
Statically Typed Trees in GCC

4th June 2004
GCC Summit 2004
Nathan Sidwell & Zachary Weinberg



www.codesourcery.com

Implementation

- Program described by a directed graph of nodes
 - tree node;
 - union tree_node {
 - struct tree_common common;
 - struct tree_type type;
 - struct tree_decl decl;
 - ...
- ```
};
typedef union tree_node *tree;
```



www.codesourcery.com

---

## Class

- Major conceptual categorization is the tree class
  - ‘c’ – constant
  - ‘1’, ‘2’, ‘<’, ‘r’ – monadic, dyadic, comparison & ref
  - ‘e’, ‘s’, ‘d’, ‘t’ – exprs, statements, decls & types
  - ‘x’ – everything else
- Each tree code is in exactly one class
  - Class does not tell you representation
  - Mapping from code by table lookup



www.codesourcery.com

---

## Substructure

- Only one of the tree union’s members is active for a particular tree code
  - This is the substructure
  - TS\_COMMON, TS\_INT\_CST, TS\_REAL\_CST, TS\_EXP, TS\_DECL ...
  - Mapping from code by function call
  - The accessor macros expect the correct substructure to be active



www.codesourcery.com

---

## Multipurposing

- A single field in a substructure can have different uses
  - Differently named accessor macros for each use
- type.values field
  - CONST\_DECLS for an ENUMERAL\_TYPE
  - Integer type describing indexes for ARRAY\_TYPE
  - FIELD\_DECLS for RECORD\_TYPE
  - ...



www.codesourcery.com

---

## Overloading

- A field for a particular tree code could point to different kinds of trees
  - An ADD\_EXPR’s operands could be DECLS, EXPRS or CONSTS
  - A TYPE’s TYPE\_NAME could be a TYPE\_DECL or an IDENTIFIER



www.codesourcery.com

## Multipurposing/Overloading

---

- Multipurposing
  - The tree code of the containing node determines what is valid
- Overloading
  - The tree code of the pointed to node can vary

CodeSOURCERY

www.codesourcery.com

## Measurements

---

- We hooked into the PCH machinery to measure tree dynamism
  - Measured at end of compilation (transient nodes missed)
  - Analyzed according to substructure or tree code
  - Counted the multipurposing & overloading of fields
  - Counted chain lengths
  - List node statistics are measured by list context

CodeSOURCERY

www.codesourcery.com

## List Distribution

---

- Lists have predictable context-dependent contents
  - TYPE is never used in C, rarely in C++
  - PURPOSE has 50% usage
  - ~7% memory size reduction by conversion to more specialized vectors

CodeSOURCERY

www.codesourcery.com

## Overloading

---

- Overloading is dynamically quite rare
  - Primary usage >90% of the time
  - Outliers make up < 1%
- An expression's operands are nearly always DECLs or CONSTs
  - Most expression statements are simple
- Source is overly complex

CodeSOURCERY

www.codesourcery.com

## Redesign

---

- Reduce runtime overhead
  - Wasted memory
  - Runtime checking
- Reduce maintainance
  - Simplify source code
  - Uniform mechanisms
  - Give language-dependent nodes equal status

CodeSOURCERY

www.codesourcery.com

## Static Typing

---

- Classify trees into one of four static kinds
  - Pointer points to one of those kinds
  - No cross-kind pointers
- Each kind of tree is made polymorphic
  - Substructures hold necessary information
  - No multipurposing

CodeSOURCERY

www.codesourcery.com

## The Four Kinds

- TYPE – ‘t’ class
- DECL – ‘d’ class
- EXPR – ‘1’, ‘2’, ‘<’, ‘r’, ‘e’, ‘s’ classes
- CONST – ‘c’ class
  
- We are not proposing removing the class concept

## Class ‘x’

- The ‘everything else’ class goes away
  - ERROR\_MARK → kind-specific errors
  - IDENTIFIER → non-tree structure
  - TREE\_LIST → dies
  - TREE\_VEC → kind-specific vectors

## Statements & Exprs

- Should statements and expressions be the same kind?
  - C-like languages distinguish them strongly
  - Measurement shows they have different dynamic contents
  - Expressions produce values and have a type, statements do not
  - Statements occur in lists, expressions do not

## Substructure

- Each of the four kinds of tree is like a C++ base class
  - Specific tree codes will add other information
  - We need checked conversions from base to derived
- C has no inheritance syntax
  - Nor an equivalent of `dynamic_cast`

## No Unions

- There will be no union of language-independent derivations
  - Conversion from derived to base is compile time checked
  - Conversions from base to derived type will be both compile and runtime checked

## Example

- `struct cst_base { ... };`
- `struct cst {  
  struct cst_base base;  
};`  
`typedef struct cst *CONST;`
- `struct cst_int {  
  struct cst_base base;  
  ...};`  
`typedef struct cst_int *INT_CONST;`
- `enum cst_kind {INT_CST, ...};`

## Accessing Common Fields

- Accessors for common fields must work when the pointer is either `CONST` or `INT_CONST`
  - `CONST c;`  
  `if (CONST_OVERFLOW (C)) ...`
  - `INT_CONST ic;`  
  `if (CONST_OVERFLOW (ic)) ...`
- Hence `CONST` is derived from `cst_base` too

CodeSourcery

www.codesourcery.com

## Example

- ```
#define base_cast(BASE, T)
((void)((&(T)->base
      == &((BASE)0)->base),
 &(T)->base)
```
- ```
#define CONST_OVERFLOW(C)
(base_cast (CONST, C)->overflow)
```
- Use `typeof` or `{...}` to avoid multiple evaluation

CodeSourcery

www.codesourcery.com

## Checked Casts

- Restrict base to derived casts to safe subset
  - Derived type is derived from base type (compile time check)
  - Object's dynamic type is the derived type (run time check)
  - ```
Foo (CONST c) {
  INT_CONST ic = INT_CONST(c);
  ...}
```

CodeSourcery

www.codesourcery.com

Example

- ```
#define derived_cast(ID, TYPE, T)
((void)((&((TYPE)0)->base
 == &((T)->base)), /* compile */
 (void)((T)->base.kind == ID
 || dcast_error (ID,T)), /* run */
 (TYPE)(T)) /* result */
```
- ```
#define INT_CONST(C)
  derived_cast (INT_CST, INT_CONST, C)
```

CodeSourcery

www.codesourcery.com

Features

- One-to-one mapping from tree code to substructure
 - Could be merged with a mapping table
 - Or augment of `tree.def` and autogenerate much of it
- No further derivation of a substructure
- `decl_lang_specific`, et al?
 - Perhaps force 3 levels - base, independent, lang_dependent

CodeSourcery

www.codesourcery.com

Tree Walkers

- `EXPR/STMT` tree walkers will need to know where the fields to walk are
 - We need meta information for each substructure to aid tree walkers
 - Use a table-driven `offsetof` implementation
Gengtype does this kind of stuff already

CodeSourcery

www.codesourcery.com

Lists

- Currently we have
 - Generic lists
 - Bespoke code for specialized lists
- We want generic code for creating bespoke lists
- Save memory with vectors not linked lists
 - Better cache utilization



www.codesourcery.com

Vector API

- Create a variable length vector type
 - Need both vectors of objects and vectors of pointers
 - A #define creates a set of static inline functions
 - Backed by generic out-of-line functions
 - They are statically typed
 - Per-type default initial length?
 - Naming convention?



www.codesourcery.com

Vector of Objects

- `VEC (Foo); // Instantiate ...`
 - `VEC_Foo /* The type */`
 - `VEC_Foo*VEC_Foo_create(unsigned);`
 - `Foo *VEC_Foo_append(VEC_Foo*,
 Foo const *);`
 - `Foo *VEC_Foo_index(VEC_Foo*,unsigned);`
 - `unsigned VEC_Foo_length(VEC_Foo*);`
 - `Foo *VEC_Foo_iterate (VEC_Foo *,
 unsigned);`



www.codesourcery.com

Example – Purpose & Value

- `struct pair {
 identifier *purpose;
 expr *value;
};`
- `VEC(pair); // Instantiate`
- `expr *purpose (VEC_pair *p, identifier *i){
 struct pair *e;
 for (ix = 0;
 (e = VEC_pair_Iterate (p, ix));
 ix++)
 if (e->purpose == i) return e->value;
 return NULL;
}`



www.codesourcery.com

Additional Functions

- `Foo *VEC_Foo_insert
 (VEC_Foo *, unsigned, Foo *);`
- `Foo *VEC_Foo_replace
 (VEC_Foo *, unsigned, Foo *);`
- `Foo *VEC_Foo_remove
 (VEC_Foo *, unsigned);`
- `void VEC_Foo_reserve
 (VEC_Foo *, unsigned);`



www.codesourcery.com

Implementation

- A vector is implemented as,
 - `struct VEC_pair {
 struct pair *elts;
 unsigned num;
 unsigned alloc;
};`
 - Simpler than VARRAY



www.codesourcery.com

Undecided

- Naming convention?
- Compile time constant size vectors?
- Unresizable vectors?
 - Using trailing array hack
- Unordered vectors?
 - Phi edge deletion

Adaptors

- There will be no cross-kind casting or use of trees
- But an expression's operands could be
 - Another expression
 - A DECL
 - A CONST

Adaptors

- Rather than have a union of all three kinds, (and some mechanism to distinguish)
 - Use the majority kind in the definition of the expression
 - Define adaptors of one kind into which another kind can be plugged

Example

- Define ADD_EXPR node's operands to be DECLs
- Define an EXPR_DECL, a DECL that just contains a pointer to an EXPR
- Define a CONST_DECL, which just contains a pointer to a CONST
 - Perhaps this is an anonymous const decl

Conversion Plan

- Do the work on a branch
 - Allows experimentation and collaboration
- But in stages that can be merged back
 - Each stage is a self contained improvement

Stage 1 – Flatten

- Remove multipurposing by creating substructures
 - This is a largely mechanical change that will force a cleanup of the accessor macro names

Stage 2 – Deoverload

- Remove abusive overloading
 - C declaration parsing
 - BINFOs
 - Template Arguments & Levels



www.codesourcery.com

Stage 3 – Peel the union

- Peel tree classes off the über union
 - Can be merged after each peel
- Identifiers – no longer a tree node
- ERROR_MARK – TYPE, EXPR & DECL errors
- Lists & Vectors – bespoke vectors
- Blocks
- Types
- Constants
- Expressions
- Declarations



www.codesourcery.com

The Trees

There is unrest in the source, there is trouble with the trees,
For the types are all the same, and memory has a squeeze.
The trouble with the decls: they are happy with their codes,
The exprs are just the same, and they grab up all the nodes.

The structs have made a union, and have an equal link,
The trees are just to samey, we will make them be distinct.
There will be no more code extension, for we found the lispy flaw,
And the trees will all be parted, by hatchet, axe and saw.

Apologies to Rush



www.codesourcery.com