Chapter 5. LARA Strategies for Data Type Conversions

Loïc Besnard

Univ. Rennes, CNRS, IRISA, France
loic.besnard@irisa.fr

Abstract

To easily explore different representations of C numerical types (double, float, fixed point, half precision...), the user should develop its applications by the introduction of data types abstraction. But when it is not the case, it becomes fastidious to do it after. In this chapter, we propose some LARA aspects developed in ANTAREX projects that automatically abstract types of applications.

1 Introduction

Error-tolerating applications are increasingly common in the emerging field of real-time HPC. Thus, recent works investigated the use of customized precision in HPC as a way to provide a breakthrough in power and performance. But, a user does not often think about the parametrization of its application in terms of types. So, in ANTAREX project, we have developed a set of LARA aspects enabling the parametrization of the types of an application.

The remainder of this chapter is organized as follows. We describe first the principles of the technique, followed by the LARA aspects developed in ANTAREX project to apply this technique without requiring user modifications of the source code. In a third section, experimental results are presented.

2 Principles of the custom precision

The custom precision is the parametrization of the user application in term of data types for facilitating the test of different representations of types.

The developed mechanism parametrizes the application by

- replacing the references to a type by a new symbol, with the possibility to exclude some types,
- replacing the formats used in the Input/Output instructions by new symbols. For example, in C language, the read of a double requires the "%lf" format and the float requires the "%f" ones.
- replacing the references to math functions (sin, sinh, ...) by new symbols.
- replacing the references to constants by new symbols. It is required for some type representations such that the half precision, the constant values are written in a specific syntax.

The developed mechanism is implemented as a set of LARA aspects. Two LARA aspects are proposed for the "custom precision" at the high level: `CustomPrecisionFor(aType)` and `CustomPrecisionForExcept(aType,[excl_1,...,excl_n])` where `aType`, `excl_1,...,excl_n` are strings. A type such that `excl_i` defined by `typedef double exl_i;` will not be affected by the abstraction(ie the typedef definition is not modified).

The parametrization creates some files in which the parameters are stored:

- `INRIA_PRECISION_DEFS_[aType].h` that defines the parameters with the original values for each new created symbols.
- `INRIA_PRECISION_DEFS_XX_template.h` in which the definition of the symbols must be updated by the user, where XX denotes the new symbol assigned to the custom type. This file may be used to test implementation of type representations such that fixed-point [1], half precision [2].

An another aspect, called
`CustomPrecisionGenParametersFor(aType)` is a utility aspect that produces a file `INRIA_PRECISION_DEF`
for `aType` some predefined type (a particular case of the template file). For example, when the original type is double and one want to test the float version, the definitions for float may be easily derived from the double one. It is the role of this aspect.

3 How to use

To illustrate the use of the developed aspects, consider the simple C++ program shown in Figure 1.

```cpp
#include <iostream>
#include <cmath>
#include <stdlib.h>
#include <vector>
using namespace std;

const int vint = 3;
const double p = 2.3;
const double tab1[2]={10.8,20.3};

vector<double> vect1;

std::vector<double> vect2;

double foo(double vv)
{
 return vv+9.9;
}

int main (int argc, char *argv[])
{
  double x;
  vect2.push_back(tab1[0]);
  x = 50.3 + p * foo(tab1[1])+ sin(vect2[0]);
    return (int) x;
}
```

Figure 1: Simple example for custom precision.

To test the double to float transformation, the user may apply the launcher aspect shown in Figure 2.

```
1  import clava.Clava;
2  import clava.ClavaJoinPoints;
3  import antarex.precision.CustomPrecision;
4
5  aspectdef Launcher
6        call CustomPrecision_Initialize();
7        call CustomPrecisionFor('double');
8        call CustomPrecisionGenParametersFor('float');
9        call CustomPrecision_Finalize();
10 end
```

Figure 2: A Data Type parametrization.

First of all, the import of the custom precision LARA aspects must be specified (line 3) by

```
import antarex.precision.CustomPrecision;
```

A session must begin by calling the `CustomPrecision_Initialize()` aspect (line 6). This aspect initializes the internal structure of the defined aspects. The line 7 is an example of call to the presented aspects for custom precision applied to a selected type ('double' will be replaced by a new symbol in the weaved code). The line 8 is the illustration of the use of the utility asking the generation of data type abstraction for the 'float' type. The custom precision session is then ended by a call to the `CustomPrecision_Finalize()` aspect (line 9). This aspect finalizes the internal structure of the defined aspects.

The effects of the execution of the launcher aspect is shown in the Figure 3. It is the new version of the user application.

```
#include <iostream>
#include <cmath>
#include <stdlib.h>
#include <vector>
#include "INRIA_PRECISION_DEFS.h"
using namespace std;
int const vint = 3;
_INRIA_DATATYPE_1 const p = _INRIA_CSTE_1;
_INRIA_DATATYPE_1 const tab1[2] = {_INRIA_CSTE_2, _INRIA_CSTE_3};
vector<_INRIA_DATATYPE_1> vect1;
std::vector<_INRIA_DATATYPE_1> vect2;

_INRIA_DATATYPE_1 foo(_INRIA_DATATYPE_1 vv) {

   return vv + _INRIA_CSTE_4;
}


int main(int argc, char * argv[]) {
   _INRIA_DATATYPE_1 x;
   vect2.push_back(tab1[0]);
   x = _INRIA_CSTE_5 + p * foo(tab1[1]) + sin_INRIA_DATATYPE_1(vect2[0]);

   return (int) x;
}
```

Figure 3: Weaved code of the Simple example.

One can observe that

- the references to the `double` type have been replaced by a new symbol `_INRIA_DATATYPE_1`,
- the constants have been replaced by new symbols prefixed by `_INRIA_CSTE_`,
- the reference to `sin` has been replaced by a new symbol `sin_INRIA_DATATYPE_1`,
- a new include, `INRIA_PRECISION_DEFS.h`, has been introduced in the component. The new generated symbols are assumed to be defined in this include.

The other generated files by the `CustomPrecisionFor(aType)` aspect are shown in Figure 4 and in Figure 5

```c
#ifndef _INRIA_PRECISION_DEFS_DOUBLE_H_
#define _INRIA_PRECISION_DEFS_DOUBLE_H_

// Original (double) Version. )

#define _INRIA_DATATYPE_1 double
/* Extracted constants */

#define _INRIA_CSTE_5 50.3
#define _INRIA_CSTE_4 9.9
#define _INRIA_CSTE_3 20.3
#define _INRIA_CSTE_2 10.8
#define _INRIA_CSTE_1 2.3
/* IO with format */

/* Referenced functions */

#define sin_INRIA_DATATYPE_1 sin

#endif
```

Figure 4: Double version of the parametrization.

The first one (Figure 4) is the file that must be used to define the `INRIA_PRECISION_DEFS.h` to have the original version (ie the `double` version).

```c
#ifndef _INRIA_PRECISION_DEFS__INRIA_DATATYPE_1_TEMPLATE_H_
#define _INRIA_PRECISION_DEFS__INRIA_DATATYPE_1_TEMPLATE_H_

// Template Version ( IT MUST BE UPDATED )

#define _INRIA_DATATYPE_1 '_TO_BE_FIXED'
/* Extracted constants */
#define _INRIA_CSTE_5 _TO_BE_FIXED (50.3)
#define _INRIA_CSTE_4 _TO_BE_FIXED (9.9)
#define _INRIA_CSTE_3 _TO_BE_FIXED (20.3)
#define _INRIA_CSTE_2 _TO_BE_FIXED (10.8)
#define _INRIA_CSTE_1 _TO_BE_FIXED (2.3)
/* IO with format */
/* Referenced functions */
#define sin_INRIA_DATATYPE_1 _TO_BE_FIXED (sin)

#endif
```

Figure 5: Float version of the parametrization.

The template file (Figure 5) may be modified and used to define the `INRIA_PRECISION_DEFS.h` to have an another typed version of the application.

```
#ifndef _INRIA_PRECISION_DEFS_half_H_
#define _INRIA_PRECISION_DEFS_half_H_

// Testing half-precision (Christian Rau, MIT licence)
#include "half.hpp"
using half_float::half;
using namespace half_float::literal;
#define _INRIA_DATATYPE_1 half
/* IO with format */
/* Constants */
#static _INRIA_DATATYPE_1 INRIA_CSTE_5 = 50.3_h ;
#static _INRIA_DATATYPE_1 #define _INRIA_CSTE_4 9.9_h ;
#static _INRIA_DATATYPE_1 #define _INRIA_CSTE_3 20.3_h ;
#static _INRIA_DATATYPE_1 #define _INRIA_CSTE_2 10.8_h ;
#static _INRIA_DATATYPE_1 #define _INRIA_CSTE_1 2.3_h ;
/* Referenced Math functions */
#define sin_INRIA_DATATYPE_1 sin
#endif
```

Figure 6: Half precision version of the parametrization.

An example of such a definition is given in Figure 6 using a half precision representation [2].

The effect of the `CustomPrecisionGenParametersFor('float')` aspect is the production of the file shown in Figure 7.

```
#ifndef _INRIA_PRECISION_DEFS_FLOAT_H_
#define _INRIA_PRECISION_DEFS_FLOAT_H_

// float Version.

#define _INRIA_DATATYPE_1 float
/* Extracted constants */
#define _INRIA_CSTE_5 50.3
#define _INRIA_CSTE_4 9.9
#define _INRIA_CSTE_3 20.3
#define _INRIA_CSTE_2 10.8
#define _INRIA_CSTE_1 2.3
/* IO with format */
/* Referenced functions */
#define sin_INRIA_DATATYPE_1 sinf
#endif
```

Figure 7: Float version of the parametrization.

It must be used to define the `INRIA_PRECISION_DEFS.h` to have a `float` version of the application.

4 Selected experimental results

The custom precision has been applied

- on a "betweenness centrality" algorithm previously written in double precision. The parametrization of the application has been generated and tested with double, float and half precision, and applied on several graphs. The maximal gain between the double and float version is around 16% with acceptable results. The half version is not applicable, because the maximal value is reached.

- on routing algorithm (process of selecting a path for traffic) previously written in double precision. The gain of around 10% between the double and float versions has been obtained. As for the betweenness algorithm, the half precision is not applicable.

5 Conclusion

This chapter presented the framework developed in the ANTAREX project to automatically apply the custom precision on C/C++ applications. The source to source transformations, by LARA aspects, on the types have been defined to facilitate the test of type representations. This technology has been successfully applied to a use case of ANTAREX project.

References

[1] Fixed-point arithmetic. `https://en.wikipedia.org/wiki/Fixed-point_arithmetic`.

[2] Half-precision floating point library. `http://half.sourceforge.net`.