cfa-cc Developer's Reference

Table of Contents

1 Overview 2
2 Compiler Framework 3
2.1 AST Representation 3
2.2 Compilation Passes 6
2.3 Data Structure Change WIP (new-ast) 8
3 Compiler Algorithm Documentation 12
3.1 Symbol Table 12
3.2 Type Environment and Unification 13
3.3 Expression Resolution 15
3.4 Assertion Satisfaction 16
4 Tests 17
4.1 Test Suites 17
4.2 Performance Reports 17

1 Overview

cfa-cc is the reference compiler for the Cforall programming language, which is a non-object-oriented extension to C.

Cforall attempts to introduce productive modern programming language features to C while maintaining as much backward-compatibility as possible, so that most existing C programs can seamlessly work with Cforall.

Since the Cforall project was dated back to the early 2000s, and only restarted in the past few years, there is a significant amount of legacy code in the current compiler codebase, with little proper documentation available. This becomes a difficulty while developing new features based on the previous implementations, and especially while diagnosing problems.

Currently, the Cforall team is also facing another problem: bad compiler performance. For the development of a new programming language, writing a standard library is an important part. The incompetence of the compiler causes building the library files to take tens of minutes, making iterative development and testing almost impossible. There is ongoing effort to rewrite the core data structure of the compiler to overcome the performance issue, but many bugs may appear during the work, and lack of documentation makes debugging extremely difficult.

This developer's reference will be continuously improved and eventually cover the compiler codebase. For now, the focus is mainly on the parts being rewritten, and also the performance bottleneck, namely the resolution algorithm. It is aimed to provide new developers to the project enough guidance and clarify the purposes and behavior of certain functions which are not mentioned in the previous Cforall research papers.

2 Compiler Framework

2.1 AST Representation

Source code input is first transformed into abstract syntax tree (AST) representation by the parser before analyzed by the compiler.

There are 4 major categories of AST nodes used by the compiler, along with some derived structures.

Declaration nodes

A declaration node represents either of:

- Type declaration: `struct, union, typedef` or type parameter (see Appendix A.3)
- Variable declaration
- Function declaration

Declarations are introduced by standard C declarations, with the usual scoping rules.

In addition, declarations can also be introduced by the `forall` clause (which is the origin of Cforall's name):

```
forall (<TypeParameterList> | <AssertionList>)
  declaration
```

Type parameters in Cforall are similar to C++ template type parameters. The Cforall declaration

```
forall (dtype T) ...
```

behaves similarly as the C++ template declaration

```
template <typename T> ...
```

Assertions are a distinctive feature of Cforall: contrary to the C++ template where arbitrary functions and operators can be used in a template definition, in a Cforall parametric function, operations on parameterized types must be declared in assertions.

Consider the following C++ template:

```
template <typename T> int foo(T t) {

  return bar(t) + baz(t);

}
```

Unless `bar` and `baz` are also parametric functions taking any argument type, they must be declared in the assertions, or otherwise the code will not compile:

```
forall (dtype T | { int bar(T); int baz(t); }) int foo (T t) {

  return bar(t) + baz(t);

}
```

Assertions are written using the usual function declaration syntax. The scope of type parameters and assertions is the following declaration.

Type nodes

A type node represents the type of an object or expression.

Named types reference the corresponding type declarations. The type of a function is its function pointer type (same as standard C).

With the addition of type parameters, named types may contain a list of parameter values (actual parameter types).

Statement nodes

Statement nodes represent the statements in the program, including basic expression statements, control flows and blocks.

Local declarations (within a block statement) are represented as declaration statements.

Expression nodes

Some expressions are represented differently in the compiler before and after resolution stage:

- Name expressions: NameExpr pre-resolution, VariableExpr post-resolution
- Member expressions: UntypedMemberExpr pre-resolution, MemberExpr post-resolution
- Function call expressions (including overloadable operators): UntypedExpr pre-resolution, ApplicationExpr post-resolution

The pre-resolution representations contain only the symbols. Post-resolution results link them to the actual variable and function declarations.

2.2 Compilation Passes

Compilation steps are implemented as passes, which follows a general structural recursion pattern on the syntax tree.

The basic work flow of compilation passes follows preorder and postorder traversal on tree data structure, implemented with visitor pattern, and can be loosely described with the following pseudocode:

```
Pass::visit (node_t node) {
  previsit(node);
  if (visit_children)
    for each child of node:
      child.accept(this);
  postvisit(node);
}
```

Operations in `previsit()` happen in preorder (top to bottom) and operations in `postvisit()` happen in postorder (bottom to top). The precise order of recursive operations on child nodes can be found in Common/PassVisitor.impl.h (old) and AST/Pass.impl.hpp (new).

Implementations of compilation passes need to follow certain conventions:

- Passes **should not** directly override the `visit` method (Non-virtual Interface principle); if a pass desires different recursion behavior, it should set `visit_children` to false and perform recursive calls manually within previsit or postvisit procedures. To enable this option, inherit from WithShortCircuiting mixin.
- previsit may mutate the node but **must not** change the node type or return null.
- postvisit may mutate the node, reconstruct it to a different node type, or delete it by returning null.
- If the previsit or postvisit method is not defined for a node type, the step is skipped. If the return type is declared as void, the original node is returned by default. These behaviors are controlled by template specialization rules; see Common/PassVisitor.proto.h (old) and AST/Pass.proto.hpp (new) for details.

Other useful mixin classes for compilation passes include:

- WithGuards allows saving values of variables and restore automatically upon exiting the current node.
- WithVisitorRef creates a wrapped entity of current pass (the actual argument passed to recursive calls internally) for explicit recursion, usually used together with WithShortCircuiting.
- WithSymbolTable gives a managed symbol table with built-in scoping rule handling (e.g. on entering and exiting a block statement)

NOTE: If a pass extends the functionality of another existing pass, due to C++ overloading resolution rules, it **must** explicitly introduce the inherited previsit and postvisit procedures to its own scope, or otherwise they will not be picked up by template resolution:

```
class Pass2: public Pass1 {
  using Pass1::previsit;
  using Pass1::postvisit;
  // new procedures
}
```

2.3 Data Structure Change WIP (new-ast)

It has been observed that excessive copying of syntax tree structures accounts for a majority of computation cost and significantly slows down the compiler. In the previous implementation of the syntax tree, every internal node has a unique parent; therefore all copies are required to duplicate everything down to the bottom. A new, experimental re-implementation of the syntax tree (source under directory AST/ hereby referred to as "new-ast") attempts to overcome this issue with a functional approach that allows sharing of common sub-structures and only makes copies when necessary.

The core of new-ast is a customized implementation of smart pointers, similar to `std::shared_ptr` and `std::weak_ptr` in C++ standard library. Reference counting is used to detect sharing and allows optimization. For a purely functional (a.k.a. immutable) data structure, all mutations are modelled by shallow copies along the path of mutation. With reference counting optimization, unique nodes are allowed to be mutated in place. This however, may potentially introduce some complications and bugs; a few issues are discussed near the end of this section.

Source: AST/Node.hpp

`class ast::Node` is the base class of all new-ast node classes, which implements reference counting mechanism. Two different counters are recorded: "strong" reference count for number of nodes semantically owning it; "weak" reference count for number of nodes holding a mere reference and only need to observe changes.

`class ast::ptr_base` is the smart pointer implementation and also takes care of resource management.

Direct access through the smart pointer is read-only. A mutable access should be obtained by calling `shallowCopy` or `mutate` as below.

Currently, the weak pointers are only used to reference declaration nodes from a named type, or a variable expression. Since declaration nodes are intended to denote unique entities in the program, weak pointers always point to unique (unshared) nodes. This may change in the future, and weak references to shared nodes may introduce some problems; see `mutate` function below.

All node classes should always use smart pointers in the structure and should not use raw pointers.

```
void ast::Node::increment(ref_type ref)
```

Increments this node's strong or weak reference count.

```
void ast::Node::decrement(ref_type ref, bool do_delete = true)
```

Decrements this node's strong or weak reference count. If strong reference count reaches zero, the node is deleted by default.

NOTE: Setting do_delete to false may result in a detached node. Subsequent code should manually delete the node or assign it to a strong pointer to prevent memory leak.

Reference counting functions are internally called by `ast::ptr_base`.

```
template<typename node_t>
node_t * shallowCopy(const node_t * node)
```

Returns a mutable, shallow copy of node: all child pointers are pointing to the same child nodes.

```
template<typename node_t>
node_t * mutate(const node_t * node)
```

If node is unique (strong reference count is 1), returns a mutable pointer to the same node. Otherwise, returns `shallowCopy(node)`.

It is an error to mutate a shared node that is weak-referenced. Currently this does not happen. The problem may appear once weak pointers to shared nodes (e.g. expression nodes) are used; special care will be needed.

NOTE: This naive uniqueness check may not be sufficient in some cases. A discussion of the issue is presented at the end of this section.

```
template<typename node_t, typename parent_t, typename field_t,
typename assn_t>
const node_t * mutate_field(const node_t * node, field_t parent_t::*
field, assn_t && val)
```

```
template<typename node_t, typename parent_t, typename coll_t, typename ind_t, typename field_t>

const node_t * mutate_field_index(const node_t * node, coll_t parent_t::* field, ind_t i, field_t && val)
```

Helpers for mutating a field on a node using pointer to member (creates shallow copy when necessary).

Issue: Undetected sharing

The `mutate` behavior described above has a problem: deeper shared nodes may be mistakenly considered as unique. This diagram shows how the problem could arise:

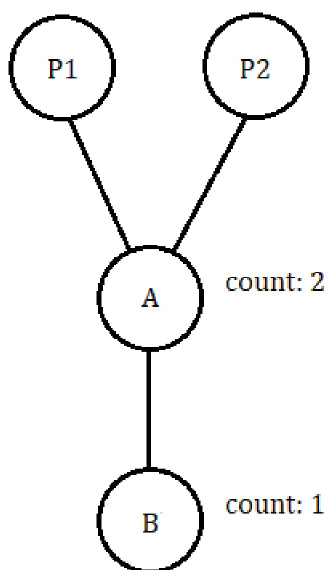

Figure 1: Deep sharing of nodes

Suppose that we are working on the tree rooted at P1, which is logically the chain P1-A-B and P2 is irrelevant, and then mutate(B) is called. The algorithm considers B as unique since it is only directly owned by A. However, the other tree P2-A-B indirectly shares the node B and is therefore wrongly mutated.

To partly address this problem, if the mutation is called higher up the tree, a chain mutation helper can be used:

Source: AST/Chain.hpp

```
template<typename node_t, Node::ref_type ref_t>

auto chain_mutate(ptr_base<node_t, ref_t> & base)
```

This function returns a chain mutator handle which takes pointer-to-member to go down the tree while creating shallow copies as necessary; see `struct _chain_mutator` in the source code for details.

For example, in the above diagram, if mutation of B is wanted while at P1, the call using chain_mutate looks like the following:

```
chain_mutate(P1.a)(&A.b) = new_value_of_b;
```

Note that if some node in chain mutate is shared (therefore shallow copied), it implies that every node further down will also be copied, thus correctly executing the functional mutation algorithm. This example code creates copies of both A and B and performs mutation on the new nodes, so that the other tree P2-A-B is untouched.

However, if a pass traverses down to node B and performs mutation, for example, in postvisit(B), information on sharing higher up is lost. Since the new-ast structure is only in experimental use with the resolver algorithm, which mostly rebuilds the tree bottom-up, this issue does not actually happen. It should be addressed in the future when other compilation passes are migrated to new-ast and many of them contain procedural mutations, where it might cause accidental mutations to other logically independent trees (e.g. common sub-expression) and become a bug.

3 Compiler Algorithm Documentation

This documentation currently covers most of the resolver, data structures used in variable and expression resolution, and a few directly related passes. Later passes involving code generation is not included yet; documentation for those will be done afterwards.

3.1 Symbol Table

NOTE: For historical reasons, the symbol table data structure was called "indexer" in the old implementation. Hereby we will be using the name SymbolTable everywhere.

The symbol table stores a mapping from names to declarations and implements a similar name space separation rule, and the same scoping rules in standard C.[1] The difference in name space rule is that `typedef` aliases are no longer considered ordinary identifiers.

In addition to C tag types (struct, union, enum), Cforall introduces another tag type, **trait**, which is a named collection of assertions.

Source: AST/SymbolTable.hpp

Source: SymTab/Indexer.h

`SymbolTable::addId(const DeclWithType * decl)`

Since Cforall allows overloading of variables and functions, ordinary identifier names need to be mangled. The mangling scheme is closely based on the Itanium C++ ABI,[2] while making adaptations to Cforall specific features, mainly assertions and overloaded variables by type. Naming conflicts are handled by mangled names; lookup by name returns a list of declarations with the same literal identifier name.

---

[1] ISO/IEC 9899:1999, Sections 6.2.1 and 6.2.3

[2] https://itanium-cxx-abi.github.io/cxx-abi/abi.html, Section 5.1

```
SymbolTable::addStruct(const StructDecl * decl)
SymbolTable::addUnion(const UnionDecl * decl)
SymbolTable::addEnum(const EnumDecl * decl)
SymbolTable::addTrait(const TraitDecl * decl)
```

Adds a tag type declaration to the symbol table.

```
SymbolTable::addType(const NamedTypeDecl * decl)
```

Adds a `typedef` alias to the symbol table.

C Incompatibility Note: Since Cforall allows using struct, union and enum type names without the keywords, `typedef` names and tag type names cannot be disambiguated by syntax rules. Currently the compiler puts them together and disallows collision. The following program is valid C but not valid Cforall:

```
struct A {};
typedef int A;
// gcc: ok, cfa: Cannot redefine typedef A
```

In actual practices however, such usage is extremely rare, and `typedef struct A A;` is not considered an error, but silently discarded. Therefore, we expect this change to have minimal impact on existing C programs.

Meanwhile, the following program is allowed in Cforall:

```
typedef int A;
void A();
// gcc: A redeclared as different kind of symbol, cfa: ok
```

3.2 Type Environment and Unification

The core of parametric type resolution algorithm.

Type Environment organizes type parameters in **equivalent classes** and maps them to actual types. Unification is the algorithm that takes two (possibly parametric) types and parameter mappings and attempts to produce a common type by matching the type environments.

The unification algorithm is recursive in nature and runs in two different modes internally:

- **Exact** unification mode requires equivalent parameters to match perfectly;
- **Inexact** unification mode allows equivalent parameters to be converted to a common type.

For a pair of matching parameters (actually, their equivalent classes), if either side is open (not bound to a concrete type yet), they are simply combined.

Within inexact mode, types are allowed to differ on their cv-qualifiers; additionally, if a type never appear either in parameter list or as the base type of a pointer, it may also be widened (i.e. safely converted). As Cforall currently does not implement subclassing similar to object-oriented languages, widening conversions are on primitive types only, for example the conversion from `int` to `long`.

The need for two unification modes come from the fact that parametric types are considered compatible only if all parameters are exactly the same (not just compatible). Pointer types also behaves similarly; in fact, they may be viewed as a primitive kind of parametric types. `int*` and `long*` are different types, just like `vector(int)` and `vector(long)` are, for the parametric type `vector(T)`.

The resolver should use the following "public" functions: [3]

Source: ResolvExpr/Unify.cc

```
bool unify(const Type *type1, const Type *type2, TypeEnvironment &env,
OpenVarSet &openVars, const SymbolTable &symtab, Type *&commonType)
```

Attempts to unify type1 and type2 with current type environment.

If operation succeeds, `env` is modified by combining the equivalence classes of matching parameters in type1 and type2, and their common type is written to `commonType`.

If operation fails, returns false.

```
bool typesCompatible(const Type * type1, const Type * type2, const
SymbolTable &symtab, const TypeEnvironment &env)
```

```
bool typesCompatibleIgnoreQualifiers(const Type * type1, const Type *
type2, const SymbolTable &symtab, const TypeEnvironment &env)
```

[3] Actual code also tracks assertions on type parameters; those extra arguments are omitted here for conciseness.

Determines if type1 and type2 can possibly be the same type. The second version ignores the outermost cv-qualifiers if present. [4]

The call has no side effect.

NOTE: No attempts are made to widen the types (exact unification is used), although the function names may suggest otherwise. E.g. typesCompatible(int, long) returns false.

3.3 Expression Resolution

The design of the current version of expression resolver is outlined in the Ph.D. Thesis from Aaron Moss [1].

A summary of the resolver algorithm for each expression type is presented below.

All overloadable operators are modelled as function calls. For a function call, interpretations of the function and arguments are found recursively. Then the following steps produce a filtered list of valid interpretations:

1) From all possible combinations of interpretations of the function and arguments, those where argument types may be converted to function parameter types are considered valid.
2) Valid interpretations with the minimum sum of argument costs are kept.
3) Argument costs are then discarded; the actual cost for the function call expression is the sum of conversion costs from the argument types to parameter types.
4) For each return type, the interpretations with satisfiable assertions are then sorted by actual cost computed in step 3. If for a given type, the minimum cost interpretations are not unique, it is said that for that return type the interpretation is ambiguous. If the minimum cost interpretation is unique but contains an ambiguous argument, it is also considered ambiguous.

Therefore, for each return type, the resolver produces either of:

- No alternatives
- A single valid alternative
- An ambiguous alternative

Note that an ambiguous alternative may be discarded at the parent expressions because a different return type matches better for the parent expressions.

[4] In `const int * const`, only the second const is ignored.

The non-overloadable expressions in Cforall are: cast expressions, address-of (unary &) expressions, short-circuiting logical expressions (&&, ||) and ternary conditional expression (?:).

For a cast expression, the convertible argument types are kept. Then the result is selected by lowest argument cost, and further by lowest conversion cost to target type. If the lowest cost is still not unique, or an ambiguous argument interpretation is selected, the cast expression is ambiguous. In an expression statement, the top level expression is implicitly cast to void.

For an address-of expression, only lvalue results are kept and the minimum cost is selected.

For logical expressions && and ||, arguments are implicitly cast to bool, and follow the rule of cast expression as above.

For the ternary conditional expression, the condition is implicitly cast to bool, and the branch expressions must have compatible types. Each pair of compatible branch expression types produce a possible interpretation, and the cost is defined as the sum of expression costs plus the sum of conversion costs to the common type.

TODO: Write a specification for expression costs.

3.4 Assertion Satisfaction

The resolver tries to satisfy assertions on expressions only when it is needed: either while selecting from multiple alternatives of a same result type for a function call (step 4 of resolving function calls), or upon reaching the top level of an expression statement.

Unsatisfiable alternatives are discarded. Satisfiable alternatives receive **implicit parameters**: in Cforall, parametric functions are designed such that they can be compiled separately, as opposed to C++ templates which are only compiled at instantiation. Given a parametric function definition:

```
forall (otype T | {void foo(T);})

void bar (T t) { foo(t); }
```

The function `bar` does not know which `foo` to call when compiled without knowing the call site, so it requests a function pointer to be passed as an extra argument. At the call site, implicit parameters are automatically inserted by the compiler.

TODO: Explain how recursive assertion satisfaction and polymorphic recursion work.

4 Tests

4.1 Test Suites

Automatic test suites are located under the `tests/` directory. A test case consists of an input CFA source file (name ending with `.cfa`), and an expected output file located in `.expect/` directory relative to the source file, with the same file name ending with `.txt`. So a test named `tuple/tupleCast` has the following files, for example:

```
tests/
..  tuple/
......  .expect/
..........  tupleCast.txt
......  tupleCast.cfa
```

If compilation fails, the error output is compared to the expect file. If compilation succeeds, the built program is run and its output compared to the expect file.

To run the tests, execute the test script `test.py` under the `tests/` directory, with a list of test names to be run, or `--all` to run all tests. The test script reports test cases fail/success, compilation time and program run time.

4.2 Performance Reports

To turn on performance reports, pass `-S` flag to the compiler.

3 kinds of performance reports are available:

- Time, reports time spent in each compilation step
- Heap, reports number of dynamic memory allocations, total bytes allocated, and maximum heap memory usage
- Counters, for certain predefined statistics; counters can be registered anywhere in the compiler as a static object, and the interface can be found at `Common/Stats/Counter.h`.

It is suggested to run performance tests with optimized build (g++ flag -O3)

References

[1] Aaron Moss (2019). C∀ Type System Implementation. UWSpace.
http://hdl.handle.net/10012/14584