C∀ Type System Implementation

by

Aaron Moss

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2019

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Doug Lea Professor, Computer Science Department, State University of New York at Oswego
Supervisor:	Peter Buhr Associate Professor, School of Computer Science, University of Waterloo
Internal Members:	Ondřej Lhoták Associate Professor, School of Computer Science, University of Waterloo
	Gregor Richards Assistant Professor, School of Computer Science, University of Waterloo
Internal-External Member:	Werner Dietl Assistant Professor, Electrical and Computer Engineering, University of Waterloo

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The C programming language has been an important software development tool for decades. C∀ is a new programming language designed with strong backwards-compatibility to take advantage of widely distributed C programming expertise and the large deployed base of C code, paired with modern language features to improve developer productivity.

This thesis presents a number of improvements to C∀. The author has developed one major new language feature, generic types, in a way that integrates naturally with both the existing polymorphism features of C∀ and the translation-unit-based encapsulation model of C. This thesis also presents a number of smaller refinements to the C∀ overload resolution rules, each of which improves the expressivity or intuitive nature of the language.

This thesis also includes a number of practical improvements to C∀ compilation performance, focused on the expression resolution pass, which is the main bottleneck. These include better algorithms for argument-parameter matching and type assertion satisfaction, as well as a new type-environment data-structure based on a novel variant of union-find. The compilation performance improvements have all been experimentally validated with a new prototype system that encapsulates the key aspects of the C∀ language; this prototype is a promising basis for future research and a technical contribution of this work.

C∀, extended and refined in this thesis, presents both an independently interesting combination of language features and a comprehensive approach to the modernization of C. This work demonstrates the hitherto unproven compiler-implementation viability of the C∀ language design, and provides a number of useful tools to implementors of other languages.

Acknowledgements

Though a doctoral thesis is an individual project, I could not have completed it without the help and support of many members of my community. This thesis would not exist in the form it does without the mentorship of my advisor, Peter Buhr, who has ably led the C∀ team while giving me both the advantage of his decades of experience and the freedom to follow my own interests.

My work on C∀ does not exist in a vacuum, and it has been a pleasure and a privilege to collaborate with the members of the C∀ team: Andrew Beach, Richard Bilson, Michael Brooks, Bryan Chan, Thierry Delisle, Glen Ditchfield, Brice Dobry, Rob Schluntz, and others. I gratefully acknowledge the financial support of the National Science and Engineering Council of Canada and Huawei Ltd. for this project. I would also like to thank of my thesis committee, Werner Dietl, Doug Lea, Ondřej Lhoták, and Gregor Richards, for the time and effort they have invested in providing constructive feedback to refine this work. I am indebted to Peter van Beek and Ian Munro for their algorithmic expertise and willingness to share their time with me. I have far too many colleagues in the Programming Languages Group and School of Computer Science to name, but I deeply appreciate their camaraderie; specifically with regard to the production of this thesis, I would like to thank Nathan Fish for recommending my writing soundtrack, and Sharon Choy for her unfailing supply of encouraging rabbit animations.

Finally, to all my friends and family who have supported me and made Kitchener-Waterloo home these past seven years, thank you, I could not have done it without you; most especially, Christina Moss, you are the best of wives and best of women, your support has kept me going through the ups and downs of research, and your partnership is key to all my successes.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The C programming language [9] has had a wide-ranging impact on the design of software and programming languages. In the 30 years since its first standardization, it has consistently been one of the most popular programming languages, with billions of lines of C code still in active use, and tens of thousands of trained programmers producing it. The TIOBE index [46] tracks popularity of programming languages over time, and C has never dropped below second place:

Table 1.1: Current top 5 places in the TIOBE index averaged over years

	2018	2013	2008	2003	1998	1993	1988
Java	1	2	1	1	18	–	–
C	**2**	**1**	**2**	**2**	**1**	**1**	**1**
C++	3	4	3	3	2	2	5
Python	4	7	6	11	22	17	–
C♯	5	5	7	8	–	–	–

The impact of C on programming language design is also obvious from Table 1.1; with the exception of Python, all of the top five languages use C-like syntax and control structures. C++ [15] is even a largely backwards-compatible extension of C. Though its lasting popularity and wide impact on programming language design point to the continued relevance of C, there is also widespread desire of programmers for languages with more expressive power and programmer-friendly features; accommodating both maintenance of legacy C code and development of the software of the future is a difficult task for a single programming language.

C∀[A] is an evolutionary modernization of the C programming language that aims to fulfill both these ends well. C∀ both fixes existing design problems and adds multiple new features to C, including name overloading, user-defined operators, parametric-polymorphic routines, and type constructors and destructors, among others. The new features make C∀ more powerful and expressive than C, while maintaining strong backward-compatibility with both C code and the procedural paradigm expected by C programmers. Unlike other popular C extensions like C++ and Objective-C, C∀ adds modern features to C without imposing an object-oriented paradigm to use them. However, these new features do impose a compile-time cost, particularly in the expression resolver, which must evaluate the typing rules of a significantly more complex type system.

This thesis is focused on making C∀ a more powerful and expressive language, both by adding new features to the C∀ type system and ensuring that both added and existing features can be efficiently implemented in `cfa-cc`, the C∀ reference compiler. Particular contributions of this work include:

- design and implementation of parametric-polymorphic ("generic") types in a manner compatible with the existing polymorphism design of C∀ (Chapter 3),
- a new expression resolution algorithm designed to quickly locate the optimal declarations for a C∀ expression (Chapter 4),
- a type environment data structure based on a novel variant of the union-find algorithm (Chapter 5),
- and as a technical contribution, a prototype system for compiler algorithm development which encapsulates the essential aspects of the C∀ type system without incurring the technical debt of the existing system or the friction-inducing necessity of maintaining a working compiler (Chapter 6).

The prototype system, which implements the algorithmic contributions of this thesis, is the first performant type-checker implementation for a C∀-style type system. As the existence of an efficient compiler is necessary for the practical viability of a programming language, the contributions of this thesis comprise a validation of the C∀ language design that was previously lacking.

Though the direction and experimental validation of this work is fairly narrowly focused on the C∀ programming language, the tools used and results obtained should be of interest

[A] Pronounced "C-for-all", and written C∀ or Cforall.

to a wider compiler and programming language design community. In particular, with the addition of *concepts* in C++20 [10], conforming C++ compilers must support a model of type assertions very similar to that in CFA, and the algorithmic techniques used here may prove useful. Much of the difficulty of type-checking CFA stems from the language design choice to allow overload selection from the context of a function call based on function return type in addition to the type of the arguments to the call; this feature allows the programmer to specify fewer redundant type annotations on functions that are polymorphic in their return type. As an example in C++:

```
template<typename T> T* zero() { return new T( 0 ); }

int* z = zero<int>();                    // must specify int twice
```

CFA allows `int* z = zero()`, which elides the second **int**. While the **auto** keyword in C++11 supports similar inference in a limited set of contexts (*e.g.* `auto z = zero<int>()`), the demonstration of the richer inference in CFA raises possibilities for similar features in future versions of C++. By contrast to C++, Java 8 [19] and Scala [42] use comparably powerful forms of type inference to CFA, so the algorithmic techniques in this thesis may be effective for those languages' compiler implementors. Type environments are also widely modelled in compiler implementations, particularly for functional languages, though also increasingly commonly for other languages (such as Rust [41]) that perform type inference; the type environment presented here may be useful to those language implementors.

One area of inquiry that is outside the scope of this thesis is formalization of the CFA type system. Ditchfield [14] defined the $F_\omega^\ni$ polymorphic lambda calculus, which is the theoretical basis for the CFA type system. Ditchfield did not, however, prove any soundness or completeness properties for $F_\omega^\ni$; such proofs remain future work. A number of formalisms other than $F_\omega^\ni$ could potentially be adapted to model CFA. One promising candidate is *local type inference* [38, 34], which describes similar contextual propagation of type information; another is the polymorphic conformity-based model of the Emerald [7] programming language, which defines a subtyping relation on types that is conceptually similar to CFA traits. These modelling approaches could potentially be used to extend an existing formal semantics for C such as Cholera [31], CompCert [25], or Formalin [24].

Chapter 2

C∀

C∀ adds a number of features to C, some of them providing significant increases to the expressive power of the language, but all designed to maintain the existing procedural programming paradigm of C and to be as orthogonal as possible to each other. To provide background for the contributions in subsequent chapters, this chapter provides a summary of the features of C∀ at the time this work was conducted.

Glen Ditchfield laid out the core design of C∀ in his 1992 PhD thesis, *Contextual Polymorphism* [14]; in that thesis, Ditchfield presents the theoretical underpinnings of the C∀ polymorphism model. Building on Ditchfield's design for contextual polymorphism as well as KW-C [8], an earlier set of (largely syntactic) extensions to C, Richard Bilson [6] built the first version of the C∀ compiler, `cfa-cc`, in the early 2000's. This early `cfa-cc` provided basic functionality, but incorporated a number of algorithmic choices that have failed to scale as C∀ has developed, lacking the runtime performance for practical use; this thesis is substantially concerned with rectifying those deficits.

The C∀ project was revived in 2015 with the intention of building a production-ready language and compiler; at the time of this writing, both C∀ and `cfa-cc` remain under active development. As this development has been proceeding concurrently with the work described in this thesis, the state of C∀ has been somewhat of a moving target; however, Moss *et al.* [30] provides a reasonable summary of the current design. Notable features added during this period include generic types (Chapter 3), constructors and destructors [43], improved support for tuples [43], reference types [30], first-class concurrent and parallel programming support [13], as well as numerous pieces of syntactic sugar and the start of an idiomatic standard library [30].

This thesis is primarily concerned with the *expression resolution* portion of C∀ type-

checking; resolution is discussed in more detail in Chapter 4, but is essentially determining which declarations the identifiers in each expression correspond to. In C, no simultaneously-visible declarations share identifiers, hence expression resolution in C is not difficult. In CFA, multiple added features make the resolution process significantly more complex. Due to this complexity, the expression-resolution pass in `cfa-cc` requires 95% of compiler runtime on some source files, making a new, more efficient procedure for expression resolution a requirement for a performant CFA compiler.

The features presented in this chapter are chosen to elucidate the design constraints of the work presented in this thesis. In some cases the interactions of multiple features make this design a significantly more complex problem than any individual feature; in other cases a feature that does not by itself add any complexity to expression resolution triggers previously rare edge cases more frequently.

2.1 Procedural Paradigm

It is important to note that CFA is not an object-oriented language. This is a deliberate choice intended to maintain the applicability of the programming model and language idioms already possessed by C programmers. This choice is in marked contrast to C++, which is a much larger and more complex language, and requires extensive developer re-training to write idiomatic, efficient code in C++'s object-oriented paradigm.

Particularly, CFA has no concept of *subclass*, and thus no need to integrate an inheritance-based form of polymorphism with its parametric and overloading-based polymorphism. While CFA does have a system of implicit type conversions derived from C's "usual arithmetic conversions" [9, §6.3.1.8] and these conversions may be thought of as something like an inheritance hierarchy, the underlying semantics are significantly different and such an analogy is loose at best. The graph structure of the CFA type conversions (discussed in Section 4.1.2) is also markedly different than an inheritance hierarchy; it has neither a top nor a bottom type, and does not satisfy the lattice properties typical of inheritance hierarchies.

Another aspect of CFA's procedural paradigm is that it retains C's translation-unit-based encapsulation model, rather than class-based encapsulation such as in C++. As such, any language feature that requires code to be exposed in header files (*e.g.* C++ templates) also eliminates encapsulation in CFA. Given this constraint, CFA is carefully designed to allow separate compilation for its added language features under the existing C usage patterns.

2.2 Name Overloading

In C, no more than one variable or function in the same scope may share the same name[A], and variable or function declarations in inner scopes with the same name as a declaration in an outer scope hide the outer declaration. This restriction makes finding the proper declaration to match to a variable expression or function application a simple matter of lexically-scoped name lookup, which can be easily and efficiently implemented. C∀, on the other hand, allows overloading of variable and function names so long as the overloaded declarations do not have the same type, avoiding the multiplication of variable and function names for different types common in the C standard library, as in the following example:

```
#include <limits.h>

const int max = INT_MAX;                                    // (1)
const double max = DBL_MAX;                                 // (2)
int max(int a, int b) { return a < b ? b : a; }             // (3)
double max(double a, double b) { return a < b ? b : a; }    // (4)

max( 7, -max );                 // uses (3) and (1), by matching int from 7
max( max, 3.14 );               // uses (4) and (2), by matching double from 3.14
max( max, -max );               // ERROR, ambiguous
int m = max( max, -max );       // uses (3) and (1) twice, by matching return type
```

The final expression in the preceding example includes a feature of C∀ name overloading not shared by C++, the ability to disambiguate expressions based on their return type. This provides greater flexibility and power than the parameter-based overload selection of C++, though at the cost of greater complexity in the resolution algorithm.

While the wisdom of giving both the maximum value of a type and the function to take the maximum of two values the same name in the example above is debatable, *e.g.* some developers may prefer MAX for the former, the guiding philosophy of C∀ is "describe, don't prescribe" — we prefer to trust programmers with powerful tools, and there is no technical reason to restrict overloading between variables and functions. However, the expressivity of C∀'s name overloading does have the consequence that simple table lookup is insufficient to match identifiers to declarations, and a type-matching algorithm must be part of expression resolution.

[A] Technically, C has multiple separated namespaces, one holding **struct**, **union**, and **enum** tags, one holding labels, one holding **typedef** names, variable, function, and enumerator identifiers, and one for each **struct** and **union** type holding the field names [9, §6.2.3].

2.2.1 Operator Overloading

C does allow name overloading in one context: operator overloading. From the perspective of the type system, there is nothing special about operators as opposed to other functions, nor is it desirable to restrict the clear and readable syntax of operators to only the built-in types. For these reasons, C∀, like C++ and many other programming languages, allows overloading of operators by writing specially-named functions where ? stands in for the operands. This syntax is more natural than the operator overloading syntax of C++, which requires "dummy" parameters to disambiguate overloads of similarly-named pre- and postfix operators[B]:

```
struct counter { int x; };

counter& ++?(counter& c) { ++c.x; return c; }          // pre-increment
counter ?++(counter& c) {                              // post-increment
    counter tmp = c; ++c; return tmp;
}
bool ?<?(const counter& a, const counter& b) {         // comparison
    return a.x < b.x;
}
```

Together, C∀'s backward-compatibility with C and the inclusion of this operator overloading feature imply that C∀ must select among function overloads using a method compatible with C's "usual arithmetic conversions" [9, §6.3.1.8], so as to present user programmers with only a single set of overloading rules.

2.2.2 Special Literal Types

Literal 0 is also used polymorphically in C; it may be either integer zero or the null value of any pointer type. C∀ provides a special type for the 0 literal, **zero_t**, so that users can define a zero value for their own types without being forced to create a conversion from an integer or pointer type; C∀ also includes implicit conversions from **zero_t** to the **int** and pointer type constructors[C] from **zero_t** for backward compatibility.

According to the C standard [9, §6.8.4.1], 0 is the only false value; any value that compares equal to zero is false, while any value that does not is true. By this rule, Boolean contexts such as **if** (x) can always be equivalently rewritten as **if** ((x) != 0). `cfa-cc`

[B] This example uses C∀'s reference types, described in Section 2.5

[C] See Section 2.5

applies this rewriting in all Boolean contexts, so any type T can be made "truthy" (that is, given a Boolean interpretation) in C∀ by defining an operator overload **int** ?!=?(T, **zero_t**). C++ takes a different approach to user-defined truthy types, allowing definition of an implicit conversion operator to **bool**; prior to the introduction of the explicit keyword for conversion operators in C++11 this approach also allowed undesired implicit conversions to all other arithmetic types, a shortcoming not shared by the C∀ design.

C∀ also includes a special type for 1, **one_t**; like **zero_t**, **one_t** has built-in implicit conversions to the various integral types so that 1 maintains its expected semantics in legacy code. The addition of **one_t** allows generic algorithms to handle the unit value uniformly for types where it is meaningful; a simple example of this is that polymorphic functions[D] in the C∀ prelude define ++x and x++ in terms of x += 1, allowing users to idiomatically define all forms of increment for a type T by defining the single function T& ?+=?(T&, **one_t**); analogous overloads for the decrement operators are also present, and programmers can override any of these functions for a particular type if desired.

C∀ previously allowed 0 and 1 to be the names of polymorphic variables, with separate overloads for **int** 0, **int** 1, and the polymorphic variable **forall**(**dtype** T) T* 0. While designing C∀ generic types (see Chapter 3), it was discovered that the parametric polymorphic zero variable is not generalizable to other types; though all null pointers have the same in-memory representation, the same cannot be said of the zero values of arbitrary types. As such, polymorphic variables, and in particular variables for 0 and 1, were phased out in favour of functions that could generate those values for a given type as appropriate.

2.3 Polymorphic Functions

The most significant type-system feature C∀ adds is parametric-polymorphic functions. Such functions are written using a **forall** clause (which gives the language its name):

```
forall(otype T)
T identity(T x) { return x; }
```

The identity function above can be applied to any complete object type (or "**otype**"). The type variable T is transformed into a set of additional implicit parameters to identity, which encode sufficient information about T to create and return a variable of that type. C∀ passes the size and alignment of the type represented by an **otype** parameter, as well as a default constructor, copy constructor, assignment operator, and destructor. Types that

[D] discussed in Section 2.3

do not have one or more of these operators visible cannot be bound to **otype** parameters, but may be bound to un-constrained **dtype** ("data type") type variables. In this design, the runtime cost of polymorphism is spread over each polymorphic call, due to passing more arguments to polymorphic functions; the experiments in Chapter 3 indicate that this overhead is comparable to that of C++ virtual function calls.

One benefit of this design is that it allows polymorphic functions to be separately compiled. The forward declaration **forall**(**otype** T) T identity(T); uniquely defines a single callable function, which may be implemented in a different file. The fact that there is only one implementation of each polymorphic function also reduces compile times relative to the template-expansion approach taken by C++, as well as reducing binary sizes and runtime pressure on instruction cache by re-using a single version of each function.

2.3.1 Type Assertions

Since bare polymorphic types do not provide a great range of available operations, C∀ provides a *type assertion* mechanism to provide further information about a type[E]:

```
forall(otype T | { T twice(T); })
T four_times(T x) { return twice( twice(x) ); }
double twice1(double d) { return d * 2.0; }

double ans = four_times( 10.5 );                // T bound to double, ans == 42.0
```

These type assertions may be either variable or function declarations that depend on a polymorphic type variable. four_times may only be called with an argument for which there exists a function named twice that can take that argument and return another value of the same type; a pointer to the appropriate function is passed as an additional implicit parameter of the call to four_times.

Monomorphic specializations of polymorphic functions can themselves be used to satisfy type assertions. For instance, twice could have been defined like this:

```
forall(otype S | { S ?+?(S, S); })
S twice2(S x) { return x + x; }
```

Specializing this polymorphic function with S = **double** produces a monomorphic function which can be used to satisfy the type assertion on four_times. `cfa-cc` accomplishes this by creating a wrapper function calling twice_2 with S bound to **double**, then providing

[E]This example introduces a convention used throughout this thesis of disambiguating overloaded names with subscripts; the subscripts do not appear in C∀ source code.

this wrapper function to four_times[F]. However, twice$_2$ also works for any type S that has an addition operator defined for it.

Finding appropriate functions to satisfy type assertions is essentially a recursive case of expression resolution, as it takes a name (that of the type assertion) and attempts to match it to a suitable declaration in the current scope[G]. If a polymorphic function can be used to satisfy one of its own type assertions, this recursion may not terminate, as it is possible that that function is examined as a candidate for its own assertion unboundedly repeatedly. To avoid such infinite loops, `cfa-cc` imposes a fixed limit on the possible depth of recursion, similar to that employed by most C++ compilers for template expansion; this restriction means that there are some otherwise semantically well-typed expressions that cannot be resolved by `cfa-cc`.

2.3.2 Traits

C∀ provides *traits* as a means to name a group of type assertions, as in the example below[H]:

```
trait has_magnitude(otype T) {
    bool ?<?(T, T);          // comparison operator
    T -?(T);                 // negation operator
    void ?{}(T&, zero_t);    // constructor from 0
};

forall(otype M | has_magnitude(M))
M abs(M m) { return m < (M){0} ? -m : m; }

forall(otype M | has_magnitude(M))
M max_magnitude(M a, M b) { return abs(a) < abs(b) ? b : a; }
```

Semantically, traits are simply a named list of type assertions, but they may be used for many of the same purposes that interfaces in Java or abstract base classes in C++ are used for. Unlike Java interfaces or C++ base classes, C∀ types do not explicitly state any inheritance relationship to traits they satisfy; this can be considered a form of structural inheritance, similar to interface implementation in Go, as opposed to the nominal inheritance model of Java and C++. Nominal inheritance can be simulated in C∀ using marker

[F] twice$_2$ could also have had a type parameter named T; C∀ specifies renaming of the type parameters, which would avoid the name conflict with the type variable T of four_times

[G] `cfa-cc` actually performs a type-unification computation for assertion satisfaction rather than an expression resolution computation; see Section 4.2.3 for details.

[H] This example uses C∀'s reference types and constructors, described in Section 2.5.

variables in traits:

```
trait nominal(otype T) {
    T is_nominal;
};

int is_nominal;                         // int now satisfies nominal
{
    char is_nominal;                    // char only satisfies nominal in this scope
}
```

Traits, however, are significantly more powerful than nominal-inheritance interfaces; firstly, due to the scoping rules of the declarations that satisfy a trait's type assertions, a type may not satisfy a trait everywhere that that type is declared, as with **char** and the nominal trait above. Secondly, because C∀ is not object-oriented and types do not have a closed set of methods, existing C library types can be extended to implement a trait simply by writing the requisite functions[I]. Finally, traits may be used to declare a relationship among multiple types, a property that may be difficult or impossible to represent in nominal-inheritance type systems[J]:

```
trait pointer_like(otype Ptr, otype El) {
    El& *?(Ptr);                        // Ptr can be dereferenced to El
};

struct list {
    int value;
    list* next;                         // may omit struct on type names
};

typedef list* list_iterator;

int& *?(list_iterator it) {
    return it->value;
}
```

In this example above, (list_iterator, **int**) satisfies pointer_like by the user-defined dereference function, and (list_iterator, list) also satisfies pointer_like by the built-in dereference operator for pointers. Given a declaration list_iterator it, *it can be either an **int** or a list,

[I] C++ only allows partial extension of C types, because constructors, destructors, conversions, and the assignment, indexing, and function-call operators may only be defined in a class; C∀ does not have any of these restrictions.

[J] This example uses C∀'s reference types, described in Section 2.5.

with the meaning disambiguated by context (*e.g.* **int** x = *it; interprets *it as **int**, while (*it).value = 42; interprets *it as list). While a nominal-inheritance system with associated types could model one of those two relationships by making El an associated type of Ptr in the pointer_like implementation, I am unaware of any nominal-inheritance system that can model both relationships simultaneously. Further comparison of C∀ polymorphism with other languages can be found in Section 3.1.1.

The flexibility of C∀'s implicit trait-satisfaction mechanism provides programmers with a great deal of power, but also blocks some optimization approaches for expression resolution. The ability of types to begin or cease to satisfy traits when declarations go into or out of scope makes caching of trait satisfaction judgments difficult, and the ability of traits to take multiple type parameters can lead to a combinatorial explosion of work in any attempt to pre-compute trait satisfaction relationships.

2.3.3 Deleted Declarations

Particular type combinations can be exempted from matching a given polymorphic function through the use of a *deleted function declaration*:

```
int somefn(char) = void;
```

This feature is based on a C++11 feature typically used to make a type non-copyable by deleting its copy constructor and assignment operator[K] or forbidding some interpretations of a polymorphic function by specifically deleting the forbidden overloads[L]. Deleted function declarations are implemented in `cfa-cc` by adding them to the symbol table as usual, but with a flag set that indicates that the function is deleted. If this deleted declaration is selected as the unique minimal-cost interpretation of an expression then an error is produced, allowing C∀ programmers to guide the expression resolver away from undesirable solutions.

[K] In previous versions of C++, a type could be made non-copyable by declaring a private copy constructor and assignment operator, but not defining either. This idiom is well-known, but depends on some rather subtle and C++-specific rules about private members and implicitly-generated functions; the deleted function form is both clearer and less verbose.

[L] Specific polymorphic function overloads can also be forbidden in previous C++ versions through use of template metaprogramming techniques, though this advanced usage is beyond the skills of many programmers.

2.4 Implicit Conversions

In addition to the multiple interpretations of an expression produced by name overloading and polymorphic functions, C∀ must support all of the implicit conversions present in C for backward compatibility, producing further candidate interpretations for expressions. As mentioned above, C does not have an inheritance hierarchy of types, but the C standard's rules for the "usual arithmetic conversions" [9, §6.3.1.8] define which of the built-in types are implicitly convertible to which other types, as well as which implicit conversions to apply for mixed arguments to binary operators. C∀ adds rules to the usual arithmetic conversions defining the cost of binding a polymorphic type variable in a function call; such bindings are cheaper than any *unsafe* (narrowing) conversion, *e.g.* **int** to **char**, but more expensive than any *safe* (widening) conversion, *e.g.* **int** to **double**. One contribution of this thesis, discussed in Section 4.1.2, is a number of refinements to this cost model to more efficiently resolve polymorphic function calls.

In the context of these implicit conversions, the expression resolution problem can be restated as finding the unique minimal-cost interpretation of each expression in the program, where all identifiers must be matched to a declaration, and implicit conversions or polymorphic bindings of the result of an expression may increase the cost of the expression. While semantically valid C∀ code always has such a unique minimal-cost interpretation, `cfa-cc` must also be able to detect and report as errors expressions that have either no interpretation or multiple ambiguous minimal-cost interpretations. Note that which subexpression interpretation is minimal-cost may require contextual information to disambiguate. For instance, in the example in Section 2.2, max(max, –max) cannot be unambiguously resolved, but **int** m = max(max, –max) has a single minimal-cost resolution. While the interpretation **int** m = (**int**)max((**double**)max, –(**double**)max) is also a valid interpretation, it is not minimal-cost due to the unsafe cast from the **double** result of max to **int**[M]. These contextual effects make the expression resolution problem for C∀ both theoretically and practically difficult, but the observation driving the work in Chapter 4 is that of the many top-level expressions in a given program, most are straightforward and idiomatic so that programmers writing and maintaining the code can easily understand them; it follows that effective heuristics for common cases can bring down compiler runtime enough that a small proportion of harder-to-resolve expressions does not inordinately increase overall compiler runtime or memory usage.

[M]The two **double** casts function as type ascriptions selecting **double** max rather than casts from **int** max to **double**, and as such are zero-cost. The **int** to **double** conversion could be forced if desired with two casts: (**double**)(**int**)max

2.5 Type Features

The name overloading and polymorphism features of C∀ have the greatest effect on language design and compiler runtime, but there are a number of other features in the type system that have a smaller effect but are useful for code examples. These features are described here.

2.5.1 Reference Types

One of the key ergonomic improvements in C∀ is reference types, designed and implemented by Robert Schluntz [43]. Given some type T, a T& ("reference to T") is essentially an automatically dereferenced pointer. These types allow seamless pass-by-reference for function parameters, without the extraneous dereferencing syntax present in C; they also allow easy aliasing of nested values with a similarly convenient syntax. The addition of reference types also eliminated two syntactic special-cases present in previous versions of C∀. Consider the a call a += b to a compound assignment operator. The previous declaration for that operator is lvalue **int** ?+=?(**int**∗, **int**). To mutate the left argument, the built-in operators were special-cased to implicitly take the address of that argument, while the special lvalue syntax was used to mark the return type of a function as a mutable reference. With references, this declaration is re-written as **int**& ?+=?(**int**&, **int**). The reference semantics generalize the implicit address-of on the left argument and allow it to be used in user-declared functions, while also subsuming the (now removed) lvalue syntax for function return types.

The C standard makes heavy use of the concept of *lvalue*, an expression with a memory address; its complement, *rvalue* (a non-addressable expression) is not explicitly named in the standard. In C∀, the distinction between lvalue and rvalue can be re-framed in terms of reference and non-reference types, with the benefit of being able to express the difference in user code. C∀ references preserve the existing qualifier-dropping implicit lvalue-to-rvalue conversion from C (*e.g.* a **const volatile int**& can be implicitly copied to a bare **int**). To make reference types more easily usable in legacy pass-by-value code, C∀ also adds an implicit rvalue-to-lvalue conversion, implemented by storing the value in a compiler-generated temporary variable and passing a reference to that temporary. To mitigate the "**const** hell" problem present in C++, there is also a qualifier-dropping lvalue-to-lvalue conversion implemented by copying into a temporary:

```
const int magic = 42;
void inc_print( int& x ) { printf("%d\n", ++x); }
```

```
inc_print( magic );                    // legal; implicitly generated code in red below:

int tmp = magic;                       // to safely strip const-qualifier
inc_print( tmp );                      // tmp is incremented, magic is unchanged
```

Despite the similar syntax, C∀ references are significantly more flexible than C++ references. The primary issue with C++ references is that it is impossible to extract the address of the reference variable rather than the address of the referred-to variable. This breaks a number of the usual compositional properties of the C++ type system, *e.g.* a reference cannot be re-bound to another variable, nor is it possible to take a pointer to, array of, or reference to a reference. C∀ supports all of these use cases without further added syntax. The key to this syntax-free feature support is an observation made by the author that the address of a reference is a lvalue. In C, the address-of operator &x can only be applied to lvalue expressions, and always produces an immutable rvalue; C∀ supports reference re-binding by assignment to the address of a reference[N], and pointers to references by repeating the address-of operator:

```
int x = 2, y = 3;
int& r = x;                            // r aliases x
&r = &y;                               // r now aliases y
int** p = &&r;                         // p points to r
```

For better compatibility with C, the C∀ team has chosen not to differentiate function overloads based on top-level reference types, and as such their contribution to the difficulty of C∀ expression resolution is largely restricted to the implementation details of matching reference to non-reference types during type-checking.

2.5.2 Resource Management

C∀ also supports the RAII ("Resource Acquisition is Initialization") idiom originated by C++, thanks to the object lifetime work of Robert Schluntz [43]. This idiom allows a safer and more principled approach to resource management by tying acquisition of a resource to object initialization, with the corresponding resource release executed automatically at object finalization. A wide variety of conceptual resources may be conveniently managed by this scheme, including heap memory, file handles, and software locks.

[N]The syntactic difference between reference initialization and reference assignment is unfortunate, but preserves the ability to pass function arguments by reference (a reference initialization context) without added syntax.

C∀'s implementation of RAII is based on special constructor and destructor operators, available via the x{ ... } constructor syntax and ^x{ ... } destructor syntax. Each type has an overridable compiler-generated zero-argument constructor, copy constructor, assignment operator, and destructor, as well as a field-wise constructor for each appropriate prefix of the member fields of **struct** types. For **struct** types the default versions of these operators call their equivalents on each field of the **struct**. The main implication of these object lifetime functions for expression resolution is that they are all included as implicit type assertions for **otype** type variables, with a secondary effect being an increase in code size due to the compiler-generated operators. Due to these implicit type assertions, assertion resolution is pervasive in C∀ polymorphic functions, even those without explicit type assertions. Implicitly-generated code is shown in red in the following example:

```
struct kv {
    int key;
    char* value;
};

void ?{} (kv& this) {                          // default constructor
    this.key{};                                // call recursively on members
    this.value{};
}

void ?{} (kv& this, int key) {                 // partial field constructor
    this.key{ key };
    this.value{};                              // default-construct missing fields
}

void ?{} (kv& this, int key, char* value) {    // complete field constructor
    this.key{ key };
    this.value{ value };
}

void ?{} (kv& this, kv that) {                 // copy constructor
    this.key{ that.key };
    this.value{ that.value };
}

kv ?=? (kv& this, kv that) {                   // assignment operator
    this.key = that.key;
    this.value = that.value;
}
```

```
void ^?{} (kv& this) {                          // destructor
    ^this.key{};
    ^this.value{};
}

forall(otype T | { void ?{}(T&); void ?{}(T&, T); T ?=?(T&, T); void ^?{}(T&); })
void foo(T);
```

2.5.3 Tuple Types

C∀ adds *tuple types* to C, a syntactic facility for referring to lists of values anonymously or with a single identifier. An identifier may name a tuple, a function may return one, and a tuple may be implicitly *destructured* into its component values. The implementation of tuples in `cfa-cc`'s code generation is based on the generic types introduced in Chapter 3, with one compiler-generated generic type for each tuple arity. This reuse allows tuples to take advantage of the same runtime optimizations available to generic types, while reducing code bloat. An extended presentation of the tuple features of C∀ can be found in [30], but the following example demonstrates the basic features:

```
[char, char] x1 = ['!', '?'];           // tuple type and expression syntax
int x2 = 2;

forall(otype T)
[T, T] swap1( T a, T b ) {
    return [b, a];                      // one-line swap syntax
}

x = swap( x );                          // destructure x1 into two elements
                                        // cannot use x2, not enough arguments

void swap2( int, char, char );

swap( x, x );                           // swap2( x2, x1 )
                                        // not swap1( x2, x2 ) due to polymorphism cost
```

Tuple destructuring breaks the one-to-one relationship between identifiers and values. Hence, some argument-parameter matching strategies for expression resolution are precluded, as well as cheap interpretation filters based on comparing number of parameters and arguments. As an example, in the call to swap(x, x) above, the second x can be re-

solved starting at the second or third parameter of swap_2, depending which interpretation of x is chosen for the first argument.

2.6 Conclusion

C∀ adds a significant number of features to standard C, increasing the expressivity and re-usability of C∀ code while maintaining backwards compatibility for both code and larger language paradigms. This flexibility does incur significant added compilation costs, however, the mitigation of which are the primary concern of this thesis.

Chapter 3

Generic Types

A significant shortcoming in standard C is the lack of reusable type-safe abstractions for generic data structures and algorithms. Broadly speaking, there are three approaches to implement abstract data structures in C. One approach is to write bespoke data structures for each context in which they are needed. While this approach is flexible and supports integration with the C type checker and tooling, it is also tedious and error prone, especially for more complex data structures. A second approach is to use **void**$*$-based polymorphism, *e.g.* the C standard library functions bsearch and qsort, which allow for the reuse of common functionality. However, basing all polymorphism on **void**$*$ eliminates the type checker's ability to ensure that argument types are properly matched, often requiring a number of extra function parameters, pointer indirection, and dynamic allocation that is otherwise unnecessary. A third approach to generic code is to use preprocessor macros, which does allow the generated code to be both generic and type checked, but errors in such code may be difficult to locate and debug. Furthermore, writing and using preprocessor macros is unnatural and inflexible. Figure 3.1 demonstrates the bespoke approach for a simple linked list with insert and head operations, while Figure 3.2 and Figure 3.3 show the same example using **void**$*$ and **#define**-based polymorphism, respectively.

C++, Java, and other languages use *generic types* (or *parameterized types*) to produce type-safe abstract data types. Design and implementation of generic types for C∀ is the first major contribution of this thesis, a summary of which is published in [30] and on which this chapter is closely based. C∀ generic types integrate efficiently and naturally with the existing polymorphic functions in C∀, while retaining backward compatibility with C in layout and support for separate compilation. A generic type can be declared in C∀ by placing a **forall** specifier on a **struct** or **union** declaration, and instantiated using

```
#include <stdlib.h>          // for malloc
#include <stdio.h>           // for printf

struct int_list { int value; struct int_list* next; };

void int_list_insert( struct int_list** ls, int x ) {
    struct int_list* node = malloc(sizeof(struct int_list));
    node->value = x; node->next = *ls;
    *ls = node;
}

int int_list_head( const struct int_list* ls ) { return ls->value; }

// all code must be duplicated for every generic instantiation

struct string_list { const char* value; struct string_list* next; };

void string_list_insert( struct string_list** ls, const char* x ) {
    struct string_list* node = malloc(sizeof(struct string_list));
    node->value = x; node->next = *ls;
    *ls = node;
}

const char* string_list_head( const struct string_list* ls )
    { return ls->value; }

// use is efficient and idiomatic

int main() {
    struct int_list* il = NULL;
    int_list_insert( &il, 42 );
    printf("%d\n", int_list_head(il));

    struct string_list* sl = NULL;
    string_list_insert( &sl, "hello" );
    printf("%s\n", string_list_head(sl));
}
```

Figure 3.1: Bespoke code for linked list implementation.

```
#include <stdlib.h>          // for malloc
#include <stdio.h>           // for printf

// single code implementation

struct list { void* value; struct list* next; };

// internal memory management requires helper functions

void list_insert( struct list** ls, void* x, void* (*copy)(void*) ) {
    struct list* node = malloc(sizeof(struct list));
    node->value = copy(x); node->next = *ls;
    *ls = node;
}

void* list_head( const struct list* ls ) { return ls->value; }

// helpers duplicated per type

void* int_copy(void* x) {
    int* n = malloc(sizeof(int));
    *n = *(int*)x;
    return n;
}

void* string_copy(void* x) { return strdup((const char*)x); }

int main() {
    struct list* il = NULL;
    int i = 42;
    list_insert( &il, &i, int_copy );
    printf("%d\n", *(int*)list_head(il));              // unsafe type cast

    struct list* sl = NULL;
    list_insert( &sl, "hello", string_copy );
    printf("%s\n", (char*)list_head(sl));              // unsafe type cast
}
```

Figure 3.2: **void***-polymorphic code for linked list implementation.

```
#include <stdlib.h>          // for malloc
#include <stdio.h>           // for printf

// code is nested in macros

#define list(N) N ## _list

#define list_insert(N) N ## _list_insert

#define list_head(N) N ## _list_head

#define define_list(N, T)                                                    \
    struct list(N) { T value; struct list(N)* next; };                       \
                                                                             \
    void list_insert(N)( struct list(N)** ls, T x ) {                        \
        struct list(N)* node = malloc(sizeof(struct list(N)));               \
        node->value = x; node->next = *ls;                                   \
        *ls = node;                                                          \
    }                                                                        \
                                                                             \
    T list_head(N)( const struct list(N)* ls ) { return ls->value; }

define_list(int, int);                  // defines int_list
define_list(string, const char*);       // defines string_list

// use is efficient, but syntactically idiosyncratic

int main() {
    struct list(int)* il = NULL;        // does not match compiler-visible name
    list_insert(int)( &il, 42 );
    printf("%d\n", list_head(int)(il));

    struct list(string)* sl = NULL;
    list_insert(string)( &sl, "hello" );
    printf("%s\n", list_head(string)(sl));
}
```

Figure 3.3: Macros for generic linked list implementation.

a parenthesized list of types after the generic name. An example comparable to the C polymorphism examples in Figures 3.1, 3.2, and 3.3 can be seen in Figure 3.4.

3.1 Design

Though a number of languages have some implementation of generic types, backward compatibility with both C and existing C∀ polymorphism present some unique design constraints for C∀ generics. The guiding principle is to maintain an unsurprising language model for C programmers without compromising runtime efficiency. A key insight for this design is that C already possesses a handful of built-in generic types (*derived types* in the language of the standard [9, §6.2.5]), notably pointer (T∗) and array (T[]), and that user-definable generics should act similarly.

3.1.1 Related Work

One approach to the design of generic types is that taken by C++ templates [15]. The template approach is closely related to the macro-expansion approach to C polymorphism demonstrated in Figure 3.3, but where the macro-expansion syntax has been given first-class language support. Template expansion has the benefit of generating code with near-optimal runtime efficiency, as distinct optimizations can be applied for each instantiation of the template. On the other hand, template expansion can also lead to significant code bloat, exponential in the worst case [22], and the costs of increased compilation time and instruction cache pressure cannot be ignored. The most significant restriction of the C++ template model is that it breaks separate compilation and C's translation-unit-based encapsulation mechanisms. Because a C++ template is not actually code, but rather a "recipe" to generate code, template code must be visible at its call site to be used. Furthermore, C++ template code cannot be type-checked without instantiating it, a time consuming process with no hope of improvement until C++ concepts [10] are standardized in C++20. C code, by contrast, only needs a function declaration to call that function or a **struct** declaration to use (by-pointer) values of that type, desirable properties to maintain in C∀.

Java [19] has another prominent implementation for generic types, introduced in Java 5 and based on a significantly different approach than C++. The Java approach has much more in common with the **void**∗-polymorphism shown in Figure 3.2; since in Java nearly all data is stored by reference, the Java approach to polymorphic data is to store pointers to arbitrary data and insert type-checked implicit casts at compile-time. This process of

```
#include <stdlib.hfa>          // for alloc
#include <stdio.h>             // for printf

forall(otype T) struct list { T value; list(T)* next; };

// single polymorphic implementation of each function
// overloading reduces need for namespace prefixes

forall(otype T) void insert( list(T)** ls, T x ) {
    list(T)* node = alloc();   // type-inferring alloc
    (*node){ x, *ls };         // concise constructor syntax
    *ls = node;
}

forall(otype T) T head( const list(T)* ls ) { return ls->value; }

// use is clear and efficient

int main() {
    list(int)* il = 0;
    insert( &il, 42 );         // inferred polymorphic T
    printf("%d\n", head(il));

    list(const char*)* sl = 0;
    insert( &sl, "hello" );
    printf("%s\n", head(sl));
}
```

Figure 3.4: C∀ generic linked list implementation.

type erasure has the benefit of allowing a single instantiation of polymorphic code, but relies heavily on Java's object model. To use this model, a more C-like language such as C∀ would be required to dynamically allocate internal storage for variables, track their lifetime, and properly clean them up afterward.

Cyclone [21] extends C and also provides capabilities for polymorphic functions and existential types which are similar to C∀'s **forall** functions and generic types. Cyclone existential types can include function pointers in a construct similar to a virtual function table, but these pointers must be explicitly initialized at some point in the code, which is tedious and error-prone compared to C∀'s implicit assertion satisfaction. Furthermore, Cyclone's polymorphic functions and types are restricted to abstraction over types with the same layout and calling convention as **void**$*$, *i.e.* only pointer types and **int**. In the C∀ terminology discussed in Section 3.2, all Cyclone polymorphism must be dtype-static. While the Cyclone polymorphism design provides the efficiency benefits discussed in Section 3.2.3 for dtype-static polymorphism, it is more restrictive than the more general model of C∀.

Many other languages include some form of generic types. As a brief survey, ML [28] was the first language to support parametric polymorphism, but unlike C∀ does not support the use of assertions and traits to constrain type arguments. Haskell [23] combines ML-style polymorphism with the notion of type classes, similar to C∀ traits, but requiring an explicit association with their implementing types, unlike C∀. Objective-C [32] is an extension to C which has had some industrial success; however, it did not support type-checked generics until recently [47], and its garbage-collected, message-passing object-oriented model is a radical departure from C. Go [20], and Rust [41] are modern compiled languages with abstraction features similar to C∀ traits: *interfaces* in Go and *traits* in Rust. Go has implicit interface implementation and uses a "fat pointer" construct to pass polymorphic objects to functions, similar in principle to C∀'s implicit forall parameters. Go does not, however, allow user code to define generic types, restricting Go programmers to the small set of generic types defined by the compiler. Rust has powerful abstractions for generic programming, including explicit implementation of traits and options for both separately-compiled virtual dispatch and template-instantiated static dispatch in functions. On the other hand, the safety guarantees of Rust's *lifetime* abstraction and borrow checker impose a distinctly idiosyncratic programming style and steep learning curve; C∀, with its more modest safety features, allows direct ports of C code while maintaining the idiomatic style of the original source.

3.1.2 C∀ Generics

The generic types design in C∀ draws inspiration from both C++ and Java generics, capturing useful aspects of each. Like C++ template types, generic **struct** and **union** types in C∀ have macro-expanded storage layouts, but, like Java generics, C∀ generic types can be used with separately-compiled polymorphic functions without requiring either the type or function definition to be visible to the other. The fact that the storage layout of any instantiation of a C∀ generic type is identical to that of the monomorphic type produced by simple macro replacement of the generic type parameters is important to provide consistent and predictable runtime performance, and to not impose any undue abstraction penalty on generic code. As an example, consider the following generic type and function:

```
forall( otype R, otype S ) struct pair { R first; S second; };

pair(const char*, int) with_len( const char* s ) {
    return (pair(const char*, int)){ s, strlen(s) };
}
```

In this example, with_len is defined at the same scope as pair, but it could be called from any context that can see the definition of pair and a declaration of with_len. If its return type were pair(**const char***, **int**)*, callers of with_len would only need the declaration **forall**(**otype** R, **otype** S) **struct** pair to call it, in accordance with the usual C rules for opaque types.

with_len is itself a monomorphic function, returning a type that is structurally identical to **struct** { **const char*** first; **int** second; }, and as such could be called from C given appropriate re-declarations and demangling flags. However, the definition of with_len depends on a polymorphic function call to the pair constructor, which only needs to be written once (in this case, implicitly by the compiler according to the usual C∀ constructor generation [43]) and can be re-used for a wide variety of pair instantiations. Since the parameters to this polymorphic constructor call are all statically known, compiler inlining can in principle eliminate any runtime overhead of this polymorphic call.

C∀ deliberately does not support C++-style partial specializations of generic types. A particularly infamous example in the C++ standard library is vector<**bool**>, which is represented as a bit-string rather than the array representation of the other vector instantiations. Complications from this inconsistency (chiefly the fact that a single bit is not addressable, unlike an array element) make the C++ vector unpleasant to use in generic contexts due to the break in its public interface. Rather than attempting to plug leaks in the template specialization abstraction with a detailed method interface, C∀ takes the more consistent position that two types with an unrelated data layout are in fact unrelated

types, and should be handled with different code. Of course, to the degree that distinct types are similar enough to share an interface, the C∀ **trait** system allows such an interface to be defined, and objects of types implementing that **trait** to be operated on using the same polymorphic functions.

Since C∀ polymorphic functions can operate over polymorphic generic types, functions over such types can be partially or completely specialized using the usual overload selection rules. As an example, the following generalization of with_len is a semantically-equivalent function which works for any type that has a len function declared, making use of both the ad-hoc (overloading) and parametric (**forall**) polymorphism features of C∀:

```
forall(otype T, otype I | { I len(T); })
pair(T, I) with_len( T s ) {
    return (pair(T,I)){ s, len(s) };
}
```

C∀ generic types also support type constraints, as in **forall** functions. For example, the following declaration of a sorted set type ensures that the set key implements equality and relational comparison:

```
forall(otype Key | { int ?==?(Key, Key); int ?<?(Key, Key); }) struct sorted_set;
```

These constraints are enforced by applying equivalent constraints to the compiler-generated constructors for this type.

3.2 Implementation

The ability to use generic types in polymorphic contexts means that the C∀ implementation must support a mechanism for accessing fields of generic types dynamically. While `cfa-cc` could in principle use this same mechanism for accessing fields of generic types in monomorphic contexts as well, such an approach would throw away compiler knowledge of static types and impose an unnecessary runtime cost. Instead, my design for generic types in `cfa-cc` distinguishes between *concrete* generic types that have a fixed memory layout regardless of type parameters and *dynamic* generic types that may vary in memory layout depending on their type parameters.

A *dtype-static* type has polymorphic parameters but is still concrete. Polymorphic pointers are an example of dtype-static types; given some type variable T, T is a polymorphic type, but T∗ has a fixed size and can therefore be represented by a **void**∗ in code generation. In particular, generic types where all parameters are un-sized (*i.e.* they do not

conform to the built-in sized trait, which is satisfied by all types the compiler knows the size and alignment of) are always concrete, as there is no possibility for their layout to vary based on type parameters of unknown size and alignment. More precisely, a type is concrete if and only if all of its sized type parameters are concrete, and a concrete type is dtype-static if any of its type parameters are (possibly recursively) polymorphic. To illustrate, the following code using the pair type from above has each use of pair commented with its class:

```
//dynamic, layout varies based on T
forall(otype T) T value1( pair(const char*, T) p ) { return p.second; }

// dtype-static, F* and T* are concrete but recursively polymorphic
forall(dtype F, otype T) T value2( pair(F*, T*) ) { return *p.second; }

pair(const char*, int) p = {"magic", 42};          // concrete
int i = value(p);
pair(void*, int*) q = {0, &i};                     // concrete
i = value(q);
double d = 1.0;
pair(double*, double*) r = {&d, &d};               // concrete
d = value(r);
```

3.2.1 Concrete Generic Types

The `cfa-cc` translator template-expands concrete generic types into new structure types, affording maximal inlining. To enable interoperation among equivalent instantiations of a generic type, `cfa-cc` saves the set of instantiations currently in scope and reuses the generated structure declarations where appropriate. In particular, tuple types are implemented as a single compiler-generated generic type definition per tuple arity, and can be instantiated and reused according to the usual rules for generic types. A function declaration that accepts or returns a concrete generic type produces a declaration for the instantiated structure in the same scope, which all callers may reuse. As an example, the concrete instantiation for pair(const char*, int) is[A]:

```
struct _pair_conc0 { const char * first; int second; };
```

A concrete generic type with dtype-static parameters is also expanded to a structure type, but this type is used for all matching instantiations. In the example above, the

[A] Field name mangling for overloading purposes is omitted.

pair(F∗, T∗) parameter to value is such a type; its expansion is below[A], and it is used as the type of the variables q and r as well, with casts for member access where appropriate:

```
struct _pair_conc1 { void* first; void* second; };
```

3.2.2 Dynamic Generic Types

In addition to this efficient implementation of concrete generic types, C∀ also offers flexibility with powerful support for dynamic generic types. In the pre-existing compiler design, **otype** (and all sized) type parameters come with implicit size and alignment parameters provided by the caller. The design for generic types presented here adds an *offset array* containing structure-member offsets for dynamic generic **struct** types. A dynamic generic **union** needs no such offset array, as all members are at offset 0, but size and alignment are still necessary. Access to members of a dynamic structure is provided at runtime via base-displacement addressing of the structure pointer and the member offset (similar to the offsetof macro), moving a compile-time offset calculation to runtime.

The offset arrays are statically generated where possible. If a dynamic generic type is passed or returned by value from a polymorphic function, `cfa-cc` can safely assume that the generic type is complete (*i.e.* has a known layout) at any call site, and the offset array is passed from the caller; if the generic type is concrete at the call site, the elements of this offset array can even be statically generated using the C offsetof macro. As an example, the body of value_2 above is implemented as:

```
_assign_T( _retval, p + _offsetof_pair[1] );                    // return *p.second
```

Here, _assign_T is passed in as an implicit parameter from **otype** T and takes two T∗ (**void**∗ in the generated code[B]), a destination and a source, and _retval is the pointer to a caller-allocated buffer for the return value, the usual C∀ method to handle dynamically-sized return types. _offsetof_pair is the offset array passed into value; this array is statically generated at the call site as:

```
size_t _offsetof_pair[] = {offsetof(_pair_conc0, first),  offsetof(_pair_conc0, second)};
```

Layout Functions

In some cases, the offset arrays cannot be statically generated. For instance, modularity is generally provided in C by including an opaque forward declaration of a structure and

[B] A GCC extension allows arithmetic on **void**∗, calculated as if **sizeof**(**void**) == 1.

associated accessor and mutator functions in a header file, with the actual implementations in a separately-compiled `.c` file. C∀ supports this pattern for generic types, implying that the caller of a polymorphic function may not know the actual layout or size of a dynamic generic type and only holds it by pointer. `cfa-cc` automatically generates *layout functions* for cases where the size, alignment, and offset array of a generic struct cannot be passed into a function from that function's caller. These layout functions take as arguments pointers to size and alignment variables and a caller-allocated array of member offsets, as well as the size and alignment of all sized parameters to the generic structure. Un-sized parameters are not passed because they are forbidden from being used in a context that affects layout by C's usual rules about incomplete types. Similarly, the layout function can only safely be called from a context where the generic type definition is visible, because otherwise the caller does not know how large to allocate the array of member offsets.

The C standard does not specify a memory layout for structs, but the System V ABI [27] does; compatibility with this standard is sufficient for C∀'s currently-supported architectures, though future ports may require different layout-function generation algorithms. This algorithm, sketched below in pseudo-C∀, is a straightforward mapping of consecutive fields into the first properly-aligned offset in the **struct** layout; layout functions for **union** types omit the offset array and simply calculate the maximum size and alignment over all union variants. Since `cfa-cc` generates a distinct layout function for each type, constant-folding and loop unrolling are applied.

```
forall(dtype T1, dtype T2, ... | sized(T1) | sized(T2) | ...)
void layout(size_t* size, size_t* align, size_t* offsets) {
    *size = 0; *align = 1;
    // set up members
    for ( int i = 0; i < n_fields; ++i ) {
        // pad to alignment
        size_t off_align = *size % alignof(field[i]);
        if ( off_align != 0 ) { *size += alignof(field[i]) - off_align; }
        // mark member, increase size, and fix alignment
        offsets[i] = *size;
        *size += sizeof(field[i]);
        if ( *align < alignof(field[i]) ) { *align = alignof(field[i]); }
    }
    // final padding to alignment
    size_t off_align = *size % *align;
    if ( off_align != 0 ) { *size += *align - off_align; }
}
```

Results of layout-function calls are cached so that they are only computed once per

type per function. Layout functions also allow generic types to be used in a function definition without reflecting them in the function signature, an important implementation-hiding constraint of the design. For instance, a function that strips duplicate values from an unsorted list(T) likely has a reference to the list as its only explicit parameter, but uses some sort of set(T) internally to test for duplicate values. This function could acquire the layout for set(T) by calling its layout function, providing as an argument the layout of T implicitly passed into that function.

Whether a type is concrete, dtype-static, or dynamic is decided solely on the basis of the type arguments and **forall** clause type parameters. This design allows opaque forward declarations of generic types, *e.g.* **forall**(**otype** T) **struct** Box; like in C, all uses of Box(T) can be separately compiled, and callers from other translation units know the proper calling conventions. In an alternate design, where the definition of a structure type is included in deciding whether a generic type is dynamic or concrete, some further types may be recognized as dtype-static — *e.g.* Box could be defined with a body { T* p; }, and would thus not depend on T for its layout. However, the existence of an **otype** parameter T means that Box *could* depend on T for its layout if this definition is not visible, and preserving separate compilation (and the associated C compatibility) is a more important design metric.

3.2.3 Applications of Dtype-static Types

The reuse of dtype-static structure instantiations enables useful programming patterns at zero runtime cost. The most important such pattern is using **forall**(**dtype** T) T* as a type-checked replacement for **void***, *e.g.* creating a lexicographic comparison function for pairs of pointers.

```
forall(dtype T)
int lexcmp( pair(T*, T)* a, pair(T*, T)* b, int (*cmp)(T*, T*) ) {
    int c = cmp( a->first, b->first );
    return c ? c : cmp( a->second, b->second );
}
```

Since pair(T*, T*) is a concrete type, there are no implicit parameters passed to lexcmp; hence, the generated code is identical to a function written in standard C using **void***, yet the C∀ version is type-checked to ensure members of both pairs and arguments to the comparison function match in type.

Another useful pattern enabled by reused dtype-static type instantiations is zero-cost *tag structures*. Sometimes, information is only used for type checking and can be omitted

at runtime. In the example below, scalar is a dtype-static type; hence, all uses have a single structure definition containing **unsigned long** and can share the same implementations of common functions, like ?+?. These implementations may even be separately compiled, unlike C++ template functions. However, the C∀ type checker ensures matching types are used by all calls to ?+?, preventing nonsensical computations like adding a length to a volume.

```
forall(dtype Unit) struct scalar { unsigned long value; };
struct metres {};
struct litres {};

forall(dtype U) scalar(U) ?+?(scalar(U) a, scalar(U) b) {
    return (scalar(U)){ a.value + b.value };
}

scalar(metres) half_marathon = { 21098 };
scalar(litres) pool = { 2500000 };
scalar(metres) marathon = half_marathon + half_marathon;
marathon + pool;                // compiler ERROR, mismatched types
```

3.3 Performance Experiments

To validate the practicality of this generic type design, microbenchmark-based tests were conducted against a number of comparable code designs in C and C++, first published in [30]. Since these languages are all C-based and compiled with the same compiler backend, maximal-performance benchmarks should show little runtime variance, differing only in length and clarity of source code. A more illustrative comparison measures the costs of idiomatic usage of each language's features. The code below shows the C∀ benchmark tests for a generic stack based on a singly-linked list; the test suite is equivalent for the other languages, code for which is included in Appendix A. The experiment uses element types **int** and pair(**short**, **char**) and pushes $N = 4M$ elements on a generic stack, copies the stack, clears one of the stacks, and finds the maximum value in the other stack.

```
#define N 4000000
int main() {
    int max = 0, val = 42;
    stack( int ) si, ti;

    REPEAT_TIMED( "push_int", N, push( si, val ); )
    TIMED( "copy_int", ti{ si }; )
```

```
    TIMED( "clear_int", clear( si ); )
    REPEAT_TIMED( "pop_int", N, int x = pop( ti ); if ( x > max ) max = x; )

    pair( short, char ) max = { 0h, '\0' }, val = { 42h, 'a' };
    stack( pair( short, char ) ) sp, tp;

    REPEAT_TIMED( "push_pair", N, push( sp, val ); )
    TIMED( "copy_pair", tp{ sp }; )
    TIMED( "clear_pair", clear( sp ); )
    REPEAT_TIMED( "pop_pair", N, pair(short, char) x = pop( tp );
        if ( x > max ) max = x; )
}
```

The four versions of the benchmark implemented are C with **void**$*$-based polymorphism, CFA with parametric polymorphism, C++ with templates, and C++ using only class inheritance for polymorphism, denoted C++obj. The C++obj variant illustrates an alternative object-oriented idiom where all objects inherit from a base object class, a language design similar to Java 4; in particular, runtime checks are necessary to safely downcast objects. The most notable difference among the implementations is the memory layout of generic types: CFA and C++ inline the stack and pair elements into corresponding list and pair nodes, while C and C++obj lack such capability and, instead, must store generic objects via pointers to separately allocated objects. Note that the C benchmark uses unchecked casts as C has no runtime mechanism to perform such checks, whereas CFA and C++ provide type safety statically.

Figure 3.5 and Table 3.1 show the results of running the described benchmark. The graph plots the median of five consecutive runs of each program, with an initial warm-up run omitted. All code is compiled at -O2 by gcc or g++ 6.4.0, with all C++ code compiled as C++14. The benchmarks are run on an Ubuntu 16.04 workstation with 16 GB of RAM and a 6-core AMD FX-6300 CPU with 3.5 GHz maximum clock frequency. I conjecture that these results scale across most uses of generic types, given the constant underlying polymorphism implementation.

The C and C++obj variants are generally the slowest and have the largest memory footprint, due to their less-efficient memory layout and the pointer indirection necessary to implement generic types in those languages; this inefficiency is exacerbated by the second level of generic types in the pair benchmarks. By contrast, the CFA and C++ variants run in noticeably less time for both the integer and pair because of the equivalent storage layout, with the inlined libraries (*i.e.* no separate compilation) and greater maturity of the C++ compiler contributing to its lead. C++obj is slower than C largely due to the cost of runtime type checking of downcasts (implemented with dynamic_cast); the outlier

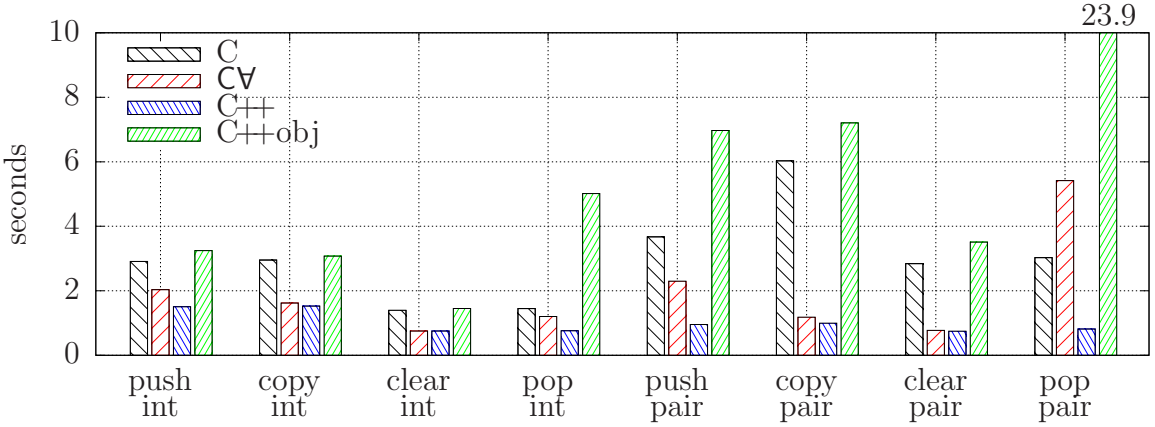


Figure 3.5: Benchmark timing results (smaller is better)

Table 3.1: Properties of benchmark code

	C	C∀	C++	C++obj
maximum memory usage (MB)	10 001	2 502	2 503	11 253
source code size (lines)	201	191	125	294
redundant type annotations (lines)	27	0	2	16
binary size (KB)	14	257	14	37

for C∀, pop pair, results from the complexity of the generated-C polymorphic code. The gcc compiler is unable to optimize some dead code and condense nested calls; a compiler designed for C∀ could more easily perform these optimizations. Finally, the binary size for C∀ is larger because of static linking with the C∀ prelude library, which includes function definitions for all the built-in operators.

C∀ is also competitive in terms of source code size, measured as a proxy for programmer effort. The line counts in Table 3.1 include implementations of pair and stack types for all four languages for purposes of direct comparison, although it should be noted that C∀ and C++ have prewritten data structures in their standard libraries that programmers would generally use instead. Use of these standard library types has minimal impact on the performance benchmarks, but shrinks the C∀ and C++ code to 39 and 42 lines, respectively. The difference between the C∀ and C++ line counts is primarily declaration duplication to implement separate compilation; a header-only C∀ library is similar in length to the C++ version. On the other hand, due to the language shortcomings mentioned at the beginning of the chapter, C does not have a generic collections library in its standard distribution, resulting in frequent re-implementation of such collection types by C programmers. C++obj does not use the C++ standard template library by construction, and, in fact, includes the definition of object and wrapper classes for **char**, **short**, and **int** in its line count, which inflates this count somewhat, as an actual object-oriented language would include these in the standard library. I justify the given line count by noting that many object-oriented languages do not allow implementing new interfaces on library types without subclassing or wrapper types, which may be similarly verbose.

Line count is a fairly rough measure of code complexity; another important factor is how much type information the programmer must specify manually, especially where that information is not type-checked. Such unchecked type information produces a heavier documentation burden and increased potential for runtime bugs and is much less common in C∀ than C, with its manually specified function pointer arguments and format codes, or C++obj, with its extensive use of un-type-checked downcasts, *e.g.* object to integer when popping a stack. To quantify this manual typing, the "redundant type annotations" line in Table 3.1 counts the number of lines on which the known type of a variable is re-specified, either as a format specifier, explicit downcast, type-specific function, or by name in a **sizeof**, **struct** literal, or new expression. The C++ benchmark uses two redundant type annotations to create new stack nodes, whereas the C and C++obj benchmarks have several such annotations spread throughout their code. The C∀ benchmark is able to eliminate *all* redundant type annotations through use of the return-type polymorphic alloc function in the C∀ standard library.

3.4 Future Work

The generic types presented here are already sufficiently expressive to implement a variety of useful library types. However, some other features based on this design could further improve C∀.

The most pressing addition is the ability to have non-type generic parameters. C already supports fixed-length array types, *e.g.* **int**[10]; these types are essentially generic types with unsigned integer parameters (*i.e.* array dimension), and allowing C∀ users the capability to build similar types is a requested feature.

The implementation mechanisms behind generic types can also be used to add new features to C∀. One such potential feature is *field assertions*, an addition to the existing function and variable assertions on polymorphic type variables. These assertions could be specified using this proposed syntax:

```
trait hasXY(dtype T) {
    int T.x;                    // T has a field x of type int
    int T.y;                    // T has a field y of type int
};
```

Implementation of these field assertions would be based on the same code that supports member access by dynamic offset calculation for dynamic generic types. Simulating field access can already be done more flexibly in C∀ by declaring a trait containing an accessor function to be called from polymorphic code, but these accessor functions impose some overhead both to write and call, and directly providing field access via an implicit offset parameter would be both more concise and more efficient. Of course, there are language design trade-offs to such an approach, notably that providing the two similar features of field and function assertions would impose a burden of choice on programmers writing traits, with field assertions more efficient, but function assertions more general; given this open design question a decision on field assertions is deferred until C∀ is more mature.

If field assertions are included in the language, a natural extension would be to provide a structural inheritance mechanism for every **struct** type that simply turns the list of **struct** fields into a list of field assertions, allowing monomorphic functions over that type to be generalized to polymorphic functions over other similar types with added or reordered fields, for example:

```
struct point { int x, y; };             // traitof(point) is equivalent to hasXY above
struct coloured_point { int x, y; enum { RED, BLACK } colour };

// works for both point and coloured_point
```

```
forall(dtype T | traitof(point)(T) )
double hypot( T& p ) { return sqrt( p.x*p.x + p.y*p.y ); }
```

C∀ could also support a packed or otherwise size-optimized representation for generic types based on a similar mechanism — nothing in the use of the offset arrays implies that the field offsets need to be monotonically increasing.

With respect to the broader C∀ polymorphism design, the experimental results in Section 3.3 demonstrate that though the runtime impact of C∀'s dynamic virtual dispatch is low, it is not as low as the static dispatch of C++ template inlining. However, rather than subject all C∀ users to the compile-time costs of ubiquitous template expansion, it is better to target performance-sensitive code more precisely. Two promising approaches are an **inline** annotation at polymorphic function call sites to create a template specialization of the function (provided the code is visible) or placing a different **inline** annotation on polymorphic function definitions to instantiate a specialized version of the function for some set of types. These approaches are complementary and allow performance optimizations to be applied only when necessary, without suffering global code bloat.

Chapter 4

Resolution Algorithms

The main task of the `cfa-cc` type-checker is *expression resolution*: determining which declarations the identifiers in each expression correspond to. Resolution is a straightforward task in C, as no simultaneously-visible declarations share identifiers, but in C∀, the name overloading features discussed in Section 2.2 generate multiple candidate declarations for each identifier. A given matching between identifiers and declarations in an expression is an *interpretation*; an interpretation also includes information about polymorphic type bindings and implicit casts to support the C∀ features discussed in Sections 2.3 and 2.4, each of which increase the number of valid candidate interpretations. To choose among valid interpretations, a *conversion cost* is used to rank interpretations. This conversion cost is summed over all subexpression interpretations in the interpretation of a top-level expression. Hence, the expression resolution problem is to find the unique minimal-cost interpretation for an expression, reporting an error if no such unique interpretation exists.

4.1 Expression Resolution

The expression resolution pass in `cfa-cc` must traverse an input expression, match identifiers to available declarations, rank candidate interpretations according to their conversion cost, and check type assertion satisfaction for these candidates. Once the set of valid interpretations for the top-level expression is found, the expression resolver selects the unique minimal-cost candidate or reports an error.

The expression resolution problem in C∀ is more difficult than the analogous problems in C or C++. As mentioned above, the lack of name overloading in C (except for built-in

operators) makes its resolution problem substantially easier. A comparison of the richer type systems in CA and C++ highlights some of the challenges in CA expression resolution. The key distinction between CA and C++ resolution is that C++ uses a greedy algorithm for selection of candidate functions given their argument interpretations, whereas CA allows contextual information from superexpressions to influence the choice among candidate functions. One key use of this contextual information is for type inference of polymorphic return types; C++ requires explicit specification of template parameters that only occur in a function's return type, while CA allows the instantiation of these type parameters to be inferred from context (and in fact does not allow explicit specification of type parameters to a function), as in the following example:

```
forall(dtype T) T& deref(T*);        // dereferences pointer
forall(otype T) T* def();            // new heap-allocated default-initialized value

int& i = deref( def() );
```

In this example, the CA compiler infers the type arguments of deref and def from the **int**& type of i; C++, by contrast, requires a type parameter on def[A], *i.e.* deref(def<**int**>()). Similarly, while both CA and C++ rank candidate functions based on a cost metric for implicit conversions, CA allows a suboptimal subexpression interpretation to be selected if it allows a lower-cost overall interpretation, while C++ requires that each subexpression interpretation have minimal cost. Because of this use of contextual information, the CA expression resolver must consider multiple interpretations of each function argument, while the C++ compiler has only a single interpretation for each argument[B]. Additionally, until the introduction of concepts in C++20 [10], C++ expression resolution has no analogue to CA assertion satisfaction checking, a further complication for a CA compiler. The precise definition of CA expression resolution in this section further expands on the challenges of this problem.

4.1.1 Type Unification

The polymorphism features of CA require binding of concrete types to polymorphic type variables. Briefly, `cfa-cc` keeps a mapping from type variables to the concrete types they are bound to as an auxiliary data structure during expression resolution; Chapter 5 describes this *environment* data structure in more detail. A *unification* algorithm is used to simultaneously check two types for equivalence with respect to the substitutions in an

[A] The type parameter of deref can be inferred from its argument.

[B] With the exception of address-of operations on functions.

environment and update that environment. Essentially, unification recursively traverses the structure of both types, checking them for equivalence, and when it encounters a type variable, it replaces it with the concrete type it is bound to; if the type variable has not yet been bound, the unification algorithm assigns the equivalent type as the bound type of the variable, after performing various consistency checks. Ditchfield [14] and Bilson [6] describe the semantics of C∀ unification in more detail.

4.1.2 Conversion Cost

C∀, like C, allows inexact matches between the type of function parameters and function call arguments. Both languages insert *implicit conversions* in these situations to produce an exact type match, and C∀ also uses the relative *cost* of different conversions to select among overloaded function candidates. C does not have an explicit cost model for implicit conversions, but the "usual arithmetic conversions" [9, §6.3.1.8] used to decide which arithmetic operators to apply define one implicitly. The only context in which C has name overloading is the arithmetic operators, and the usual arithmetic conversions define a *common type* for mixed-type arguments to binary arithmetic operators. Since for backward-compatibility purposes the conversion costs of C∀ must produce an equivalent result to these common type rules, it is appropriate to summarize [9, §6.3.1.8] here:

- If either operand is a floating-point type, the common type is the size of the largest floating-point type. If either operand is **_Complex**, the common type is also **_Complex**.
- If both operands are of integral type, the common type has the same size[C] as the larger type.
- If the operands have opposite signedness, the common type is **signed** if the **signed** operand is strictly larger, or **unsigned** otherwise. If the operands have the same signedness, the common type shares it.

Beginning with the work of Bilson [6], C∀ defines a *conversion cost* for each function call in a way that generalizes C's conversion rules. Loosely defined, the conversion cost counts

[C]Technically, the C standard defines a notion of *rank* in [9, §6.3.1.1], a distinct value for each **signed** and **unsigned** pair; integral types of the same size thus may have distinct ranks. For instance, though **int** and **long** may have the same size, **long** always has greater rank. The standard-defined types are declared to have greater rank than any types of the same size added as compiler extensions.

the implicit conversions utilized by an interpretation. With more specificity, the cost is a lexicographically-ordered tuple, where each element corresponds to a particular kind of conversion. In Bilson's design, conversion cost is a 3-tuple, $(unsafe, poly, safe)$, where $unsafe$ is the count of unsafe (narrowing) conversions, $poly$ is the count of polymorphic type bindings, and $safe$ is the sum of the degree of safe (widening) conversions. Degree of safe conversion is calculated as path weight in a directed graph of safe conversions between types; Bilson's version of this graph is in Figure 4.1a. The safe conversion graph is designed such that the common type c of two types u and v is compatible with the C standard definitions from [9, §6.3.1.8] and can be calculated as the unique type minimizing the sum of the path weights of $\overrightarrow{uc}$ and $\overrightarrow{vc}$. The following example lists the cost in the Bilson model of calling each of the following functions with two **int** parameters, where the interpretation with the minimum total cost will be selected:

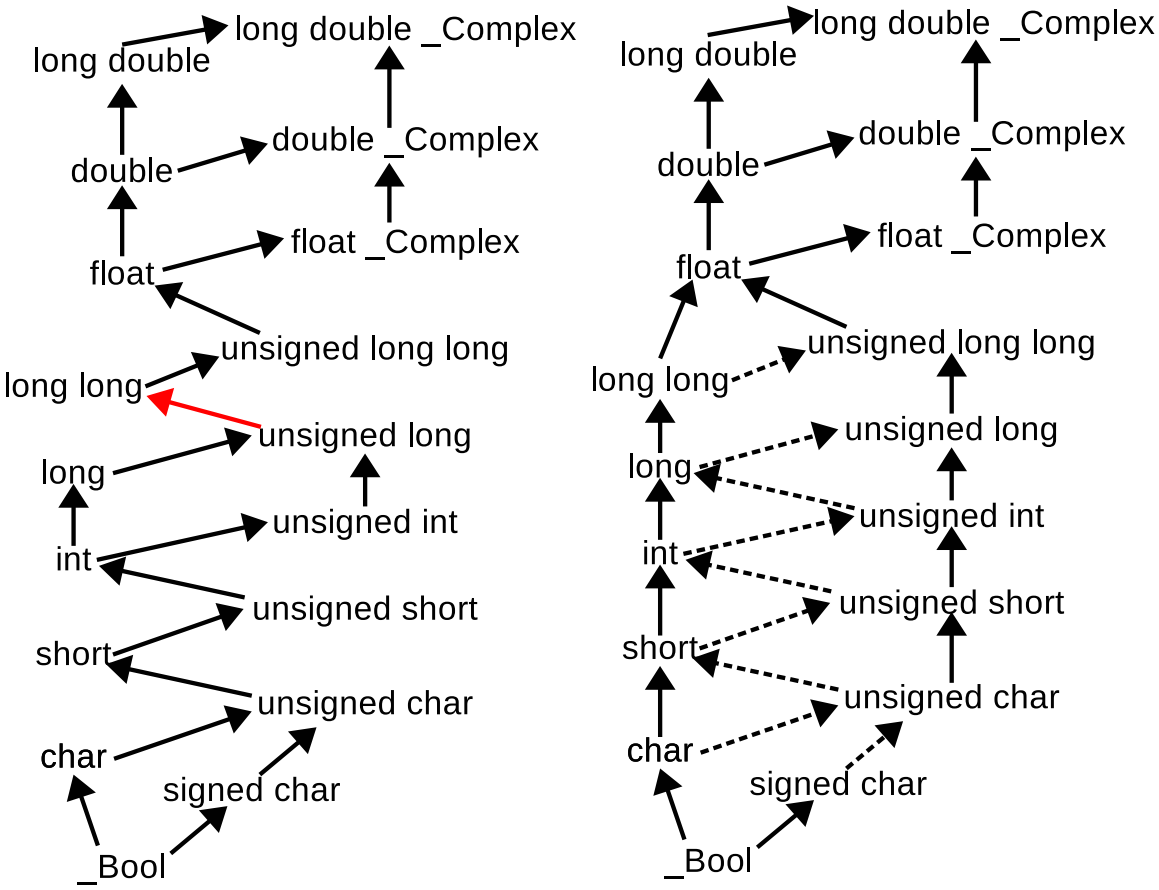
```
void f1(char, long);                    // (1,0,1)
void f2(short, long);                   // (1,0,1)
forall(otype T) void f3(T, long);       // (0,1,1)
void f4(long, long);                    // (0,0,2)
void f5(int, unsigned long);            // (0,0,2)
void f6(int, long);                     // (0,0,1)
```

Note that safe and unsafe conversions are handled differently; C∀ counts distance of safe conversions (*e.g.* **int** to **long** is cheaper than **int** to **unsigned long**), while only counting the number of unsafe conversions (*e.g.* **int** to **char** and **int** to **short** both have unsafe cost 1, as in f_1 and f_2 above). These costs are summed over the parameters in a call; in the example above, the cost of the two **int** to **long** conversions for f_4 sum equal to the one **int** to **unsigned long** conversion for f_5.

As part of adding reference types to C∀ (see Section 2.5), Schluntz added a new $reference$ element to the cost tuple, which counts the number of implicit reference-to-rvalue conversions performed so that candidate interpretations can be distinguished by how closely they match the nesting of reference types; since references are meant to act almost indistinguishably from lvalues, this $reference$ element is the least significant in the lexicographic comparison of cost tuples.

I also refined the C∀ cost model as part of this thesis work. Bilson's C∀ cost model includes the cost of polymorphic type bindings from a function's type assertions in the *poly* element of the cost tuple; this has the effect of making more-constrained functions more expensive than less-constrained functions, as in the following example, based on differing iterator types:

```
forall(dtype T | { T& ++?(T&); }) T& advance1(T& i, int n);
forall(dtype T | { T& ++?(T&); T& ?+=?(T&, int)}) T& advance2(T& i, int n);
```



(a) Bilson

(b) Extended

Figure 4.1: Safe conversion graphs. In both graphs, plain arcs have cost $safe = 1, sign = 0$ while dashed sign-conversion arcs have cost $safe = 1, sign = 1$. As per [9, §6.3.1.8], types promote to types of the same signedness with greater rank, from **signed** to **unsigned** with the same rank, and from **unsigned** to **signed** with greater size. The arc from **unsigned long** to **long long** (highlighted in red in 4.1a) is deliberately omitted in 4.1b, as on the presented system **sizeof**(**long**) == **sizeof**(**long long**).

In resolving a call to advance, the binding to the T& parameter in the assertions is added to the *poly* cost in Bilson's model. However, type assertions actually make a function *less* polymorphic, and as such functions with more type assertions should be preferred in type resolution. In the example above, if the meaning of advance is "increment i n times", $\mathsf{advance}_1$ requires an n-iteration loop, while $\mathsf{advance}_2$ can be implemented more efficiently with the ?+=? operator; as such, $\mathsf{advance}_2$ should be chosen over $\mathsf{advance}_1$ whenever its added constraint can be satisfied. Accordingly, a *specialization* element is now included in the C∀ cost tuple, the values of which are always negative. Each type assertion subtracts 1 from *specialization*, so that more-constrained functions cost less, and thus are chosen over less-constrained functions, all else being equal. A more sophisticated design would define a partial order over sets of type assertions by set inclusion (*i.e.* one function would only cost less than another if it had a strict superset of assertions, rather than just more total assertions), but I did not judge the added complexity of computing and testing this order to be worth the gain in specificity.

I also incorporated an unimplemented aspect of Ditchfield's earlier cost model. In the example below, adapted from [14, p.89], Bilson's cost model only distinguished between the first two cases by accounting extra cost for the extra set of **otype** parameters, which, as discussed above, is not a desirable solution:

```
forall(otype T, otype U) void f1(T, U);    // polymorphic
forall(otype T) void f2(T, T);             // less polymorphic
forall(otype T) void f3(T, int);           // even less polymorphic
forall(otype T) void f4(T*, int);          // least polymorphic
```

The new cost model accounts for the fact that functions with more polymorphic variables are less constrained by introducing a *var* cost element that counts the number of type variables on a candidate function. In the example above, f_1 has $var = 2$, while the others have $var = 1$.

The new cost model also accounts for a nuance unhandled by Ditchfield or Bilson, in that it makes the more specific f_4 cheaper than the more generic f_3; f_4 is presumably somewhat optimized for handling pointers, but the prior C∀ cost model could not account for the more specific binding, as it simply counted the number of polymorphic unifications. In the modified model, each level of constraint on a polymorphic type in the parameter list results in a decrement of the *specialization* cost element, which is shared with the count of assertions due to their common nature as constraints on polymorphic type bindings. Thus, all else equal, if both a binding to T and a binding to T∗ are available, the model chooses the more specific T∗ binding with $specialization = -1$. This process is recursive, such that T∗∗ has $specialization = -2$. This calculation works similarly for generic types, *e.g.* box(T) also has specialization cost -1. For multi-argument generic types, the least-

specialized polymorphic parameter sets the specialization cost, *e.g.* the specialization cost of pair(T, S∗) is −1 (from T) rather than −2 (from S). Specialization cost is not counted on the return type list; since *specialization* is a property of the function declaration, a lower specialization cost prioritizes one declaration over another. User programmers can choose between functions with varying parameter lists by adjusting the arguments, but the same is not true in general of varying return types[D], so the return types are omitted from the *specialization* element. Since both $vars$ and *specialization* are properties of the declaration rather than any particular interpretation, they are prioritized less than the interpretation-specific conversion costs from Bilson's original 3-tuple.

A final refinement I have made to the C∀ cost model is with regard to choosing among arithmetic conversions. The C standard [9, §6.3.1.8] states that the common type of **int** and **unsigned int** is **unsigned int** and that the common type of **int** and **long** is **long**, but does not provide guidance for making a choice among conversions. Bilson's `cfa-cc` uses conversion costs based off Figure 4.1a. However, Bilson's design results in inconsistent and somewhat surprising costs, with conversion to the next-larger same-sign type generally (but not always) double the cost of conversion to the **unsigned** type of the same size. In the redesign, for consistency with the approach of the usual arithmetic conversions, which select a common type primarily based on size, but secondarily on sign, arcs in the new graph are annotated with whether they represent a sign change, and such sign changes are summed in a new $sign$ cost element that lexicographically succeeds $safe$. This means that sign conversions are approximately the same cost as widening conversions, but slightly more expensive (as opposed to less expensive in Bilson's graph), so maintaining the same signedness is consistently favoured. This refined conversion graph is shown in Figure 4.1b.

With these modifications, the current C∀ cost tuple is as follows:

$$(unsafe, poly, safe, sign, vars, specialization, reference)$$

4.1.3 Expression Cost

The mapping from C∀ expressions to cost tuples is described by Bilson in [6], and remains effectively unchanged with the exception of the refinements to the cost tuple described above. Nonetheless, some salient details are repeated here for the sake of completeness.

[D] In particular, as described in Section 4.1.3, cast expressions take the cheapest valid and convertible interpretation of the argument expression, and expressions are resolved as a cast to **void**. As a result of this, including return types in the *specialization* cost means that a function with return type T∗ for some polymorphic type T would *always* be chosen over a function with the same parameter types returning **void**, even for **void** contexts, an unacceptably counter-intuitive result.

On a theoretical level, the resolver treats most expressions as if they were function calls. Operators in C∀ (both those existing in C and added features like constructors) are all modelled as function calls. In terms of the core argument-parameter matching algorithm, overloaded variables are handled the same as zero-argument function calls, aside from a different pool of candidate declarations and setup for different code generation. Similarly, an aggregate member expression a.m can be modelled as a unary function m that takes one argument of the aggregate type. Literals do not require sophisticated resolution, as in C the syntactic form of each implies their result types: 42 is **int**, `"hello"` is **char**$*$, *etc.*[E].

Since most expressions can be treated as function calls, nested function calls are the primary component of complexity in expression resolution. Each function call has an *identifier* that must match the name of the corresponding declaration, and a possibly-empty list of *arguments*. These arguments may be function call expressions themselves, producing a tree of function-call expressions to resolve, where the leaf expressions are generally nullary functions, variable expressions, or literals. A single instance of expression resolution consists of matching declarations to all the identifiers in the expression tree of a top-level expression, along with inserting any conversions and satisfying all assertions necessary for that matching. The cost of a function-call expression is the sum of the conversion costs of each argument type to the corresponding parameter and the total cost of each subexpression, recursively calculated. C∀ expression resolution must produce either the unique lowest-cost interpretation of the top-level expression, or an appropriate error message if none exists. 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 g_1 to an **int** suitable for f, for a total cost of $(1, 0, 0, 0, 0, 0, 0)$ for f($g_1(42)$). If g_2 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 g_2, for a total cost of $(0, 0, 1, 0, 0, 0, 0)$ for f($g_2(42)$); as this is cheaper, g_2 is chosen. Due to this design, all valid interpretations of subexpressions must in general be propagated to the top

[E]Struct literals (*e.g.* (S){ 1, 2, 3 } for some struct S) are a somewhat special case, as they are known to be of type S, but require resolution of the implied constructor call described in Section 2.5.2.

of the expression tree before any can be eliminated, a lazy form of expression resolution, as opposed to the eager expression resolution allowed by C or C++, where each expression can be resolved given only the resolution of its immediate subexpressions.

If there are no valid interpretations of the top-level expression, expression resolution fails and must produce an appropriate error message. If any subexpression has no valid interpretations, the process can be short-circuited and the error produced at that time. If there are multiple valid interpretations of a top-level expression, ties are broken based on the conversion cost, calculated as above. If there are multiple minimal-cost valid interpretations of a top-level expression, that expression is said to be *ambiguous*, and an error must be produced. Multiple minimal-cost interpretations of a subexpression do not necessarily imply an ambiguous top-level expression, however, as the subexpression interpretations may be disambiguated based on their return type or by selecting a more-expensive interpretation of that subexpression to reduce the overall expression cost, as in the example above.

The C∀ resolver uses type assertions to filter out otherwise-valid subexpression interpretations. An interpretation can only be selected if all the type assertions in the **forall** clause on the corresponding declaration can be satisfied with a unique minimal-cost set of satisfying declarations. Type assertion satisfaction is tested by performing type unification on the type of the assertion and the type of the declaration satisfying the assertion. That is, a declaration that satisfies a type assertion must have the same name and type as the assertion after applying the substitutions in the type environment. Assertion-satisfying declarations may be polymorphic functions with assertions of their own that must be satisfied recursively. This recursive assertion satisfaction has the potential to introduce infinite loops into the type resolution algorithm, a situation which `cfa-cc` avoids by imposing a hard limit on the depth of recursive assertion satisfaction (currently 4); this approach is also taken by C++ to prevent infinite recursion in template expansion, and has proven to be effective and not unduly restrictive of the expressive power of C∀.

Cast expressions must be treated somewhat differently than functions for backwards compatibility purposes with C. In C, cast expressions can serve two purposes, *conversion* (*e.g.* (**int**)3.14), which semantically converts a value to another value in a different type with a different bit representation, or *coercion* (*e.g.* **void**∗ p; (**int**∗)p;), which assigns a different type to the same bit value. C provides a set of built-in conversions and coercions, and user programmers are able to force a coercion over a conversion if desired by casting pointers. The overloading features in C∀ introduce a third cast semantic, *ascription* (*e.g.* **int** x; **double** x; (**int**)x;), which selects the overload that most-closely matches the cast type. However, since ascription does not exist in C due to the lack of overloadable identifiers, if a cast argument has an unambiguous interpretation as a conversion argument then it must

be interpreted as such, even if the ascription interpretation would have a lower overall cost. This is demonstrated in the following example, adapted from the C standard library:

```
unsigned long long x;
(unsigned)(x >> 32);
```

In C semantics, this example is unambiguously upcasting 32 to **unsigned long long**, performing the shift, then downcasting the result to **unsigned**, at cost $(1, 0, 3, 1, 0, 0, 0)$. If ascription were allowed to be a first-class interpretation of a cast expression, it would be cheaper to select the **unsigned** interpretation of ?>>? by downcasting x to **unsigned** and upcasting 32 to **unsigned**, at a total cost of $(1, 0, 1, 1, 0, 0, 0)$. However, this break from C semantics is not backwards compatible, so to maintain C compatibility, the C∀ resolver selects the lowest-cost interpretation of the cast argument for which a conversion or coercion to the target type exists (upcasting to **unsigned long long** in the example above, due to the lack of unsafe downcasts), using the cost of the conversion itself only as a tie-breaker. For example, in **int** x; **double** x; (**int**)x;, both declarations have zero-cost interpretations as x, but the **int** x interpretation is cheaper to cast to **int**, and is thus selected. Thus, in contrast to the lazy resolution of nested function-call expressions discussed above, where final interpretations for each subexpression are not chosen until the top-level expression is reached, cast expressions introduce eager resolution of their argument subexpressions, as if that argument was itself a top-level expression.

4.2 Resolution Algorithms

C∀ expression resolution is not, in general, polynomial in the size of the input expression, as shown in Section 4.2.1. While this theoretical result is daunting, its implications can be mitigated in practice. `cfa-cc` does not solve one instance of expression resolution in the course of compiling a program, but rather thousands; therefore, if the worst case of expression resolution is sufficiently rare, worst-case instances can be amortized by more-common easy instances for an acceptable overall runtime, as shown in Section 6.4. Secondly, while a programmer *can* deliberately generate a program designed for inefficient compilation[F], source code tends to follow common patterns. Programmers generally do not want to run the full compiler algorithm in their heads, and as such keep mental shortcuts in the form of language idioms. If the compiler can be tuned to handle idiomatic code more efficiently, then the reduction in runtime for idiomatic (but otherwise difficult) resolution instances can make a significant difference in total compiler runtime.

[F] See for instance [22], which generates arbitrarily large C++ template expansions from a fixed-size source file.

4.2.1 Worst-case Analysis

Expression resolution has a number of components that contribute to its runtime, including argument-parameter type unification, recursive traversal of the expression tree, and satisfaction of type assertions.

If the bound type for a type variable can be looked up or mutated in constant time (as asserted in Table 5.2), then the runtime of the unification algorithm to match an argument to a parameter is usually proportional to the complexity of the types being unified. In C, complexity of type representation is bounded by the most-complex type explicitly written in a declaration, effectively a small constant; in C∀, however, polymorphism can generate more-complex types:

```
forall(otype T) pair(T) wrap(T x, T y);

wrap(wrap(wrap(1, 2), wrap(3, 4)), wrap(wrap(5, 6), wrap(7, 8)));
```

To resolve the outermost wrap, the resolver must check that pair(pair(**int**)) unifies with itself, but at three levels of nesting, pair(pair(**int**)) is more complex than either pair(T) or T, the types in the declaration of wrap. Accordingly, the cost of a single argument-parameter unification is $O(d)$, where d is the depth of the expression tree, and the cost of argument-parameter unification for a single candidate for a given function call expression is $O(pd)$, where p is the number of parameters. This bound does not, however, account for the higher costs of unifying two polymorphic type variables, which may in the worst case result in a recursive unification of all type variables in the expression (as discussed in Chapter 5). Since this recursive unification reduces the number of type variables, it may happen at most once, for an added $O(p^d)$ cost for a top-level expression with $O(p^d)$ type variables.

Implicit conversions are also checked in argument-parameter matching, but the cost of checking for the existence of an implicit conversion is again proportional to the complexity of the type, $O(d)$. Polymorphism also introduces a potential expense here; for a monomorphic function there is only one potential implicit conversion from argument type to parameter type, while if the parameter type is an unbound polymorphic type-variable then any implicit conversion from the argument type could potentially be considered a valid binding for that type variable. C∀, however, requires exact matches for the bound type of polymorphic parameters, removing this problem. An interesting question for future work is whether loosening this requirement incurs a significant compiler runtime cost in practice; preliminary results from the prototype system described in Chapter 6 suggest it does not.

Considering the recursive traversal of the expression tree, polymorphism again greatly expands the worst-case runtime. Let i be the number of candidate declarations for each function call; if all of these candidates are monomorphic, then there are no more than i unambiguous interpretations of the subexpression rooted at that function call. Ambiguous minimal-cost subexpression interpretations may also be collapsed into a single *ambiguous interpretation*, as the presence of such a subexpression interpretation in the final solution is an error condition. One safe pruning operation during expression resolution is to discard all subexpression interpretations with greater-than-minimal cost for their return type, as such interpretations cannot beat the minimal-cost interpretation with their return type for the overall optimal solution. As such, with no polymorphism, each declaration can generate no more than one minimal-cost interpretation with its return type, so the number of possible subexpression interpretations is $O(i)$ (note that in C, which lacks overloading, $i \leq 1$). With polymorphism, however, a single declaration (like `wrap` above) can have many concrete return types after type variable substitution, and could in principle have a different concrete return type for each combination of argument interpretations. Calculated recursively, the bound on the total number of candidate interpretations is $O(i^{p^d})$, each with a distinct type.

Given these calculations of number of subexpression interpretations and matching costs, the upper bound on runtime for generating candidates for a single subexpression d levels up from the leaves is $O(i^{p^d} \cdot pd)$. Since there are $O(p^d)$ subexpressions in a single top-level expression, the total worst-case cost of argument-parameter matching with the overloading and polymorphism features of C∀ is $O(i^{p^d} \cdot pd \cdot p^d)$. Since the size of the expression is $O(p^d)$, letting $n = p^d$ this simplifies to $O(i^n \cdot n^2)$

This bound does not yet account for the cost of assertion satisfaction, however. C∀ uses type unification on the assertion type and the candidate declaration type to test assertion satisfaction; this unification calculation has cost proportional to the complexity of the declaration type after substitution of bound type variables; as discussed above, this cost is $O(d)$. If there are $O(a)$ type assertions on each declaration, there are $O(i)$ candidates to satisfy each assertion, for a total of $O(ai)$ candidates to check for each declaration. However, each assertion candidate may generate another $O(a)$ assertions, recursively until the assertion recursion limit r is reached, for a total cost of $O((ai)^r \cdot d)$. Now, a and i are properties of the set of declarations in scope, while r is defined by the language spec, so $(ai)^r$ is essentially a constant for purposes of expression resolution, albeit a very large one. It is not uncommon in C∀ to have functions with dozens of assertions, and common function names (*e.g.* `?{}`, the constructor) can have hundreds of overloads.

It is clear that assertion satisfaction costs can be very large, and in fact a method for heuristically reducing these costs is one of the key contributions of this thesis, but it should

be noted that the worst-case analysis is a particularly poor match for actual code in the case of assertions. It is reasonable to assume that most code compiles without errors, as an actively-developed project is compiled many times, generally with relatively few new errors introduced between compiles. However, the worst-case bound for assertion satisfaction is based on recursive assertion satisfaction calls exceeding the limit, which is an error case. In practice, then, the depth of recursive assertion satisfaction should be bounded by a small constant for error-free code, which accounts for the vast majority of problem instances.

Similarly, uses of polymorphism like those that generate the $O(d)$ bound on unification or the $O(i^{p^d})$ bound on number of candidates are rare, but not completely absent. This analysis points to type unification, argument-parameter matching, and assertion satisfaction as potentially costly elements of expression resolution, and thus profitable targets for algorithmic investigation. Type unification is discussed in Chapter 5, while the other aspects are covered below.

4.2.2 Argument-Parameter Matching

Pruning possible interpretations as early as possible is one way to reduce the real-world cost of expression resolution, provided that a sufficient proportion of interpretations are pruned to pay for the cost of the pruning algorithm. One opportunity for interpretation pruning is by the argument-parameter type matching, but the literature [5, 6, 12, 18, 36, 37] provides no clear answers on whether candidate functions should be chosen according to their available arguments, or whether argument resolution should be driven by the available function candidates. For programming languages without implicit conversions, argument-parameter matching is essentially the entirety of the expression resolution problem, and is generally referred to as "overload resolution" in the literature. All expression-resolution algorithms form a DAG of interpretations, some explicitly, some implicitly; in this DAG, arcs point from function-call interpretations to argument interpretations, as in Figure 4.2

Note that some interpretations may be part of more than one super-interpretation, as with the p_2 interpretation of p_B, while some valid subexpression interpretations, like the f_2 interpretation of f_B, are not used in any interpretation of their superexpression.

Overload resolution was first seriously considered in the development of compilers for the Ada programming language, with different algorithms making various numbers of passes over the expression DAG, these passes being either top-down or bottom-up. Baker's algorithm [5] takes a single pass from the leaves of the expression tree up, pre-computing argument candidates at each step. For each candidate function, Baker attempts to match argument types to parameter types in sequence, failing if any parameter cannot be matched.

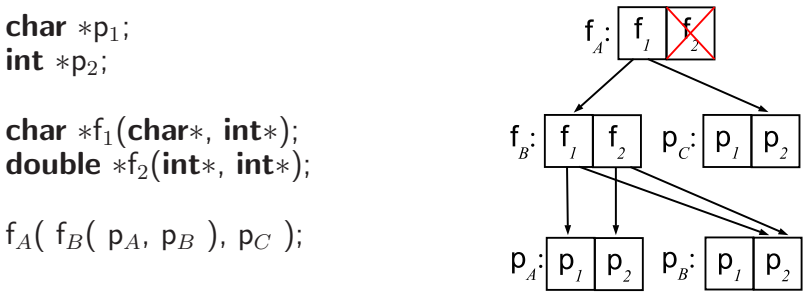


Figure 4.2: Resolution DAG for a simple expression, annotated with explanatory subscripts. Functions that do not have a valid argument matching are covered with an X.

Bilson [6] similarly pre-computes argument-candidates in a single bottom-up pass in the original `cfa-cc`, but then explicitly enumerates all possible argument combinations for a multi-parameter function. These argument combinations are matched to the parameter types of the candidate function as a unit rather than individual arguments. Bilson's approach is less efficient than Baker's, as the same argument may be compared to the same parameter many times, but does allow a more straightforward handling of polymorphic type-binding and tuple-typed expressions.

Unlike Baker and Bilson, Cormack's algorithm [12] requests argument candidates that match the type of each parameter of each candidate function, in a single pass from the top-level expression down; memoization of these requests is presented as an optimization. As presented, this algorithm requires the parameter to have a known type, which is a poor fit for polymorphic type parameters in C∀. Cormack's algorithm can be modified to request argument interpretations of *any* type when provided an unbound parameter type, but this eliminates any pruning gains that could be provided by the algorithm.

Ganzinger and Ripken [18] propose an approach (later refined by Pennello *et al.* [36]) that uses a top-down filtering pass followed by a bottom-up filtering pass to reduce the number of candidate interpretations; they prove that a small number of such iterations is sufficient to converge to a solution for the overload resolution problem in the Ada programming language. Persch *et al.* [37] developed a similar two-pass approach where the bottom-up pass is followed by the top-down pass. These approaches differ from Baker, Bilson, or Cormack in that they take multiple passes over the expression tree to yield a solution by applying filtering heuristics to all expression nodes. This approach of filtering out invalid types is unsuited to C∀ expression resolution, however, due to the presence of polymorphic functions and implicit conversions.

Some other language designs solve the matching problem by forcing a bottom-up or-

der. C++, for instance, defines its overload-selection algorithm in terms of a partial order between function overloads given a fixed list of argument candidates, implying that the arguments must be selected before the function. This design choice improves worst-case expression resolution time by only propagating a single candidate for each subexpression, but type annotations must be provided for any function call that is polymorphic in its return type, and these annotations are often redundant:

```
template<typename T> T* malloc() { /* ... */ }

int* p = malloc<int>();              // T = int must be explicitly supplied
```

C∀ saves programmers from redundant annotations with its richer inference:

```
forall(dtype T | sized(T)) T* malloc();

int* p = malloc();                   // Infers T = int from left-hand side
```

Baker [5] left empirical comparison of different overload resolution algorithms to future work; Bilson [6] described an extension of Baker's algorithm to handle implicit conversions and polymorphism, but did not further explore the space of algorithmic approaches to handle both overloaded names and implicit conversions. This thesis closes that gap in the literature by performing performance comparisons of both top-down and bottom-up expression resolution algorithms, with results reported in Chapter 6.

4.2.3 Assertion Satisfaction

The assertion satisfaction algorithm designed by Bilson [6] for the original `cfa-cc` is the most-relevant prior work to this project. Before accepting any subexpression candidate, Bilson first checks that that candidate's assertions can all be resolved; this is necessary due to Bilson's addition of assertion satisfaction costs to candidate costs (discussed in Section 4.1.2). If this subexpression interpretation ends up not being used in the final resolution, then the (sometimes substantial) work of checking its assertions ends up wasted. Bilson's assertion checking function recurses on two lists, need and newNeed, the current declaration's assertion set and those implied by the assertion-satisfying declarations, respectively, as detailed in the pseudo-code below (ancillary aspects of the algorithm are omitted for clarity):

```
List(List(Declaration)) checkAssertions(
        List(Assertion) need, List(Assertion) newNeed, List(Declaration) have,
        Environment env ) {
    if ( is_empty(need) ) {
```

```
        if ( is_empty( newNeed ) ) return { have };
        else return checkAssertions( newNeed, {}, have, env );
    }

    Assertion a = head(need);
    Type adjType = substitute( a.type, env );
    List(Declaration) candidates = decls_matching( a.name );
    List(List(Declaration)) alternatives = {}
    for ( Declaration c : candidates ) {
        Environment newEnv = env;
        if ( unify( adjType, c.type, newEnv ) ) {
            append( alternatives,
                checkAssertions(
                    tail(need), append(newNeed, c.need), append(have, c), newEnv ) );
        }
    }
    return alternatives;
}
```

One shortcoming of this approach is that if an earlier assertion has multiple valid candidates, later assertions may be checked many times due to the structure of the recursion. Satisfying declarations for assertions are not completely independent of each other, since the unification process may produce new type bindings in the environment, and these bindings may not be compatible between independently-checked assertions. Nonetheless, with the environment data-structures discussed in Chapter 5, I have found it more efficient to produce a list of possibly-satisfying declarations for each assertion once, then check their respective environments for mutual compatibility when combining lists of assertions together.

Another improvement I have made to the assertion resolution scheme in `cfa-cc` is to consider all assertion-satisfying combinations at one level of recursion before attempting to recursively satisfy any newNeed assertions. Monomorphic functions are cheaper than polymorphic functions for assertion satisfaction because they are an exact match for the environment-adjusted assertion type, whereas polymorphic functions require an extra type binding. Thus, if there is any mutually-compatible set of assertion-satisfying declarations that does not include any polymorphic functions (and associated recursive assertions), then the optimal set of assertions does not require any recursive newNeed satisfaction. More generally, due to the C∀ cost-model changes I introduced in Section 4.1.2, the conversion cost of an assertion-satisfying declaration is no longer dependent on the conversion cost of its own assertions. As such, all sets of mutually-compatible assertion-satisfying declarations

can be sorted by their summed conversion costs, and the recursive newNeed satisfaction pass is required only to check the feasibility of the minimal-cost sets. This optimization significantly reduces wasted work relative to Bilson's approach, as well as avoiding generation of deeply-recursive assertion sets, for a significant performance improvement relative to Bilson's `cfa-cc`.

Making the conversion cost of an interpretation independent of the cost of satisfying its assertions has further benefits. Bilson's algorithm checks assertions for all subexpression interpretations immediately, including those that are not ultimately used; I have developed a *deferred* variant of assertion checking that waits until a top-level interpretation has been generated to check any assertions. If the assertions of the minimal-cost top-level interpretation cannot be satisfied then the next-most-minimal-cost interpretation's assertions are checked, and so forth until a minimal-cost satisfiable interpretation (or ambiguous set thereof) is found, or no top-level interpretations are found to have satisfiable assertions. In the common case where the code actually does compile, this saves the work of checking assertions for ultimately-rejected interpretations, though it does rule out some pruning opportunities for subinterpretations with unsatisfiable assertions or which are more expensive than a minimal-cost polymorphic function with the same return type. The experimental results in Chapter 6 indicate that this is a worthwhile trade-off.

Optimizing assertion satisfaction for common idioms has also proved effective in C∀; the code below is an unexceptional print statement derived from the C∀ test suite that nonetheless is a very difficult instance of expression resolution:

```
sout | "one" | 1 | "two" | 2 | "three" | 3 | "four" | 4 | "five" | 5 | "six" | 6
    | "seven" | 7 | "eight" | 8 | "nine" | 9 | "ten" | 10 | "end" | nl | nl;
```

The first thing that makes this expression so difficult is that it is 23 levels deep; Section 4.2.1 indicates that the worst-case bounds on expression resolution are exponential in expression depth. Secondly, the ?|? operator is significantly overloaded in C∀ — there are 74 such operators in the C∀ standard library, and while 9 are arithmetic operators inherited from C, the rest are polymorphic I/O operators that look similar to:

```
forall( dtype ostype | ostream( ostype ) )
ostype& ?|? ( ostype&, int );
```

Note that ostream is a trait with 25 type assertions, and that the output operators for the other arithmetic types are also valid for the **int**-type parameters due to implicit conversions. On this instance, deferred assertion satisfaction saves wasted work checking assertions on the wrong output operators, but does nothing about the 23 repeated checks of the 25 assertions to determine that ofstream (the type of sout) satisfies ostream.

To solve this problem, I have developed a *cached* variant of assertion checking. During

the course of checking the assertions of a single top-level expression, the results are cached for each assertion checked. The search key for this cache is the assertion declaration with its type variables substituted according to the type environment to distinguish satisfaction of the same assertion for different types. This adjusted assertion declaration is then run through the CFA name-mangling algorithm to produce an equivalent string-type key.

One superficially-promising optimization, which I did not pursue, is caching assertion-satisfaction judgments among top-level expressions. This approach would be difficult to correctly implement in a CFA compiler, due to the lack of a closed set of operations for a given type. New declarations related to a particular type can be introduced in any lexical scope in CFA, and these added declarations may cause an assertion that was previously satisfiable to fail due to an introduced ambiguity. Furthermore, given the recursive nature of assertion satisfaction and the possibility of this satisfaction judgment depending on an inferred type, an added declaration may break satisfaction of an assertion with a different name and that operates on different types. Given these concerns, correctly invalidating a cross-expression assertion satisfaction cache for CFA is a non-trivial problem, and the overhead of such an approach may possibly outweigh any benefits from such caching.

The assertion satisfaction aspect of CFA expression resolution bears some similarity to satisfiability problems from logic, and as such other languages with similar trait and assertion mechanisms make use of logic-program solvers in their compilers. For instance, Matsakis [26] and the Rust team have developed a PROLOG-based engine to check satisfaction of Rust traits. The combination of the assertion satisfaction elements of the problem with the conversion-cost model of CFA makes this logic-solver approach difficult to apply in `cfa-cc`, however. Expressing assertion resolution as a satisfiability problem ignores the cost optimization aspect, which is necessary to decide among what are often many possible satisfying assignments of declarations to assertions. (MaxSAT solvers [29], which allow weights on solutions to satisfiability problems, may be a productive avenue for future investigation.) On the other hand, the deeply-recursive nature of the satisfiability problem makes it difficult to adapt to optimizing solver approaches such as linear programming. To maintain a well-defined programming language, any optimization algorithm used must provide an exact (rather than approximate) solution; this constraint also rules out a whole class of approximately-optimal generalized solvers. As such, I opted to continue Bilson's approach of designing a bespoke solver for CFA assertion satisfaction, rather than attempting to re-express the problem in some more general formalism.

4.3 Conclusion & Future Work

As the results in Chapter 6 show, the algorithmic approaches I have developed for C∀ expression resolution are sufficient to build a practically-performant C∀ compiler. This work may also be of use to other compiler construction projects, notably to members of the C++ community as they implement the new Concepts [10] standard, which includes type assertions similar to those used in C∀, as well as the C-derived implicit conversion system already present in C++.

I have experimented with using expression resolution rather than type unification to check assertion satisfaction; this variant of the expression resolution problem should be investigated further in future work. This approach is more flexible than type unification, allowing for conversions to be applied to functions to satisfy assertions. Anecdotally, this flexibility matches user-programmer expectations better, as small type differences (*e.g.* the presence or absence of a reference type, or the usual conversion from **int** to **long**) no longer break assertion satisfaction. Practically, the resolver prototype discussed in Chapter 6 uses this model of assertion satisfaction, with no apparent deficit in performance; the generated expressions that are resolved to satisfy the assertions are easier than the general case because they never have nested subexpressions, which eliminates much of the theoretical differences between unification and resolution. The main challenge to implement this approach in `cfa-cc` is applying the implicit conversions generated by the resolution process in the code-generation for the thunk functions that `cfa-cc` uses to pass type assertions to their requesting functions with the proper signatures.

One C∀ feature that could be added to improve the ergonomics of overload selection is an *ascription cast*; as discussed in Section 4.1.3, the semantics of the C cast operator are to choose the cheapest argument interpretation which is convertible to the target type, using the conversion cost as a tie-breaker. An ascription cast would reverse these priorities, choosing the argument interpretation with the cheapest conversion to the target type, only using interpretation cost to break ties[G]. This would allow ascription casts to the desired return type to be used for overload selection:

```
int f1(int);
int f2(double);
int g1(int);
double g2(long);

f((double)42);          // select f2 by argument cast
```

[G] A possible stricter semantics would be to select the cheapest interpretation with a zero-cost conversion to the target type, reporting a compiler error otherwise.

```
(as double)g(42);      // select g2 by return ascription cast
(double)g(42);         // select g1 NOT g2 because of parameter conversion cost
```

Though performance of the existing resolution algorithms is promising, some further optimizations do present themselves. The refined cost model discussed in Section 4.1.2 is more expressive, but requires more than twice as many fields; it may be fruitful to investigate more tightly-packed in-memory representations of the cost-tuple, as well as comparison operations that require fewer instructions than a full lexicographic comparison. Integer or vector operations on a more-packed representation may prove effective, though dealing with the negative-valued *specialization* field may require some effort.

Parallelization of various phases of expression resolution may also be useful. The algorithmic variants I have introduced for both argument-parameter matching and assertion satisfaction are essentially divide-and-conquer algorithms, which solve subproblem instances for each argument or assertion, respectively, then check mutual compatibility of the solutions. While the checks for mutual compatibility are naturally more serial, there may be some benefit to parallel resolution of the subproblem instances.

The resolver prototype built for this project and described in Chapter 6 would be a suitable vehicle for many of these further experiments, and thus a technical contribution of continuing utility.

Chapter 5

Type Environment

One key data structure for expression resolution is the *type environment*. As discussed in Chapter 4, being able to efficiently determine which type variables are bound to which concrete types or whether two type environments are compatible is a core requirement of the resolution algorithm. Furthermore, expression resolution involves a search through many related possible solutions, so the ability to re-use shared subsets of type-environment data and to switch between environments quickly is desirable for performance. In this chapter, I discuss a number of type-environment data-structure variants, including some novel variations on the union-find [17] data structure introduced in this thesis. Chapter 6 contains empirical comparisons of the performance of these data structures when integrated into the resolution algorithm.

5.1 Definitions

For purposes of this chapter, a *type environment* T is a set of *type classes* $\left\{T_1, T_2, \cdots, T_{|T|}\right\}$. Each type class T_i contains a set of *type variables* $\left\{v_{i,1}, v_{i,2}, \cdots, v_{i,|T_i|}\right\}$. Since the type classes represent an equivalence relation over the type variables the sets of variables contained in two distinct classes in the same environment must be *disjoint*. Each individual type class T_i may also be associated with a *bound*, b_i; this bound contains the *bound type* that the variables in the type class are replaced with, but also includes other information in `cfa-cc`, including whether type conversions are permissible on the bound type and what sort of type variables are contained in the class (data types, function types, or variadic tuples).

Table 5.1: Summary of type environment operations.

$find(T, v_{i,j}) \rightarrow T_i \mid \mathsf{fail}$	Locate class for variable
$report(T_i) \rightarrow \{v_{i,j} \cdots\}$	List variables for class
$bound(T_i) \rightarrow b_i \mid \mathsf{fail}$	Get bound for class
$insert(T, v_{i,1})$	New single-variable class
$add(T_i, v_{i,j})$	Add variable to class
$bind(T_i, b_i)$	Set or update class bound
$unify(T, T_i, T_j) \rightarrow \mathsf{pass} \mid \mathsf{fail}$	Combine two type classes
$split(T, T_i) \rightarrow T'$	Revert the last $unify$ operation on T_i
$combine(T, T') \rightarrow \mathsf{pass} \mid \mathsf{fail}$	Merge two environments
$save(T) \rightarrow H$	Get handle for current state
$backtrack(T, H)$	Return to handle state

The following example demonstrates the use of a type environment for unification:

```
forall(otype F) F f(F, F);
forall(otype G) G g(G);

f( g(10), g(20) );
```

Expression resolution starts from an empty type environment; from this empty environment, the calls to g can be independently resolved. These resolutions result in two new type environments, $T = \{\{\mathsf{G}_1\} \rightarrow \mathbf{int}\}$ and $T' = \{\{\mathsf{G}_2\} \rightarrow \mathbf{int}\}$; the calls to g have generated distinct type variables G_1 and G_2, each bound to **int** by unification with the type of its argument (10 and 20, both **int**). To complete resolution of the call to f, both environments must be combined; resolving the first argument to f produces a new type environment $T'' = \{\{\mathsf{G}_1, \mathsf{F}_1\} \rightarrow \mathbf{int}\}$: the new type variable F_1 has been introduced and unified with G_1 (the return type of g(10)), and consequently bound to **int**. To resolve the second argument to f, T'' must be checked for compatibility with T'; since F_1 unifies with G_2, their type classes must be merged. Since both F_1 and G_2 are bound to **int**, this merge succeeds, producing the final environment $T'' = \{\{\mathsf{G}_1, \mathsf{F}_1, \mathsf{G}_2\} \rightarrow \mathbf{int}\}$.

Type environments in `cfa-cc` need to support eleven basic operations, summarized in Table 5.1. The first six operations are straightforward queries and updates on these data structures: The lookup operation $find(T, v_{i,j})$ produces T_i, the type class in T that contains variable $v_{i,j}$, or an invalid sentinel value for no such class. The other two query operations act on type classes, where $report(T_i)$ produces the set $\left\{v_{i,1}, v_{i,2}, \cdots, v_{i,|T_i|}\right\}$ of all type variables in a class T_i and $bound(T_i)$ produces the bound b_i of that class, or a sentinel indicating no bound is set.

The update operation $insert(T, v_{i,1})$ creates a new type class T_i in T that contains only the variable $v_{i,1}$ and no bound; due to the disjointness property, $v_{i,1}$ must not belong to any other type class in T. The $add(T_i, v_{i,j})$ operation adds a new type variable $v_{i,j}$ to class T_i; again, $v_{i,j}$ cannot exist elsewhere in T. $bind(T_i, b_i)$ mutates the bound for a type class, setting or updating the current bound.

The $unify$ operation is the fundamental non-trivial operation a type-environment data-structure must support. $unify(T, T_i, T_j)$ merges a type class T_j into another T_i, producing a failure result and leaving T in an invalid state if this merge fails. It is always possible to unify the type variables of both classes by simply taking the union of both sets; given the disjointness property, no checks for set containment are required, and the variable sets can simply be concatenated if supported by the underlying data structure. $unify$ depends on an internal $unifyBound$ operation, which may fail. In `cfa-cc`, $unifyBound(b_i, b_j) \rightarrow b_i' \mid \mathsf{fail}$ checks that the type classes contain the same sort of variable, takes the tighter of the two conversion permissions, and checks if the bound types can be unified. If the bound types cannot be unified (*e.g.* **struct** A with **int***), then $unifyBound$ fails, while other combinations of bound types may result in recursive calls. For instance, unifying R* with S* for type variables R and S results in a call to $unify(T, find(\mathsf{R}), find(\mathsf{S}))$, while unifying R* with **int*** results in a call to $unifyBound$ on **int** and the bound type of the class containing R. As such, a call to $unify(T, T_i, T_j)$ may touch every type class in T, not just T_i and T_j, collapsing the entirety of T into a single type class in extreme cases. For more information on C∀ unification, see [6]. The inverse of $unify$ is $split(T, T_i)$, which produces a new environment T' that is the same as T except that T_i has been replaced by two classes corresponding to the arguments to the previous call to $unify$ on T_i. If there is no prior call to $unify$ on T_i (*i.e.* T_i is a single-element class) T_i is absent in T'.

Given the nature of the expression resolution problem as a backtracking search, caching and concurrency are both useful tools to decrease runtime. However, both of these approaches may produce multiple distinct descendants of the same initial type environment, which have possibly been mutated in incompatible ways. As such, to effectively employ either caching or concurrency, the type environment data structure must support an efficient method to check if two type environments are compatible and merge them if so. $combine(T, T')$ attempts to merge an environment T' into another environment T, producing pass if successful or leaving T in an invalid state and producing fail otherwise. The invalid state of T on failure is not important, given that a combination failure results in the resolution algorithm backtracking to a different environment. *combine* proceeds by calls to *insert*, *add*, and $unify$ as needed, and can be roughly thought of as calling $unify$ on every pair of classes in T that have variables $v_{i,j}'$ and $v_{i,k}'$ in the same class T_i' in T'. Like $unify$, *combine* can always find a mutually-consistent partition of type variables into

classes (in the extreme case, all type variables from T and T' in a single type class), but may fail due to inconsistent bounds on merged type classes.

Finally, the backtracking access patterns of the compiler can be exploited to reduce memory usage or runtime through use of an appropriately designed data structure. The set of mutations to a type environment across the execution of the resolution algorithm produce an implicit tree of related environments, and the backtracking search typically focuses only on one leaf of the tree at once, or at most a small number of closely-related nodes as arguments to *combine*. As such, the ability to save and restore particular type environment states is useful, and supported by the $save(T) \to H$ and $backtrack(T, H)$ operations, which produce a handle for the current environment state and mutate an environment back to a previous state, respectively. These operations can be naively implemented by a deep copy of T into H and vice versa, but have more efficient implementations in persistency-aware data structures such as the persistent union-find introduced in Section 5.2.5.

5.2 Approaches

5.2.1 Naïve

The type environment data structure used in Bilson's [6] original implementation of `cfa-cc` is a simple translation of the definitions in Section 5.1 to C++ code; a TypeEnvironment contains a list of EqvClass type equivalence classes, each of which contains the type bound information and a tree-based sorted set of type variables. This approach has the benefit of being easy to understand and not imposing life-cycle or inheritance constraints on its use, but, as can be seen in Table 5.2, does not support many of the desired operations with any particular efficiency. Some variations on this structure may improve performance somewhat; for instance, replacing the EqvClass variable storage with a hash-based set reduces search and update times from $O(\log n)$ to amortized $O(1)$, while adding an index for the type variables in the entire environment removes the need to check each type class individually to maintain the disjointness property. These improvements do not change the fundamental issues with this data structure, however.

5.2.2 Incremental Inheritance

One more invasive modification to this data structure that I investigated is to support swifter combinations of closely-related environments in the backtracking tree by storing a

reference to a *parent* environment within each environment, and having that environment only store type classes that have been modified with respect to the parent. This approach provides constant-time copying of environments, as a new environment simply consists of an empty list of type classes and a reference to its (logically identical) parent; since many type environments are no different than their parent, this speeds backtracking in this common case. Since all mutations made to a child environment are by definition compatible with the parent environment, two descendants of a common ancestor environment can be combined by iteratively combining the changes made in one environment, then that environment's parent, until the common ancestor is reached, again re-using storage and reducing computation in many cases.

For this environment, I also employed a lazily-generated index of type variables to their containing class, which could be in either the current environment or an ancestor. Any mutation of a type class in an ancestor environment causes that class to be copied into the current environment before mutation, as well as added to the index, ensuring all local changes to the type environment are listed in its index. However, not adding type variables to the index until lookup or mutation preserves the constant-time environment copy operation in the common case in which the copy is not mutated from its parent during its life-cycle.

This approach imposes some performance penalty on *combine* if related environments are not properly linked together, as the entire environment needs to be combined rather than just the difference, but is correct as long as the "null parent" base-case is properly handled. The life-cycle issues are somewhat more complex, as many environments may descend from a common parent, and all of these need their parent to stay alive for purposes of lookup. These issues can be solved by "flattening" parent nodes into their children before the parent's scope ends, but given the tree structure of the inheritance graph it is more straightforward to store the parent nodes in reference-counted or otherwise automatically garbage-collected heap storage.

5.2.3 Union-Find

Given the nature of the classes of type variables as disjoint sets, another natural approach to implementing a type environment is the union-find disjoint-set data-structure [17]. Union-find efficiently implements two operations over a partition of a collection of elements into disjoint sets; $find(x)$ locates the *representative* of x, the element which canonically names its set, while $union(r, s)$ merges two sets represented by r and s, respectively. The union-find data structure is based on providing each element with a reference to its parent element,

such that the root of a tree of elements is the representative of the set of elements contained in the tree. $find$ is then implemented by a search up to the parent, generally combined with a *path compression* step that links nodes more directly to their ancestors to speed up subsequent searches. *union* involves making the representative of one set a child of the representative of the other, generally employing a rank- or size-based heuristic to ensure that the tree remains somewhat balanced. If both path compression and a balancing heuristic are employed, both *union* and $find$ run in amortized $O(\alpha(n))$ worst-case time; this inverse Ackermann bound is a small constant for all practical values of n [45].

The union-find $find$ and *union* operations have obvious applicability to the $find$ and $unify$ type environment operations in Table 5.1, but the union-find data structure must be augmented to fully implement the type environment operations. In particular, the type-class bound cannot be easily included in the union-find data structure, as the requirement to make it the class representative breaks the balancing properties of *union*, and requires too-close integration of the type environment $unifyBound$ internal operation. This issue can be solved by including a side map from class representatives to the type-class bound. If placeholder values are inserted in this map for type classes without bounds then this also has the useful property that the key set of the map provides an easily obtainable list of all the class representatives, a list which cannot be derived from the union-find data structure without a linear search for class representatives through all elements.

5.2.4 Union-Find with Classes

Another type environment operation not supported directly by the union-find data structure is *report*, which lists the type variables in a given class, and similarly *split*, which reverts a $unify$ operation. Since the union-find data structure stores only links from children to parents and not vice-versa, there is no way to reconstruct a class from one of its elements without a linear search over the entire data structure, with $find$ called on each element to check its membership in the class. The situation is even worse for the *split* operation, which requires extra information to maintain the order that each child is added to its parent node. Unfortunately, the literature [44, 16, 35] on union-find does not present a way to keep references to children without breaking the asymptotic time bounds of the algorithm; I have discovered a method to do so, which, despite its simplicity, seems to be novel.

The core idea of this "union-find with classes" data structure and algorithm is to keep the members of each class stored in a circularly-linked list. Aho, Hopcroft, and Ullman also include a circularly-linked list in their 1974 textbook [2]. However, the algorithm presented

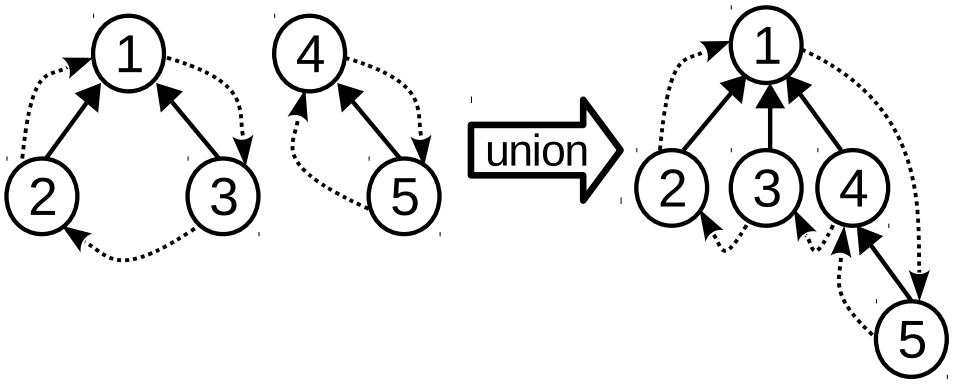


Figure 5.1: Union operation for union-find with classes. Solid lines indicate parent pointers, dashed lines are next pointers.

by Aho *et al.* has an entirely flat class hierarchy, where all elements are direct children of the representative, giving constant-time $find$ at the cost of linear-time *union* operations. In my version, the list data structure does not affect the layout of the union-find tree, maintaining the same asymptotic bounds as union-find. In more detail, each element is given a next pointer to another element in the same class; this next pointer initially points to the element itself. When two classes are unified, the next pointers of the representatives of those classes are swapped, splicing the two circularly-linked lists together as illustrated in Figure 5.1. Importantly, though this approach requires an extra pointer per element, it does maintain the linear space bound of union-find, and because it only requires updating the two root nodes in *union* it does not asymptotically increase runtime either. The basic approach is compatible with all path-compression techniques, and allows the members of any class to be retrieved in time linear in the size of the class simply by following the next pointers from any element.

If the path-compression optimization is abandoned, union-find with classes also encodes a reversible history of all the *union* operations applied to a given class. Theorem 1 demonstrates that the next pointer of the representative of a class always points to a leaf from the last-added subtree. This property is sufficient to reverse the most-recent *union* operation by finding the ancestor of that leaf that is an immediate child of the representative, breaking its parent link, and swapping the next pointers back[A]. Once the *union* operation has been reversed, Theorem 1 still holds for the reduced class, and the process can be repeated recursively until the entire set is split into its component elements.

[A]Union-by-size may be a more appropriate approach than union-by-rank in this instance, as adding two known sizes is a reversible operation, but the rank increment operation cannot be reliably reversed.

Theorem 1. *The* next *pointer of a class representative in the union-find with classes algorithm, without path compression, points to a leaf from the most-recently-added subtree.*

Proof. By induction on the height of the tree.
Base case: A height 1 tree by definition includes only a single item. In such a case, the representative's next pointer points to itself by construction, and the representative is the most-recently-added (and only) leaf in the tree.
Inductive case: By construction, a tree T of height greater than 1 has children of the root (representative) node that were representative nodes of classes merged by *union*. By definition, the most-recently-added subtree T' has a smaller height than T, thus by the inductive hypothesis before the most-recent *union* operation, the next pointer of the root of T' pointed to one of the leaf nodes of T'; by construction the next pointer of the root of T points to this leaf after the *union* operation. □

On its own, union-find, like the naïve approach, has no special constraints on life-cycle or inheritance, but it can be used as a building block in more sophisticated type environment data structures.

5.2.5 Persistent Union-Find

Given the backtracking nature of the resolution algorithm discussed in Section 5.1, the abilities to quickly switch between related versions of a type environment and to de-duplicate shared data among environments are both assets to performance. Conchon and Filliâtre [11] present a persistent union-find data structure based on the persistent array of Baker [4, 3].

In Baker's persistent array, an *array reference* contains either a pointer to the array or a pointer to an *edit node*; these edit nodes contain an array index, the value in that index, and another array reference pointing either to the array or a different edit node. By construction, these array references always point to a node more like the actual array, forming a tree of edits rooted at the actual array. Reads from the actual array at the root can be performed in constant time, as with a non-persistent array. The persistent array can be mutated in constant time by directly modifying the underlying array, then replacing its array reference with an edit node containing the mutated index, the previous value at that index, and a reference to the mutated array. If the current array reference is not the root, mutation consists simply of constructing a new edit node encoding the change and referring to the current array reference.

The mutation algorithm at the root is a special case of the key operation on persistent arrays, *reroot*. A rerooting operation takes any array reference and makes it the root node of the array. This operation is accomplished by tracing the path from some edit node to actual array at the root node, recursively applying the edits to the underlying array and replacing each edit node's successor with the inverse edit. In this way, any previous state of the persistent array can be restored in time proportional to the number of edits to the current state of the array. While *reroot* does maintain the same value mapping in every version of the persistent array, the internal mutations it performs break thread-safety, and thus it must be used behind a lock in a concurrent context. Also, the root node with the actual array may in principle be anywhere in the tree, and does not provide information to report its leaf nodes, so some form of automatic garbage collection is generally required for the data structure. Since the graph of edit nodes is tree-structured, reference counting approaches suffice for garbage collection; Conchon and Filliâtre [11] also observe that if the only *reroot* operations are for backtracking then the tail of inverse edit nodes may be elided, suggesting the possibility of stack-based memory management.

While Conchon and Filliâtre [11] implement their persistent union-find data structure over a universe of integer elements in the fixed range $[1, N]$, the type environment problem needs more flexibility. In particular, an arbitrary number of type variables may be added to the environment. As such, a persistent hash table is a more suitable structure than a persistent array, providing the same expected asymptotic time bounds, while allowing a dynamic number of elements. Besides replacing the underlying array with a hash table, the other major change in this approach is to replace the two types of array references, Array and Edit, with four node types, Table, Edit, Add, and Remove, where Add adds a new key-value pair, Remove removes a key-value pair, and Edit mutates an existing key-value pair. In this variant of `cfa-cc`, this persistent hash-table is used as the side map discussed in Section 5.2.3 for class bounds. The actual union-find data structure is slightly modified from this approach, with a Base node containing the root union-find data structure, Add nodes adding new elements, AddTo nodes defining the union of two type classes, and Remove and RemoveFrom nodes as inverses of the previous two elements, for purposes of maintaining the edit list. Figure 5.2 demonstrates the structure of a simple example. Making AddTo and RemoveFrom single nodes provides semantic information missing from the raw array updates in Conchon and Filliâtre's data structure. RemoveFrom is implemented using the "leaf of last union" approach discussed in Section 5.2.4; this does, however, preclude the use of path-compression algorithms in this persistent union-find data structure.

This added semantic information on *union* operations in the persistent union-find edit tree exposes a new option for combining type environments. If the type environments are part of the same edit tree, one environment T' can be combined with another T by only

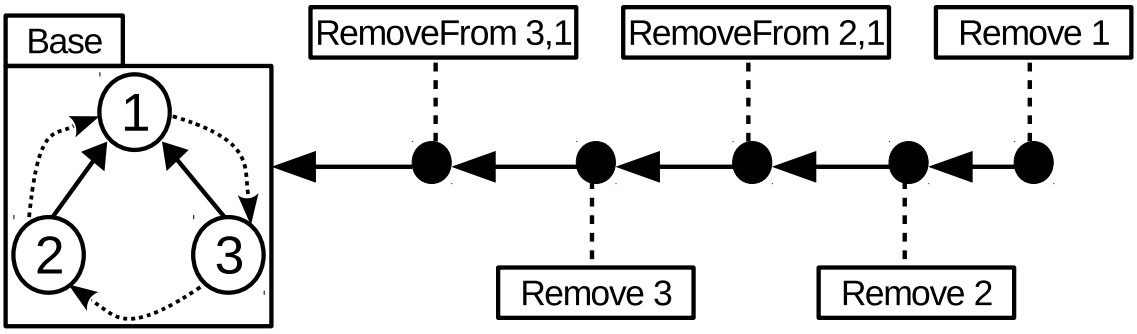


Figure 5.2: Persistent union-find data structure. Shows the edit nodes to reverse back to an empty structure.

testing the edits on the path from T' to T in both the persistent union-find data structure describing the classes and the persistent hash table containing the class bounds. This approach is generally more efficient than testing the compatibility of all type classes in T', as only those that are actually different than those in T must be considered. However, the improved performance comes at the cost of some flexibility, as the edit-tree link must be maintained between any two environments to be combined under this algorithm.

The procedure for $combine(T, T')$ based on edit paths is as follows: The shared edit trees for classes and bindings are rerooted at T, and the path from T' to T is followed to create a list of actual edits. By tracking the state of each element, redundant changes such as an Edit followed by an Edit can be reduced to their form in T' by dropping the later (more like T) Edit for the same key; Add and Remove cancel similarly. This procedure is repeated for both the class edit-tree and the binding edit-tree. When the list of net changes to the environment is produced, the additive changes are applied to T. For example, if a type class exists in T' but not T, the corresponding Add edit is applied to T, but in the reverse situation the Remove edit is not applied to T, as the intention is to produce a new environment representing the union of the two sets of type classes; similarly, AddTo edits are applied to unify type-classes in T that are united in T', but RemoveFrom edits that split type classes are not. A new environment, T'', can always be constructed with a consistent partitioning of type variables; in the extreme case, all variables from both T and T' are united in a single type class in T''. *combine* can fail to unify the bound types; if any class in T' has a class bound that does not unify with the merged class in T'', then *combine* fails.

Table 5.2: Worst-case analysis of type environment operations. n is the number of type classes, m the maximum size of a type class, and p the edit distance between two environments or a single environment and the empty environment; $u(n)$ captures the recursive cost of class unification.

	Naïve	**Incremental**	**Union-Find**	**Persistent U-F**
find	$O(n)$	$O(p)$	$O(\alpha(m))$	$O(\log m)$
report	$O(m)$	$O(m)$	$O(nm\alpha(m))$	$O(m)$
bound	$O(1)$	$O(1)$	$O(1)$	$O(1)$
insert	$O(1)$	$O(1)$	$O(1)$	$O(1)$
add	$O(1)$	$O(m)$	$O(1)$	$O(1)$
bind	$O(1)$	$O(1)$	$O(1)$	$O(1)$
unify	$O(m + u(n))$	$O(m + u(n))$	$O(1 + u(n))$	$O(1 + u(n))$
split	—	—	—	$O(\log m)$
combine	$O(n^2m + nu(n))$	$O(p^2m + pu(n))$	$O(nm\alpha(m) + nu(n))$	$O(p \log m + pu(n))$
save	$O(nm)$	$O(1)$	$O(nm)$	$O(1)$
backtrack	$O(nm)$	$O(pm)$	$O(nm)$	$O(p)$

5.3 Analysis

In this section, I present asymptotic analyses of the various approaches to the type environment data structure discussed in the previous section. My results are summarized in Table 5.2; in all cases, n is the number of type classes, m is the maximum size of a type class, and p the number of edits between two environments or one environment and the empty environment. $u(n)$ captures the recursive cost of class unification, which is kept separate so that the $O(n)$ number of recursive class unifications can be distinguished from the direct cost of each recursive step.

5.3.1 Naïve and Incremental

The naïve type environment data structure does not have an environment-wide index for type variables, but does have an index for each type class, assumed to be hash-based here. As a result, every class's index must be consulted for a *find* operation, at an overall cost of $O(n)$. The incremental variant builds an overall hash-based index as it is queried, but may need to recursively check its parent environments if its local index does not contain a

type variable, and may have as many parents as times it has been modified, for cost $O(p)$. It should be noted that subsequent queries for the same variable execute in constant time.

Since both naïve and incremental variants store complete type classes, the cost of a *report* operation is simply the time needed to output the contained variables, $O(m)$. Since the type classes store their bounds, *bound* and *bind* are both $O(1)$ given a type class. Once a $find$ operation has already been performed to verify that a type variable does not exist in the environment, the data structures for both these variants support adding new type classes (the *insert* operation) in $O(1)$. Adding a variable to a type class (the *add* operation) can be done in $O(1)$ for the naïve implementation, but the incremental implementation may need to copy the edited type class from a parent at cost $O(m)$.

The linear storage of type classes in both variants also leads to $O(m)$ time for the variable-merging step in $unify$, plus the usual $u(n)$ recursion term for $unifyBound$. The naïve *combine* operation must traverse each of the classes of one environment, merging in any class of the other environment that shares a type variable. Since there are at most n classes to unify, the unification cost is $O(nm + nu(n))$, while traversal and $find$ costs to locate classes to merge total $O(n^2m)$, for an overall cost of $O(n^2m + nu(n))$. The incremental *combine* operation works similarly, but only needs to consider classes modified in either environment with respect to the common ancestor of both environments, allowing the n cost terms to be substituted for p, for an overall cost of $O(p^2m + pu(n))$. Neither variant supports the *split* operation to undo a $unify$.

The naïve environment does nothing to support *save* and *backtrack*, so these operations must be implemented by making a deep copy of the environment on *save*, then a destructive overwrite on *backtrack*, each at a cost of $O(nm)$. The incremental environment supports $O(1)$ *save* by simply setting aside a reference to the current environment, then proceeding with a new, empty environment with the former environment as a parent. *backtrack* to a parent environment may involve destroying all the intermediate environments if this backtrack removes the last reference to the backtracked-from environment; this cost is $O(pm)$.

5.3.2 Union-Find

The union-find data structure is designed to support $find$ efficiently, and thus for any variant, the cost is simply the distance up the tree to the representative element. For basic union-find, this is amortized to the inverse Ackermann function, $O(\alpha(m))$, essentially a small constant, though the loss of path compression in persistent union-find raises this cost to $O(\log m)$. Basic union-find is not designed to support the *report* operation, however,

so it must be simulated by checking the representative of every type variable, at cost $O(nm\alpha(m))$. Persistent union-find, on the other hand, uses the "with classes" extension to union-find described in Section 5.2.4 to run *report* in $O(m)$ time.

All union-find environment variants described here use a secondary hash table to map from class representatives to bindings, and as such can perform *bound* and *bind* in $O(1)$, given the representative. *insert* is also a $O(1)$ operation for both basic and persistent union-find. Since *add* simply involves attaching a new child to the class root, it is also a $O(1)$ operation for all union-find environment variants.

Union-find is also designed to support $unify$ in constant time, and as such, given class representatives, the variable-merging cost of $unify$ for both variants is $O(1)$ to make one representative the child of the other, plus the $O(u(n))$ term for $unifyBound$. Basic union-find does not support *split*, but persistent union-find can accomplish it using the mechanism described in Section 5.2.4 in $O(\log m)$, the cost of traversing up to the root of a class from a leaf without path compression. *combine* on the basic union-find data structure works similarly to the data structures discussed above in Section 5.3.1, with a $O(nu(n))$ term for the $O(n)$ underlying $unify$ operations, and a $O(n\alpha(m))$ term to find the classes which need unification by checking the class representatives of each corresponding type variable in both environments for equality. Persistent union-find uses a different approach for *combine*, discussed in Section 5.2.5. Discounting recursive $unify$ operations included in the $u(n)$ $unifyBound$ term, there may be at most $O(p)$ $unify$ operations performed, at cost $O(pu(n))$. Each of the $O(p)$ steps on the edit path can be processed in the $O(\log m)$ time it takes to find the current representative of the modified type class, for a total runtime of $O(p \log m + pu(n))$.

In terms of backtracking operations, the basic union-find data structure only supports deep copies, for $O(nm)$ cost for both *save* and *backtrack*. Persistent union-find, as the name suggests, is more optimized, with $O(1)$ cost to *save* a backtrack-capable reference to the current environment state, and $O(p)$ cost to revert to that state (possibly destroying no-longer-used edit nodes along the path).

5.4 Conclusion & Future Work

This chapter presents the type environment abstract data type, some type-environment data-structures optimized for workloads encountered in the expression resolution problem, and asymptotic analysis of each data structure. Chapter 6 provides experimental performance results for a representative set of these approaches. One contribution of this thesis

is the union-find with classes data structure for efficient retrieval of union-find class members, along with a related algorithm for reversing the history of *union* operations in this data structure. This reversible history contributes to the second novel contribution of this chapter, a type environment data structure based off the persistent union-find data structure of Conchon and Filliâtre [11]. This persistent union-find environment uses the *split* operation introduced in union-find with classes and the edit history of the persistent data structure to support an environment-combining algorithm that only considers the edits between the environments to be merged.

This persistent union-find data structure is efficient, but not thread-safe; as suggested in Section 4.3, it may be valuable to parallelize the C∀ expression resolver. However, allowing multiple threads concurrent access to the persistent data structure is likely to result in "reroot thrashing", as different threads reroot the data structure to their own versions of interest. This contention could be mitigated by partitioning the data structure into separate subtrees for each thread, with each subtree having its own root node, and the boundaries among them implemented with a lock-equipped ThreadBoundary edit node. Alternatively, the concurrent hash trie of Prokopec *et al.* [39, 40] may be a useful hash-table replacement.

Chapter 6

Experiments

I implemented a prototype system to test the practical effectiveness of the various algorithms described in Chapters 4 and 5. This prototype system implements the expression resolution pass of the C∀ compiler, `cfa-cc`, with a simplified version of the C∀ type system and a parser to read in problem instances, and is published online under a permissive licence[A]. The resolver prototype allows for quicker iteration on algorithms due to its simpler language model and lack of a requirement to generate runnable code, yet captures enough of the nuances of C∀ to have predictive power for the runtime performance of algorithmic variants in `cfa-cc` itself.

`cfa-cc` can generate realistic test inputs for the resolver prototype from equivalent C∀ code; the generated test inputs currently comprise all C∀ code currently in existence[B], 9,000 lines drawn primarily from the standard library and compiler test suite. This code includes a substantial degree of name overloading for common library functions and a number of fundamental polymorphic abstractions, including iterators and streaming input/output. `cfa-cc` is also instrumented to produce a number of code metrics. These metrics were used to construct synthetic test inputs during development of the resolver prototype; these synthetic inputs provided useful design guidance, but the performance results presented in this chapter are based on the more realistic directly-generated inputs.

[A] `https://github.com/cforall/resolv-proto`

[B] Though C∀ is backwards-compatible with C, the lack of **forall** functions and name overloading in C mean that the larger corpus of C code does not provide challenging test instances for `cfa-cc`.

6.1 Resolver Prototype Features

The resolver prototype can express most of the CFA features described in Chapter 2. It supports both monomorphic and polymorphic functions, with type assertions for polymorphic functions. Traits are not explicitly represented, but `cfa-cc` inlines traits before the resolver pass, so this is a faithful representation of the existing compiler. The prototype system supports variable declarations as well as function declarations, and has a lexical-scoping scheme and CFA-like overloading rules.

The type system of the resolver prototype also captures key aspects of the CFA type system. *Concrete types* represent the built-in arithmetic types of CFA, along with the implicit conversions among them. Each concrete type is represented by an integer identifier, and the conversion cost from x to y is $|y - x|$, a safe conversion if $y > x$, or an unsafe conversion if $y < x$. This scheme is markedly simpler than the graph of conversion costs in CFA (Figure 4.1), but captures the essentials of the design. For simplicity, **zero_t** and **one_t**, the types of `0` and `1`, are represented by the type corresponding to **int**. *Named types* are analogues to CFA aggregates, such as structs and unions; aggregate fields are encoded as unary functions from the struct type to the field type, with the function named based on the field name. Named types also support type parameters, and as such can represent generic types as well. Generic named types are used to represent the built-in parameterized types of CFA as well; `T*` is encoded as `#$ptr<T>`. CFA arrays are also represented as pointers, to simulate array-to-pointer decay, while top-level reference types are replaced by their referent to simulate the variety of reference conversions. *Function types* have first-class representation in the prototype as well; CFA function pointers are represented as variables with the appropriate function type, though CFA polymorphic function pointers cannot be represented, as the prototype system stores information about type assertions in function declarations rather than in the function type. *Void* and *tuple types* are also supported in the prototype, to express the multiple-return-value functions in CFA, though varargs functions and **ttype** tuple-typed type variables are absent from the prototype system. The prototype system also does not represent type qualifiers (*e.g.* **const**, **volatile**), so all such qualifiers are stripped during conversion to the prototype system.

The resolver prototype supports three sorts of expressions in its input language. The simplest are *value expressions*, which are expressions declared to be a certain type; these implement literal expressions in CFA, and, already being typed, are passed through the resolver unchanged. The second sort, *name expressions*, represent a variable expression in CFA; these contain the name of a variable or function, and are matched to an appropriate declaration overloading that name. The third input expression, the *function expression*, represents a call to a function, with a name and zero or more argument subexpressions. As

is usual in C∀, operators are represented as function calls; however, as mentioned above, the prototype system represents field access expressions a.f as function expressions as well.

The main area for future expansion in the design of the resolver prototype is conversions. Cast expressions are implemented in the output language of the resolver, but cannot be expressed in the input. The only implicit conversions supported are among the arithmetic-like concrete types, which capture most, but not all, of C∀'s built-in implicit conversions[C]. Future work should include a way to express implicit (and possibly explicit) conversions in the input language, with an investigation of the most efficient way to handle implicit conversions, and potentially a design for user-defined conversions.

6.2 Resolver Prototype Design

As discussed above, for speed of development the resolver prototype works over a simplified version of the C∀ type system. The build system for the resolver prototype uses a number of conditional compilation flags to switch among algorithm variants while retaining maximally shared code. A distinct executable name is also generated for each algorithmic variant so that distinct variants can be more easily tested against each other.

The primary architectural difference between the resolver prototype and `cfa-cc` is that the prototype system uses a simple mark-and-sweep garbage collector for memory management, while `cfa-cc` uses a manual memory-management approach. This architectural difference affects the mutation patterns used by both systems: `cfa-cc` frequently makes deep clones of multi-node object graphs to ensure that there is a single "owner" for each object which can safely delete it later; the prototype system, by contrast, relies on its garbage collector to handle ownership, and can often copy pointers rather than cloning objects. The resolver prototype thus only needs to clone nodes that it modifies, and can share un-modified children between clones; the tree mutator abstraction in the prototype is designed to take advantage of this property. The key design decision enabling this is that all child nodes are held by **const** pointer, and thus cannot be mutated once they have been stored in a parent node. With minimal programming discipline, it can thus be ensured that any expression is either mutable or shared, but never both; the Dotty research compiler for Scala takes a similar architectural approach [33].

Given the significantly better performance results from the resolver prototype than `cfa-cc` and profiling data showing that memory allocation is a large component of `cfa-cc` runtime, I attempted to port this garbage collector to `cfa-cc`, but without success. The

[C]Notable absences include **void**∗ to other pointer types, or 0 to pointer types.

GC could be used for memory management with few changes to the code-base, but without a substantial re-write to enforce the same "**const** children" discipline, `cfa-cc` could not take advantage of the potential to share sub-objects; without sharing of sub-objects the GC variant of `cfa-cc` must do all the same allocations and deletions and garbage-collector overhead degraded performance unacceptably (though it did fix some known memory leaks introduced by failures of the existing manual memory-management scheme).

Another minor architectural difference between the prototype system and `cfa-cc` is that `cfa-cc` makes extensive use of the pointer-based std::list, std::set, and std::map data structures, while the prototype uses the array-based std::vector and the hash-based unordered_ variants of set and map instead. Porting the prototype to use the pointer-based data structures resulted in modest performance regressions, whereas preliminary results from porting `cfa-cc` to use std::vector over std::list also showed performance regressions, in some cases significant. The relative performance impact of this architectural difference is unclear, and thus excluded from consideration.

The final difference between `cfa-cc` and the resolver prototype is that, as an experiment in language usability, the prototype performs resolution-based rather than unification-based assertion satisfaction, as discussed in Section 4.3. This change enables coding patterns not available in `cfa-cc`, *e.g.* a more flexible approach to type assertion satisfaction and better handling of functions returning polymorphic type variables that do not exist in the parameter list. The experimental results in Section 6.3 indicate that this choice is not a barrier to a performant resolver.

6.3 Prototype Experiments

The primary performance experiments for this thesis are conducted using the resolver prototype on problem instances generated from actual C∀ code using the method described in Section 6.1. The prototype is compiled in 24 variants over 3 variables, with variants identified by the hyphen-separated concatenation of their short codes, *e.g.* BU-IMM-BAS for bottom-up traversal, immediate assertion satisfaction, basic type environment. The variables and their values are as follows:

Traversal direction The order in which arguments are matched with parameters, as discussed in Section 4.2.2.

Bottom-up (BU) Baker-style bottom-up pass, searching for function candidates based on the available argument interpretations.

Combined (CO) Bilson-style bottom-up pass, where argument interpretations are combined into a single interpretation for each set of options.

Top-down (TD) Cormack-style top-down pass, searching for argument interpretations based on function candidate parameter types. The TD-* variants of the resolver prototype implement a caching system to avoid re-computation of the same argument interpretation with the same type.

Assertion satisfaction The algorithm for finding satisfying declarations for type assertions, as discussed in Section 4.2.3.

Immediate (IMM) All assertions are checked for satisfaction immediately upon generating a candidate interpretation. The techniques discussed in Section 4.2.3 for environment combination and level-by-level consideration of recursive assertions are applied here.

Deferred (DEF) As in IMM, but only checks minimal-cost top-level interpretations after all top-level interpretations have been generated.

Deferred Cached (DCA) As in DEF, but uses the caching optimization discussed in Section 4.2.3.

Type Environment The type environment data structure used, as discussed in Chapter 5.

Basic (BAS) Bilson-style type environment with hash-based equivalence class storage, as discussed in Section 5.2.1.

Incremental Inheritance (INC) Incremental-inheritance variant sharing unmodified common parent information among environments, as discussed in Section 5.2.2.

Persistent union-find (PER) Union-find-based environment, using the persistent variant discussed in Section 5.2.5 for backtracking and combination. This variant requires that all pairs of type arguments used as arguments to *combine* descend from a common root environment; this requirement is incompatible with the caching used in the top-down traversal direction, and thus no TD-*-PER algorithms are tested.

To test the various algorithms, the resolver prototype is compiled using g++ 6.5.0 with each of the 24 valid combinations of variables[D], and then timed running each of the C∀-derived test inputs. Terminal output is suppressed for all tests to avoid confounding factors

[D] Namely, all combinations except TD-*-PER.

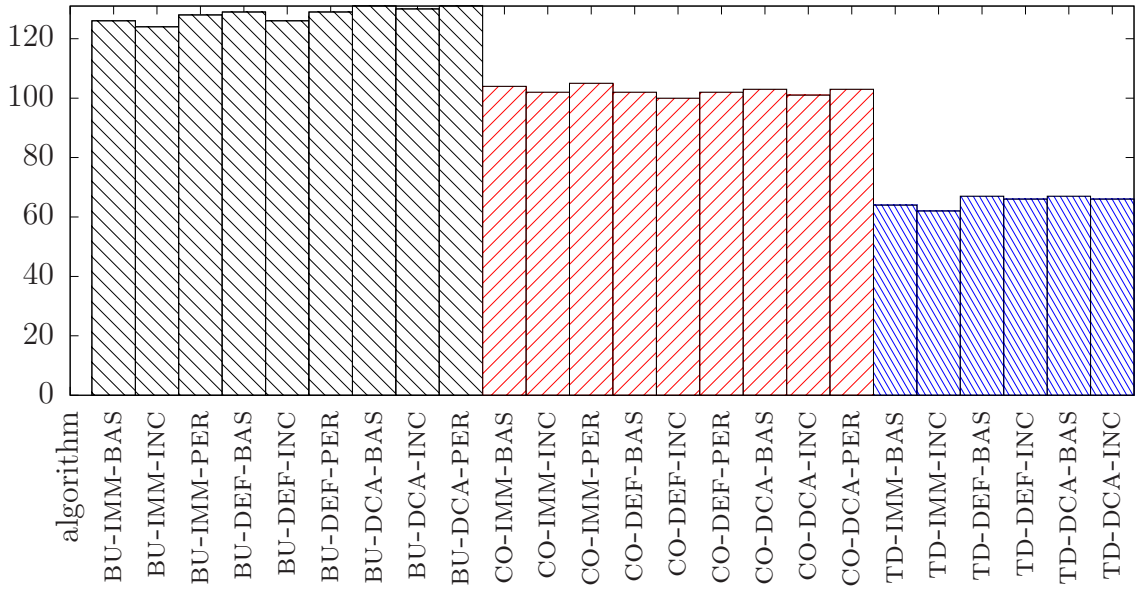


Figure 6.1: Number of tests completed for each algorithmic variant

in the timing results, and all tests are run three times in series, with the median result reported in all cases. The medians are representative data points; considering test cases that took at least 0.2 s to run, the average run was within 2% of the reported median runtime, and no run diverged by more than 20% of median runtime or 5.5 s. The memory results are even more consistent, with no run exceeding 2% difference from median in peak resident set size, and 93% of tests recording identical peak memory usage within the 1 KB granularity of the measurement software. All tests were run on a machine with 128 GB of RAM and 64 cores running at 2.2 GHz.

As a matter of experimental practicality, test runs that exceeded 8 GB of peak resident memory usage are excluded from the data set. This restriction is justifiable by real-world use, as a compiler that is merely slow may be accommodated with patience, but one that uses in excess of 8 GB of RAM may be impossible to run on many currently deployed computer systems. 8 GB of RAM is not typical of the memory usage of the best-performing two variants, BU-DCA-BAS and BU-DCA-PER, which were able to run all 131 test inputs to completion with maximum memory usage of 70 MB and 78 MB, respectively. However, this threshold did eliminate a significant number of algorithm-test variants, with the worst-performing variant, TD-IMM-INC, only completing 62 test inputs within the memory bound. Full results for tests completed by algorithm variant are presented in Figure 6.1. As can

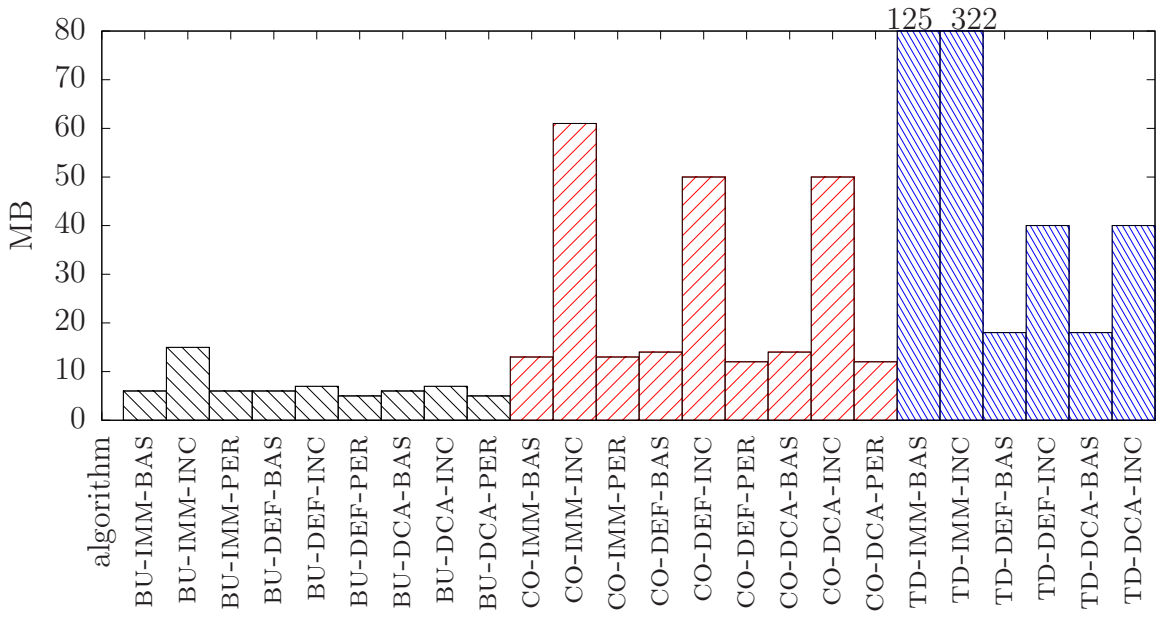


Figure 6.2: Average peak resident set size for each algorithmic variant over the 56 test inputs all variants complete.

be seen from these results, traversal direction is clearly the dominant variable in memory usage, with the BU-* variants performing better than the CO-* variants, which in turn out-perform the TD-* variants.

To provide a more holistic view of performance, I have considered the results from the 56 test inputs that all algorithms are able to complete within the memory bound. Limiting consideration to these algorithms provides an apples-to-apples comparison among algorithms, as the excluded inputs are harder instances, which take more time and memory for the algorithms that are able to solve them. Figures 6.2 and 6.3 show the mean peak memory and runtime, respectively, of each algorithm over the inputs in this data set. These averages are not themselves meaningful, but do enable an overall comparison of relative performance of the different variants. Selecting only these 56 "easy" test inputs does bias the average values downward, but has little effect on the relative trends; similar trends can be seen in the graphs of the BU-* algorithms over the 124 (of 131) test inputs that all complete, which have been omitted to save space.

It can be seen from these results that the top-down, immediate assertion-satisfaction (TD-IMM-*) variants are particularly inefficient, as they check a significant number of assertions without filtering to determine if the arguments can be made to fit. It is also

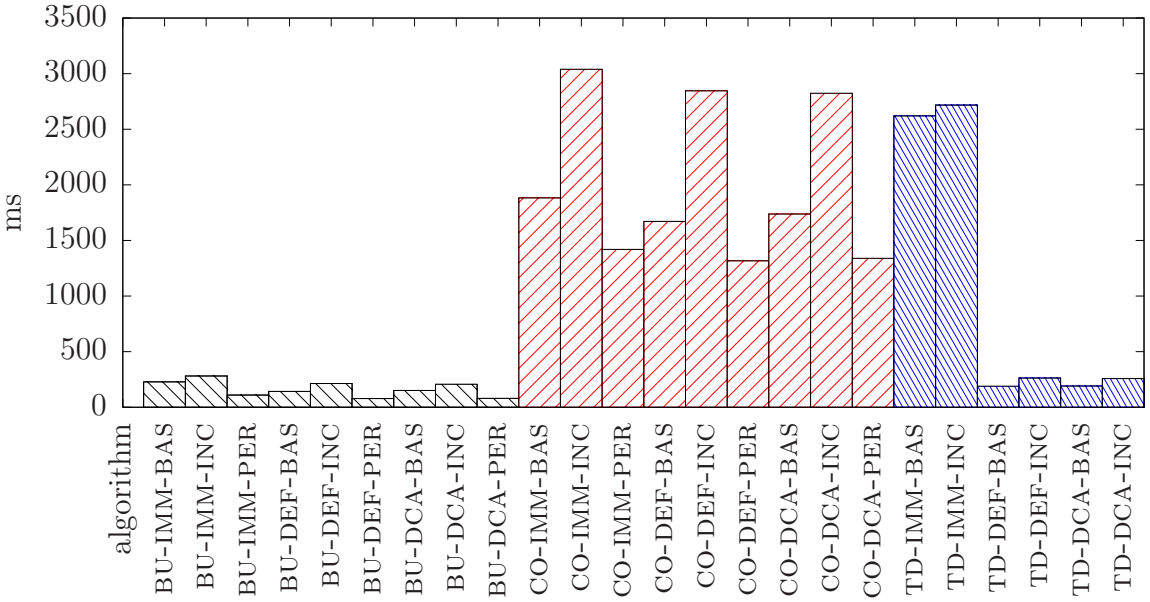


Figure 6.3: Average runtime for each algorithmic variant over the 56 test inputs all variants complete.

clear that the bottom-up (BU) traversal order is better than both top-down (TD) and the Bilson-style bottom-up-combined (CO) orders. While the advantage of BU over CO is clear, in that it performs less redundant work if a prefix of a combination fails, the advantage of BU over TD provides an answer for an open question from Baker [5]. I believe that bottom-up is superior because it must only handle each subexpression once to form a list of candidate interpretations, whereas the top-down approach may do similar work repeatedly to resolve a subexpression with a variety of different types, a shortcoming that cannot be fully addressed by the memoization scheme employed in the TD algorithm.

With regard to assertion satisfaction, immediate (IMM) satisfaction is an inferior solution, though there is little performance difference between deferred (DEF) and deferred-cached (DCA) for instances that both can complete; particularly notable is that the DCA caching-scheme does not have a noticeable impact on peak memory usage. Since the DCA algorithm can solve some particularly hard instances that DEF cannot, it is the recommended approach.

The incremental-inheritance (INC) type environment also often uses upwards of double the memory required by the other variants, in addition to being consistently slower on these easy tests; aside from BU-IMM-BAS performing worse than BU-IMM-INC on average

when larger tests are considered, these results hold for the other variants. It is apparent from these results that any efficiencies from the inheritance mechanism are insufficient to pay for the added complexity of the data structure. Aside from that, the persistent union-find (PER) type environment generally performs better than the basic (BAS) environment, with similar peak memory usage and an average speedup factor of nearly 2, though the requirements of the PER environment for automatic garbage collection and a shared history for combination make retrofitting it into older code difficult.

6.4 Instance Difficulty

To characterize the difficulty of expression-resolution problem instances, the test suites must be explored at a finer granularity. As discussed in Section 4.2.1, a single top-level expression is the fundamental problem instance for resolution, yet the test inputs discussed above are composed of thousands of top-level expressions, like the actual source code they are derived from. To pull out the effects of these individual problems, the resolver prototype is instrumented to time resolution for each expression, and also to report some relevant properties of the expression. This instrumented resolver is then run on a set of difficult test instances; to limit the data collection task, these runs are restricted to the best-performing BU-DCA-PER algorithm and test inputs taking more than 1 s to complete.

The 13 test inputs thus selected contain 20,632 top-level expressions among them, which are separated into order-of-magnitude bins by runtime in Figure 6.4. As can be seen from this figure, overall runtime is dominated by a few particularly difficult problem instances — the 60% of expressions that resolve in under 0.1 ms collectively take less time to resolve than any of the 0.2% of expressions that take at least 100 ms to resolve. On the other hand, the 46 expressions in that 0.2% take 38% of the overall time in this difficult test suite, while the 201 expressions that take between 10 and 100 ms to resolve consume another 30%.

Since the top centile of expression-resolution instances requires approximately two-thirds of the resolver's time, optimizing the resolver for specific hard problem instances has proven to be an effective technique for reducing overall runtime. The data indicates that the number of assertions necessary to resolve has the greatest effect on runtime, as seen in Figure 6.5. However, since the number of assertions required is only known once resolution is finished, the most-promising pre-resolution metric of difficulty is the nesting depth of the expression; as seen in Figure 6.6, expressions of depth > 10 in this data-set are uniformly difficult. Figure 6.7 presents a similar pattern for number of subexpressions, though given that the expensive tail of problem instances occurs at approximately twice the

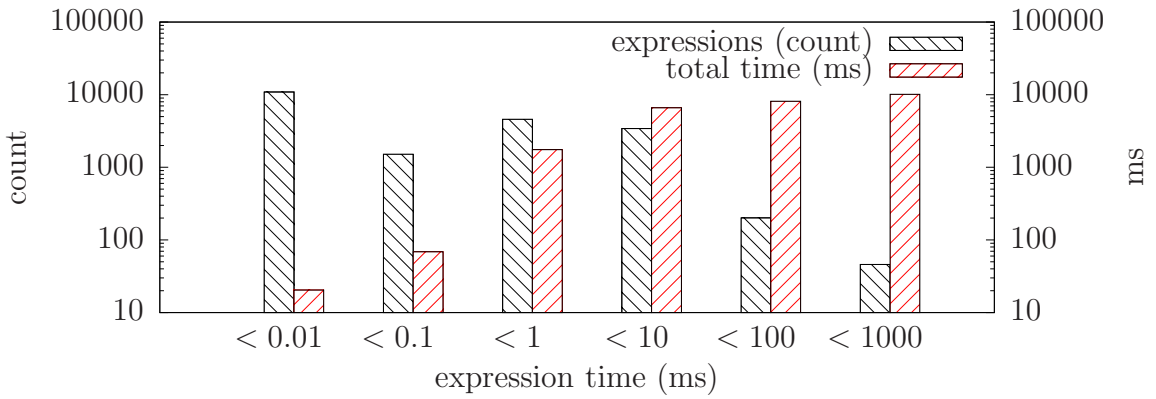


Figure 6.4: Histogram of top-level expression resolution runtime, binned by order-of-magnitude. The left series counts the expressions in each bin according to the left axis, while the right series reports the summed runtime of resolution for all expressions in that bin. Note that both y-axes are log-scaled.

depth values, it is reasonable to believe that the difficult expressions in question are deeply-nested invocations of binary functions rather than wider but shallowly-nested expressions.

6.5 C∀ Results

I have integrated a number of the algorithmic techniques discussed in this chapter into `cfa-cc`. This integration took place over a period of months while `cfa-cc` was under active development on a number of other fronts, so it is not possible to completely isolate the effects of the algorithmic changes, but I believe the algorithmic changes to have had the most-significant effects on performance over the study period. To generate this data, representative commits from the `git` history of the project were checked out and compiled, then run on the same machine used for the resolver prototype experiments discussed in Section 6.3. To negate the effects of changes to the C∀ standard library on the timing results, 55 test files from the test suite of the oldest C∀ variant are compiled with the `-E` flag to inline their library dependencies, and these inlined files are used to test the remaining `cfa-cc` versions.

I performed two rounds of modification to `cfa-cc`; the first round moved from Bilson's original combined-bottom-up algorithm to an un-combined bottom-up algorithm, denoted CFA-CO and CFA-BU, respectively. A top-down algorithm was not attempted in `cfa-cc`

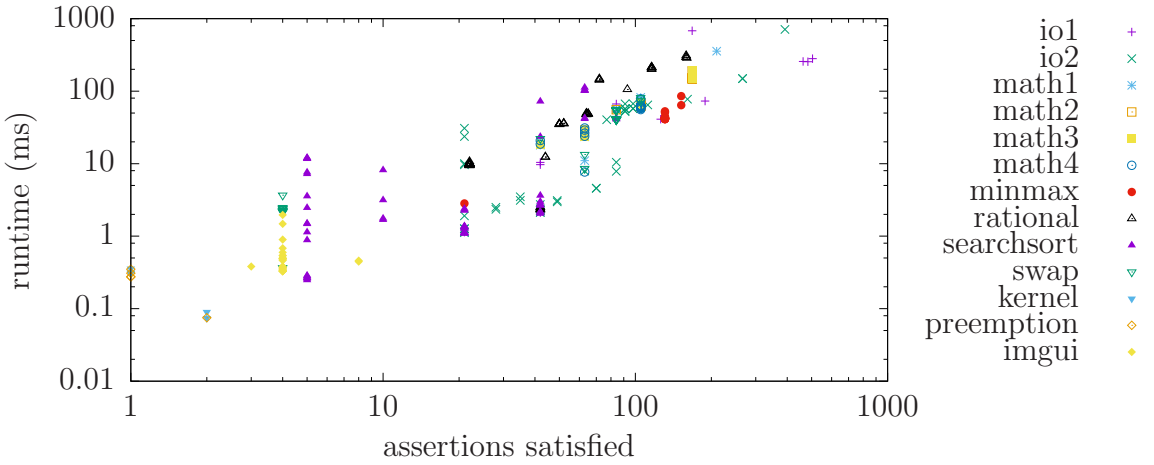


Figure 6.5: Top-level expression resolution time by number of assertions resolved. Source input file for each expression listed in legend; note log scales on both axes.

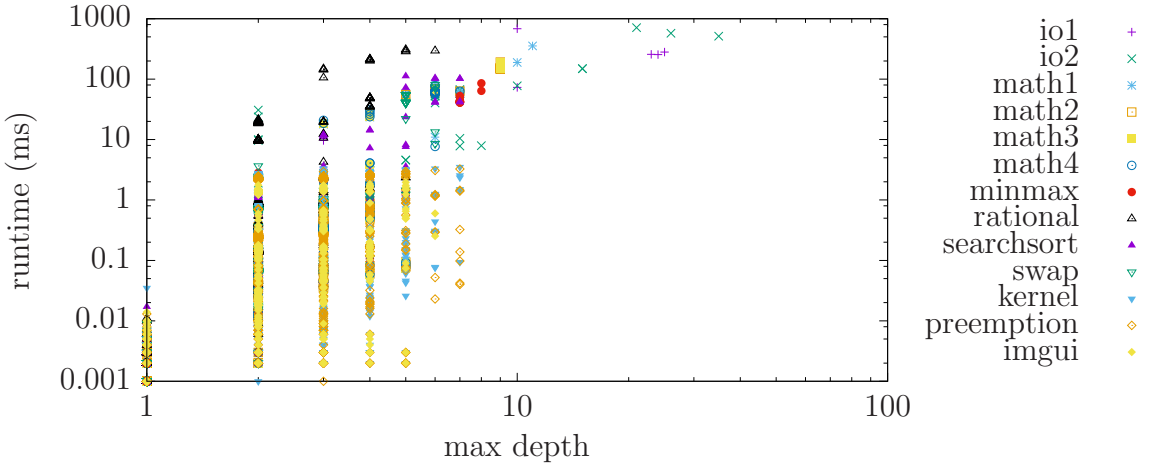


Figure 6.6: Top-level expression resolution time by maximum nesting depth of expression. Note log scales on both axes.

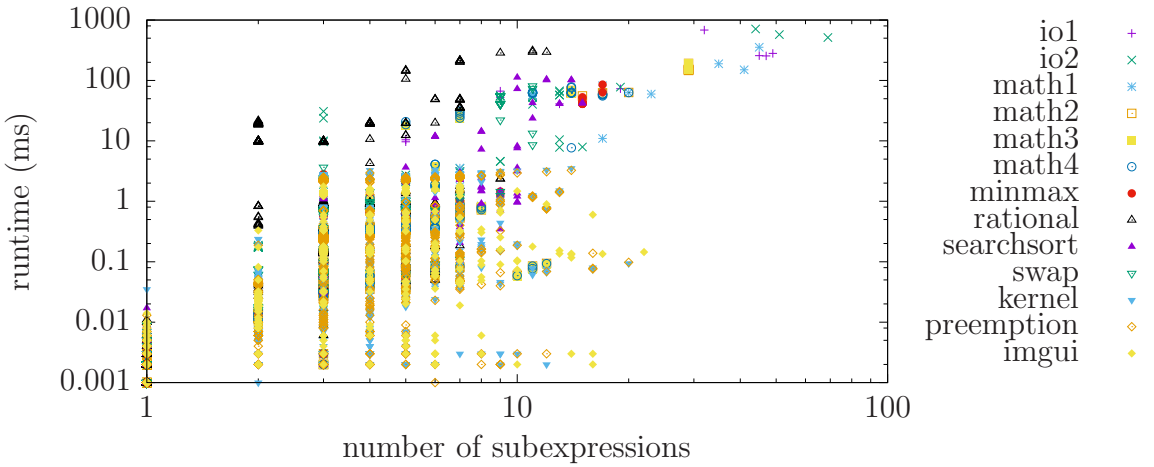


Figure 6.7: Top-level expression resolution time by number of subexpressions. Note log scales on both axes.

due to its poor performance in the prototype. The second round of modifications addressed assertion satisfaction, taking Bilson's original CFA-IMM algorithm and modifying it to use the deferred approach CFA-DEF. Due to time constraints, a deferred-cached assertion satisfaction algorithm for `cfa-cc` could not be completed, but both preliminary results from this effort and the averaged prototype results from Section 6.3 indicate that assertion satisfaction caching is not likely to be a fruitful optimization for `cfa-cc`. The new environment data structures discussed in Section 6.3 have not been successfully merged into `cfa-cc` due to their dependencies on the garbage-collection framework in the prototype; I spent several months modifying `cfa-cc` to use similar garbage collection, but due to `cfa-cc` not being designed to use such memory management the performance of the modified compiler was non-viable. It is possible that the persistent union-find environment could be modified to use a reference-counted pointer internally without changing the entire memory-management framework of `cfa-cc`, but such an attempt is left to future work.

As can be seen in Figures 6.8–6.10, the time and peak memory results for these five versions of `cfa-cc` show that assertion resolution dominates total resolution cost, with the CFA-DEF variant running consistently faster than the others on more expensive test cases, and the speedup from the deferred approach increasing with the difficulty of the test case. The results from `cfa-cc` for CFA-CO *vs.* CFA-BU do not mirror those from the prototype; I conjecture this is mostly due to the different memory-management schemes and sorts of data required to run type unification and assertion satisfaction calculations, as `cfa-cc` performance has proven to be particularly sensitive to the amount of heap allocation

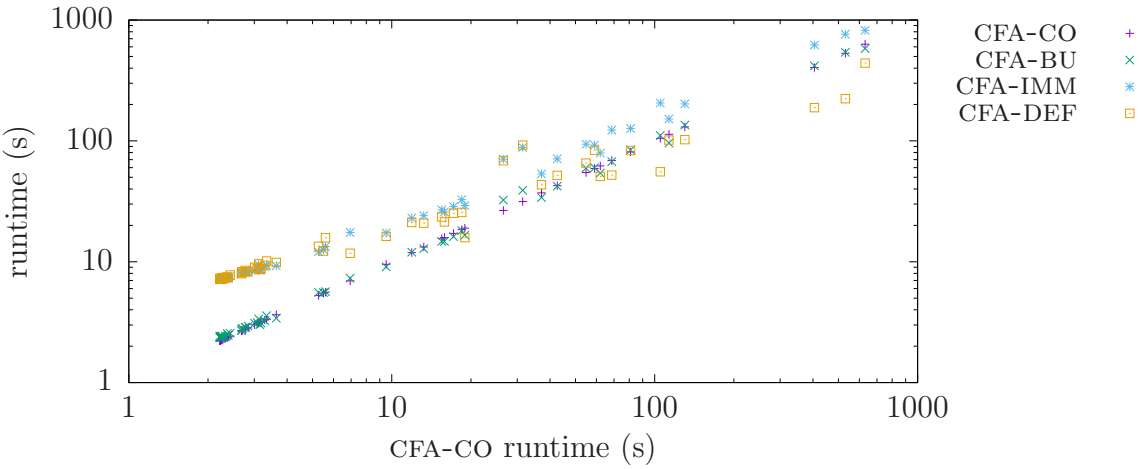


Figure 6.8: `cfa-cc` runtime against CFA-CO baseline. Note log scales on both axes.

performed. This data also shows a noticeable regression in compiler performance in the eleven months between CFA-BU and CFA-IMM, which use the same resolution algorithms; this approximate doubling in runtime is not due to expression resolution, as no integration work happened in this time, but I am unable to ascertain its actual cause. To isolate the effects of the algorithmic changes from this unrelated performance regression, the speedup results in Figure 6.9 are shown with respect to the start of each modification round, CFA-BU *vs.* CFA-CO and CFA-DEF *vs.* CFA-IMM. It should also be noted with regard to the peak memory results in Figure 6.10 that the peak memory usage does not always occur during the resolution phase of the compiler.

6.6 Conclusion

The dominant factor in the cost of C∀ expression resolution is assertion satisfaction. Reducing the total number of assertions satisfied, as in the deferred satisfaction algorithm, is consistently effective at reducing runtime, and caching results of these satisfaction problem instances has shown promise in the prototype system. The results presented here also demonstrate that a bottom-up approach to expression resolution is superior to top-down, settling an open question from Baker [5]. The persistent union-find type environment introduced in Chapter 5 has also been demonstrated to be a modest performance improvement on the naïve approach.

Given the consistently strong performance of the BU-DCA-IMM and BU-DCA-PER vari-

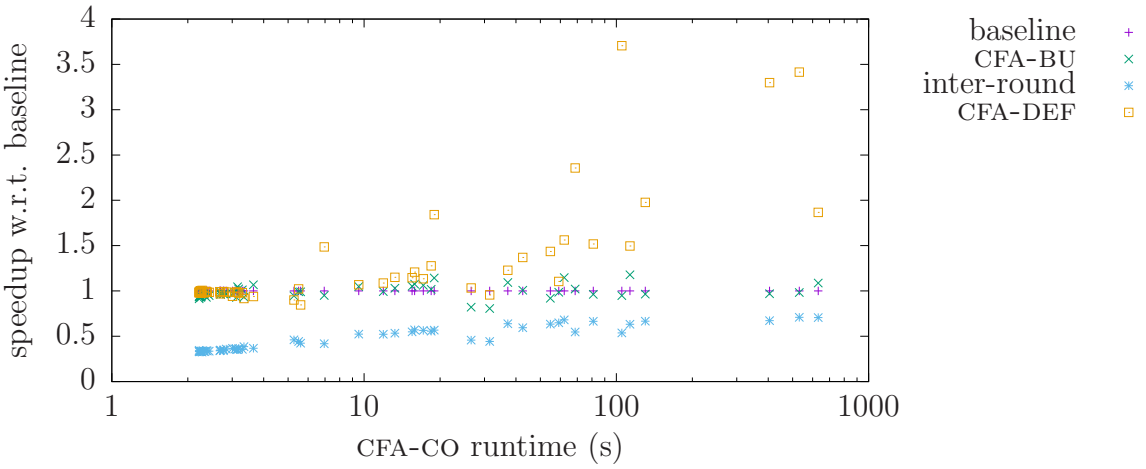


Figure 6.9: `cfa-cc` speedup against against CFA-CO baseline runtime. To isolate the effect of algorithmic changes, CFA-BU speedup is *vs.* CFA-CO while CFA-DEF speedup is *vs.* CFA-IMM. The 'inter-round' series shows slowdown between CFA-BU and CFA-IMM.

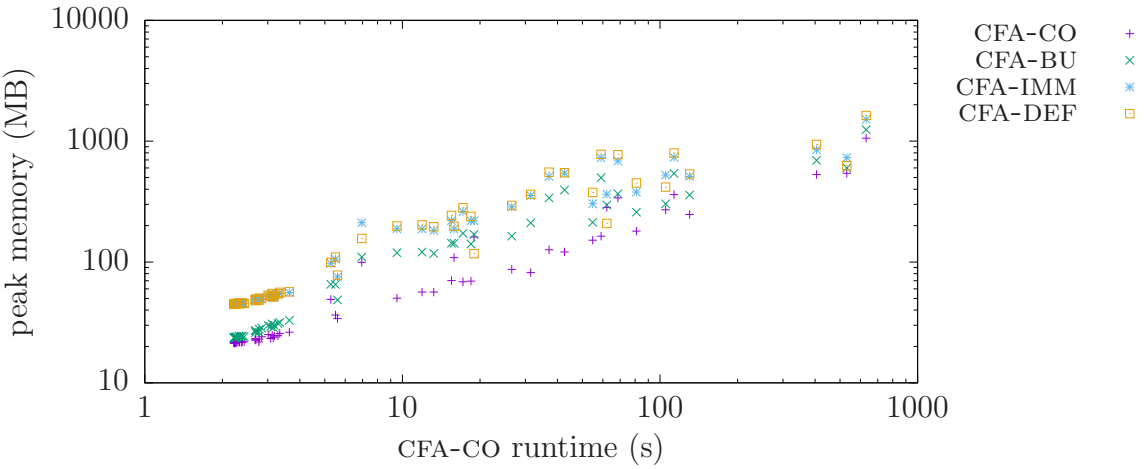


Figure 6.10: `cfa-cc` peak memory usage against CFA-CO baseline runtime. Note log scales on both axes.

ants of the resolver prototype, the results in this chapter demonstrate that it is possible to develop a C∀ compiler with acceptable runtime performance for widespread use, an important and previously unaddressed consideration for the practical viability of the language. However, the less-marked improvement in Section 6.5 from retrofitting these algorithmic changes onto the existing compiler leave the actual development of a performant C∀ compiler to future work. Characterization and elimination of the performance deficits in the existing `cfa-cc` has proven difficult, though runtime is generally dominated by the expression resolution phase; as such, building a new C∀ compiler based on the resolver prototype contributed by this work may prove to be an effective strategy.

Chapter 7

Conclusion

Decades after its first standardization, the C language remains a widely-used tool and a vital part of the software development landscape. The C∀ language under development at the University of Waterloo represents an evolutionary modernization of C with expressive modern language features paired with strong C backwards-compatibility. This thesis has contributed to these project goals in a variety of ways, including the addition of a generic-types language feature (Chapter 3) and refinement of the C∀ overload selection rules to produce a more expressive and intuitive model (Section 4.1.2). Based on the technical contribution of the resolver prototype system (Section 6.1), I have also made significant improvements to C∀ compilation performance, including un-combined bottom-up expression traversal (Section 4.2.2), deferred-cached assertion satisfaction (Section 4.2.3), and a novel persistent union-find type environment data structure (Section 5.2.5). The combination of these practical improvements and added features significantly improve the viability of C∀ as a practical programming language.

Further improvements to the C∀ type system are still possible, however. One area suggested by this work is development of a scheme for user-defined conversions; to integrate properly with the C∀ conversion model, there would need to be a distinction between safe and unsafe conversions, and possibly a way to denote conversions as explicit-only or non-chainable. Another place for ongoing effort is improvement of compilation performance; I believe the most promising direction for that effort is rebuilding the C∀ compiler on a different framework than Bilson's `cfa-cc`. The resolver prototype presented in this work has good performance and already has the basics of C∀ semantics implemented, as well as many of the necessary core data structures, and would be a viable candidate for a new compiler architecture. An alternate approach would be to fork an existing C compiler

such as Clang [1], which would need to be modified to use one of the resolution algorithms discussed here, as well as various other features introduced by Bilson [6].

More generally, the algorithmic techniques described in this thesis may be useful to implementors of other programming languages. In particular, the demonstration of practical performance for polymorphic return-type inference suggests the possibility of eliding return-type-only template parameters in C++ function calls, though integrating such an extension into C++ expression resolution in a backwards-compatible manner may be challenging. The C$\forall$ expression resolution problem also bears some similarity to the *local type inference* model put forward by Pierce & Turner [38] and Odersky *et al.* [34]; compiler implementors for languages like Scala [42], which performs type inference based on this model, may be able to profitably adapt the algorithms and data structures presented in this thesis.

References

[1] Clang: a C language family frontend for LLVM. https://clang.llvm.org/.

[2] Alfred V Aho, John E Hopcroft, and Jeffrey D Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, USA, 1974.

[3] Henry G. Baker. Shallow binding makes functional arrays fast. *SIGPLAN Not.*, 26(8):145–147, August 1991.

[4] Henry G. Baker, Jr. Shallow binding in lisp 1.5. *Commun. ACM*, 21(7):565–569, July 1978.

[5] Theodore P. Baker. A one-pass algorithm for overload resolution in Ada. *Transactions on Programming Languages and Systems*, 4(4):601–614, October 1982.

[6] Richard C. Bilson. Implementing overloading and polymorphism in C∀. Master's thesis, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, 2003. http://plg.uwaterloo.ca/theses/BilsonThesis.pdf.

[7] Andrew P Black and Norman C Hutchinson. Typechecking polymorphism in Emerald. Technical report, Cambridge Research Laboratory, Digital Equipment Corporation, 1990.

[8] P. A. Buhr, David Till, and C. R. Zarnke. Assignment as the sole means of updating objects. *Softw. Pract. Exper.*, 24(9):835–870, September 1994.

[9] *C Programming Language ISO/IEC 9889:2011-12*. https://www.iso.org/standard/-57853.html, 3rd edition, 2012.

[10] *C++ Programming language – Extensions for concepts ISO/IEC TS 19217:2015*. https://www.iso.org/standard/64031.html, 2015.

[11] Sylvain Conchon and Jean-Christophe Filliâtre. A persistent union-find data structure. In *Proceedings of the 2007 workshop on Workshop on ML*, pages 37–46. ACM, 2007.

[12] Gordon V. Cormack. An algorithm for the selection of overloaded functions in Ada. *SIGPLAN Not.*, 16(2):48–52, February 1981.

[13] Thierry Delisle. Concurrency in C∀. Master's thesis, School of Computer Science, University of Waterloo, 2018. https://uwspace.uwaterloo.ca/handle/10012/12888.

[14] 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.

[15] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Boston, 1st edition, 1990.

[16] Zvi Galil and Giuseppe F Italiano. Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys (CSUR)*, 23(3):319–344, 1991.

[17] Bernard A Galler and Michael J Fisher. An improved equivalence algorithm. *Communications of the ACM*, 7(5):301–303, 1964.

[18] Harald Ganzinger and Knut Ripken. Operator identification in ADA: Formal specification, complexity, and concrete implementation. *SIGPLAN Notices*, 15(2):30–42, February 1980.

[19] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *Java Language Specification*, Java SE 8 edition, 2015.

[20] Robert Griesemer, Rob Pike, and Ken Thompson. *Go Programming Language*. Google, 2009. http://golang.org/ref/spec.

[21] Dan Grossman. Quantified types in an imperative language. *Transactions on Programming Languages and Systems*, 28(3):429–475, May 2006.

[22] Josh Haberman. Making arbitrarily-large binaries from fixed-size C++ code. http://blog.reverberate.org/2016/01/making-arbitrarily-large-binaries-from.html , 2016.

[23] Haskell. *Haskell 2010 Language Report*, Simon Marlow edition, 2010. https://-haskell.org/definition/haskell2010.pdf.

[24] Robbert Krebbers. An operational and axiomatic semantics for non-determinism and sequence points in c. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 101–112, New York, NY, USA, 2014. ACM.

[25] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009.

[26] Nicholas Matsakis. Lowering Rust traits to logic. http://smallcultfollowing.com/-babysteps/blog/2017/01/26/lowering-rust-traits-to-logic/, January 2017.

[27] Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. System V application binary interface. *AMD64 Architecture Processor Supplement, Draft v0*, 99, 2013.

[28] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[29] Antonio Morgado, Federico Heras, Mark Liffiton, Jordi Planes, and Joao Marques-Silva. Iterative and core-guided MaxSAT solving: A survey and assessment. *Constraints*, 18(4):478–534, 2013.

[30] Aaron Moss, Robert Schluntz, and Peter A. Buhr. C∀ : Adding modern programming language features to C. *Softw. Pract. Exper.*, 48(12):2111–2146, December 2018. http://dx.doi.org/10.1002/spe.2624.

[31] Michael Norrish. *C formalised in HOL*. PhD thesis, University of Cambridge, 1998.

[32] Objective-C. https://developer.apple.com/library/archive/documentation/Cocoa/-Conceptual/ProgrammingWithObjectiveC, 2014.

[33] Martin Odersky. Dotty. https://github.com/lampepfl/dotty. Acessed: 2019-02-22.

[34] Martin Odersky, Christoph Zenger, and Matthias Zenger. Colored local type inference. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, pages 41–53, New York, NY, USA, 2001. ACM.

[35] Md. Mostofa Ali Patwary, Jean Blair, and Fredrik Manne. Experiments on union-find algorithms for the disjoint-set data structure. In Paola Festa, editor, *Experimental Algorithms*, pages 411–423, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[36] Tom Pennello, Frank DeRemer, and Richard Meyers. A simplified operator identification scheme for Ada. *SIGPLAN Notices*, 15(7 and 8):82–87, July 1980.

[37] Guido Persch, Georg Winterstein, Manfred Dausman, and Sophia Drossopoulou. Overloading in preliminary Ada. *SIGPLAN Not.*, 15(11):47–56, November 1980. Proceedings of the ACM-SIGPLAN Symposium on the Ada Programming Language.

[38] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, January 2000.

[39] Aleksandar Prokopec, Phil Bagwell, and Martin Odersky. Cache-aware lock-free concurrent hash tries. Technical report, EPFL, 2011.

[40] Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky. Concurrent tries with efficient non-blocking snapshots. In *ACM SIGPLAN Notices*, volume 47, pages 151–160. ACM, 2012.

[41] *Rust Programming Language*, 2015. https://doc.rust-lang.org/reference.html.

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

[43] Robert Schluntz. Resource management and tuples in C∀. Master's thesis, School of Computer Science, University of Waterloo, 2017. https://uwspace.uwaterloo.ca/-handle/10012/11830.

[44] Robert E. Tarjan and Jan van Leeuwen. Worst-case analysis of set union algorithms. *J. ACM*, 31(2):245–281, March 1984.

[45] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, April 1975.

[46] TIOBE Index. http://www.tiobe.com/tiobe_index.

[47] Xcode 7 release notes. https://developer.apple.com/library/content/documentation/-Xcode/Conceptual/RN-Xcode-Archive/Chapters/xc7_release_notes.html, 2015.

Appendix A

Generic Stack Benchmarks

This appendix includes the generic stack code for all four language variants discussed in Section 3.3. Throughout, /∗∗∗/ designates a counted redundant type annotation; these include **sizeof** on a known type, repetition of a type name in initialization or return statements, and type-specific helper functions. The code is reformatted slightly for brevity.

A.1 C

```
typedef struct node {
    void ∗ value;
    struct node ∗ next;
} node;
typedef struct stack {
    struct node ∗ head;
} stack;
void copy_stack( stack ∗ s, const stack ∗ t, void ∗ (∗copy)( const void ∗ ) ) {
    node ∗∗ cr = &s->head;
    for (node ∗ nx = t->head; nx; nx = nx->next) {
        ∗cr = malloc( sizeof(node) ); /∗∗∗/
        (∗cr)->value = copy( nx->value );
        cr = &(∗cr)->next;
    }
    ∗cr = NULL;
}
void clear_stack( stack ∗ s, void (∗ free_el)( void ∗ ) ) {
```

```
    for ( node * nx = s->head; nx; ) {
        node * cr = nx;
        nx = cr->next;
        free_el( cr->value );
        free( cr );
    }
    s->head = NULL;
}
stack new_stack() {
    return (stack){ NULL }; /***/
}
stack * assign_stack( stack * s, const stack * t, void * (*copy_el)( const void * ),
                      void (*free_el)( void * ) ) {
    if ( s->head == t->head ) return s;
    clear_stack( s, free_el ); /***/
    copy_stack( s, t, copy_el ); /***/
    return s;
}
_Bool stack_empty( const stack * s ) {
    return s->head == NULL;
}
void push_stack( stack * s, void * v ) {
    node * n = malloc( sizeof(node) ); /***/
    *n = (node){ v, s->head }; /***/
    s->head = n;
}
void * pop_stack( stack * s ) {
    node * n = s->head;
    s->head = n->next;
    void * v = n->value;
    free( n );
    return v;
}
```

A.2 C∀

```
forall( otype T ) {
    struct node {
        T value;
        node(T) * next;
    };
```

```
struct stack { node(T) * head; };
void ?{}( stack(T) & s, stack(T) t ) { // copy
    node(T) ** cr = &s.head;
    for ( node(T) * nx = t.head; nx; nx = nx->next ) {
        *cr = alloc();
        ((*cr)->value){ nx->value };
        cr = &(*cr)->next;
    }
    *cr = 0;
}
void clear( stack(T) & s ) with( s ) {
    for ( node(T) * nx = head; nx; ) {
        node(T) * cr = nx;
        nx = cr->next;
        ^(*cr){};
        free( cr );
    }
    head = 0;
}
void ?{}( stack(T) & s ) { (s.head){ 0 }; }
void ^?{}( stack(T) & s) { clear( s ); }
stack(T) ?=?( stack(T) & s, stack(T) t ) {
    if ( s.head == t.head ) return s;
    clear( s );
    s{ t };
    return s;
}
_Bool empty( const stack(T) & s ) {
    return s.head == 0;
}
void push( stack(T) & s, T value ) with( s ) {
    node(T) * n = alloc();
    (*n){ value, head };
    head = n;
}
T pop( stack(T) & s ) with( s ) {
    node(T) * n = head;
    head = n->next;
    T v = n->value;
    ^(*n){};
    free( n );
    return v;
```

```
    }
}
```

A.3 C++

```
template<typename T> struct stack {
    struct node {
        T value;
        node * next;
        node( const T & v, node * n = nullptr ) :
            value( v ), next( n ) {}
    };
    node * head;
    void copy( const stack<T> & o ) {
        node ** cr = &head;
        for ( node * nx = o.head; nx; nx = nx->next ) {
            *cr = new node{ nx->value }; /***/
            cr = &(*cr)->next;
        }
        *cr = nullptr;
    }
    void clear() {
        for ( node * nx = head; nx; ) {
            node * cr = nx;
            nx = cr->next;
            delete cr;
        }
        head = nullptr;
    }
    stack() : head( nullptr ) {}
    stack( const stack<T> & o ) { copy( o ); }
    ~stack() { clear(); }
    stack & operator=( const stack<T> & o ) {
        if ( this == &o ) return *this;
        clear();
        copy( o );
        return *this;
    }
    bool empty() const {
        return head == nullptr;
    }
```

```
    void push( const T & value ) {
        head = new node{ value, head }; /***/
    }
    T pop() {
        node * n = head;
        head = n->next;
        T v = std::move( n->value );
        delete n;
        return v;
    }
};
```

A.4 C++obj

```
struct stack {
    struct node {
        ptr<object> value;
        node * next;
        node( const object & v, node * n = nullptr ) :
                value( v.new_copy() ), next( n ) {}
    };
    node * head;
    void copy( const stack & o ) {
        node ** cr = &head;
        for ( node * nx = o.head; nx; nx = nx->next ) {
            *cr = new node{ *nx->value }; /***/
            cr = &(*cr)->next;
        }
        *cr = nullptr;
    }
    void clear() {
        for ( node * nx = head; nx; ) {
            node * cr = nx;
            nx = cr->next;
            delete cr;
        }
        head = nullptr;
    }
    stack() : head( nullptr ) {}
    stack( const stack & o ) { copy( o ); }
    ~stack() { clear(); }
```

```
    stack & operator=( const stack & o ) {
        if ( this == &o ) return *this;
        clear();
        copy( o );
        return *this;
    }
    bool empty() const {
        return head == nullptr;
    }
    void push( const object & value ) {
        head = new node{ value, head }; /***/
    }
    ptr<object> pop() {
        node * n = head;
        head = n->next;
        ptr<object> v = std::move( n->value );
        delete n;
        return v;
    }
};
```