cfa-cc Developer's Reference

Fangren Yu

January 13, 2021

Contents

1 Overview

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

Since the C∀ project dates 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 documentation. The lack of documentation makes it difficult to develop new features from the current implementation and diagnose problems.

Currently, the C∀ team is also facing poor compiler performance. For the development of a new programming language, writing standard libraries is an important component. The slow compiler causes building of the library files to take tens of minutes, making iterative development and testing almost impossible. There is an ongoing effort to rewrite the core data-structure of the compiler to overcome the performance issue, but many bugs have appeared during this work, and lack of documentation is hampering debugging.

This developer's reference manual begins the documentation and should be continuously improved until it eventually covers the entire compiler codebase. For now, the focus is mainly on the parts being rewritten, and also the primary performance bottleneck, namely the resolution algorithm. Its aimed is to provide new project developers with guidance in understanding the codebase, and clarify the purpose and behaviour of certain functions that are not mentioned in the previous C∀ research papers [1, 2, 3].

2 Compiler Framework

C∀ source code is first transformed into an abstract syntax tree (AST) by the parser before analyzed by the compiler.

2.1 AST Representation

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

2.1.1 Declaration Nodes

A declaration node represents either of:

- type declaration: **struct**, **union**, **typedef** or type parameter (see Appendix A.1, p. 10)
- variable declaration
- function declaration

Declarations are introduced by standard C declarations, with the usual scoping rules. In addition, declarations can also be qualified by the **forall** clause (which is the origin of C∀'s name):

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

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

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

behaves similarly to the C++ template declaration

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

Assertions are a distinctive feature of C∀, similar to *interfaces* in D and Go, and *traits* in Rust. Contrary to the C++ template where arbitrary functions and operators can be used in a template definition, in a C∀ parametric function, operations on parameterized types must be declared in assertions. Consider the following C++ template:

```
template forall<typename T> T foo( T t ) {
    return t + t * t;
}
```

where there are no explicit requirements on the type T. Therefore, the C++ compiler must deduce what operators are required during textual (macro) expansion of the template at each usage. As a result, templates cannot be compiled. C∀ assertions specify restrictions on type parameters:

```
forall( dtype T | { T ?+?( T, T ); T ?*?( T, T ) } ) int foo ( T t ) {
    return t + t * t;
}
```

Assertions are written using the usual C∀ function declaration syntax. Only types with operators "+" and "*" work with this function, and the function prototype is sufficient to allow separate compilation.

Type parameters and assertions are used in the following compiler data-structures.

2.1.2 Type Nodes

Type nodes represent 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).

2.1.3 Statement Nodes

Statement nodes represent the executable statements in the program, including basic expression statements, control flows and blocks. Local declarations (within a block statement) are represented as declaration statements.

2.1.4 Expression Nodes

Some expressions are represented differently before and after the 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 representation contains only the symbols. Post-resolution links 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 workflow of compilation passes follows preorder and postorder traversal on the AST 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 follow certain conventions:

- Passes **should not** directly override the visit method (Non-virtual Interface principle); if a pass desires different recursion behaviour, it should set visit_children to false and perform recursive calls manually within previsit or postvisit procedures. To enable this option, inherit from the WithShortCircuiting mixin.
- previsit may mutate the node but **must not** change the node type or return nullptr.
- postvisit may mutate the node, reconstruct it to a different node type, or delete it by returning nullptr.
- 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 behaviours 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 and restoring variable values automatically upon entering/exiting the current node.
- WithVisitorRef creates a wrapped entity for the 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 are not picked up by template resolution:

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

2.3 Data Structure Change (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 the entire subtree. 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 the C++ standard library. Reference counting is used to detect sharing and allowing certain optimizations. For a purely functional (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.

2.3.1 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 property 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 structure definitions versus raw pointers. Function

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

increments this node's strong or weak reference count. Function

```
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. **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. Function

```
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. Function

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

returns a mutable pointer to the same node, if the node is unique (strong reference count is 1); otherwise, it returns shallowCopy(node). It is an error to mutate a shared node that is weak-referenced. Currently this does not happen. A problem may appear once weak pointers to shared nodes (*e.g.*, expression nodes) are used; special care is 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. Functions

```
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)
```

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

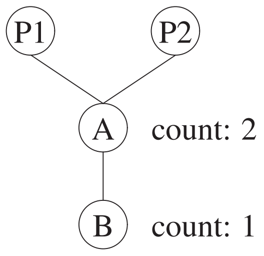

Figure 1: Deep sharing of nodes

2.3.2 Issue: Undetected Sharing

The mutate behaviour described above has a problem: deeper shared nodes may be mistakenly considered as unique. Figure 1 shows how the problem could arise: Given the tree rooted at P1, which is logically the chain P1-A-B, and P2 is irrelevant, assume 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.

2.3.3 Source: AST/Chain.hpp

Function

```
template<typename node_t, Node::ref_type ref_t>
auto chain_mutate(ptr_base<node_t, ref_t> & base)
```

returns a chain mutator handle that 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: if some node in chain mutate is shared (therefore shallow copied), it implies that every node further down is also 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 compiler algorithm documentation covers most of the resolver, data structures used in variable and expression resolution, and a few directly related passes. Later passes involving code generation are not included yet; documentation for those will be done latter.

3.1 Symbol Table

NOTE: For historical reasons, the symbol-table data-structure is called indexer in the old implementation. Hereby, the name is changed to SymbolTable. The symbol table stores a mapping from names to declarations, implements a similar name-space separation rule, and provides the same scoping rules as 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**), C∀ introduces another tag type, trait, which is a named collection of assertions.

3.1.1 Source: AST/SymbolTable.hpp

Function

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

provides name mangling of identifiers, since C∀ allows overloading of variables and functions. The mangling scheme is closely based on the Itanium C++ ABI,[2] while making adaptations to C∀ 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 identifier name. Functions

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

add a tag-type declaration to the symbol table. Function

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

adds a **typedef** alias to the symbol table.

C Incompatibility Note: Since C∀ allows using **struct**, **union** and **enum** type-names without a prefix keyword, as in C++, **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 invalid C∀ (and C++):

```
struct A {};
typedef int A; // gcc: ok, cfa: Cannot redefine typedef A
struct A sa; // C disambiguates via struct prefix
A ia;
```

In practices, such usage is extremely rare, and hence, this change (as in C++) has minimal impact on existing C programs. The declaration

```
struct A {};
typedef struct A A; // A is an alias for struct A
A a;
struct A b;
```

is not an error because the alias name is identical to the original. Finally, the following program is allowed in C∀:

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

because the function name is mangled.

[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

3.2 Type Environment and Unification

The following core ideas underlie the parametric type-resolution algorithm. A type environment organizes type parameters into **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 information in 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 combined.

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

The need for two unification modes comes 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 *(T) / vector(T), respectively.

The resolver uses the following **public** functions:[3]

3.2.1 Source: ResolvExpr/Unify.cc

Function

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

returns a boolean indicating if the unification succeeds or fails after attempting to unify type1 and type2 within current type environment. If the unify 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 the unify fails, nothing changes. Functions

```
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)
```

return a boolean indicating if types type1 and type2 can possibly be the same type. The second version ignores the outermost cv-qualifiers if present.[4] These function have no side effects.

NOTE: No attempt is 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 by Aaron Moss [3]. A summary of the resolver algorithm for each expression type is presented below.

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

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

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, that return type is ambiguous. If the minimum cost interpretation is unique but contains an ambiguous argument, it is also ambiguous.

Therefore, for each return type, the resolver produces:

- no alternatives
- a single valid alternative
- an ambiguous alternative

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

The *non*-overloadable expressions in C∀ 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 rules fr cast expression 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 the expression costs plus the sum of conversion costs to the common type.

3.4 Conversion and Application Cost

There were some unclear parts in the previous documentation in the cost system, as described in the Moss thesis [3], section 4.1.2. Some clarification are presented in this section.

1. Conversion to a type denoted by parameter may incur additional cost if the match is not exact. For example, if a function is declared to accept (T, T) and receives (**int**, **long**), T is deducted **long** and an additional widening conversion cost is added for **int** to T.
2. The specialization level of a function is the sum of the least depth of an appearance of a type parameter (counting pointers, references and parameterized types), plus the number of assertions. A higher specialization level is favoured if argument conversion costs are equal.

3. Coercion of pointer types is only allowed in explicit cast expressions; the only allowed implicit pointer casts are adding qualifiers to the base type and cast to **void***, and these counts as safe conversions. Note that implicit cast from **void** * to other pointer types is no longer valid, as opposed to standard C.

3.5 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 C∀, parametric functions may be separately compiled, as opposed to C++ templates which are only compiled at instantiation. Given the 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.

Implementation of implicit parameters is discussed in Appendix A.3.

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 (suffix .cfa), and an expected output file located in the tests/.expect/ directory, with the same file name ending with suffix .txt. For example, the test named tests/tuple/tupleCast.cfa 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 the compilation succeeds but does not generate an executable, the compilation output is compared to the expect file. If the compilation succeeds and generates an executable, the executable is run and its output is 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 (or make all-tests) to run all tests. The test script reports test cases fail/success, compilation time and program run time. To see all the options available for test.py using the --help option.

4.2 Performance Reports

To turn on performance reports, pass the -XCFA -S flag to the compiler. Three kinds of performance reports are available:

1. Time, reports time spent in each compilation step
2. Heap, reports number of dynamic memory allocations, total bytes allocated, and maximum heap memory usage
3. 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 optimization (g++ flag -O3).

A Appendix

A.1 Kinds of Type Parameters

A type parameter in a forall clause has 3 kinds:

1. dtype: any data type (built-in or user defined) that is not a concrete type.
   A non-concrete type is an incomplete type such as an opaque type or pointer/reference with an implicit (pointer) size and implicitly generated reference and dereference operations.
2. otype: any data type (built-in or user defined) that is concrete type.
   A concrete type is a complete type, *i.e.*, types that can be used to create a variable, which also implicitly asserts the existence of default and copy constructors, assignment, and destructor[5].
3. ttype: tuple (variadic) type.
   Restricted to the type for the last parameter in a function, it provides a type-safe way to implement variadic functions. Note however, that it has certain restrictions, as described in the implementation section below.

A.2 GNU C Nested Functions

C∀ is designed to be mostly compatible with GNU C, an extension to ISO C99 and C11 standards. The C∀ compiler also implements some language features by GCC extensions, most notably nested functions.

In ISO C, function definitions are not allowed to be nested. GCC allows nested functions with full lexical scoping. The following example is taken from GCC documentation[6]:

```
void bar( int * array, int offset, int size ) {
    int access( int * array, int index ) { return array[index + offset]; }
    int i;
    /* ... */
    for ( i = 0; i < size; i++ )
        /* ... */ access (array, i) /* ... */
}
```

GCC nested functions behave identically to C++ lambda functions with default by-reference capture (stack-allocated, lifetime ends upon exiting the declared block), while also possible to be passed as arguments with standard function pointer types.

A.3 Implementation of Parametric Functions

C∀ implements parametric functions using the implicit parameter approach: required assertions are passed to the callee by function pointers; size of a parametric type must also be known if referenced directly (*i.e.*, not as a pointer).

The implementation is similar to the one from Scala[7], with some notable differences in resolution:

1. All types, variables, and functions are candidates of implicit parameters
2. The parameter (assertion) name must match the actual declarations.

For example, the C∀ function declaration

[5] C∀ implements the same automatic resource management (RAII) semantics as C++.
[6] https://gcc.gnu.org/onlinedocs/gcc/Nested-Functions.html
[7] https://www.scala-lang.org/files/archive/spec/2.13/07-implicits.html

```
forall( otype T | { int foo( T, int ); } )
int bar(T);
```

after implicit parameter expansion, has the actual signature[8]

```
int bar( T, size_t, void (*)(T&), void (*)(T&), int (*)(T, int) );
```

The implicit parameter approach has an apparent issue: when the satisfying declaration is also parametric, it may require its own implicit parameters too. That also causes the supplied implicit parameter to have a different **actual** type than the **nominal** type, so it cannot be passed directly. Therefore, a wrapper with matching actual type must be created, and it is here where GCC nested functions are used internally by the compiler.

Consider the following program:

```
int assertion(int);

forall( otype T | { int assertion(T); } )
void foo(T);

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

The C∀ compiler translates the program to non-parametric form[9]

```
// ctor, dtor and size arguments are omitted
void foo(T, int (*)(T));

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

However, when bar(1) is called, foo cannot be directly provided as an argument:

```
bar(1, foo); // WRONG: foo has different actual type
```

and an additional step is required:

```
{
    void _foo_wrapper(int t) {
        foo( t, assertion );
    }
    bar( 1, _foo_wrapper );
}
```

Nested assertions and implicit parameter creation may continue indefinitely. This issue is a limitation of implicit parameter implementation. In particular, polymorphic variadic recursion must be structural (*i.e.*, the number of arguments decreases in any possible recursive calls), otherwise code generation gets into an infinite loop. The C∀ compiler sets a limit on assertion depth and reports an error if assertion resolution does not terminate within the limit (as for templates in C++).

[8] **otype** also requires the type to have constructor and destructor, which are the first two function pointers preceding the one for **foo**.

[9] In the final code output, T needs to be replaced by an opaque type, and arguments must be accessed by a frame pointer offset table, due to the unknown sizes. The presented code here is simplified for better understanding.

References

[1] Richard C. Bilson. Implementing overloading and polymorphism in C∀. Master's thesis, School of Computer Science, University of Waterloo, 2003. http://plg.uwaterloo.ca/theses/BilsonThesis.pdf. 1

[2] Glen Jeffrey Ditchfield. *Contextual Polymorphism*. PhD thesis, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, 1992. http://plg.uwaterloo.ca/theses/DitchfieldThesis.pdf. 1

[3] Aaron Moss. *C∀ Type System Implementation*. PhD thesis, School of Computer Science, University of Waterloo, 2019. https://uwspace.uwaterloo.ca/handle/10012/14584. 1, 7, 8