Arithmetic Explorer Tools

Walter.Schreppers@ua.ac.be

March 25, 2002

Overview

How to broaden the user base of Arithmos library (students, researchers and alike)? We need to make the usage and experimentation with the library as simple as possible. In this seminar we discuss the following tools to extend the Arithmetic Explorer:

- Precompiler
- AELab
- Python extension.

Precompiler Motivation

The precompiler needs to be a tool to ease the use of any numeric library, not just Arithmos.

The motivation for this tool:

- Compare different numeric libraries.
- Manual conversion is error prone.
- Other translation programs are not generic.

Expectations

- Learn the usage of a library, just by looking at the output of the precompiler.
- Convert existing algorithms so that they use the new numeric library.
- Attract people to our numeric library.
- Write one source file, and use it with different libraries.

Specification

- A suitable format for the conversion file to specify the types of variables which must be converted and how constants must be converted.
- Use variable scoping to convert assignments and comparisons correctly.
- The ability to skip conversion in for loops for higher performance.
- Ability to skip the conversion of specific variables or functions.
- Convert <stdio.h> printf instructions into <iostream> cout instructions.

Implementation

Here is an overview of the implementation:

- Lexer to split up the source code into tokens.
- Parser class receives tokens to do scoping and recognize certain key instructions which need special handling:
  - stdio printf instructions
  - for loops
  - return instructions
- XML format for the conversion configuration file. We wrote our own lightweight XML parser.
- Matcher class, to find suitable conversion for each token which may need conversion.

Parser and Preparser

- The Parser is the centre of the precompiler and uses the other classes to do the conversion of the key C/C++ statements and variables.
- The Convert class is also called by the Parser to convert assignments to variables and to convert return statements.
- The PreParser, inherits the Parser class and is used to generate a configfile for variable skipping.

Scoping

Here is the simplified algorithm to do variable scoping.

1. Copy everything literally up to the first word token.
2. Check for the defined types in the configuration file, add variable to Variable queue if known type is found.
3. Whenever an { or } is encountered open or close the scope on Variables queue is called respectively.
4. Check for reserved words, if these are not printf, for or return then just copy the word literally.
5. Look in variables list to see if word is a name of a known variable.
6. Finally check for potential other variable types, these consist of two consecutive non reserved words.
7. goto step 1.

Precompiler Usage

The available options for the precompiler:

```
Options:
  -nofor              don't convert variables in for loops
  -c <configfile>     skip variables
  -x <convertfile>    xml file to specify conversion of types
  -radix <size>       set the radix size, default size=2
  -precision <size>   set the precision size, default size=24
  -exp <low> <up>     set the exponent default low=-126, upper=127
  -expbits <bitsize>  set the nr of bits the exponent must use
  -round <n|p|m|z>    set the rounding type, default = n
                       n = round to nearest
                       p = round to plus inf
                       m = round to min inf
                       z = round to zero
  -outputverbose      set verbose output
  -outputmpieee <p|d|b|y|h|r|f>+ set the Mpieee output type,
                                  default = d
                       p = parameter
                       d = decimal
                       b = binary
                       y = binary representation
                       h = hexidecimal representation
                       r = rational
                       f = flags
  -outputrational <d|r|f>+ set the output type, default = r
                       d = decimal
                       r = rational
                       f = flags
  -default            insert precision settings after the main(){
  -noprintf           don't convert printf statements
  -preparse           use preparser to make a list of all constructed
                       variables in <outputfile>
```

XML Parser

The precompiler uses a conversion file which is in the XML format. To parse this file we used our own lightweight XML parser. This parser constructs an n-airy tree of the file which can be traversed. Which is basically the DOM (Document Object Model) specification of XML.

Why did we write our own XML parser?

- The Qt DOM classes are fine but are not free of license and are still slower than our own parser.
- Other free of license alternatives, like expat are either too slow or memory intensive.
- Speed is important, implementing only the essentials makes our parser the fastest around. Our xml parser was also used by the baseML project http://baseml.baselabs.org. By using it they had a speed improvement of 400% compared to the Qt libraries.

Conversion file

The conversion file must comply with the following DTD:

```
<!ELEMENT include (#PCDATA)>
<!ATTLIST include global (true|false) #IMPLIED>

<!ELEMENT source (keyword|token)+>
<!ATTLIST source name CDATA #REQUIRED>

<!ELEMENT assignment (keyword|token)>
<!ATTLIST assignment name CDATA #REQUIRED>

<!ELEMENT target (keyword)>
<!ATTLIST target name CDATA #REQUIRED>

<!ELEMENT keyword (#PCDATA)>
<!ELEMENT token (#PCDATA)>

<!ELEMENT convert ((source|assignment|target),target,operation?)>
<!ATTLIST convert name CDATA #IMPLIED>

<!ELEMENT operation (#PCDATA)>
```

Example

A concrete example, consider the following code:

```
int pow(int a,int n){
  if(n<0) return 0;
  for(int i=0;i<n;i=i+1){
    a*=a;
  }
  return a;
}

int main(){
  int a=3,b=1,c=3;
  if(a+b == 3){
    char c;
    c=32; //c is a char here!
  }
  else{
    c=32+pow(a,b); //c is a long int
  }
  printf("c=%i",c);
  return 0;
}
```

Example (2)

If we want to change the int variables into BigInt variables we use the following configuration file:

```
<?xml version="1.0"?>
<document>

  <include> BigInt.hh     </include>
  <source name="sint">
    <keyword> int </keyword>
  </source>
  <assignment name="integer">
    <token> integer </token>
  </assignment>
  <target name="tint">
    <keyword> BigInt </keyword>
  </target>

  <!-- the conversion rules -->
  <convert name="int to bigint">
    <source name="sint"/>
    <target name="tint"/>
  </convert>
  <convert name="integer to bigint">
    <assignment name="integer"/>
    <target name="tint"/>
    <operation> toQuotedString </operation>
  </convert>
</document>
```

Example (3)

When we run the precompiler with the source file and the configuration file we get this output file:

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

#include "BigInt.hh"

#include <stdio.h>

BigInt pow(BigInt a,BigInt n){
  if(n<BigInt( "0" )) return BigInt( "0" );
  for(BigInt i= BigInt( "1" );i<n;i=i+BigInt( "1" )){
    a*=a;
  }
  return a;
}

int main(){
  BigInt a= BigInt( "3" );BigInt b= BigInt( "1" );
  BigInt c= BigInt( "3" );
  if(a+b == BigInt( "3" )){
    char c;
    c=32; //c is a char here!
  }
  else{
    c=BigInt( "32" )+pow(a,b); //c is a long int
  }
  cout<<"c="<<c<<"\n";
  return 0;
}
```

Example (4)

When we use the -nofor option we can see that the forloop inside pow() is not converted.

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

#include "BigInt.hh"

#include <stdio.h>

BigInt pow(BigInt a,BigInt n){
  if(n<BigInt( "0" )) return BigInt( "0" );
  for(int i=1;i<n;i=i+1){
    a*=a;
  }
  return a;
}

int main(){
  BigInt a= BigInt( "3" );BigInt b= BigInt( "1" );
  BigInt c= BigInt( "3" );
  if(a+b == BigInt( "3" )){
    char c;
    c=32; //c is a char here!
  }
  else{
    c=BigInt( "32" )+pow(a,b); //c is a long int
  }
  cout<<"c="<<c<<"\n";
  return 0;
}
```

Example (5)

When we use the -noprintf option we can see that the printf instruction is not changed and some includes were left out.

```
#include "BigInt.hh"

#include <stdio.h>

BigInt pow(BigInt a,BigInt n){
  if(n<BigInt( "0" )) return BigInt( "0" );
  for(BigInt i= BigInt( "1" );i<n;i=i+BigInt( "1" )){
    a*=a;
  }
  return a;
}

int main(){
  BigInt a= BigInt( "3" );BigInt b= BigInt( "1" );
  BigInt c= BigInt( "3" );
  if(a+b == BigInt( "3" )){
    char c;
    c=32; //c is a char here!
  }
  else{
    c=BigInt( "32" )+pow(a,b); //c is a long int
  }
  printf("c=%i\n",c);
  return 0;
}
```

Example (6)

When we use an extra -preparse option the output file will contain a list of the variables with their function name (/ for global variable) and original type.

```
pow             pow             int
pow             a               int
pow             n               int
pow             i               int
main            a               int
main            b               int
main            c               int
```

The Configuration file

Using the skip configuration file with the -c <configfile> option. This file has the following syntax

- Each line containing '::' is a configuration element. The string left of this mark is a function name and the string to the right is a variable name. When this variable is found in the corresponding function, it will not be converted.
- An empty function name means this variable will be skipped in every function.
- An empty variable name defines a function that must be skipped intirely.
- Giving the same function name and variable name means that the return type of the function must not be changed.
- Lines without '::' are ignored.
- Lines starting with '#' are also ignored.

Skipping Configuration Example

Let's say we want the pow function to return an int and we want to skip conversion of variable c in the main() function. We would make the following simple configuration file:

```
#this specifies that the return type of the pow function
#may not be converted
pow::pow

#the variable c inside the main function may not be converted
main::c
```

Skipping Configuration Example (2)

When using this skip configuration file the output file of the precompiler will look like this:

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

#include "BigInt.hh"

#include <stdio.h>
BigInt f= BigInt( "1" );
int  pow(BigInt a,BigInt n){
  if(n<BigInt( "0" )) return 0;
  for(BigInt i= BigInt( "1" );i<n;i=i+BigInt( "1" )){
    a*=a;
  }
  return a;
}

int main(){
  BigInt a= BigInt( "3" );BigInt b= BigInt( "1" );
  int  c= 3;
  if(a+b == BigInt( "3" )){
    char c;
    c=32; //c is a char here!
  }
  else{
    c=32+pow(a,b); //c is a long int
  }
  cout<<"c="<<c<<"\n";
  return 0;
}
```

MPFR++ Example

Mpfr++ is a C++ wrapper for the mpfr library, which is a multipricision floating point library based on the GMP libs. Now we take a simple piece of code, using float variables that we want to convert to mpfr++ variables.

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

int main(){
  float a=12345;
  float b=0.345;
  a=a+1;
  cout<<"a="<<a<<endl;
  cout<<"b="<<b<<endl;
}
```

MPFR++ Example (2)

To convert it we use the following mpfr.xml configuration file:

```xml
<?xml version="1.0"?>
<document>
  <include> Mpfr++.hh </include>
  <source name="sfloat">
    <keyword> float </keyword>
  </source>
  <assignment name="int">
    <token> integer </token>
  </assignment>
  <assignment name="dec">
    <token> decimal </token>
  </assignment>
  <target name="tmpi">
    <keyword> MpfrReal </keyword>
  </target>

  <convert name="int to bigint">
    <source name="sfloat"/>
    <target name="tmpi"/>
  </convert>
  <convert name="int to mpfr">
    <assignment name="int"/>
    <target name="tmpi"/>
    <operation> toString </operation>
  </convert>
  <convert name="dec to mpfr">
    <assignment name="dec"/>
    <target name="tmpi"/>
    <operation> toString </operation>
  </convert>
</document>
```

MPFR++ Example (3)

The resulting output file looks like this:

```
#include "Mpfr++.hh"

#include <iostream>
using namespace std;

int main(){
  MpfrReal a= string( "12345" );
  MpfrReal b= string( "0.345" );
  a=a+string( "1" );
  cout<<"a="<<a<<endl;
  cout<<"b="<<b<<endl;
}
```

Which can then be compiled when the gmp and mpfr libs are installed together with the mpfr++ class files.

Unfortunately there is one thing missing, a way to specify the precision. For the Arithmos library this is done with the extra command line options. We see here it would be better to specify these settings in the XML conversion file. Then we could let the precompiler add something like:

```
MpfrReal::SetDefaultPrecision(999);
```

Conclusions and Issues

Some issues with the implementation:

- We need an extra tag to do precision settings, this will make a lot of specific Arithmos command line options obsolete. It will also give the possibility to change default precision settings for other libraries.

- Optionally: change the skip configuration file to a XML format.

- The printf to cout conversion is customized for Arithmos, not for generic conversions. It works for mpfr++ but it might give problems for other libraries (due to custom precision settings etc.)

- Implement a version of Dijkstra's shortest path algorithm (the BigInt <-> Mpleee problem).

Conclusions and Issues (2)

Why is the shortest path algorithm needed?

Consider this code fragment:

```
int a=3.2;
float b=2;
```

What happens here is that a decimal constant is assigned to an int variable and an integer constant assigned to a float variable.

Conclusions and Issues (3)

With the supplied convert.xml configuration file included with the precompiler sources, this will be converted into :

```
BigInt a=BigInt( MpIeee( "3.2" ) );
MpIeee b=MpIeee( BigInt( "2" ) );
```

The first conversion (of the int) is done the way we want to. Meaning we create a MpIeee of the decimal constant and give this MpIeee object to the BigInt constructor. The second however could be done shorter since an integer value can also be assigned to an MpIeee. The problem is the duality and to solve it we would have to add another convert rule. The problem with adding such a rule is that in this case, an int might always be converted to an MpIeee first then to BigInt.

Conclusion: some kind of precedence to rules has to be made.

Solutions: implementing either the shortest path algorithm or making the conversion file more redundant.

The end?

So we have a tool to unify most multi precision libraries. By defining the needed configuration files for these different libraries we can make the usage of them transparent to their users.

They can easily use and compare the different libraries without having to change their C or C++ sources. Should it ever be needed they are even able to create their own configuration files for other purposes or new libraries which were not yet supported by our precompiler.

This might satisfy people who have knowledge of C or C++ but not the Matlab users. To target this audience we have another tool, AELab.

AELab

AELab is a compiler which can transform scripts with a Matlab syntax into C++ code.

Motivation:

- The Matlab environment only supports variables with double precision.
- AELab we will be able to use multipricision libraries, first the Arithmos library and in the future also other libraries will be possible.
- The AELab compiler can be easily integrated into the Arithmetic Explorer.
- It is free of cost and licenses, ideal for students.

Specification

We have the following basic specification for our AELab.

1. A way to parse the matlab like script files and convert them into C++ code.

2. A way of making new functions, possibly also in custom C++ code for better performance, that can be used within the scripts.

3. Implementing a matrix class to support the many matrix operations of matlab.

4. Create a gui so that AELab can be used in the Arithmetic Explorer.

Implementation

1. We implemented lex/yacc to parse the matlab like script files.

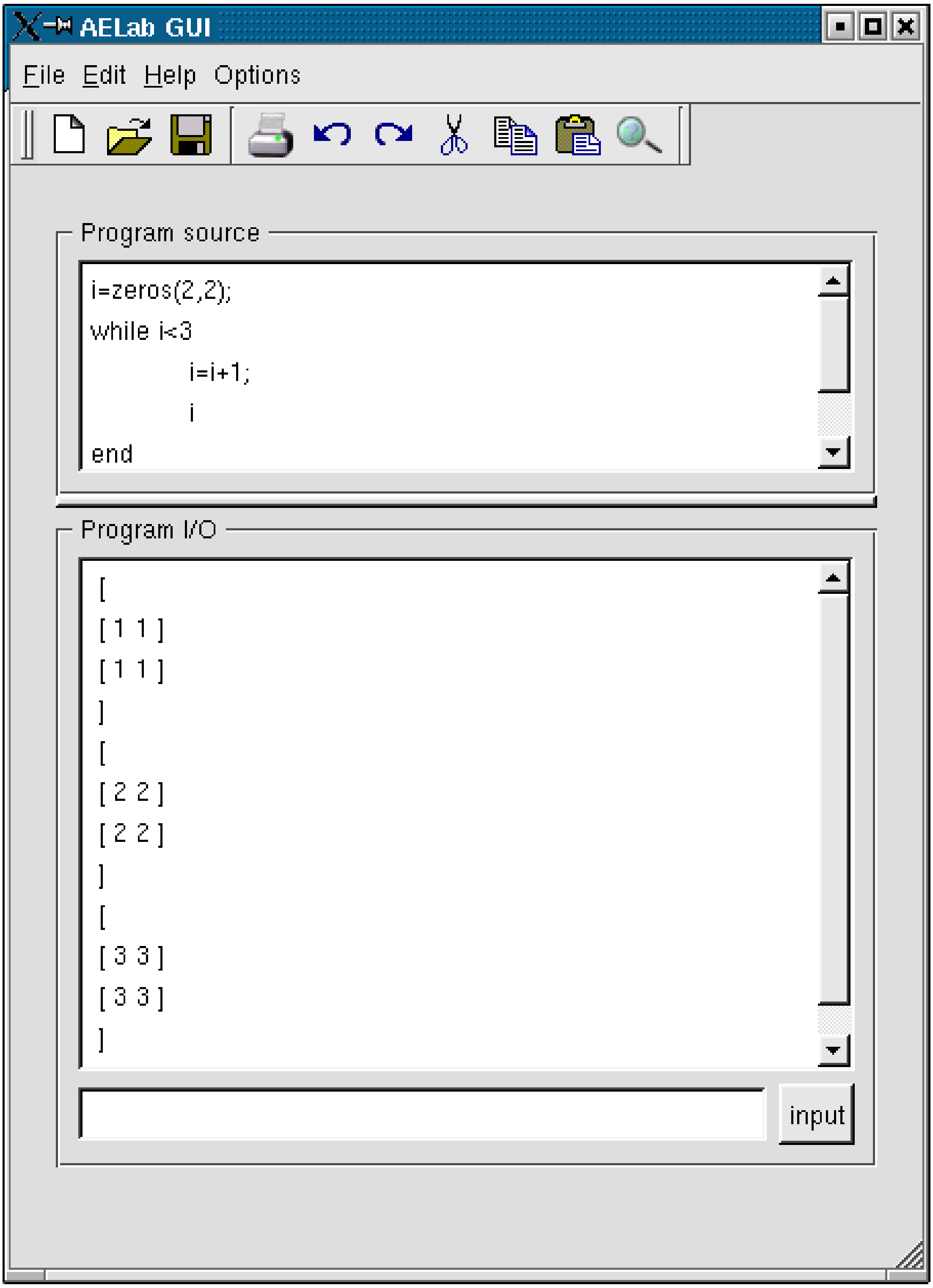
2. The FuncFinder class takes care of finding previously compiled functions and custom C++ functions to use in our scripts.

3. For now the matrix class is a 2 dimensional vector of doubles. We have 2 students working on a multidimensional templated implementation of this class.

4. The gui is called AElab GUI, with this tool people can easily edit script files and immediately run them or compile them to executables.

AELab GUI

Here's the AELab GUI running a simple script.



AELab Conclusion

This concludes the AELab tool in the future it might replace the current integrated Parser in the Arithmetic Explorer.

It basically all depends on how well our students implement the matrix class since this is the core variable type for every compiled script.

The matrix and indexation classes should at least be verry stable, later on more focus can be put on optimizing it for speed.

In the following section we will discuss a way to extend the Arithmetic Explorer so that is able to run python scripts.

Python Extension - Outline

Outline:

- We can use SWIG to create wrappers for the Arithmos Library Classes.
- This would enable us to use the Arithmos Library types inside Python scripts.
- Finally the Python compiler can be used from a gui, similar to the way AELab is being used in the AELab GUI.