YaCF: Yet Another Compiler Framework

Technical Report 02/2013

http://cap.pcg.ull.es/es/system/files/private/YaCF.pdf

February 2013

Ruymán Reyes*
Francisco de Sande
Universidad de La Laguna
fsande@ull.es

Abstract

YaCF, *Yet another Compiler Framework*, is a source to source (StS) translator framework designed to create source to source translators, code analysis tools or just to teach compiler technology without the need of learning large pieces of code.

Taking advantage of Python introspection capabilities and its inherent code flexibility, using YaCF this kind of tools can be built with lower effort.

The YaCF translation system has been designed to ease the writing of these source transformations or manipulations. Using a set of patterns, based on widely known object oriented patterns, implementing code transformation is only a matter of writing a few lines of code.

YaCF has been designed to ease the burden on compiler writers. Its components are independent from each other and can be used in full source to source drivers or in small test transformations. Several subclasses, modules and packages have been included within YaCF to solve particular problems within source-to-source (StS) code translations.

Keywords: Compiler, Productivity, Python, OpenACC, accULL, Accelerators, GPGPU, CUDA, OpenCL

*Readers may contact the authors through Francisco de Sande, Departamento de EIO y Computación, Office 87, Universidad de La Laguna, La Laguna, S/C de Tenerife 38270, +34 922318178, fsande@ull.es.

1. Motivation

In the High Performance Computing Group at La Laguna University [1] we have been working for more than ten years in the field of *Directive-based programming for High Performance Computing* [11, 12, 31, 9, 13, 10]. Since the very begining, the guideliness of our research has always been in the direction of narrowing the distance between HPC users and tools. We want to provide to the scientific community with high-level programming tools to leverage the development coding effort [20].

In our opinion, the lack of general purpose high level parallel languages is a major drawback that limits the spread of High Performance Computing (HPC). There is a division between the users who have the needs of HPC techniques and the experts that design and develop the languages as, in general, the users do not have the skills necessary to exploit the tools involved in the development of the parallel applications. Any effort to narrow the gap between users and tools by providing higher level programming languages and increasing their simplicity of use is thus welcome.

Compiler support is required to implement most of the higher-level programming models. The design and implementation of a compiler is not a trivial task, it requires tremendous amounts of work and significant amounts of patience to deal with developing quirks and hints in production codes. Some of the aforementioned programming models rely on the support of a commercial or an experimental compiler which is described in this section.

The extension of languages through the usage of directives requires a broad knowledge of compiler technologies and techniques. The previous work used ad-hoc compilers to extend a language with a set of features. Given the speed at which the HPC field is advancing, and the amount of new languages that are being developed, a significant amount of development and effort would be required to design ad-hoc compilers to cover specific languages or to extend features, and we would be unable to focus our attention on adding new features or investigating new research guidelines.

For example, with the irruption of GPU in the HPC arena [25], we decided to study different possible annotation schemes or language extensions to facilitate the automatic or directed generation of GPU code. Our previous `llCoMP` compiler was unsuitable for this task as it was designed ad-hoc to extend the OpenMP language with a particular set of features. With the experience adquired from working on `llCoMP`, and having comprehensively reviewed the bibliography, we detected the four key characteristics that we require in a research compiler in order for it to meet our needs:

- A flexible parser: We wanted to explore several different annotation schemes, language extensions and idioms, thus, a flexible front end where this modifications could be done quickly was our priority. As our work was for experimental purposes, we did not intent to parse commercial codes; our priority was not stability (i.e. speed and memory usage were not constraints).

- Portability: It should be possible to use the compiler on several different platforms, from laptops to clusters. In addition, different users, such as students or collaborators, should be able to use it without having to invest too much time and energy in learning how to use it. The compiler should be easily movable from one machine to another and it has to be written in a common and portable language.

- Debuggability: The user needs to be able to run the StS process step by step or be able to show what each phase is doing at any given point. One of the potential uses of this

compilation framework is to teach compiler technology, thus it is desirable to be able to review or stop any process of the translation, so the user can easily see what is going on. As such, the inclusion of a graphical visualization of the Abstract Syntax Tree (AST) at any given moment would be a plus.

- Simplicity: Powerful but simple Object Oriented approach.

2. Background and Related Work

Before embracing ourselves with the titanic task of building our own compiler and runtime infrastructure, we evaluated different options and extensively reviewed the works in the bibliography.

The most relevant compiler infrastructures are detailed in the following paragraphs, and at the end of this section we conclude by providing a table comparing each of the aforementioned key features.

2.1 GCC

The GNU compiler collection (GCC) [16] is a compiler developed by the GNU project that supports various programming languages. It has been ported to a wide variety of processor architectures and is deployed in several different machines and platforms. It supports C, C++, Objective-C, Fortran, Java, Ada, Go and many others. Several tools are required for its construction (e.g. Perl, Flex, Bison, GMP, MPC ...), and some optimization passes are only enabled if external libraries are available. It can be extended with plugins, which can operate on the intermediate representation (GIMPLE).

Each of the language compilers is a separate program that inputs source code and outputs machine code. All have a common internal structure: a per-language front end parsers the source code in that language and produces an AST.

These are converted to the middle-end representation, which is gradually lowered towards its final, low-level form. GCC is mainly written in C.

2.1.1 General Usage/Workflow

GNU is organised into several passes. Language front end is invoked only once in order to parse the entire input, and may use different intermediate representations and language-specific tree codes. After finishing the parsing, the front end must translate the representation used to a representation understood by the language independent portions of the compiler. The C compiler calls the Gimplifier, while the Fortran translates its internal representation to GENERIC and then following this it is then lowered to GIMPLE. The conversion from GENERIC to GIMPLE is called *Gimplification*. After the code is in the language-independent IR, it is possible to call to different passes that are handled by the pass-manager. This package is in charge of running all of the individual passes in the correct order. Passes create several different parallel structures (like the control flow graph or the handlers for the OpenMP expansion). After the tree-optimization phases are completed, the code is transformed to RTL so register optimizations can be applied.

2.1.2 Intermediate Representation

GCC uses three different IR: GENERIC, GIMPLE and RTL. Front-end modules generate code into GENERIC, a High Level internal representation, which is then translated to a more manageable IR called GIMPLE.

GIMPLE is a family of IR based on the tree data structure. Two levels of this IR are implemented: High-Level GIMPLE - produced by the middle end when lowering the GENERIC language that is targeted by all the language front ends; and Low-Level GIMPLE - obtained by linearizing all the high-level control flow structures of high-level GIMPLE.

GIMPLE is then lowered to RTL (Register Transfer Language), which is an assembler language for an abstract machine with infinite registers. It represents low-level features (registers, memory addressing, bitfield operations, compare-and-branch ...), and it is commonly represented in a LISP-like form.

2.1.3 Unparsing

GCC does not offer unparsing features by itself, as it is meant to be a full source-to-binary compiler. Undoing the parsing could be possible at GENERIC level, but several semantic details are lost after translating to GIMPLE. It may be possible to obtain a C-like representation of GIMPLE by using the `-fdump-tree-gimple` flag, which is useful for debugging purposes. However, a more complete method is required for implementing a source-to-source translation system.

2.2 Open64

Open64 [6] is an open-source multi-platform compiler, derived from the SGI compilers. It was released as GPL in 2000, after which the University of Delaware took care of the project.

2.2.1 General Usage/Workflow

The compilation flow of Open64 is shown in Figure 1. Each step of the compilation flow is followed by a lowering of the IR (see Section 2.2.2). Open64 is a full source-to-binary compiler, thus, the end-point of the flow is assembler code suitable for generating the final binary.

The C and C++ front ends are based on GNU technology, while the Fortran one is the SGI Pro64 Fortran front end. Both provide a very high level IR for the input program units, stored as `.B` files. The VHO operates on this file (VHL level) to generate the phases that follow.

The LNO (Loop Nest Optimisation) module features several transformations to improve code performance by taking advantage of the Data Cache, e.g. transforming loop to work on sub-matrices that fit in the cache (Cache Blocking), loop interchange, etc. LNO also simplifies the expressions to facilitate the tasks of the following steps, generates SIMD code and vector intrinsics and also leverages OpenMP directives into intermediate code.

A driver controls the execution of the compiler, deciding which modules to load and executes the compilation plan. The driver is responsible for invoking all steps, and managing the modules input flags. Communication between modules is performed using intermediate temporary files.

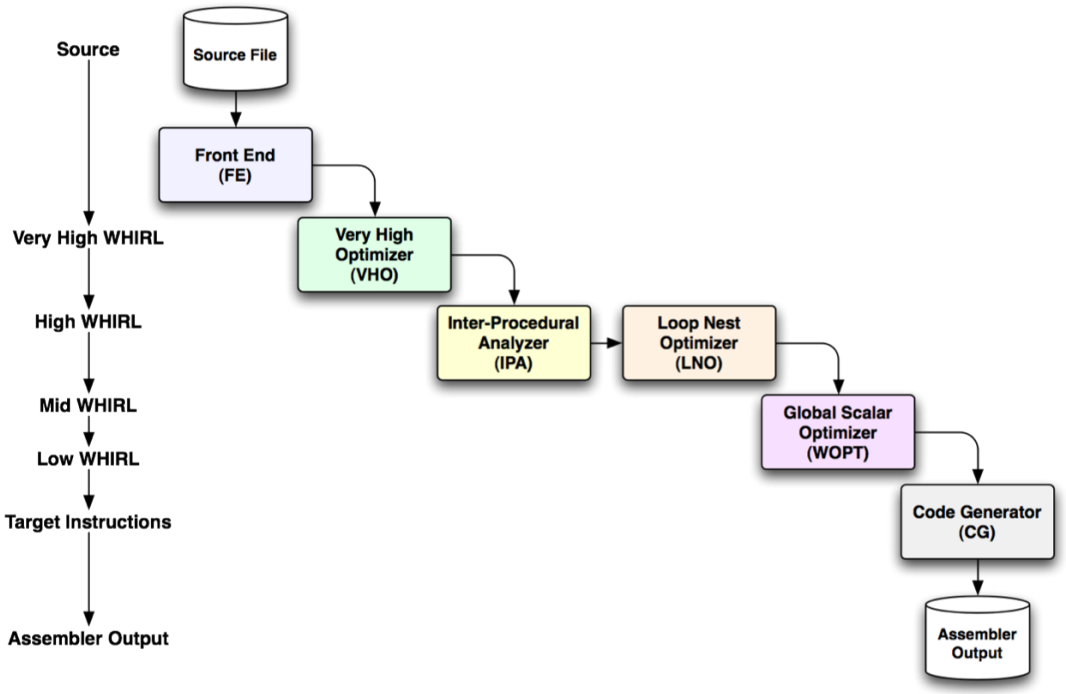

Figure 1

Compilation Flow of Open64

2.2.2 Intermediate Representation

The IR is called WHIRL, and it is composed of 5 levels (from the closest to the original to the closest to the binary): Very High (VH), High (H), Mid (M), Low (L), and Very Low (VL) level. The front end translate the original file into WHIRL which is then passed to the back end. Each optimisation is designed to work on a particular level of WHIRL, and WHIRL *lowerers* are called to translate WHIRL from the current level to the next lower level. Finally, the code generator translates the lowest level of WHIRL to its own internal representation that matches the target instruction. Because lowering is done only gradually, each lowering step is simpler and easier, which decreases the overall complexity of the translation. A description of the lowering actions for each level is shown in Figure 2. A WHIRL file generated by the front end consist of WHIRL instructions and WHIRL symbol tables. The instructions contain references to the symbol table. WHIRL instructions are linked in tree form.

2.2.3 Unparsing

Very High level WHIRL can be translated back to C and Fortran source code using the appropriate Open64 tools, but almost no optimisation is performed at this level. It is possible to export the intermediate representation to a file, or to output different stages of the compilation.

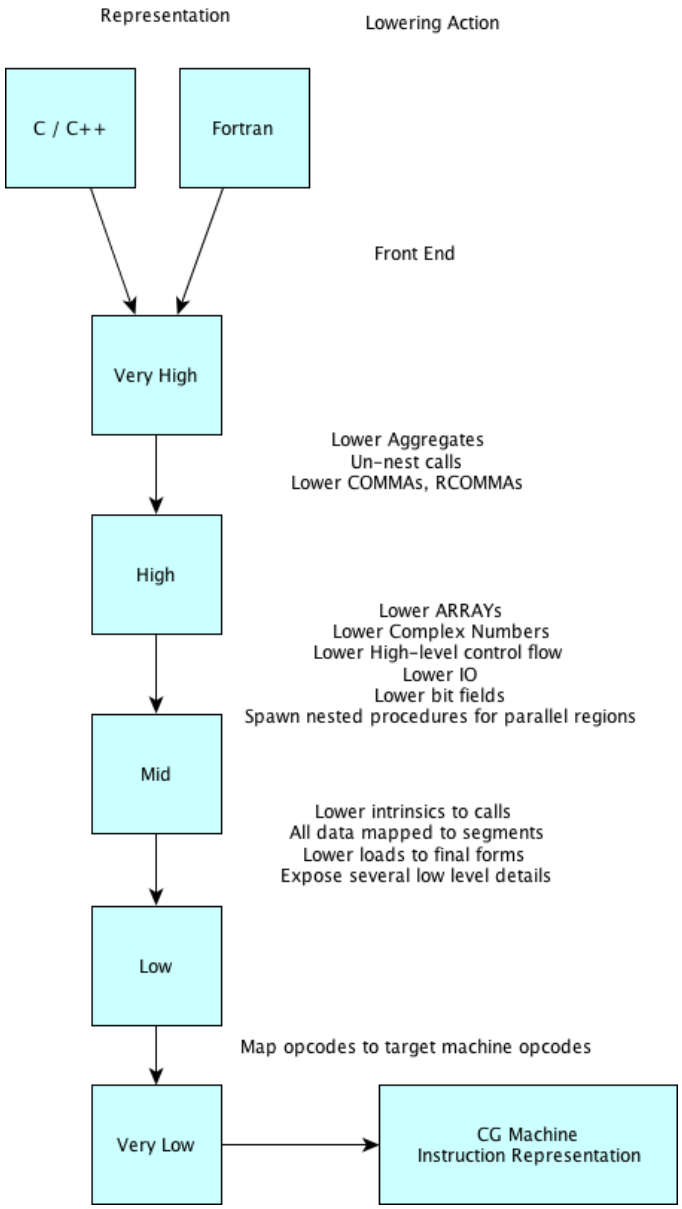

Figure 2

Levels of WHIRL and Lowering actions

2.2.4 Querying and AST Traversal

The class *WN_TREE_ITER_base* implements a STL-compatible iterator for WHIRL containers (*WN_TREE_CONTAINER*). It is possible to define different traversal orders (pre-order or post-order iteration). Comparison between two trees of WHIRL containers is possible through the overloaded operators. To apply a particular operation to the WHIRL tree, the user can call the function *WN_TREE_WALK*, which receives a WHIRL node, an operation and an instance of *WN_TREE_ITER* to indicate how to traverse the tree. The operation is executed for each WHIRL node. Listing 1 shows an example of calling a Tree Traversal to implement a simple node counting operation. Querying for a node or a particular set of node can be implemented using Operations or by implementing a derived class from *WN_TREE_ITER_base*.

```
// function object example: count the number of whirl nodes
struct WN_count {
  INT num_nodes;
  WN_count() : num_nodes(0){}
  void operator()(WN*) {
    ++num_nodes;
  }
};

int main () {
 ...
 WN_TREE_walk_pre_order (wn, WN_count());
 ..
}
```

Listing 1

Open64 Tree Traversal example

2.3 LLVM

LLVM [18] (Low Level Virtual Machine) is a compiler infrastructure written in C++. It supports several different kinds of optimizations for programs written in arbitrary programming languages. Several different front ends have been created which take advantage of the language-agnostic design, such as Objective-C, Fortran, Ada, etc.

To achieve this language-agnostic design, it is necessary to have a common intermediate representation in which code optimisations can be applied without the need for specific-language optimisations. The IR of LLVM is a Static Single Assignment (SSA) form with a simple, language-independent type system that exposes the primitives commonly used to implement high-level language features.

LLVM is compatible with standard makefiles and can use GCC as a C and C++ parser. Object files compiled with LLVM can be linked with object files built with gcc using the LLVM linker. Notice that LLVM object files contain LLVM IR/bytecode, not machine code.

2.3.1 General Usage / Workflow

Figure 3 illustrates the compilation worfklow of a typical LLVM driver.

After collecting the command line options, which instruct the compiler driver about the passes that it should run, the driver reads the configuration files for each pass - this will vary depending on the kind of input files pointed by the user. These configuration files can be provided by the user or by the tools. Each phase that is going to be executed can result in the invocation of one or more actions. An action is either a whole program or a function in a dynamically linked shared library. In this step, the driver determines the sequence of actions that must be executed. Actions will always be executed in a deterministic order. The actions required to support the original request from the user are executed sequentially and deterministically. All actions result in either

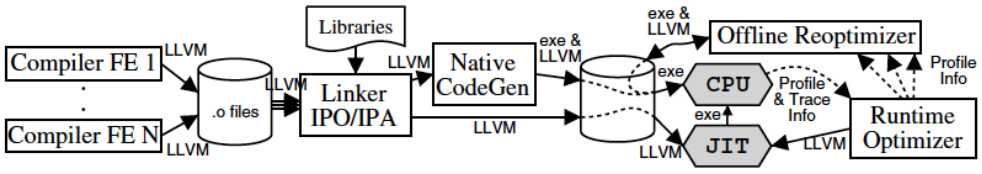

Figure 3

LLVM Workflow

the invocation of a whole program to perform the action, or the loading of a dynamically linkable shared library and invocation of a standard interface function within that library.

The compiler driver (`llvmc`) splits every compilation task into the following five distinct phases:

- **Preprocessing**: This phase can be invoked for those languages supporting preprocessing.
- **Translation**: Converts the source language input into the IR.
- **Optimization**: All optimizations are performed on the IR, according to the options provided.
- **Linking**: The inputs are combined to form a complete program.

2.3.2 Intermediate Representation

The LLVM code representation describes a program using an abstract RISC-like instruction set but with key high-level information that enables effective code analysis. This includes type information, explicit control flow graphs, and an explicit dataflow representation [7]. The main features of the LLVM IR are:

- A low-level, language-independent type system that can be used to implement data types and operations from high-level languages.
- Instructions for performing type conversions and low-level address arithmetic while preserving type information.
- Two low-level exception-handling instructions for implementing language specific exception semantics.

The LLVM code representation is designed to also be used as a human readable assembly language representation, facilitating development and debugging.

LLVM programs are composed of modules, each of which is a translation unit of the input programs. Each module consists of functions, global variables and symbol table entries. Modules can be merged together by the LLVM linker. Listing 2 shows an example of the `Hello World` module.

```
; Declare the string constant as a global constant.
@.str = private unnamed_addr constant [13 x i8] c"hello world\0A\00"

; External declaration of the puts function
declare i32 @puts(i8* nocapture) nounwind

; Definition of main function
define i32 @main() { ; i32()*
  ; Convert [13 x i8]* to i8 *...
  %cast210 = getelementptr [13 x i8]* @.str, i64 0, i64 0

  ; Call puts function to write out the string to stdout.
  call i32 @puts(i8* %cast210)
  ret i32 0
}

; Named metadata
!1 = metadata !{i32 42}
!foo = !{!1, null}
```

Listing 2

LLVM Code Representation

2.3.3 Unparsing

LLVM has been designed as a full source-to-binary compiler. Although it is possible to extract the different levels of Intermediate Representation to Text Files, it is not possible to recover the original language from these representations as it is completely agnostic to the original source language.

Theoretically speaking, it would be possible to use the IR to recreate a new source and port it to another language by creating an IR-to-language converter. However, as far as we are aware such a tool does not exist.

2.3.4 Querying and IR Traversal

It is possible to traverse the internal representation of LLVM using different iterator classes available in the framework. The following is a list of the available iterators:

- `Module::iterator` Iterates through the functions in the module (source file)
- `Function::iterator` Iterates through basic blocks in the module
- `BasicBlock::iterator` Iterates through instructions in a block

These iterators can be extended to create new ones. The code motion is implemented with operations on the nodes (`EraseFromParent`, `RemoveFromParent`, etc). Data dependency, Call Graph and Alias information, among others, can be printed to a Dot graph then to an external file.

```
1 // Basic ROSE translator
2 #include "rose.h"
3 int main(int argc, char **argv) {
4   // Build the AST used by ROSE
5   SgProject* sp = frontend(argc, argv);
6   // Run internal consistency test on AST
7   AstTests::runAllTests(sp);
8   // Generate source code from AST and call the vendor's compiler
9   return backend(sp);
10 }
```

Listing 3

ROSE driver example

2.4 ROSE

ROSE [19] is an open source compiler framework designed to facilitate building programs for applying source code transformations. It is primarly tailored to design and implement static analysis tools, source code transformations, loop optimizations, performance analysis and even static security checks of source codes. It has support for C, C++, Fortran and OpenMP.

As usual, ROSE uses a three layer approach (front end, middle end and back end). The result of running a ROSE driver is the generation of a new source code.

ROSE uses the front end of the Edison Design Group (EDG) for C and C++, whereas Open Source Fortran is used to generate the AST from Fortran sources.

Although the license of the front ends is not free, it is possible to redistribute it with the framework. The resulting IR of this front end is translated into a AST. This AST preservers most of the original source information (i.e. comments, preprocessor directives, original line numbers and so on), making it possible to completely *unparse* the AST into a source file.

2.4.1 General Usage / Workflow

Figure 4 illustrates the different compiler phases of the ROSE compiler framework. After parsing the source, ROSE converts the IR from the EDG front end into the AST. This AST is further processed by several transformation and optimization tools. Developers can add new tools or implement new transformations extending the class hierarchy.

Listing 3 shows an example driver routine. Line 5 invokes the front end, which returns the AST of the original program. This AST is *unparsed* to rebuild the original source.

2.4.2 Intermediate Representation

The internal representation of ROSE is SAGE III (derived from SAGE++, [5]). SAGE III is automatically generated with a tool included within ROSE. When a code is parsed by the front end, a connection code translate EDG IR into SAGE III. This enables to distribute EDG binary files along ROSE while respecting licensing concerns.

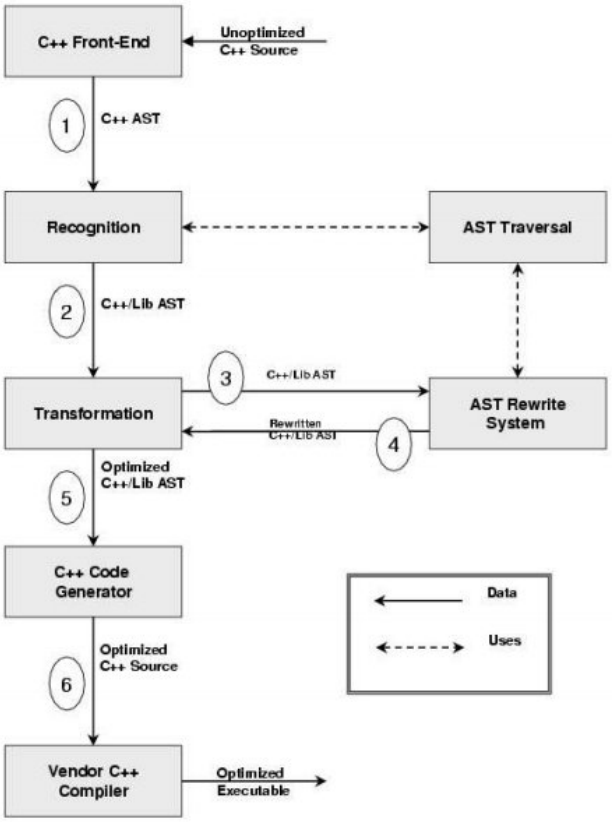

Figure 4

Usual Workflow of a ROSE driver

2.4.3 Unparsing

As mentioned before, ROSE outputs a source file. It attempts to produce an output file as close to the original as possible, however, some differences may arise. Most of these differences come from the fact that there is more than one way to write the same expression in the original language. For reasons that will be described in the following section of this document, it is worth highlighting the following differences: (1) Variable declarations are normalized to separated declarations, (2) Normalization of member access from a pointer, and (3) Array indexing is represented through pointer arithmetic.

2.4.4 Querying

One of the critical features of a StS tool is the ability to search for particular nodes or subtrees. In ROSE, this is called `querying`. ROSE offers several querying features for nodes and/or subtrees. Common queries are already predefined (for example, looking for a particular variable declaration), and a developer can implement their own queries implementing the *NodeQuery* interface.

```
class MyVisitor:
public AstSimpleProcessing {
  protected :
  void virtual visit (SgNode* node) ;
}

MyVisitor::visit (SgNode* node) {
  cout << node->get_class_name() << endl;
}
```

Listing 4

ROSE Traversal example

2.4.5 AST Traversal

ROSE aids the library writer by providing a traversal mechanism. This mechanism visits all the nodes of the AST in a predefined order. It can be used to compute attributes or to perform an analysis of the code.

Based on a fixed traversal order, the framework provides inherited attributes to pass information down the AST (top-down processing) and synthesized attributes for passing information up to the AST (bottom-up processing). Inherited attributes can be used to propagate context information along the edges of the AST, whereas synthesized attributes can be used to compute values based on the information in the subtree. One function for computing inherited attributes and one function for computing synthesized attributes must be implemented when attributes are used.

Different interfaces are provided which will allow either one, both, or no attributes to be used; in the latter case it is a simple traversal with a visit method called at each node. AST Traversal is offered through the *AST*Processing* classes. An example is shown in Listing 4.

2.5 Cetus

Cetus [8] is a StS framework designed to implement code transformations. The current version supports ANSI C via ANTLR 2 [29], and it is implemented in Java. Cetus derives from the original POLARIS [28] compiler framework, although the Cetus project attempts to be more general than the previous one.

2.5.1 General Usage / Workflow

As usual, Cetus is divided into three different layers; the front end, the middle end (with the IR) and the back-end. Figure 5 illustrate the Cetus architecture.

Compiler writers usually only need to extend the *Driver* class to implement any kind of transformations, as shown in Listing 5

```
public class MyDriver extends Driver {
  public void run(String[] args) {
    parseCommandLine(args);
    parseFiles();
    if (getOptionValue("parse-only") != null) {
      System.err.println("parsing␣finished␣and␣parse-only␣option␣set");
      Tools.exit(0);
    }
    runPasses();
    PrintTools.printlnStatus("Printing...", 1);
    try {
      program.print();
    } catch (IOException e) {
      System.err.println("could␣not␣write␣output␣files:␣" + e);
      Tools.exit(1);
    }
  }
  public static void main(String[] args) {
    (new MyDriver()).run(args);
  }
}
```

Listing 5

Java code to create a driver using the Cetus API

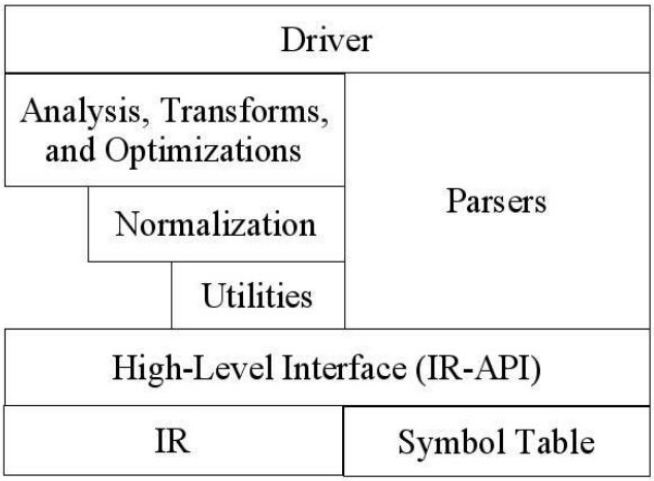

Figure 5

Cetus architecture

2.5.2 Intermediate Representation

In Cetus, the IR follows a hierarchical statement structure, directly reflecting the block structure of a program. A class hierarchy is used to implement the IR. Although this block structure and the subsequent implementation of the *Unparsing* and *Transversal* classes make it difficult to implement languages other than C, some additional abstract classes are provided to ease this task. A C++ front end is in progress, and there are plans to implement Java and Fortran90 back ends.

The IR has been designed to be easily understood by users. The contents of a source file are stored in `traslation units` and `procedures` represent individual functions. Procedures include a list of simple or compound statements, representing the program control flow in a hierarchical manner. Each node of the IR can be annotated with comments or directives.

2.5.3 Querying and Traversal

Iterators following the Java Programming style are available. Listing 6 shows an example of a DepthFirst iteration. Other iterators, such as Bread-first or Flat are available. The `next(c)` method returns the next object of the class `c`.

Other traversals apart from basic syntactic order are available. For example, it is possible to instantiate a caller traversal, which iterates across the calltree of a program.

Querying is implemented by means of iterators. The *instanceof* Java operator can be used to check if a particular node of the IR is an instance of a class.

2.6 Mercurium

Mercurium [14] is a StS compilation infrastructure designed to implement the StarSs programming model [30], although it has been extended as well.

```
/* Iterate depth-first over program which is instance of Program */
DepthFirstIterator dfs_iter = new DepthFirstIterator(program);

while (dfs_iter.hasNext()) {
  Object o = dfs_iter.next();
  if (o instanceof Loop) {
    System.out.print("Found instance of Loop");
  }
}
```

Listing 6

Java code to iterate through the Cetus IR and print a message whenever a `for` loop is traversed

```
#pragma hlt unroll factor(24)
for (i = 0; i < 100; i++) {
  a[i] = i + 1;
}
```

Listing 7

Loop unrolling in Mercurium

Mercurium is composed by a set of plugins written in C++ that are automatically loaded by the compiler following instructions from a compilation configuration file. High-level transformations at IR level are implemented similarly to those available in Cetus and ROSE.

2.6.1 General Usage / Workflow

Figure 6 shows the Mercurium workflow. The compiler receives input in the form the source code written in C, C++ or Fortran and processes it through the front end, to generate a high-level interface. Later, a set of transformations are applied. Some parts of the original code may be outlined to an external file, enabling other back ends to process these external parts.

As in Cetus (see Section 2.5) Mercurium also features a component that is responsible for high-level transformations. These code transformations are aimed to code optimization. Different loop transformations are available to the programmer through the `#pragma hlt` directive. Listing 7 shows an example of the directive to perform *loop-unrolling*.

Listing 8 shows an example of *loop collapse* in Mercurium: given a perfect nest of regular loops, loop collapse creates a single loop that iterates over the n-dimensional iteration space.

Several other loop transformations (blocking, interchange, fusion, distribution, etc.) are also available in mercurium through the `#pragma hlt` directive.

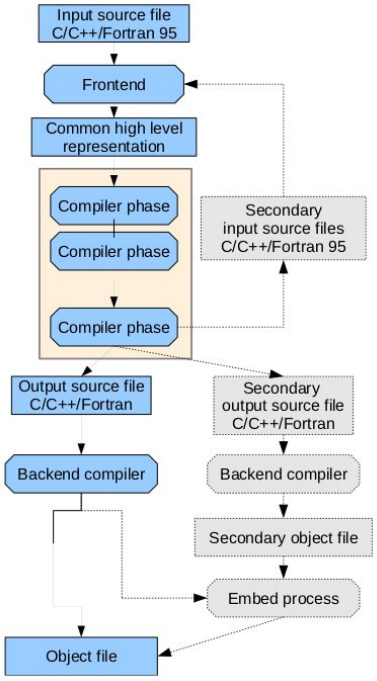

Figure 6

Mercurium workflow

2.6.2 Intermediate Representation

The IR of Mercurium is based on an augmented abstract syntax tree. A class hierarchy of nodes is exposed to the developer and different operators allowing them to manipulate the trees are available.

New code is created using plain source using stream operators.

2.6.3 Querying and AST Traversal

Although possible, it is not necessary to work direclty with `TL::AST_t`, as several wrappers (named `LangConstruct`) are available.

AST Traversal can be achieved through the usage of predicate classes, which assert that some boolean properties on the tree nodes are matched. Pre-created predicates are available to walk only specific node (for example, FunctionDefinitio::predicate).

2.6.4 Unparsing

Mercurium is capable of unparsing AST nodes using an overloaded stream operator. Redirecting any pointer to an IR subtree recreates its original C (or C++) code.

```
#pragma hlt collapse
for (i = 0; i < 100; i++)
  for (j = 0; j < 200; j++)
    for (k = 0; k < 300; k++)
      a[i][j][k] = i * j * k;
```

Listing 8

Loop collapse in Mercurium

2.7 Final Remarks

Table 1 shows an overview of the different characteristics of the compiler frameworks that we have studied in this Section.

LLVM, Open64 and GCC are powerful compilers. The internet is full of information about them and new information is being added regularly. When we started our research back in 2008, we found that finding information on these compilers was far more difficult than it is today. However, despite all of the available information, writing a compiler pass for any of them still remains a challenge due to the enormity of their codebases.

In these three cases, transformations have to be implemented in the Internal Representation, which means writing transformations at low level. This is great for implementing classic compiler optimisations, but it may give rise to explaining loop level transformations (like loop tiling) to students. In the case of Open64 or GCC, working with the front end is not an easy task. Programmers are expected to work after the front end has parsed the code into the IR, so adding new features to a language requires in-depth knowledge of the particular front end. Depending on how the new features modify the original language the generation of WHIRL (Open64) or GIMPLE (GCC) have to be modified as well.

ROSE is a great tool for developers wanting to write code analysis tools, or even simple source-to-source translation. The High-Level intermediate representation facilitates code motion and it is possible to print the status of the tree, verify its correctness and insert or remove children to any node without too much effort. However, the front end is a black-box which the user has little or no access to (none if using Fortran).

The Cetus project was in its early stages when we first carried out this survey, and it was not very stable. However, their ideas about a flexible and easy accessible IR were interesting, and our design was inspired by this work.

Mercurium was not completely public and accessible four years ago. Even today, not enough documentation is available, and playing around with the front end is not an easy task.

As none of the available tools completely satisfied our needs, we decided to write our own compiler translator. We did not aim to produce a high-quality commercial compiler, but an easy-going research tool capable of performing code transformations with little development and without excessive bootstrapping time (i.e. time from not knowing anything from the compiler architecture to being able to do useful work).

The pycparser project [4] featured a nearly complete C parser written in Python. The availability and maturity of this project greatly influenced our decision to tackle the laborious task of implementing an entire C front end.

Table 1

Final comparison of compilers

Feature	Open64	GCC	ROSE	Cetus	Mercurium	LLVM
Flexible Parser	No	No	No	Some	Some	No
Portability	Some	Some	Some	Some	Some	Some
Step-by-step	No	No	Some	Some	No	No
StS Transform.	No	No	Yes	Yes	Some	No
Recover orig. file	No	No	Yes	Yes	Some	No
Documentation	Some	Some	Yes	Some	No	Yes

3. Design Considerations and Basic Concepts

In this Section we discuss some design donsiderations and introduce basic concepts.

As the aim of `YaCF` is to develop StS transformations, the IR chosen is an augmented syntax tree. Details of the IR are provided in Section 4.. Part of the information used to augment the AST is the Symbol Table (ST), as described in Section 5.. Outside these packages, a `bin` directory contains Python driver scripts to perform particular tasks, such as implementing code transformations. The majority of the work in the `YaCF` Frontend and the IR is derived from the `pycparser` project [4].

`YaCF` Components have been grouped together into three packages: FRONTEND, MIDDLEEND and BACKEND, through which the Internal Representation (IR) of `YaCF` is used. Details of each package can be found in Sections 6., 7. and 8.. Figure 7 illustrates the overall StS transformation process.

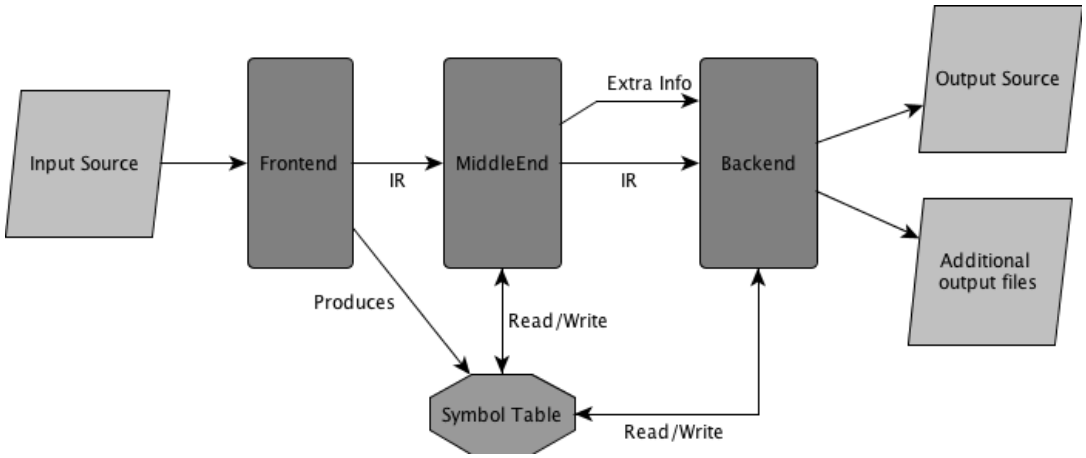

Figure 7

Overall translation workflow executed by a typical `YaCF` driver

The typical use case for a `YaCF` developer is to implement a StS transformation. A `YaCF` developer would normally use `YaCF` to implement a StS transformation. This is the most common application of the tool. Code transformations are usually implemented at IR level which means

the source will have had to have been parsed first. The user would then like to recover the original file after the transformation, so once modified, the file has to be unparsed (or re-written) once again into the input language.

In `YaCF` a driver is the script that orchestrates a code transformation. From the driver, the parser is called, the IR is generated and it is passed to the next modules for processing. Usually drivers end up calling a Writer to unparse the final AST into the original input language. However, this is not mandatory such as when drivers are used to gather statistics from the source code.

The code translation can be split into two separate steps: (1) Searching for a particular pattern or idiom in the code and (2) Applying the desired transformation on the nodes matching the criteria. Within `YaCF` these two tasks are implemented in two class hierarchies: The *Filter* (1) and the *Mutator* (2).

A *Filter* is an implementation of the generic Visitor Pattern [21], which traverses the IR looking for matching nodes. The Visitor Pattern design provides a way of separating an algorithm from the object structure on which it operates. A practical result of this separation is the ability to add new operations to existing object structures without modifying those structures. A *Mutator*, or *Transformer*, is a class that contains a *Filter*; it is designed to apply the contents of a specified function to each node matching the *Filter*.

The entire `YaCF` framework is built using these two basic concepts. Complex transformations are composed by several nested *Filters* and *Mutators*. Usually, a *Runner* class is used to group together several transformations required to accomplish a major code transformation. *Runner* classes contain source storage facilities and code templates, and they are used to prepare the environment before (and after) calling a set of *Mutators*.

3.1 Filter

All *Filters* are derived from the *GenericFilterVisitor* class. This class, derived from the original *Visitor* class from the pycparser project [4] implements several top-down traversals of the IR. A *ReverseVisitor* that traverses the tree from bottom-up is also available. However, this visitor requires the parent link to be set, thus, it is not possible to traverse raw AST nodes. IR levels are discussed later in Section 4..

To implement a new *Filter*, it is necessary to extend the *GenericFilterVisitor* class, as shown in Listing 9. The constructor of this *Filter* has to call the constructor of the parent (call to *super* in the Listing). The *condition_func* determines the matching condition for the *Filter*. The parameter of this condition function is always the current node being visited, and it must return `True` if the node matches the criteria or `False` otherwise. Any other returning value is considered an error and will raise an exception. The condition function in Listing 9 checks that the type of the current node is a declaration. Additional information about nodes and subtree types is available in Section 6..

More complex *Filters* can be written taking advantage of the *Visitor* pattern. Overriding the *visit* method for a particular kind of node forces all traversals to execute the contents of the method. Listing 10 shows an example of this situation. Suppose we want a *Filter* to match all declarations inside a function called `foo`. We can extend the code from Listing 9 with an additional method *visit_FuncDef* (see Listing 10). The method is called each time a node of that kind is visited. If the node is the one we are looking for, we set the variable to `True`. When visiting declaration nodes, if the variable is `True` we know we are inside the desired function, thus, we mark that this is the desired node.

```
class ExampleFilter(GenericFilterVisitor):
   """ Returns the first node matching the example node
   """
   def __init__(self):
      def condition(node):
         if type(node) == c99_ast.Decl:
            return True
         return False
      super(ExampleFilter, self).__init__(condition_func = condition)
```

Listing 9

A simple implementation of a *Filter* that will iterate throught all the declarations of a given subtree

Notice that: (1) The *visit_FuncDef* method is called before checking the condition and that (2) as the new method overrides any of the defaults, to continue traversing down the AST we have to manually call the *generic_visit* method for each of the attributes of the FuncDef node that we want to visit.

The *GenericFilterVisitor* class implements a variety of methods:

- *apply*: Looks for the first node matching the criteria and returns.
- *iterator*: Iterates over all of the matching nodes preserving the grammatical ordering.
- *fast_iterator*: Iterates over all matching nodes in a deep first search fashion (it is faster but does not guarantee grammatical order).

3.2 Mutator

Mutators are created by extending the *AbstractMutator* class. The code transformations (*aka* mutations) usually contain a *Filter* that selects which nodes will be transformed. A *mutatorFunction* method has to be specified, and it must contain the code to perform the desired transformation.

Mutators modify the IR, but they must ensure its consistence (i.e. update Symbol table, parent links, and so on). Additional information about the Internal Representation is available in Section 4..

Note: Recursive Mutators
When implementing a *Mutator*, it is important to take into account the kind of transformation being applied. If the transformation alters the IR in such a way that might match again the condition, it will enter into an infinite loop, matching and applying the same *Mutator* repeatedly.

4. Internal Representation

`YaCF` has been designed as a StS translation tool, thus, it does not offer functionality to generate low-level code. The IR is based on an annotated (sometimes called augmented) high-level tree-layered AST, which maintains a close relation to the original source, while facilitating the work of

```
class ExampleFilter(GenericFilterVisitor):
   """ Returns the first node matching the example node
   """
   def __init__(self):
      self._inside_foo = False
      def condition(node):
         if type(node) == c99_ast.Decl \
             and self._inside_foo = True:
              return True
         return False
      super(ExampleFilter, self).__init__(condition_func = condition)

   def visit_FuncDef(self,node):
      if node.name == "foo":
         self._inside_foo = True
      self.generic_visit(node.body)
      self._inside_foo = False
```

Listing 10

A more complex example of *Filter* where only those declarations inside a particular function will be traversed

```
class ExampleMutator(AbstractMutator):
   """ Apply a mutation
   """
   def filter(self, ast):
      def is_decl:
         if type(node) == c_ast.Decl:
             return True
          return False
      return DeclFilter(ast, condition_func = is_decl)
   def mutatorFunction(self, ast):
      # .... do something here with the matching node
      return ast
```

Listing 11

Example of a *Mutator* that will apply a transformation to all declarations within a subtree

```
pi_omp = 0.0f;
#pragma omp parallel for private(i,local) reduction (+: pi_omp)
  for (i = 0; i < N; i++) {
     local = (i + 0.5)*w;
     pi_omp = pi_omp + 4.0/(1.0 + local*local);
  }
pi_omp *= w;
```

Listing 12

Implementation of the π computation using OpenMP

the developer of code transformations. Developers only need to work with tree-like structures resembling the structure of the original code, rather than focusing on low-level intermediate code. Each node of the IR denotes an element of the original language. For example, in the C front end, the `if` statement or a function definition are nodes of the tree structure. Figure 8 shows an example of a subtree. Figure 9 shows the IR structure corresponding to a more complex C code (Listing 12) that computes an aproximation for the constant π.

The IR syntax is abstract, which means that it does not represent every detail appearing in the real syntax. For instance, grouping parenthesis are implicit in the tree structure, and a syntactic construct such as an `if-condition-then expression` is denoted by a single node with two branches. Each node of the IR is a Python class. Nodes of the IR contain three different kind of attributes: simple attributes (any Python class), a child node (a reference to a son) or a list of children nodes (when an attribute of the node may contain several values). *Filters* (see Section 3.1) use children and list of children nodes to traverse the tree. Other elements of the Framework rely on simple attributes to work (for example, a *Mutator* can alter the `name` attribute of a node). Information might be added to each node after a translation or an analysis is run, a process referred to as annotation. Traditionally, transformations used the *setattr* and *getattr* functions to set and get information from the IR. Recent versions of `YaCF` feature a new `annotate` dictionary to store the new attributes using a more homogeneous interface.

All nodes of the IR inherit from the *IRNode* class, and contain the following base attributes and methods:

- `parent`: Provides a reference to the direct parent node, if any.
- `coord`: Coordinates of the equivalent lexeme for the node in the file (i.e. line number).
- *show()*: Prints the node in a human-readable form (but not in the original language).
- *children()*: Returns all descendant nodes from the current ones.
- *getRootNode()*: Traverses the parent links up to the top node.
- *getContainerAttribute()*: Returns the attribute of the parent node (if any) linking with the current node.

Depending on the information available on the IR, we distinguish the following incremental levels (i.e. states):

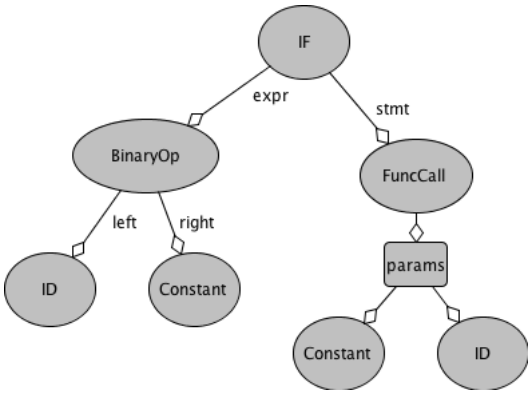

Figure 8

IR for the statement `if(a>1) printf("%d",a)`

- **IR-1** or **AST**: This is the result of *parsing* a source code with any of the implemented front ends. It contains only basic parsing information. The parent attribute is not set. Neither the ST nor the Writer are available at this level. A description of the FRONTEND package which provides details on the creation of the AST for a particular language, is available in Section 6.. Here we only describe the overall IR structure.
- **IR-2**: This is the result of processing a **IR-1** subtree with the *AstToIR* class. After the *annotation* process, a ST is available for the entire subtree, it is possible to re-print any node of the IR in the original language, and the parent link is properly set. At this level, two additional attributes (`sequence` and `depth`) are available.
- **IR-3**: The back ends might need to add further information to the nodes in the AST. To represent this fact, we will refer to the IR-3 level whenever we are describing a process in which a `YaCF` component adds information to the `IR-2`. If a transformation requires the information added by another one, we specify that using the notation `IR-3.name` where `name` is the name of the component which augments the IR. For example, a transformation requiring the information from the CUDA back end will require the `IR-3.CUDA`.

4.0.1 The *AstToIR* class

In order to transform the IR-1 into a functional IR-2, it is necessary to use the *AstToIR* transformer class. The *AstToIR* class has been implemented following the *Flyweight* pattern [15]. This forces subsequents calls to the *apply* method of this class with the same node to reuse precomputed information (particularly the Symbol Table (ST)). The root node of the tree (`FileAST`) is used to maintain a cache of the currently available IR. The *AstToIR* class is not generic to all front ends and some of them might have their own implementation, that will always be derived from the original. The *AstToIR* class performs the following tasks:

- Replaces the *str* method with a call to the appropriate writer/unparser
- Connects the parent link. Using a deep first search strategy, the parent link in each node of the IR is connected. The parent link of the root node (usually a `FileAST` node) is set to `None`.

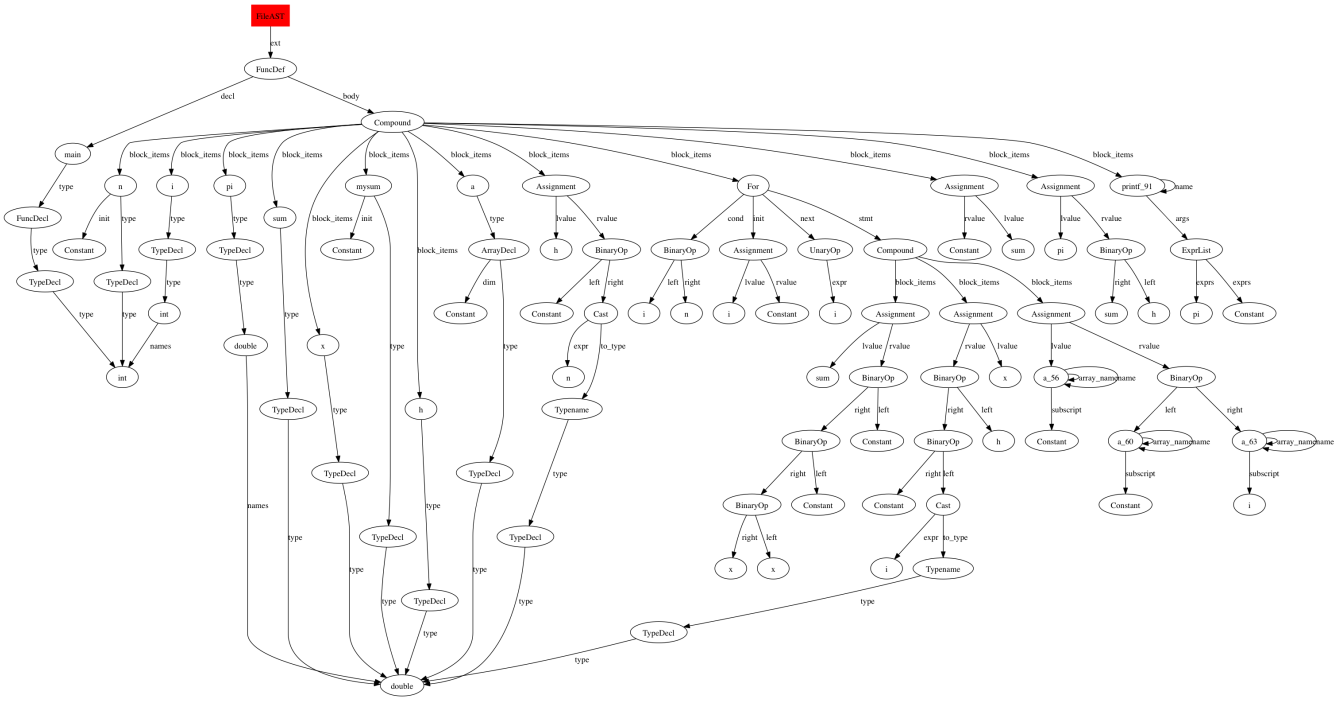

Figure 9

IR generated for the main loop of the π computation example shown in Listing 12

- Creates the Symbol Table: an object of the ST class is created (see Section 5.) and the class *SymbolTableBuilder* is used to initialize the ST and update the information of the IR.

To apply the aforementioned transformations, the user has to call the *annotate* method. Using the *getSymbolTable* method of the same class it is possible to retrieve the ST that has been created during the annotation process.

4.1 Manipulating the IR

It is possible to manipulate the IR using standard Python tools, like `setattr` or `getattr`. However, for convenience, a set of tools is available in the TOOLS.TREE package. These tools work on the `IR-2` level, but they do not require information from the ST as they only work within the tree.

Three operations to manipulate the IR are implemented: *Insert*, *Replace* and *Remove*. Figure 10 shows the insertion of a new subtree into the IR. Tools to manipulate the IR work on two stages: Firstly they build an instance of the tool and then they apply it to a subtree. This enables `YaCF` to reuse part of the process when the same operation is applied several times to the same subtree.

InsertTool performs insertions on the subtree. Its constructor receives a subtree to be inserted. The *apply* method receives the node that will be the parent of the subtree and the attribute where it is going to be inserted. If the attribute where the node will be inserted is a list of nodes, it is possible to specify a position inside the list, choosing one of the following values: `begin`, `end`, `after` and `before`. The values `after` and `before` allow users to insert the node immediately

```
it = InsertTool(subtree = new_subtree)
it.apply(node1, 'attribute', position = "begin")
it.apply(node2, 'attribute', position = "after")
```

Listing 13

Inserting a subtree inside the main IR

```
ReplaceTool(new_node = new_subtree, old_node = old).apply(old.parent,
   'attr')
```

Listing 14

Replace a subtree inside the main IR

after or before an existing node on the list, which is specified by the parameter `prev`. A code example for this operation is shown in Listing 13.

In a similar fashion to the *InsertTool*, a *ReplaceTool* and a *RemoveTool* are also available. Figures 10, 11 and 12 show the effect of these operations on the IR. Listing 14 shows an example code to perform a replacement of a subtree within the main IR.

5. Symbol Table

The `YaCF` ST is designed to be independent of the parsing process. It can be created or updated at any stage of the StS translation. The ST is implemented as an extension of a Python dictionary. Usual ST operations, such as *lookUp* and *addSymbol*, are implemented in this class.

The creation of the ST for an IR is performed through the builder class *SymbolTableBuilder*, which is a *Visitor* that creates the IR by traversing the tree in grammatical order.

The easiest way to create a ST is to instantiate an object of the *SymbolTable* class and then invoke the *SymbolTableBuilder* to initialize it, as shown in Listing 15.

Each element of the ST is stored as a `Symbol` object, and holds the following information:

- `name`: Name of the Symbol.
- `node`: Reference to the original node in the AST.
- `type`: Type of the node, reference in the IR.
- `scope`: Scope information (see Section 5.1).
- `btype`: Basic type of the ID (for example, in C, if the identifier is an integer).
- `sizeExpression`: Expression to determine the size of the variable.
- `extra`: A dictionary holding optional information for different compiler stages.

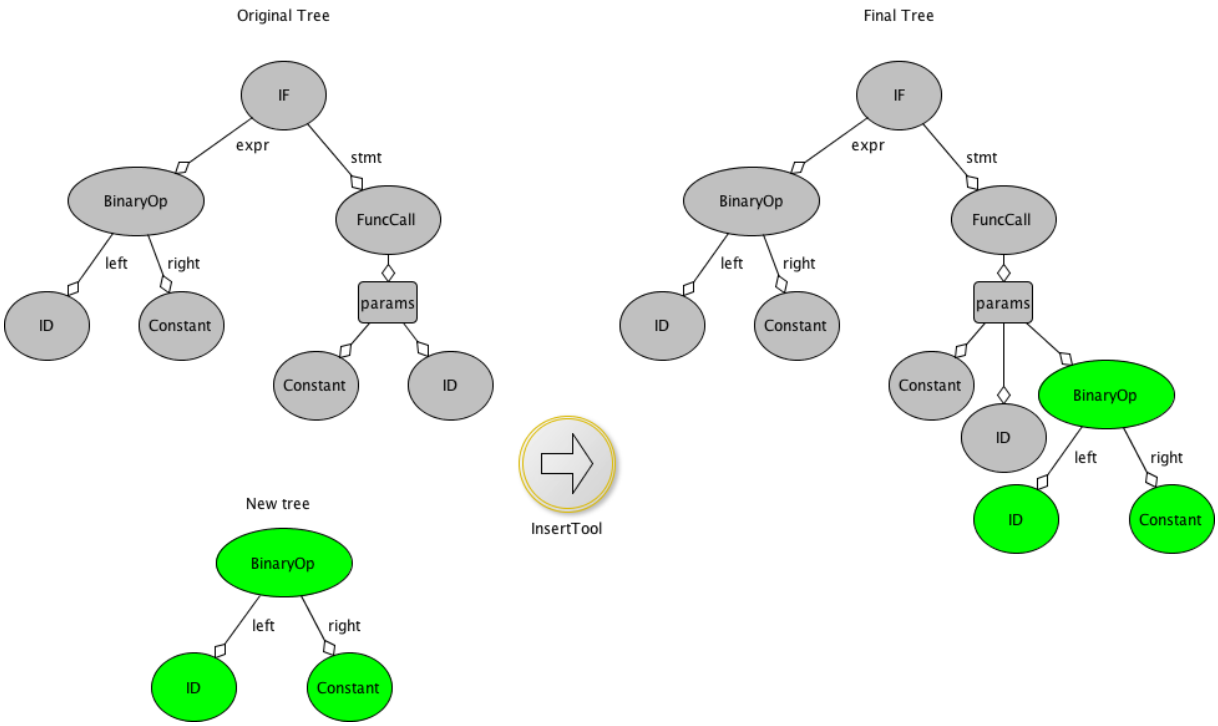

Figure 10

Insertion of a new parameter in a function call

`Symbol` objects can be printed to a string for debugging purposes. They can also be compared using two simple rules: (1) Two *Symbol* instances are equivalent iff they have the same name and the same scope, and (2) *Symbol* instance A is greater than *Symbol* instance B if and only if `scope`(A) is greater than `scope`(B).

5.1 Scope Information

As the ST can be created at any stage of the translation process, and it might even be created for a subtree disconnected from the main IR of the code; the coordinate of the lexeme relative to the IRNode is not enough to locate an identifier in the ST. To replace the line number information, each node of the AST is decorated with two additional attributes: the sequence number and the depth. The sequence number is assigned sequentially to each node of the IR in grammatical order, starting from zero. The depth value indicates the number of grammatical nested scopes preceding the declaration. For this reason, each symbol has a `Scope` attribute, which describes the proper scope of the declaration. Algorithm 1 is used to insert an element in the *SymbolTable*, whereas the Algorithm 2 is used as the lookup for symbols.

Listing 17 shows a C code with several name conflicts. Identifier `i` is declared several times (lines 3, 6 and 12). Printing the ST corresponding to the code in Listing 17 produces the output shown in Listing 16.

Notice how each declaration of `i` has a separate entry in the ST (lines 20, 22 and 27 in Listing 17). Level 0 of the ST contains general language declarations and implicit types, like GNU built-in types, or basic language types.

```
from Frontend.SymbolTable import SymbolTable, SymbolTableBuilder
st = SymbolTable()
tsv = SymbolTableBuilder(symbol_table = st)
tsv.visit(some_ast)
```

Listing 15

Initialization of a *SymbolTable*

Algorithm 1 Insertion of a symbol in the Symbol Table

function ADDSYMBOL($decl, depth$)
  $sizeExpression \leftarrow buildSizeExpression(decl)$
  $nsymbol \leftarrow Symbol(decl, depth, sizeExpression)$
  for all symbol in $st[depth]$ **do**
    if $nsymbol = symbol$ **then return**
    end if
  end for
  $st[depth].insort(nsymbol)$
end function

Algorithm 2 Looks for the declaration of an identifier in the Symbol Table

function LOOKUP(id)
  $depth_{act} \leftarrow min(id.depth, len(st))$
  while $depth_{act} \geq 0$ **do**
    for all $symbolinst[depth_{act}]$ **do**
      if $id.sequence \in symbol.scope$ **then**
        if $symbol.name = id.name$ **then return** $symbol$
        end if
      end if
    end for
  end whilereturn Identifier Not Found
end function

```
 Symbol table
==========================
 Level : 0

[{float: float, (0, None)}
{int: int, (0, None)}
{char: char, (0, None)}
{double: double, (0, None)}
{FILE: FILE, (0, None)}
{bool: bool, (0, None)}

 Level : 1

[{foo: char, (1, 73)}
{func: int, (32, 73)}
{main: int, (67, 73)}]

 Level : 2

[{i: int, (7, 33)}
{j: int, (11, 33)}
{i: int, (39, 68)}]

 Level : 3

[{foo: char, (16, 32)}
{i: int, (21, 32)}]
```

Listing 16

Content of the ST after analyzing the code in Listing 17

These declarations are introduced by the builder before visiting the tree. The declarations of the code start in level 1, with the declaration of *foo* as a char. The numbers enclosed in parenthesis represent the start and end of the scope of the corresponding declaration. In the case of *foo*, the declaration is valid between sequence numbers 1 (beginning of the file) and 73 (end of the file). Notice that sequence number are not related in any way with line numbers in the source code. Function declarations appear on level 1 and their scope covers the declaration itself and continues right throught to the end of the file. Basic type (`btype`) of a function declaration represents its type.

Scope depth information suffices to differentiate the declarations in lines 2 and 5 in Listing 17, however, to distinguish the declarations in lines 2 and 12, it is necessary to take into account their sequence numbers.

5.2 Computing the Size of Elements

In some situations it is necessary to have access to the size of a particular element in the ST. This information is usually machine dependant (integers or doubles do not always have the same size), thus, it is not possible to have that information to hand when transforming the source. However, it is possible to extract an expression that computes the size of the element in terms of the original code, and, when compiled with the machine-dependant compiler, will generate the real size of the element.

When building the ST using the *SymbolTableBuilder*, information about the number of elements and the basic type of the declarations is stored in attributes. This information is used to create an

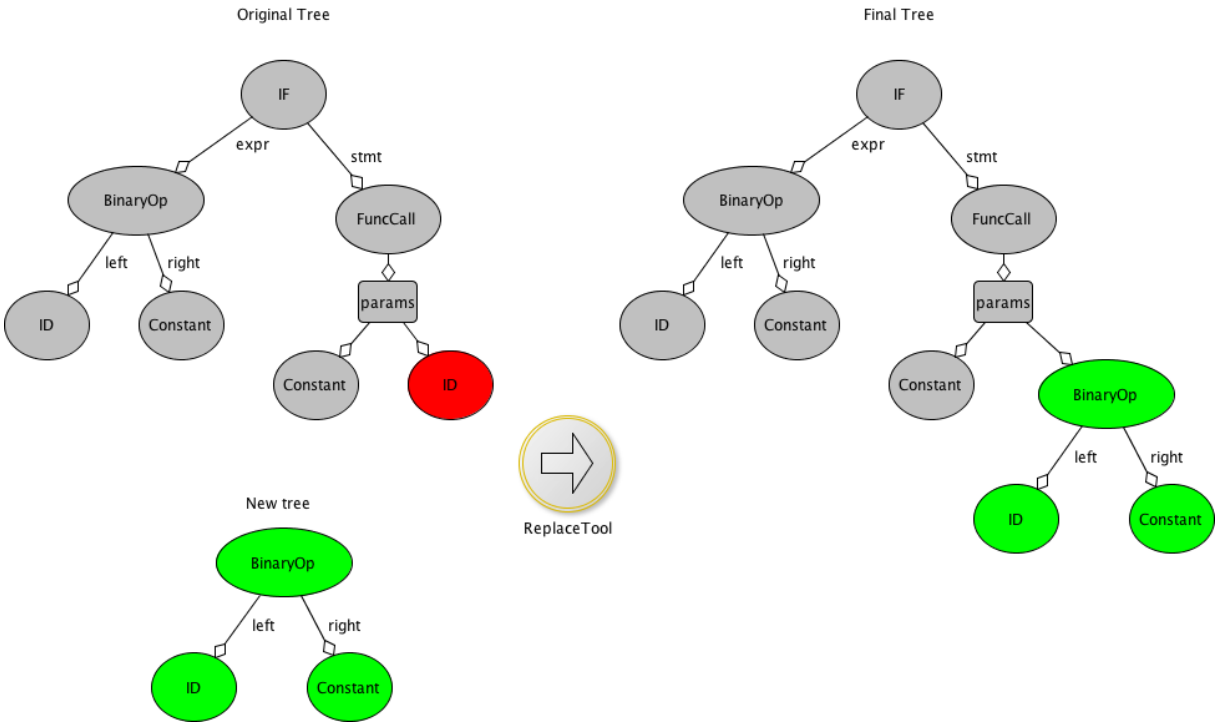

Figure 11

Replace operation of a parameter in a function call

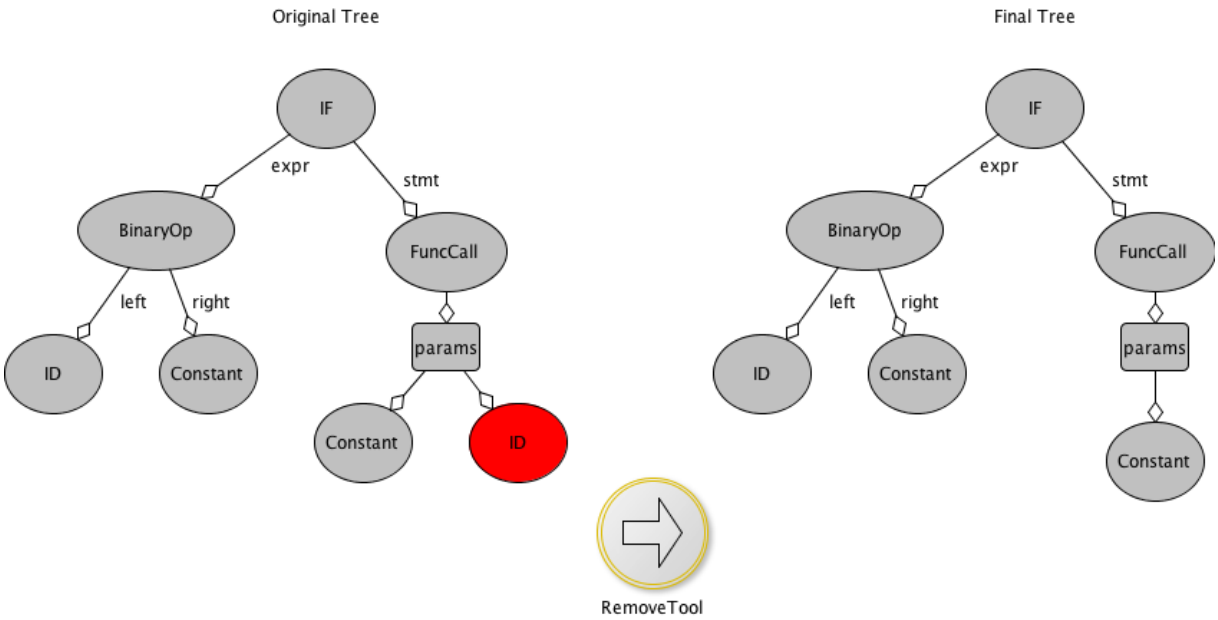

Figure 12

Remove operation of a parameter in a function call

expression that, when evaluated, will compute the total size of the declaration in memory. Notice that it is not possible to know information about pointers at translation time, but it is possible to compute that information at runtime. For example, if an structure contains a `void` pointer, the computed expression will not reflect the real size of the structure. However, the computed expression is sufficient to estimate the size of structures or arrays.

6. The FRONTEND

The FRONTEND package is based on the `pycparser` Python module. `pycparser` [4] is a C lexical and syntax analyzer completely written in Python. We have derived most of the structure of the FRONTEND from `pycparser`.

The `YaCF` FRONTEND package contains several components:

- C99, OpenMP, OpenACC and GNU syntax analyzers.
- Abstract Syntax Tree (AST).
- Symbol Table.
- Internal Representation: an extension of the AST with additional information. This information is used in source code transformations.

Figure 13 shows the `YaCF` class hierarchy. Class *PLYParser* is the base class of the `YaCF` FRONTEND.

PLYParser controls some syntax errors and holds general information about parsers. Each parser in `YaCF` must inherit from *PLYParser*. Lexical analyzers for differente languages inherit

```
1  char * foo;
2  int func () {
3    int i,j;
4    {
5      char * foo;
6      int i;
7      printf("%d",i);
8    }
9  }
10
11 int main() {
12   int i;
13   foo = NULL;
14   func()
15   printf("%d",i);
16   if (foo) printf("%s", foo);
17 }
```

Listing 17

C code example with nested declaration scopes

from *C99Lexer*. The design allows the developer to extend each lexer with specific rules. In a similar fashion, all parsers inherit from class *C99Parser*.

Parsers are implemented following a `Factory` Pattern, enabling the developer to combine different extensions of the C language into one new parser. The bottom section of Figure 13 shows the *AstToIR* class of the `YaCF` parser created by the *FrontendFactory* class combining C99 and OpenACC parsers.

6.1 Defining a New Language

Languages are defined in the FRONTEND package by creating a Python module containing a file named `languagename_ast.cfg` which lists the nodes of the language. All the AST nodes in `YaCF` are configured in this file enabling the compiler to create the AST with the correct information. Listing 18 shows an example of this file. In line 1, an array declaration node (`ArrayDecl`) is specified. The list of attributes is specified as a list in the same line. For example, an array declaration node has two attributes: the `type` and the `dimension` of the array. This information will be filled by the right parser using syntax-directed translation. The attributes with an asterisk indicate to the compiler that the node has a child node, while two asterisk indicate a sequence of child nodes. The `Compound` node in line 8 represents a list of blocks in the source code.

From the configuration file, `YaCF` generates a set of AST Python classes. Each class represents a node in the AST. With this information the parser can fill in all the information. Listing 19 shows part of the OpenMP parser. Each grammar production features a method and each method

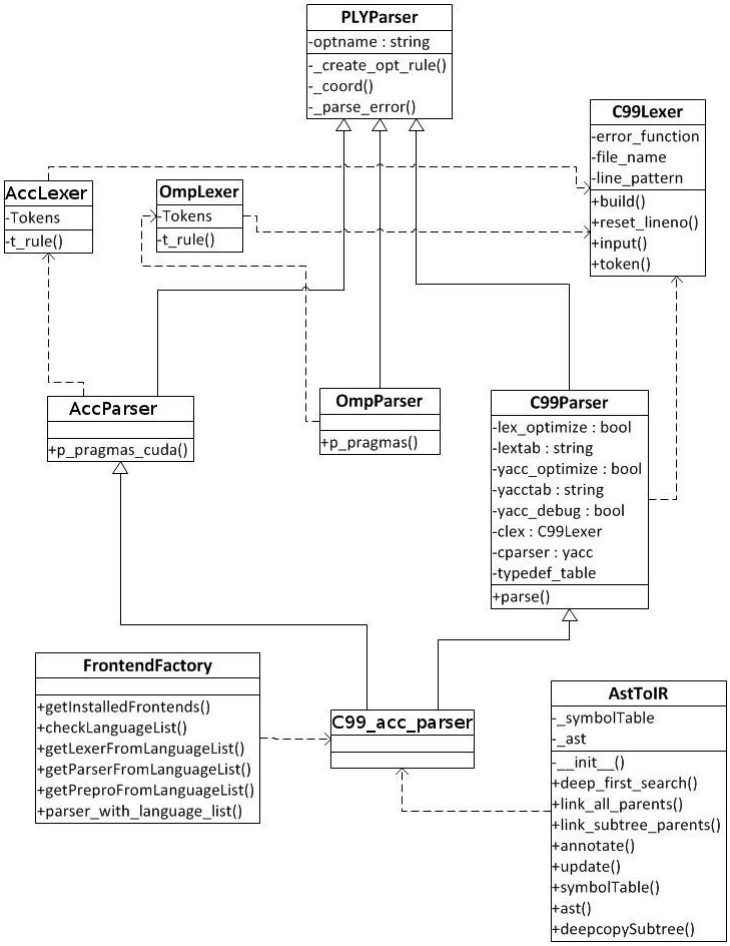

Figure 13

FRONTEND package class hierarchy

```
ArrayDecl: [type*, dim*]
ArrayRef: [name*, subscript*]
Assignment: [op, lvalue*, rvalue*]
BinaryOp: [op, left*, right*]
Break: []
Case: [expr*, stmt*]
Cast: [to_type*, expr*]
Compound: [block_items**]
```

Listing 18

Part of the C99 AST configuration file

```
1 def p_clause_2(self, p):
2   """ clause : REDUCTION LPAREN reduction_operator COLON
       identifier_list RPAREN """
3   p[0] = [omp_ast.OmpClause(type = p[3] , name = 'REDUCTION',
       identifiers = p[5], coord = self._coord(p.lineno(1)))]
4
5 def p_clause_3(self, p):
6   """ clause : NOWAIT """
7   p[0] = [omp_ast.OmpClause(type = str(p[1]) , name = str(p[1]).upper
       (), identifiers = None, coord = self._coord(p.lineno(1)))]
```

Listing 19

Extract from the OpenMP parser of `YaCF` showing the interpretation of the `reduction` and `nowait` clauses

creates the appropriate AST node. For example, the OpenMP `reduction` clause is parsed in line 2 while an AST node with the reduction type and the identifiers is created in line 3.

7. The MIDDLEEND

The input of the package MIDDLEEND is an IR and the output is another IR with the same, or higher, level. The packages in MIDDLEEND are commonly used as for intermediate processing to optimise or prepare codes so the BACKEND packages can proceed. Some packages used for analysis of codes can also be found in the MIDDLEEND.

7.1 Data Dependency Analysis

Some situations might require an analysis of the data dependency in the code. For these situations, `YaCF` features a (basic) Data Dependency Analysis tool. Dependency analysis produces execution-order constraints between statements.

Listing 20 shows a block statement with a typical expression statement. If statement S_1 precedes S_2 in their given execution order, we write $S_1 \lhd S_2$ (Notation from [24]). A dependence between two statements in a program is a relation that constraints their execution order. A control dependence is a constraint that arises from the control flow of the program, such as S_2 with S_3 and S_4 in Listing 20. These dependences are written as $S_1 \delta^c S_2$. A data dependence is a constraint that arises from the flow of data between statements, such as S_3 and S_4 in Listing 20. If we reorder these statements, the result could be incorrect.

Data dependencies can be classified into four types:

1. If $S_1 \lhd S_2$ and the former sets a value that the latter uses, we call this a flow (or true) dependence, and it is written as $S_1 \delta^f S_2$.

2. If $S_1 \lhd S_2$, S_1 uses a particular variable's value and S_2 sets it, then we have an antidependence (written $S_1 \delta^a S_2$).

```
1 S1 a = b + c
2 S2 if (a > 10) goto L1
3 S3  d = b * e
4 S4  e = d + 1
5 S5 L1: d = e / 2
```

Listing 20

Control and data dependency example extracted from [24]

```
1  a[3] = a[5] * a[i];
2  x = h * ((double) i - 0.5);
3  sum += 4.0 / (1.0 + x * x);
```

Listing 21

Example of a SESE block statement with variable dependencies

3. If $S_1 \lhd S_2$, and both of them set the same variable variable then we have an output dependence (written $S_1 \delta^o S_2$).

4. If $S_1 \lhd S_2$, and both of them read a variable then we have an input dependence (written $S_1 \delta^i S_2$).

It is possible to extend the definition of data dependencies from the statements to the variables themselves: let V_1 and V_2 be variables declared in a program; and let there be S an statment of that program containing an expression involving both V_1 and V_2. Expressions can either read variables or write them. If statement S modifies or updates the value of variable V_2 by solving an expression in which V_1 is involved, we can state that V_2 depends on V_1 ($V_1 \delta^f V_2$). Variable dependency, like statement dependency, is transitive, i.e. if V_2 depends on V_1 and V_3 depends on V_2, then V_3 depends also on V_1:

$V_1 \delta^f V_2 \wedge V_2 \delta^f V_3 \Longrightarrow V_1 \delta^f V_3$

A Data Dependency Analysis module is available within `YaCF` (DATAANALYSIS.LLCSCOPE). It is capable of generating a dependency graph of the variables used in a SESE block of statements.

The class *DGraph* implements the dependency graph using a dictionary containing instances of *Dnode*. *Dnode* associates a node of the ST with a list of predecessors and a list of successors. A predecessor of a variable is any variable to which the current variable has a dependency, i.e. if the current variable is V_2, it predecessor list will contain all V_1 satisfying $V_1 \delta V_2$. A successor of a variable is any variable which the current variable creates a dependency, i.e. if the current variable is V_1, the list of sucessors contains all V_2 satisfying $V_1 \delta V_2$.

The *DGraphBuilder* populates a *DGraph* data structure. Each occurrence of an ID creates an instance of *DNode* and adds it to the current *DGraph* instance if it has not already been inserted. When the *DGraphBuilder* visit an assignment expression, the predecessor and successor lists of the nodes involved in the assignment are updated. There is no interprocedural analysis support currently available in `YaCF` . This forces the analysis to assume that all variables passed through

a function call are read and written. It also creates dependencies among them. The algorithm used to populate each *DGraph* instance is shown in Algorithm 3.

Algorithm 3 Analysis of an Assignment node

```
function VISIT_ASSIGNMENT(node, visited_nodes)
    rvalue ← visit(node.rvalue)
    lvalue ← visit(node.lvalue)
    for all r in rvalue do
        for all l in lvalue do
            succ(l) ← succ(l) + l
            pred(r) ← pred(r) + s
        end for
    end for
end function
```

7.1.1 Data Dependency Graph

It is possible to print the dependency graph of the variables inside a block statement whenever it is required by using the *DataDependencyTool*.

Each node of the graph represents a variable used in the block statement. A dependency between V_1 and V_2 is represented by an arrow from the node of V_1 to the node of V_2. If a cycle appears in the graph (i.e. $V_1 \delta^f V_2$ and $V_2 \delta^f V_1$) it means that the variable is being reused, for example, in a reduction.

Figure 14 shows the variable dependency graph generated for the code in Listing 21. Variable `sum` has a cycle over itself in the graph representing the reduction operation. The value of `sum` is computed based on the values of the X variable, which in turn is computed based on values from `h` and the loop variable `i`. A blue color on a node indicates that the variable is read-only, whereas the red color implies that the variable is both read and read-only. A green color (not shown in Figure 14) is used when the variable is only written.

7.1.2 Checking dependencies

Instances of class *DGraph* contain a method *checkDependency*. This method accepts two instances of class *Dnode* as parameters and checks if the first one has a dependency with the second one (i.e. checks if the latter is in the predecessor set of the former or in any of the predecessor sets of any of its predecessors). Transitive dependencies are checked traversing the successor and predecessor lists.

7.2 Loop Analysis

A major part of the optimization effort in any compiler is usually devoted to loops. `YaCF` provides developers with a tool to perform loop analysis (*ParametrizeLoopTool*).

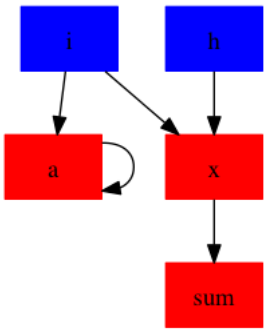

Figure 14

Variable dependency graph for the code in Listing 21

Applying *ParametrizeLoopTool* to a loop node of the IR (i.e. a `For`) node will create a dictionary containing information about the loop. Some restrictions apply. Current implementation limits the parameters captured from a given loop to canonical loop or canonical loop nests. The most relevant parameters extracted by *ParametrizeLoopTool* are:

- `loop_variable`: Variable to iterate.
- `iteration_expression`: Expression to define the next loop step.
- `init_value`: Starting value for the iteration variable.
- `first_iteration`: Value for the first iteration (expression).
- `last_iteration`: Value for the last iteration (expression).
- `loop_stride`: Distance between two iterations (expression).
- `number_of_iterations`: Total number of iterations (expression).

This information is stored in IR format (i.e. subtree expressions). For example, Table 2 details the information extracted by the *ParametrizeLoopTool* from the loop in Listing 22.

Parameter	**Value**
Loop variable	i
Stride	1
Condition Node	$i \leq N$
Last iteration (It_{last})	$N-1$
Number of iterations	$(It_{last} - It_{first})/1$
Iterator	$i+=1$

Table 2

Information extracted from the loop in Listing 22

```
1 for (int i = 0; i < N; i++)
2  a[i] = b[i];
```

Listing 22

A canonical C loop

Notice that some of the parameters extracted are not constant values, but expressions written in the IR language. These expressions will then be rewritten in the original language by a Writer. The final value will be computed at execution time.

7.3 Loop Optimizations

Due to the nature of StS compilers, loop optimizations play a dominant role in code optimization. YaCF implements several loop optimizations as interchange, unswitch, unroll, or tiling. These transformations can be reffered to by different names in the literature, and many of them are also implemented in other StS compilers such as Cetus (Section 2.5) or Mercurium (Section 2.6).

The optimizations available in YaCF have been implemented following the *Mutator* and *Visitor* software patterns [15] in the MIDDLEEND.LOOP.MUTATOR directory. This directory contains the key *Mutators* responsible for carrying out processing on the intermediate code IR-2 (Section 4.). For example, the module LOOPTILING.PY contains the *LoopTilingMutator* which is responsible for implementing a rectangular loop tiling on the AST type supplied, i.e. a tiling with constant tile sizes (see Section 7.3.4 for details).

These *Mutator* make extensive use of tools such as *ParametrizeLoopTool* for handling loops, or those available in the TOOLS.TREE package such as *ReplaceTool*.

In YaCF the programmer is responsible for verifying the safety of certain optimizations which are not always applicable. That is, there is no guarantee that some transformations such as loop tiling, retain the original semantics of the source code after being applied. Ensuring program correctness is the responsability of the YaCF user.

7.3.1 Loop Common

The package COMMON.PY in the MIDDLEEND.LOOP directory contains a number of *Mutators* and *Filters* common to many optimizations and drivers, the *LoopFilter* is one of them. This *Filter* searches for a *For* node in the entire AST.

The default behavior of *LoopFilter* is to iterate over the AST to return all the encountered *For* nodes. The *Filter* can be parametrized with an *identifier* parameter to discriminate on the loop index variable, specifying the loop or loops it will search for.

7.3.2 Loop Interchange

This transformation is the process of exchanging the order of two perfectly nested loops. One major purpose of *loop interchange* is to improve the cache performance for accessing array elements. It is not always safe to exchange the iteration variables due to dependencies between statements

```
double a[N][M], b[N];

for (int i = 0; i < N; i++)
  for (int j = 0; i < M; j++)
    a[i][j] = a[i][j] * b[j];
```

Listing 23

Loop nest before aplying *loop interchange*

```
double a[N][M], b[N];

for (int j = 0; i < M; j++)
  for (int i = 0; i < N; i++)
    a[i][j] = a[i][j] * b[j];
```

Listing 24

The result of applying *loop interchange* to the loop nest in Listing 23

for the order in which they must execute. To determine whether a compiler can safely interchange loops a dependence analysis must first be carried out.

In the basic example shown in Listings 23 and 24 the effect of the transformation can be observed.

In `YaCF`, this effect is the default behaviour of the *LoopInterchangeMutator* when providing it a subtree of the AST containing the outermost loop. Nevertheless, *LoopInterchangeMutator* enables more useful applications such as swapping two nested loops when there are other loops between in a perfect nesting. This feature is useful, particularly when the number of loop nests is greater than two, for implementing some versions or *loop tiling* as we will see in Section 7.3.4.

7.3.3 Strip-mining

Strip-mining, also known as *loop sectioning*, is a loop-transformation technique for enabling SIMD-encodings of loops, as well as providing a means of improving memory performance. By fragmenting a large loop into smaller segments or strips, this technique transforms the loop structure in two ways:

- It increases the temporal and spatial locality in the data cache if the data are reusable in different passes of an algorithm.
- It reduces the number of iterations of the loop by a factor of the length of each "strip", or number of operations being performed per SIMD operation.

Strip-mining is equivalent to a rectangular tiling (see Section 7.3.4) being applied to simple loops. The transformation has been successfully applied to code optimization in vectorial computers [17, 2, 3].

```
1 double a[N];
2
3 for (int i = 0; i < N; i++)
4   a[i] = 0;
```

Listing 25

An example of a simple C canonical loop *zeroing* an array

```
1 double a[N];
2
3 for (int _i_ = 0; i < N; i += B)
4   for (int i = _i_; i < min(_i_ + B, N); i++)
5     a[i] = 0;
```

Listing 26

Loop in Listing 25 after aplying strip-mining with strip size B

Listing 26 shows the effect of applying the *LoopStripMiningMutator* to the loop shown in Listing 25.

In Listing 26 we observe (line 3) that a new loop has been introduced enclosing the original loop. The new loop runs over the original iteration space in blocks of size B. Its index variable sets the start and end of the new inner loop (line 4), which now iterates over each block. As a result of applying *strip-mining*, the iterations will execute in consecutive blocks of size B indexed by `_i_`. The limit `min(_i_ + B, N)` ensures that the new code does not run extra iterations.

As all dependencies of a program are lexicographically positive, *strip-mining* is always safe to apply [36].

7.3.4 Loop Tiling

Loop tiling, also known as *loop blocking* was promoted by Francois Irigoin and Michael Wolfe at the end of the 80s [34, 35]. It is one of the most important iteration reordering loop optimizations. From *loop tiling* it is possible to extract beneficial properties for both parallel machines and for multiple level cache monoprocessors exposing space locality [36]. *Loop tiling* includes those loop otimizations which reorder its iteration space. Some examples of these are *loop interchange*, *loop skewing*, or *strip-mining* among others. All these transformations change the order in which iterations are executed, while preserving the order of the statements into each iteration.

`YaCF` implements square or rectangular tiling [36]. It is named rectangular from the shape of the blocks that run the iteration space when *loop tiling* is applied to a two-dimensional space (two nested loops). Formally, the name is kept for higher dimensions as the geometric properties of the rectangles are also preserved.

In the following paragraphs we briefly describe the algorithm that `YaCF` features for automatic rectangular tiling (for further reading, see also [32]).

```
1 double a[N][N], b[N][N], c[N][N];
2
3 for (int i = 1; i < N; i++)
4   for (int j = 1; j < N; j++)
5     for (int k = 1; k < N; k++)
6       c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

Listing 27

Square matrices product, example of a perfect loop nest

Listing 27 shows a series of three perfectly nested loops. It corresponds to a square matrices multiplication code, and provides a good example of how to obtain the same benefits as those mentioned with *strip-mining* in Section 7.3.3, but now applied to nested loops.

The first optimization used by loop tiling is *strip-mining*. By applying *strip-mining* to each of the loops in Listing 27 (lines 3, 4 and 5) three new loops are created as shown in Listing 28 (lines 3, 5 and 7) and the bounds of the original loops (lines 4, 6, and 8) are changed as described in Section 7.3.3 .

So, *strip-mining* itself can produce a program for the matrix product using 6 loops. Nevertheless, the order in these loops is not correct for our purposes, as it does not correspond to a matrix product algorithm with a blocked iteration space. *Strip-mining* itself does not split a nested loop iteration space into strips.

The second transformation we are going to apply is *loop interchange*. To obtain a correct code, we need to move the new loops generated by `strip-mining` outwards, and move the original loops inwards.

This is achieved with *loop interchange* and the resulting code is shown in Listing 29.

Note that it is safe to apply *loop interchange* for the matrix product code, but it is not for many other useful programs including some partial differential equation methods.

Algorithm 4 Automatic *loop tiling* in `YaCF`

function LOOPTILINGMUTATOR($ast, indexes, sizes$)
  for $i = 0$ to $length(indexes) - 2$ **do**
    $loop \leftarrow LoopFilter(ast, indexes[i])$
    $LoopStripMiningMutator(loop, sizes[i])$
    $LoopInterchangeMutator(loop, indexes[i], indexes[i+1])$
  end for
  $loop \leftarrow LoopFilter(ast, indexes[-1])$
  $LoopStripMiningMutator(loop, sizes[-1])$
  for $i = length(indexes) - 2$ to 0 **do**
    $LoopInterchangeMutator(ast, indexes[i+1], indexes[i])$
  end for
end function

```
double a[N][N], b[N][N], c[N][N];

for (int _i_ = 1; _i_ < N; _i_+= B)
for (int i = _i_; i < min(N, _i_ + B); i++)
  for (int _j_ = 1; _j_ < N; _j_+= B)
  for (int j = _j_; j < min(N, _j_ + B); j++)
    for (int _k_ = 1; _k_ < N; _k_+= B)
    for (int k = _k_; k < min(N, _k_ + B); k++)
      c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

Listing 28

Square matrices product with its loops incorrectly ordered after applying strip-mining

```
double a[N][N], b[N][N], c[N][N];

for (int _i_ = 1; _i_ < N; _i_+= B)
for (int _j_ = 1; _j_ < N; _j_+= B)
for (int _k_ = 1; _k_ < N; _k_+= B)
  for (int i = _i_; i < min(N, _i_+B); i++)
    for (int j = _j_; j < min(N, _j_+B); j++)
      for (int k = _k_; k < min(N, _k_+B); k++)
        c[i][j] = c[i][j] + a[i][k] * b[k][j];
```

Listing 29

The effect of applying square *loop tiling* with tile size B to the code in Listing 27

```
...
for (int i = 0; i < N ; i++)
   if (b[i] != 0)
     a[i] = b[i] * c[i]
   else
     break;
...
```

Listing 30

An example of a loop body with a `break` statement

Algorithm 4 describes the implementation of tiling by `YaCF`. The algorithm receives the AST subtree corresponding to the outermost loop, an index list of the loops to be processed, and the block sizes to apply when processing each loop in the former list. It is worth noticing that it is common for sizes to be different to each loop.

The reader may observe that it is possible to obtain a similar code to that shown in Listing 29 by repeatedly applying *strip-mining* and *loop-interchange* over each nested loop in the code shown in Listing 27. Repeatedly applying *strip-mining* and *loop-interchange* corresponds to the first `for` loop and the last *LoopStripMiningMutator* in Algorithm 4.

Now, the only thing that remains to be done is reorder the series of inner loops to restore their original order. To obtain the final code (Listing 29) it is enough to apply several *LoopInterchangeMutator* transformations to the loop list inversely. That process is performed by the second loop in Algorithm 4.

7.4 The *Outliner*

To facilitate certain transformations or specific analysis phases, it is easier to extract a piece of code from a code block to an external function, a process which is called Outlining. Although this might be seen as a trivial task, it is critical to take into account several side effects that might occur while applying this code motion technique. For example, in C, any variable used inside a block whose value is modified has to be passed by reference. This ensures that the value modified in the function is the original one and not a copy residing in the stack.

It is also important to take into account potential breaks in the code. For example, suppose that we want to extract the body of the loop in Listing 30 to an external function (i.e. each iteration of the loop will call the outlined function). This will generate the code in 31. However, this code is not correct as the break statement is not inside a `for` loop. The compiler has to analyse the loop and provide a proper replacement to keep the original code working. In this case, we have decided to replace the `break` statements by `return` statements with a known value, and to check inside the loop body for that return value.

More complex situations can arise when labels and jump statements (i.e. `GOTO` statements) are involved. For the sake of simplicity, in the current section we will assume that the code segment that we want to outline is Single entry - Single exit block (SESE).

The class *OutlinerMutator* from the package MIDDLEEND.FUNCTIONS implements a function outliner. This *Mutator* creates a new function, called the outlined function. The body of the

```
void outlined_function(int * a, int * b, int * c, int i)
{
   if (b[i] != 0)
     a[i] = b[i] * c[i];
   else
     break;
}
...
for (int i = 0; i < N ; i++)
  outlined_function(a,b,c,i);
```

Listing 31

Incorrect extraction of the loop body. The `break` statement is no longer syntactically correct.

outlined function will be a copy the block statement that we want to outline. It also replaces the original (passed by reference) block in the IR by a function call to the previously created outlined function. The *Mutator* returns a reference to this new call inside the original tree, and the method *get_outlined_function* can be used to retrieve the outlined function. The *Mutator* has been designed so the outlined function can be written onto an external file, and compiled independently. The outlined function can be injected on the original AST, or not, depending on the instance parameters.

It is also possible to retrieve only the function definition, or the type declarations required to define the variables used inside the function body. Constructor-time parameters enable developers to instruct the *Mutator* to not replace the original code and just extract the outlined function, or to change the name of the outlined function.

To avoid replacing all occurrences of the variables in the outlined function by references to the parameters, the *Mutator* takes advantage of the scope nesting to create new local replacements of the parameter variables. However, this changes the default C pass-by-reference parameter passing scheme to a copy-and-restore, so this transformation can produce side-effects in some situations, particularly when using threads. Instead of using this transformation, it is possible to use a traditional replacement of all occurences of R/W parameters by references by changing a parameter in the constructor. The final decision is left up to the `YaCF` user.

8. The BACKEND

The BACKEND package contains transformations whose destination is a programming language. These transformations usually operate over the IR and then use a Writer to generate a final code.

All subpackages under this package follow the same structure. The subpackage is named after the destination target or platform (for example, a back end named Cuda will generate CUDA code). If the back end is meant to be run alone (it is not part of another), it has to contain a *Runner* class that specifies how to create the destination code. Also, inside this subpackage, there must be a FILTER directory containing the *Filters* used, a MUTATORS directory with the *Mutators* used, and a WRITERS directory if any particular Writer is required. A TEST directory may appear also, containing testing scripts for each back end implementation. Documentation of the back end has

```
1   <%
2      from Backends.Common.TemplateEngine.Functions import decl_of_id
3   %>
4   int ${functionName} ( ${','.join(loop_parameters['inside_vars'])} )
        {
5           ## This function is just a test
6           %for elem in loop_parameters['inside_vars']:
7              ${decl_of_id(elem)}
8           %endfor
9
10        }
```

Listing 32

Example of template mixing C and Python code with the template tags

to be available on the `init` file of the package. If the back-end uses any template file, it must be stored under the TEMPLATE directory.

A set of common classes for *Visitors*, *Filters* and *Transformers* are stored in the COMMON sub-package. These common classes have been already described in Section 3.. Commonly used Filters are defined in the GENERICVISTIORS module whereas common used *Mutators* are defined in the ASTSUPPORT module.

8.1 The Template Subsystem

When working in StS transformations, there are situations where the back-end writer has to fill a pattern written in the destination language with parameters of the current code.

Using the IR manipulation tools, the process will require several insertions and replacements of nodes, together with manually created instances of nodes to recreate the tree. If the parameters of the library change at any point, or we want to implement additional operations, it would be necessary to re-write most of the code as the new IR will be different.

To facilitate the manipulation of the IR, it is possible to create a template. A template is a Python string with placeholders for variables, indicated by `${...}$`. The template engine (Mako [22] in the current version of `YaCF`) will replace these placeholders with the values of the variables at runtime, and generate a new string with the information. This string can be parsed with the *parse_snippet* method of the *Mutator* class, generating a new IR.

An example of a template is shown in Listing 32. Placeholders can also be used to insert raw Python code, that will be evaluated when parsing the template (see Line 4). Python functions used inside templates can be declared on external modules (Line 2 for the import, Line 7 for the usage). Some commonly used functions are available on the FUNCTIONS module of the TEMPLATEENGINE directory. Template comments (i.e. code that will not be evaluated by the template engine and that will not appear on the destination code) can be specified using double hash `(##)` before the text (Line 6). More complex control structures, like if/else or `for` statements (Line 5) are available. These loop control structures enable back-end writers to express complex code patterns in the source code in a clear and readable way.

```
1   self.parse_snippet(template_code, {'reduction_vars' :
       reduction_vars,
2                  'shared_vars' : shared_vars}, name = 'Retrieve',
                      show = False)
```

Listing 33

Calling the *parse_snippet* from the *Mutator* to generate the AST of the code after filling the template

To parse a template, the user has to specify all the variables of the template as it is shown in Listing 33.

The `name` and `show` parameters are used for debugging purposes. In the event of an error parsing the template, an exception is raised. If the show parameter is set to `True`, the template is also printed to the standard output.

The code generated by the template must be syntactically correct thus, attention is required in situations where the snippet might not form a valid C code.

8.2 The DOT Back end

To facilitate understanding the IR representation, and to alleviate the development effort when creating source transformations, it is necessary to provide developers with tools to represent the IR at any point. A DOT back end has been implemented to facilitate this task.

DOT [33] is a plain text graph description language. The language allow to describe graphs that both humans and computer programs can use.

The DOT back end contains a *Visitor* which creates a representation of the graph in the DOT language. Nodes in the DOT graph are nodes of the DOT language. Arcs in the DOT graph represent the relation between IR nodes (which node contains which one).

This back-end is used by the *DOTDebugTool* to create snapshots of the translation process and debug the internal IR. However, it can be used standalone to print the results of a StS translation to a file, or as part of any other tool. Listing 34 shows part of the output of the DOT commands used to generate the Figure 9.

8.3 The Writer Classes

A Writer is a class implementing a *Visitor* pattern which traverses the IR generating a source code. This source code could be the original, like in a C-to-C translator, or a different one. Writers can be applied to any AST subtree, although some implementations of the Writer might require features only available on augmented IR.

As is typical in other classes following the Visitor pattern, each method *visits* an element of the AST/IR, and performs an action.

8.3.1 OffsetWriter

When unparsing codes (i.e. recovering the original source from the IR), it is interesting to recover not only the source code but some part of the original indenting, or at least make a best-effort

```
digraph G {
FileAST__DOT__0[label = "FileAST"];
FileAST__DOT__0 [shape=box, color=red, style=filled];
FuncDef__DOT__1[label = "FuncDef"];
FileAST__DOT__0-> FuncDef__DOT__1[label = "ext"];
Compound__DOT__2[label = "Compound"];
FuncDef__DOT__1-> Compound__DOT__2[label = "body"];
i_3[label = "i"];
Compound__DOT__2-> i_3[label = "block_items"];
TypeDecl__DOT__4[label = "TypeDecl"];
i_3-> TypeDecl__DOT__4[label = "type"];
int_5[label = "int"];
TypeDecl__DOT__4-> int_5[label = "type"];
int_5-> int[label = "names"];
n_7[label = "n"];
Compound__DOT__2-> n_7[label = "block_items"];
Constant__DOT__8[label = "Constant"];
n_7-> Constant__DOT__8[label = "init"];
TypeDecl__DOT__9[label = "TypeDecl"];
n_7-> TypeDecl__DOT__9[label = "type"];
TypeDecl__DOT__9-> int[label = "type"];
pi_11[label = "pi"];
Compound__DOT__2-> pi_11[label = "block_items"];
TypeDecl__DOT__12[label = "TypeDecl"];
pi_11-> TypeDecl__DOT__12[label = "type"];
double_13[label = "double"];
TypeDecl__DOT__12-> double_13[label = "type"];
double_13-> double[label = "names"];
sum_15[label = "sum"];
Compound__DOT__2-> sum_15[label = "block_items"];
TypeDecl__DOT__16[label = "TypeDecl"];
sum_15-> TypeDecl__DOT__16[label = "type"];
TypeDecl__DOT__16-> double[label = "type"];
...
```

Listing 34

Example of the *DOT* Back end output

attempt to generate readable code. To accomplish this task, a basic class from where writers can be inherited is available. *OffsetWriter* offers basic features to write strings to a file descriptor (standard output by default) providing an offset value. When writing the output, offset number of spaces will be prepended to the string.

8.3.2 C99Writer

C99Writer unparses a C99 [23] IR back to C99 code. Some considerations have to be taken into account:

- Comments are not restored as they are not in the IR.
- Original parenthesis are lost during parsing, Writer has its own rules to apply parenthesis to expressions.
- It does not handle `pragma` statements.

8.3.3 OmpWriter

OmpWriter unparses a C99 IR with OpenMP [27] annotations back to the original. The considerations are the same as those mentioned above, except that any OpenMP pragma is printed conforming 3.0 revision of the standard ([26]).

8.4 The CUDA Back end

The CUDA module features a set of *Mutators* capable of generating CUDA code from OpenMP sources. In the following paragraphs we focus on the components that are currently supported: The *Kernelize* and the *Platform* classes. In order to extract and write a CUDA kernel from a Loop first the Loop parameters need to be extracted using the *ParametrizeLoopTool* as described in Section 7.2.

8.4.1 Platform

The variables used in the loop have to be classified to decide its location in the device memory. Table 3 shows how input variables are transformed.

Kernel Parameter	**Value**
Kernel Parameters	All variables used in the kernel except from register variables
Reduction Variables	Variables in the reduction list of the loop parameters
Registers	All variables declared inside the kernel
Code	Loop body

Table 3

Placement of variables according to loop information

The class *Platform* also performs several analysis over the kernel code to facilitate optimization on further phases. Three parameters are extracted from the analysis: **Number of Flops**, **Number of Memory Accesses** and **Divergence factor**.

8.4.2 Kernelize

The *Kernelize* receives the parameters of the kernel (as returned by `Platform`) and writes the final device kernel. Writing the kernel makes use of the Template Subsystem. Templates are filled with the parameters. Some convenience functions are implemented on the template layer to convert Symbols into different forms, such as declarations, parameters or pointers. These handler functions, stored in BACKENDS.COMMON.TEMPLATEENGINE.FUNCTIONS work on Symbol nodes and are able to make basic representation transformations on the template itself. In addition to writing the kernel, *Kernelize* also calls the *inlineCalledFunctions* to ensure that the functions called from the kernel are inlined and ensures that the only functions called but not defined are native functions of the device (such as `sqrt` or `exp`). A configuration parameter enables this method to replace the precise version of these functions by a less precise, but faster, implementation on the device hardware. *Kernelize* uses separate kernels for different situations, depending on the kind of kernel (1D, 2D or 3D kernels) or if there are reductions involved.

8.5 The OPENCL Back end

In a similar fashion to the CUDA Backend, the OpenCL back end implements the generation of OpenCL code. The structure of the back end is the same. The *Platform* is just a placeholder to call the CUDA version as the parameters required are the same. The **Kernelize** implements OpenCL specific semantics using a different set of Templates for each different class of kernel.

9. Final Remarks

In this Chapter we have provided a detailed description of our StS tool: `YaCF`. The `YaCF` tool has enabled us to fast-prototype languages, techniques and optimizations with very little development effort on our part, and it has provided us with a steady learning curve. `YaCF` is not a production-ready compiler. Transformations are not entirely safe and might generate incorrect code. However, the tool suffices for a controlled research environment in which the developer is focused on the features offered, rather than on its completeness.

Acknowledgments

This work has been partially supported by the EU (FEDER), the Spanish MEC (Plan Nacional de I+D+I, contracts TIN2008-06570-C04-03 and TIN2011-24598), HPC-EUROPA2 (project number 228398) and the Canary Islands Government, ACIISI (contract PI2008/285).

References

[1] "High Performance Computing Group at La Laguna University," `http://cap.pcg.ull.es/`. 2

[2] R. Allen and K. Kennedy, "Automatic Translation of Fortran Programs to Vector Form," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 4, 1987, pp. 491–542. 38

[3] R. Allen and K. Kennedy, "Vector Register Allocation," *IEEE Trans. Computers*, vol. 41, no. 10, 1992, pp. 1290–1317. 38

[4] E. Bendersky, "Pycparse," 2009, http://code.google.com/p/pycparser/ [Online; Last accessed October 2012]. 17, 18, 19, 30

[5] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Narayana, S. Srinivas, and B. Winnicka, "Sage++: An Object-Oriented Toolkit and Class Library for Building Fortran and C++ Restructuring Tools," *In The second annual object-oriented numerics conference (OON-SKI)*, 1994, pp. 122–136. 10

[6] S. C. Chan, G. R. Gao, B. Chapman, T. Linthicum, and A. Dasgupta, "Open64 compiler infrastructure for emerging multicore/manycore architecture All Symposium Tutorial.," *IPDPS*. 2008, p. 1, IEEE. 4

[7] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, Oct. 1991, pp. 451–490. 8

[8] C. Dave, H. Bae, S.-J. Min, S. Lee, R. Eigenmann, and S. Midkiff, "Cetus: A Source-to-Source Compiler Infrastructure for Multicores," *Computer*, vol. 42, no. 12, 2009, pp. 36–42. 12

[9] A. J. Dorta, J. M. Badía, E. S. Quintana, and F. de Sande, "Implementing OpenMP for clusters on top of MPI," *Proc. of the 12th European PVM/MPI Users' Group Meeting*, Sorrento, Italy, September 18–21 2005, vol. 3666 of *LNCS*, pp. 148–155, Springer-Verlag. 2

[10] A. J. Dorta, J. M. Badía, E. S. Quintana, and F. de Sande, "Parallelizing Dense Linear Algebra Operations with Task Queues in `llc`," *Proc. of the 14th European PVM/MPI Users' Group Meeting*, Paris, France, 2007, Lecture Notes in Computer Science, Springer-Verlag. 2

[11] A. J. Dorta, J. A. González, C. Rodríguez, and F. de Sande, "Towards Structured Parallel Programming," *Proc. Fourth European Workshop on OpenMP (EWOMP 2002)*, Rome, Italy, September 2002. 2

[12] A. J. Dorta, J. A. González, C. Rodriguez, and F. de Sande, "`llc`: A parallel skeletal language," *Parallel Processing Letters*, vol. 13, no. 3, September 2003, pp. 437–448. 2

[13] A. J. Dorta, P. López, and F. de Sande, "Basic skeletons in `llc`," *Parallel Computing*, vol. 32, no. 7–8, September 2006, pp. 491–506. 2

[14] R. Ferrer, A. Duran, X. Martorell, and E. Ayguad, "Unrolling Loops Containing Task Parallelism," *Languages and Compilers for Parallel Computing, 22nd International Workshop, LCPC 2009, Newark, DE, USA*, G. R. Gao, L. L. Pollock, J. Cavazos, and X. Li, eds. Oct. 2009, vol. 5898 of *Lecture Notes in Computer Science*, pp. 416–423, Springer. 14

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, USA, 1994. 23, 37

[16] B. J. Gough and R. M. Stallman, *An Introduction to GCC*, Network Theory Ltd., 2004. 3

[17] D. J. Kuck, K. R. H. L. B, and M. Wolfe, "The structure of an advanced vectorizer for pipelined processors," 1980. 38

[18] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, Washington, DC, USA, 2004, CGO '04, pp. 75–, IEEE Computer Society. 7

[19] C. Liao, D. J. Quinlan, T. Panas, and B. R. de Supinski, "A ROSE-Based OpenMP 3.0 Research Compiler Supporting Multiple Runtime Libraries," *IWOMP*, 2010, pp. 15–28. 10

[20] P. López, A. J. Dorta, E. Mediavilla, and F. de Sande, "Generation of Microlensing Magnification Patterns with High Performance Computing Techniques," *Applied Parallel Computing. State of the Art in Scientific Computing. Proceedings of the 8th International Workshop, PARA 2006, Umeä, Sweden, June 18-21, 2006*, Umeä, Sweden, 2007, vol. 4699 of *Lecture Notes in Computer Science*, pp. 351–360, Springer-Verlag. 2

[21] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003. 19

[22] J. McNeil, *Python 2. 6 Text Processing Beginner's Guide*, Packt Publishing, Limited, 2010. 44

[23] R. Meyers, "Introducing C99," *C/C++ Users Journal*, vol. 18, no. 10, Oct. 2000, pp. 49–53. 47

[24] S. S. Muchnick, *Advanced compiler design and implementation*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997. 33, 34

[25] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable Parallel Programming with CUDA," *Queue*, vol. 6, no. 2, 2008, pp. 40–53. 2

[26] OpenMP Architecture Review Board, *OpenMP Application Program Interface v. 3.0*, May 2008. 47

[27] OpenMP Architecture Review Board, "OpenMP Official Web site," 2012, http://www.openmp.org [Online; Last accessed October 2012]. 47

[28] D. A. Padua, R. Eigenmann, J. Hoeflinger, P. Petersen, P. Tu, S. Weatherford, and K. Faigin, *Polaris: A New-Generation Parallelizing Compiler for MPPs*, Tech. Rep. CSRD 1306, Univ. of Illinois at Urbana-Champaign, 1993. 12

[29] Q. Parr, T.J., "ANTLR: a predicated-LL(k) parser generator," *Software - Practice and Experience*, vol. 25, no. 7, 1995, pp. 789–810. 12

[30] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta, "Hierarchical Task-Based Programming With StarSs," *IJHPCA*, vol. 23, no. 3, 2009, pp. 284–299. 14

[31] J. Rodríguez-Rosa, A. J. Dorta, C. Rodríguez, and F. de Sande, "Exploiting task and data parallelism," *Proc. of the Fifth European Workshop on OpenMP (EWOMP 2003)*, Aachen, Germany, September 2003, pp. 107–116. 2

[32] R. Schreiber and J. Dongarra, *Automatic Blocking of Nested Loops*, Technical report CS-90-108, NASA Ames Research Center, May 1990. 39

[33] M. Simionato, *An Introduction to GraphViz and dot*, O'Reilly Community Press, 2004. 45

[34] M. Wolfe, "Iteration Space Tiling for Memory Hierarchies," *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, Philadelphia, PA, USA, 1989, pp. 357–361, Society for Industrial and Applied Mathematics. 39

[35] M. Wolfe, "More iteration space tiling," *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, New York, NY, USA, 1989, Supercomputing '89, pp. 655–664, ACM. 39

[36] J. Xue, *Loop Tiling for Parallelism*, Kluwer International Series in Engineering and Computer Science. Kluwer Academic, 2000. 39