Compiler optimization in C++

Jens Hoffmann

Chair of Algorithms and Data Structures
Albert-Ludwigs University Freiburg

Seminar Java vs. C++, winter semester 2010

Outline

Outline

Outline

Compiler structure

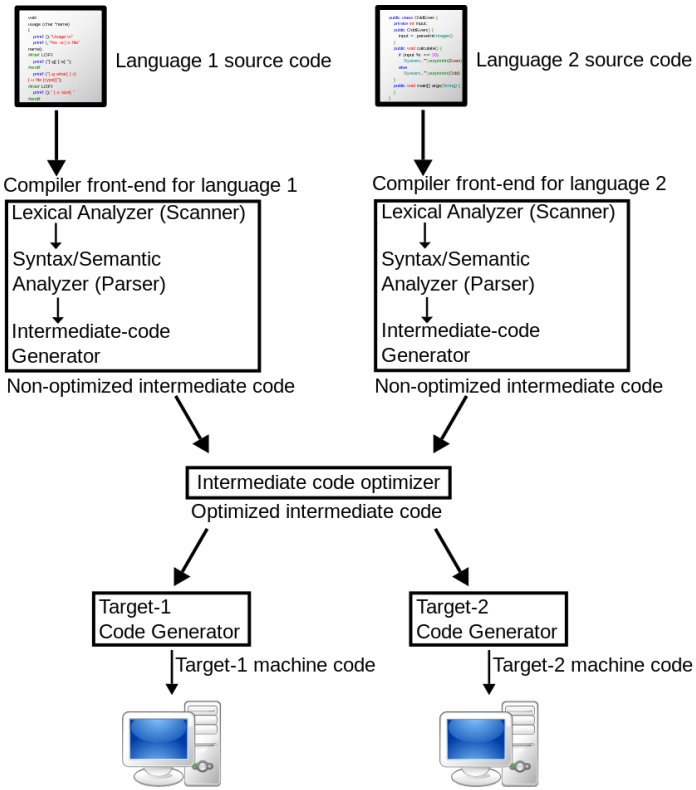

Figure: Generic compiler

- Frontend checks the high-level source code in terms of syntax and semantics.
- Middle-end is where the **optimizations** take place.
- Backend translates **intermediate representation (IR)** into the target assembly code.

Compiler structure

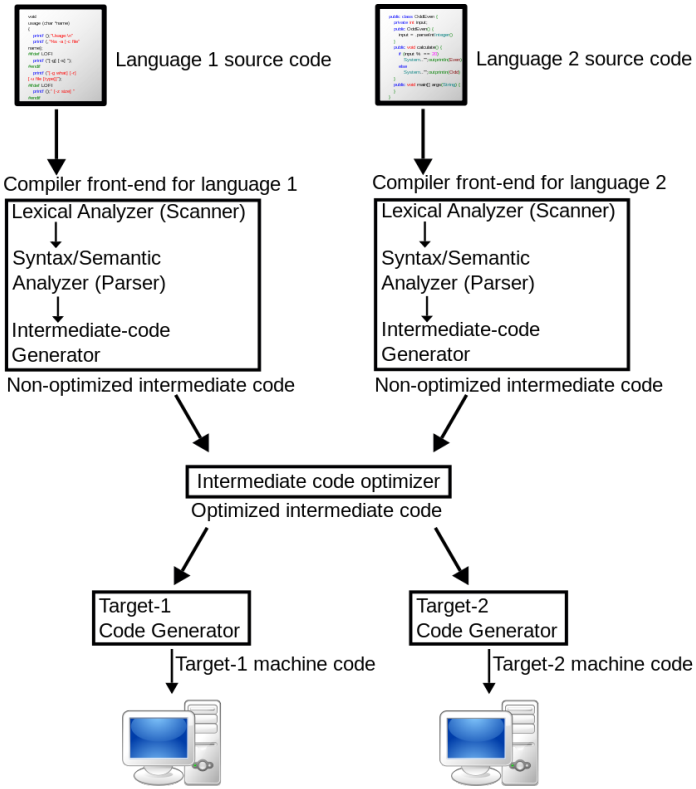

Figure: Generic compiler

Compiler structure

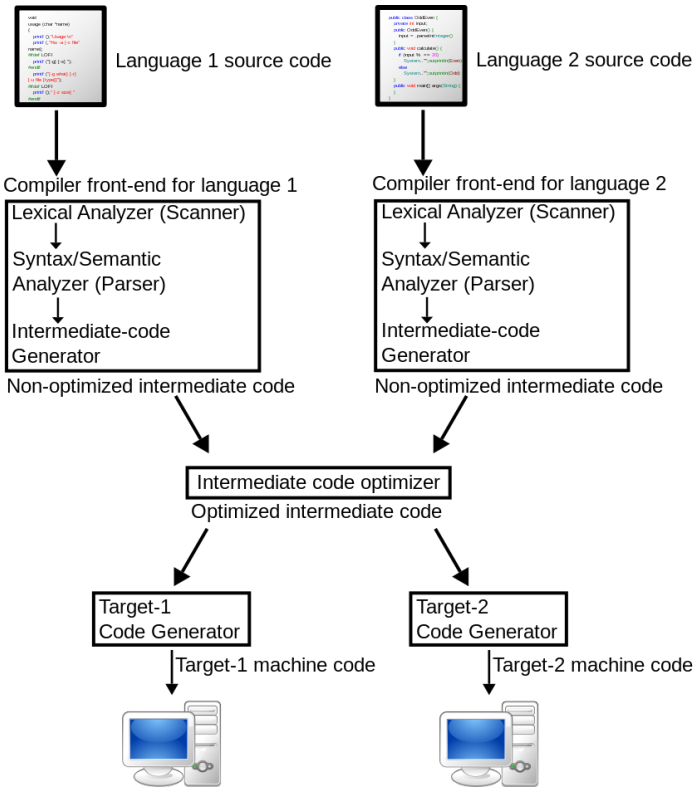

Figure: Generic compiler

- Frontend checks the high-level source code in terms of syntax and semantics.
- Middle-end is where the **optimizations** take place.
- Backend translates **intermediate representation (IR)** into the target assembly code.

Compiler structure

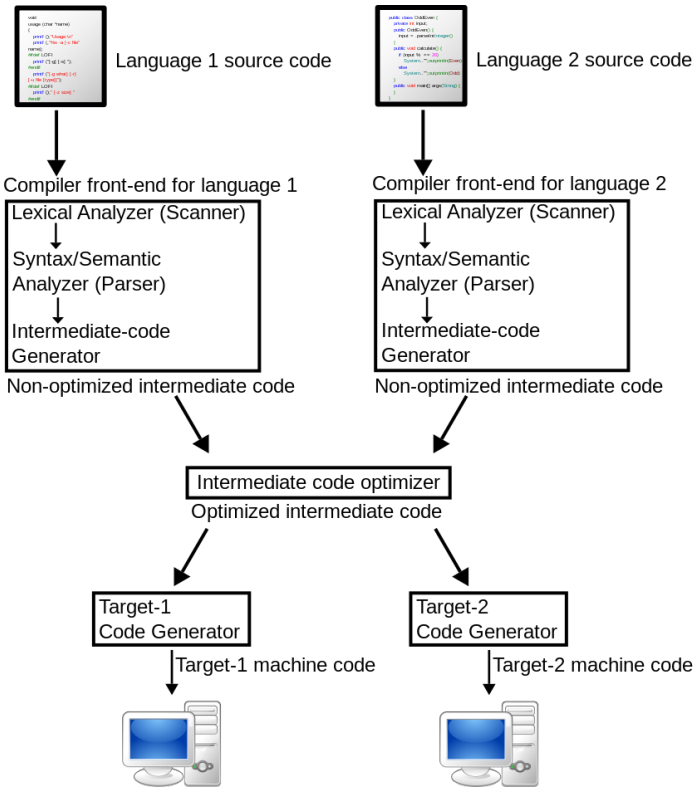

Figure: Generic compiler

- Frontend checks the high-level source code in terms of syntax and semantics.
- Middle-end is where the **optimizations** take place.
- Backend translates **intermediate representation (IR)** into the target assembly code.

Compiler structure

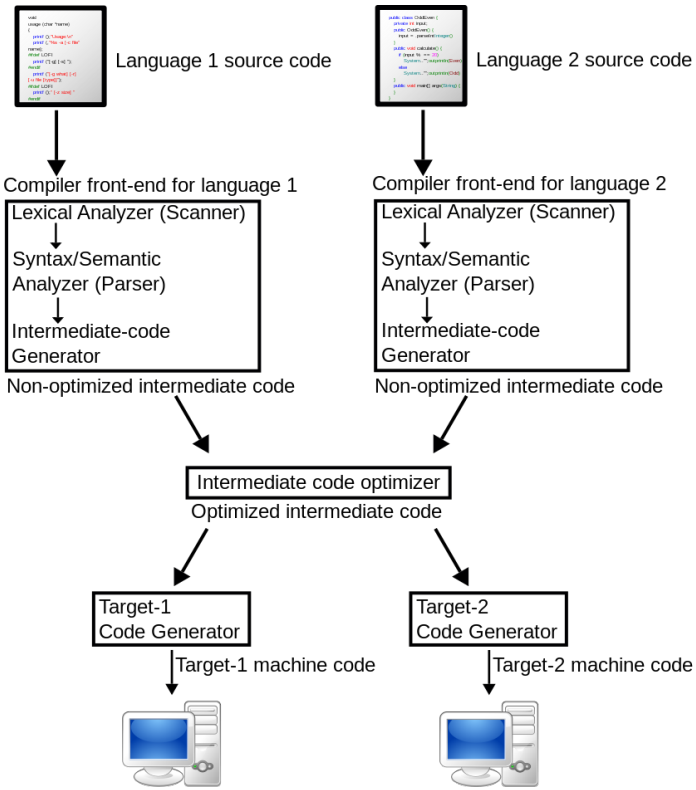

Figure: Generic compiler

- Frontend checks the high-level source code in terms of syntax and semantics.
- Middle-end is where the **optimizations** take place.
- Backend translates **intermediate representation (IR)** into the target assembly code.

Compiler structure

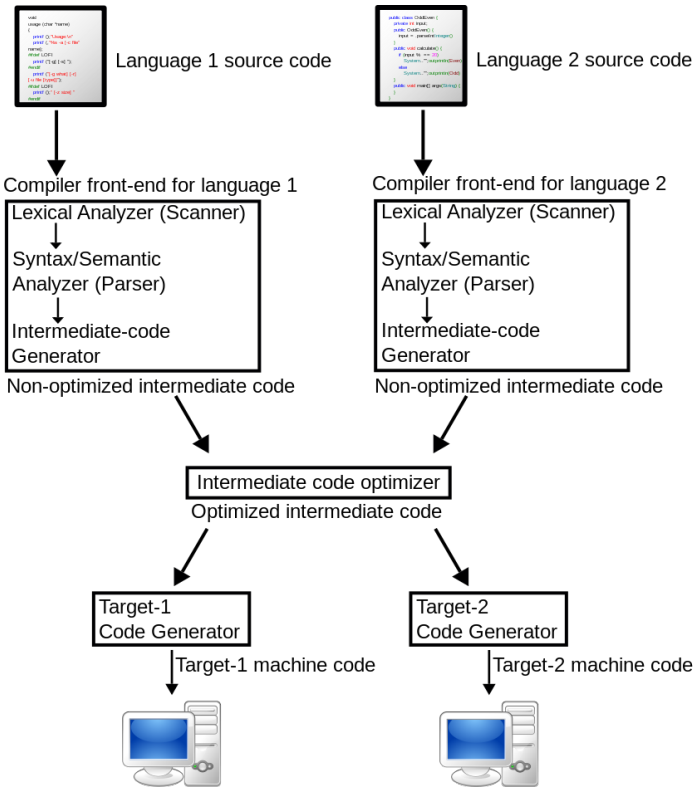

Figure: Generic compiler

- Frontend checks the high-level source code in terms of syntax and semantics.
- Middle-end is where the **optimizations** take place.
- Backend translates **intermediate representation (IR)** into the target assembly code.

GCC compilation in multiple phases

1. Parsing pass
   - Turns the code in tokens.
   - Result is a language dependant IR.
2. Gimplification pass
   - Get a language independant IR.
   - In GCC target IR is the GIMPLE language.
3. Pass manager
   - Run individual passes in correct order.
   - Bookkeeping.
4. Tree SSA pass
   - Single Static Assignment form.
   - Pass optimizes on trees.
5. RTL pass
   - Register Transfer Language.
   - Pass optimizes on statements (near to assembler).

GCC compilation in multiple phases

1. Parsing pass
   - Turns the code in tokens.
   - Result is a language dependant IR.
2. Gimplification pass
   - Get a language independant IR.
   - In GCC target IR is the GIMPLE language.
3. Pass manager
   - Run individual passes in correct order.
   - Bookkeeping.
4. Tree SSA pass
   - Single Static Assignment form.
   - Pass optimizes on trees.
5. RTL pass
   - Register Transfer Language.
   - Pass optimizes on statements (near to assembler).

GCC compilation in multiple phases

1. Parsing pass
   - Turns the code in tokens.
   - Result is a language dependant IR.
2. Gimplification pass
   - Get a language independant IR.
   - In GCC target IR is the GIMPLE language.
3. Pass manager
   - Run individual passes in correct order.
   - Bookkeeping.
4. Tree SSA pass
   - Single Static Assignment form.
   - Pass optimizes on trees.
5. RTL pass
   - Register Transfer Language.
   - Pass optimizes on statements (near to assembler).

GCC compilation in multiple phases

1. Parsing pass
   - Turns the code in tokens.
   - Result is a language dependant IR.
2. Gimplification pass
   - Get a language independant IR.
   - In GCC target IR is the GIMPLE language.
3. Pass manager
   - Run individual passes in correct order.
   - Bookkeeping.
4. Tree SSA pass
   - Single Static Assignment form.
   - Pass optimizes on trees.
5. RTL pass
   - Register Transfer Language.
   - Pass optimizes on statements (near to assembler).

GCC compilation in multiple phases

1. Parsing pass
   - Turns the code in tokens.
   - Result is a language dependant IR.
2. Gimplification pass
   - Get a language independant IR.
   - In GCC target IR is the GIMPLE language.
3. Pass manager
   - Run individual passes in correct order.
   - Bookkeeping.
4. Tree SSA pass
   - Single Static Assignment form.
   - Pass optimizes on trees.
5. RTL pass
   - Register Transfer Language.
   - Pass optimizes on statements (near to assembler).

GCC compilation in multiple phases

1. Parsing pass
   - Turns the code in tokens.
   - Result is a language dependant IR.
2. Gimplification pass
   - Get a language independant IR.
   - In GCC target IR is the GIMPLE language.
3. Pass manager
   - Run individual passes in correct order.
   - Bookkeeping.
4. Tree SSA pass
   - Single Static Assignment form.
   - Pass optimizes on trees.
5. RTL pass
   - Register Transfer Language.
   - Pass optimizes on statements (near to assembler).

GCC compilation in multiple phases

1. Parsing pass
   - Turns the code in tokens.
   - Result is a language dependant IR.
2. Gimplification pass
   - Get a language independant IR.
   - In GCC target IR is the GIMPLE language.
3. Pass manager
   - Run individual passes in correct order.
   - Bookkeeping.
4. Tree SSA pass
   - Single Static Assignment form.
   - Pass optimizes on trees.
5. RTL pass
   - Register Transfer Language.
   - Pass optimizes on statements (near to assembler).

GCC compilation in multiple phases

1. Parsing pass
   - Turns the code in tokens.
   - Result is a language dependant IR.
2. Gimplification pass
   - Get a language independant IR.
   - In GCC target IR is the GIMPLE language.
3. Pass manager
   - Run individual passes in correct order.
   - Bookkeeping.
4. Tree SSA pass
   - Single Static Assignment form.
   - Pass optimizes on trees.
5. RTL pass
   - Register Transfer Language.
   - Pass optimizes on statements (near to assembler).

GCC compilation in multiple phases

1. Parsing pass
   - Turns the code in tokens.
   - Result is a language dependant IR.
2. Gimplification pass
   - Get a language independant IR.
   - In GCC target IR is the GIMPLE language.
3. Pass manager
   - Run individual passes in correct order.
   - Bookkeeping.
4. Tree SSA pass
   - Single Static Assignment form.
   - Pass optimizes on trees.
5. RTL pass
   - Register Transfer Language.
   - Pass optimizes on statements (near to assembler).

GCC compilation in multiple phases

1. Parsing pass
   - Turns the code in tokens.
   - Result is a language dependant IR.
2. Gimplification pass
   - Get a language independant IR.
   - In GCC target IR is the GIMPLE language.
3. Pass manager
   - Run individual passes in correct order.
   - Bookkeeping.
4. Tree SSA pass
   - Single Static Assignment form.
   - Pass optimizes on trees.
5. RTL pass
   - Register Transfer Language.
   - Pass optimizes on statements (near to assembler).

GCC compilation in multiple phases

1. Parsing pass
   - Turns the code in tokens.
   - Result is a language dependant IR.
2. Gimplification pass
   - Get a language independant IR.
   - In GCC target IR is the GIMPLE language.
3. Pass manager
   - Run individual passes in correct order.
   - Bookkeeping.
4. Tree SSA pass
   - Single Static Assignment form.
   - Pass optimizes on trees.
5. RTL pass
   - Register Transfer Language.
   - Pass optimizes on statements (near to assembler).

GCC compilation in multiple phases

1. Parsing pass
   - Turns the code in tokens.
   - Result is a language dependant IR.
2. Gimplification pass
   - Get a language independant IR.
   - In GCC target IR is the GIMPLE language.
3. Pass manager
   - Run individual passes in correct order.
   - Bookkeeping.
4. Tree SSA pass
   - Single Static Assignment form.
   - Pass optimizes on trees.
5. RTL pass
   - Register Transfer Language.
   - Pass optimizes on statements (near to assembler).

GCC compilation in multiple phases

1. Parsing pass
   - Turns the code in tokens.
   - Result is a language dependant IR.
2. Gimplification pass
   - Get a language independant IR.
   - In GCC target IR is the GIMPLE language.
3. Pass manager
   - Run individual passes in correct order.
   - Bookkeeping.
4. Tree SSA pass
   - Single Static Assignment form.
   - Pass optimizes on trees.
5. RTL pass
   - Register Transfer Language.
   - Pass optimizes on statements (near to assembler).

GCC compilation in multiple phases

1. Parsing pass
   - Turns the code in tokens.
   - Result is a language dependant IR.
2. Gimplification pass
   - Get a language independant IR.
   - In GCC target IR is the GIMPLE language.
3. Pass manager
   - Run individual passes in correct order.
   - Bookkeeping.
4. Tree SSA pass
   - Single Static Assignment form.
   - Pass optimizes on trees.
5. RTL pass
   - Register Transfer Language.
   - Pass optimizes on statements (near to assembler).

GCC compilation in multiple phases

1. Parsing pass
   - Turns the code in tokens.
   - Result is a language dependant IR.
2. Gimplification pass
   - Get a language independant IR.
   - In GCC target IR is the GIMPLE language.
3. Pass manager
   - Run individual passes in correct order.
   - Bookkeeping.
4. Tree SSA pass
   - Single Static Assignment form.
   - Pass optimizes on trees.
5. RTL pass
   - Register Transfer Language.
   - Pass optimizes on statements (near to assembler).

GCC compilation in multiple phases

1. Parsing pass
   - Turns the code in tokens.
   - Result is a language dependant IR.
2. Gimplification pass
   - Get a language independant IR.
   - In GCC target IR is the GIMPLE language.
3. Pass manager
   - Run individual passes in correct order.
   - Bookkeeping.
4. Tree SSA pass
   - Single Static Assignment form.
   - Pass optimizes on trees.
5. RTL pass
   - Register Transfer Language.
   - Pass optimizes on statements (near to assembler).

GCC language lowering

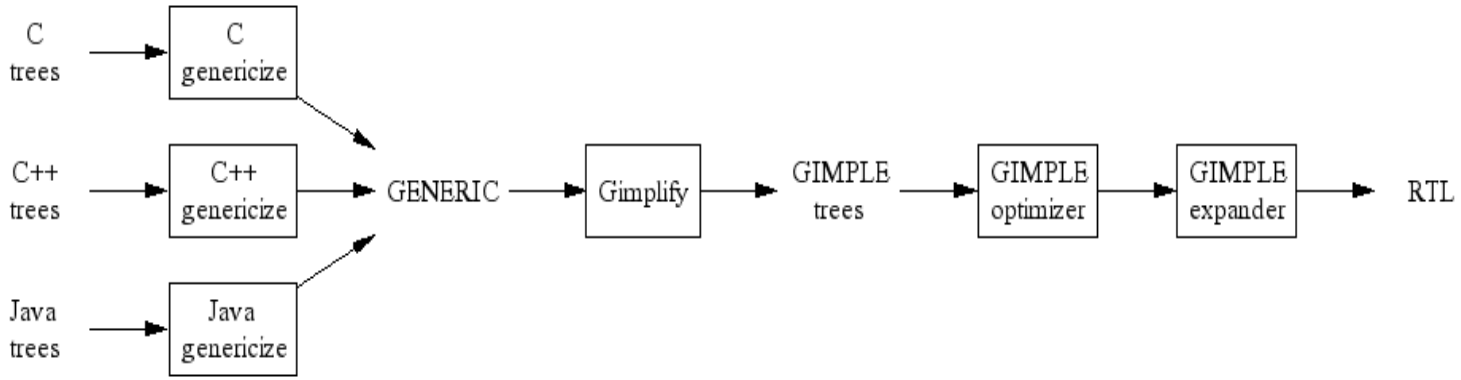

Figure: IR flow in GCC

- Each phase lowers its IR closer to the machine.
- GCC lowering: Parse Tree -> GENERIC -> GIMPLE -> SSA -> RTL

GCC language lowering

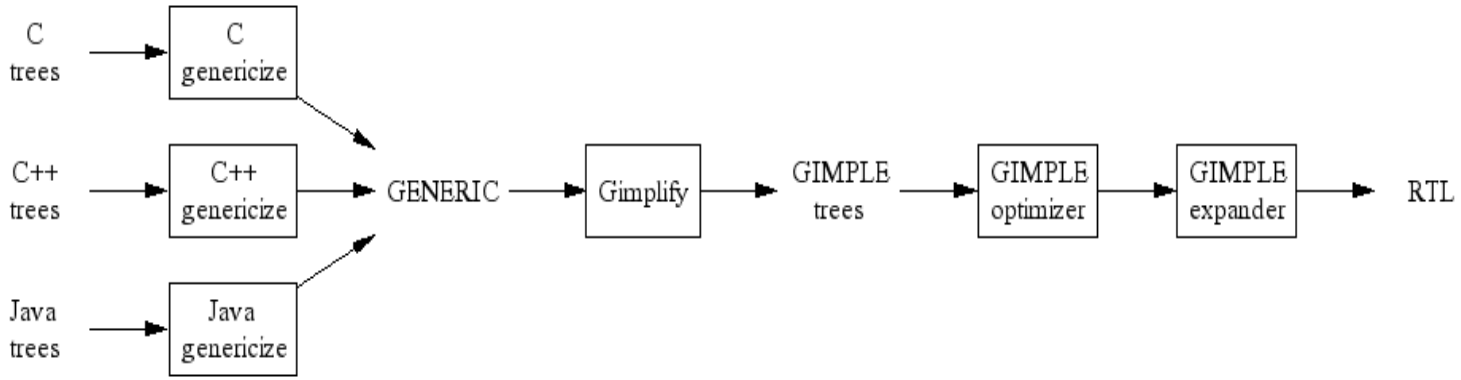

Figure: IR flow in GCC

- Each phase lowers its IR closer to the machine.
- GCC lowering: Parse Tree -> GENERIC -> GIMPLE -> SSA -> RTL

Outline

Optimizer Needs ...

Needs

- Need #1 is to have a language independant IR.
- Need #2 is to have an IR that can be executed abstractly.

Solutions

- Solution #1: Let compiler have its own language abstraction: GIMPLE.
- Solution #2: Let compiler work with trees.

Optimizer Needs ...

Needs

- Need #1 is to have a language independant IR.
- Need #2 is to have an IR that can be executed abstractly.

Solutions

- Solution #1: Let compiler have its own language abstraction: GIMPLE.
- Solution #2: Let compiler work with trees.

Optimizer Needs ...

Needs

- Need #1 is to have a language independant IR.
- Need #2 is to have an IR that can be executed abstractly.

Solutions

- Solution #1: Let compiler have its own language abstraction: GIMPLE.
- Solution #2: Let compiler work with trees.

Optimizer Needs ...

Needs

- Need #1 is to have a language independant IR.
- Need #2 is to have an IR that can be executed abstractly.

Solutions

- Solution #1: Let compiler have its own language abstraction: GIMPLE.
- Solution #2: Let compiler work with trees.

Optimizer Needs ...

Needs

- Need #1 is to have a language independant IR.
- Need #2 is to have an IR that can be executed abstractly.

Solutions

- Solution #1: Let compiler have its own language abstraction: GIMPLE.
- Solution #2: Let compiler work with trees.

Optimizer Needs ...

Needs

- Need #1 is to have a language independant IR.
- Need #2 is to have an IR that can be executed abstractly.

Solutions

- Solution #1: Let compiler have its own language abstraction: GIMPLE.
- Solution #2: Let compiler work with trees.

Need #1: GCC and Language Independency

Problem with language independent IRs

Language independant IRs are hairy to program for front-end programmers.

Solution

- GCC provides a 'close-to-the-program-front-end-target-language' (CTTPFETL) that still can have language dependencies: GENERIC.
- First introduced with GCCs Java front-end around 2000.

Need #1: GCC and Language Independency

Problem with language independent IRs

Language independant IRs are hairy to program for front-end programmers.

Solution

- GCC provides a 'close-to-the-program-front-end-target-language' (CTTPFETL) that still can have language dependencies: GENERIC.
- First introduced with GCCs Java front-end around 2000.

Need #1: GCC and Language Independency

Problem with language independent IRs

Language independant IRs are hairy to program for front-end programmers.

Solution

- GCC provides a 'close-to-the-program-front-end-target-language' (CTTPFETL) that still can have language dependencies: GENERIC.
- First introduced with GCCs Java front-end around 2000.

Need #1: GCC and Language Independency

Problem with language independent IRs

Language independant IRs are hairy to program for front-end programmers.

Solution

- GCC provides a 'close-to-the-program-front-end-target-language' (CTTPFETL) that still can have language dependencies: GENERIC.
- First introduced with GCCs Java front-end around 2000.

Need #1: GCC and GIMPLE

Problem with GENERIC

GCC optimizers do not work well with GENERIC.

Solution

GCC internally lowers GENERIC to its GIMPLE language.

Need #1: GCC and GIMPLE

Problem with GENERIC

GCC optimizers do not work well with GENERIC.

Solution

GCC internally lowers GENERIC to its GIMPLE language.

Need #1: GENERIC versus GIMPLE - An Example

```
1   a = foo ();
2   b = a + 10;
3   c = 5;
4   if (a > b + c)
5     c = b++ / a + (b * a);
6   bar (a, b, c);
```

```
1   a = foo ();
2   b = a + 10;
3   c = 5;
4   T1 = b + c;
5   if (a > T1)
6     {
7       T2 = b / a;
8       T3 = b * a;
9       c = T2 + T3;
10      b = b + 1;
11    }
12  bar (a, b, c);
```

Figure: Comparison between GENERIC (left) and GIMPLE (right)

GENERIC versus GIMPLE- Abstract

- GENERIC
  - The GENERIC syntax looks quite similar to C/C++!
  - This is why the C/C++ front-ends directly produce the GIMPLE form.
  - But: Irregular in structure plus side effects.

  → Hard to have optimizations on GENERIC.
- GIMPLE
  - Complex expressions broken down into 'I-can-not-break-this-down-anymore-expressions'
  - New variables for any of these atomic expressions.
  - GIMPLE representation is more regular in structure.

  → GIMPLE is ideal to do optimizations on it.

GENERIC versus GIMPLE- Abstract

- GENERIC
  - The GENERIC syntax looks quite similar to C/C++!
  - This is why the C/C++ front-ends directly produce the GIMPLE form.
  - But: Irregular in structure plus side effects.

  → Hard to have optimizations on GENERIC.
- GIMPLE
  - Complex expressions broken down into 'I-can-not-break-this-down-anymore-expressions'
  - New variables for any of these atomic expressions.
  - GIMPLE representation is more regular in structure.

  → GIMPLE is ideal to do optimizations on it.

GENERIC versus GIMPLE- Abstract

- GENERIC
  - The GENERIC syntax looks quite similar to C/C++!
  - This is why the C/C++ front-ends directly produce the GIMPLE form.
  - But: Irregular in structure plus side effects.

  → Hard to have optimizations on GENERIC.
- GIMPLE
  - Complex expressions broken down into 'I-can-not-break-this-down-anymore-expressions'
  - New variables for any of these atomic expressions.
  - GIMPLE representation is more regular in structure.

  → GIMPLE is ideal to do optimizations on it.

GENERIC versus GIMPLE- Abstract

- GENERIC
  - The GENERIC syntax looks quite similar to C/C++!
  - This is why the C/C++ front-ends directly produce the GIMPLE form.
  - But: Irregular in structure plus side effects.

  → Hard to have optimizations on GENERIC.
- GIMPLE
  - Complex expressions broken down into 'I-can-not-break-this-down-anymore-expressions'
  - New variables for any of these atomic expressions.
  - GIMPLE representation is more regular in structure.

  → GIMPLE is ideal to do optimizations on it.

GENERIC versus GIMPLE- Abstract

- GENERIC
  - The GENERIC syntax looks quite similar to C/C++!
  - This is why the C/C++ front-ends directly produce the GIMPLE form.
  - But: Irregular in structure plus side effects.

  → Hard to have optimizations on GENERIC.
- GIMPLE
  - Complex expressions broken down into 'I-can-not-break-this-down-anymore-expressions'
  - New variables for any of these atomic expressions.
  - GIMPLE representation is more regular in structure.

  → GIMPLE is ideal to do optimizations on it.

GENERIC versus GIMPLE- Abstract

- GENERIC
  - The GENERIC syntax looks quite similar to C/C++!
  - This is why the C/C++ front-ends directly produce the GIMPLE form.
  - But: Irregular in structure plus side effects.

  → Hard to have optimizations on GENERIC.
- GIMPLE
  - Complex expressions broken down into 'I-can-not-break-this-down-anymore-expressions'
  - New variables for any of these atomic expressions.
  - GIMPLE representation is more regular in structure.

  → GIMPLE is ideal to do optimizations on it.

GENERIC versus GIMPLE- Abstract

- GENERIC
  - The GENERIC syntax looks quite similar to C/C++!
  - This is why the C/C++ front-ends directly produce the GIMPLE form.
  - But: Irregular in structure plus side effects.

  → Hard to have optimizations on GENERIC.
- GIMPLE
  - Complex expressions broken down into 'I-can-not-break-this-down-anymore-expressions'
  - New variables for any of these atomic expressions.
  - GIMPLE representation is more regular in structure.

  → GIMPLE is ideal to do optimizations on it.

GENERIC versus GIMPLE- Abstract

- GENERIC
  - The GENERIC syntax looks quite similar to C/C++!
  - This is why the C/C++ front-ends directly produce the GIMPLE form.
  - But: Irregular in structure plus side effects.

  → Hard to have optimizations on GENERIC.
- GIMPLE
  - Complex expressions broken down into 'I-can-not-break-this-down-anymore-expressions'
  - New variables for any of these atomic expressions.
  - GIMPLE representation is more regular in structure.

  → GIMPLE is ideal to do optimizations on it.

GENERIC versus GIMPLE- Abstract

- GENERIC
  - The GENERIC syntax looks quite similar to C/C++!
  - This is why the C/C++ front-ends directly produce the GIMPLE form.
  - But: Irregular in structure plus side effects.

  → Hard to have optimizations on GENERIC.
- GIMPLE
  - Complex expressions broken down into 'I-can-not-break-this-down-anymore-expressions'
  - New variables for any of these atomic expressions.
  - GIMPLE representation is more regular in structure.

  → GIMPLE is ideal to do optimizations on it.

GENERIC versus GIMPLE- Abstract

- GENERIC
  - The GENERIC syntax looks quite similar to C/C++!
  - This is why the C/C++ front-ends directly produce the GIMPLE form.
  - But: Irregular in structure plus side effects.

  → Hard to have optimizations on GENERIC.
- GIMPLE
  - Complex expressions broken down into 'I-can-not-break-this-down-anymore-expressions'
  - New variables for any of these atomic expressions.
  - GIMPLE representation is more regular in structure.

  → GIMPLE is ideal to do optimizations on it.

Optimizer Needs ...

We still haven't talked about ...

Needs

- Need #1 is to have a language independant IR.
- Need #2 is to have an IR that can be executed abstractly.

Solutions

- Solution #1: Let compiler have its own language abstraction: GENERIC & GIMPLE.
- Solution #2: Let compiler work on trees.

... let's talk about!

Optimizer Needs ...

We still haven't talked about ...

Needs

- Need #1 is to have a language independant IR.
- Need #2 is to have an IR that can be executed abstractly.

Solutions

- Solution #1: Let compiler have its own language abstraction: GENERIC & GIMPLE.
- Solution #2: Let compiler work on trees.

... let's talk about!

Optimizer Needs ...

We still haven't talked about ...

Needs

- Need #1 is to have a language independant IR.
- Need #2 is to have an IR that can be executed abstractly.

Solutions

- Solution #1: Let compiler have its own language abstraction: GENERIC & GIMPLE.
- Solution #2: Let compiler work on trees.

... let's talk about!

Compiler and Trees

Before 2003 GCC directly lowered the GIMPLE IR into a statement-by-statement form and optimized then (RTL pass).

Problem

Informations about functions as total or relations between functions are lost.

Solution

In GCC since 2003: Tree SSA project to simplify the RTL pass and to implement high-level optimizers.

Compiler and Trees

Before 2003 GCC directly lowered the GIMPLE IR into a statement-by-statement form and optimized then (RTL pass).

Problem

Informations about functions as total or relations between functions are lost.

Solution

In GCC since 2003: Tree SSA project to simplify the RTL pass and to implement high-level optimizers.

Compiler and Trees

Before 2003 GCC directly lowered the GIMPLE IR into a statement-by-statement form and optimized then (RTL pass).

Problem

Informations about functions as total or relations between functions are lost.

Solution

In GCC since 2003: Tree SSA project to simplify the RTL pass and to implement high-level optimizers.

Tree SSA Example

- SSA: Static Single Assignment

```
1   a = foo ();
2   b = a + 10;
3   c = 5;
4   T1 = b + c;
5   if (a > T1)
6     {
7       T2 = b / a;
8       T3 = b * a;
9       c = T2 + T3;
10      b = b + 1;
11    }
12  bar (a, b, c);
```

```
1   a_1 = foo ();
2   b_1 = a_1 + 10;
3   c_1 = 5;
4   T1_1 = b_1 + c_1;
5   if (a_1 > T1_1)
6     {
7       T2_1 = b_1 / a_1;
8       T3_1 = b_1 * a_1;
9       c_2 = T2_1 + T3_1;
10      b_2 = b_1 + 1;
11    }
12  b_3 = φ(b_1, b_2);
13  c_3 = φ(c_1, c_2);
14  bar (a_1, b_3, c_3);
```

Figure: GIMPLE versus SSA

Outline

Overview

- Optimizations that can be done 'on the fly' while transforming GIMPLE to SSA:
  - Constant propagation
  - Copy propagtion
  - Redundancy elimination
  - Propagation of predicate expressions
- Furhter optimizations implemented as separate passes:
  - Sparse Conditional Constant Propagation (CCP)
  - Partial Redundancy Elimination (PRE)
  - Dead Code Elimination (DC)

Overview

- Optimizations that can be done 'on the fly' while transforming GIMPLE to SSA:
  - Constant propagation
  - Copy propagtion
  - Redundancy elimination
  - Propagation of predicate expressions
- Furhter optimizations implemented as separate passes:
  - Sparse Conditional Constant Propagation (CCP)
  - Partial Redundancy Elimination (PRE)
  - Dead Code Elimination (DC)

Overview

- Optimizations that can be done 'on the fly' while transforming GIMPLE to SSA:
  - Constant propagation
  - Copy propagtion
  - Redundancy elimination
  - Propagation of predicate expressions
- Furhter optimizations implemented as separate passes:
  - Sparse Conditional Constant Propagation (CCP)
  - Partial Redundancy Elimination (PRE)
  - Dead Code Elimination (DC)

Overview

- Optimizations that can be done 'on the fly' while transforming GIMPLE to SSA:
  - Constant propagation
  - Copy propagtion
  - Redundancy elimination
  - Propagation of predicate expressions
- Furhter optimizations implemented as separate passes:
  - Sparse Conditional Constant Propagation (CCP)
  - Partial Redundancy Elimination (PRE)
  - Dead Code Elimination (DC)

Overview

- Optimizations that can be done 'on the fly' while transforming GIMPLE to SSA:
  - Constant propagation
  - Copy propagtion
  - Redundancy elimination
  - Propagation of predicate expressions
- Furhter optimizations implemented as separate passes:
  - Sparse Conditional Constant Propagation (CCP)
  - Partial Redundancy Elimination (PRE)
  - Dead Code Elimination (DC)

Overview

- Optimizations that can be done 'on the fly' while transforming GIMPLE to SSA:
  - Constant propagation
  - Copy propagtion
  - Redundancy elimination
  - Propagation of predicate expressions
- Furhter optimizations implemented as separate passes:
  - Sparse Conditional Constant Propagation (CCP)
  - Partial Redundancy Elimination (PRE)
  - Dead Code Elimination (DC)

Overview

- Optimizations that can be done 'on the fly' while transforming GIMPLE to SSA:
  - Constant propagation
  - Copy propagtion
  - Redundancy elimination
  - Propagation of predicate expressions
- Furhter optimizations implemented as separate passes:
  - Sparse Conditional Constant Propagation (CCP)
  - Partial Redundancy Elimination (PRE)
  - Dead Code Elimination (DC)

Overview

- Optimizations that can be done 'on the fly' while transforming GIMPLE to SSA:
  - Constant propagation
  - Copy propagtion
  - Redundancy elimination
  - Propagation of predicate expressions
- Furhter optimizations implemented as separate passes:
  - Sparse Conditional Constant Propagation (CCP)
  - Partial Redundancy Elimination (PRE)
  - Dead Code Elimination (DC)

Overview

- Optimizations that can be done 'on the fly' while transforming GIMPLE to SSA:
  - Constant propagation
  - Copy propagtion
  - Redundancy elimination
  - Propagation of predicate expressions
- Furhter optimizations implemented as separate passes:
  - Sparse Conditional Constant Propagation (CCP)
  - Partial Redundancy Elimination (PRE)
  - Dead Code Elimination (DC)

Constant propagation

- When a constant assignment $a_i = C$ is found, it is stored in a hash table.
- Successive occurrences of a_i are replaced with C.
- Copy propagation similar.

Constant propagation

- When a constant assignment $a_i = C$ is found, it is stored in a hash table.
- Successive occurrences of a_i are replaced with C.
- Copy propagation similar.

Constant propagation

- When a constant assignment $a_i = C$ is found, it is stored in a hash table.
- Successive occurrences of a_i are replaced with C.
- Copy propagation similar.

Redundancy elimination

- When an assignment of the form $a_i = b_j \oplus c_k$ is found, the expression $b_j \oplus c_k$ is stored in a hash table.
- Successive occurrences of $b_j \oplus c_k$ *within the same sub-tree*, are replaced with a_i.

Redundancy elimination

- When an assignment of the form $a_i = b_j \oplus c_k$ is found, the expression $b_j \oplus c_k$ is stored in a hash table.
- Successive occurrences of $b_j \oplus c_k$ *within the same sub-tree*, are replaced with a_i.

Propagation of predicate expressions

- When a conditional statement of the form $if(a_i == C)$ is found, the assignment $a_i = C$ is inserted into a hash table.
- When processing the "then" clause of the conditional, successive occurences of a_i are replaced with C in that sub-tree.

Propagation of predicate expressions

- When a conditional statement of the form $if(a_i == C)$ is found, the assignment $a_i = C$ is inserted into a hash table.
- When processing the "then" clause of the conditional, successive occurences of a_i are replaced with C in that sub-tree.

Sparse conditional constant propagation (SCP)

- SCP does constant/copy propagation and dead code eliminations simultaneously/efficiently.
- Algorithm by Cooper and Torczon 2005.

Sparse conditional constant propagation (SCP)

- SCP does constant/copy propagation and dead code eliminations simultaneously/efficiently.
- Algorithm by Cooper and Torczon 2005.

Partial redundancy elimination (PRE)

```
1 if (condition) {
    // ...
3   x = y + 42;
  }
5 else {
    // ...
7 }
  z = y + 42;
9 // ...
```

Figure: Partial Redundancy

```
1 if (condition) {
    // ...
3   x = y + 42;
    z = x;
5 }
  else {
7   // ...
    z = y + 42;
9 }
```

Figure: PRE optimization

Dead code elimination (DCE)

Remove all statements in the program that have no effect on its output.

Outline

List of all Strategies in GCC 4.4.3

-falign-functions[=n] -falign-jumps[=n] -falign-labels[=n] -falign-loops[=n] -fassociative-math -fauto-inc-dec
-fbranch-probabilities -fbranch-target-load-optimize -fbranch-target-load-optimize2 -fbtr-bb-exclusive -fcaller-saves
-fcheck-data-deps -fconserve-stack -fcprop-registers -fcrossjumping -fcse-follow-jumps -fcse-skip-blocks
-fcx-fortran-rules -fcx-limited-range -fdata-sections -fdce -fdce -fdelayed-branch -fdelete-null-pointer-checks -fdse
-fdse -fearly-inlining -fexpensive-optimizations -ffast-math -ffinite-math-only -ffloat-store -fforward-propagate
-ffunction-sections -fgcse -fgcse-after-reload -fgcse-las -fgcse-lm -fgcse-sm -fif-conversion -fif-conversion2
-findirect-inlining -finline-functions -finline-functions-called-once -finline-limit=n -finline-small-functions -fipa-cp
-fipa-cp-clone -fipa-matrix-reorg -fipa-pta -fipa-pure-const -fipa-reference -fipa-struct-reorg -fipa-type-escape
-fira-algorithm=algorithm -fira-region=region -fira-coalesce -fno-ira-share-save-slots -fno-ira-share-spill-slots
-fira-verbose=n -fivopts -fkeep-inline-functions -fkeep-static-consts -floop-block -floop-interchange -floop-strip-mine
-fmerge-all-constants -fmerge-constants -fmodulo-sched -fmodulo-sched-allow-regmoves -fmove-loop-invariants
-fmudflap -fmudflapir -fmudflapth -fno-branch-count-reg -fno-default-inline -fno-defer-pop -fno-function-cse
-fno-guess-branch-probability -fno-inline -fno-math-errno -fno-peephole -fno-peephole2 -fno-sched-interblock
-fno-sched-spec -fno-signed-zeros -fno-toplevel-reorder -fno-trapping-math -fno-zero-initialized-in-bss
-fomit-frame-pointer -foptimize-register-move -foptimize-sibling-calls -fpeel-loops -fpredictive-commoning

List of all Strategies in GCC 4.4.3

In the following we will discuss each of them ...

List of all Strategies in GCC 4.4.3

Figure: Just Kidding

GCC's optimization levels

The strategies above can be categorized as follows:

- Strategies that do not increase the programs size and run fast (-O1 option).
- Strategies that do not increase the programs size and need a plenty of time (-O2 option).
- Strategies that do increase the programs size and need a plenty of time (-O3 options).
- Strategies that do decrease the programs size (-Os option).
- Default: No optimization at all (-O0 option) for debugging.
- -O0 $\subset$ -O1 $\subset$ -O2 $\subset$ -O3.

GCC's optimization levels

The strategies above can be categorized as follows:

- Strategies that do not increase the programs size and run fast (-O1 option).
- Strategies that do not increase the programs size and need a plenty of time (-O2 option).
- Strategies that do increase the programs size and need a plenty of time (-O3 options).
- Strategies that do decrease the programs size (-Os option).
- Default: No optimization at all (-O0 option) for debugging.
- -O0 $\subset$ -O1 $\subset$ -O2 $\subset$ -O3.

GCC's optimization levels

The strategies above can be categorized as follows:

- Strategies that do not increase the programs size and run fast (-O1 option).
- Strategies that do not increase the programs size and need a plenty of time (-O2 option).
- Strategies that do increase the programs size and need a plenty of time (-O3 options).
- Strategies that do decrease the programs size (-Os option).
- Default: No optimization at all (-O0 option) for debugging.
- -O0 $\subset$ -O1 $\subset$ -O2 $\subset$ -O3.

GCC's optimization levels

The strategies above can be categorized as follows:

- Strategies that do not increase the programs size and run fast (-O1 option).
- Strategies that do not increase the programs size and need a plenty of time (-O2 option).
- Strategies that do increase the programs size and need a plenty of time (-O3 options).
- Strategies that do decrease the programs size (-Os option).
- Default: No optimization at all (-O0 option) for debugging.
- -O0 $\subset$ -O1 $\subset$ -O2 $\subset$ -O3.

GCC's optimization levels

The strategies above can be categorized as follows:

- Strategies that do not increase the programs size and run fast (-O1 option).
- Strategies that do not increase the programs size and need a plenty of time (-O2 option).
- Strategies that do increase the programs size and need a plenty of time (-O3 options).
- Strategies that do decrease the programs size (-Os option).
- Default: No optimization at all (-O0 option) for debugging.
- -O0 $\subset$ -O1 $\subset$ -O2 $\subset$ -O3.

GCC's optimization levels

The strategies above can be categorized as follows:

- Strategies that do not increase the programs size and run fast (-O1 option).
- Strategies that do not increase the programs size and need a plenty of time (-O2 option).
- Strategies that do increase the programs size and need a plenty of time (-O3 options).
- Strategies that do decrease the programs size (-Os option).
- Default: No optimization at all (-O0 option) for debugging.
- -O0 $\subset$ -O1 $\subset$ -O2 $\subset$ -O3.

Summary

- Thanks to the GIMPLE language talking about compiler optimization becomes language independant (in GCC).
- Since about 2003 GCC works with the Tree SSA pass instead of directly compiling to RTL.
- With its -Ox flags, GCC provides 4 optimization levels with increasing optimization complexity.

- Outlook
  - Loop optimizations are a big field of subject. We haven't talked about this yet.
  - What's going on in RTL, that is what low-level optimizations can be done?

Summary

- Thanks to the GIMPLE language talking about compiler optimization becomes language independant (in GCC).
- Since about 2003 GCC works with the Tree SSA pass instead of directly compiling to RTL.
- With its -Ox flags, GCC provides 4 optimization levels with increasing optimization complexity.

- Outlook
  - Loop optimizations are a big field of subject. We haven't talked about this yet.
  - What's going on in RTL, that is what low-level optimizations can be done?

Summary

- Thanks to the GIMPLE language talking about compiler optimization becomes language independant (in GCC).
- Since about 2003 GCC works with the Tree SSA pass instead of directly compiling to RTL.
- With its -Ox flags, GCC provides 4 optimization levels with increasing optimization complexity.

- Outlook
  - Loop optimizations are a big field of subject. We haven't talked about this yet.
  - What's going on in RTL, that is what low-level optimizations can be done?

Summary

- Thanks to the GIMPLE language talking about compiler optimization becomes language independant (in GCC).
- Since about 2003 GCC works with the Tree SSA pass instead of directly compiling to RTL.
- With its -Ox flags, GCC provides 4 optimization levels with increasing optimization complexity.

- Outlook
  - Loop optimizations are a big field of subject. We haven't talked about this yet.
  - What's going on in RTL, that is what low-level optimizations can be done?

Summary

- Thanks to the GIMPLE language talking about compiler optimization becomes language independant (in GCC).
- Since about 2003 GCC works with the Tree SSA pass instead of directly compiling to RTL.
- With its -Ox flags, GCC provides 4 optimization levels with increasing optimization complexity.

- Outlook
  - Loop optimizations are a big field of subject. We haven't talked about this yet.
  - What's going on in RTL, that is what low-level optimizations can be done?

Summary

- Thanks to the GIMPLE language talking about compiler optimization becomes language independant (in GCC).
- Since about 2003 GCC works with the Tree SSA pass instead of directly compiling to RTL.
- With its -Ox flags, GCC provides 4 optimization levels with increasing optimization complexity.

- Outlook
  - Loop optimizations are a big field of subject. We haven't talked about this yet.
  - What's going on in RTL, that is what low-level optimizations can be done?

For Further Reading I

- Proceedings of the GCC Developers Summit 2003, 171-193.
  Introduction to GENERIC, GIMPLE, Tree SSA.
- C. Keith D., L. Torczon.
  Engineering a Compiler.
  Morgan Kaufmann, 2005
- http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
  Description of each optimization option in GCC
- http://gcc.gnu.org/onlinedocs/gccint/
  GCC Internals Documentation
- http://en.wikipedia.org/wiki/Static_single_assignment_form
  Wiki on SSA

For Further Reading II

- http://www.public.asu.edu/ kbai3/docs/Gimple.pdf
  Ke Bai - GIMPLE In GCC
  Presentation at the Department of Computer Science, Arizona State University, 2010
- www.phoronix.com/scan.php?page=article&item=gcc_45_benchm
  Benchmarks on the new GCC 4.5.0
- http://gcc.gnu.org/wiki/PythonFrontEnd
  Python front end for GCC

Figures I

- Generic compiler on page 5: Wikipedia.
- GCC passes and IRs on page 27: GCC internals documentation.
- Comparison between GENERIC and GIMPLE on page 42: GCC summit 2003.
- On page **??**: GCC summit 2003.