

Types and type based optimizations in GCC

Honza Hubička

Institute of Computer Science
Charles University
Prague

Mathematics and Statistics
University of Calgary
Calgary

GNU Cauldron 2015

Optimization infrastructure is optimized for...

- up to GCC 2.7: local optimization, instruction selection, register allocation, retargetability

Optimization infrastructure is optimized for...

- up to GCC 2.7: local optimization, instruction selection, register allocation, retargetability
- GCC 2.9: Codegen for RISC — scheduling, alias analysis, dealing with register windows, limited addressing modes, ...

Optimization infrastructure is optimized for...

- up to GCC 2.7: local optimization, instruction selection, register allocation, retargetability
- GCC 2.9: Codegen for RISC — scheduling, alias analysis, dealing with register windows, limited addressing modes, ...
- Early GCC 3 series: CFG transformations, basic loop optimizations, global optimization via dataflow, profile guided optimizations

Optimization infrastructure is optimized for...

- up to GCC 2.7: local optimization, instruction selection, register allocation, retargetability
- GCC 2.9: Codegen for RISC — scheduling, alias analysis, dealing with register windows, limited addressing modes, ...
- Early GCC 3 series: CFG transformations, basic loop optimizations, global optimization via dataflow, profile guided optimizations
- Tree-SSA: high level global optimization, loop optimization

Optimization infrastructure is optimized for...

- up to GCC 2.7: local optimization, instruction selection, register allocation, retargetability
- GCC 2.9: Codegen for RISC — scheduling, alias analysis, dealing with register windows, limited addressing modes, ...
- Early GCC 3 series: CFG transformations, basic loop optimizations, global optimization via dataflow, profile guided optimizations
- Tree-SSA: high level global optimization, loop optimization
- Dataflow branch: more robust framework for dataflow

Optimization infrastructure is optimized for...

- up to GCC 2.7: local optimization, instruction selection, register allocation, retargetability
- GCC 2.9: Codegen for RISC — scheduling, alias analysis, dealing with register windows, limited addressing modes, ...
- Early GCC 3 series: CFG transformations, basic loop optimizations, global optimization via dataflow, profile guided optimizations
- Tree-SSA: high level global optimization, loop optimization
- Dataflow branch: more robust framework for dataflow
- IPA: inlining, unreachable code removal, constant propagation

Optimization infrastructure is optimized for...

- up to GCC 2.7: local optimization, instruction selection, register allocation, retargetability
- GCC 2.9: Codegen for RISC — scheduling, alias analysis, dealing with register windows, limited addressing modes,
...
- Early GCC 3 series: CFG transformations, basic loop optimizations, global optimization via dataflow, profile guided optimizations
- Tree-SSA: high level global optimization, loop optimization
- Dataflow branch: more robust framework for dataflow
- IPA: inlining, unreachable code removal, constant propagation
- LTO: scalable whole program optimization

Optimization infrastructure is optimized for...

- up to GCC 2.7: local optimization, instruction selection, register allocation, retargetability
- GCC 2.9: Codegen for RISC — scheduling, alias analysis, dealing with register windows, limited addressing modes, ...
- Early GCC 3 series: CFG transformations, basic loop optimizations, global optimization via dataflow, profile guided optimizations
- Tree-SSA: high level global optimization, loop optimization
- Dataflow branch: more robust framework for dataflow
- IPA: inlining, unreachable code removal, constant propagation
- LTO: scalable whole program optimization

What is next?

Type based optimizations

What we already do

- Alias analysis (TBAA, `restrict`)
- Devirtualization (type tracking, type inheritance graph analysis)
- Loop bounds discovery
- Dependence analysis
- Data alignment analysis
- (IPA)-SRA

Type based optimizations

What we already do

- Alias analysis (TBAA, `restrict`)
- Devirtualization (type tracking, type inheritance graph analysis)
- Loop bounds discovery
- Dependence analysis
- Data alignment analysis
- (IPA)-SRA

Type based optimizations

What we already do

- Alias analysis (TBAA, `restrict`)
- Devirtualization (type tracking, type inheritance graph analysis)
- Loop bounds discovery
- Dependence analysis
- Data alignment analysis
- (IPA)-SRA

What we may want to do

- Structure reorganization
- Optimizing away heap allocations
- More advanced optimizations of datastructures...

Outline

- 1 Types in GCC
- 2 Existing optimizations
 - Alias analysis
 - Devirtualization
- 3 How we can improve?

Types in GCC

GCC types are represented as trees of the following types:

- ❶ VOID_TYPE
- ❷ **Basic types:** OFFSET_TYPE, ENUMERAL_TYPE, BOOLEAN_TYPE, INTEGER_TYPE, REAL_TYPE, NULLPTR_TYPE, FIXED_POINT_TYPE, POINTER_BOUNDS_TYPE

Types in GCC

GCC types are represented as trees of the following types:

- 1 `VOID_TYPE`
- 2 **Basic types:** `OFFSET_TYPE`, `ENUMERAL_TYPE`,
`BOOLEAN_TYPE`, `INTEGER_TYPE`, `REAL_TYPE`,
`NULLPTR_TYPE`, `FIXED_POINT_TYPE`,
`POINTER_BOUNDS_TYPE`
- 3 **Derived types:** `POINTER_TYPE`, `REFERENCE_TYPE`,
`COMPLEX_TYPE`, `VECTOR_TYPE`

Types in GCC

GCC types are represented as trees of the following types:

- ❶ `VOID_TYPE`
- ❷ **Basic types:** `OFFSET_TYPE`, `ENUMERAL_TYPE`,
`BOOLEAN_TYPE`, `INTEGER_TYPE`, `REAL_TYPE`,
`NULLPTR_TYPE`, `FIXED_POINT_TYPE`,
`POINTER_BOUNDS_TYPE`
- ❸ **Derived types:** `POINTER_TYPE`, `REFERENCE_TYPE`,
`COMPLEX_TYPE`, `VECTOR_TYPE`
- ❹ **Aggregates:** `ARRAY_TYPE`, `RECORD_TYPE`, `UNION_TYPE`,
`QUAL_UNION_TYPE`
- ❺ **Functions:** `FUNCTION_TYPE`, `METHOD_TYPE`

Types in GCC

GCC types are represented as trees of the following types:

- ❶ `VOID_TYPE`
- ❷ **Basic types:** `OFFSET_TYPE`, `ENUMERAL_TYPE`,
`BOOLEAN_TYPE`, `INTEGER_TYPE`, `REAL_TYPE`,
`NULLPTR_TYPE`, `FIXED_POINT_TYPE`,
`POINTER_BOUNDS_TYPE`
- ❸ **Derived types:** `POINTER_TYPE`, `REFERENCE_TYPE`,
`COMPLEX_TYPE`, `VECTOR_TYPE`
- ❹ **Aggregates:** `ARRAY_TYPE`, `RECORD_TYPE`, `UNION_TYPE`,
`QUAL_UNION_TYPE`
- ❺ **Functions:** `FUNCTION_TYPE`, `METHOD_TYPE`
- ❻ `LANG_TYPE`

Types in GCC

GCC types are represented as trees of the following types:

- ❶ `VOID_TYPE`
- ❷ **Basic types:** `OFFSET_TYPE`, `ENUMERAL_TYPE`,
`BOOLEAN_TYPE`, `INTEGER_TYPE`, `REAL_TYPE`,
`NULLPTR_TYPE`, `FIXED_POINT_TYPE`,
`POINTER_BOUNDS_TYPE`
- ❸ **Derived types:** `POINTER_TYPE`, `REFERENCE_TYPE`,
`COMPLEX_TYPE`, `VECTOR_TYPE`
- ❹ **Aggregates:** `ARRAY_TYPE`, `RECORD_TYPE`, `UNION_TYPE`,
`QUAL_UNION_TYPE`
- ❺ **Functions:** `FUNCTION_TYPE`, `METHOD_TYPE`
- ❻ `LANG_TYPE`

All types precisely describe memory layout and most of semantics. Basic types are built by helpers in `tree.c` and memory is set by `stor-layout.c`.

Types in GCC — the exotic part

Types are not completely language independent:

- Language frontend can hold additional info in a front-end specific way (such as template parameters). Issues with debug info now (almost) solved by early debug.

Types in GCC — the exotic part

Types are not completely language independent:

- Language frontend can hold additional info in a front-end specific way (such as template parameters). Issues with debug info now (almost) solved by early debug.
- `TYPE_NAME` holds name of type (in source language sense). Representation depends on the front-end (`TYPE_DECL` wrt `IDENTIFIER_NODE`)

Types in GCC — the exotic part

Types are not completely language independent:

- Language frontend can hold additional info in a front-end specific way (such as template parameters). Issues with debug info now (almost) solved by early debug.
- `TYPE_NAME` holds name of type (in source language sense). Representation depends on the front-end (`TYPE_DECL` wrt `IDENTIFIER_NODE`)
- `TYPE_STUB_DECL` basically helps debug info to identify identical types of different names. It has some frontend-specific meaning in it. For example in C++ it can be used to determine anonymous namespace

Types in GCC — the exotic part

Types are not completely language independent:

- Language frontend can hold additional info in a front-end specific way (such as template parameters). Issues with debug info now (almost) solved by early debug.
- `TYPE_NAME` holds name of type (in source language sense). Representation depends on the front-end (`TYPE_DECL` wrt `IDENTIFIER_NODE`)
- `TYPE_STUB_DECL` basically helps debug info to identify identical types of different names. It has some frontend-specific meaning in it. For example in C++ it can be used to determine anonymous namespace
- Definition of `TYPE_MAIN_VARIANT` originally chained a different qualification of same type, but for FEs do more

Types in GCC — the exotic part

Types are not completely language independent:

- Language frontend can hold additional info in a front-end specific way (such as template parameters). Issues with debug info now (almost) solved by early debug.
- `TYPE_NAME` holds name of type (in source language sense). Representation depends on the front-end (`TYPE_DECL` wrt `IDENTIFIER_NODE`)
- `TYPE_STUB_DECL` basically helps debug info to identify identical types of different names. It has some frontend-specific meaning in it. For example in C++ it can be used to determine anonymous namespace
- Definition of `TYPE_MAIN_VARIANT` originally chained a different qualification of same type, but for FEs do more
- Variants of types are deep copies. They get easily out of sync.

Outline

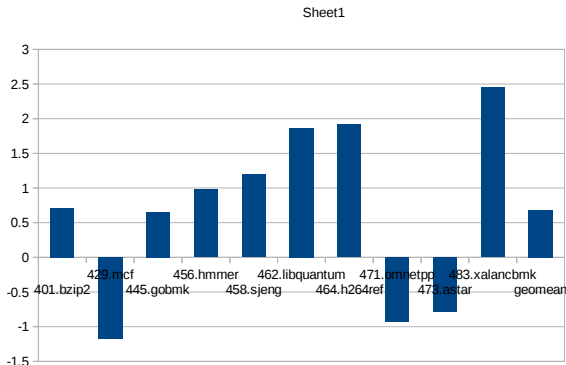
- 1 Types in GCC
- 2 Existing optimizations
 - Alias analysis
 - Devirtualization
- 3 How we can improve?

Outline

- 1 Types in GCC
- 2 Existing optimizations
 - Alias analysis
 - Devirtualization
- 3 How we can improve?

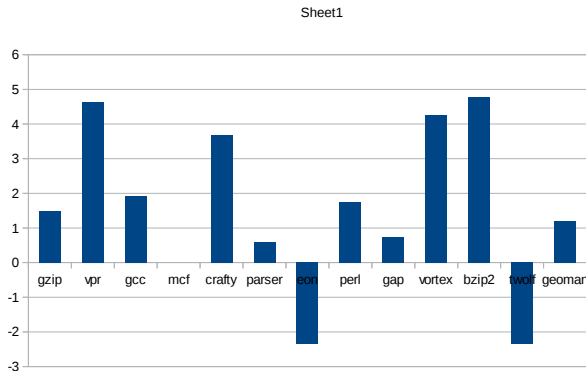
Why do we still care about TBAA?

Effect of `-fstruct-aliasing` on SPECINT2006, Intel I7 (in percent)



Why do we still care about TBAA?

Effect of `-fstruct-aliasing` on SPECINT2000, AMD Hammer, in 2003



TBAA overview

- `TYPE_CANONICAL` of each type gives a type that matters for TBAA
- Canonical types gets assigned alias set in partly frontend specific way `get_alias_set`
- Alias sets forms a partial order (transitive dag). TBAA queries then simplify to simple subset operations on alias set (`alias_set_subset_p`)
- Special alias set 0 is subset of everything and everything is subset of 0
- Alias set information is tagged to types and in RTL world to memory accesses via `MEM_ATTR` and thus readily available across the compilation. Langhooks are used when new middle-end types are introduced.

Canonical types WRT alias sets

- `TYPE_CANONICAL` define an equivalence across types.
This equivalence propagate up.

For example canonical type of `char` and `const char` are equivalent.

- `get_alias_set` define an coarser equivalence merging types without propagating up.

For example `int` and `unsigned_int` are equivalent, while `int *` and `unsigned_int *` are not.

Making TBAA to work with real code

`alias.c` implements number of hacks to get common code working

- Vector and array types alias components

Making TBAA to work with real code

`alias.c` implements number of hacks to get common code working

- Vector and array types alias components
- Until recently all pointers was equivalent to `void *`. Now we ignore qualifiers.

```
const int* const* alias int **
```

Making TBAA to work with real code

`alias.c` implements number of hacks to get common code working

- Vector and array types alias components
- Until recently all pointers was equivalent to `void *`. Now we ignore qualifiers.

```
const int* const* alias int **
```

- TBAA is used only to resolve true dependencies to make C++ placement new defines almost always.

LTO fun

- All frontend defined semantics is lost in translation and rebuilt
- `canonical_type_hash` define LTO canonical type equivalence.

LTO fun

- All frontend defined semantics is lost in translation and rebuilt
- `canonical_type_hash` define LTO canonical type equivalence.
 - Coarser than all type equivalence of all programming languages and equivalences implied by interoperability rules
 - Conflict with `useless_type_conversion`

LTO fun

- All frontend defined semantics is lost in translation and rebuilt
- `canonical_type_hash` define LTO canonical type equivalence.
 - Coarser than all type equivalence of all programming languages and equivalences implied by interoperability rules
 - Conflict with `useless_type_conversion`
- LTO's `get_alias_set` hook is coarser than all languages we support and a lot of globing is now in generic code
- Mixing `-fstrict-aliasing` and `-fno-strict-aliasing` translation units quickly lead to disabling TBAA completely.
- LTO TBAA is a lot less aggressive than non-LTO

LTO TBAA plans

LTO TBAA hitrate can be improved about 4 times.

LTO TBAA plans

LTO TBAA hitrate can be improved about 4 times.

- Review all language standards for interoperability rules and fix bugs (Fortran 70% done, Ada and Java seems easy). Document the code and add testcases.

LTO TBAA plans

LTO TBAA hitrate can be improved about 4 times.

- Review all language standards for interoperability rules and fix bugs (Fortran 70% done, Ada and Java seems easy). Document the code and add testcases.
- Untie `TYPE_CANONICAL` from `useless_type_conversion`

LTO TBAA plans

LTO TBAA hitrate can be improved about 4 times.

- Review all language standards for interoperability rules and fix bugs (Fortran 70% done, Ada and Java seems easy). Document the code and add testcases.
- Untie `TYPE_CANONICAL` from `useless_type_conversion`
- Restore TBAA for pointer types (important for C++). Done for non-LTO, LTO needs more love

LTO TBAA plans

LTO TBAA hitrate can be improved about 4 times.

- Review all language standards for interoperability rules and fix bugs (Fortran 70% done, Ada and Java seems easy). Document the code and add testcases.
- Untie `TYPE_CANONICAL` from `useless_type_conversion`
- Restore TBAA for pointer types (important for C++). Done for non-LTO, LTO needs more love
- Make canonical type merging finer when certain languages are not combined (i.e. no Fortran in translation unit)

LTO TBAA plans

LTO TBAA hitrate can be improved about 4 times.

- Review all language standards for interoperability rules and fix bugs (Fortran 70% done, Ada and Java seems easy). Document the code and add testcases.
- Untie `TYPE_CANONICAL` from `useless_type_conversion`
- Restore TBAA for pointer types (important for C++). Done for non-LTO, LTO needs more love
- Make canonical type merging finer when certain languages are not combined (i.e. no Fortran in translation unit)
- Add support for C++ ODR based type merging when there are no conflicts with other language (C++ inter-unit TBAA almost as strong as intra-unit!)

LTO TBAA plans

LTO TBAA hitrate can be improved about 4 times.

- Review all language standards for interoperability rules and fix bugs (Fortran 70% done, Ada and Java seems easy). Document the code and add testcases.
- Untie `TYPE_CANONICAL` from `useless_type_conversion`
- Restore TBAA for pointer types (important for C++). Done for non-LTO, LTO needs more love
- Make canonical type merging finer when certain languages are not combined (i.e. no Fortran in translation unit)
- Add support for C++ ODR based type merging when there are no conflicts with other language (C++ inter-unit TBAA almost as strong as intra-unit!)
- Consider streaming TBAA dag and having separate oracles for queries within translation unit and across

Outline

- 1 Types in GCC
- 2 Existing optimizations
 - Alias analysis
 - Devirtualization
- 3 How we can improve?

Devirtualization

- 1 `ipa-devirt`: Build type inheritance graph for the translation unit

Devirtualization

- ① `ipa-devirt`: Build type inheritance graph for the translation unit
- ② `ipa-polymorphic-call`: Detect type of memory locations
 - Location is dynamically typed by constructor call, `VPTR` store
 - Variables have static type which however may be temporarily changed during construction/destruction
 - Placement new can change types
 - Invocation of a method puts constraint on type of this pointer.

Devirtualization

- ① `ipa-devirt`: Build type inheritance graph for the translation unit
- ② `ipa-polymorphic-call`: Detect type of memory locations
 - Location is dynamically typed by constructor call, `VPTR` store
 - Variables have static type which however may be temporarily changed during construction/destruction
 - Placement new can change types
 - Invocation of a method puts constraint on type of this pointer.
- ③ `ipa-prop` and `get_dynamic_type`: Propagate type information forwards to polymorphic call sites

Devirtualization

- ❶ `ipa-devirt`: Build type inheritance graph for the translation unit
- ❷ `ipa-polymorphic-call`: Detect type of memory locations
 - Location is dynamically typed by constructor call, VPTR store
 - Variables have static type which however may be temporarily changed during construction/destruction
 - Placement new can change types
 - Invocation of a method puts constraint on type of this pointer.
- ❸ `ipa-prop` and `get_dynamic_type`: Propagate type information forwards to polymorphic call sites
- ❹ `ipa-devirt` Walk the type inheritance graph and if the known type info leads to only one virtual method, devirtualize.

LTO fun

- To build type inheritance graph we need to identify identical types across units
- TBAA sense of types is too coarse
- Structural compare is too fine

LTO fun

- To build type inheritance graph we need to identify identical types across units
- TBAA sense of types is too coarse
- Structural compare is too fine
- Fortunately C++ standard has an answer:
One Definition Rule (ODR) equivalence

LTO fun

- To build type inheritance graph we need to identify identical types across units
- TBAA sense of types is too coarse
- Structural compare is too fine
- Fortunately C++ standard has an answer:
One Definition Rule (ODR) equivalence
- Types with the same mangles names should match across units

LTO fun

- To build type inheritance graph we need to identify identical types across units
- TBAA sense of types is too coarse
- Structural compare is too fine
- Fortunately C++ standard has an answer:
One Definition Rule (ODR) equivalence
- Types with the same mangles names should match across units
- Lovely extra: ODR violation warnings

Clash with GIMPLE type system

- In C++ memory references asserts the type of object accessed. Seeing `(struct foo *ptr)->bar` one knows that `ptr` points to `struct foo` or its derived type
- In Gimple `(struct foo *ptr)->bar` is just an fancy `pointer_plus`

Outline

- 1 Types in GCC
- 2 Existing optimizations
 - Alias analysis
 - Devirtualization
- 3 How we can improve?

GCC are too expressive

Types consume a lot of memory (especially with LTO).

Type conversions confuse optimizers

Lets throw away what we do not need.

- Some information matters only for debug info
 - Type name
 - Source level qualifiers

GCC are too expressive

Types consume a lot of memory (especially with LTO).

Type conversions confuse optimizers

Lets throw away what we do not need.

- Some information matters only for debug info
 - Type name
 - Source level qualifiers
- Some information in type really matters only for memory access
 - Alias sets
 - Alignment and qualifiers, . . .

GCC are too expressive

Types consume a lot of memory (especially with LTO).

Type conversions confuse optimizers

Lets throw away what we do not need.

- Some information matters only for debug info
 - Type name
 - Source level qualifiers
- Some information in type really matters only for memory access
 - Alias sets
 - Alignment and qualifiers, . . .
- Some information matters only for devirtualization
 - ODR names
 - Explicit type inheritance hierarchy
- Some information matters for type tracking

GCC are too expressive

Types consume a lot of memory (especially with LTO).

Type conversions confuse optimizers

Lets throw away what we do not need.

- Some information matters only for debug info
 - Type name
 - Source level qualifiers
- Some information in type really matters only for memory access
 - Alias sets
 - Alignment and qualifiers, ...
- Some information matters only for devirtualization
 - ODR names
 - Explicit type inheritance hierarchy
- Some information matters for type tracking

OK we need almost everything. ...

A bit too many equivalences

- `TYPE_STUB_DECL` (debug info equivalence)

A bit too many equivalences

- `TYPE_STUB_DECL` (debug info equivalence)
- `TYPE_MAIN_VARIANT` (unqualified variant)

A bit too many equivalences

- `TYPE_STUB_DECL` (debug info equivalence)
- `TYPE_MAIN_VARIANT` (unqualified variant)
- `TYPE_CANONICAL` (coarser `TYPE_MAIN_VARIANT`, FE defined)
The “canonical” type for this type node, which is used by frontends to compare the type for equality with another type. If two types are equal (based on the semantics of the language), then they will have equivalent `TYPE_CANONICAL` entries.

A bit too many equivalences

- `TYPE_STUB_DECL` (debug info equivalence)
- `TYPE_MAIN_VARIANT` (unqualified variant)
- `TYPE_CANONICAL` (coarser `TYPE_MAIN_VARIANT`, FE defined)
The “canonical” type for this type node, which is used by frontends to compare the type for equality with another type. If two types are equal (based on the semantics of the language), then they will have equivalent `TYPE_CANONICAL` entries.
- `get_alias_set` equivalence (coarser `TYPE_CANONICAL`)

A bit too many equivalences

- `TYPE_STUB_DECL` (debug info equivalence)
- `TYPE_MAIN_VARIANT` (unqualified variant)
- `TYPE_CANONICAL` (coarser `TYPE_MAIN_VARIANT`, FE defined)
The “canonical” type for this type node, which is used by frontends to compare the type for equality with another type. If two types are equal (based on the semantics of the language), then they will have equivalent `TYPE_CANONICAL` entries.
- `get_alias_set` equivalence (coarser `TYPE_CANONICAL`)
- `useless_type_conversion_p`, `types_compatible_p`
(implicit GIMPLE types; coarser `TYPE_CANONICAL`)

A bit too many equivalences

- `TYPE_STUB_DECL` (debug info equivalence)
- `TYPE_MAIN_VARIANT` (unqualified variant)
- `TYPE_CANONICAL` (coarser `TYPE_MAIN_VARIANT`, FE defined)
The “canonical” type for this type node, which is used by frontends to compare the type for equality with another type. If two types are equal (based on the semantics of the language), then they will have equivalent `TYPE_CANONICAL` entries.
- `get_alias_set` equivalence (coarser `TYPE_CANONICAL`)
- `useless_type_conversion_p`, `types_compatible_p` (implicit GIMPLE types; coarser `TYPE_CANONICAL`)
- LTO canonical type equivalence (used to get TBAA working across language units, coarser than all `TYPE_CANONICAL`)

A bit too many equivalences

- `TYPE_STUB_DECL` (debug info equivalence)
- `TYPE_MAIN_VARIANT` (unqualified variant)
- `TYPE_CANONICAL` (coarser `TYPE_MAIN_VARIANT`, FE defined)
The “canonical” type for this type node, which is used by frontends to compare the type for equality with another type. If two types are equal (based on the semantics of the language), then they will have equivalent `TYPE_CANONICAL` entries.
- `get_alias_set` equivalence (coarser `TYPE_CANONICAL`)
- `useless_type_conversion_p`, `types_compatible_p` (implicit GIMPLE types; coarser `TYPE_CANONICAL`)
- LTO canonical type equivalence (used to get TBAA working across language units, coarser than all `TYPE_CANONICAL`)
- LTO type merging equivalence (finer than most)

A bit too many equivalences

- `TYPE_STUB_DECL` (debug info equivalence)
- `TYPE_MAIN_VARIANT` (unqualified variant)
- `TYPE_CANONICAL` (coarser `TYPE_MAIN_VARIANT`, FE defined)
The “canonical” type for this type node, which is used by frontends to compare the type for equality with another type. If two types are equal (based on the semantics of the language), then they will have equivalent `TYPE_CANONICAL` entries.
- `get_alias_set` equivalence (coarser `TYPE_CANONICAL`)
- `useless_type_conversion_p`, `types_compatible_p` (implicit GIMPLE types; coarser `TYPE_CANONICAL`)
- LTO canonical type equivalence (used to get TBAA working across language units, coarser than all `TYPE_CANONICAL`)
- LTO type merging equivalence (finer than most)
- C++ ODR type equivalence (coarser LTO type merging)

Longer term plans

Can we move information where it belongs?

- Early debug info
- `MEM_REFS` with explicit alias sets, alignment properties etc.
- Gimple types expressing operation semantics (i.e. one type per `types_compatible_p`)
- Separate type inheritance graph

Can we prevent information loss?

Longer term plans

Can we move information where it belongs?

- Early debug info
- `MEM_REFS` with explicit alias sets, alignment properties etc.
- Gimple types expressing operation semantics (i.e. one type per `types_compatible_p`)
- Separate type inheritance graph

Can we prevent information loss?

- Many optimizations care about type of dereference to memory. How do we want to track it in GIMPLE?

Thank you!

