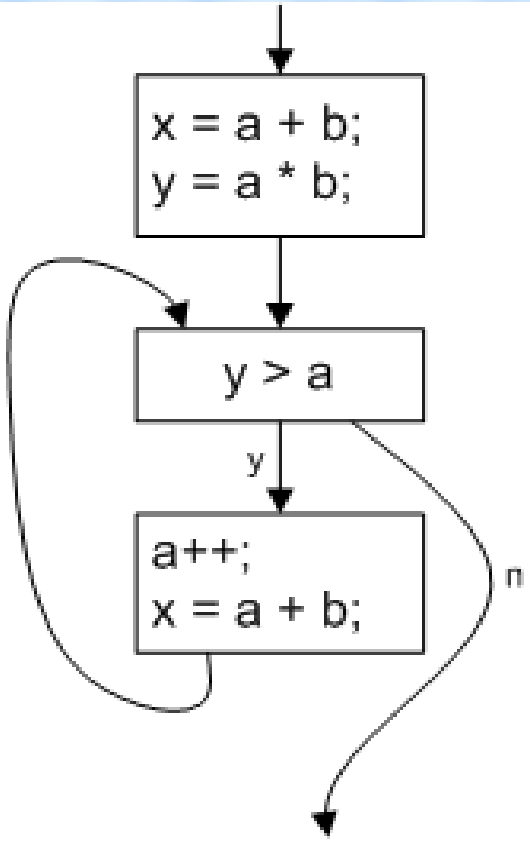
Анатомия GCC

Синявин А. В.

Ликбез №1 / CFG

Control flow graph

```
x = a + b;
y = a * b;
while( y > a )
{
    a++;
    x = a + b;
}
```

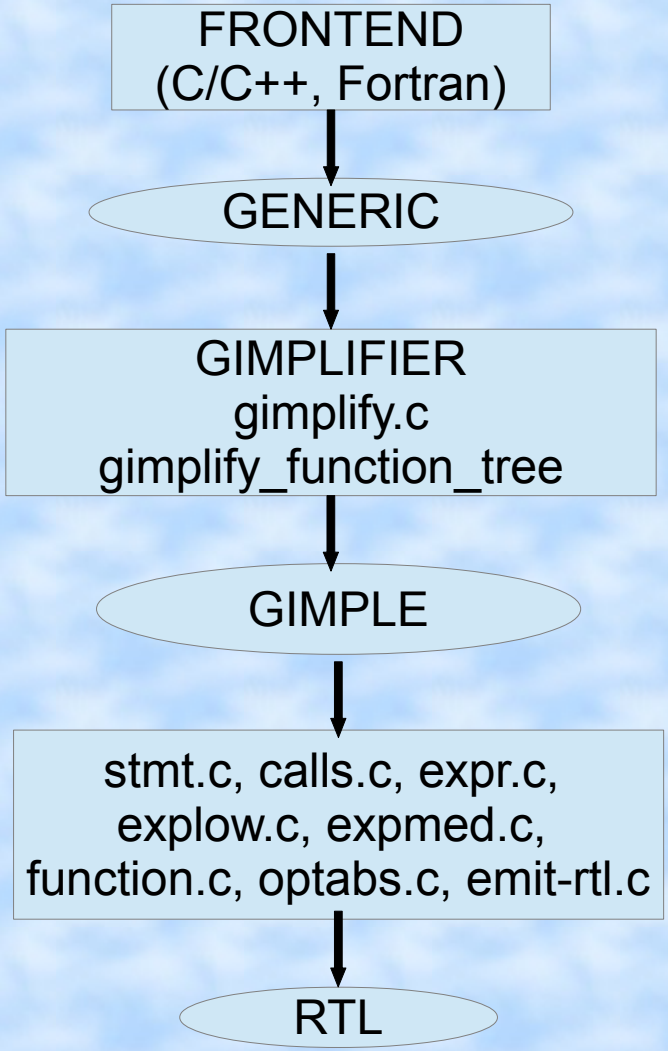


Базовый блок — максимальный участок, который обладает свойствами:

- поток управления может входить только через первую команду

- управление покидает блок без останова или ветвления, за исключением, быть может последней команды

IR-ы в GCC


FRONTEND
(C/C++, Fortran)
GENERIC
GIMPLIFIER
gimplify.c
gimplify_function_tree
GIMPLE
stmt.c, calls.c, expr.c,
explow.c, expmed.c,
function.c, optabs.c, emit-rtl.c
RTL

GIMPLE

- GIMPLE = GENERIC + SIMPLE
  - испытал влияние SIMPLE IL компилятора McCAT
  - унаследован от GENERIC путём трансформирования в трёх-операндные кортежи
  - вводятся промежуточные переменные для хранения промежуточных значений
    - (a+b)*c => t1 = a+b; t2 = t1 * c
  - все структуры управления => conditional jumps
  - удаляются лексические области видимости
  - Etc
- gimplify.c / gimplify_function_tree

Основы GENERIC

- основа GENERIC - указатель на структуру tree

```
typedef union tree_node *tree;    // coretypes.h
```

- TREE_CODE(t)

```
enum tree_code {                  // type.h
    #include "all-tree.def"

    MAX_TREE_CODES
};
```

```
/*
  all-tree.def  создаётся  при сборке GCC.

  Коды GIMPLE нужно  смотреть в tree.def
*/
```

GENERIC / IDENTIFIER_NODE

- Некоторые свойства
  - IDENTIFIER_POINTER(t)
  - IDENTIFIER_LENGTH(t)
  - Etc.

GENERIC / declarations

- Типы деклараций
  - LABEL_DECL
  - CONST_DECL (enum - константы)
  - RESULT_DECL (value returned by function)
  - TYPE_DECL (typedef декларации)
  - VAR_DECL, PARM_DECL, FIELD_DECL (переменная, параметр, поле структуры)
  - etc
- Некоторые свойства деклараций
  - DECL_NAME () - имя декларации в виде IDENTIFIER_NODE()
  - TREE_TYPE () - тип
  - etc

GENERIC / containers

- TREE_LIST — однонаправленный список. Каждый элемент списка содержит по два элемента tree (TREE_PURPOSE / TREE_VALUE).

- TREE_VEC — простой вектор из tree

GENERIC / types

- Типы данных
  - VOID_TYPE, INTEGER_TYPE, REAL_TYPE, FIXED_POINT_TYPE, COMPLEX_TYPE, ENUMERAL_TYPE, BOOLEAN_TYPE, POINTER_TYPE, etc.
- Пример #1:

```
// var — tree node, некоторая переменная типа int
tree type = TREE_TYPE(var);

if ( TREE_CODE (type) == INTEGER_TYPE )

{

    // do something

}
```

GENERIC / types

- Пример #2:

```
// var — tree node, некоторая переменная типа int *
tree type = TREE_TYPE(var);

if ( TREE_CODE(type) == POINTER_TYPE )

{

    action1();

}


tree type2 = TREE_TYPE(type);

if ( TREE_CODE (type2) == INTEGER_TYPE )

{

    action2();

}
```

GENERIC / types

- Некоторые свойства
  - TYPE_SIZE — в виде tree (закладка на то, что размер типа может не вычислится в compile-time)
  - TYPE_ALIGN (int)
  - TYPE_NAME — декларация TYPE_DECL (не IDENTIFIER_NODE)
  - TYPE_STRUCTURAL_EQUALITY_P
  - etc
- Некоторые C++ специфичные свойства
  - CP_TYPE_CONST_P, CP_TYPE_VOLATILE_P, CP_TYPE_RESTRICT_P
  - Поддержка ООП-фич
  - etc
- Java/FORTRAN/etc специфичные свойства

Ликбез №2 / cv-квалификаторы

буфера s1 и s2 не пересекаются

restrict поддерживается C99, но не C++ стандартом

```
void * memcpy(
        void * restrict s1,
        const void * restrict s2,
        size_t s)
{
      // ...
}
```

cv-квалификаторы:

- const
- volatile
- restrict

Опасность вечного цикла.
Предотвратить размещение переменной на регистре

Поддерживается стандартом C/C++

```
volatile bool cancel = false;
while ( !cancel )
{
    ;
}
```

GENERIC / attributes

- Атрибуты, которые описываются через __attribute__ в C

  void f () __attribute__ ((always_inline));

- Представлен как TREE_LIST
  - TREE_PURPOSE (как IDENTIFIER_NODE — имя атрибута)
  - TREE_VALUE (TREE_LIST — список аргументов)
- DECL_ATTRIBUTES(decl) - список атрибутов для декларации

GENERIC / выражения

- TREE_CODE(e)
  - константные выражения: INTEGER_CST, REAL_CST, FIXED_CST, COMPLEX_CST, VECTOR_CST, STRING_CST
  - Ссылки в память: ARRAY_REF, ARRAY_RANGE_REF, TARGET_MEM_REF, ADDR_EXPR, INDIRECT_REF, MEM_REF, COMPONENT_REF
  - Унарные & бинарные выражения: NEGATE_EXPR, ABS_EXPR, BIT_NOT_EXPR, TRUTH_NOT_EXPR, PREDECREMENT_EXPR, PREINCREMENT_EXPR, POSTDECREMENT_EXPR, POSTINCREMENT_EXPR, FIX_TRUNC_EXPR, FLOAT_EXPR
  - Векторные выражения

GENERIC / константные выражения

Например, для INTEGER_CST

- TYPE_TREE
- можем получить значение константы

```
if ( TREE_CODE(t) == INTEGER_CST )
{
        tree itype = TYPE_TREE(t);
      #define GET_CST(t) (unsigned int)((TREE_INT_CST_HIGH(t) << \
                              HOST_BITS_PER_WIDE_INT) + TREE_INT_CST_LOW(t))
        int value = GET_CST(t);
}
```

GENERIC / унарные & бинарные выражения

Выражение	Описание
NEGATE_EXPR	-x (x — целое или вещественное)
ABS_EXPR	ABS(x) (x — целое или вещественное)
BIT_NOT_EXPR	~x (x — целое)
TRUTH_NOT_EXPR	!x (x — целое или булевское)
PREDECREMENT_EXPR, PREINCREMENT_EXPR, POSTDECREMENT_EXPR, POSTINCREMENT_EXPR	--x ++x x-- x++ (x — целое или вещественное)
FIX_TRUNC_EXPR	преобразование вещественного x к целому. Округление в сторону 0
FLOAT_EXPR	преобразование целого выражения x к вещественному выражению
COMPLEX_EXPR	создаёт комплексное число из двух операндов x1 и x2 (целое или вещественное)
CONJ_EXPR	комплексно сопряжённое число для x

x,x1, x2 — в GENERIC выражения, в GIMPLE - переменные

GENERIC / унарные & бинарные выражения

Выражение	Описание
REALPART_EXPR IMAGPART_EXPR	действительная / мнимая часть комплексного x
LSHIFT_EXPR RSHIFT_EXPR	x1 << x2 x1 > > x2
BIT_IOR_EXPR BIT_XOR_EXPR BIT_AND_EXPR	x1 | x2 x1 ^ x2 x1 & x2
TRUTH_ANDIF_EXPR TRUTH_ORIF_EXPR	x1 && x2 («short circuit scheme») x1 || x2 («short circuit scheme»)
TRUTH_AND_EXPR TRUTH_OR_EXPR TRUTH_XOR_EXPR	x1 && x2 (no «short circuit scheme») x1 || x2 (no «short circuit scheme») x1 logic_xor x2
POINTER_PLUS_EXPR	p + x (p должен быть pointer/reference тип, x — unsigned int)
PLUS_EXPR MINUS_EXPR MULT_EXPR	x1 + x2 x1 — x2 x1 * x2 x1/x2 — целые или вещественные, тип должен быть одинаковый flag_wrapv / flag_trapv контролируют арифметическое переполнение

p,x1, x2 — в GENERIC выражения, в GIMPLE - переменные

GENERIC / унарные & бинарные выражения

Выражение	Описание
MULT_HIGHPART_EXPR	(?)
RDIV_EXPR	x1 / x2 - вещественное деление
TRUNC_DIV_EXPR FLOOR_DIV_EXPR CEIL_DIV_EXPR ROUND_DIV_EXPR	целочисленное деление округление в сторону 0 округление в сторону -inf округление в сторону +inf округление в сторону ближайшего целого
TRUNC_MOD_EXPR FLOOR_MOD_EXPR CEIL_MOD_EXPR ROUND_MOD_EXPR	остаток от деления Смысл TRUNC / FLOOR / CEIL / ROUND такой же как и выше
EXACT_DIV_EXPR	целочисленное деление (?)

x1, x2 — в GENERIC выражения, в GIMPLE - переменные

GENERIC / унарные & бинарные выражения

Выражение	Описание
LT_EXPR LE_EXPR GT_EXPR GE_EXPR EQ_EXPR NE_EXPR	x1 < x2 x1 <= x2 x1 > x2 x1 >= x2 x1 == x2 x1 != x2 скалярные целые, вещественные

ORDERED_EXPR
UNORDERED_EXPR
	определяет являются x1 и x2 упорядоченными.
	if (is_nan(x1) || is_nan(x2))
	  return false;
	else
	  return true;

UNLT_EXPR UNLE_EXPR UNGT_EXPR UNGE_EXPR UNEQ_EXPR LTGT_EXPR	

GENERIC / basic statements

- Присутствуют в GENERIC, но скорее всего отсутствуют в GIMPLE. (Можно выяснить в gimplify.c / gimplify_expr)
  - ASM_EXPR, DECL_EXPR, LABEL_EXPR, GOTO_EXPR, RETURN_EXPR, LOOP_EXPR, EXIT_EXPR, SWITCH_STMT, CASE_LABEL_EXPR
  - BIND_EXPR
  - GENERIC OMP инструкции (точно отсутствуют в GIMPLE, конвертируются в GIMPLE OMP инструкции в gimple.c): OMP_PARALLEL, OMP_FOR, OMP_SECTIONS, OMP_SECTION, OMP_SINGLE, OMP_MASTER, OMP_ORDERED, OMP_CRITICAL, OMP_RETURN, OMP_CONTINUE, OMP_ATOMIC, OMP_CLAUSE

GIMPLE / high & low

Instruction	High GIMPLE	Low GIMPLE
GIMPLE_ASM	x	x
GIMPLE_ASSIGN	x	x
GIMPLE_BIND	x	
GIMPLE_CALL	x	x
GIMPLE_CATCH	x	
GIMPLE_COND	x	x
GIMPLE_DEBUG	x	x
GIMPLE_EH_FILTER	x	
GIMPLE_GOTO	x	x
GIMPLE_LABEL	x	x
GIMPLE_NOP	x	x
GIMPLE_OMP_ATOMIC_LOAD	x	x
GIMPLE_OMP_ATOMIC_STORE	x	x
GIMPLE_OMP_CONTINUE	x	x
GIMPLE_OMP_CRITICAL	x	x
GIMPLE_OMP_FOR	x	x
GIMPLE_OMP_MASTER	x	x
GIMPLE_OMP_ORDERED	x	x
GIMPLE_OMP_PARALLEL	x	x
GIMPLE_OMP_RETURN	x	x
GIMPLE_OMP_SECTION	x	x
GIMPLE_OMP_SECTIONS	x	x
GIMPLE_OMP_SECTIONS_SWITCH	x	x
GIMPLE_OMP_SINGLE	x	x
GIMPLE_PHI		x
GIMPLE_RESX		x
GIMPLE_RETURN	x	x
GIMPLE_SWITCH	x	x
GIMPLE_TRY	x	

gimple_code(g)

GIMPLE / passes

High / low	фаза	описание
High GIMPLE	Удаление бесполезных инструкций (Remove useless statements)	Примитивный алгоритм удаление мёртвого кода. Упрощение if-инструкций с константными условиями. Удаление пустых GIMPLE_BIND. И т.д.
	добавление проверок обращение к памяти (обращение по NULL, выход за пределы буфера) mudflap_1	Добавление кода, для контроля обращений к памяти
	OpenMP lowering	
	OpenMP expansion	Удаление GIMPLE OpenMP инструкций и замена на библиотечных вызовов
Что-то между high & low	Lower control flow	Удаление COND_EXPR и замена на GIMPLE_GOTO, удаление BIND_EXPR

GIMPLE / passes

Low / high	фаза	описание
Что-то между high & low	lower exception handling control flow	Удаление GIMPLE_TRY, GIMPLE_CATCH, GIMPLE_EH_FILTER и замена их эквивалентами для представление в CF.
	Построение CF Build the control flow graph	Выделение базовых блоков и создание передач управления между ними
	Find all referenced variables	Поиск переменных для SSA формы
	Построение SSA формы Enter SSA form	
Low GIMPLE	Сгенерировать предупреждения для неинициализированных переменных Warn for uninitialized variables	

GIMPLE / passes

Low / high	фаза	описание
Low GIMPLE	- dead code elimination - dominator optimizations - forward propagation - copy renaming - PHI node optimizations - May-alias optimization - profiling	Различные оптимизации
Low GIMPLE	lower complex arithmetic	Замена операций над комплексными числами на покомпонентные операции
Low GIMPLE	- scalar replacement - dead store elimination - tail recursion elimination - forward store motion … - mudflap statement annotation	Различные оптимизации

GIMPLE / passes

Low / high	фаза	описание
Low GIMPLE	Удаление SSA формы Leave SSA	Цитата из gccint.pdf «At the same time, we eliminate as many single-use temporaries as possible, **so the intermediate language is no longer GIMPLE, but GENERIC**»
Low GIMPLE	- return value optimization - return slot optimization - loop invariant motion - loop nest optimization - removal of empty loops - unrolling of small loops - etc.	Оптимизации, которые не требуют SSA

GIMPLE / GIMPLE_ASSIGN

Функция	Описание
gimple gimple_build_assign (tree lhs, tree rhs)	«lhs» = «rhs»
gimple gimple_build_assign_with_ops (enum tree_code subcode, tree lhs, tree op1, tree op2, tree op3)	«lhs» = «subcode» («op1», «op2», «op3»)
enum tree_code gimple_assign_rhs_code (const_gimple gs)	tree code для правой части
tree gimple_assign_lhs (gimple g)	tree для левой части инструкции
tree * gimple_assign_lhs_ptr (gimple g)	tree * для левой части инструкции
tree gimple_assign_rhs[1,2,3] (gimple g)	tree для операнда [1,2,3] правой части инструкции
tree gimple_assign_rhs[1,2,3]_ptr (gimple g)	tree * для операнда [1,2,3] правой части инструкции
Etc (see also gimple.h / gimple.c)	

GIMPLE / lexical scopes

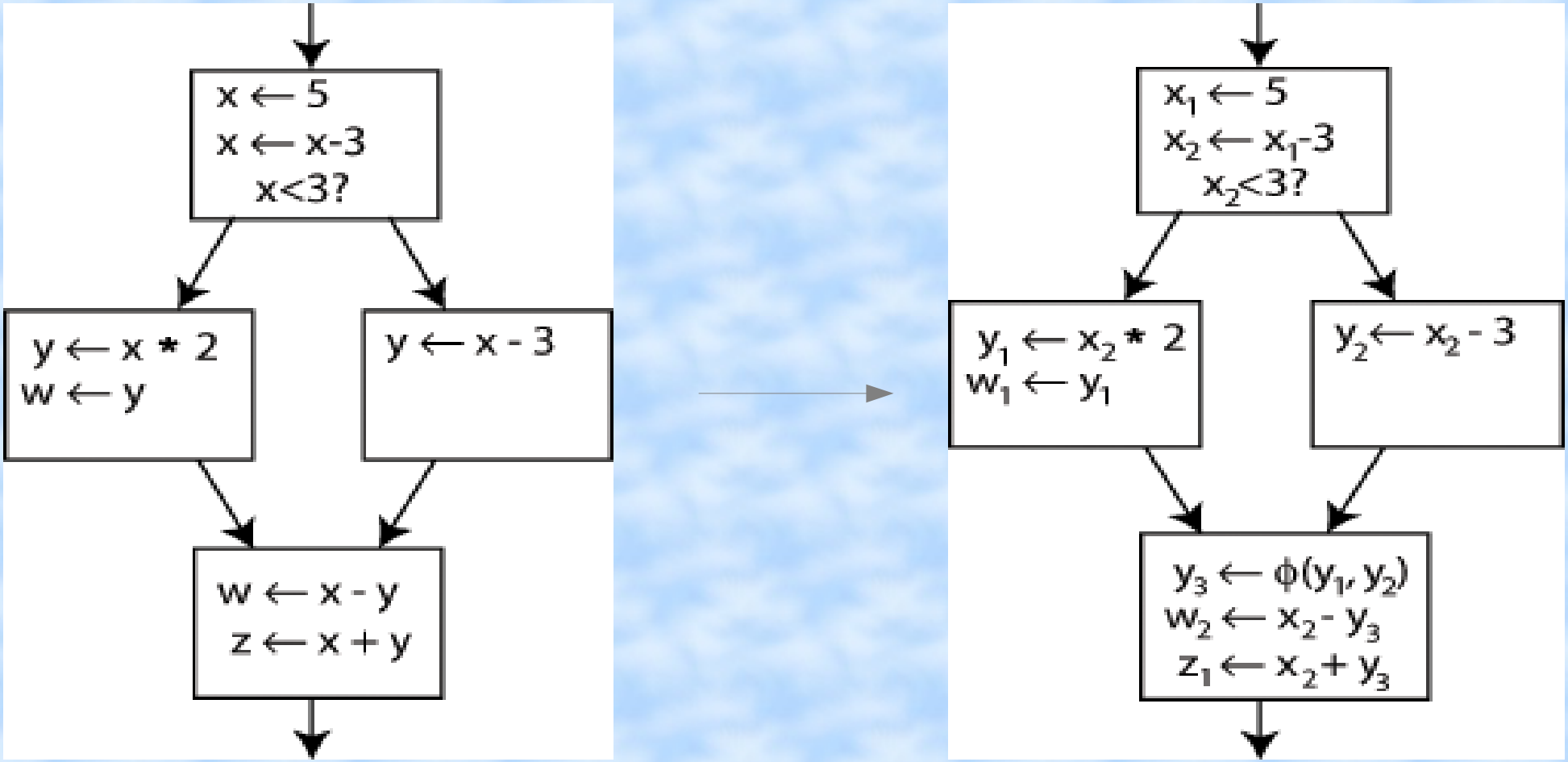
- GIMPLE_BIND <VARS, BLOCK, BODY>
  - VARS — набор переменных задекларированных в scope
  - BLOCK — используется для debug info
  - BODY — набор gimple-инструкций в блоке
- Мотивация:
  - анализ корректности указателей
  - Корректный вызов конструкторов и деструкторов для объектов

```
{
    int * pi;
    ...
    {
        int i = 0;
        ...
        pi = &i;
        ...
    }
    … i ...
}
```

```
{
    Cobj1 o1;
    ...
    {
        Cobj2 o2;
        ...
        {
            Cobj3 o3;
            ...
        }
    }
}
```

Ликбез №3 / SSA

Static Single Assignment form



Доступ в CFG

Обход ББ для функции

current_function_decl

(для проходов, которые делаются по функциям, декларация текущей функции хранится в current_function_decl)

```
basic_block b = ...;

FOR_EACH_BB ( b )
{
      edge e;
      edge_iterator ei;

      // b->index    - номер базового блока
      // b->succs    - массив исходящих рёбер
      // b->preds    - массив входящих рёбер

      FOR_EACH_EDGE( e, ei, b->succs )
      {
            // e->src     - базовый блок начало ребра
            // e->dest    - базовый блок конец ребра
      }
}
```

GIMPLE / SSA / SSA_NAME

- Вводится новый тип tree SSA_NAME
  (TREE_CODE(t) == SSA_NAME)
- tree SSA_NAME_VAR(t)
- int SSA_VERSION(t)

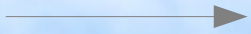
GIMPLE / SSA / def stmts

```
tree var ;
gimple stmt_def ;

stmt_def = SSA_NAME_DEF_STMT(var);
if ( gimple_code(stmt_def) == GIMPLE_PHI )
{
    // stmt_def  - phi - функция
}
else
{
    // stmt_def — обычная GIMPLE-инструкция
}
```

GIMPLE / SSA / virtual SSA_NAMEs

```
{
   int a, b, *p;
   if (...)
      p = &a;
   else
      p = &b;
   *p = 5;
   return *p;
}
```



```
{
   int a, b, *p;
   if (...)
      p = &a;
   else
      p = &b;

   # a = VDEF <a>
   # b = VDEF <b>
   *p = 5;

   # VUSE <a>
   # VUSE <b>
   return *p;
}
```

p = 5; может быть определением как **a**, так **b**

GIMPLE / SSA / итераторы #1

```
ssa_op_iter iter;
tree var;
gimple stmt = ...;
FOR_EACH_SSA_TREE_OPERAND(
                var, stmt, iter, SSA_OP_ALL_OPERANDS )
{
        // do something with var
}
```

Режимы работы итераторов:

```
#define SSA_OP_VIRTUAL_USES     (SSA_OP_VUSE)
#define SSA_OP_VIRTUAL_DEFS     (SSA_OP_VDEF)
#define SSA_OP_ALL_VIRTUALS     (SSA_OP_VIRTUAL_USES |SSA_OP_VIRTUAL_DEFS)
#define SSA_OP_ALL_USES         (SSA_OP_VIRTUAL_USES | SSA_OP_USE)
#define SSA_OP_ALL_DEFS         (SSA_OP_VIRTUAL_DEFS | SSA_OP_DEF)
#define SSA_OP_ALL_OPERANDS     (SSA_OP_ALL_USES | SSA_OP_ALL_DEFS)
```

GIMPLE / SSA / итераторы #2

- FOR_EACH_SSA_TREE_OPERAND
- FOR_EACH_SSA_USE_OPERAND
- FOR_EACH_SSA_DEF_OPERAND
- FOR_EACH_PHI_ARG
- FOR_EACH_PHI_OR_STMT_DEF

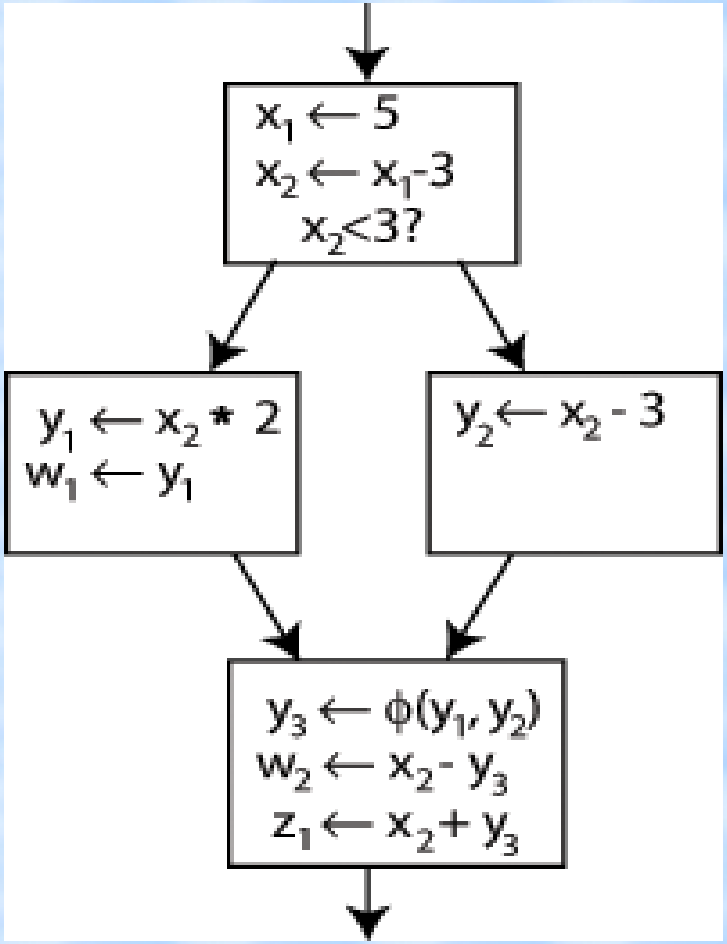
GIMPLE / SSA / phi - функции

```
basic_block bb = ...;
gimple_seq  phis = phi_nodes( bb );
gimple_stmt_iterator gsi;

for( gsi = gsi_start(phis); !gsi_end_p(gsi) ;
                                gsi_next(&gsi) )
{
    gimple phi = gsi_stmt(gsi);

    edge e;
    edge_iterator ei;

    FOR_EACH_EDGE( e, ei, bb->preds )
    {
       tree arg = PHI_ARG_DEF_FROM_EDGE ( phi, e );
    }
}
```



GENERIC / ARRAY_REF

temp_5 = data[0]; // data is array of float

```
tree t = gimple_assign_rhs1 (stmt);
if (TREE_CODE(t) == ARRAY_REF)
{
      tree var = TREE_OPERAND(t,0);
      tree index = TREE_OPERAND(t, 1);
      if (TREE_CODE(var) == VAR_DECL)
                if ( TREE_CODE(TYPE_TREE(var)) == ARRAY_TYPE )
                    if ( TREE_CODE(TYPE_TREE(TYPE_TREE(var))) == REAL_TYPE )
                                ...
      if (TREE_CODE(index) == INTEGER_CST)
                if ( TREE_CODE(TYPE_TREE(index)) == INTEGER_TYPE )
                            ...
}
```

GENERIC / MEM_REF

temp_6 = *loc_data_5; // pointer to float

```
tree t = gimple_assign_rhs1 (stmt);
if (TREE_CODE(t) == MEM_REF)
{
    tree base = TREE_OPERAND(t, 0);
    tree index = TREE_OPERAND(t, 1);
    if (TREE_CODE(base) == SSA_NAME)
        if ( TREE_CODE(TYPE_TREE(base)) == POINTER_TYPE )
            if ( TREE_CODE(TYPE_TREE(TYPE_TREE(base))) == REAL_TYPE )
                ...
    if (TREE_CODE(index) == INTEGER_CST)
        ...
}
```

GENERIC / COMPONENT_REF

temp_5 = myFolder.data[0]; // myFolder is struct
// data is float field

```
tree t = gimple_assign_rhs1 (stmt);
if (TREE_CODE(t) == ARRAY_REF)
{
    tree base = TREE_OPERAND(t, 0);
    if (TREE_CODE(base) == COMPONENT_REF)
    {
        if (TREE_CODE(TYPE_TREE(base)) == ARRAY_TYPE)
            …
        tree dcl_struct = TREE_OPERAND(base, 0);
        tree dcl_field = TREE_OPERAND(base, 1);
        if (TREE_CODE(dcl_struct) == VAR_DECL)
            if (TREE_CODE(TYPE_TREE(dcl_struct)) == RECORD_TYPE)
                …
        if (TREE_CODE(dcl_field) == FIELD_DECL)
            if (TREE_CODE(TYPE_TREE(dcl_field)) == ARRAY_TYPE)
                ...
    }
}
```

GENERIC / ADDR_EXPR

```
float stubVar = 353.0f;
printf("----> %p\n", &stubVar);
```

```
if (gimple_code(stmt) == GIMPLE_CALL)
{
      tree param0 = gimple_call_arg(stmt, 0);
      tree param1 = gimple_call_arg(stmt, 1);
      if ( TREE_CODE(param1) == ADDR_EXPR )
        if ( TREE_CODE(TYPE_TREE(param1)) == POINTER_TYPE )
          if ( TREE_CODE(TYPE_TREE(TYPE_TREE(param1))) == POINTER_TYPE )
            if  ( TREE_CODE(TREE_OPERAND(param1,0)) == VAR_DECL )
                  …
}
```

Как добыть знания?

	Source code	gccint.pdf
Работа с CFG	basic-block.h cfghooks.h cfghooks.c	15. Control Flow Graph
Работа с циклами	cfgloop.h cfgloop.c	14. Analysis & representation of loops
Работа с объектами tree	tree.h tree.def tree-iterator.h tree-iterator.c	11. Generic
Работа с gimple-инструкциями	gimple.h gimple.def gimple.c gimple-iterator.c	12. Gimple
GENERIC vs GIMPLE	gimplify.c (функция gimplify_function_tree)	12. Gimple
Работа с SSA	tree-ssa-operands.h	13.2. SSA Operands 13.3. Static Single Assignment
Взаимосвязь GIMPLE как структуры данных с GIMPLE как текстовым дампом	gimple-pretty-print.h gimple-pretty-print.c tree-pretty-print.h tree-pretty-print.c	-

Q & A ?