Optimization of C∀ Compiler with Case Studies

Fangren Yu – University of Waterloo

January 19, 2021

Abstract

C∀ is an evolutionary, non-object-oriented extension of the C programming language, featuring a parametric type-system, and is currently under active development. The reference compiler for the C∀ language, cfa-cc, has some of its major components dated back to the early 2000s, which are based on inefficient data structures and algorithms. This report introduces improvements targeting the expression resolution algorithm, suggested by a recent prototype experiment on a simplified model, which are implemented in cfa-cc to support the full C∀ language. These optimizations speed up the compiler by a factor of 20 across the existing C∀ codebase, bringing the compilation time of a mid-sized C∀ source file down to the 10-second level. A few problem cases derived from realistic code examples are analyzed in detail, with proposed solutions. This work is a critical step in the C∀ project development to achieve its eventual goal of being used alongside C for large software systems.

Acknowledgements

I would like to thank everyone in the C∀ team for their contribution towards this project. Programming language design and development is a tough subject and requires a lot of teamwork. Without the collaborative efforts from the team, this project could not have been a success. Specifically, I would like to thank Andrew Beach for introducing me to the C∀ codebase, Thierry Delisle for maintaining the test and build automation framework, Michael Brooks for providing example programs of various experimental language and type system features, and most importantly, Professor Martin Karsten for recommending me to the C∀ team, and my supervisor, Professor Peter Buhr for encouraging me to explore deeply into intricate compiler algorithms. Finally, I gratefully acknowledge the help from Aaron Moss, former graduate from the team and the author of the precedent thesis work, to participate in the C∀ team's virtual conferences and email correspondence, and provide many critical arguments and suggestions. 2020 had been an unusually challenging year for everyone and we managed to keep a steady pace.

Contents

1 Introduction **4**

2 AST restructuring **4**
- 2.1 Memory model with sharing . . . 4
- 2.2 Merged resolver calls . . . 5
- 2.3 Special function lookup . . . 6
- 2.4 Improvement of function type representation . . . 7
- 2.5 Improvement of pruning steps . . . 8
- 2.6 Shared sub-expression case: Unique expressions . . . 8

3 Analysis of resolver algorithm complexity **9**
- 3.1 Unbound return type . . . 10
- 3.2 ttype resolution infinite recursion . . . 10
- 3.3 Reused assertions in nested generic type . . . 12

4 Analysis of type system correctness **13**

5 Timing results **14**

6 Conclusion **15**

References **16**

1 Introduction

C∀ language, developed by the Programming Language Group at the University of Waterloo, has a long history, with the initial language design in 1992 by Glen Ditchfield [3] and the first proof-of-concept compiler built in 2003 by Richard Bilson [2]. Many new features have been added to the language over time, but the core of C∀'s type-system — parametric functions introduced by the forall clause (hence the name of the language) providing parametric overloading — remains mostly unchanged.

The current C∀ reference compiler, cfa-cc, is designed using the visitor pattern [8] over an abstract syntax tree (AST), where multiple passes over the AST modify it for subsequent passes. cfa-cc still includes many parts taken directly from the original Bilson implementation, which served as the starting point for this enhancement work to the type system. Unfortunately, the prior implementation did not provide the efficiency required for the language to be practical: a C∀ source file of approximately 1000 lines of code can take a multiple minutes to compile. The cause of the problem is that the old compiler used inefficient data structures and algorithms for expression resolution, which involved significant copying and redundant work.

This report presents a series of optimizations to the performance-critical parts of the resolver, with a major rework of the compiler data-structures using a functional-programming approach to reduce memory complexity. The improvements were suggested by running the compiler builds with a performance profiler against the C∀ standard-library source-code and a test suite to find the most underperforming components in the compiler algorithm.

The C∀ team endorses a pragmatic philosophy that focuses on practical implications of language design and implementation rather than theoretical limits. In particular, the compiler is designed to be expressive with respect to code reuse while maintaining type safety, but compromise theoretical soundness in extreme corner cases. However, when these corner cases do appear in actual usage, they need to be thoroughly investigated. A case-by-case analysis is presented for several of these corner cases, some of which point to certain weaknesses in the language design with solutions proposed based on experimental results.

2 AST restructuring

2.1 Memory model with sharing

A major rework of the AST data-structure in the compiler was completed as the first step of the project. The majority of this work is documented in my prior report documenting the compiler reference-manual [9]. To summarize:

- AST nodes (and therefore subtrees) can be shared without copying.
- Modifications are performed using functional-programming principles, making copies for local changes without affecting the original data shared by other owners. In-place mutations are permitted as a special case when there is no sharing. The logic is implemented by reference counting.
- Memory allocation and freeing are performed automatically using smart pointers [1].

The resolver algorithm, designed for overload resolution, uses a significant amount of reused, and hence copying, for the intermediate representations, especially in the following two places:

- Function overload candidates are computed by combining the argument candidates bottom-up, with many being a common term. For example, if n overloads of a function `f` all take an integer for the first parameter but different types for the second, *e.g.*, `f( int, int )`, `f( int, double )`, etc., the first term is copied n times for each of the generated candidate expressions. This copying is particularly bad for deep expression trees.
- In the unification algorithm and candidate elimination step, actual types are obtained by substituting the type parameters by their bindings. Let n be the complexity (*i.e.*, number of nodes in representation) of the original type, m be the complexity of the bound type for parameters, and k be the number of occurrences of type parameters in the original type. If every substitution needs to be deep-copied, these copy step takes $O(n+mk)$ time and memory, while using shared nodes it is reduced to $O(n)$ time and $O(k)$ memory.

One of the worst examples for the old compiler is a long chain of I/O operations:

```
sout | 1 | 2 | 3 | 4 | ...;   // print integer constants
```

The pipe operator is overloaded by the C∀ I/O library for every primitive type in the C language, as well as I/O manipulators defined by the library. In total, there are around 50 overloads for the output stream operation. On resolving the n-th pipe operator in the sequence, the first term, which is the result of sub-expression containing $n-1$ pipe operators, is reused to resolve every overload. Therefore at least $O(n^2)$ copies of expression nodes are made during resolution, not even counting type unification cost; combined with the two large factors from number of overloads of pipe operators, and that the "output stream type" in C∀ is a trait with 27 assertions (which adds to complexity of the pipe operator's type) this makes compiling a long output sequence extremely slow. In the new AST representation, only $O(n)$ copies are required and the type of the pipe operator is not copied at all. Reduction in space complexity is especially important, as preliminary profiling results on the old compiler build showed over half of the time spent in expression resolution is on memory allocations.

Since the compiler codebase is large and the new memory model mostly benefits expression resolution, some of the old data structures are still kept, and a conversion pass happens before and after the general resolve phase. Rewriting every compiler module will take longer, and whether the new model is correct was unknown when this project started, therefore only the resolver is currently implemented with the new data structure.

2.2 Merged resolver calls

The pre-resolve phase of compilation, inappropriately called "validate" in the compiler source code, has a number of passes that do more than simple syntax and semantic validation; some passes also normalizes the input program. A few of these passes require type information for expressions, and therefore, need to call the resolver before the general resolve phase. There are three notable places where the resolver is invoked:

- Generate default constructor, copy constructor and destructor for user-defined `struct` types.
- Resolve `with` statements (the same as in Pascal [5]), which introduces fields of a structure directly into a scope.
- Resolve `typeof` expressions (cf. `decltype` in C++); note that this step may depend on symbols introduced by `with` statements.

Since the constructor calls are one of the most expensive to resolve (reason given in Section 2.3), this pre-resolve phase was taking a large amount of time even after the resolver was changed to the more efficient new implementation. The problem is that multiple resolutions repeat a significant amount of work. Therefore, to better facilitate the new resolver, every step that requires type information should be integrated as part of the general resolver phase.

A by-product of this work is that reversed dependence between with statement and typeof can now be handled. Previously, the compiler was unable to handle cases such as:

```
struct S { int x; };
S foo();
typeof( foo() ) s; // type is S
with (s) {
    x; // refers to s.x
}
```

since the type of s is unresolved when handling with expressions because the with pass follows the typeof pass (interchanging passes only interchanges the problem). Instead, the new (and correct) approach is to evaluate typeof expressions when the declaration is first seen during resolution, and it suffices because of the declaration-before-use rule.

2.3 Special function lookup

Reducing the number of function looked ups for overload resolution is an effective way to gain performance when there are many overloads but most of them are trivially wrong. In practice, most functions have few (if any) overloads but there are notable exceptions. Most importantly, constructor ?{}, destructor ^?{}, and assignment ?=? are generated for every user-defined type (struct and union in C), and in a large source file there can be hundreds of them. Furthermore, many calls are generated for initializing variables, passing arguments and copying values. This fact makes them the most overloaded and most called functions.

In an object-oriented programming language, the object-method types are scoped within a class, so a call such as obj.f() only needs to perform lookup in the method table corresponding to the type of obj. C∀ on the other hand, does not have methods, and all types are open, *i.e.*, new operations can be defined on them without inheritance; at best a C∀ type can be constrained by a translation unit. However, the "big 3" operators have a unique property enforced by the language rules: the first parameter must be a reference to its associated type, which acts as the this parameter in an object-oriented language. Since C∀ does not have class inheritance, the reference type must always match exactly. Therefore, argument-dependent lookup can be implemented for these operators by using a dedicated, fast symbol-table.

The lookup key for the special functions is the mangled type name of the first parameter. To handle generic types, the type parameters are stripped off, and only the base type is matched. Note a constructor (destructor, assignment operator) may not take an arbitrary this argument, *e.g.*, forall(dtype T) void ?{}(T &), thus guaranteeing that if the this type is known, all possible overloads can be found by searching with this given type. In the case where the this argument itself is overloaded, it is resolved first and all possible result types are used for lookup.

Note that for a generated expression, the particular variable for the this argument is fully known, without overloads, so the majority of constructor-call resolutions only need to check for one given object type. Explicit constructor calls and assignment statements sometimes require lookup for multiple types. In the extremely rare case that the this-argument type is unbound, all necessary types are guaranteed to be checked, as for the previous lookup without the argument-dependent lookup; fortunately, this complex case almost never happens in practice. An example is found in the library function new:

```
forall( dtype T | sized( T ), ttype TT | { void ?{}( T &, TT ); } )
T * new( TT p ) { return &(*malloc()){ p }; }
```

as malloc may return a pointer to any type, depending on context.

Interestingly, this particular declaration actually causes another complicated issue, making the complex checking of every constructor even worse. Section 3.2, p. 10 presents a detailed analysis of this problem.

The "callable" operator ?() (cf. operator() in C++) can also be included in this special operator list, as it is usually only on user-defined types, and the restriction that the first argument must be a reference seems reasonable in this case.

2.4 Improvement of function type representation

Since substituting type parameters with their bound types is one fundamental operation in many parts of resolver algorithm (particularly unification and environment binding), making as few copies of type nodes as possible helps reducing memory complexity. Even with the new memory management model, allocation is still a significant factor of resolver performance. Conceptually, operations on type nodes of the AST should be performed in functional-programming style, treating the data structure as immutable and only copying when necessary. The in-place mutation is a mere optimization that does not change the logic for operations.

However, the model was broken for function types by an inappropriate design. Function types require special treatment due to the existence of assertions that constrain the types it supports. Specifically, it must be possible to distinguish two different kinds of type parameter usage:

```
forall( dtype T ) void foo( T * t ) {
    forall( dtype U ) void bar( T * t, U * u ) { ... }
}
```

Here, only U is a free parameter in the nested declaration of function bar, as T must be bound at the call site when resolving bar.

Moreover, the resolution algorithm also has to distinguish type bindings of multiple calls to the same function, *e.g.*:

```
forall( dtype T ) int foo( T x );
int i = foo( foo( 1.0 ) );
```

The inner call has binding (T: double) while the outer call has binding (T: int). Therefore a unique representation for the free parameters is required in each expression. This type binding was previously done by creating a copy of the parameter declarations inside the function type and fixing references afterwards. However, fixing references is an inherently deep operation that does not work well with the functional-programming style, as it forces eager evaluation on the entire syntax tree representing the function type.

The revised approach generates a unique ID value for each function call expression instance and represents an occurrence of a free-parameter type with a pair of generated ID and original parameter declaration, so references are unique and a shallow copy of the function type is possible.

Note that after the change, all declaration nodes in the syntax-tree representation now map one-to-one with the actual declarations in the program, and therefore are guaranteed to be unique. This property can potentially enable more optimizations, and some related ideas are presented at the end of Section 2.6.

2.5 Improvement of pruning steps

A minor improvement for candidate elimination is to skip the step on the function overloads and only check the results of function application. As function calls are usually by name (versus pointers to functions), the name resolution rule dictates that every function candidate necessarily has a different type; indirect function calls are rare, and when they do appear, there are even fewer cases with multiple interpretations, and these rarely match exactly in argument type. Since function types have a much more complex representation (with multiple parameters and assertions) than data types, checking equality on them also takes longer.

A brief test of this approach shows that the number of function overloads considered in expression resolution increases by an amount of less than 1 percent, while type comparisons in candidate elimination are reduced by more than half. This improvement is consistent over all C∀ source files in the test suite.

2.6 Shared sub-expression case: Unique expressions

Unique expression denotes an expression evaluated only once to prevent unwanted side effects. It is currently only a compiler artifact, generated for tuple-member expression of the form:

```
struct S { int a; int b; };
S s;
s.[a, b]; // tuple member expression, type is [int, int]
```

If the aggregate expression is function call, it cannot be evaluated multiple times:

```
S makeS();
makeS().[a, b]; // this should only generate a unique S
```

Before code generation, the above expression is internally represented as

```
[uniqueExpr{makeS()}.a, uniqueExpr{makeS()}.b];
```

and the unique expression is expanded to[1]

```
({
    if ( !_unique_var_evaluated ) {
        _unique_var_evaluated = true;
        _unique_var = makeS();
    }
    _unique_var;
})
```

at code generation, where _unique_var and _unique_var_evaluated are generated variables whose scope covers all appearances of the same expression. The conditional check ensures a single call to makeS() even though there are logically multiple calls because of the tuple field expansion.

[1]This uses gcc's statement expression syntax, where ({block}) evaluates to value of last expression in the block.

Note that although the unique expression is only used for tuple expansion now, it is a generally useful construction, and is seen in other programming languages, such as Scala's `lazy val` [7]; therefore it may be worthwhile to introduce the unique expression to a broader context in C∀ and even make it directly available to programmers.

In the compiler's visitor pattern, however, this creates a problem where multiple paths to a logically unique expression exist, so it may be modified more than once and become ill-formed; some specific intervention is required to ensure unique expressions are only visited once. Furthermore, a unique expression appearing in more than one places is copied on mutation so its representation is no longer unique.

Currently, special cases are required to keep everything synchronized, and the methods are different when mutating the unique expression instance itself or its underlying expression:

- When mutating the underlying expression (visit-once guard)

```
void InsertImplicitCalls::previsit( const ast::UniqueExpr * unqExpr ) {
    if ( visitedIds.count( unqExpr->id ) ) visit_children = false;
    else visitedIds.insert( unqExpr->id );
}
```

- When mutating the unique instance itself, which actually creates copies

```
auto mutExpr = mutate( unqExpr ); // internally calls copy when shared
if ( ! unqMap.count( unqExpr->id ) ) {
    ...
} else {
    mutExpr->expr = unqMap[mutExpr->id]->expr;
    mutExpr->result = mutExpr->expr->result;
}
```

Such workarounds are difficult to fit into the common visitor pattern, which suggests the memory model may need different kinds of nodes to accurately represent this feature in the AST.

Given that declaration nodes are unique, it is possible for AST nodes to be divided into three different types:

- **Singleton** with only one owner (declarations);
- **No-copy** with multiple owners but cannot be copied (unique expression example presented here);
- **Copy** by functional-programming style, which assumes immutable data structures that are copied on mutation.

The boilerplate code can potentially handle these three cases differently.

3 Analysis of resolver algorithm complexity

The focus of this section is to identify and analyze some realistic cases that cause the resolver algorithm to have an exponential runtime. As previous work has shown [6, § 4.2.1], the overload resolution problem in C∀ has worst-case exponential complexity; however, only few specific patterns can trigger the exponential complexity in practice. Implementing heuristic-based optimization for those selected cases is helpful to alleviate the problem.

3.1 Unbound return type

The interaction of return-type overloading and polymorphic functions creates function calls with unbounded return-type, and is further complicated by the presence of assertions. The prime example of a function with unbound return type is the type-safe version of C malloc:

```
forall( dtype T | sized( T ) )
T * malloc( void ) { return (T *)malloc( sizeof(T) ); } // call C malloc
int * i = malloc(); // type deduced from left-hand size ⇒ no size argument or return cast
```

An unbound return-type is problematic in resolver complexity because a single match of a function call with an unbound return type may create multiple candidates. In the worst case, consider a function declared that returns any otype (defined page 12):

```
forall( otype T ) T anyObj( void );
```

As the resolver attempts to satisfy the otype constraint on T, a call to anyObj() in an expression, without the result type known, creates at least as many candidates as the number of complete types currently in scope; with generic types it becomes even worse, *e.g.*, assuming a declaration of a generic pair is available at that point:

```
forall( otype T, otype U ) struct pair { T first; U second; };
```

Then an anyObj() call can result in arbitrarily complex types, such as pair(pair(int, int), pair(int, int)), and the depth can grow indefinitely until a specified parameter-depth limit, thus creating exponentially many candidates. However, the expected types allowed by parent expressions are practically very few, so most of those interpretations are invalid; if the result type is never bound up to the top level, by the semantic rules it is ambiguous if there is more than one valid binding and resolution fails quickly. It is therefore reasonable to delay resolving assertions on an unbound parameter in a return type; however, with the current cost model, such behavior may further cause irregularities in candidate selection, such that the presence of assertions can change the preferred candidate, even when order of expression costs are supposed to stay the same. A detailed analysis of this issue is presented in Section 4, p. 13.

3.2 ttype resolution infinite recursion

ttype (“tuple type”) is a relatively new addition to the language that attempts to provide type-safe variadic argument semantics. Unlike regular dtype parameters, ttype is only valid in a function parameter-list, and may only appear once as the last parameter type. At the call site, a ttype parameter is bound to the tuple type of all remaining function-call arguments.

There are two kinds of idiomatic ttype usage: one is to provide flexible argument forwarding, similar to the variadic template in C++ (**template**<**typename**... args>), as shown below in the implementation of unique_ptr

```
forall( dtype T )
struct unique_ptr {
    T * data;
};
forall( dtype T | sized( T ), ttype Args | { void ?{}( T &, Args ); })
void ?{}( unique_ptr( T ) & this, Args args ) {
    this.data = new( args ); // forward constructor arguments to dynamic allocator
}
```

The other usage is to implement structural recursion in the first-rest pattern:

```
forall( otype T, ttype Params | { void process( T ); void func( Params ); })
void func( T arg1, Params p ) {
    process( arg1 );
    func( p );  // recursive call until base case of one argument
}
```

For the second use case, it is imperative the number of parameters in the recursive call goes down, since the call site must deduce all assertion candidates, and that is only possible if by observation of the argument types (and not their values), the recursion is known to be completed in a finite number of steps.

In recent experiments, however, a flaw in the type-binding rules can lead to the first kind of ttype use case producing an invalid candidate and the resolver enters an infinite loop. This bug was discovered in an attempt to raise the assertion recursive-depth limit and one of the library programs took exponentially longer to compile. The cause of the problem is the following set of functions:

```
// unique_ptr  declaration from above

forall( dtype T | sized( T ), ttype Args | { void ?{}( T &, Args ); } ) { // distribute forall clause
    void ?{}( counter_data( T ) & this, Args args );
    void ?{}( counter_ptr( T ) & this, Args args );
    void ?{}( unique_ptr( T ) & this, Args args );
}

forall( dtype T | sized( T ), ttype TT | { void ?{}( T &, TT ); } )
T * new( TT p ) { return &(*malloc()){ p }; }
```

In the expression (*malloc()){p}, the type of the object being constructed is unknown, since the return-type information is not immediately available. That causes every constructor to be searched, and while normally a bound ttype cannot be unified with any free parameter, it is possible with another free ttype. Therefore, in addition to the correct option provided by the assertion, 3 wrong options are examined, each of which again requires the same assertion, for an unknown base-type T and ttype argument, which becomes an infinite loop until the specified recursion limit and resolution is fails. Moreover, during the recursion steps, the number of candidates grows exponentially, since there are always 3 options at each step.

Unfortunately, ttype to ttype binding is necessary, to allow indirectly calling a function provided in an assertion.

```
forall( dtype T | sized( T ), ttype Args | { void ?{}( T &, Args ); })
void ?{}( unique_ptr( T ) & this, Args args ) { this.data = (T *)new( args ); } // constructor call
```

Here the constructor assertion is used by the new(args) call to indirectly call the constructor on the allocated storage. Therefore, it is hard, perhaps impossible, to solve this problem by tweaking the type binding rules. An assertion caching algorithm can help improve this case by detecting cycles in recursion.

Meanwhile, without a caching algorithm implemented, some changes in the C∀ source code are enough to eliminate this problem, at least in the current codebase. Note that the issue only happens with an overloaded variadic function, which rarely appears in practice, since the idiomatic use cases are for argument forwarding and self-recursion. The only overloaded ttype function so far discovered in all of C∀ standard library is the constructor, and by utilizing the argument-dependent lookup process described in Section 3.1, p. 10, adding a cast before the constructor call removes the issue.

```
T * new( TT p ) { return &(*(T * )malloc()){ p }; }
```

3.3 Reused assertions in nested generic type

The following test of deeply nested, dynamic generic type reveals that locally caching reused assertions is necessary, rather than just a resolver optimization, because recomputing assertions can result in bloated generated code size:

```
struct nil {};
forall( otype L, otype R )
struct cons { L l; R r; };

int main() {
    #if  N==0
    nil x;
    #elif N==1
    cons( size_t, nil ) x;
    #elif N==2
    cons( size_t, cons( size_t, nil ) ) x;
    #elif N==3
    cons( size_t, cons( size_t, cons( size_t, nil ) ) ) x;
    // similarly for N=4,5,6
    #endif
}
```

At the declaration of x, it is implicitly initialized by generated constructor call, with signature:

```
forall( otype L, otype R ) void ?{}( cons( L, R ) & );
```

where the otype constraint contains the 4 assertions:

```
void ?{}( L & ); // default constructor
void ?{}( L &, L & ); // copy constructor
void ^?{}( L & ); // destructor
L & ?=?( L &, L & ); // assignment
```

Table 1: Compilation results of nested cons test

N	Assertions resolved	Binary size (KB)
3	170	70
4	682	219
5	2,730	829
6	10,922	3,261

Now since the right hand side of outermost cons is again a cons, recursive assertions are required. Table 1 shows when the compiler does not cache and reuse already resolved assertions, it becomes a problem, as each of these 4 pending assertions again asks for 4 more assertions one level below. Without caching, the number of resolved assertions grows exponentially, which is unnecessary since there are only $n+1$ different types involved. Even worse, this problem causes exponentially many wrapper functions to be generated at the backend, resulting in a huge binary. As the local functions are implemented by emitting executable code on the stack [4], it means that compiled code also has exponential run time. This problem has practical implications, as nested collection types are frequently used in real production code.

4 Analysis of type system correctness

In Moss' thesis [6, § 4.1.2, p. 45], the author presents the following example:

> The cost model of C∀ precludes a greedy bottom-up resolution pass, as constraints and costs introduced by calls higher in the expression tree can change the interpretation of those lower in the tree, as in the following example:
>
> ```
> void f( int );
> double g₁( int );
> int g₂( long );
> f( g( 42 ) );
> ```
>
> Considered independently, g_1(42) is the cheapest interpretation of g(42), with cost (0, 0, 0, 0, 0, 0, 0) since the argument type is an exact match. However, in context, an unsafe conversion is required to downcast the return type of g1 to an int suitable for f, for a total cost of (1, 0, 0, 0, 0, 0, 0) for f(g1(42)). If g2 is chosen, on the other hand, there is a safe upcast from the int type of 42 to long, but no cast on the return of g2, for a total cost of (0, 0, 1, 0, 0, 0, 0) for f(g2(42)); as this is cheaper, g2 is chosen. Due to this design, all valid interpretations of subexpressions must in general be propagated to the top ...

The cost model in the thesis sums up all costs in resolving sub-expressions and chooses the global optimal option. However, the current implementation in cfa-cc, based on the original Bilson model [2, § 2.2.4, p. 32], does eager resolution and only allows local optimal candidate selections to propagate upwards:

> From the set of candidates whose parameter and argument types have been unified and whose assertions have been satisfied, those whose sub-expression interpretations have the smallest total cost of conversion are selected ... The total cost of conversion for each of these candidates is then calculated based on the implicit conversions and polymorphism involved in adapting the types of the sub-expression interpretations to the formal parameter types.

With this model, the algorithm picks g1 in resolving the f(g(42)) call, which is undesirable.

There is further evidence that shows the Bilson model is fundamentally incorrect, following the discussion of unbound return type in Section 3.1, p. 10. By the conversion-cost specification, a binding from a polymorphic type-parameter to a concrete type incurs a polymorphic cost of 1. It remains unspecified *when* the type parameters should become bound. When the parameterized types appear in function parameters, they can be deduced from the argument type, and there is no ambiguity. In the unbound return case, however, the binding may happen at any stage in expression resolution, therefore it is impossible to define a unique local conversion cost. Note that type binding happens exactly once per parameter in resolving the entire expression, so the global binding cost is unambiguously 1.

In the current compiler implementation, there is a notable inconsistency in handling this case. For any unbound parameter that does *not* come with an associated assertion, it remains unbound to the parent expression; for those that do, however, they are immediately bound in the assertion resolution step, and concrete result types are used in the parent expressions. Consider the following example:

```
forall( dtype T ) T * f( void );
void h( int * );
```

The expression `h( f() )` eventually has a total cost of 1 from binding (T: int), but in the eager-resolution model, the cost of 1 may occur either at the call to `f` or at call to `h`, and with the assertion resolution triggering a binding, the local cost of `f()` is (0 poly, 0 spec) with no assertions, but (1 poly, -1 spec) with an assertion:

```
forall( dtype T | { void g( T * ); } ) T * f( void );
void g( int * );
void h( int * );
```

and that contradicts the principle that adding assertions should make expression cost lower. Furthermore, the time at which type binding and assertion resolution happens is an implementation detail of the compiler, not part of the language definition. That means two compliant C∀ compilers, one performing immediate assertion resolution at each step, and one delaying assertion resolution on unbound types, can produce different expression costs and therefore different candidate selection, making the language rule itself partially undefined, and therefore, unsound. By the above reasoning, the updated cost model using global sum of costs should be accepted as the standard. It also allows the compiler to freely choose when to resolve assertions, as the sum of total costs is independent of that choice; more optimizations regarding assertion resolution can also be implemented.

5 Timing results

For the timing results presented here, the C∀ compiler is built with gcc 9.3.0, and tested on a server machine running Ubuntu 20.04, 64GB RAM and 32-core 2.2 GHz CPU. Timing is reported by the `time` command and an experiment is run using 8 cores, where each core is at 100% CPU utilization.

On the most recent build, the C∀ standard library (≈1.3 MB of source code) compiles in 4 minutes 47 seconds total processor time (single thread equivalent), with the slowest file taking 13 seconds. The test suite (178 test cases, ≈2.2MB of source code) completes within 25 minutes total processor time, with the slowest file taking 23 seconds. In contrast, the library build with the old compiler takes 85 minutes total, 5 minutes for the slowest file. The full test-suite takes too long with old compiler build and is therefore not run, but the slowest test cases take approximately 5 minutes. Overall, the most recent build compared to an old build is consistently faster by a factor of 20.

Additionally, 6 selected C∀ source files with distinct features from the library and test suite are used to illustrate the compiler performance change after each of the implemented optimizations. Test files are from the most recent build and run through the C preprocessor to expand header file, perform macro expansions, but no line number information (`gcc -E -P`). Table 2 shows the selected tests:

- lib/fstream (112 KB)
- lib/mutex (166 KB): implementation of concurrency primitive
- lib/vector (43 KB): container example, similar to C++ vector
- lib/stdlib (64 KB): type-safe wrapper to void *-based C standard library functions
- test/io2 (55 KB): application of I/O library
- test/thread (188 KB): application of threading library

versus C∀ compiler builds picked from the git commit history that implement the optimizations incrementally:

- old resolver
- #0 is the first working build of the new AST data structure
- #1 implements special symbol table and argument-dependent lookup
- #2 implements late assertion-satisfaction
- #3 implements revised function-type representation
- #4 skips pruning on expressions for function types (most recent build)

Reading left to right for a test shows the benefit of each optimization on the cost of compilation.

Table 2: Compile time of selected files by compiler build, in seconds

Test case	old	#0	#1	#2	#3	#4
lib/fstream	86.4	25.2	10.8	9.5	7.8	7.1
lib/mutex	210.4	77.4	16.7	15.1	12.6	11.7
lib/vector	17.2	8.9	3.1	2.8	2.4	2.2
lib/stdlib	16.6	8.3	3.2	2.9	2.6	2.4
test/io2	300.8	53.6	43.2	27.9	19.1	16.3
test/thread	210.9	73.5	17.0	15.1	12.6	11.8

Results are average of 5 runs (3 runs if time exceeds 100 seconds)

6 Conclusion

Over the course of 8 months of active research and development of the C∀ type system and compiler algorithms, performance of the reference C∀ compiler, cfa-cc, has been greatly improved. Now, mid-sized C∀ programs are compiled reasonably fast. Currently, there are ongoing efforts by the C∀ team to augment the standard library and evaluate its runtime performance, and incorporate C∀ with existing software written in C; therefore this project is especially meaningful for these practical purposes.

Accomplishing this work was difficult. Analysis conducted in the project is based significantly on heuristics and practical evidence, as the theoretical bounds and average cases for the expression resolution problem differ. As well, the slowness of the initial compiler made attempts to understand why and where problems exist extremely difficult because both debugging and validation tools (*e.g.*, gdb, valgrind, pref) further slowed down compilation time. However, by the end of the project, I had found and fixed several significant problems and new optimizations are easier to introduce and test. The reduction in the development cycle benefits the C∀ team as a whole.

Some potential issues of the language, which happen frequently in practice, have been identified. Due to the time constraint and complex nature of these problems, a handful of them remain unsolved, but some constructive proposals are made. Notably, introducing a local assertion cache in the resolver is a reasonable solution for a few remaining problems, so that should be the focus of future work.

The C∀ team are planning on a public alpha release of the language as the compiler performance, given my recent improvements, is now useable. Other parts of the system, such as the standard library, have made significant gains due to the speed up in the development cycle. Ideally, the remaining problems should be resolved before release, and the solutions will also be integral to drafting a formal specification.

References

[1] Andrei Alexandrescu. Smart pointers. In *Modern C++ Design: Generic Programming and Design Patterns Applied*, chapter 7. Addison-Wesley, 2001. 4

[2] 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. 4, 13

[3] 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. 4

[4] gcc 9.3 Manual. *Nested Functions*, 2019. https://gcc.gnu.org/onlinedocs/gcc-9.3.0/gcc/-Nested-Functions.html. 13

[5] Kathleen Jensen and Niklaus Wirth. *Pascal User Manual and Report, ISO Pascal Standard*. Springer–Verlag, 4th edition, 1991. Revised by Andrew B. Mickel and James F. Miner. 5

[6] Aaron Moss. *C∀ Type System Implementation*. PhD thesis, School of Computer Science, University of Waterloo, 2019. https://uwspace.uwaterloo.ca/handle/10012/14584. 9, 13

[7] *Scala Language Specification, Version 2.11*. École Polytechnique Fédérale de Lausanne, 2016. http://www.scala-lang.org/files/archive/spec/2.11. 9

[8] vistor pattern. https://en.wikipedia.org/wiki/Visitor_pattern, 2020. WikipediA. 4

[9] Fangren Yu. cfa-cc developer's reference manual. Technical report, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, August 2020. https://-cforall.uwaterloo.ca/doc/Fangren_Yu_Report_S20.pdf. 4