Developer's guide

Version 1.0, Last updated March 19, 2018
Author: Valentin Clement

Contents

Introduction

This document describes the internal and external components of the CLAW Compiler. It gives an in-depth overview of the architecture of the compiler, its libraries as well as how to implement new transformations.

Conventions used in this document

Keywords and punctuation that are part of the actual specification will appear in typewriter font:

`!$claw`

Italic font is used where a keyword or other name must be used:

`!$claw` *directive-name*

Optional syntax is enclosed in square brackets; an option that may be repeated more than once is followed by ellipses:

`!$claw` *directive-name* [*clause* [[,] *clause*]. . .]

Chapter 1

Architecture

1.1 OMNI Compiler

The CLAW Compiler is based on the OMNI Compiler [1]. The OMNI Compiler Project provides a set of programs to build source-to-source translator for C/C++ and Fortran. It is jointly developed by the Programming Environment Research Team of the RIKEN AICS and the HPCS Lab. of the University of Tsukuba both located in Japan. The CLAW Compiler is a contributor to this project. The CLAW Compiler is using the Fortran front-end and back-end as well as the C back-end of the OMNI Compiler. Similar transformation mechanism could be applied without to much effort to C or C++ code by using their corresponding fron-end. The OMNI Compiler Project is actively developed as of today. Contributions or issues can be reported on their master repository [2].

1.1.1 The Fortran front-end

The Fortran front-end is the program that parses Fortran source code and that generates its Intermediate Representation (IR). The Fotran grammar is written in `yacc` and the processing and IR generation is written in `C`. This program can be called individually as shown in Listing 1.1.

```
1 F_Front -M my_module_dir -o my_program.xml my_program.f90
```

Listing 1.1: Call F_Front

As the OMNI Compiler acts as an actual compiler, it also has a notion of module file. When a Fortran module is parsed with the front-end, a `.xmod` file is generated. This file contains subroutines and function signatures and type definitions. This file is then needed to parse other Fortran code depending on the module. This mechanism allows the cross-file transformation capabilities of the CLAW Compiler.

```
1 MODULE m1
2   USE m2
3 END MODULE m1
```

Listing 1.2: module_m1.f90

```
1 MODULE m2
2   USE m3
```

```
3 END MODULE m2
```

Listing 1.3: module_m2.f90

```
1 MODULE m3
2 END MODULE m3
```

Listing 1.4: module_m3.f90

In the simple example shown in listings 1.2, 1.3 and 1.4, module m1 depends on module m2 and then module m2 depends on module m3. These source files should be processed by the front-end as shown in listing 1.5 to respect the dependency order.

```
1 F_Front -o module_m3.xml module_m3.f90       # produces m3.xmod
2 F_Front -M . -o module_m2.xml module_m2.f90 # uses m3.xmod and produces m2.xmod
3 F_Front -M . -o module_m1.xml module_m1.f90 # uses m2.xmod and produces m1.xmod
```

Listing 1.5: Parse module with dependencies

The `-M` option tells the front-end where to search and store the module files. Several path can be given for the search but the first one will be used to store the newly created `.xmod` files.

1.1.2 XcodeML/F

XcodeML/F is the IR used by the translator in the CLAW Compiler. This IR is based on the XML format and is described in a specification document [3, 4]. It allows to have an high-level representation of Fortran 2008 programs.
Listing 1.6 is a simple Fortran program. Its XcodeML/F IR is shown in Listing 1.7. A typical XcodeML/F translation unit is composed of the followings sections:

- Type table with all the type definitions used in the translation unit (Listing 1.7 line 6-24).
- Global symbols table listing all the symbols at global scope in the translation unit (Listing 1.7 line 25-29).
- Global declarations section listing the actual function and/or module declarations in the translation unit (Listing 1.7 line 30-82).

```
1  PROGRAM xcodeml_sample
2    IMPLICIT NONE
3
4    INTEGER :: my_integer
5    REAL(KIND=8), DIMENSION(10) :: my_double_precision_real_array
6
7    DO my_integer = 1, 10
8      my_double_precision_real_array(my_integer) = my_integer ** 2
9    END DO
10 END PROGRAM xcodeml_sample
```

Listing 1.6: Basic Fortran program

```
<XcodeProgram source="basic_fortran.f90"
              language="Fortran"
              time="2016-10-21 15:25:40"
              compiler-info="XcodeML/Fortran-FrontEnd"
              version="1.0">
  <typeTable>
    <FbasicType type="R7fc66b5051e0" ref="Freal">
      <kind>
        <FintConstant type="Fint">8</FintConstant>
      </kind>
    </FbasicType>
    <FbasicType type="R7fc66b505380" ref="R7fc66b5051e0"/>
    <FbasicType type="A7fc66b505450" ref="R7fc66b505380">
      <indexRange>
        <lowerBound>
          <FintConstant type="Fint">1</FintConstant>
        </lowerBound>
        <upperBound>
          <FintConstant type="Fint">10</FintConstant>
        </upperBound>
      </indexRange>
    </FbasicType>
    <FfunctionType type="F7fc66b503f10" return_type="Fvoid" is_program="true"/>
  </typeTable>
  <globalSymbols>
    <id type="F7fc66b503f10" sclass="ffunc">
      <name>xcodeml_sample</name>
    </id>
  </globalSymbols>
  <globalDeclarations>
    <FfunctionDefinition lineno="1" file="basic_fortran.f90">
      <name type="F7fc66b503f10">xcodeml_sample</name>
      <symbols>
        <id type="Fint" sclass="flocal">
          <name>my_integer</name>
        </id>
        <id type="A7fc66b505450" sclass="flocal">
          <name>my_double_precision_real_array</name>
        </id>
      </symbols>
      <declarations>
        <varDecl lineno="4" file="basic_fortran.f90">
          <name type="Fint">my_integer</name>
        </varDecl>
        <varDecl lineno="5" file="basic_fortran.f90">
          <name type="A7fc66b505450">my_double_precision_real_array</name>
        </varDecl>
      </declarations>
      <body>
        <FdoStatement lineno="7" file="basic_fortran.f90">
          <Var type="Fint" scope="local">my_integer</Var>
          <indexRange>
            <lowerBound>
              <FintConstant type="Fint">1</FintConstant>
            </lowerBound>
            <upperBound>
              <FintConstant type="Fint">10</FintConstant>
            </upperBound>
            <step>
              <FintConstant type="Fint">1</FintConstant>
```

```
61              </step>
62            </indexRange>
63            <body>
64              <FassignStatement lineno="8" file="basic_fortran.f90">
65                <FarrayRef type="R7fc66b505380">
66                  <varRef type="A7fc66b505450">
67                    <Var type="A7fc66b505450" scope="local">
                          my_double_precision_real_array</Var>
68                  </varRef>
69                  <arrayIndex>
70                    <Var type="Fint" scope="local">my_integer</Var>
71                  </arrayIndex>
72                </FarrayRef>
73                <FpowerExpr type="Fint">
74                  <Var type="Fint" scope="local">my_integer</Var>
75                  <FintConstant type="Fint">2</FintConstant>
76                </FpowerExpr>
77              </FassignStatement>
78            </body>
79          </FdoStatement>
80        </body>
81      </FfunctionDefinition>
82    </globalDeclarations>
83  </XcodeProgram>
```

Listing 1.7: XcodeML/F IR

1.1.3 The Fortran back-end

The Fortran back-end is the program that decompiles IR back to Fortran code. It is a simple Java program that implements a visitor pattern across the AST. It can be called as a standalone program (See Listing 1.8) or directly within the translator as it is done in the CLAW Compiler (See Listing 1.9).

```
1 java -cp <omni-install-path>/share/xcalablemp/om-f-back.jar:<omni-install-path>/
      share/xcalablemp/om-exc-tools.jar xcodeml.f.util.omx2f -l xcodeml.xml
```

Listing 1.8: Execute the Fortran back-end as a standalone

```
1 PrintWriter writer =
2   new PrintWriter(new BufferedWriter(new FileWriter(outputFile)));
3 XmToolFactory toolFactory = new XmToolFactory("F");
4 XmOption.setCoarrayNoUseStatement(true);
5 XmOption.setDebugOutput(false);
6 XmDecompiler decompiler = toolFactory.createDecompiler();
7 XmDecompilerContext context = toolFactory.createDecompilerContext();
8 // xcodemlDocument is the XcodeML/F DOM document to write to file
9 decompiler.decompile(context, xcodemlDocument, writer);
```

Listing 1.9: Fortran back-end called from Java

The C back-end is a similar program but working on the XcodeML/C IR.

1.2 CLAW Compiler

The `clawfc` is the program handed to the end-user. It performs all the needed step to have a source-to-source translation of a Fortran source code. As shown in Figure 1.1, it is composed by various programs under-the-hood. The complete workflow is devided as follows:

1. **FPP**: The Fortran source code is preprocessed by a standard Fortran preprocessor. The `FC` environment variable is used by the CMake build system to determine it. **FPP** is needed to have a syntactically correct Fortran code to pass to the parser. Code with preprocessor directive is impossible to parse and to transform.
2. **F_Front**: The preprocessed Fortran source code is then parsed by the OMNI Compiler Fortran front-end to produce the input XcodeML/F IR.
3. **CX2T**: The input XcodeML/F IR is manipulated according to the rules of the transformations. It produces the output XcodeML IR.
4. **F_Back**: The output XcodeML/F IR is analyzed to produce the transformed Fortran code.
5. **C_Back**: The output XcodeML/C IR is analyzed to produce the transformed C code.

`F_Front`, `F_Back`, `C_Back` are part of the OMNI Compiler, `FPP` is provided by the installed Fortran compiler. The XcodeML/F to XcodeML translator `CX2T` is actively developed as part of the CLAW project.

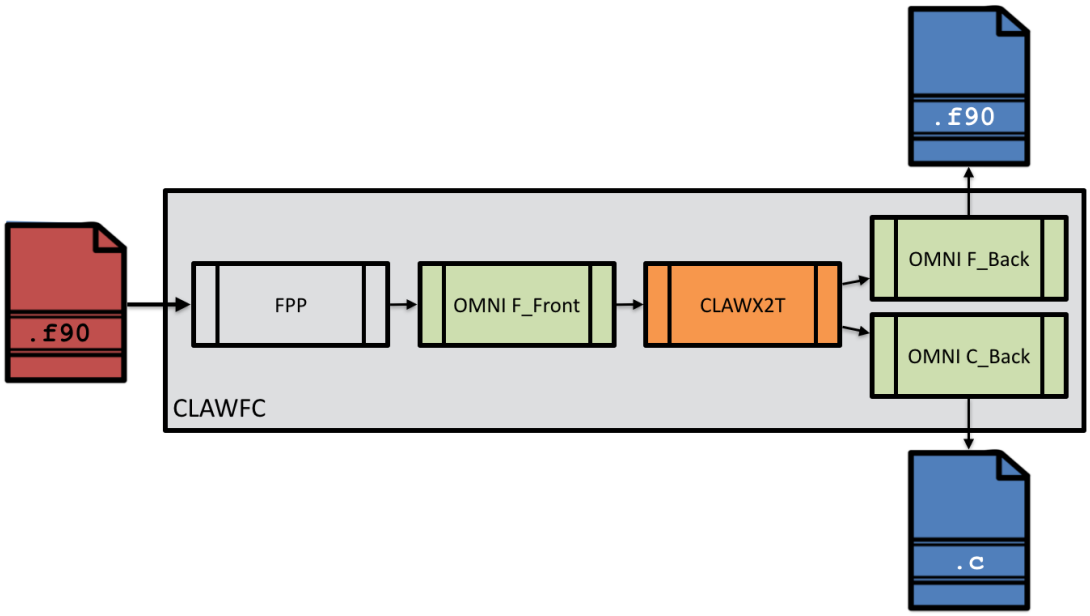

Figure 1.1: CLAW Compiler main workflow.

1.3 CLAW XcodeML/F to XcodeML translator (CX2T)

The XcodeML/F to XcodeML translator is the intelligence behind the CLAW Compiler. It understands the CLAW directive language, generates directive and translation unit transformation instances and and apply them to the to the XcodeML AST.

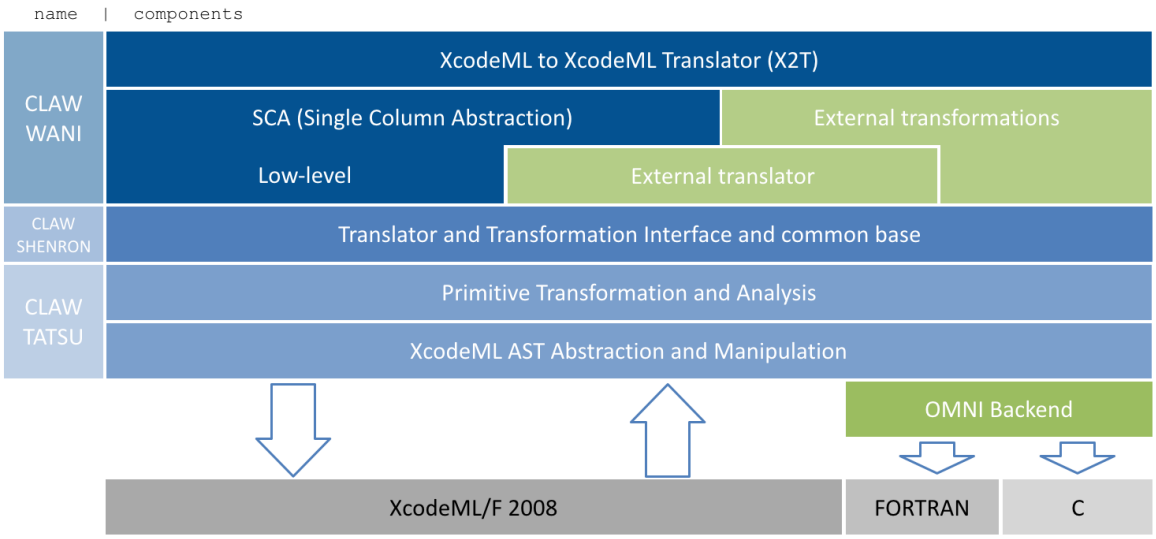

Figure 1.2: CLAW XcodeML/F Translator library stack.

Figure 1.2 shows the stack of libraries used in the the CLAW XcodeML/F Translator. All blue blocks are actively developed as part of the CLAW project. Green part are external libraries coming from the OMNI Compiler project or from other projects using the global CLAW Compiler infrastructure.

Chapter 2

CLAW Directive Language

The CLAW Directive Language is specified in the *CLAW Directive Language Specification* document [5]. To parse and validate this directive language, a parser is needed in the heart of the CLAW Compiler. This section gives an in-depth description of this parser and its implementation.

2.1 CLAW directive language parser

The CLAW Directive Language parser is based on the ANTLR project [6]. ANTLR is a parser generator. From a grammar file, ANTLR generates a parser that is then used in the CLAW XcodeML/F to XcodeML translator to interpret the directives.

```
1  grammar Claw;
2
3  @header
4  {
5  import cx2x.translator.language.base.ClawLanguage;
6  }
7
8  /*-----------------------------------------------------------------------
9   * PARSER RULES
10  *-----------------------------------------------------------------------*/
11 analyze returns [ClawLanguage l]
12   @init{
13     $l = new ClawLanguage();
14   }
15   :
16     CLAW ACC EOF
17     { $l.setDirective(ClawDirective.PRIMITIVE); }
18   | CLAW OMP EOF
19     { $l.setDirective(ClawDirective.PRIMITIVE); }
20 ;
21
22 /*-----------------------------------------------------------------------
23  * LEXER RULES
24  *-----------------------------------------------------------------------*/
25 CLAW          : 'claw';
26 ACC           : 'acc';
27 OMP           : 'omp';
28 IDENTIFIER    : [a-zA-Z_$] [a-zA-Z_$0-9-]* ;
```

Listing 2.1: ANTLR Grammar Example

Listing 2.1 is a minimalist ANTLR example that support only two directives `!$claw acc` or `!$claw omp`. In this grammar, there are two important sections, the parser and the lexer sections. The lexer section defines what will be recognized as a token. Here we defined the directives, the clauses but also more complex construct such as the `IDENTIFIER` on line 28. It described the possible element to compose an identifier. This `IDENTIFIER` can then be used later in the parser rules. The parser rules define the actual grammar of the language. In the case of this small example, the grammar says that the directive string must begins with `claw` and then be followed by `acc` or `omp`. The code in `{}` is java code that is triggered when the grammar rule is activated. In the example, it sets a value of the object returned after the analysis. The `@init{}` section allows to execute some Java code before triggering the actual parse. The `@header{}` allows to insert Java code before the parser class.

```
1 javac -classpath <antlr_jar> org.antlr.v4.Tool -o . -package cx2x.translator.
      language.parser Claw.g4
```

Listing 2.2: ANTLR parser generation command

The full CLAW ANTLR grammar is defined in the following file: `cx2t/src/claw/wani/language/parser/Claw.g4`

Chapter 3

CLAW Compiler Configuration

The different transformation enabled in the CLAW Compiler and their order of application is configurable in the CLAW Compiler global configuration file. Each transformation might also have different options that can be activated via the configuration file. The available transformations are defined in transformation set files (see Section 3.1).

3.1 Transformation Set

A transformation set is a group of transformations that are packed together and that can be used during the translation. Each transformation set must be described in a transformation set configuration file once. Listing 3.1 shows the transformation set configuration file for the CLAW internal transformation set.

```
1 <transformations>
2   <transformation name="internal-xcodeml"
3     type="independent" trigger="translation_unit"
4     class="cx2x.translator.transformation.utility.XcodeMLWorkaround" />
5
6   <transformation name="openacc-continuation"
7     type="independent" trigger="directive"
8     class="cx2x.translator.transformation.openacc.OpenAccContinuation" />
9 </transformations>
```

Listing 3.1: CLAW internal transformation set configuration file

A transformation set configuration file defines several characteristics of each transformation defined into it:

- **name**: A unique name given to the transformation. This name is used when defining the activation and the application order of the transformation.
- **type**: Type of the transformation as describe in Section 4.1. Only two values are possible `dependent` or `independent`.
- **trigger**: How the transformation is triggered as described in Section 4.2. Only two values are possible `directive` or `translation_unit`.
- **dircetive**: When `directive` value is defined for the trigger option, this define which directive triggers it (e.g. `acc`, `omp` or `claw`).
- **class**: Fully qualified class name implementing the transformation.

3.1.1 External Transformation Set

External transformations can be implemented separately from the CLAW Compiler infrastructure and plugged into it easily. Section 4.4 explains in detail how to create and plug external transformations.
External transformations must be defined in an external transformation set configuration file. This transformation set must specify the name of the JAR file containing the transformation classes in the jar attribute of the `transformations` element as shown in Figure 3.2.

```
1 <transformations jar="claw-external-set.jar">
2   <transformation name="add-print"
3     type="independent" trigger="translation_unit"
4     class="external.transformation.AddPrint"/>
5 </transformations>
```

Listing 3.2: External transformation set configuration file

The JAR files specified in external transformation sets can be located anywhere on disk. These location have to be set in the `CLAW_TRANS_SET_PATH` environment variable. Several paths can be provided and they must be separated by `:`.

3.2 CLAW Compiler Global Configuration

The global configuration gives information about which transformation sets are used, which transformation are enabled, in which order the transformations are applied and several configuration key-value pair that can be accessed by the transformations them-self.
The configuration is separated in different sections:

- **global**: this section encapsulates several key-value pair configuration parameters. The attribute `type` of the global element defines if the configuration is a root configuration or an extension of the default one. User can overwrite partially or entirely the default configuration. The only mandatory key-value pair in the global section is the `translator` key.
- **sets**: this section defines which transformation sets are used. The name corresponds to the filename of the transformation set configuration file without extension.
- **groups**: this section defines which transformations are enabled. The name corresponds to the name defined in each transformation set configuration file. Only transformations part of the used transformation sets can be defined here. The order is defined by the order of declaration from top to bottom.

3.2.1 Default Configuration

```
1 <claw version="1.0">
2   <!-- Global transformation parameters -->
3   <global type="root">
4     <!-- Define the translator to be used. -->
5     <parameter key="translator" value="cx2x.translator.ClawTranslator" />
6
7     <!-- Default general values -->
8     <parameter key="default_target" value="gpu" />
9     <parameter key="default_directive" value="openacc" />
```

```
10    </global>
11
12    <!-- Transformation sets -->
13    <sets>
14      <set name="claw-internal-set" />
15      <set name="claw-low-level-set" />
16      <set name="claw-high-level-set" />
17    </sets>
18
19    <!-- Transformation groups and order -->
20    <groups>
21      <!-- Low-level transformations -->
22      <group name="remove" />
23      <group name="directive-primitive" />
24      <group name="array-transform" />
25      <group name="loop-extract" />
26      <group name="loop-hoist" />
27      <group name="loop-fusion" />
28      <group name="loop-interchange" />
29      <group name="on-the-fly" />
30      <group name="kcache" />
31      <group name="if-extract" />
32      <!-- High-level transformations -->
33      <group name="parallelize" />
34      <group name="parallelize-forward" />
35      <!-- internal applied at the end -->
36      <group name="openacc-continuation" />
37    </groups>
38  </claw>
```

Listing 3.3: CLAW Compiler default configuration

3.2.2 User Defined Configuration

Listing 3.4 is an example of a user configuration that can be passed to the CLAW Compiler. It uses one transformation set and enable one transformation only. The transformation set is the one described in Listing 3.2. This configuration will overwrite entirely the default configuration as its `type` is `root`.

```
1  <claw version="1.0">
2    <global type="root">
3      <parameter key="translator" value="cx2x.translator.ClawTranslator"/>
4    </global>
5    <sets>
6      <set name="claw-external-set"/>
7    </sets>
8    <groups>
9      <group name="add-print"/>
10   </groups>
11 </claw>
```

Listing 3.4: Example of user root configuration

Listing 3.5 is an example of a user extension configuration. Extension configuration overwrite only partially the default configuration. In this example, only the `groups` element is overwritten with a new order for transformation.

```
<claw version="1.0">
  <global type="extension"/>
  <groups>
    <group name="internal-xcodeml"/>
    <group name="remove"/>
    <group name="directive-primitive"/>
    <group name="array-transform"/>
    <group name="loop-extract"/>
    <group name="loop-fusion"/>
    <group name="loop-hoist"/>
    <group name="loop-interchange"/>
    <group name="on-the-fly"/>
    <group name="kcache"/>
    <group name="if-extract"/>
    <group name="parallelize"/>
    <group name="parallelize-forward"/>
  </groups>
</claw>
```

Listing 3.5: Example of user extension configuration

Chapter 4

Transformation

A transformation is the the basic representation of a manipulation of the AST triggered by a directive or for the whole translation unit. Each translation unit transformation or directive transformation is implemented in its own transformation class. If the directive can be used as a block directive (with a `!$end <directive>` directive), the transformation class must inherits from the `ClawBlockTransformation` super class. Otherwise, it can inherits from the basic `ClawTransformation` super class.

4.1 Type of transformation

Transformations are divided into two distinct groups. The independent and the dependent transformations. The first one, as its name implies, is performed independently of any other transformations. Translation unit transformations are always independent. The dependent transformation, in the other hand, is applied only when it can be combined with another dependent transformation of the same kind in its group. Most of the transformations are independent. The best example for the dependent transformation is the loop fusion transformation. As shown in Listing 4.1, there are two CLAW directives (line 4 and 9). These directives will both trigger a dependent loop fusion transformations. Alone, those transformations have no effect on the AST. The translation engine will analyze if the first one can be transformed with the second one. If so, the transformations will be grouped together and the fusion will be done. Otherwise, the transformations are just ignored as they have to depend on at least one other transformation. Checks that match dependent transformation together are specific to the kind of transformation.

```
1  PROGRAM loop_fusion
2    INTEGER :: i
3
4    !$claw loop-fusion
5    DO i = 1, 10
6      ! Loop body
7    END DO
8
9    !$claw loop-fusion
10   DO i = 1, 10
11     ! Loop body
12   END DO
13 END PROGRAM loop_fusion
```

Listing 4.1: Loop fusion example

4.2 Trigger transformation

Transformations can be triggered by a directive or they can be applied for each translation unit independently from the directives present in it or not. This information is specified in the configuration file of the transformation set (see Section 3.1). Directive triggered transformations must implement the constructor passing the `ClawPragma` parameter.

4.3 Transformation application order

The first step in the translation of a translation unit (an XcodeML IR), is the detection of all the directives. Each directive will trigger the creation of an instance of the transformation class it belongs to, if any. For example, a loop-fusion directive will trigger the creation of a `LoopFusion` instance. On this instance, the `analyze()` method is called in order to determine if the transformation can take place. If the analysis step is successful, the transformation is added to its transformation group. Once all the transformation instances have been analyzed and categorized by groups, the actual code transformation can take place on the XcodeML IR. All transformations in a group are applied one after another in a *fifo* order. The order in which groups are processed is determined by the configuration file.

```
<claw version="1.0">
  <!-- Global transformation parameters -->
  <global type="root">
    <!-- Define the translator to be used. -->
    <parameter key="translator" value="cx2x.translator.ClawTranslator" />

    <!-- Default general values -->
    <parameter key="default_target" value="gpu" />
    <parameter key="default_directive" value="openacc" />
  </global>

  <!-- Transformation sets -->
  <sets>
    <set name="claw-internal-set" />
    <set name="claw-low-level-set" />
    <set name="claw-high-level-set" />
  </sets>

  <!-- Transformation groups and order -->
  <groups>
    <!-- Low-level transformations -->
    <group name="remove" />
    <group name="directive-primitive" />
    <group name="array-transform" />
    <group name="loop-extract" />
    <group name="loop-hoist" />
    <group name="loop-fusion" />
    <group name="loop-interchange" />
    <group name="on-the-fly" />
    <group name="kcache" />
    <group name="if-extract" />
    <!-- High-level transformations -->
    <group name="parallelize" />
    <group name="parallelize-forward" />
    <!-- internal applied at the end -->
    <group name="openacc-continuation" />
  </groups>
</claw>
```

Listing 4.2: CLAW default configuration

As shown in Figure 4.2, the transformation order is specified the configuration file under the XML format. Each transformation group name refers to its name in the transformation set it belongs to. Global configuration and transformation set file are described in details in Chapter 3.

4.4 Add a new transformation

A transformation in the `clawfc` is represented as a class that inherits from the `ClawTranformation` or `ClawBlockTransformation` super class. In order to add a new transformation into the the CLAW Compiler, the following steps must be done.

1. Create a new class that inherits from one of the transformation super class.
2. *(Optional)* Define the directive that will trigger the transformation and add it to the CLAW directive language parser if the transformation is triggered by a directive.
3. *(Optional)* Detect and categorize the new transformation if the transformation is triggered by a directive in the translator.
4. Add the transformation to a transformation set configuration and to the global configuration.

As an example, the next 4 subsection describe those steps with more details.

4.4.1 New transformation class

The transformation created at this step will be a simple independent transformation triggered by a directive. It will then inherits from `ClawTransformation` super class.

```
1  /*
2   * This file is released under terms of BSD license
3   * See LICENSE file for more information
4   */
5  package cx2x.wani.transformation;
6
7  import cx2x.wani.language.ClawPragma;
8  import cx2x.wani.transformation.ClawTransformation;
9  import cx2x.shenron.transformation.Transformation;
10 import cx2x.shenron.translator.Translator;
11 import claw.tatsu.xcodeml.xnode.common;
12
13 /**
14  * Simple transformation for documentation example
15  */
16 public class MyFirstTransformation extends ClawTransformation {
17
18   // Constructor that received the analyzed pragma as argument
19   public MyFirstTransformation(ClawPragma directive) {
20     super(directive);
21   }
22
23   // The analyzis step
```

```
24    public boolean analyze(XcodeProgram xcodeml, Translator translator) {
25      return true;
26    }
27
28    // The transformation step
29    public void transform(XcodeProgram xcodeml, Translator translator,
30                          Transformation other) throws Exception
31    {
32      removePragma();
33    }
34
35    // Only used by dependent transformation
36    public boolean canBeTransformedWith(Transformation other) {
37      return false; // Independent transformation
38    }
39  }
```

Listing 4.3: MyFirstTransformation.java

The transformation class shown in Listing 4.3 is really simple. It just inherits from `ClawTransformation` on line 16. Therefore, it has to implements the `analyze()`, `transform()` and `canBeTransformedWith()` methods as they are abstract in the super class.
The `analyze()` method does not perform any check and just return `true` to tell the translation engine that the transformation can be applied.
The `transform()` method is pretty simple. It delete the pragma that triggered the transformation from the AST. It will then not be in the resulting transformed code.

4.4.2 Directive and CLAW directive language parser

If the new transformation class is triggered by a directive, it needs to be set up in the directive language parser in order to be instantiated. Listing 4.4 is the directive that is used in the current example.

```
1 !$claw mydirective
```

Listing 4.4: Example directive

To parse the directive, two files need to be modified. First, the new directive needs to be added to the `ClawDirective` enumeration. It is added as the last element of the enumeration in the Listing 4.5.

```
1  package cx2t.wani.language;
2
3  public enum ClawDirective {
4    ARRAY_TRANSFORM,
5    ARRAY_TO_CALL,
6    DEFINE,
7    IGNORE,
8    KCACHE,
9    LOOP_FUSION,
10   LOOP_INTERCHANGE,
11   LOOP_HOIST,
12   LOOP_EXTRACT,
13   PRIMITIVE,
14   PARALLELIZE,
```

```
15    REMOVE,
16    VERBATIM,
17    MYDIRECTIVE
18  }
```

Listing 4.5: ClawDirective.java

Then, the CLAW directive language parser has to be modified to understand this new directive and return the correct value from the enumeration.

```
1  grammar Claw;
2
3  @header
4  {
5  import cx2t.wani.ClawConstant;
6  import cx2t.wani.language.*;
7  import cx2t.tatsu.common.*;
8  }
9
10 /*----------------------------------------------------------------------------
11  * PARSER RULES
12  *----------------------------------------------------------------------------*/
13
14 // Entry point of the parsing
15 analyze returns [ClawPragma cp]
16   @init{ $cp = new ClawPragma(); }:
17   CLAW directive[$cp] EOF
18 ;
19
20 directive[ClawPragma cp]:
21   // The new directive
22   MYDIRECTIVE { $cp.setDirective(ClawDirective.MYDIRETIVE); }
23 ;
24
25 /*----------------------------------------------------------------------------
26  * LEXER RULES
27  *----------------------------------------------------------------------------*/
28
29 // Start point
30 CLAW          : 'claw';
31 MYDIRECTIVE   : 'mydirective';
```

Listing 4.6: Claw.g4

In Listing 4.6, there is a minimal version of the ANTLR grammar file of the CLAW directive language parser. The token `mydirective` is added in the lexer rules (line 31) and is then used in the parser rules (line 22). ANTLR grammar file accepts Java code to be inserted during the parsing. On line 22, the enumeration value for the new directive is set in the analyzed pragma object.

4.4.3 Detection and categorization

The new directive has its transformation class and can be analyzed by the CLAW directive language parser. Now, it needs to be detected in order to create the transformation instance. This happens in the `ClawTranslator` class. The `generateTransformation()` method detects all the pragmas in the translation unit and categorize them.

As shown in Listing 4.7, a new case is added to the switch statement that categorize the transformation. Here, a new instance of `MyFirstTransformation` transformation is created and added to the correct group as long as its analysis step succeed.

```
1 switch(analyzedPragma.getDirective()) {
2   case MYDIRECTIVE:
3     addTransformation(xcodeml, new MyFirstTransformation(analyzedPragma));
4     break;
5 }
```

Listing 4.7: ClawTranslator.java

4.4.4 Enable the new transformation

Finally, the new transformation needs to be enabled by adding it to the configuration file.
Listing 4.8 shows the transformation set configuration including the new transformation class.

```
1 <transformations>
2   <transformation name="delete-pragma" type="independent" trigger="directive"
3     class="cx2t.wani.transformation.MyFirstTransformation"/>
4 </transformations>
```

Listing 4.8: claw-dummy-set.xml

Listing 4.9 uses the transformation set from Listing 4.8 and enables its only transformation.

```
1 <claw version="1.0">
2   <global type="extension"/>
3   <sets>
4     <set name="claw-dummy"/>
5   </sets>
6   <groups>
7     <group name="delete-pragma"/>
8   </groups>
9 </claw>
```

Listing 4.9: claw-dummy.xml

4.4.5 Test the new transformation

The last step is to compile and install the new version of the CLAW Compiler containing the new directive and transformation class. Once it is done, it can be directly tested with a simple Fortran program.
Listing 4.10 is the original code. Once it is transformed by the CLAW Compiler, it will produce the code in Listing 4.11. The command to call the CLAW Compiler is shown in Listing 4.12.

```
1 program mydirective_test
2   integer :: i
3   !$claw mydirective
```

```
4   do i=1,10
5     print *, 'First loop body:',i
6   end do
7 end program mydirective_test
```

Listing 4.10: original.f90

```
1 PROGRAM loop_fusion
2   INTEGER :: i
3
4   DO i = 1 , 10 , 1
5    PRINT * ,"First loop body:" , i
6   END DO
7 END PROGRAM loop_fusion
```

Listing 4.11: transformed.f90

```
1 $ clawfc -o transformed.f90 original.f90
```

Listing 4.12: Call the compiler

Chapter 5

XcodeML AST abstraction and manipulation library

As the XcodeML IR is based on the XML format, it can be manipulated with any language that can read and write XML files. It can even be manipulated by hand depending on the user's knowledge of the XcodeML IR specification. To ease this task, an AST abstraction and manipulation library is included in the `CX2T` program. This library helps to traverse the AST, access and update the type table and add, modify or delete nodes.
This section aims to describe very simple examples. For more information about the libraries, the complete javadoc[7, 8] can be found online.

5.1 The Xnode class

The main class of this library is the `Xnode` class that is the base of any node in the AST. Any specialization of a node like `XcodeML`, `XcodeProgram`, `Xmod` or `XbasicType` inherits from `Xnode`. This class has basic methods to perform the following actions:

- Query, add, update or remove an attribute of a node (See Listing 5.1).
- Query, add or remove children of a node (See Listing 5.2).
- Clone a node (See Listing 5.3).
- Match nodes in the siblings, descendants or ancestor of a node (See Listing 5.4).

Each `Xnode` instance has a specific opcode that matches the XcodeML/F element tag. This opcode is also used when creating new nodes or matching nodes in a tree. All the opcodes are members of the `Xcode` class. The `Xattr` class contains all the members to access and match attributes of a node.

```
1  // Gather a node
2  Xnode n = ...
3  // Check if a node has an attribute
4  boolean hasAttributeAllocatable = n.hasAttribute(Xattr.IS_ALLOCATABLE);
5  // Get value from a boolean attribute
6  boolean attributeAllocatable = n.getBooleanAttribute(Xattr.IS_ALLOCATABLE);
7  // Get value from a boolean attribute
8  String type = n.getAttribute(Xattr.TYPE);
9  // Set value of a boolean attribute
10 n.setAttribute(Xattr.IS_ALLOCATABLE, true);
11 // Set value of an attribute
```

```
12 n.setAttribute(Xattr.TYPE, "VALUE_OF_TYPE");
13 // Remove an attribute
14 n.removeAttribute(Xattr.IS_ALLOCATABLE);
```

Listing 5.1: Xnode attribute methods

```
1  // Gather a node
2  Xnode n = ...
3  // Get first child
4  Xnode fChild = n.firstChild();
5  // Get last child
6  Xnode lChild = n.lastChild();
7  // Iterate through children
8  Xnode crt = n.firstChild();
9  while(crt != null) {
10   crt = crt.nextSibling()
11 }
12 // Add a child
13 Xnode intConst = xcodeml.createIntConstant(10);
14 n.append(xcodeml.createNode(intConst);
```

Listing 5.2: Xnode node methods

```
1 // Gather a node
2 Xnode n = ...
3 // Get first child
4 Xnode clone = n.cloneNode();
```

Listing 5.3: Xnode cloning

```
1 // Gather a node
2 Xnode n = ...
3 // Match all arrayRef nodes in sub-tree
4 List<Xnode> arrayRefs = n.matchAll(Xcode.FARRAYREF);
5 // Match all arrayRef nodes in the siblings of the node
6 List<Xnode> arrayRefs = n.matchSibling(Xcode.FARRAYREF);
7 // Match first arrayRef node in the direct descendant of the node
8 Xnode arrayRef = n.matchDirectDescendant(Xcode.FARRAYREF);
```

Listing 5.4: Xnode match

5.2 Access the type table

The type table of an XcodeML/F translation unit is a child node of the root. The `XcodeML` class gives access to this type table. `XcodeProgram` and `Xmod` are child classes of `XcodeML`. In the example in Listing 5.5, the code is checking every function definition to find declaration of allocatable arrays. On line 9, the type table is accessed to check if the type of the declaration is `FbasicType`. If so, the type is retrieved from the type table (line 12) and its information can be queried to check its nature.

```
1  public boolean analyze(XcodeProgram xcodeml, Translator translator) {
2    // Get all function definitions in the translation unit
3    List<XfunctionDefinition> fctDefs = xcodeml.getAllFctDef();
4    for(XfunctionDefinition fctDef : fctDefs) {
5      // Get the declarations of the function
6      List<Xnode> declarations = fctDef.getDeclarationTable().values();
7      for(Xnode decl : declarations) {
8        if(decl.opcode() == Xcode.VARDECL) {
9          if(!xcodeml.getTypeTable().isBasicType(decl)) {
10           continue; // Only FbasicType can be array
11         }
12         XbasicType bt = xcodeml.getTypeTable().getBasicType(decl);
13         if(bt != null && bt.isArray() && bt.isAllocatable()) {
14           System.out.println(decl.decl.matchSeq(Xcode.NAME).value());
15         }
16       }
17     }
18   }
19 }
```

Listing 5.5: Access the type table

5.3 Traverse the AST

Listing 5.6 shows a simple way to go from a pragma and visit its next siblings.

```
1 public void transform(XcodeProgram xcodeml, Translator translator,
2                       Transformation other)
3 {
4   Xnode crt = _claw.getPragma();
5   while(crt != null){
6     System.out.println("Current node: " + crt.opcode());
7     crt = crt.nextSibling();
8   }
9 }
```

Listing 5.6: XcodeML AST traverse

5.4 Add nodes

A new node can be easily added in the AST as shown in the Listing 5.7. On line 4, the new "node" object is created by passing the "opcode" of the node. This code refers the the XcodeML/F specification defined in the `Xcode` enumeration.
On line 5, a new value is assigned to the node. `FpragmaStatement` node accept text values.
Finally, on line 6, the node is inserted in the AST. In this case, the `insertAfter` will add the new node after the current node.

```
1 public void transform(XcodeProgram xcodeml, Translator translator,
2                       Transformation other)
3 {
```

```
4   Xnode myNewNode = xcodeml.createNode(Xcode.FPRAGMASTATEMENT);
5   myNewNode.setValue("omp parallel");
6   _claw.getPragma().insertAfter(myNewNode);
7 }
```

Listing 5.7: XcodeML add node example

5.5 Modify nodes

```
1 public void transform(XcodeProgram xcodeml, Translator translator,
2                       Transformation other)
3 {
4   Xnode pragma = _claw.getPragma();
5   // Get current values from the node
6   String oldValue = pragma.value();
7   int line = pragma.lineNo();
8   // Update the value directly in the AST
9   pragma.setValue("acc routine seq");
10  pragma.setLineNo(line + 1);
11 }
```

Listing 5.8: XcodeML update node example

5.6 Delete nodes

```
1 public void transform(XcodeProgram xcodeml, Translator translator,
2                       Transformation other)
3 {
4   Xnode pragma = _claw.getPragma();
5   XnodeUtil.safeDelete(pragma); // Delete the node in the AST
6 }
```

Listing 5.9: XcodeML delete node example

List of figures

Listings

Bibliography

[1] Programming Environment Research Team at RIKEN AICS & HPCS Lab. at University of Tsukuba, "The omni compiler project," 1992-2016. [Online; accessed 2015-10-15].

[2] Programming Environment Research Team at RIKEN AICS & HPCS Lab. at University of Tsukuba, "Omni compiler - git repository," 2016. [Online; accessed 2016-12-21], url: https://github.com/omni-compiler/omni-compiler.

[3] Programming Environment Research Team at RIKEN AICS & HPCS Lab. at University of Tsukuba, *XcodeML/Fortran 95 specification*, 01 2011. ver. 0.91J, url: http://omni-compiler.org/xcodeml.html.

[4] Programming Environment Research Team at RIKEN AICS & HPCS Lab. at University of Tsukuba, *XcodeML/Fortran 2008 specification*, 07 2017. ver. 1.0, url: https://github.com/omni-compiler/XcodeML-Specification.

[5] Center for Climate System Modeling, *CLAW language specification*, 07 2017. v0.4.0, url: https://github.com/claw-project/claw-language-specification.

[6] T. Parr, *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd ed., 2013.

[7] Center for Climate System Modeling, *CLAW Online Documentation*, 03 2018. url: https://claw-project.github.io/claw-documentation/.

[8] Center for Climate System Modeling, *CLAW X2T Javadoc*, 03 2018. url: https://claw-project.github.io/claw-documentation/javadoc/index.html.