Mercurium

A source-to-source compiler

Patrick Ziegler

RWTH Aachen

patrick.ziegler@rwth-aachen.de

July 11, 2016

Goal

- Become familiar with Mercurium
- Understand the structure of the compiler
- Be able to design extensions for C/C++

Overview

What is Mercurium?

Mercurium is a source-to-source compiler:

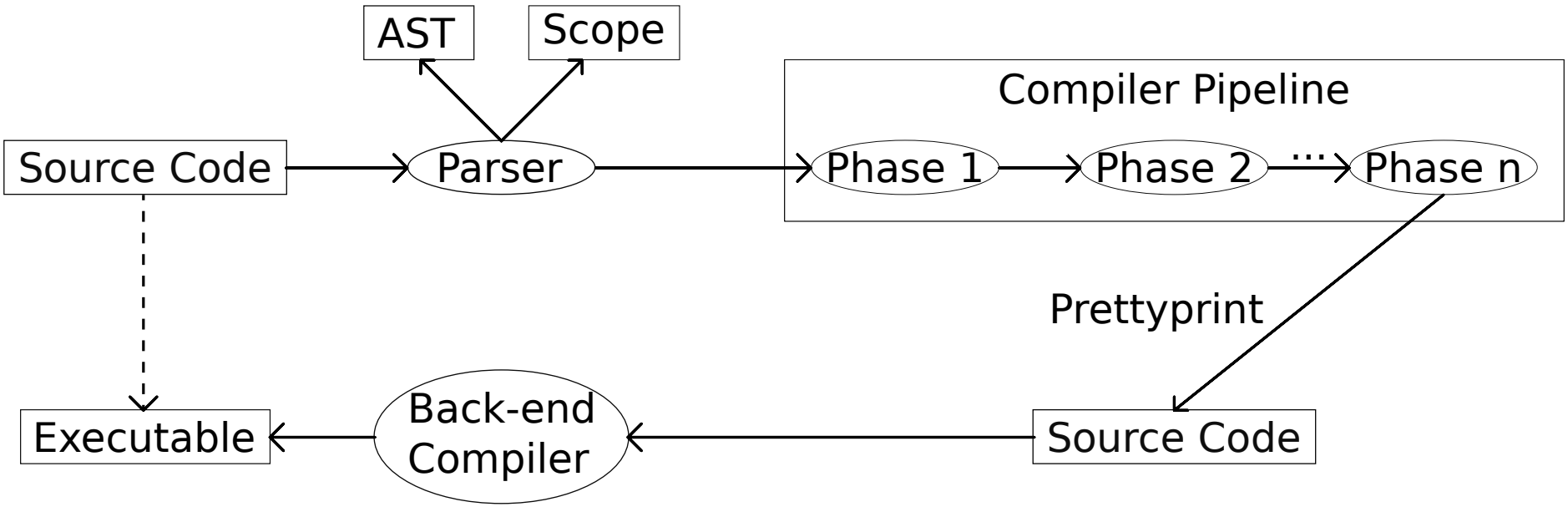
- Developed by the Barcelona Supercomputing Center
- Only alters the source code
- Compilation done by an underlying compiler
- Multiple phases, each altering the source code

What is it used for?

- "Easy" implementation of new features
- Write phases on a source code level
- Flexible behavior of the compiler

No compilation.
No optimization.
No platform related tasks.
There are already good compilers for these tasks.

Compiler design



Scope

The "context" of the current program:

- A lookup table for all the functions, variables etc.
- In general, more than one

Example:

```
extern int read();
int func(void){
  int x = read();
  int i, j=0;
  for(i=0;i<x;++i)
    j+= i*x;
  return j;
}
```

Name	Type
read	function,int
func	function,int
x	int
i	int
j	int

Abstract Syntax Tree

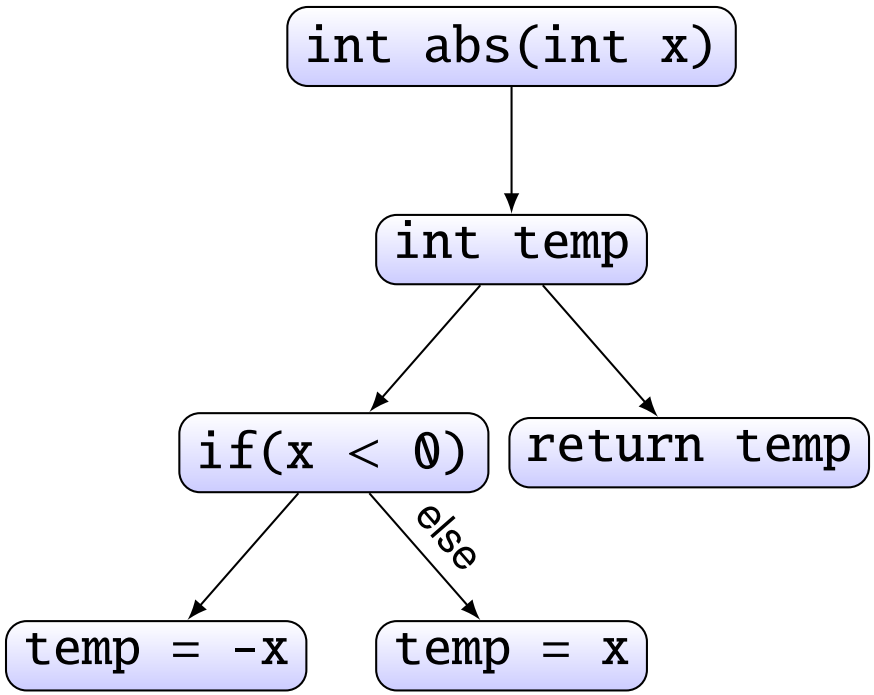
Contains the syntax of the source code:

- Expressions are tokenized
- Hierarchical order to represent flow

In combination with the scope ⇒ source code.

Example:

```
int abs(int x){
  int temp;
  if(x < 0)
    temp = -x;
  else
    temp = x;
  return temp;
}
```



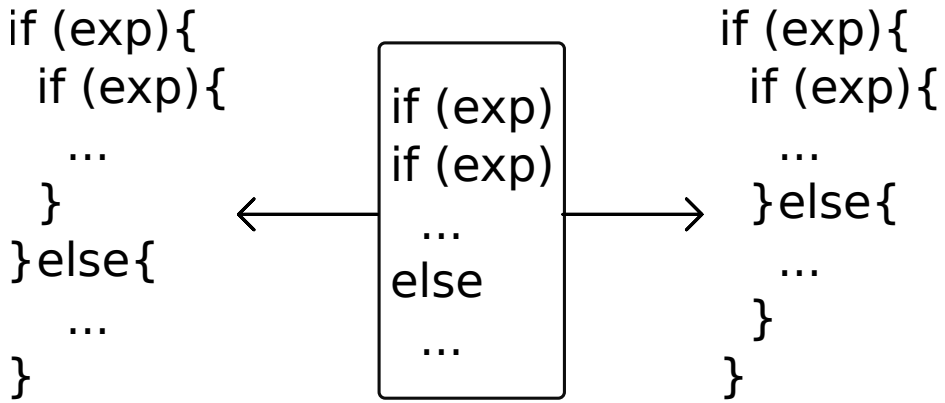
Abstract Syntax Tree

Ambiguity will be fixed during the parsing process.

Example : dangling else problem

The grammar doesn't dictate, which expression has to be used.

$\Rightarrow$ C links the else always to the nearest if



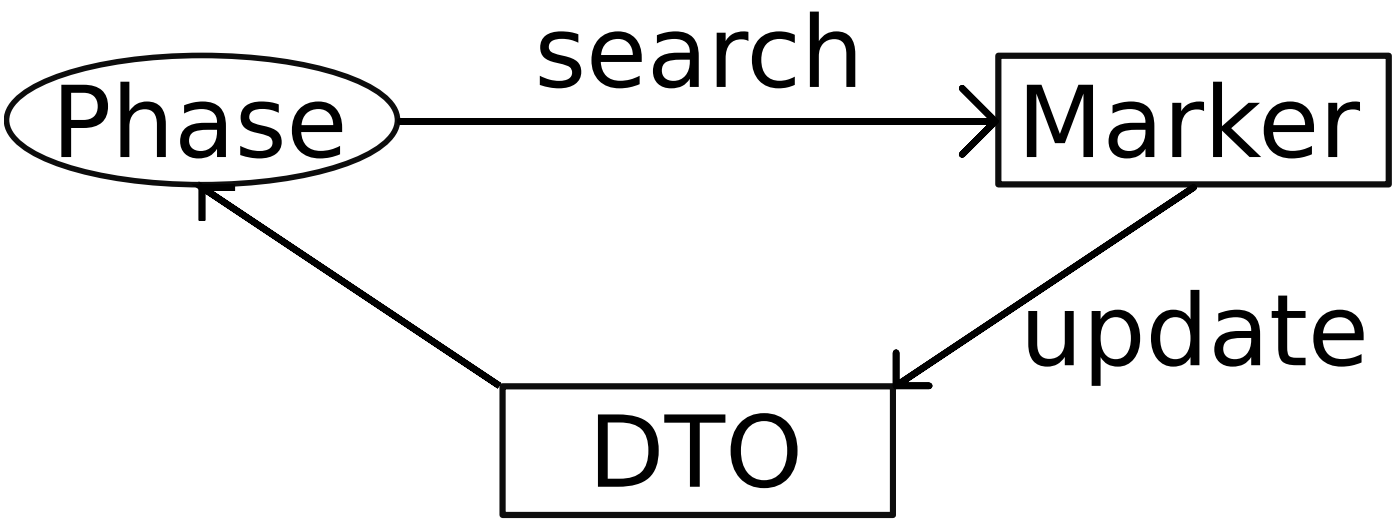
Compiler phases

Each phase is a dynamically loaded library

- The libraries are written in C++
- Backed by their own SDK
- They receive the current AST and Scope
- A code transformation is done by altering the AST
- The altered AST and Scope will be passed to the next phase ⇒ pipeline

⇒ The source code will be modified on the fly

Phase strategy

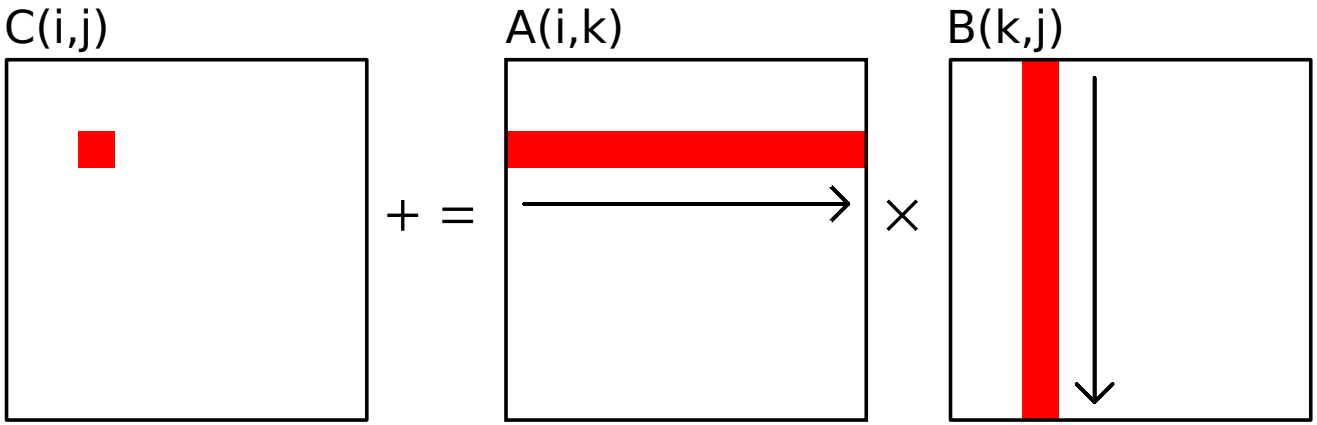


Matrix-matrix multiplication

- $2n^3$ computations
- $3n^2$ data accesses

Problem : Cache mismatches for large matrices

Solution : Reuse of cached data (through blocking)



Example

Multiply two 1000×1000 matrices:

```
int xx,yy,kk,x,y,k;
for(xx = 0; xx < 1000; xx+=4)
  for(yy = 0; yy < 1000; yy+=4)
    for(kk = 0; kk < 1000; kk+=4)
      for(x = xx; x < (1000 <= xx+4 ? 1000 : xx + 4); ++x)
        for(y = yy; y < (1000 <= yy+4 ? 1000 : yy + 4); ++y)
          for(k = kk; k < (1000 <= kk+4 ? 1000 : kk + 4); ++k)
            C[y][x] += A[y][k] * B[k][x];
```

Pretty ugly and prone to errors!

Example

Much better:

```
int x,y,k;
#pragma hlt block(4,4,4)
for(x = 0; x < 1000; ++x)
  for(y = 0; y < 1000; ++y)
    for(k = 0; k < 1000; ++k)
      C[y][x] += A[y][k] * B[k][x];
```

The compiler will do the work for us.

Design

- Traverse through the AST and look for #pragma htl block(...)
- Get the block sizes and the loop statements
- Create a blocked version for each loop
- Order the blocked loops
- Replace the old version with the new one

Note:The traversing is already done by the phase.

Implementation

Get all the information out of the pragma:

Pragma line Parameters

```
#pragma hlt block(...)
{
...     Statement
}
```

```
void HLTPragmaPhase::do_loop_block(TL::PragmaCustomStatement construct){
  Nodecl::NodeclBase loop_body = get_statement_from_pragma(construct);
  TL::PragmaCustomLine custom_line = construct.get_pragma_line();
  TL::PragmaCustomParameter clause = custom_line.get_parameter();
  TL::ObjectList<Nodecl::NodeclBase> block_sizes = clause.get_arguments_as_expressions();
  ...
}
```

Implementation

There is metadata between the loops, so we can't just get the successor.

⇒ Visit all nodes in the statement and look for for-loops.

```
class LoopVisitor : ExhaustiveVisitor<void>{
  TL::ObjectList<Nodecl::ForStatement> loops;
  virtual void visit(const Nodecl::ForStatement& node){
    loops.append(node);
    walk(node.get_statement());
  }
  public :
  LoopVisitor(Nodecl::NodeclBase initial_point){
    walk(initial_point);
  }
};
```

Implementation

Split each loop into two loops. One for the block, the other for the entries.

```
for(unsigned int i=0;i<this->block_sizes.size();++i){
  current_loop = loops[i];
  ...
  //MIN(a,b) = a < b ? a : b
  a = "("+upper_bound+")";
  b = "("+var_name+var_name+"+"+blocksize+")";
  min = "(␣(" + a + "␣<␣" + b + "␣)␣?␣" + a + "␣:␣" + b + "␣)";
  TL::Source outer_loop,inner_loop;
  outer_loop << "for␣(␣"
             << declaration+var_name+"␣=␣"+lower_bound+";"
             << var_name+var_name+"<="+upper_bound+";"
             << var_name+var_name+"+="+blocksize+")"
             ;
  inner_loop << "for␣(␣"
             << declaration + "␣=␣"+var_name+var_name+";"
             << var_name+"<="+min+";"
             << var_name+"+="+step+")"
             ;
  outer_loops.append(outer_loop);
  inner_loops.append(inner_loop);
}
```

Implementation

Create the blocked version, parse it and replace the #pragma expression.

```
TL::Source complete_loop;
for(unsigned int i=0;i<this->block_sizes.size();i++){
  complete_loop << outer_loops[i];
}
for(unsigned int i=0;i<this->block_sizes.size();i++){
  complete_loop << inner_loops[i];
}
complete_loop << current_loop.get_statement().prettyprint();
constuct.replace(complete_loop.parse_statement(scope,Source::DEFAULT));
```

It works!

The intermediate file:

```
for (int xx = 0; xx <= 999; xx += 4)
{
  for (int yy = 0; yy <= 999; yy += 4)
  {
    for (int kk = 0; kk <= 999; kk += 4)
    {
      for (int x = xx; x <= (999 < xx + 4 ? 999 : xx + 4); x += 1)
      {
        for (int y = yy; y <= (999 < yy + 4 ? 999 : yy + 4); y += 1)
        {
          for (int k = kk; k <= (999 < kk + 4 ? 999 : kk + 4); k += 1)
          {
            C[y][x] += A[y][k] * B[k][x];
          }
        }
      }
    }
  }
}
```

Summary

- Problem-specific templates
- Improved readability without loosing performance
- Transformation done by the compiler

Outlook

- Parallelization
- Memory-dependent data
- While-loops
- Recursion