|("||")|("!")
percent	%
open_bracket	[
close_bracket	]
open_par	(
close_par	)
open_brace	{
close_brace	}
semi_colon	;
colon	:
white_space	tabs or spaces
integer	[0–9][0–9]*[fFlL]?
decimal	[0–9]*(".")[0–9][0–9]*([eE][+−]?[0–9][0–9]*)?[fFlL]?
sym_and	&
comma	,
tilde	∼

(a) if `T` is a C/C++ reserved word

i. and if reserved word is `for`: we open a new scope because the running variable(s) of the `for` loop belongs to the scope of that loop. Just after the loop definition we check if an `open_brace` is found. If this is the case, we do not have to open a new scope because this was already done. If it is not found, we have to close the opened scope when finding the next `semi_colon`. The same situation arises with the parameters of a function where we want these to belong to the scope of the function. In case of a function definition we close the scope on the first occurrence of a `semi_colon`, while in the case of a function implementation we close it when we find the matching `close_brace`. Because `for` loops can be nested, we use a separate stack to determine whether to close a scope in the variables stack on a `semi_colon` or on a `close_brace`. The `-nofor` option of the precompiler, discussed later, causes the variables defined in the `for` loop definition to be added as type `tOther`.

ii. and if reserved word is `return`: we look at the return type of the current function and if it is a `tKnown` type we convert the expression after the return as if it were an assignment to a `tKnown` variable.

   iii. in other cases, we copy word literally to output file.
   iv. goto step 1.
  (b) if `T` is a previously defined function or variable (using the functions and variables stack):
   i. if `T` is a variable, we convert any assignments or comparisons to this variable if it is of type `tKnown` and if it is not in the skip variables list (this list is built when the user provides a skip configuration file). `tOther` variables and assignments to them are copied literally.
   ii. If `T` is a function, we just copy `T` literally to the output file.
  (c) if `T` is one of the `tKnown` type declarations we add a `tKnown` variable/function to our variables/functions stack. Again, when it is in the skip variables list, we add it as a `tOther` variable here.
  (d) if `T` is a `tOther` type declaration, add a `tOther` variable/function to the variables/functions stack.
  (e) if `T` is `printf` we use the Printconverter class to convert the `printf` statement into a `cout` statement in the output file.
  (f) in other cases just copy the word literally to output file.
  (g) goto step 1.

(3) `T` is `open_brace`: We open a new scope by pushing a scope element onto our variables stack unless the scope stack suggests otherwise.
(4) `T` is `close_brace`: We close a new scope by popping all elements from our variables stack up to and including the scope element.
(5) `T` is `semi_colon`: It is always copied literally and, as mentioned in 2a(i), it is possible that a scope is closed depending on the contents of the scope stack.
(6) The token `T` is something different: We copy it as is to the output file.
(7) Goto step 1.

Insight into the variable scoping algorithm of the parser is important to understand some of the warnings that are raised by the precompiler. These warnings are discussed in Section 3.2.

2.3 Performance of the Precompiler

Sometimes the conversion rules are such that the precompiler has a choice between multiple conversions. This is, for example, the case when we add conversion rules to the conversion file in Figure 2, as is done in Figure 3.

In Figure 3 there are multiple conversion paths from the `MpIeee` to `Bigint` data type. In Examples B.1, B.2, and B.3 we illustrate how these rules are used. It also shows why conversion rules between target types are needed. To avoid superfluous conversions, the precompiler builds a graph $G = (V, E)$ where the edges in E correspond to the conversion rules and the vertices in V correspond to the known keywords and tokens. The precompiler then uses an unweighted shortest-path algorithm [Weiss 1997] to find a minimal set of conversion rules from a source token or keyword to a target keyword.

Without the shortest path algorithm, superfluous conversions are generated, as is clear from the difference between the two precompiler outputs in Example B.2 and Example B.3 starting from Example B.1. The unweighted shortest-path

```
<?xml version="1.0"?>
  <!---------- the conversion rules ------------>
  ...
  <convert name="mpieee to rational">
    <target name="tfloat"/> <target name="trational"/>
    <operation> toConstructor </operation>
  </convert>
  <convert name="rational to bigint">
    <target name="trational"/> <target name="tint"/>
    <operation> toConstructor </operation>
  </convert>
  <convert name="mpieee to bigint">
    <target name="tfloat"/> <target name="tint"/>
    <operation> toConstructor </operation>
  </convert>
</document>
```

Fig. 3. Additional conversion rules for the MpIeee library.

```
#include <stdio.h>
int main(){
  double b = 2;
  int i    = 2.345;
  return 0;
}
```

Example B.1. Mixed type assignment (test2.cpp).

algorithm that we have implemented is similar to Dijkstra's algorithm for finding weighted shortest paths. The advantage of the unweighted algorithm is a better run time complexity for nondense graphs: $O(|E| + |V|)$ instead of $O(|E| + |V|^2) = O(|V|^2)$ for the weighted shortest-path algorithm. In dense graphs where $|E| = O(|V|^2)$, the weighted shortest-path algorithm is optimal and has the same complexity as the unweighted algorithm [Weiss 1997]. The graphs of our conversion files are not dense and quite small, thus performance is more than adequate.

The output of the standard unweighted shortest-path algorithm is a set of successive vertices $v_1, \ldots, v_n \in V$, where v_1 is the source vertex and v_n is the destination vertex. For the purpose of the transcription, the optimal path needs to be expressed in terms of conversion rules, in other words, as consecutive edges $e_1, \ldots, e_n \in E$. We have therefore slightly modified the algorithm so that it includes labeling of the edges E with unique keys corresponding to conversion rules and then expresses paths in function of edges rather than vertices.

3. USING THE PRECOMPILER

To automatically transcribe C/C++ code to code that uses a C++ library of choice, the precompiler needs a conversion configuration file, of which the syntax and semantics have already been discussed in Section 2.1. In this section, we discuss

```
#include <iostream>
#include <iomanip>
using namespace std;

#include "BigInt.hh"
#include "MpIeee.hh"
#include "Rational.hh"
#include "ArithmosIO.hh"
#include <stdio.h>

int main(){
  MpIeee b=  MpIeee( "2" );
  BigInt i=  BigInt( MpIeee( "2.345" ) );
  return 0;
}
```

Example B.2. C++ conversion of test2.cpp, with shortest-path algorithm used to find minimal set of conversion rules.

```
#include <iostream>
#include <iomanip>
using namespace std;

#include "BigInt.hh"
#include "MpIeee.hh"
#include "Rational.hh"
#include "ArithmosIO.hh"
#include <stdio.h>

int main(){
  MpIeee b=  MpIeee( "2" );
  BigInt i=  BigInt( Rational( MpIeee( "2.345" ) ) );
  return 0;
}
```

Example B.3. C++ conversion of test2.cpp, without shortest-path algorithm.

additional features of the precompiler, which allow for ease of use and for manual tuning of the automatic conversion procedure.

We start off by going over the conversion of the C code in Example A.1, based on the conversion rules in Figure 2 and Figure 3. The conversion rules refer to two conversion functions, `toConstructor` and `toStringConstructor`. As mentioned in Section 2.1, these are implemented as members of the `ConvertConfig` class. The code for these functions is given in Figure 4.

The effect can easily be understood from the generated code in Example A.2. More advanced conversion functions, as needed for `typedef` declarations, can be achieved with different conversion functions. We refer the reader to the algorithm documentation [Schreppers 2006] for more elaborate examples of such conversion functions.

```
...
string
ConvertConfig::toStringConstructor( const ConvertElem& cElem,
                                    const string& value )
{
  return cElem.target.keyword + "( \"" + value + "\" )";
}

string
ConvertConfig::toConstructor( const ConvertElem& cElem,
                              const string& value )
{
  return cElem.target.keyword + "( " + value + " )";
}
...
```

Fig. 4. Conversion functions `toStrintConstructor` and `toConstructor` in `ConvertConfig` class.

The code in Example A.2 is the output code generated by the precompiler.[3] The readability of the generated code is good, except for the conversion of `printf` into `cout` instructions, which would look better if done by hand.

Example A.1 also shows that scoping is indeed crucial for correct conversion. Without variable scoping, assignments to both variables with name `k` would be handled in the same way by the precompiler, resulting in erroneous assignments to the variable `k` of type `char`. Note also that the `return` statement has been converted consistently: according to the conversion file, every `float` type declaration, including the return type of the function, should be converted from `float` into `MpIeee`. And notice how time consuming it would be to convert the initialization of a C matrix definition without the precompiler.

A closer look at the code in Example A.2 shows that the type of all integer variables has been transcribed from `int` to `BigInt`, including the running variables `i` and `j` in the `for`-loops. It is very unlikely that this is as the user intends, and we will indicate below how the precompiler can be told to skip the conversion of certain variables. Also, the first few lines of code in the `main` function in Example A.2 are not the result of transcription. These lines of code initialize the floating-point environment of the multiprecision library `MpIeee`. Including such additional lines of code is also a feature of the precompiler, as discussed below.

3.1 Command Line Options

When calling the precompiler without arguments, the command line options are listed in familiar `*nix` style. The abbreviated output is shown in Figure 5.

The `-preparse` option can be used to get a list of variables that will be converted using the current configuration file. We used this option on the source code shown in Example A.1 to generate the output given in Example A.3. Using the `-constants` option on the source code from Example A.1 will generate the output given in Example A.4. Skipping conversion for specific variables and

[3]Some white space has been removed to make it fit on one page.

```
./precompile

Usage: ./precompile [options] <inputfile> <outputfile>

Options:
  -x <configfile>    specifies the conversion configuration file
  -preparse          preparse rather than precompile to generate a list of all
                     variables and functions that would normally be converted.
  -c <skipfile>      specifies a file containing functions and/or variables
                     which must not be converted

  more options follow ...
```

Fig. 5. Executing precompiler without arguments.

```
func        func        float
func        a           float&
func        k           float
func        i           int
func        j           int
main        b           float
main        c           float
main        d           double
```

Example A.3. Output of the precompiler with the `-preparse` option.

functions can be done with the `-skip` option. Skipping all loop variables created in C++ `for`-loop declarations can be done with the `-nofor` option.[4] By default `printf` instructions are converted into `std::cout` instructions. This can be disabled using the `-noprintf` option. A successful `printf` conversion can be seen when comparing Examples A.1 and A.2.

For other and more advanced options, like adding initialization code to the `main` function, we refer the reader to the documentation supplied in the algorithm submission [Schreppers 2006].

3.2 Warnings

There are a number of instances where manual intervention is needed for correct precompilation. In these instances the precompiler raises a warning.

An important area where user interaction is necessary is when the source code contains calls to the `stdlib.h` C functions `calloc`, `malloc`, `realloc` and `free`. The transcription of these functions will work fine when converting from one basic C/C++ type to another basic C/C++ type (e.g., conversion from `int` into `long int`), but not when conversion into a C++ class is required. The reason is that the constructor of the class will most likely allocate additional memory. Furthermore, when calling `calloc` or `malloc`, often the `sizeof` function is used to compute how much memory needs to be allocated. When using the `sizeof` function for C++ classes the code can crash when executed. For these reasons the precompiler gives a warning when the source code contains calls to `calloc`,

[4]This option is used to optimize loops in precompiled code.

```
<!ELEMENT skip ( (function,variable) | function | variable ) >
<!ATTLIST skip name CDATA #IMPLIED >
<!ELEMENT function >
<!ATTLIST function name CDATA #REQUIRED >
<!ELEMENT variable >
<!ATTLIST variable name CDATA #REQUIRED >
```

Fig. 6. DTD for the skip configuration file.

```
CONSTANTS IN FILE : 'test.cpp'
constant: '1' at line 5, col 13
constant: '5' at line 5, col 17
constant: '1' at line 6, col 15
constant: '3' at line 6, col 19
constant: '32' at line 7, col 14
constant: '5' at line 8, col 14
constant: '2' at line 11, col 5
constant: '3' at line 12, col 10
constant: '0' at line 16, col 11
constant: '0' at line 16, col 15
constant: '3' at line 17, col 12
constant: '3' at line 17, col 15
constant: '1.0' at line 19, col 7
constant: '2.0' at line 19, col 12
constant: '3' at line 19, col 17
constant: '4' at line 20, col 7
constant: '5' at line 20, col 10
constant: '6' at line 20, col 13
constant: '7' at line 21, col 7
constant: '8' at line 21, col 10
constant: '9' at line 21, col 13
constant: '3' at line 23, col 5
constant: '26' at line 23, col 7
constant: '1' at line 26, col 27
constant: '2' at line 26, col 30
constant: '0' at line 27, col 10
```

Example A.4. Output of the precompiler with the `-constants` option.

`malloc`, `realloc` and `free`. The user then needs to convert this part of the code manually, using the `new` C++ statement for allocating the class or using the Standard Template Library (STL) `vector` class [Musser et al. 2001] to allocate instances of the converted type.

Warnings are also issued by the precompiler when variables with "incompatible" types occur in a same expression. As discussed in the algorithm for variable scoping, variables are either tagged as `tKnown`, if they are of a source type listed in the conversion file, or tagged as `tOther` otherwise. When a variable or function of type `tOther` is assigned to or compared with a variable of type `tKnown`, a warning is raised. In Example C.1 we see a converted file with a mixed expression for which the precompiler raises a warning: `Assignment or comparison to`

```
int main(){
  MpIeee a; MpIeee b= MpIeee( "2" );
  MpIeee c= MpIeee( "1" );
  int n=3;
  a = b + c / n;
  cout << "a=" << a << endl;
  return 0;
}
```

Example C.1. Mixed expresssions initial output with warning.

```
int main(){
  MpIeee a; MpIeee b= MpIeee( "2" );
  MpIeee c= MpIeee( "1" );
  MpIeee n= MpIeee( "3" );
  a = b + c / n;
  cout << "a=" << a << endl;
  return 0;
}
```

Example C.2. Mixed expresssions converted without warnings.

`type MpIeee may contain wrong conversion for 'n' at line: 6`. When we extend the conversion file to also convert the variable `n`, no warnings are raised and the output is given in Example C.2.

When a `tKnown` variable is assigned to a `tOther` variable no warning is given, even though this could lead to a compilation error after the precompilation phase. The reason for this is that warnings for assignments to `tOther` variables will clutter the more important warnings for assignments to `tKnown` variables. The precompiler also gives a warning whenever the right hand side of an assignment to a `tKnown` variable is a call to a function, which is implemented in an external header file. Such a warning is necessary since the precompiler does not recursively parse the include files.

4. PRECOMPILATION OF SCIENTIFIC LIBRARIES

To illustrate the applicability of the precompiler, we have converted two existing numerical libraries into libraries that use multiprecision floating-point data types. The first is the GNU Scientific Library which is written in C. The challenge here is not only the type conversion but also the transition from C to C++ since our target types are classes. The second library is the C++ version of Numerical Recipes. The resulting multiprecision libraries can be used standalone or in our MPL programming environment [Schreppers et al. 2005].

4.1 The GNU Scientific Library

We have applied the precompiler to convert the GNU Scientific Library `GSL` so that it uses the C++ multiprecision data type `MpIeee` instead of `float`, `double` and `long double` [Cuyt 2004].

Table II shows the successfully converted directories. The directories which are marked with an * superscript needed minor manual changes. Only

Table II. Successfully Converted GSL Directories (* = needed minor manual modifications)

`blas *`	`doc`	`min`	`rng *`
`block *`	`eigen`	`monte *`	`roots *`
`cblas *`	`err`	`multifit *`	`siman`
`cdf *`	`fft`	`multimin *`	`sort`
`cheb`	`fit`	`multiroots *`	`statistics*`
`combination`	`gsl`	`ntuple *`	`sum`
`complex`	`histogram`	`ode-initval *`	`sys *`
`const`	`integration *`	`permutation *`	`test`
`deriv`	`interpolation`	`poly *`	`utils`
`dht`	`linalg *`	`qrng`	`vector *`
`diff`	`matrix *`	`randist *`	`wavelet`

two directories in the GSL library are not listed in Table II `specfunc` and `ieee-utils`. The reasons for this are explained below.

We comment on some of the most common manual modifications needed to resolve compilation problems.

—Problems related to the target `MpIeee` library:
  - —'?' statements : These C short-hand `if` statements within an assignment can cause compilation errors due to the unary operators + and − combined with the delayed evaluation techniques used by `MpIeee`. For example when writing `MpIeee a = bTest ? b : -b;`, this will generate a compilation error because `-b` returns a DelMpIeee (delayed type `MpIeee`) object. The manual work around is done by changing the statement into : `MpIeee a = bTest ? b : MpIeee( -b );`. This occurs in directory `sys`.
  - —Constants: Often mathematical constants such as `Pi, machine epsilon, e`, . . . are used in the GSL routines. For a proper conversion, these constants have to be replaced by function calls to the appropriate multiprecision implementation of these constants. These multiprecision counterparts, if available in the target library, are very library specific and hence automatic conversion is not obvious. The `-constants` option will help in identifying constants for which manual conversion is required by the user.

—Precompiler-related problems:
  - —External functions: Very often, and certainly in large libraries such as GSL, functions are defined in external header files. Even though the precompiler does not parse header files recursively, its default behavior gives good but not perfect conversion results due to this lack of recursive checking. As described in Section 2.2, the processing of function calls implies that, whenever they occur in the right hand side of an assignment to a variable of type `tKnown`, the parameters and return type of the call are converted to the type of the left hand side. Function calls which are assigned to a variable of type `tOther`, or do not occur in an assignment statement, are copied literally. For the conversion of the GSL library, this gives good results because both the function definition and the call are processed by running the precompiler on all files in the GSL directory. In certain cases, however, the automatic conversion of function calls in the right-hand side

of assignments leads to errors. This is only the case in the `matrix` and `vector` directories of GSL. For example the function `gsl_matrix_get` has `double` as a return type, while one of its parameters is of type `int`. The automatic conversion by the precompiler of a constant integer parameter to the target type for `double`, is therefore incorrect. This problem occurs in the directories `matrix` , `vector`, `linalg`, `multimin`, `multifit` and `rng` and has been resolved manually.

- —`printf` to `cout` conversion: Although this automatic conversion works in most cases there are a few limitations. One of the many `printf` variations allows changing of the value of its parameters through a pointer object. Since this cannot be converted using a regular `cout` instruction a warning is issued. This occurs once in the `test` directory and we converted it manually. Another limitation is that the format string has to be declared in the `printf` statement and not defined as a `char*` variable somewhere else.

—Problems specific to the conversion of GSL:

- —Writing matrix or block objects to file using `fprintf` gives non-POD warnings. Such errors can be manually corrected by rewriting the statements using `std::fstream`.
- —When converting GSL `complex` type tuples to multiprecision, we get compilation errors. This is mainly due to the assignment of constant 0- or 1-valued tuples, for which GSL uses an unnamed `struct`. Such a construction is not allowed for classes. This problem occurs in the directories `block`, `matrix` and `vector` and is manually solved by leaving out the complex datatypes.[5]
- —Ambiguity errors: when compiling the original GSL library, a warning is given in the `statistics` directory. The sources contain a macro wrapper around functions in `minmax_source.c` to turn on debugging output. The macro contains a construct of the form `BASE FUNCTION(gsl_stats,max)` where BASE is a `double` in the original source and `MpIeee` in the converted source. Unfortunately the functions in the GSL library do not all respect the macro definition and some functions give back an `unsigned int` instead of the required `BASE` type. With `BASE` being a `double` this only gives a compiler warning, but after conversion to `MpIeee` this gives an error. This is fixed manually by returning the correct `BASE` type.
- —The `ieee-utils` directory does not compile after conversion. We get errors in the form of: `make_float_bigendian(MpIeee*)::<anonymous union>::f with constructor not allowed in union`. Again this is a C to C++ problem, where anonymous unions on classes are not allowed. Fortunately all functionality implemented in the `ieee-utils` directory is also available in the multiprecision `MpIeee` class library (for example, functions to check if a number is Not-a-Number, is finite, etc.).
- —Problems in the `specfunc` directory can be fixed by doing some manual changes (like implementing the missing atan2 function for `MpIeee`) but since the used algorithms for the special functions are specific for hardware precisions, it is left out.

[5]A complex version of the MpIeee multiprecision datatype is under development.

—Macros sometimes have wrong effects after conversion. One is the `GSL_MAX` macro used in the `monte`, `multiroots`, `roots` and `cdf` directories. This gives the same type of error as described for '?' statements. The same macro also gives a problem analogous to the problem encountered with external functions. In the second part of the '?'-statement, a `tKnown` assignment to a `tOther` left hand side is copied literally by the precompiler, causing a compilation error after conversion.

To improve efficiency of the precompiled code, extra effort has been expended in the conversion of the `cblas` routines. In these routines GSL uses defines for `BASE` and `INDEX` types, the former for data and the latter for indexing. We converted every occurrence of a `BASE` define using a find/replace script and added an extra precompiler rule to convert assignments to the `BASE` type. We manually disabled the complex routines to get a fully functional multiprecision `blas` subset.

Apart from some problems due to the transcription from C to C++ and the occasional failure of a macro after precompilation, the conversion of the GSL library is quite satisfactory. The scripts, configuration files and precompiler sources used to automate the `GSL` are included in the algorithm submission [Schreppers 2006].

4.2 Numerical Recipes

We have also applied the precompiler to the C++ version of Numerical Recipes [Press et al. 2002]. This library is constructed around a `typedef DP`, which is defined as `double`. In the ideal case, all that is needed for transcription to a multiprecision data type is the replacement of this `typedef`. In practice the precompiler is yet again a valuable tool since the `typedef` is not used properly in all the sources. The remaining instances of `double` in the sources are replaced using the precompiler. After conversion about 290 routines could be compiled successfully. There are 28 files which failed due to missing functionality in the MpIeee target library (atan2, ldexp, ...). This shows that precompilation from C++ to C++ using different data types is much more successful and would even be completely successful if the target library provides all the functionality available for the source type.

Of the 290 successfully transcribed files, 270 are linked into a multiprecision library. The other 20 or so files are left out of the library because they are test files or separate programs with a `main` routine not suited to be included in a library.

5. CONCLUSION

Based on our experience with the automatic transcription of the GSL and Numerical Recipes library, we can conclude that our precompiler is a stable tool for the automatic conversion of C/C++ source code into new code that uses the data types of a C/C++ library. The reason we cannot get all converted code working without manual intervention is not shortcomings of the precompiler but rather missing functionality for the target data type or to the transition from C into C++.

The precompiler saves a lot of time and avoids errors that would otherwise occur during manual conversion. Also the readability of the automatically generated code is good. Exactly how much time is gained with the precompiler depends on how long it takes to write a custom conversion file or, if one has already been written, on how much source code needs to be converted and on the complexity of that code.

The precompiler has been developed in such a way that the performance of the precompiled code is very similar to the performance of manually converted code. One of the ways to achieve this has been explained in Section 2.3.

Of course, whether the conversion is done manually or automatically, the performance of the transcribed code also depends on the performance of the C++ (multiprecision) library and on the (in)appropriate use of the data types of that library. A good example where manual conversion is inevitable is discussed in the previous section for the CBLAS routines.

Finally, we observe that the precompiler also has an unintended but quite useful application: it is an easy way to get familiar with a C++ (multiprecision) library. Starting from own C/C++ code, anyone can experiment with a multiprecision library using the precompiler to do the conversion. By looking at the precompiled code, the user can instantly see how to use the library for his or her specific applications.

REFERENCES

Bailey, D. H. 1990. MPFUN: A portable high performance multiprecision package. Technical rep. RNR-90-022, NASA Ames Research Center.

Bailey, D. H. 1993a. Multiprecision translation and execution of Fortran programs. *ACM Trans. Math. Softw. 19*, 3, 288–319.

Bailey, D. H. 1993b. Automatic translation of Fortran programs to multiprecision. Tech. rep., NAS Applied Research Branch.

Bailey, D. H. 1995. MPFUN90: A Fortran-90 based multiprecision system. *ACM Trans. Math. Softw. 21*, 4, 379–387.

Bailey, D. H., Hida, Y., Li, X. S., and Thompson, B. 2002. Arprec: An arbitrary precision computation package. Tech. rep., Lawrence Berkeley National Laboratory, Berkeley, CA.

Beazley, D. 1998. SWIG users manual. Technical rep. UUCS-98-012, Department of Computer Science, University of Utah.

Bia, A., Carrasco, R. C., and Forcada, M. L. 2001. Identifying a reduced DTD from marked up documents. In *Proceedings of the IX Spanish Symposium on Pattern Recognition and Image Analysis (SNRFAI-'01)*. 385–390.

Cuyt, A. 2004. *libMpIeee*. University of Antwerp. http://www.mpieee.ua.ac.be.

Dalheimer, M. K. 2002. *Programming with Qt: Writing Portable GUI Applications on Unix and Win32*, 2nd Ed. O'Reilly & Associates, Inc., Newton, MA.

Galassi, M. et al. 2001. *GNU Scientific Library Reference Manual: Edition 1.0 for GSL Version 1.0*. Network Theory, Bristol, UK.

Haible, B.. 1997. CLN, a class libary for numbers. http://www.ginac.de/CLN/.

Harold, E. R. and Means, W. S. 2001. *XML in a Nutshell*. O'Reilly & Associates, Inc., Newton, MA.

Kim, E. E. 2001. A triumph of simplicity: James Clark on markup languages and XML. *Dr. Dobb's J. Softw. Tools 26*, 7, 56, 58–60.

Musser, D. R., Derge, G. J., and Saini, A. 2001. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*, 2nd Ed. Addison Wesley, Boston, MA.

Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. 2002. *Numerical Recipes in C++*. Cambridge University Press, Cambridge, UK.

SCHREPPERS, W. AND CUYT, A. 2008. A C/C++ Precompiler for autogeneration of multiprecision programs. *ACM Trans. Math. Softw. 34*, 1, Article 5.

SCHREPPERS, W., BACKELJAUW, F., AND CUYT, A. A. M. 2005. Mpl: A multiprecision Matlab—like environment. In *International Conference on Computational Science*, V. S. Sunderam, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, Eds. Lecture Notes in Computer Science, vol. 3514. Springer, Berlin, Germany, 295–303.

SMITH, D. M. 1991. Algorithm 693: A FORTRAN package for floating-point multiple-precision arithmetic. *ACM Trans. Math. Softw. 17*, 2, 273–283.

WEISS, M. A. 1997. *Data Structures and Algorithm Analysis in C*++ (2nd Ed.). Addison-Wesley Publishing Company, Boston, MA.

XML Schema 2001. XML Schema Part 1: Structures, W3C Recommendation. http://www.w3c.org/TR/xmlschema-1/.

ZIMMERMANN, L. AND THE POLKA PROJECT. 2004. *MPFR*, 2.1.0 ed. INRIA Lorraine and LORIA.

Received July 2005; revised May 2006, January 2007; accepted February 2007