Type Resolution in C∀

by

Fangren Yu

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2025

© Fangren Yu 2025

Author's Declaration

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

C∀ (C-for-all) is an evolutionary extension of the C programming language, which introduces many modern programming language features to C. C∀ has a type system built around parametric polymorphism, and the polymorphic functions are prefixed by a **forall** declaration of type parameters, giving the language its name.

This thesis presents a series of work I did on type resolution in C∀. Every function, including the built-in C operators, can be overloaded in C∀, therefore resolving function overloads and generic type parameters is at the heart of C∀ expression analysis. This thesis focuses on the interactions of various C∀ language features such as reference and generic types in type resolution, analyzes the known issues and presents improvements to the type system that fix several of the problems. Ideas for future work are also given for further improving the consistency of the C∀ type-system at a language design level.

Acknowledgements

I would like to thank my supervisor Peter Buhr for his long-term support of my research, Ondrej Lhotak and Gregor Richards for reading and polishing this thesis, and most importantly, everyone in the C∀ team for actively pushing the limits of C∀'s type system. This work would not have been possible without the insights from the actual development practices. A special mention goes to Aaron Moss for laying the foundation of my work and helping me understand the working details of his prototype experiments.

Finally, a special thank you to Huawei Canada for funding this work.

Table of Contents

Author's Declaration ii

Abstract iii

Acknowledgements iv

List of Figures vii

List of Tables viii

1 Introduction 1

1.1 Types . . . 1

1.2 Overloading . . . 2

1.2.1 Operator Overloading . . . 6

1.2.2 Function Overloading . . . 6

1.2.3 Variable Overloading . . . 7

1.2.4 Constant Overloading . . . 8

1.3 Type Inferencing . . . 8

1.3.1 Type-Inferencing Issues . . . 10

1.4 Polymorphism . . . 11

1.4.1 Polymorphic Function . . . 11

1.4.2 Traits . . . 13

1.4.3 Generic Types . . . 13

1.5 Language Comparison . . . 15

1.6 Contributions . . . 18

2 C∀ Features and Type System Interactions **19**
2.1 Reference Types . . . 19
2.2 Tuple Types . . . 22
2.3 Tuple Implementation . . . 24
2.4 **inline** Substructure . . . 29

3 Resolution Algorithms **34**
3.1 Expression Cost-Model . . . 35
3.2 Type Unification . . . 42
3.3 Assertion Satisfaction . . . 44
3.3.1 Caching Assertion Resolution Results . . . 47
3.4 Compiler Implementation Considerations . . . 48

4 Related Work **50**
4.1 Polymorphism . . . 50
4.2 Tuples/Variadics . . . 51
4.3 Type Resolution . . . 52

5 Future Work **54**
5.1 Closed Trait Types . . . 54
5.2 Associated Types . . . 56
5.3 User-defined Conversions . . . 57

6 Conclusion **59**

References **60**

List of Figures

1.1 Implicit/Explicit Trait Inferencing 17

2.1 Nesting 24
2.2 Alternate Tuple Implementation 25
2.3 C Variadic Maximum Function 27
2.4 Nested Named Aggregate 30

3.1 C Expression Conversions: T1 operator T2 41
3.2 C∀ Total-Ordering Expression Conversions 41
3.3 C∀ Partial-Ordering Expression Conversions 41

5.1 basic_ostream trait 55

List of Tables

1.1 Overload Features in Programming Languages 15

3.1 Compile time of selected files by compiler build in seconds 35

Chapter 1

Introduction

This thesis is exploratory work I did to understand, fix, and extend the C∀ type-system, specifically, the *type-resolver* used to satisfy call-site assertions among overloaded variable and function names to allow polymorphic function calls. Assertions are the operations a function uses within its body to perform its computation. For example, a polymorphic function summing an array needs a size, zero, assignment, and plus for the array element-type, and a subscript operation for the array type.

```
T sum( T a[], size_t size ) {
    T total = { 0 };          // size, 0 for type T
    for ( i; size )
        total += a[i];        // + and subscript for T
    return total;
}
```

In certain cases, if the resolver fails to find an exact assertion match, it attempts to find a *best* match using reasonable type conversions. Hence, C∀ follows the current trend of replacing nominal inheritance with traits composed of assertions for type matching. The over-arching goal in C∀ is to push the boundary on localized assertion matching, with advanced overloading resolution and type conversions that match programmer expectations in the C programming language. Together, the resulting type-system has a number of unique features making it different from other programming languages with expressive, static, type-systems.

1.1 Types

All computers have multiple types because computer architects optimize the hardware around a few basic types with well defined (mathematical) operations: boolean, integral, floating-point, and occasionally strings. A programming language and its compiler present ways to declare types that ultimately map into the ones provided by the underlying hardware. These language types are thrust upon programmers with their syntactic/semantic rules and restrictions. These rules are then used to transform a language expression to a unique hardware expression. Modern programming-languages allow user-defined types and generalize across multiple types using polymorphism. Type systems can be static, where each variable has a fixed type during execution and an expression's type is determined at compile time, or dynamic, where each variable can change type during execution and so an expression's type is reconstructed on each evaluation. Expressibility,

generalization, and safety are all bound up in a language's type system, and hence, directly affect the capability, build time, and correctness of program development.

1.2 Overloading

> There are only two hard things in Computer Science: cache invalidation and *naming things*. — Phil Karlton

Overloading allows programmers to use the most meaningful names without fear of name clashes within a program or from external sources, like include files. Experience from C++ and C∀ developers shows the type system can implicitly and correctly disambiguate the majority of overloaded names, *i.e.*, it is rare to get an incorrect selection or ambiguity, even among hundreds of overloaded (variables and) functions. In many cases, a programmer is unaware of name clashes, as they are silently resolved, simplifying the development process.

Disambiguating among overloads is implemented by examining each call site and selecting the best matching overloaded function based on criteria like the types and number of arguments and the return context. Since the hardware does not support mixed-mode operands, such as `2 + 3.5`, the type system must disallow it or (safely) convert the operands to a common type. Like overloading, the majority of mixed-mode conversions are silently resolved, simplifying the development process. This approach matches with programmer intuition and expectation, regardless of any *safety* issues resulting from converted values. Depending on the language, mix-mode conversions can be explicitly controlled using a cast.

Namespace pollution refers to loading the global or other namespaces with many names, resulting in paranoia that the compiler could make wrong choices for overloaded names causing failure. This fear leads to coding styles where names are partitioned by language mechanisms and qualification is used to make names unique. This approach defeats the purpose of overloading and places an additional coding burden on both the developer and user. As well, many namespace systems provide a mechanism to open their scope returning to normal overloading, *i.e.*, no qualification. While namespace mechanisms are very important and provide a number of crucial program-development features, protection from overloading is overstated. Similarly, lexical nesting is another place where duplicate naming issues arise. For example, in object-oriented programming, class member names *shadow* names within members. Some programmers qualify all member names with `class::` or `this->` to make them unique from names defined in members. Even nested lexical blocks result in shadowing, *e.g.*, multiple nested loop-indices called `i`, silently changing the meaning of `i` at lower scope levels. Again, coding styles exist requiring all variables in nested block to be unique to prevent name shadowing problems. Depending on the language, these possible ambiguities can be reported (as warnings or errors) and resolved explicitly using some form of qualification and/or cast. For example, if variables can be overloaded, shadowed variables of different type can produce ambiguities, indicating potential problems in lower scopes.

Formally, overloading is defined by Strachey as one kind of *ad hoc polymorphism*:

> In ad hoc polymorphism there is no single systematic way of determining the type of the result from the type of the arguments. There may be several rules of limited

> extent which reduce the number of cases, but these are themselves ad hoc both in scope and content. All the ordinary arithmetic operators and functions come into this category. It seems, moreover, that the automatic insertion of transfer functions by the compiling system is limited to this. [35, p. 37]

where a *transfer function* is an implicit conversion to help find a matching overload:

> The problem of dealing with polymorphic operators is complicated by the fact that the range of types sometimes overlap. Thus for example 3 may be an integer or a real and it may be necessary to change it from one type to the other. The functions which perform this operation are known as transfer functions and may either be used explicitly by the programmer, or, in some systems, inserted automatically by the compiling system. [35, p. 35]

The differentiating characteristic between parametric polymorphism and overloading is often stated as: polymorphic functions use one algorithm to operate on arguments of many different types, whereas overloaded functions use a different algorithm for each type of argument. A similar differentiation is applicable for overloading and default parameters.

different implementations	**same implementation**
void foo(**int**);	**void** foo(**int**, **int** = 5); *// default value*
void foo(**int**, **int**);	

However, this distinguishing characteristic is vague. For example, should the operation abs be overloaded or polymorphic or both?

overloading	**polymorphic**
int abs(**int**);	**forall**(T | { **void** ?{}(T &, **zero_t**); **int** ?<?(T, T); T -?(T); })
double abs(**double**);	T abs(T);

Here, there are performance advantages for having specializations and code-reuse advantages for the generalization.

The Strachey definitions raise several questions.

1. Is overloading polymorphism?

   In type theory, polymorphism allows an overloaded type name to represent multiple different types. For example, generic types overload the type name for a container type.

   ```
   List<int> li;   List<double> ld;   List<struct S> ls;
   ```

   For subtyping, a derived type masquerades as a base type, where the base and derived names cannot be overloaded. Instead, the mechanism relies on structural typing among the types. In both cases, the polymorphic mechanisms apply in the type domain and the names are in the type namespace. In the following C example:

   ```
   struct S { int S; };
   struct S s;
   void S( struct S S ) { S.S = 1; };
   enum E { E };
   enum E e = E;
   ```

the overloaded names S and E are separated into the type and object domain, and C uses the type kinds **struct** and **enum** to disambiguate the names. In general, types are not overloaded because inferencing them is difficult to imagine in a statically typed programming language.

```
struct S { int i; }; // 1                    S s;
struct S { int i, j }; // 2                  if ( ... ) s = (S){ 3 }; // 1
union S { int i; double d; }; // 3           else s = 3; // 4
typedef int S; // 4
```

On the other hand, in ad-hoc overloading of variables and/or functions, the names are in the object domain and each overloaded object name has an anonymous associated type.

```
double foo( int );   char foo( void );   int foo( double, double );
double foo;   char foo;   int foo;
```

Here, the associated type cannot be extracted using **typeof/decltype** for typing purposes, as **typeof**(foo) is always ambiguous. To disambiguate the type of foo requires additional information from the call-site or a cast.

```
double d = foo( 3 );   char c = foo();   int i = foo( 3.5, 4.1 );  // exact match
d = foo;   c = foo;   i = foo;                // exact match
i = ((double (*)( int ))foo)( 3.5 );         // no exact match, explicit cast => conversions
```

Hence, depending on your perspective, overloading may not be polymorphism, as no single overloaded entity represents multiple types.

2. Does ad-hoc polymorphism have a single systematic way of determining the type of the result from the type of the arguments?

   For exact type matches in overloading, there is a systematic way of matching arguments to parameters, and a function denotes its return type rather than using type inferencing. This matching is just as robust as other polymorphic analysis. The ad-hoc aspect might be the implicit transfer functions (conversions) applied to arguments to create an exact parameter type-match, as there may be multiple conversions leading to different exact matches. Note, conversion issues apply to both non-overloaded and overloaded functions. Here, the selection of the conversion functions can be considered the *opinion* of the language (type system), even if the technique used is based on sound rules, like maximizing conversion accuracy (non-lossy). The difference in opinion results when the language conversion rules differ from a programmer's expectations. However, without implicit conversions, programmers may have to write an exponential number of functions covering all possible exact-match cases among all reasonable types. C∀'s *opinion* on conversions must match C's and then rely on programmers to understand the effects. That is, let the compiler do the heavy-lifting of selecting a *best* set of conversions that maximize safety. Hence, removing implicit conversions from C∀ is not an option (as is true in most other programming languages), so it must do the best possible job to get it right.

3. Why are there two forms of *overloading* (general and parametric) in different programming languages?

   General overloading occurs when the type-system *knows* a function's parameters and return types (or a variable's type for variable overloading). In functional programming-languages, there is always a return type. If a return type is specified, the compiler does not have to inference the function body. For example, the compiler has complete knowledge about builtin

types and their overloaded arithmetic operators. In this context, there is a fixed set of overloads for a given name that are completely specified. Overload resolution then involves finding an exact match between a call and the overload prototypes based on argument type(s) and possibly return context. If an *exact* match is not found, the call is either ill formed (ambiguous) or further attempts are made to find a *best* match using transfer functions (conversions). As a consequence, no additional runtime information is needed per call, *i.e.*, the call is a direct transfer (branch) with possibly converted arguments.

Parametric overloading occurs when the type-system does not know a function's parameter and return types. Types are inferred from constant types and/or constraint information that includes typing. Parametric overloading starts with a universal/generic type used to define parameters and returns.

```
forall( T ) T identity( T t ) { return t; }
int i = identify( 3 );
double d;
double e = identity( d );
```

Here, the type system is substituting the argument type for T, and ensuring the return type also matches. This mechanism can be extended to overloaded parametric functions with multiple type parameters.

```
forall( T ) [T, T] identity( T t1, T t2 ) { return [t1, t2 ]; } // return tuple
int j;
[i, j] = identity( 3, 4 );
forall( T, U ) [T, U] identity( T t1, U t2 ) { return [t1, t2 ]; }
[i, d] = identity( i, d );
```

Here, the type system is using techniques from general overloading, number and argument types, to disambiguate the overloaded parametric functions. However, this exercise is moot because the types are so universal they have no practical operations within the function, modulo any general operations carried by every type, like copy or print. This kind of polymorphism is restricted to moving type instances as abstract entities; if type erasure is used, there is no way to recover the original type to perform useful actions on its value(s). Hence, parametric overloading requires additional information about the universal types to make them useful.

This additional information often comes as a set of operations that must be supply for a type, *e.g.*, C∀/Rust/Go have traits, C++ template has concepts, Haskell has type-classes. These operations can then be used in the body of the function to manipulate the type's value. Here, a type binding to T must have available a ++ operation with the specified signature.

```
forall( T | T ?++( T, T ) ) // trait
T inc( T t ) { return t++; } // change type value
int i = 3
i = inc( i )
double d = inc( 3.5 );
```

Given a qualifying trait, are its elements inferred or declared? In the example, the type system infers **int** for T, infers it needs an appropriately typed ++ operator, and finds it in the enclosing environment, possibly in the language's prelude defining basic types and their operations. This implicit inferencing is expensive if matched with implicit conversions when there is no exact match. Alternatively, types opt-in to traits via declarations.

```
trait postinc { T ?++( T, T ) }
type int provide postinc;
```

In this approach, after inferring **int** for T, **int** is examined for any necessary traits required in the function, so there is no searching. Once all traits are satisfied, the compiler can inline them, pass functions pointers directly, or pass a virtual table composing the trait.

The following sections illustrate how overloading is provided in C∀.

1.2.1 Operator Overloading

Many programming languages provide general overloading of the arithmetic operators [42] across the basic computational types using the number and type of parameters and returns. However, in some languages, arithmetic operators may not be first class, and hence, cannot be overloaded. Like C++, C∀ allows general operator overloading for user-defined types. The C∀ syntax for operator names uses the `'?'` character to denote a parameter, *e.g.*, left and right unary operators: ?++ and ++?, and binary operators ?+? and ?<=?. Here, a user-defined type is extended with an addition operation with the same syntax as a builtin type.

```
struct S { int i, j };
S ?+?( S op1, S op2 ) { return (S){ op1.i + op2.i, op1.j + op2.j }; }
S s1, s2;
s1 = s1 + s2;          // infix call
s1 = ?+?( s1, s2 );    // direct call
```

C∀ excludes overloading the comma operator, short-circuit logical operators && and ||, and ternary conditional operator ? (*e.g.*, max = x > y ? x : y), all of which are short-hand control structures rather than operators. C++ *does* overload the comma and short-circuit logical operators, where overloading the comma operator is esoteric and rarely used, and the short-circuit operators do not exhibit short-circuit semantics, both of which are confusing and/or inconsistent.

1.2.2 Function Overloading

Many programming languages provide general overloading for functions [41], as long as their prototypes differ in the number and type of parameters. A few programming languages also use the return type for selecting overloaded functions (see below).

```
void f( void );          // (1): no parameter
void f( char );          // (2): overloaded on the number and parameter type
void f( int, int );      // (3): overloaded on the number and parameter type
f( 'A' );                // select (2)
```

The type system examines each call site and first looks for an exact match and then a best match using conversions.

Ada, Scala, and C∀ type-systems also use the return type in resolving a call, to pinpoint the best overloaded name. Essentially, the return types are *reversed curried* into output parameters of the function. For example, in many programming languages with overloading, the following functions are ambiguous without using the return type.

```
int f( int );                   // (1); overloaded on return type and parameter
double f( int );                // (2); overloaded on return type and parameter
int i = f( 3 );                 // select (1)
double d = f( 3 );              // select (2)
```

Alternatively, if the type system uses the return type, there is an exact match for each call, which again matches with programmer intuition and expectation. This capability can be taken to the extreme, where the only differentiating factor is the return type.

```
int random( void );             // (1); overloaded on return type
double random( void );          // (2); overloaded on return type
int i = random();               // select (1)
double d = random();            // select (2)
```

Again, there is an exact match for each call. As for operator overloading, if there is no exact match, a set of minimal implicit conversions can be added to find a best match.

```
short int = random();           // select (1), unsafe
long double = random();         // select (2), safe
```

1.2.3 Variable Overloading

Unlike most programming languages, C∀ has variable overloading within a scope, along with shadow overloading in nested scopes. Shadow overloading is also possible for functions, in languages supporting nested-function declarations, *e.g.*, C++ named, nested, lambda functions.

```
void foo( double d );
int v;                          // (1)
double v;                       // (2) variable overloading
foo( v );                       // select (2)
{
    int v;                      // (3) shadow overloading
    double v;                   // (4) and variable overloading
    foo( v );                   // select (4)
}
```

It is interesting that shadowing (see namespace pollution in Section 1.2, p. 2) is considered a normal programming-language feature with only slight software-engineering problems. However, variable overloading within a scope is often considered dangerous, without any evidence to corroborate this claim. In contrast, function overloading in C++ occurs silently within the global scope from **#include** files all the time without problems.

In C∀, the type system simply treats an overloaded variable as an overloaded function returning a value with no parameters. Hence, no effort is required to support this feature as it is available for differentiating among overloaded functions with no parameters.

```
int MAX = 2147483647;           // (1); overloaded on return type
long int MAX = ...;             // (2); overloaded on return type
double MAX = ...;               // (3); overloaded on return type
int i = MAX;                    // select (1)
long int i = MAX;               // select (2)
double d = MAX;                 // select (3)
```

Hence, the name MAX can replace all the C type-specific names, *e.g.*, INT_MAX, LONG_MAX, DBL_MAX, *etc*. The result is a significant reduction in names to access common constants.

1.2.4 Constant Overloading

C∀ is unique in providing restricted constant overloading for the values 0 and 1, which have special status in C. For example, the value 0 is both an integer and a pointer literal, so its meaning depends on context. In addition, several operations are defined in terms of values 0 and 1. For example, **if** and iteration statements in C compare the condition with 0, and the increment and decrement operators are semantically equivalent to adding or subtracting the value 1.

```
if ( x ) ++x;       ⇒   if ( x != 0 ) x += 1;
for ( ; x; --x )    ⇒   for ( ; x != 0; x -= 1 )
```

To generalize this feature, both constants are given types **zero_t** and **one_t** in C∀, which allows overloading various operations for new types that seamlessly work within the special 0 and 1 contexts. The types **zero_t** and **one_t** have special builtin implicit conversions to the various integral types, and a conversion to pointer types for 0, which allows standard C code involving 0 and 1 to work.

```
struct S { int i, j; };
void ?{}( S & s, zero_t ) { s.[i,j] = 0; }      // constant constructors
void ?{}( S & s, one_t ) { s.[i,j] = 1; }
S ?=?( S & dst, zero_t ) { dst.[i,j] = 0; return dst; } // constant assignments
S ?=?( S & dst, one_t ) { dst.[i,j] = 1; return dst; }
S ?+=?( S & s, one_t ) { s.[i,j] += 1; return s; } // increment/decrement each field
S ?-=?( S & s, one_t ) { s.[i,j] -= 1; return s; }
int ?!=?( S s, zero_t ) { return s.i != 0 && s.j != 0; } // constant comparison
S s = 0;                                        // initialization
s = 0;                                          // assignments
s = 1;
if ( s ) ++s;                                   // unary ++/- - come implicitly from +=/-=
```

Here, type S is first-class with respect to the basic types, working with all existing implicit C mechanisms.

1.3 Type Inferencing

Every variable has a type, but association between them can occur in different ways: at the point where the variable comes into existence (declaration) and/or on each assignment to the variable.

```
double x;                                       // type only
float y = 3.1D;                                 // type and initialization
auto z = y;                                     // initialization only
z = "abc";                                      // assignment
```

For type-only, the programmer specifies the initial type, which remains fixed for the variable's lifetime in statically typed languages. For type-and-initialization, the specified and initialization types may not agree requiring an implicit/explicit conversion. For initialization-only, the compiler may select the type by melding programmer and context information. When the compiler

participates in type selection, it is called *type inferencing*. Note, type inferencing is different from type conversion: type inferencing *discovers* a variable's type before setting its value, whereas conversion has two typed variables and performs a (possibly lossy) value conversion from one type to the other. Finally, for assignment, the current variable and expression types may not agree. Discovering a variable or function type is complex and has limitations. The following covers these issues, and why this scheme is not amenable with the C∀ type system.

One of the first and most powerful type-inferencing systems is Hindley–Milner [7]. Here, the type resolver starts with the types of the program constants used for initialization and these constant types flow throughout the program, setting all variable and expression types.

```
auto f() {
    x = 1;   y = 3.5;                   // set types from constants
    x = // expression involving x, y and other local initialized variables
    y = // expression involving x, y and other local initialized variables
    return x, y;
}
auto w = f();                           // typing flows outwards

void f( auto x, auto y ) {
    x = // expression involving x, y and other local initialized variables
    y = // expression involving x, y and other local initialized variables
}
s = 1;   t = 3.5;                       // set types from constants
f( s, t );                              // typing flows inwards
```

In both overloads of f, the type system works from the constant initializations inwards and/or outwards to determine the types of all variables and functions. Like template meta-programming, there can be a new function generated for the second f depending on the types of the arguments, assuming these types are meaningful in the body of f. Inferring type constraints, by analysing the body of f is possible, and these constraints must be satisfied at each call site by the argument types; in this case, parametric polymorphism can allow separate compilation. In languages with type inferencing, there is often limited overloading to reduce the search space, which introduces the naming problem. Note, return-type inferencing goes in the opposite direction to Hindley–Milner: knowing the type of the result and flowing back through an expression to help select the best possible overloads, and possibly converting the constants for a best match.

There are multiple ways to indirectly specify a variable's type, *e.g.*, from a prior variable or expression.

gcc / C∀	**C++**	
`#define expr 3.0 * i`		
`typeof(expr) x = expr;`	`auto x = 3.0 * i;`	`// use type of initialization expression`
`int y;`	`int y;`	
`typeof(y) z = y;`	`auto z = y;`	`// use type of initialization expression`

Here, type(expr) computes the same type as **auto** righ-hand expression. The advantages are:

- Not determining or writing long generic types, *e.g.*, given deeply nested generic types.

```
typedef T1(int).T2(float).T3(char).T ST;  // C∀ nested type declaration
ST x, y, x;
```

This issue is exaggerated with C++ templates, where type names are 100s of characters long, resulting in unreadable error messages.

- Ensuring the type of secondary variables match a primary variable.

```
int x;                          // primary variable
typeof(x) y, z, w;              // secondary variables match x's type
```

If the type of x changes, the type of the secondary variables correspondingly updates. There can be strong software-engineering reasons for binding the types of these variables.

Note, the use of **typeof** is more restrictive, and possibly safer, than general type-inferencing.

```
int x;                                  int x;
type(x) y = ... // complex expression   auto y = ... // complex expression
type(x) z = ... // complex expression   auto z = ... // complex expression
```

On the left, the types of y and z are fixed (branded), whereas on the right, the types of y and z can fluctuate.

1.3.1 Type-Inferencing Issues

Each kind of type-inferencing system has its own set of issues that affect the programmer in the form of convenience, restrictions, or confusions.

A convenience is having the compiler use its overarching program knowledge to select the best type for each variable based on some notion of *best*, which simplifies the programming experience.

A restriction is the conundrum in type inferencing of when to *brand* a type. That is, when is the type of the variable/function more important than the type of its initialization expression(s). For example, if a change is made in an initialization expression, it can cascade type changes producing many other changes and/or errors. At some point, a variable's type needs to remain constant and the initializing expression needs to be modified or be in error when it changes. Often type-inferencing systems allow restricting (*branding*) a variable or function type, so the compiler can report a mismatch with the constant initialization.

```
void f( int x, int y ) {  // brand function prototype
    x = // expression involving x, y and other local initialized variables
    y = // expression involving x, y and other local initialized variables
}
s = 1;   t = 3.5;
f( s, t ); // type mismatch
```

In Haskell, it is common for programmers to brand (type) function parameters.

A confusion is blocks of code where all declarations are **auto**, as is now common in C++. As a result, understanding and changing the code becomes almost impossible. Types provide important clues as to the behaviour of the code, and correspondingly to correctly change or add new code. In these cases, a programmer is forced to re-engineer types, which is fragile, or rely on an IDE that can re-engineer types for them. For example, given:

```
auto x = ...
```

and the need to write a function to compute using x

```
void rtn( type of x parm );
rtn( x );
```

A programmer must re-engineer the type of x's initialization expression, reconstructing the possibly long generic type-name. In this situation, having the type name or its short alias is essential.

C∀'s type system tries to prevent type-resolution mistakes by relying heavily on the type of the left-hand side of assignment to pinpoint correct types within an expression. Type inferencing defeats this goal because there is no left-hand type. Fundamentally, type inferencing tries to remove explicit typing from programming. However, writing down types is an important aspect of good programming, as it provides a check of the programmer's expected type and the actual type. Thinking carefully about types is similar to thinking carefully about date structures, often resulting in better performance and safety. Similarly, thinking carefully about storage management in unmanaged languages is an important aspect of good programming, versus implicit storage management (garbage collection) in managed language.[1] Given **typedef** and **typeof**, and the strong desire to use the left-hand type in resolution, no attempt has been made in C∀ to support implicit type-inferencing. Should a significant need arise, this decision can be revisited.

1.4 Polymorphism

C∀ provides polymorphic functions and types, where a polymorphic function can constrain types using assertions based on traits.

1.4.1 Polymorphic Function

The signature feature of the C∀ type-system is parametric-polymorphic functions [5, 6, 9], generalized using a **forall** clause (giving the language its name).

```
forall( T ) T identity( T val ) { return val; }
int forty_two = identity( 42 );               // T is bound to int, forty_two == 42
```

This identity function can be applied to an *object type*, *i.e.*, a type with a known size and alignment, which is sufficient to stack allocate, default or copy initialize, assign, and delete. The C∀ implementation passes the size and alignment for each type parameter, as well as auto-generated default and copy constructors, assignment operator, and destructor. For an incomplete *data type*, *e.g.*, pointer/reference types, this information is not needed.

```
forall( T * ) T * identity( T * val ) { return val; }
int i, * ip = identity( &i );
```

Unlike C++ template functions, C∀ polymorphic functions are compatible with *separate compilation*, preventing compilation and code bloat.

To constrain polymorphic types, C∀ uses *type assertions* [33, pp. 37-44] to provide further type information, where type assertions may be variable or function declarations that depend

[1] There are full-time Java consultants, who are hired to find memory-management problems in large Java programs, *e.g.*, Monika Beckworth.

on a polymorphic type variable. Here, the function twice works for any type T with a matching addition operator.

```
forall( T | { T ?+?(T, T); } ) T twice( T x ) { return x + x; }
int val = twice( twice( 3 ) );              // val == 12
```

The closest approximation to parametric polymorphism and assertions in C is type-unsafe (**void** *) functions, like qsort for sorting an array of unknown values.

```
void qsort( void * base, size_t nmemb, size_t size, int (*cmp)( const void *, const void * ) );
```

Here, the polymorphism is type-erasure, and the parametric assertion is the comparison function, which is explicitly passed.

```
enum { N = 5 };
double val[N] = { 5.1, 4.1, 3.1, 2.1, 1.1 };
int cmp( const void * v1, const void * v2 ) { // compare two doubles
    return *(double *)v1 < *(double *)v2 ? -1 : *(double *)v2 < *(double *)v1 ? 1 : 0;
}
qsort( val, N, sizeof( double ), cmp );
```

The equivalent type-safe version in C∀ is a wrapper over the C version.

```
forall( ET | { int ?<?( ET, ET ); } )       // type must have < operator
void qsort( ET * vals, size_t dim ) {
    int cmp( const void * t1, const void * t2 ) { // nested function
        return *(ET *)t1 < *(ET *)t2 ? -1 : *(ET *)t2 < *(ET *)t1 ? 1 : 0;
    }
    qsort( vals, dim, sizeof(ET), cmp );    // call C version
}
qsort( val, N );                            // deduct type double, and pass builtin < for double
```

The nested function cmp is implicitly built and provides the interface from typed C∀ to untyped (**void** *) C. Providing a hidden cmp function in C++ is awkward as lambdas do not use C calling conventions and template declarations cannot appear in block scope. Call-site inferencing and nested functions provide a localized form of inheritance. For example, the C∀ qsort can be made to sort in descending order by locally changing the behaviour of <.

```
{
    int ?<?( double x, double y ) { return x > y; } // locally override behaviour
    qsort( vals, 10 );                      // descending sort
}
```

The local version of ?<? overrides the built-in ?<? so it is passed to qsort. The local version performs ?>?, making qsort sort in descending order. Hence, any number of assertion functions can be overridden locally to maximize the reuse of existing functions and types, without the construction of a named inheritance hierarchy. A final example is a type-safe wrapper for C malloc, where the return type supplies the type/size of the allocation, which is impossible in most type systems.

```
static inline forall( T & | sized(T) )
T * malloc( void ) {
    if ( _Alignof(T) <= __BIGGEST_ALIGNMENT__ ) return (T *)malloc( sizeof(T) ); // C allocation
    else return (T *)memalign( _Alignof(T), sizeof(T) );
}
```

```
// select type and size from left-hand side
int * ip = malloc(); double * dp = malloc(); [[aligned(64)]] struct S {...} * sp = malloc();
```

The sized assertion passes size and alignment as a data object has no implicit assertions. Both assertions are used in malloc via **sizeof** and **_Alignof**. In practice, this polymorphic malloc is unwrapped by the C compiler and the **if** statement is elided producing a type-safe call to malloc or memalign.

This mechanism is used to construct type-safe wrapper-libraries condensing hundreds of existing C functions into tens of C∀ overloaded functions. Here, existing C legacy code is leveraged as much as possible; other programming languages must build supporting libraries from scratch, even in C++.

1.4.2 Traits

C∀ provides *traits* to name a group of type assertions, where the trait name allows specifying the same set of assertions in multiple locations, preventing repetition mistakes at each function declaration.

```
forall( T ) trait sumable {
    void ?{}( T &, zero_t ); // 0 literal constructor
    T ?+=?( T &, T );        // assortment of additions
    T ?+?( T, T );
    T ++?( T & );
    T ?++( T & );
};
```

```
forall( T | sumable( T ) )   // use trait
T sum( T a[], size_t size ) {
    T total = 0;         // initialize by 0 constructor
    for ( i; size )
        total += a[i]; // select appropriate +
    return total;
}
```

Traits are implemented by flattening them at use points, as if written in full by the programmer. Flattening often results in overlapping assertions, *e.g.*, operator +. Hence, trait names play no part in type equivalence. In the example, type T is an object type, and hence, has the implicit internal trait otype.

```
trait otype( T & | sized(T) ) {
    void ?{}( T & );                    // default constructor
    void ?{}( T &, T );                 // copy constructor
    void ?=?( T &, T );                 // assignment operator
    void ^?{}( T & );                   // destructor
};
```

These implicit functions are used by the sumable operator ?+=? for the right side of ?+=? and return.

If the array type is not a builtin type, an extra type parameter and assertions are required, like subscripting. This case is generalized in polymorphic container-types, such as a list with insert and remove operations, and an element type with copy and assignment.

1.4.3 Generic Types

A significant shortcoming of 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.

1. Write bespoke data structures for each context. While this approach is flexible and supports integration with the C type checker and tooling, it is tedious and error prone, especially for more complex data structures.
2. Use **void** *-based polymorphism, *e.g.*, the C standard library functions bsearch and qsort, which allow for the reuse of code with common functionality. However, this approach eliminates the type checker's ability to ensure argument types are properly matched, often requiring a number of extra function parameters, pointer indirection, and dynamic allocation that is otherwise unnecessary.
3. Use an internal macro capability, like C++ templates, to generate code that is both generic and type checked, but errors may be difficult to interpret. Furthermore, writing complex template macros is difficult and complex.
4. Use an external macro capability, like M4 [19], to generate code that is generic code, but errors may be difficult to interpret. Like internal macros, writing and using external macros is equally difficult and complex.

C++, Java, and other languages use *generic types* to produce type-safe abstract data-types. C∀ generic types integrate efficiently and naturally with the existing polymorphic functions, while retaining backward compatibility with C and providing separate compilation. For concrete parameters, the generic-type definition can be inlined, like C++ templates, if its definition appears in a header file (*e.g.*, **static inline**).

A generic type can be declared by placing a **forall** specifier on a **struct** or **union** declaration and instantiated using a parenthesized list of types after the type name.

```
forall( F, S ) struct pair {
    F first;  S second;
};
forall( F, S ) // object
S second( pair( F, S ) p ) { return p.second; }
forall( F *, S * ) // sized
S * second( pair( F *, S * ) p ) { return p.second; }
```

```
pair( double, int ) dpr = { 3.5, 42 };
int i = second( dpr );
pair( void *, int * ) vipr = { 0p, &i };
int * ip = second( vipr );
double d = 1.0;
pair( int *, double * ) idpr = { &i, &d };
double * dp = second( idpr );
```

C∀ generic types are *fixed* or *dynamic* sized. Fixed-size types have a fixed memory layout regardless of type parameters, whereas dynamic types vary in memory layout depending on the type parameters. For example, the type variable T * is fixed size and is represented by **void** * in code generation; whereas, the type variable T is dynamic and set at the point of instantiation. The difference between fixed and dynamic is the complexity and cost of field access. For fixed, field offsets are computed (known) at compile time and embedded as displacements in instructions. For dynamic, field offsets are compile-time computed at the call site, stored in an array of offset values, passed as a polymorphic parameter, and added to the structure address for each field dereference within a polymorphic function. See [24, § 3.2] for complete implementation details.

Currently, C∀ generic types allow assertion. For example, the following declaration of a sorted set-type ensures the set key supports equality and relational comparison.

```
forall( Elem, Key | { _Bool ?==?( Key, Key ); _Bool ?<?( Key, Key ); } )
struct Sorted_Set { Elem elem; Key key; ... };
```

Table 1.1: Overload Features in Programming Languages

Feature \ Language	Ada	C++	C∀	Java	Scala	Swift	Rust	Haskell
general[a]	O[b]/F	O/F/M	V/O/F	M[c]	O/M	O/F/M	no	no
general constraints[d]	T/#//R[e]	T/#	T/#/R	T/#	T/#/N/R	T/#/N/R	T/#/N	T/R
univ. type constraints[f]	no[g]	C	T	B	B	T	T	T
Safe/Unsafe arg. conv.	no	S/U[h]	S/U	S[i]	S	no[j]	no	no

[a] overloadable entities: V ⇒ variable, O ⇒ operator, F ⇒ function, M ⇒ member
[b] except assignment
[c] not universal
[d] T ⇒ parameter type, # ⇒ parameter count, N ⇒ parameter name; R ⇒ return type
[e] parameter names can be used to disambiguate among overloads but not create overloads
[f] A ⇒ concept, T ⇒ interface/trait/type-class, B ⇒ type bounds
[g] generic cannot be overloaded
[h] no conversions allowed during template parameter deduction
[i] unsafe (narrowing) conversion only allowed in assignment or initialization to a primitive (builtin) type
[j] literals only, Int → Double (Safe)

However, the operations that insert/remove elements from the set should not appear as part of the generic-types assertions.

```
forall( Elem | /* any assertions on element type */ ) {
    void insert( Sorted_Set set, Elem elem ) { ... }
    bool remove( Sorted_Set set, Elem elem ) { ... } // false ⇒ element not present
    ... // more set operations
} // distribution
```

(Note, the **forall** clause can be distributed across multiple functions.) For software-engineering reasons, the set assertions would be refactored into a trait to allow alternative implementations, like a Java **interface**.

In summation, the C∀ type system inherits *nominal typing* for concrete types from C; however, without inheritance in C∀, nominal typing cannot be extended to polymorphic subtyping. Instead, C∀ adds *structural typing* and uses it to generate polymorphism. Here, traits are like interfaces in Java or abstract base-classes in C++, but without the nominal inheritance relationships. Instead, each polymorphic function or generic type defines the structural requirements needed for its execution, which is fulfilled at each call site from the lexical environment, like Go [13] or Rust [29] interfaces. Hence, lexical scopes and nested functions are used extensively to mimic subtypes, as in the qsort example, without managing a nominal inheritance hierarchy.

1.5 Language Comparison

Because C∀'s type system is focused on overloading and overload resolution, Table 1.1 compares C∀ with a representative set of popular programming languages and their take on overloading. The first row classifies whether there is general overloading, and what entities may be overloaded.

variable	operator	function	member
int foo;	**int** ?+?(**int**);	**int** foo(**int**);	**class** C {
double foo;	**double** ?+?(**double**);	**double** foo(**int**, **int**);	**int** foo(**int**);
			double foo(**int**, **int**);
			};

The second row classifies the specialization mechanisms used to distinguish among the general overload capabilities.

type	number	name	return
func foo(_ x : Int)	**func** foo(_ x : Int)	**func** foo(x : Int)	**func** foo() -> Int
func foo(_ x : Double)	**func** foo(_ x : Int, _ y : Int)	**func** foo(y : Int)	**func** foo() -> String
foo(3)	foo(3)	foo(x : 3)	**var** i : Int = foo()
foo(3.5)	foo(3, 3)	foo(y : 3);	**var** s : String = foo();

The third row classifies if generic functions can be overloaded based on differing type variables, number of type variables, and type-variable constraints. The following examples illustrates the first two overloading cases.

```
func foo<T>( a : T );              // foo1
func foo<T>( a : T,  b : T );      // foo2
func foo<T,U>( a : T,  b :  U );   // foo3
foo( a : 3.5 );                    // foo1
foo( a : 2, b : 2 );               // foo2
foo( a : 2, b : 2.5 );             // foo3
```

Here, the overloaded calls are disambiguated using argument types and their number.

However, the parameter operations are severely restricted because universal types have few operations. For example, Swift provides a print operation for its universal type, and the Java Object class provides general methods: toString, hashCode, equals, finalize, *etc*. This restricted mechanism still supports a few useful functions, where the parameters are abstract entities, *e.g.*:

```
func swap<T>( _ a : inout T, _ b : inout T ) { let temp : T = a;  a = b;  b = temp; }
var ix = 4, iy = 3;
swap( &ix, &iy );
var fx = 4.5, fy = 3.5;
swap( &fx, &fy );
```

To make a universal function useable, an abstract description is needed for the operations used on the parameters within the function body. Type matching these operations can be done by using techniques like C++ template expansion, or explicit stating, *e.g.*, interfaces, subtyping (inheritance), assertions (traits), type classes, type bounds. The mechanism chosen can affect separate compilation or require runtime type information (RTTI).

concept

```
concept range = requires( T& t ) {
    ranges::begin(t); // equality-preserving for forward iterators
    ranges::end (t);
};
```

traits/type-classes

C∀

```
forall( T ) trait sumable {
    void ?{}( T &, zero_t );
    T ?+=?( T &, T );  };
forall( T | sumable( T ) )
T sum( T a[], size_t size ) {
    T total = 0;
    for ( i; size ) total += a[i];
    return total;  }
struct S { int i, j; };
void ?{}( S & s, zero_t ) { s.[i, j] = 0; }
void ?{}( S & s ) { s{0}; }
void ?{}( S & s, int i, int j ) { s.[i, j] = [i, j]; }
S ?+=?( S & l, S r ) { l.[i, j] += r.[i, j]; }
int main() {        // trait inferencing
    sout | sum( (int []){ 1, 2, 3 }, 3 );
    sout | sum( (double []){ 1.5, 2.5, 3.5 }, 3 );
    sout | sum( (S []){ {1,1}, {2,2}, {3,3} }, 3 ).[i, j];  }
```

Haskell

```
class Sumable a where
    szero :: a
    sadd :: a -> a -> a
ssum ::  Sumable a => [a] -> a
ssum (x:xs) = sadd x (ssum xs)
ssum [] = szero
data S = S Int Int deriving Show
instance Sumable Int where
    szero = 0
    sadd = (+)
instance Sumable Float where
    szero = 0.0
    sadd = (+)
instance Sumable S where
    szero = S 0 0
    sadd (S l1 l2) (S r1 r2) = S (l1 + r1) (l2 + r2)
main = do
    print ( ssum [1::Int, 2, 3] )
    print ( ssum [1.5::Float, 2.5, 3.5] )
    print ( ssum [(S 1 1), (S 2 2), (S 3 3)] )
```

Figure 1.1: Implicit/Explicit Trait Inferencing

```
forall( T | { T ?+?( T, T ) } )
```

inheritance/type bounds

```
class CarPark[Vehicle >: bike <: bus](val lot : Vehicle)
```

One differentiating feature among these specialization techniques is the ability to implicitly or explicitly infer the trait information at a class site. Figure 1.1 compares the sumable trait and polymorphic sum function (see Section 1.4.2, p. 13) for C∀ and Haskell. Here, the C∀ type system inferences the trait functions at each call site, so no additional specification is necessary by the programmer. The Haskell program requires the programmer to explicitly bind the trait and to each type that can be summed. (The extra casts in the Haskell program are necessary to differentiate between the two kinds of integers and floating-point numbers.) Ada also requires explicit binding, creating a new function name for each generic instantiation.

```
function int_Twice is new Twice( Integer, "+" ⇒ "+" );
function float_Twice is new Twice( Float, "+" ⇒ "+" );
```

Finally, there is a belief that certain type systems cannot support general overloading, *e.g.*, Haskell. As Table 1.1, p. 15 shows, there are multiple languages with both general and parametric overloading, so the decision to not support general overloading is based on design choices made by the language designers not any reason in type theory.

The fourth row classifies if conversions are attempted beyond exact match.

```
int foo( double );        // 1
double foo( int );        // 2
int i = foo( 3 );         // 1 : 3 to 3.0 argument conversion => all conversions are safe
```

```
double d = foo( 3.5 );    // 1 : int to double result conversion => all conversions are safe
```

1.6 Contributions

The C∀ compiler performance and type capability have been greatly improved through my development work.

1. The compilation time of various C∀ library units and test programs has been reduced by an order of magnitude, from minutes to seconds (see Table 3.1, p. 35), which made it possible to develop and test more complicated C∀ programs that utilize sophisticated type system features. The details of compiler optimization work are covered in a previous technical report [43], which essentially forms part of this thesis.
2. This thesis presents a systematic review of the new features added to the C∀ language and its type system. Some of the more recent inclusions to C∀, such as tuples and generic structure types, were not well tested during development due to the limitation of compiler performance. Several issues coming from the interactions of various language features are identified and discussed in this thesis; some of them I have resolved, while others are given temporary fixes and need to be reworked in the future.
3. Finally, this thesis provides constructive ideas for fixing a number of high-level issues in the C∀ language design and implementation, and gives a path for future improvements to the language and compiler.

Chapter 2

C∀ Features and Type System Interactions

This chapter discusses C∀ features introduced over time by multiple people and their interactions with the type system.

2.1 Reference Types

Reference types were added to C∀ by Robert Schluntz and Aaron Moss [25]. The C∀ reference type generalizes the C++ reference type (and its equivalent in other modern programming languages) by providing both mutable and immutable forms and cascading referencing and dereferencing. Specifically, C∀ attempts to extend programmer intuition about pointers to references. That is, use a pointer when its primary purpose is manipulating the address of storage, *e.g.*, a top/head/tail pointer or link field in a mutable data structure. Here, manipulating the pointer address is the primary operation, while dereferencing the pointer to its value is the secondary operation. For example, *within* a data structure, *e.g.*, stack or queue, all operations involve pointer addresses and the pointer may never be dereferenced because the referenced object is opaque. Alternatively, use a reference when its primary purpose is to alias a value, *e.g.*, a function parameter that does not copy the argument, for performance reasons. Here, manipulating the value is the primary operation, while changing the pointer address is the secondary operation. Succinctly, if the address changes often, use a pointer; if the value changes often, use a reference. Java has mutable references but no pointers. C++ has mutable pointers but immutable references; here, references match with functional programming. However, the consequence is asymmetric semantics between pointer and reference. C∀ adopts a uniform policy between pointers and references where mutability is a separate property made at the declaration.

The following examples show how pointers and references are treated uniformly in C∀.

```
1  int x = 1, y = 2, z = 3;
2  int * p1 = &x, ** p2 = &p1,  *** p3 = &p2,   // pointers to x
3      & r1 = x,  && r2 = r1,   &&& r3 = r2;   // references to x
4  int * p4 = &z, & r4 = z;

6  *p1 = 3; **p2 = 3; ***p3 = 3;              // different ways to change x to 3
7   r1 =  3; r2 = 3;       r3 = 3;           // change x: implicit dereference *r1, **r2, ***r3
8  **p3 = &y;     *p3 = &p4;                    // change p1, p2
9  // cancel implicit dereferences (&*)**r3, (&(&*)*r3, &(&*)r4
10 &r3 = &y; &&r3 = &&r4;                     // change r1, r2
```

Like pointers, references can be cascaded, *i.e.*, a reference to a reference, *e.g.*, && r2.[1] Usage of a reference variable automatically performs the same number of dereferences as the number of references in its declaration, *e.g.*, r2 becomes **r2. Finally, to reassign a reference's address needs a mechanism to stop the auto-referencing, which is accomplished by using a single reference to cancel all the auto-dereferencing, *e.g.*, &r3 = &y resets r3's address to point to y. C∀'s reference type (including multi-de/references) is powerful enough to describe the lvalue rules in C by types only. As a result, the C∀ type checker now works on just types without using the notion of lvalue in an expression. (C∀ internals still use lvalue for code generation purposes.)

The current reference typing rules in C∀ are summarized as follows:

1. For a variable x with declared type T, the variable-expression x has type reference to T, even if T itself is a reference type.
2. For an expression e with type $T\ \&_1 ... \&_n$, *i.e.*, T followed by n references, where T is not a reference type, the expression $\&T$ (address of T) has type $T*$ followed by $n-1$ references.
3. For an expression e with type $T*\&_1 ... \&_n$, *i.e.*, $T*$ followed by n references, the expression $*T$ (dereference T) has type T followed by $n+1$ references. This rule is the reverse of the previous rule, such that address-of and dereference operators are perfect inverses.
4. When matching argument and parameter types at a function call, the number of references on the argument type is stripped off to match the number of references on the parameter type.[2] In an assignment context, the left-hand-side operand-type is always reduced to a single reference.

Under this ruleset, a type parameter is never bound to a reference type in a function-call context.

```
forall( T ) void f( T & );
int & x;
f( x );  // implicit dereference
```

The call applies an implicit dereference once to x so the call is typed f(**int** &) with T = **int**, rather than with T = **int** &.

As with a pointer type, a reference type may have qualifiers, where **const** is most common.

```
int x = 3;                              // mutable
const int cx = 5;                       // immutable
int * const cp = &x,                    // immutable pointer pointer/reference
    & const cr = cx;
const int * const ccp = &cx,            // immutable value and pointer/reference
          & const ccr = cx;
```

pointer	reference	
*cp = 7;	cr = 7;	// allowed
cp = &x;	cr = &x;	// error, assignment of read–only variable
*ccp = 7;	*ccr = 7;	// error, assignment of read–only location
ccp = &cx;	ccr = &cx;	// error, assignment of read–only variable

[1] C++ uses && for rvalue reference, a feature for move semantics and handling the **const** Hell problem.

[2] C∀ handles the **const** Hell problem by allowing rvalue expressions to be converted to reference values by implicitly creating a temporary variable, with some restrictions. As well, there is a warning that the output nature of the reference is lost. Hence, a single function handles **const** and non-**const** as constness is handled at the call site.

Interestingly, C does not give a warning/error if a **const** pointer is not initialized, while C++ does. Hence, type & **const** is similar to a C++ reference, but CFA does not preclude initialization with a non-variable address. For example, in systems programming, there are cases where an immutable address is initialized to a specific memory location.

```
int & const mem_map = *0xe45bbc67p; // hardware mapped registers ('p' for pointer)
```

Finally, qualification is generalized across all pointer/reference declarations.

```
const * const * const * const ccccp = ...
const & const & const & const cccr = ...
```

In the initial CFA reference design, the goal was to make the reference type a *real* data type *vs.* a restricted C++ reference, which is mostly used for choosing the argument-passing method, *i.e.*, by-value or by-reference. However, there is an inherent ambiguity for auto-dereferencing: every argument expression involving a reference variable can potentially mean passing the reference's value or address. For example, in

```
int & x;
forall( T ) void foo( T );
forall( T ) void bar( T & );
foo( x );                               // means pass by value
bar( x );                               // means pass by reference
```

the call to foo must pass x by value, implying auto-dereference, while the call to bar must pass x by reference, implying no auto-dereference.

My analysis shows without any restrictions, this ambiguity limits the behaviour of reference types in CFA polymorphic functions, where a type T can bind to a reference or non-reference type. This ambiguity prevents the type system treating reference types the same way as other types, even if type variables could be bound to reference types. The reason is that CFA uses a common *object trait* (constructor, destructor and assignment operators) to handle passing dynamic concrete type arguments into polymorphic functions, and the reference types are handled differently in these contexts so they do not satisfy this common interface.

Moreover, there is also some discrepancy in how the reference types are treated in initialization and assignment expressions. For example, in line 3 of the example code on page 19:

```
int & r1 = x,  && r2 = r1,  &&& r3 = r2;   // references to x
```

each initialization expression is implicitly dereferenced to match the types, *e.g.*, &x, because an address is always required and a variable normally returns its value; C++ does the same implicit dereference when initializing its reference variables. For lines 6 and 9 of the previous example code:

```
 r1 =  3; r2 = 3;  r3 = 3;              // change x: implicit dereference *r1, **r2, ***r3
&r3 = &y; &&r3 = &&r4;                  // change r1, r2
```

there are no actual assignment operators defined for reference types that can be overloaded; instead, all reference assignments are handled by semantic actions in the type system. In fact, the reassignment of reference variables is setup internally to use the assignment operators for pointer types. Finally, there is an annoying issue (although purely syntactic) for setting a mutable reference to a specific address like null, **int** & r1 = *0p, which looks like dereferencing a null pointer. Here, the expression is rewritten as **int** & r1 = &(*0p), like the variable dereference of x

above. However, the implicit & needs to be cancelled for an address, which is done with the *, *i.e.*, &* cancel each other, giving 0p. Therefore, the dereferencing operation does not actually happen and the expression is translated into directly initializing the reference variable with the address. Note, the same explicit reference is used in C++ to set a reference variable to null.

```
int & ip = *(int *)nullptr;
```

which is used in certain systems-programming situations.

When generic types were introduced to C∀ [24], some thought was given to allow reference types as type arguments.

```
forall( T ) struct vector { T t; };          // generic type
vector( int & ) vec;                         // vector of references to ints
```

While it is possible to write a reference type as the argument to a generic type, it is disallowed in assertion checking, if the generic type requires the object trait (see page 21) for the type argument, a fairly common use case. Even if the object trait can be made optional, the current compiler implementation often misbehaves by adding undesirable auto-dereference on the referenced-to value rather than the reference variable itself, as intended. Some tweaks are necessary to accommodate reference types in polymorphic contexts and it is unclear what can or cannot be achieved. Currently, there are contexts where the C∀ programmer is forced to use a pointer type, giving up the benefits of auto-dereference operations and better syntax with reference types.

2.2 Tuple Types

The addition of tuples to C∀ can be traced back to the original design by David Till in K-W C [39, 3], a predecessor project of C∀. The primary purpose of tuples is to eliminate output parameters or creating an aggregate type to return multiple values from a function, called a multiple-value-returning (MVR) function. Traditionally, returning multiple values is accomplished via (in/)output parameters or packing the results in a structure. The following examples show these two techniques for a function returning three values.

```
int foo( int &p1, int &p2 );  // in/out parameters
int x, y = 3, z = 4;
x = foo( y, z );  // return 3 values: 1 out, 2 in/out
```

```
struct Ret { int x, y, z; } ret;
Ret foo( int p1, int p2 );  // return structure
ret = foo( 3, 4 );  // return 3 values: 3 out
```

Like Go, K-W C allows direct return of multiple values into a tuple.

```
[int, int, int] foo( int p1, int p2 );
[x, y, z] = foo( 3, 4 );  // return 3 values into a tuple
```

Along with making returning multiple values a first-class feature, tuples were extended to simplify a number of other common contexts that normally require multiple statements and/or additional declarations.

```
[x, y, z] = 3;              // x = 3; y = 3; z = 3, where types may be different
[x, y] = [y, x];            // int tmp = x; x = y; y = tmp;
void bar( int, int, int );
bar( foo( 3, 4 ) );         // int t0, t1, t2; [t0, t1, t2] = foo( 3, 4 ); bar( t0, t1, t2 );
x = foo( 3, 4 ).1;          // int t0, t1, t2; [t0, t1, t2] = foo( 3, 4 ); x = t1;
```

For the call to bar, the three results (tuple value) from foo are *flattened* into individual arguments. Flattening is how tuples interact with parameter and subscript lists, and with other tuples, *e.g.*:

```
[ [ x, y ], z, [a, b, c] ] = [2, [3, 4], foo( 3, 4) ]  // structured
[ x, y, z, a, b, c] = [2, 3, 4, foo( 3, 4) ]      // flattened, where foo results are t0, t1, t2
```

Note, in most cases, a tuple is just compile-time syntactic-sugar for a number of individual assignments statements and possibly temporary variables. Only when returning a tuple from a function is there the notion of a tuple value.

Overloading in the C∀ type-system must support complex composition of tuples and C type conversions using a conversion cost scheme giving lower cost to widening conversions that do not truncate a value.

```
[ int, int ] foo1( int );                    // overloaded foo functions
[ double ] foo2( int );
void bar( int, double, double );
bar( foo( 3 ), foo( 3 ) );
```

The type resolver only has the tuple return types to resolve the call to bar as the foo parameters are identical. The resulution involves unifying the flattened foo return values with bar's parameter list. However, no combination of foos is an exact match with bar's parameters; thus, the resolver applies C conversions to obtain a best match. The resulting minimal cost expression is bar(foo_1(3), foo_2(3)), where the two possible coversions are (**int**, **int**, **double**) to (**int**, **double**, **double**) with a safe (widening) conversion from **int** to **double** versus (**double**, **int**, **int**) to (**int**, **double**, **double**) with one unsafe (narrowing) conversion from **double** to **int** and two safe conversions from **int** to **double**. Go provides a simplified mechanism where only one tuple returning function call is allowed and there are no implicit type conversions.

```
func foo( int ) ( int, int, int ) { return 3, 7, 8 }
func bar( int, int, int ) { ... } // types must match
bar( foo( 3 ) ) // only one tuple returning call
```

Hence, programmers cannot take advantage of the full power of tuples but type match is straightforward.

K-W C also supported tuple variables, but with a strong distinction between tuples and tuple values/variables.

> Note that tuple variables are not themselves tuples. Tuple variables reference contiguous areas of storage, in which tuple values are stored; tuple variables and tuple values are entities which appear during program execution. Tuples, on the other hand, are compile-time constructs; they are lists of expressions, whose values may not be stored contiguously. [39, p. 30]

Fundamentally, a tuple value/variable is just a structure (contiguous areas) with an alternate (tuple) interface. A tuple value/variable is assignable (like structures), its fields can be accessed using position rather than name qualification, and it can interact with regular tuples.

```
[ int, int, int ] t1, t2;
t1 = t2;                                     // tuple assignment
t1.1 = t2.0;                                 // position qualification
```

C++

```
struct S {
    struct T { // not flattened
        int i, j;
    };
    int i, j;
};
```

C/C++

```
struct S2 {
    struct /* unnamed */ { // flatten
        int i, j;
    };
    int i, j;
};
```

tuple

```
[
    [ // flatten
        1, 2
    ]
    1, 2
]
```

Figure 2.1: Nesting

```
int x, y, z;
t1 = [ x, y, z ];                   // interact with regular tuples
[ x, y, z ] = t1;
bar( t2 );                          // bar defined above
```

Figure 2.1 shows the difference is nesting of structures and tuples. The left C++ nested-structure is named so it is not flattened. The middle C/C++ nested-structure is unnamed and flattened, causing an error because i and j are duplication names. The right C∀ nested tuple cannot be named and is flattened. C allows named nested-structures, but they have issues (see Section 2.4, p. 29). Note, it is common in C to have an unnamed **union** so its fields do not require qualification.

I identified the primary issues for tuples in the C∀ type system are polymorphism and conversions. Specifically, does it make sense to have a generic (polymorphic) tuple type, as is possible for a structure?

```
forall( T, S ) [ T, S ] GT; // polymorphic tuple type
GT( int, char ) gt;
GT( int, double ) gt;
gt = [ 3, 'a' ];  // select correct gt
gt = [ 3, 3.5 ];
```

and what is the cost model for C conversions across multiple values?

```
gt = [ 'a', 3L ];  // select correct gt
```

2.3 Tuple Implementation

As noted, traditional languages manipulate multiple values by in/out parameters and/or structures. K-W C adopted the structure for tuple values or variables, and as needed, the fields are extracted by field access operations. As well, for the tuple-assignment implementation, the left-hand tuple expression is expanded into assignments of each component, creating temporary variables to avoid unexpected side effects. For example, the tuple value returned from foo is a structure, and its fields are individually assigned to a left-hand tuple, x, y, z, *or* copied directly into a corresponding tuple variable.

In the second implementation of C∀ tuples by Rodolfo Gabriel Esteves [10], a different strategy is taken to handle MVR functions. The return values are converted to output parameters passed in by pointers. When the return values of a MVR function are directly used in an assignment expression, the addresses of the left-hand operands can be directly passed into the

Till K-W C implementation

```
struct _tuple2 { int _0; int _1; }
struct _tuple2 gives_two() {
    ... struct _tuple2 ret = { r1, r2 };
    return ret;
}
int x, y;
struct _tuple2 _tmp = gives_two();
x = _tmp._0; y = _tmp._1;
```

Esteves C∀ implementation

```
void gives_two( int * r1, int * r2 ) {
    ... *r1 = ...; *r2 = ...;
    return;
}
int x, y;

gives_two( &x, &y );
```

Figure 2.2: Alternate Tuple Implementation

function; composition of MVR functions is handled by creating temporaries for the returns. For example, given a function returning two values:

```
[int, int] gives_two() { int r1, r2; ... return [ r1, r2 ]; }
int x, y;
[x, y] = gives_two();
```

Figure 2.2 shows the two implementation approaches. In the left approach, the return statement is rewritten to pack the return values into a structure, which is returned by value, and the structure fields are individually assigned to the left-hand side of the assignment. In the right approach, the return statement is rewritten as direct assignments into the passed-in argument addresses. The upside of the right implementation is consistence and no copying. The downside is indirection within gives_two to access values, unless values get hoisted into registers for some period of time, which is common.

Interestingly, in the third implementation of C∀ tuples by Robert Schluntz [31, § 3], the MVR functions revert back to structure based, and this remains in the current version of C∀. The reason for the reversion is a uniform approach for tuple values/variables making tuples first-class types in C∀, *i.e.*, allow tuples with corresponding tuple variables. This reversion was possible, because in parallel with Schluntz's work, generic types were added independently by Moss [24], and the tuple variables leveraged the same implementation techniques as for generic variables [31, § 3.7]. For example, these two tuples:

```
[double, double] x;
[int, double, int] y;
```

are transformed internally into two generic structures:

```
forall( T0 &, & T1 | sized( T0 ) | sized( T1 ) )
struct _tuple2_ {
    T0 field_0 ;   T1 field_1 ;
};
forall( T0 &, T1 &, T2 & | sized( T0 ) | sized( T1 ) | sized( T2 ) )
struct _tuple3_ {
    T0 field_0 ;   T1 field_1 ;   T2 field_2 ;
};
```

and the declarations become instances of these generic structure types:

```
_tuple2_( double, double ) x;
```

```
_tuple3_( int, double, int ) y;
```

Now types _tuple2_ and _tuple3_ are available for any further 2 or 3 tuple-types in the translation unit, simplifying internal code transformations by memoizing a small set of tuple structures. Ultimately, these generic types are lowered to specific C structures during code generation. Scala, like C++, provides tuple types through a library using this structural expansion, *e.g.*, Scala provides tuple sizes 1 through 22 via hand-coded generic data-structures.

However, after experience gained building the CFA runtime system, I convinced them making tuple-types first-class seems to add little benefit. The main reason is that tuples usages are largely unstructured,

```
[int, int] foo( int, int );        // unstructured function
typedef [int, int] Pair;           // tuple type
Pair bar( Pair );                  // structured function
int x = 3, y = 4;
[x, y] = foo( x, y );              // unstructured call
Pair ret = [3, 4];                 // tuple variable
ret = bar( ret );                  // structured call
```

Basically, creating the tuple-type Pair is largely equivalent to creating a **struct** type, and creating more types and names defeats the simplicity that tuples are trying to achieve. Furthermore, since operator overloading in CFA is implemented by treating operators as overloadable functions, tuple types are very rarely used in a structured way. When a tuple-type expression appears in a function call (except assignment expressions, which are handled differently by mass- or multiple-assignment expansions), it is always flattened, and the tuple structure of function parameter is not considered a part of the function signatures. For example, these two prototypes for foo:

```
void f( int, int );
void f( [ int, int ] );
f( 3, 4 );  // ambiguous call
```

have the same signature (a function taking two **int**s and returning nothing), and therefore invalid overloads. Note, the ambiguity error occurs at the call rather than at the second declaration of f, because it is possible to have multiple equivalent prototype definitions of a function. Furthermore, ordinary polymorphic type-parameters are not allowed to have tuple types.

```
forall( T ) T foo( T );
int x, y, z;
[x, y, z] = foo( [x, y, z] );  // substitute tuple type for T
```

Without this restriction, the expression resolution algorithm can create too many argument-parameter matching options. For example, with multiple type parameters,

```
forall( T, U ) void f( T, U );
f( [1, 2, 3, 4] );
```

the call to f can be interpreted as T = [1] and U = [2, 3, 4, 5], or T = [1, 2] and U = [3, 4, 5], and so on. The restriction ensures type checking remains tractable and does not take too long to compute. Therefore, tuple types are never present in any fixed-argument function calls, because of the flattening.

Finally, a type-safe variadic argument signature was added by Robert Schluntz [31, § 4.1.2] using **forall** and a new tuple parameter-type, denoted by the keyword ttype in Schluntz's imple-

```
double maxd( int count, ... ) { // ellipse parameter
    double max = 0;
    va_list args;
    va_start( args, count );
    for ( int i = 0; i < count; i += 1 ) {
        double num = va_arg( args, double );
        if ( num > max ) max = num;
    }
    va_end(args);
    return max;
}
printf( "%g\n", maxd( 4, 25.0, 27.3, 26.9, 25.7 ) );
```

Figure 2.3: C Variadic Maximum Function

mentation, but changed to the ellipsis syntax similar to C++'s template parameter pack. For C variadics, *e.g.*, va_list, the number and types of the arguments must be conveyed in some way, *e.g.*, printf uses a format string indicating the number and types of the arguments.

```
int printf( const char * format, ... );  // variadic list of variables to print
```

Figure 2.3 shows an N argument maxd function using the C untyped va_list interface. In the example, the first argument is the number of following arguments, and the following arguments are assumed to be **double**; looping is used to traverse the argument pack from left to right. The va_list interface is walking up the stack (by address) looking at the arguments pushed by the caller. (Compiler-specific ABI knowledge is needed for arguments pushed using registers.)

There are two common patterns for using variadic functions in C∀.

1. Argument forwarding to another function, *e.g.*:

```
forall( T *, TT ... | { void ?{}( T &, TT ); } ) // constructor assertion
T * new( TT tp ) { return ((T *)malloc()){ tp }; }  // call constructor on storage
```

   Note, the assertion on T requires it to have a constructor ?{}. The function new calls malloc to obtain storage and then invokes T's constructor passing the tuple pack by flattening it over the constructor's arguments, *e.g.*:

```
struct S { int i, j; };
void ?{}( S & s, int i, int j ) { s.[ i, j ] = [ i, j ]; }  // constructor
S * sp = new( 3, 4 );  // flatten [3, 4] into call ?{}( 3, 4 );
```

2. Structural recursion for processing the argument-pack values one at a time, *e.g.*:

```
forall( T | { int ?<?( T, T ); } )
T max( T v1, T v2 ) { return v1 < v2 ? v2 : v1; }

forall( T, TT ... | { T max( T, T ); T max( TT ); } )
T max( T arg, TT args ) { return max( arg, max( args ) ); }
```

   The first non-recursive max function is the polymorphic base-case for the recursion, *i.e.*, find the maximum of two identically typed values with a less-than (<) operator. The second recursive max function takes two parameters, T and the TT tuple pack, handling all argument lengths greater than two. The recursive function computes the maximum for the first argument and the maximum value of the rest of the tuple pack. The call of max with one

argument is the recursive call, where the tuple pack is converted into two arguments by taking the first value (lisp car) from the tuple pack as the first argument (flattening) and the remaining pack becomes the second argument (lisp cdr). The recursion stops when the argument pack is empty. For example, max(2, 3, 4) matches with the recursive function, which performs **return** max(2, max([3, 4])) and one more step yields **return** max(2, max(3, 4)), so the tuple pack is empty.

As an aside, polymorphic functions are precise with respect to their parameter types, *e.g.*, max states all argument values must be the same type, which logically makes sense. However, this precision precludes normal C conversions among the base types, *e.g.*, this mix-mode call max(2h, 2l, 3.0f, 3.0ld) fails because the types are not the same. Unfortunately, this failure violates programmer intuition because there are specialized two-argument non-polymorphic versions of max that work, *e.g.*, max(3, 3.5). Allowing C conversions for polymorphic types will require a significant change to the type resolver.

Currently in C∀, variadic polymorphic functions are the only place tuple types are used. My analysis showed many wrapper functions are generated to implement both user-defined generic-types and polymorphism with variadics, because C∀ compiles polymorphic functions versus template expansion. Fortunately, the only permitted operations on polymorphic function parameters are given by the list of assertion (trait) functions. Nevertheless, this small set of functions eventually needs to be called with flattened tuple arguments. Unfortunately, packing the variadic arguments into a rigid **struct** type and generating all the required wrapper functions is significant work and largely wasted because most are never called. Interested readers can refer to pages 77-80 of Robert Schluntz's thesis to see how verbose the translator output is to implement a simple variadic call with 3 arguments. As the number of arguments increases, *e.g.*, a call with 5 arguments, the translator generates concrete **struct** types for a 4-tuple and a 3-tuple along with all the polymorphic type data for them. An alternative approach is to put the variadic arguments into an array, along with an offset array to retrieve each individual argument. This method is similar to how the C va_list object is used (and how C∀ accesses polymorphic fields in a generic type), but the C∀ variadics generate the required type information to guarantee type safety (like the printf format string). For example, given the following heterogeneous, variadic, typed print and usage:

```
forall( T, TT ... | { void print( T ); void print( TT ); } )
void print( T arg , TT rest ) {
    print( arg );
    print( rest );
}
```

```
void print( int i ) { printf( "%d ", i ); }
void print( double d ) { printf( "%g ", d ); }
... // other types
int i = 3 ; double d = 3.5;
print( i, d );
```

it would look like the following using the offset-array implementation approach.

```
void print( T arg, char * _data_rest, size_t * _offset_rest ) {
    print( arg );
    print( *((typeof rest.0)*) _data_rest,  // first element of rest
        _data_rest + _offset_rest[0],       // remainder of data
        _offset_rest + 1);                  // remainder of offset array
}
```

where the fixed-arg polymorphism for T can be handled by the standard **void** *-based C∀ polymorphic calling conventions, and the type information can be deduced at the call site. Note, the

variadic print supports heterogeneous types because the polymorphic T is not returned (unlike variadic max), so there is no cascade of type relationships.

Turning tuples into first-class values in C∀ does have a few benefits, namely allowing pointers to tuples and arrays of tuples to exist. However, it seems unlikely that these types have realistic use cases that cannot be achieved with structures. And having a pointer-to-tuple type potentially forbids the simple offset-array implementation of variadic polymorphism. For example, in the case where a type assertion requests the pointer type TT * in the above example, it forces the tuple type to be a **struct**, and thus incurring a high cost. My conclusion is that tuples should not be structured (first-class), rather they should be unstructured. This agrees with Rodolfo's original description:

> As such, their [tuples] use does not enforce a particular memory layout, and in particular, does not guarantee that the components of a tuple occupy a contiguous region of memory. [10, pp. 74–75]

allowing the simplified implementations for MVR and variadic functions.

Finally, a missing feature for tuples is using an MVR in an initialization context. Currently, this feature is *only* possible when declaring a tuple variable.

```
[int, int] ret = gives_two();          // no constructor call (unstructured)
Pair ret = gives_two();                // constructor call (structured)
```

However, this forces the programer to use a tuple variable and possibly a tuple type to support a constructor, when they actually want separate variables with separate constructors. And as stated previously, type variables (structured tuples) are rare in general C∀ programming so far. To address this issue, while retaining the ability to leverage constructors, I proposed the following new tuple-like declaration syntax.

```
[ int x, int y ] = gives_two();
```

where the semantics is:

```
T t0, t1;
[ t0, t1 ] = gives_two();
T x = t0;  // constructor call
T y = t1;  // constructor call
```

and the implementation performs as much copy elision as possible. Currently, this new declaration form is parsed by C∀, showing its syntax is viable, but it is unimplemented because of downstream resolver issues.

2.4 inline Substructure

As mentioned, C allows an anonymous aggregate type (**struct** or **union**) to be embedded (nested) within another one (see Figure 2.1, p. 24), *e.g.*, a tagged union.

```
struct S {
    unsigned int tag;
    union { // anonymous nested aggregate
```

original

```
struct S {
    struct T {
        int i, j;
    } t; // warning without declaration
    struct T w;
    int k;
};
```

rewritten

```
struct T {
    int i, j;
};
struct S {
    struct T t;
    struct T w;
    int k;
};
```

Figure 2.4: Nested Named Aggregate

```
        int x;  double y;  char z;
    };
} s;
```

Here, the **union** combines its field into a common block of storage, and because there is no variable-name overloading in C, all of the union field names must be unique. Furthermore, because the union is unnamed, these field-names are hoisted into the **struct**, giving direct access, *e.g.*, s.x; hence, the union field names must be unique with the structure field names. The same semantics applies to a nested anonymous **struct**:

original

```
struct S {
    struct { int i, j; };
    struct { int k, l; };
};
```

rewritten

```
struct S {
    int i, j;
    int k, l;
};
```

However, unlike the union which provides storage sharing, there is no semantic difference between the nested anonymous structure and its rewritten counterpart. Hence, the nested anonymous structure provides no useful capability.

Nested *named* aggregates are allowed in C but there is no qualification operator, like the C++ type operator '::', to access an inner type. To compensate for the missing type operator, all named nested aggregates are hoisted to global scope, regardless of the nesting depth, and type usages within the nested type are replaced with global type name. Hoisting nested types can result in name collisions among types at the global level, which defeats the purpose of nesting the type. Figure 2.4 shows the nested type T is hoisted to the global scope and the declaration rewrites within structure S. Hence, the possible accesses are:

```
struct S s;
s.i = 1;
s.t.i = 2;
s.w = (struct T){ 7, 8 };
struct T x = { 5, 6 }; // use (un)nested type name
s.t = (struct T){ 2, 3 };
```

where T is used without qualification even though it is nested in S. It is for these reasons that nested types are not used in C, and if used, are extremely confusing.

C++ chose to change this semantics:

> **Change:** A struct is a scope in C++, not in C.
> **Rationale:** Class scope is crucial to C++, and a struct is a class.
> **Effect on original feature:** Change to semantics of well-defined feature.
> **Difficulty of converting:** Semantic transformation.
> **How widely used:** C programs use **struct** extremely frequently, but the change is only noticeable when **struct**, enumeration, or enumerator names are referred to outside the **struct**. The latter is probably rare.
>
> ISO/IEC 14882:1998 (C++ Programming Language Standard) [16, C.1.2.3.3]

C∀ chose to adopt the C++ non-compatible change for nested types, since C++'s change has already forced certain coding changes in C libraries that must be parsed by C++. C∀ also added the ability to access from a variable through a type to a field.

```
struct S s;  s.i;  s.T.i;
```

See the use case for this feature at the end of this section.

A polymorphic extension to nested aggregates appears in the Plan-9 C dialect, used in the Bell Labs' Plan-9 research operating-system. The feature is called *unnamed substructures* [38, § 3.3], which continues to be supported by gcc and clang using the extension (-fplan9-extensions). The goal is to provided the same effect as a nested aggregate with the aggregate type defined elsewhere, which requires it be named.

```
union U {                               // unnested named
    int x;  double y;  char z;
} u;
struct W {
    int i;  double j;  char k;
} w;
struct S {
    struct W;                           // Plan-9 substructure
    unsigned int tag;
    union U;                            // Plan-9 substructure
} s;
```

Note, the position of the substructure is normally unimportant, unless there is some form of memory or **union** overlay. Like an anonymous nested type, a named Plan-9 nested type has its field names hoisted into **struct** S, so there is direct access, *e.g.*, s.x and s.i. Hence, the field names must be unique, unlike C++ nested types, but the type names are at a nested scope level, unlike type nesting in C. In addition, a pointer to a structure is automatically converted to a pointer to an anonymous field for assignments and function calls, providing containment inheritance with implicit subtyping, *i.e.*, U <: S and W <: S, *e.g.*:

```
void f( union U * u );
void g( struct W * );
union U * up;  struct W * wp;  struct S * sp;
up = &s;                                // assign pointer to U in S
wp = &s;                                // assign pointer to W in S
f( &s );                                // pass pointer to U in S
g( &s );                                // pass pointer to W in S
```

Note, there is no value assignment, such as, w = s, to copy the W field from S.

Unfortunately, the Plan-9 designers did not look ahead to other useful features, specifically nested types. This nested type compiles in C++ and C∀.

```
struct R {
    struct T;                   // forward declaration, conflicts with Plan-9 syntax
    struct S {                  // nested types, mutually recursive reference
        S * sp;  T * tp;  ...
    };
    struct T {
        S * sp;  T * tp;  ...
    };
};
```

Note, the syntax for the forward declaration conflicts with the Plan-9 declaration syntax.

C∀ extends the Plan-9 substructure by allowing polymorphism for values and pointers, where the extended substructure is denoted using **inline**.

```
struct S {
    inline struct W;                    // extended Plan-9 substructure
    unsigned int tag;
    inline U;                           // extended Plan-9 substructure
} s;
```

Note, the declaration of U is not prefixed with **union**. Like C++, C∀ allows optional prefixing of type names with their kind, *e.g.*, **struct**, **union**, and **enum**, unless there is ambiguity with variable names in the same scope. In addition, a semi-non-compatible change is made so that Plan-9 syntax means a forward declaration in a nested type. Since the Plan-9 extension is not part of C and rarely used, this change has minimal impact. Hence, all Plan-9 semantics are denoted by the **inline** qualifier, which clearly indicates the usage of Plan-9 definitions. Finally, the following code shows the value and pointer polymorphism.

```
void f( U, U * );                       // value, pointer
void g( W, W * );                       // value, pointer
U u, * up;   S s, * sp;   W w, * wp;
u = s;   up = sp;                       // value, pointer
w = s;   wp = sp;                       // value, pointer
f( s, &s );                             // value, pointer
g( s, &s );                             // value, pointer
```

In general, non-standard C features (gcc) do not need any special treatment, as they are directly passed through to the C compiler. However, I found the Plan-9 semantics allow implicit conversions from the outer type to the inner type, which means the C∀ type resolver must take this information into account. Therefore, the C∀ resolver must implement the Plan-9 features and insert necessary type conversions into the translated code output. In the current version of C∀, this is the only kind of implicit type conversion other than the standard C arithmetic conversions.

Plan-9 polymorphism can result in duplicate field names. For example, the *diamond pattern* [36, § 6.1][4, § 4] can result in nested fields being embedded twice.

```
struct A { int x; };
struct B { inline A; };
```

```
struct C { inline A; };
struct D {
    inline B;  // B.x
    inline C;  // C.x
} d;
```

Because the **inline** structures are flattened, the expression d.x is ambiguous, as it can refer to the embedded field either from B or C. gcc generates a syntax error about the duplicate member x. The equivalent C++ definition compiles:

```
struct A { int x; };
struct B : public A {};
struct C : public A {};
struct D : public B, C {  // multiple inheritance
} d;
```

and again the expression d.x is ambiguous. While C++ has no direct syntax to disambiguate x, *e.g.*, d.B.x or d.C.x, it is possible with casts, ((B)d).x or ((C)d).x. Like C++, C∀ compiles the Plan-9 version and provides direct qualification and casts to disambiguate x. While ambiguous definitions are allowed, duplicate field names are poor practice and should be avoided if possible. However, when a programmer does not control all code, this problem can occur and a naming workaround must exist.

Chapter 3

Resolution Algorithms

Recapping, C∀'s type-system provides expressive polymorphism: variables can be overloaded, functions can be overloaded by argument and return types, tuple types, generic (polymorphic) functions and types (aggregates) can have multiple type parameters with assertion restrictions; in addition, C's multiple implicit type-conversions must be respected. This generality leads to internal complexity and correspondingly higher compilation cost directly related to type resolution. The reason is that the type resolver must analyze *each* component of an expression with many possible forms of overloading, type restrictions, and conversions. Designing a ruleset that is expressive, behaves as expected, *i.e.*, matches C programmer expectations, and can be efficiently implemented is a very challenging task.

I first worked on the C∀ type-system as a co-op student. At that time, compiling a medium-sized C∀ program using advanced polymorphism took multiple minutes (5+ minutes) [43, § 5]. I found a number of type-resolution problems, which resulted in the following changes to the type-resolution algorithm.

1. new AST data structure
2. special symbol table and argument-dependent lookup
3. late assertion-satisfaction
4. revised function-type representation
5. skip pruning on expressions for function types

Table 3.1 shows improvements for selected tests with accumulated reductions in compile time across each of the 5 fixes. The large reduction in compilation time significantly improves the development of the C∀'s runtime because of its frequent compilation cycles.

Since then, many new features utilizing the expressiveness of C∀'s type system have been implemented, such as generic container types similar to those in C++'s standard template library. During the development of multiple C∀ programs and libraries, more weaknesses and design flaws have been discovered within the type system. Some of those problems arise from the newly introduced language features described in the previous chapter. In addition, fixing unexpected interactions within the type system has presented challenges. This chapter describes in detail the type-resolution rules currently in use and some major problems I have identified. Not all of those problems have immediate solutions, because fixing them may require redesigning parts of the C∀ type system at a larger scale, which correspondingly affects the language design.

Table 3.1: Compile time of selected files by compiler build in seconds

Test case	Original resolver	Accumulative fixes 1	2	3	4	5
lib/fstream	86.4	25.2	10.8	9.5	7.8	7.1
lib/mutex	210.4	77.4	16.7	15.1	12.6	11.7
lib/vector	17.2	8.9	3.1	2.8	2.4	2.2
lib/stdlib	16.6	8.3	3.2	2.9	2.6	2.4
test/io2	300.8	53.6	43.2	27.9	19.1	16.3
test/thread	210.9	73.5	17.0	15.1	12.6	11.8

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

3.1 Expression Cost-Model

CA has been using a total-order expression cost-model to resolve ambiguity of overloaded expressions from the very beginning. Most CA operators can be overloaded in CA; hence, they are treated the same way as other function calls with the same rules for overload resolution.

In CA, candidates of an overloaded expression are ranked by numerical cost elements, which accounts for any type conversions needed from a call-site argument type to the matching function parameter type, as well as polymorphic types and restrictions introduced by the **forall** clause. Currently, the expression cost has the following components, ranked from highest to lowest. where lower cost is better.

1. **Unsafe** cost representing a narrowing conversion of arithmetic types, *e.g.*, **int** to **short**, and qualifier-dropping conversions for pointer and reference types. Narrowing conversions have the potential to lose (truncate) data. A programmer must decide if the computed data-range can safely be shorted in the smaller storage. Warnings for unsafe conversions are helpful.

```
void f( short p );
f( 3L );                        // unsafe conversion, possible warning
```

2. **Polymorphic** cost for a function parameter type that is or contains a polymorphic type variable. The fewer polymorphic parameters means a more specific function, which may be able to compute faster or more accurately, like specialization templates in C++.

```
forall( T ) f( T p, int i );    // 1
forall( T ) f( T p, T i );      // 2
f( 3, 4 );                      // 1
```

3. **Safe** cost representing a widening conversion *e.g.*, **short** to **int**, qualifier-adding conversions for pointer and reference types, and value conversion for enumeration constants. When all conversions are safe, closer conversions are ranked better, *e.g.*, **short** to **int** versus **short** to **long int**.

```
void f( long int p );           // 1
void f( double p );             // 2
f( 3h );                        // 1, short constant
```

4. **Variable** cost that counts the number of polymorphic variables, if any, introduced by the **forall** clause in the function declaration. Fewer polymorphic variables implies more specificity.

```
forall( T, T ) f( T p1, T p2 );      // 1
forall( T, U ) f( T p1, U p2 );      // 2
f( 3, 4 );                           // 1
f( 3, 4.5 );                         // 2
```

5. **Specialization** cost counting the number of restrictions introduced by type assertions. Fewer restriction means fewer parametric variables passed at the function call giving better performance.

```
forall( T | { T ?+?( T, T ) } ) void f( T );          // 1
forall( T | { T ?+?( T, T ), void f( T, T ) } );      // 2
f( 42 );                                              // 1
```

Cost tuples are compared in lexicographical order, from unsafe (highest) to specialization (lowest), with ties moving to the next lowest item. At a subexpression level, the lowest cost candidate for each result type is included as a possible interpretation of the expression; at the top level, all possible interpretations of different types are considered (generating a total ordering) and the overall lowest cost is selected as the final interpretation of the expression. Glen Ditchfield first proposed this costing model [8, § 4.4.5] to generate a resolution behaviour that is reasonable to C programmers based on existing conversions in the C programming language. This model carried over into the first implementation of the C∀ type-system by Richard Bilson [1, § 2.2], and was extended but not redesigned by Aaron Moss [24, chap. 4]. Moss's work began to show problems with the underlying cost model; these design issues are part of this work.

In many languages that support function/method overloading, such as C++ and Java, a partial-order system decides which interpretation of the expression is selected. Hence, sometimes the candidates are incomparable (none are considered a best match), and only when one candidate is considered better than all others (maximal with respect to the partial order) is the expression unambiguous. Specifically, the resolution algorithms used in C++ and Java are greedy, selecting the best match for each subexpression without considering the higher-level ones (bottom-up). Therefore, at each resolution step, the arguments are already given unique interpretations, so the ordering only needs to compare different sets of conversion targets (function parameter types) on the same set of input.

My conclusion is that trying to use such a system in C∀ is problematic because of the presence of return-type overloading of functions and variables. Specifically, C∀ expression resolution considers multiple interpretations of argument subexpressions with different types, *e.g.*: so it is possible that both the selected function and the set of arguments are different, and cannot be compared with a partial-ordering system. This situation arises often in C∀, even in the simple expression f(x), where both the function name f and variable name x are overloaded (examples to follow).

Ada is another programming language that has overloading based on return type. Ada has no type conversions but has subtyping so subtypes are convertible to supertypes. There are only two preference rules in Ada overload resolution:

> There is a preference for the primitive operators (and ranges) of the root numeric types root_integer and root_real. In particular, if two acceptable interpretations of a constituent of a complete context differ only in that one is for a primitive operator (or range) of the type root_integer or root_real, and the other is not,

However, I was unable to generate any Ada example program that demonstrates this preference. In contrast, the C∀ overload resolution-system is at the other end of the spectrum, as it tries to order all legal interpretations of an expression and chooses the best one according to cost, occasionally giving unexpected results rather than an ambiguity.

Interestingly, the C∀ cost-based model can sometimes make expression resolution too permissive because it always attempts to select the lowest cost option, and only when there are multiple options tied at the lowest cost does it report the expression is ambiguous. The reason is that there are so many elements in the cost tuple, the type system "tries too hard to discover a match", and therefore, ties are uncommon. Other than the case of multiple exact matches, where all have cost zero, incomparable candidates under a partial ordering can often have different expression costs since different kinds of implicit conversions are involved, resulting in seemingly arbitrary overload selections.

There are currently at least three different situations where the polymorphic cost element of the cost model does not yield a candidate selection that is justifiable, and one of them is clearly wrong.

1. Polymorphic exact match versus non-polymorphic inexact match.

```
forall( T ) void f( T );          // 1
void f( long );                   // 2
f( 42 );                          // currently selects 2
```

   Under the current cost model, option 1 incurs a polymorphic cost from matching the argument type **int** to type variable T, and option 2 incurs a safe cost from integer promotion of type **int** to **long**. Since polymorphic cost is ranked above safe conversion cost, option 2 is considered to have lower cost and gets selected.

   In contrast, the template inferencing and overload resolution rules in C++ selects option 1 (converting **forall** to the equivalent function template declaration). C++ performs template argument inferencing and overload candidate ranking in two separate steps.

   - In the first step, the type parameters are deduced for each primary function template, and if the corresponding template instantiation succeeds, the resulting function prototype is added to the resolution candidate set.
   - In the second step, the implicit conversions (if any) applied to argument types are compared after taking away top-level qualifiers and references. It then prefers an exact match, followed by basic type promotions (roughly corresponds to safe conversion in C∀), and then other kinds of conversions (roughly corresponds to unsafe conversion in C∀). Only when the type conversions are the same does it prioritize a non-template candidate.

   In this example, option 1 produces the prototype **void** f(**int**), which gives an exact match and therefore takes priority. The C++ resolution rules effectively make option 2 a specialization that only applies to type **long** exactly,[1] while the current C∀ rules make option 2 apply for all integral types ranked lower than **long** as well. This difference could be explained as compensating for C∀ polymorphic functions being separately compiled versus template inlining; hence, calling them requires passing type information and assertions increasing the

[1] C++ does have explicit template specializations, however they do not participate directly in overload resolution and can sometimes lead to unintuitive results.

runtime cost. We are learning that having the type system consider performance may be inappropriate.

2. Having a lower total polymorphic cost does not always mean a function is more specialized. The following example is from Moss's thesis, which discusses some improvements to the C∀ cost-model. He claims the following function prototypes are increasingly more constrained:

```
forall( T, U ) void f( T, U );          // 1 polymorphic
forall( T ) void f( T, T );             // 2 less polymorphic
forall( T ) void f( T, int );           // 3 even less polymorphic
forall( T ) void f( T *, int );         // 4 least polymorphic
```

   This argument is not entirely correct. Although it is true that both the sequence 1, 2 and 1, 3, 4 are increasingly more constrained on the argument types, option 2 is not comparable to either of option 3 or 4; they actually describe independent constraints on the two arguments. Specifically, option 2 says the two arguments must have the same type, while option 3 states the second argument must have type **int**. Because two constraints can independently be satisfied, neither should be considered a better match when trying to resolve a call to f with argument types (**int**, **int**); reporting such an expression as ambiguous is more appropriate. The example illustrates the limitation of using a numerical cost value as it cannot represent these complicated cases.

3. A generic type can require more type variables to describe a more specific type. For example, consider a generic function taking a pair-type requires two type variables.

```
forall( T, U ) void f( pair( T, U ) );  // 1
```

   Add a function taking any type.

```
forall( T ) void f( T );                // 2
```

   Passing a pair variable to f

```
pair(int, double) p;
f( p );
```

   gives a cost of 1 poly, 2 variable for the pair overload, versus a cost of 1 poly, 1 variable for the unconstrained overload. Programmer expectation is to select option 1 because of the exact match, but the cost model selects 2; it is not possible to write a specialization for f that works on any pair type and gets selected by the type resolver as intended. As a result, simply counting the number of polymorphic type variables is no longer correct to order the function candidates as being more constrained.

These inconsistencies are not easily solvable in the current cost-model, meaning that currently the C∀ codebase has to workaround these defects. One potential solution is to mix the conversion cost and C++-like partial ordering of specializations. For example, observe that the first three elements (unsafe, polymorphic and safe conversions) in the C∀ cost-tuple are related to the argument/parameter types, while the other two elements (polymorphic variable and assertion counts) are properties of the function declaration. Using this observation, it is reasonable to have an ordering that compares the argument/parameter conversion costs first and uses the partial ordering of specializations as a tiebreaker. Hence, the C++ template-specialization ordering can be applied to C∀ with a slight modification.

In the meantime, some other improvements have been made to the expression cost system. Most notably, the expression resolution algorithm now consistently uses the globally minimal

cost interpretation, as discussed in a previous technical report. While implementing the change, there are also two detailed issues that need to be addressed for the new rules to fully work.

The first deals with a common idiom in C∀ that creates many overloads with equal cost. Many C library functions have multiple versions for different argument types. For example, the absolute-value function is defined for basic arithmetic types with different names, since C does not support overloading.

```
int abs( int );                          double fabs( double );                  cabs, cabsf, and cabsl
long labs( long );                       float fabsf( float );                   for _Complex
long long int llabs( long long int );    long double fabsl( long double );
```

It is cumbersome for programmers to remember these function names and select the correct one. If an incorrect version is picked, the program compiles but with potential negative consequences such as using an integral version with a floating-point argument. In C∀, these functions are wrapped by functions with the overloaded name abs, which results in multiple overloads with the same total cost when some conversion is needed. For example, **long** x = abs(42) resolves to **long** abs(**long**) with **int** argument 42 converted to **long** or **int** abs(**int**) converting the result to **long**. In this example, the choice could be arbitrary because both yield identical results. However, for **int** i = abs(-9223372036854775807LL), the result is -1 due to the narrowing conversion from **long** to **int** on the assignment, suggesting at warning to the programmer to reconsider the type of i or the compiler generates an error as the argument is too large for an **int** parameter. The C∀ system library has multiple overload groupings of C functions into a single name, so usage should not report an ambiguity or warning unless there is a significant chance of error.

While testing the effects of the updated cost rule, the following example was found in the C∀ standard library.

```
static inline double __to_readyQ_avg( unsigned long long intsc ) {
    if ( unlikely( 0 == intsc ) ) return 0.0;
    else return log2( intsc ); // implicit conversion happens here
}
```

This helper function is used for performance logging as part of computing a geometric mean; it is called during summing of logarithmic values. However, the function name log2 is overloaded in C∀ for both integer and floating point types. In this case, the integer overload returns an integral result, truncating any small fractional part of the logarithm, so the sum is slightly incorrect. When experimenting with the updated cost rules, it flagged the log2 call as an ambiguous expression. When asked, the developer expected the floating-point overload because of return-type overloading. This mistake went unnoticed because the truncated component was insignificant in the performance logging. To correct this mistake, I changed the resolution algorithm to favour a lower conversion cost up the expression tree when the total global cost is equal.

Another change addresses the issue that C arithmetic expressions have unique meanings governed by its arithmetic conversion rules. From the C language reference manual:

1. First, if the corresponding real type of either operand is **long double**, the other operand is converted, without change of type domain, to a type whose corresponding real type is **long double**.

2. Otherwise, if the corresponding real type of either operand is **double**, the other operand is converted, without change of type domain, to a type whose corresponding real type is **double**.
3. Otherwise, if the corresponding real type of either operand is **float**, the other operand is converted, without change of type domain, to a type whose corresponding real type is **float**.[2]
4. Otherwise, the integer promotions are performed on both operands. Then the following rules are applied to the promoted operands:
   a. If both operands have the same type, then no further conversion is needed.
   b. Otherwise, if both operands have signed integer types or both have unsigned integer types, the operand with the type of lesser integer conversion rank is converted to the type of the operand with greater rank.
   c. Otherwise, if the operand that has unsigned integer type has rank greater or equal to the rank of the type of the other operand, then the operand with signed integer type is converted to the type of the operand with unsigned integer type.
   d. Otherwise, if the type of the operand with signed integer type can represent all of the values of the type of the operand with unsigned integer type, then the operand with unsigned integer type is converted to the type of the operand with signed integer type.
   e. Otherwise, both operands are converted to the unsigned integer type corresponding to the type of the operand with signed integer type.

[17, § 6.3.1.8]

Figure 3.1 shows the C arithmetic conversions graphically. Rule c. says an unsigned type is safely convertible to an signed type with greater rank, while rule d. says a signed type is unsafely convertible to an unsigned type. Therefore, these two rules cover every possible case. Rule e. is the odd-ball rule because it is really a demotion because signed to unsigned (marked in red) is unsafe. Finally, assignment allows demotion of any larger type into a smaller type, with a possibly lossy conversion (and often no warning). These promotion conversions are applied greedily at local points within an expression. Because there is no overloading in C, except for builtin operators, no cost model is needed to differentiate among alternatives that could result in ambiguous matches.

Figure 3.2 shows the current C∀ total-order arithmetic-conversions graphically. Here, the unsafe cost of signed to unsigned is factored into the ranking, so the safe conversion is selected over an unsafe one. Furthermore, an integral option is taken before considering a floating option. This model locally matches the C approach, but provides an ordering when there are many overload alternatives. However, as Moss pointed out overload resolution by total cost has problems, *e.g.*, handling cast expressions.

> . . . 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:

[2]For example, addition of a **double _Complex** and a **float** entails just the conversion of the **float** operand to **double** (and yields a **double _Complex** result).

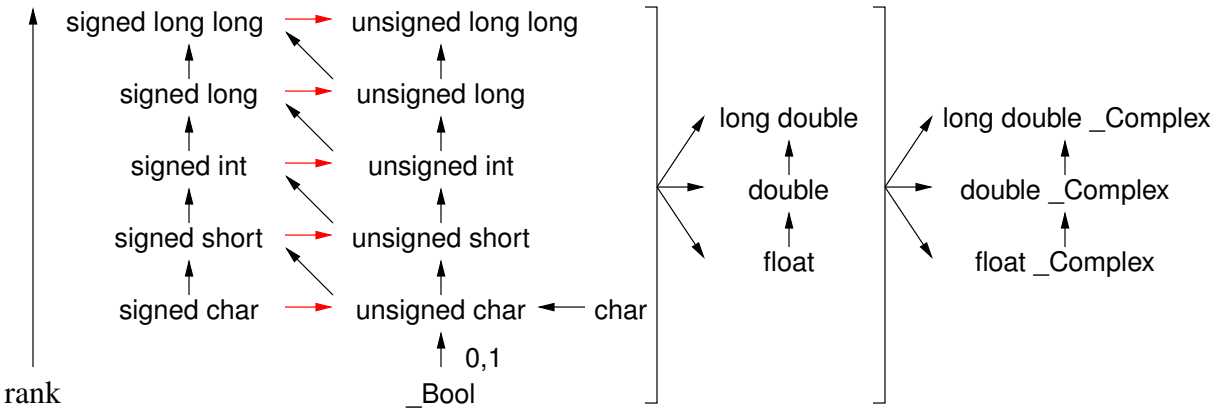


Figure 3.1: C Expression Conversions: T1 operator T2

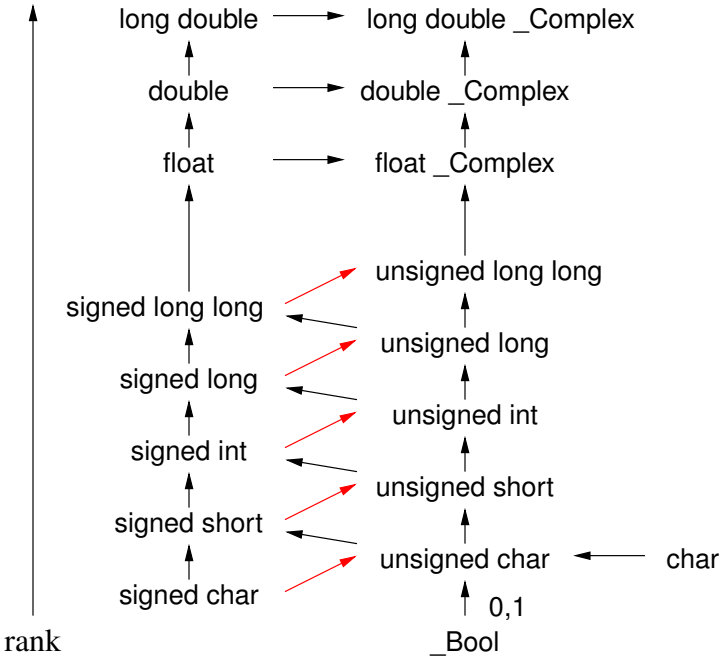


Figure 3.2: C∀ Total-Ordering Expression Conversions

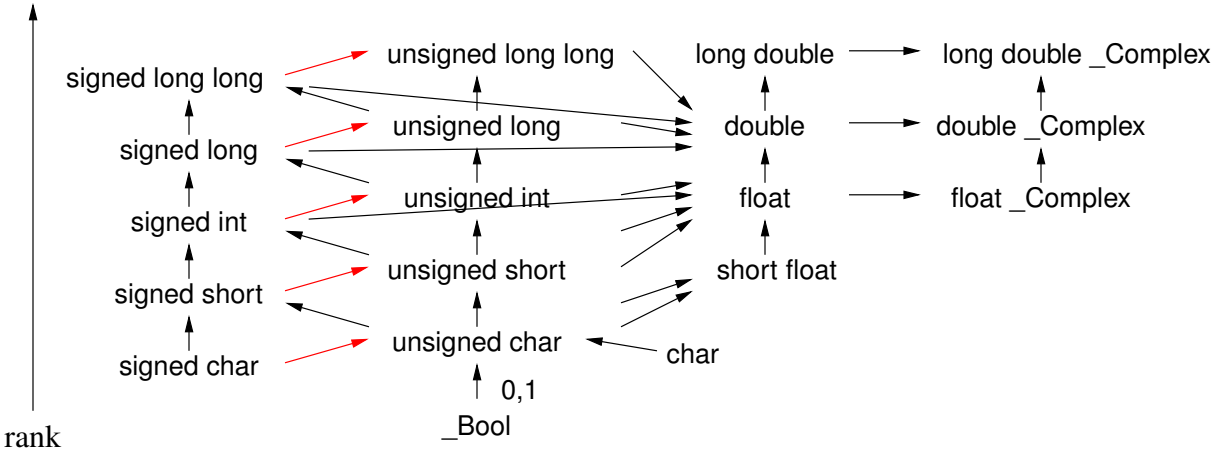


Figure 3.3: C∀ Partial-Ordering Expression Conversions

```
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). [Ed., both approached have an unsafe cast from **long long** to **unsigned** (truncation), the difference is when the truncation occurs, *i.e.*, before or after the shift, potentially generating different results.] 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. [24, pp. 46-47]

However, a cast expression is unnecessary to have such inconsistency to C semantics. An implicit argument-parameter type conversion in a function calls can replicate this issue without an explicit cast.

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

The argument generation has the same effect as using an explicit cast to coerce the type of expression x >> 32 to **unsigned**. This example shows the problem is not coming from the cast expressions, but from modelling the C builtin operators as overloaded functions. As a result, a different rule is used to select the builtin function candidates to fix this problem: if an expression has any legal interpretations as a C builtin operation, only the lowest cost one is kept, regardless of the result type.

Figure 3.3 shows my alternative C∀ partial-order arithmetic-conversions graphically. The idea here is to first look for the best integral alternative because integral calculations are exact and cheap. If no integral solution is found, than there are different rules to select among floating-point alternatives. This approach reduces the search space by partitioning into two categories.

3.2 Type Unification

Type unification is the algorithm that assigns values to each (free) type parameter such that the types of the provided arguments and function parameters match.

C∀ does not attempt to do any type *inference* (see Section 1.3, p. 8): it has no anonymous functions (*i.e.*, lambdas, commonly found in functional programming and also used in C++ and Java), and the variable types must all be explicitly defined (no auto typing). This restriction makes the unification problem more tractable in C∀, as the argument types at each call site are usually all specified. There is a single exception case when the function return type contains a free type variable that does not occur in any of the argument types, and subsequently passed into the parent expression. One such example is the C∀ wrapper for malloc which also infers size argument from the inferred return type.

```
forall( T * ) T * malloc() {
    return ( T *)malloc( sizeof(T) ); // calls C malloc with the size inferred from context
}
int * i = malloc();   // infer int for T from LHS
```

A top level expression whose type still contains an unbounded type variable is considered ill-formed as such an expression is inherently ambiguous.

The unification algorithm in C∀ is originally presented in Richard Bilson's thesis and it has remained as the basis of the algorithm currently in use. Some additions have been made in order to accommodate for the newly added type features to the language. To summarize, the C∀ type unification has two minor variants: an *exact* mode and an *inexact* mode. The exact mode is required since the type matching algorithm operates recursively and the inner types often have to match exactly, *e.g.*, there is no common type for the pointer types **int** * and **long** * while there is for **int** and **long**. The inexact mode is applied at top level argument-parameter matching, and attempts to find an assignment to the type variables such that the argument types can be converted to parameter types with minimal cost as defined in the previous section. With the introduction of generic record types, the parameters must match exactly as well; currently there are no covariance or contravariance supported for the generics.

I made one simplification to the C∀ language that makes modelling the type system easier: polymorphic function pointer types are no longer allowed. The polymorphic function declarations themselves are still treated as function pointer types internally, however the change means that formal parameter types can no longer be polymorphic. Previously it was possible to write function prototypes such as

```
void f( forall( T | { T -?( T ); } ) T (*p)( T, T ) );
```

Notably, the unification algorithm implemented in the C∀ compiler has never managed to trace the assertion parameters on the formal types at all, and the problem of determining if two assertion sets are compatible is likely undecidable in general, given the ability of synthesizing more complicated types by the nesting of generics. Essentially, the reason for disallowing this construct is that it does not provide a useful type feature. A function operates on the call-site arguments together with any local and global variables. When the function is polymorphic, the types are inferred at each call site. On each invocation, the types to be operated on are determined from the arguments provided, and therefore, there is no need to pass a polymorphic function pointer, which can take any type in principle. For example, consider a polymorphic function that takes one argument of type T and polymorphic function pointer.

```
forall( T ) void f( T x, forall( U ) void (* g)( U ) );
```

Making g polymorphic in this context is almost certainly unnecessary, since it can only be called inside the body of f and the types of the argument must be known, although it can potentially depend on T. Moreover, requesting a function parameter to be able to work on any input type would impose too much constraint on the arguments, as it only needs to make each call inside the body of f valid.

Hence, rewriting the prototype (or potentially, some compound type synthesized from T):

```
forall( T ) void f( T x, void (* g)( T ) );
```

is sufficient, so g is no longer a polymorphic type itself. This *monomorphization* conversion is

readily supported in C∀, either by explicitly assigning a polymorphic function name to a compatible function pointer type, or implicitly done in inferring assertion parameters (which is discussed next). This technique is applicable to argument passing, which is just an assignment to a function parameter variable. In theory, there might be edge cases where the supplied function g is called on arguments of different types inside the body of f and so needs to be polymorphic, but this case is rare and its benefit seems to be minimal in practice.

The result of this change is that the unification algorithm no longer needs to distinguish open (unbounded) and closed (bounded) type-variables, as the latter is not allowed to exist. The only type variables that need to be handled are those introduced by the **forall** clause from the function prototype. The subtype relationship among function types is now also rendered redundant since none of the function parameter or return types can be polymorphic, and no basic types or non-polymorphic function types are subtypes of any other type. Therefore, the goal of (exact) type unification becomes finding a substitution that produces identical types. The assertion set that needs to be resolved is just the declarations on the function prototype, which also simplifies the assertion satisfaction algorithm, which is discussed further in the next section.

My implementation sketch stores type unification results in a type-environment data-structure, which represents all the type variables currently in scope as equivalent classes, together with their bound types and information such as whether the bound type is allowed to be opaque (*i.e.*, a forward declaration without definition in scope) and whether the bounds are allowed to be widened. In the general approach commonly used in functional languages, the unification variables are given a lower bound and an upper bound to account for covariance and contravariance of types. C∀ does not implement any variance with its generic types and does not allow polymorphic function types, therefore no explicit upper bound is needed and one binding value for each equivalence class suffices. However, to handle type conversions, the type environment needs to keep track of which type variables can be converted. This behaviour is notably different from C++ template argument inferencing that enforces an exact match everywhere unless the template argument types are explicitly given. For example, a polymorphic maximum function in C∀ can be called with arguments of different arithmetic types and the result follows the usual arithmetic conversion rules.

```
forall( T | {int ?<? ( T, T ); } ) T max ( T, T );
max( 42, 3.14 );                    // C∀ implicitly infers T == double
max<double>(42, 3.14);              // C++ requires explicit type specification
```

From a theoretical standpoint, the simplified implementation of the type environment has its shortcomings. Some cases do not work nicely with this implementation, and hence, some compromise has to be made. A more detailed discussion follows in Section 3.4, p. 48.

3.3 Assertion Satisfaction

The assertion-satisfaction problem greatly increases the complexity of C∀ expression resolution. Past experiments have shown that the majority of compilation time is spent in resolving the assertions for those expressions that takes the longest time to resolve. Even though a few heuristics-based optimizations have been introduced to the compiler, this remains the most costly part of a C∀ compilation. The major difficulty of resolving assertions is that the problem can become

recursive, since the expression used to satisfy an outstanding assertion can have its own assertions, and in theory this can go on indefinitely. Detecting infinite recursion cases in general is not algorithmically possible and it is not attempted in the compiler. Instead, a fixed maximum depth of recursive assertions is imposed. This approach is also taken by C++ compilers as template argument inferencing is also similarly undecidable in general.

In previous versions of C∀, this number was set at 4; as the compiler has become more optimized and capable of handling more complex expressions in a reasonable amount of time, I have increased the limit to 8 and it has not led to problems. Only rarely is there a case where the infinite recursion produces an exponentially growing assertion set, causing minutes of time wasted before the limit is reached. Fortunately, it is very hard to generate this situation with realistic C∀ code, and the ones that have occurred have clear characteristics, which can be prevented by alternative approaches. In fact, some of my performance optimizations come from analyzing these problematic cases. One example is analysed in this section.

My analysis shows that while the assertion satisfaction problem in isolation looks like just another expression to resolve, its recursive nature makes some techniques for expression resolution no longer possible. The most significant impact is that type unification has a side effect, namely editing the type environment (equivalence classes and bindings), which means if one expression has multiple associated assertions it is dependent, as the changes to the type environment must be compatible for all the assertions to be resolved. Particularly, if one assertion parameter can be resolved in multiple different ways, all of the results need to be checked to make sure the change to type variable bindings are compatible with other assertions to be resolved. A naive algorithm that simply picks any pending assertion to resolve and continue in a depth-first search could be very inefficient and especially prone to an infinite loop. In many cases, these problems can be avoided by examining other assertions that provide insight on the desired type binding: if one assertion parameter can only be matched by a unique option, the type bindings can be updated confidently without the need for backtracking.

The Moss algorithm currently used in C∀ was developed using a simplified type system that captures most of C∀'s type system features. The simulation results were then ported back to the actual language. The simulator used a mix of breadth- and depth-first search in a staged approach. To resolve a set of assertions, the algorithm first attempts to resolve each assertion item individually. There are three possible outcomes on resolving each assertion:

1. If no matches are found, the algorithm terminates with a failure (ambiguity).
2. If exactly one match is found, the type environment is updated immediately with this result, affecting the resolution of remaining assertions.
3. If multiple matches are found, the assertion candidates with their updated type environments are stored in a list that is checked for compatibility at the end.

When all assertion items are resolved successfully, the algorithm attempts to combine the ambiguously resolved assertions to produce mutually compatible assignments. If any new assertions are introduced by the selected candidates, the algorithm is applied recursively, until there are none pending resolution or the recursion limit is reached, which results in a failure.

However, I identify that in practice the efficiency of this algorithm can be sensitive to the order of resolving assertions. Suppose an unbound type variable T appears in two assertions:

```
forall( T | { void foo( T ), void bar( T ) } ) T f( T );
void foo( int );
void bar( int );   void bar( double );   ...
f( 3 );
```

where one can be uniquely resolved and allow the type T to be inferred immediately, and another has many ways to be resolved, each resulting in T being bound to a different concrete type. If the first assertion is examined by the algorithm, the inferred type can then be utilized in resolving the second assertion eliminating many incorrect options without producing a list of candidates requiring further checks. In practice, this have a significant impact when an unbound type T is declared to satisfy the basic *object assertions*[3] of having a default constructor, destructor, and copy assignment operations. Since these functions are implicitly defined for almost every type in scope, there can be hundreds or even thousands of matches to these functions with an unspecified operand type.

Based on this observation, I implemented an optimization to the assertion resolution algorithm that only attempts to resolve object assertions after all other assertions are resolved successfully. As well, it further delays resolving object assertions with an unbound first-argument type, *i.e.*, the type of the argument being constructed or destructed, until no progress can be made otherwise. This simple optimization on assertion-resolution order eliminates over 80 percent of unbound object-lifetime function lookups. In most cases, the operand type can be inferred by resolving other assertions first, and then the object lifetime functions can be looked up efficiently, since these functions are indexed by the operand type in the identifier table of the compiler. Although the unbound parameter case appears infrequently in practice, it is potentially very costly due to thousands of wasted type unification runs each time it occurs. As a result, this optimization is able to produce an overall compilation speedup of around 10 percent.

The issue of having unbound parameters also limits the capability of the assertion resolution algorithm. Assertion matching is implemented to be more restrictive than expression resolution in general, in that the parameter types must match exactly, rather than just merely callable. If a function declaration includes the assertion **void** f(T) and only a f(**long**) is in scope, trying to resolve the assertion with T == **int** does not work. However, loosening the assertion matching requirement causes many assertion variables to have multiple matches and makes the delayed combination step too costly.

Given all the issues caused by assertion resolution creating new type variable bindings, a natural progression is to restrict free type-variables such that all the type variables are bound when the expression reaches the assertion resolution stage. A type variable introduced by the **forall** clause of function declaration can appear in parameter types, return types and assertion variables. If it appears in parameter types, it can be bound when matching the arguments to parameters at the call site. If it only appears in the return type, it can be eventually determined from the call-site context. Currently, type resolution cannot do enough return-type inferencing while performing eager assertion resolution: the return type information is unknown before the parent expression is resolved, unless the expression is an initialization context where the variable type is known. By delaying the assertion resolution until the return type becomes known, this problem can be circumvented. The truly problematic case occurs if a type variable does not

[3]The term is borrowed from object-oriented languages although C∀ is not object-oriented.

appear in either of the parameter or return types and only appears in assertions or variables (*associated types*).

```
forall( T | { void foo( T ) } ) int f( float ) {
    T t;
}
```

This case is rare, so forcing every type variable to appear at least once in parameter or return types does not limit the expressiveness of C∀ type system to a significant extent. Section 5.2, p. 56 presents a proposal for including type declarations in traits rather than having all type variables appear in the trait parameter list, which provides equivalent functionality to an unbound type parameter in assertion variables, serves as a guidance to the resolution algorithm that works in more general cases than the specific optimization for object assertions mentioned above, and also addresses some of the variable cost issue discussed in Section 3.1, p. 35.

3.3.1 Caching Assertion Resolution Results

In Aaron Moss's prototype design and experiments, a potential optimization of caching the result of already resolved assertions is discussed. Based on the experiment results, this approach can improve the performance of expression resolution in general, and sometimes allow difficult instances of assertion resolution problems to be solved that are otherwise infeasible, *e.g.*, when the resolution encounters an infinite loop.

I identify that the tricky problem in implementing this approach is that the resolution algorithm has side effects, namely modifying the type bindings in the environment. If the modifications are cached, *i.e.*, the results that cause the type bindings to be modified, it is also necessary to store the changes to type bindings, too. Furthermore, in cases where multiple candidates can be used to satisfy one assertion parameter, all of them must be cached including those that are not eventually selected, since the side effect can produce different results depending on the context.

The original design of C∀ includes unrestricted polymorphic formal parameters with assertions on themselves, making the problem more complicated as new declarations can be introduced in scope during expression resolution. Here is an example taken from Bilson:

```
void f( forall( T | { T -?( T ); } ) T (*p)( T, T ) );
forall( U, V | { U -?( U ); V -?( V ); } ) U g( U, V ) );
f( g );
```

The inner assertion parameter on the *closed* type-variable T is used to satisfy the assertions on U and V in this example.

However, as per the previous discussions on this topic, polymorphic function-pointer types have been removed from C∀, since correctly implementing assertion matching is impossible in general. Without closed parameters (and therefore no have-set for assertions) the set of declarations in scope remains unchanged, while resolving an expression. The current C∀ implementation also does not attempt to widen any bound type-parameters to satisfy an assertion. Note, this restriction does mean certain kinds of expressions cannot be resolved, *e.g.*:

```
forall( T | { void f( T ); } ) void g( T );
void f( long );
g( 42 );
```

The call g(42) is rejected since no attempt is made to widen the parameter type T from **int** to **long**. Such problem could be mitigated by allowing inexact matches of assertions, but cannot be eliminated completely, if T is matched in a parameterized type, including pointers and references:

```
forall( T | {void f( T*);})  void g( T );
void f( long * );
g( 42 );
```

Here the only way to resolve the call g(42) is to allow assertion resolution to widen the parameter T, but even with inexact matching, **int** * cannot be converted to **long** *.

3.4 Compiler Implementation Considerations

C∀ is still an experimental language and there is no formal specification of expression resolution rules yet. Currently, there is only one reference implementation, namely the cfa–cc translator, which is under active development and the specific behaviour of the implementation can change frequently as new features are added. (This situation is standard for any new programming language.)

Ideally, the goal of expression resolution involving polymorphic functions is to find the set of type variable assignments such that the global conversion cost is minimal and all assertion variables can be satisfied. Unfortunately, there are a lot of complications involving implicit conversions and assertion variables; hence, fully achieving this goal is unrealistic. And as noted, the C∀ compiler is not covering all the edge cases for its current type-system design. Instead it makes a few restrictions to simplify the resolution algorithm so most expressions in actual code still pass type checking within a reasonable amount of time.

As previously mentioned, C∀ polymorphic type resolution is based on a modified version of a unification algorithm, where both equivalence (exact) and subtyping (inexact) relations are considered. However, the implementation of the type environment is simplified; it only stores a tentative type binding with a flag indicating whether *widening* is possible for an equivalence class of type variables. Formally speaking, I concluded the type environment used in C∀ is only capable of representing *lower-bound* constraints. This simplification works most of the time, given the following properties of the existing C∀ type system and the resolution algorithms:

1. Type resolution almost exclusively proceeds in bottom-up order, which naturally produces lower bound constraints. Since all identifiers can be overloaded in C∀, little information can be gained from top-down analysis. In principle, it is possible to detect non-overloaded function names and perform top-down resolution for those; however, Moss' prototype experiments showed this optimization does not give a meaningful performance benefit, and therefore was not implemented.
2. Few nontrivial subtyping relationships are present in C∀, *e.g.*, the arithmetic types presented in Figure 3.2, p. 41 and qualified pointer/reference types. The reason is that C∀ lacks nominal inheritance subtyping present in object-oriented languages, and the generic types do not support covariance on type parameters. As a result, named types such as structures are always matched by strict equivalence, even when type parameters exist.

3. Unlike functional programming where subtyping between arrow types exists, *i.e.*, if $T_2 <: T_1$ and $U_1 <: U_2$ then $T_1 \to T_2 <: U_1 \to U_2$, C∀ uses C function pointer types and the parameter/return types must match exactly to be compatible.

C∀ does attempt to incorporate upstream type information propagated from a variable declaration with initializer, since the type of the variable being initialized is known. However, the current type-environment representation is flawed in handling such type inferencing, when the return type in the initializer is polymorphic. Currently, an inefficient workaround is performed to create the necessary effect. The following is an annotated example of the workaround.

```
// If resolution is unsuccessful with a target type, try again without, since it sometimes succeeds
// when it does not with a target type binding. For example:
forall( T ) T & ?[]( T *, ptrdiff_t );
const char * x = "hello world";
int ch = x[0];
// * T is bound to int
// * (x: const char *) is unified with int *, which fails
```

The problem is that the type system can only represent the constraints $T = int$ and $int <: T$, but since the type information flows in the opposite direction, the proper constraint for this case is $T <: int$, which cannot be represented in the simplified type environment. Currently, an attempt to resolve with equality constraint generated from the initialized variable is still made, since it is often the correct type binding (especially in the case where the initialized variable is a structure), and when such attempt fails, the resolution algorithm is rerun without the initialization context.

One additional remark to make here is that C∀ does not provide a mechanism to explicitly specify values for polymorphic type parameters. In C++ for example, users may specify template arguments in angle brackets, which is necessary when automatic inferencing fails, *e.g.*, max<**double**>(42, 3.14). There are some partial workarounds such as adding casts to the arguments, but they are not guaranteed to work in all cases. If a type parameter appears in the function return type, however, using an ascription (return) cast

```
(return double (*)( double, double ))max( 42, 3.14 );  // tell resolver what type to use
```

forces the desired type binding, since the cast only compiles when the expression type matches exactly with the target.

Chapter 4

Related Work

4.1 Polymorphism

ML [22] was the first language to support parametric polymorphism. Like C∀, it supports unconstrained, universal, type parameters. C∀ differs by adding assertions and traits to constrain type arguments. Haskell [15] combines ML-style polymorphism, polymorphic data types, and type inference with the notion of type classes, collections of overloadable methods. The class/type-class association constrain type arguments by indicating the set of functions that become implicit assertion arguments and specify the implementation of these operations. As pointed out (see Figure 1.1, p. 17), Haskell requires an explicit association between types and constraints (type-class). Otherwise, Haskell does not provide general overloading. C∀ differs by allowing general overloading and constraining type arguments with traits. Most importantly, C∀ automates selection of the assertion arguments at the function call-sites based on the set of operations in scope at that point to generalize reuse.

C++ provides three disjoint polymorphic extensions to C: overloading, inheritance, and templates. Selecting among them and understanding how they interact is part of the challenge in C++ program development. General overloading is available, subtyping inheritance can be single or multiple, templates are typed macro-expansion over a universal type, which precludes separate compilation. Universal template types are constrained using concepts [18], but only as a guideline, as the template expansion can still discover additional constraints. Template expansion can result in code bloat and poor error messages. Type inferencing is available using **auto**, precluding using the return type for overload selection. C∀ differs by providing a simplified, uniform facility for polymorphic code, eliminating subtyping polymorphism, and encompassing overloading among parametric functions and universal (generic) types, all of which are separately compilable. Overload resolution uses the return type and arithmetic conversions to make precise function selections versus generating ambiguities, at the cost of type inferencing. Both C∀ call-site inferencing and C++ template expansion search in the local environment to satisfy explicit assertions or to find named functions to complete the template, respectively.

Cyclone [14] also provides capabilities for polymorphic functions and existential types, similar to C∀'s **forall** functions and generic types. Cyclone's 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 potentially error prone. 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 C∀ terms, all Cyclone polymorphism must be an incomplete data-type **forall**(T *) (see Section 1.4.1, p. 11), which provides the efficiency benefits of fixed-size types (see page 14). C∀ differs by adding object and variadic kinds of polymorphism, which provide more expressive reuse forms. Smith and Volpano [34] present Polymorphic C, an ML dialect with polymorphic functions, C-like syntax, and pointer types. While these language purport to be C replacements, they are significantly different from C, and hence, not a practical C replacement. C∀ differs in providing better imperative-style polymorphism, while retaining backwards syntax and semantic compatibility with C.

Objective-C [26] and its successor Swift [20] are used in iOS phone applications. Swift has polymorphic functions and generics, object subtyping, traits via extensions, general function/method overloading, including parameter names and return type. Objective-C communication is via message passing rather than function call, while swift uses function call unless interacting with Objective-C. Swift and C∀'s type-systems are very similar, minus the object-oriented subtyping in C∀, which is felt to be unnecessary. The GObject [11] framework adds object-oriented programming to C with runtime type-checking and reference-counting garbage collection, as a modernization path for existing C code-bases. Vala [40] compiles to GObject-based C, but requires additional language syntax over GObject. C∀'s path for modernizing works with the existing C type system and runtime, *i.e.*, not adding object-oriented types or garbage collection. Java [12] has object-oriented subtyping, generic interfaces that act like traits, which are type checked at compilation and type erased at runtime similar to C∀'s, and general overloading on methods. However, in Java, each class carries its own table of method pointers, and use of interfaces must be pre-declared with the implements keyword, whereas C∀ passes trait pointers at call-site maintaining a C-compatible layout. Java is also garbage-collected.

D [2], Go, and Rust [29] are compiled languages with generic functions and types using traits similar to C∀: *interfaces* in D and Go, and *traits* in Rust. D and Go are garbage-collected languages; Rust is not garbage collected. Go's generic types and functions are limited to a small fixed-set provided by the compiler, with no language facility to define more. Rust also possesses more powerful abstraction capabilities for writing generic code than Go. While Rust's borrow checker provides strong safety guarantees, it is complex and difficult to learn, and imposes a distinctly idiomatic programming style different from C. C∀, with its modest safety features, has a comparable type-system to Rust's that also maintains C backwards compatibility, which provides a modernization path for existing C code-bases.

4.2 Tuples/Variadics

Many programming languages have some form of tuple construct and/or variadic functions, *e.g.*, SETL, C, KW-C, C++, D, Go, Java, ML, and Scala. SETL [32] is a high-level mathematical programming language, with tuples being one of the primary data types. Tuples in SETL allow subscripting, dynamic expansion, and multiple assignment. KW-C [3], a predecessor of C∀, introduced tuples to C as an extension of the C syntax, taking much of its inspiration from SETL. This work added multiple return value functions (MRVF), tuple mass and multiple assignment, and record-member access, giving unstructured tuples. Structured tuples (tuple variables) are also introduced including multiple coercions between structured and unstructured tuples.

Like C++, D provides tuples through a library variadic-template structure. Go does not have tuples but supports MRVF. Tuples are a fundamental abstraction in most functional programming languages, such as Standard ML [23], Haskell, and Scala [30], which decompose tuples using pattern matching. From KW-C unstructured tuples, C∀ took MRVF, mass and multiple assignment, and record-member access. While C∀ has some structured-tuple capabilities, Section 2.3, p. 24, my analysis suggests this feature might be removed.

An alternative to tuple types is variadic (variable number of arguments) functions or types. C provides variadic functions through va_list objects, but the programmer is responsible for managing the number of arguments and their types; thus, the mechanism is type unsafe. C++11 introduced std::tuple as a library variadic-template structure. Tuples are a generalization of std::pair allowing for arbitrary length, fixed-size aggregation of heterogeneous values. Operations include std::get<N> to extract values, std::tie to create a tuple of references used for assignment, and lexicographic comparisons. C++17 proposes *structured bindings* [37] to eliminate pre-declaring variables and the use of std::tie for binding the results. This extension requires the use of **auto** to infer the types of the new variables; hence, complicated expressions with a nonobvious type must be documented with some other mechanism. Furthermore, structured bindings are not a full replacement for std::tie, as it always declares new variables. Java's variadic functions appear similar to C's but are type safe using homogeneous arrays. C∀'s heterogeneous variadic functions (see page 26) provide a type-safe version of C variadic, although limited in features, which fits into the overall C∀ type-system design.

4.3 Type Resolution

Psyche-C [21] is a tool built for inferencing missing type and variable declarations of incomplete C programs, which can also be viewed as a dialect of C with type inferencing. As PsycheC is built for analyzing standard C programs, it does not have any kind of overloading or polymorphism. Instead, all top-level variables and function parameters may have indeterminate types. The scope of the problem PsycheC deals with is quite the opposite of that in C∀, but the techniques used to solve for the unknown types are similar.

In PsycheC, a unification variable is assigned to every identifier with unknown type, and each use of a variable generates a constraint on its type. Just like in C∀, the most common kinds of expression contexts PsycheC deals with are assignments and function calls, which creates inequality constraints: the right hand side of the assignment is a subtype of the left hand side; function argument passing is treated the same as assignment. Since both sides of the inequality can have unknown types, type resolution needs to handle both upper bound and lower bound constraints. Instead of directly solving with lower and upper bounds, the authors developed a two-phase unification algorithm where the equivalences are first resolved and substituted into the inequalities, and showed that by ordering the inequalities using the partial order of concrete types involved, a second pass of unification treating inequalities as equivalences is sufficient to resolve all type variables. However, PsycheC does not consider the hierarchy of arithmetic types, as they are all mutually convertible in C, and therefore, considered *equivalent* in unification. It also lacks structured types except simple pointers, meaning their strategy of unifying subtypes as equivalences does not work in a more general setting such as in C∀.

Pierce and Turner [28] presented a *local* type-inference method that restricts the propagation of types and use of unification variables within adjacent syntax-tree nodes. The technique provides sufficient type checking and inference for the ML family of languages. Odersky, Zenger, and Zenger [27] refined this method by introducing a coloured type-system that distinguishes inherited types (given directly by context) and synthesized types (deducted from the term itself), to handle bidirectional type propagation in a more concise manner. Their algorithm is designed for functional programming, where a first-class (untyped) lambda expression is common and type inference must also deduce the argument types of lambdas requiring bidirectional inference. Given that C∀ does not have lambda expressions, but the rest of the type inference cases are quite similar, implementing this type-inference algorithm for C∀ expression resolution might be promising. Moreover, the current C∀ implementation also keeps unification variables *almost* local, except when a type variable does not appear in the parameter list. Hence, it cannot be given a value from the argument types; rather, it is passed upwards as a free variable for matching. In all other cases, the substituted minimal types are passed upwards as the result types. This process means the current implementation of C∀ expression resolution semantically agrees with Pierce's type-inference system in most of the cases. A question remains whether these methods can be adapted to handle C∀ assertion resolution.

Chapter 5

Future Work

The following are feature requests related to type-system enhancements that have surfaced during the development of the Cforall language and library, but have not been implemented yet. Currently, developers must work around these missing features, sometimes resulting in inefficiency. The following sections discuss new features I am proposing to fix these problems.

5.1 Closed Trait Types

Currently, Cforall does not have any closed types, as open types are the basis of its unique type-system, allowing new functions to be added at any time to override existing ones for trait satisfaction. Locally-declared nested-functions,[1] or a function-pointer variable can be used to make a type satisfy a certain trait temporarily in a limited lexical scope. However, the lack of closed types in such a *duck typing* scheme presents two problems.

1. Library implementers normally do not want users to override certain operations and cause the behaviour of polymorphic invocations to change.
2. Caching and reusing resolution results in the compiler is affected, as newly introduced declarations can participate in assertion resolution; as a result, previously invalid subexpressions suddenly become valid, or alternatively cause ambiguity in assertions.

Sometimes those interfaces are fairly complicated, *e.g.*, the Cforall I/O library traits istream and ostream each have over 20 operations. Without the ability to store and reuse assertion resolution results, each time the compiler encounters an I/O operation (denoted by the pipe operator ?|? like the <<, >> stream operators in C++) it has to resolve the same set of assertions again, causing a lot of repetitive work. Previous experiments have shown that the I/O assertions often account for over half of the number of assertions resolved in a Cforall translation unit. Hence, eliminating repetitive assertion resolution, which are very unlikely to change by new overloads, can provide a significant performance benefit.

Figure 5.1 shows the basic output-stream trait in Cforall and the overloaded I/O operator is declared as:

```
forall( ostype & | basic_ostream( ostype ) ) ostype & ?|?( ostype &, int ); // one for each basic type
```

The Cforall polymorphic calling convention means all of these trait functions are repeatedly resolved at each output-operator call-site and then become implicit arguments of the call. This creates both

[1] Nested functions are not a feature in C but supported by gcc for multiple decades and are used heavily in Cforall.

```
forall( ostype & ) trait basic_ostream {
    // private
    bool sepPrt$( ostype & );                       // get separator state (on/off)
    void sepReset$( ostype & );                     // set separator state to default state
    void sepReset$( ostype &, bool );               // set separator and default state
    const char * sepGetCur$( ostype & );            // get current separator string
    void sepSetCur$( ostype &, const char [] );     // set current separator string
    bool getNL$( ostype & );                        // get newline
    bool setNL$( ostype &, bool );                  // set newline
    bool getANL$( ostype & );                       // get auto newline (on/off)
    bool setANL$( ostype &, bool );                 // set auto newline (on/off)
    bool getPrt$( ostype & );                       // get fmt called in output cascade
    bool setPrt$( ostype &, bool );                 // set fmt called in output cascade
    // public
    void nlOn( ostype & );                          // turn auto-newline state on
    void nlOff( ostype & );                         // turn auto-newline state off
    void sep( ostype & );                           // turn separator state on
    void nosep( ostype & );                         // turn separator state off
    bool sepOn( ostype & );                         // set default state to on
    bool sepOff( ostype & );                        // set default state to off
    const char * sepGet( ostype & );                // get separator string
    void sepSet( ostype &, const char [] );         // set separator to string (15 maximum)
    const char * sepGetTuple( ostype & );           // get tuple separator string
    void sepSetTuple( ostype &, const char [] );    // set tuple separator to string (15 maximum)
    void ends( ostype & );                          // end of output statement
    int fmt( ostype &, const char format[], ... ) __attribute__(( format(printf, 2, 3) ));
};
```

Figure 5.1: basic_ostream trait

compile-time and run-time overheads. As stated, resolving assertions takes the most amount of time during compilation, and it is common to see in the current codebase and test suite that the majority of compilation time is spent in I/O statements. With closed types, the I/O assertions can be resolved once and this information reused everywhere. Furthermore, closed types can have a different calling convention of pushing a virtual-table pointer rather than individual arguments onto the stack. Although for I/O, this benefit might not be significant, since these operations involve system calls that are inherently slow.

A C∀ closed trait type is planned to be working similarly to a Haskell type class that requires an explicit instance declaration. The syntax for the closed trait might look like:

```
forall( ostype & ) closed trait basic_ostream { /* above declarations */ };
trait basic_ostream struct ofstream { ... }; // bind trait to type, implement basic_ostream
```

At the trait declaration, all the required trait functions must be inferred and specialized in the usual way. The list of inferred and specialized functions is then saved as an array of function pointers (vtable), and polymorphic functions using the closed type are passed the vtable, instead of a list of implicit parameters.

```
ostype ?|?( ostype &, int, basic_ostream_vtable * ); // parametric vtable
```

```
sout | 42;  /* becomes */  ?|?( sout, 42, ofstream_trait );  // generated for closed type ofstream
```

Closed types can provide additional encapsulation. In Figure 5.1, the functions ending with dollar sign $ are meant to be private (internal to the library) so client code should not be able to see or call these functions directly. This kind of information hiding is impossible with the current trait and assertion system, because the trait information must be visible at the call site for the polymorphic call to resolve. With closed traits and types, it is possible to forward declare a closed trait (which does not make sense otherwise) and its vtable without exposing any details to the client. That is, all the information necessary to call trait functions is now captured in a single vtable pointer built at compile time and the type of the vtable pointer is opaque.

5.2 Associated Types

The analysis presented in Section 3.3, p. 44 shows if all type parameters have to be bound before assertion resolution, the complexity of resolving assertions becomes much lower as every assertion parameter can be resolved independently. That is, by utilizing information from higher up the expression tree for return value overloading, most of the type bindings can be resolved. However, there are scenarios where some intermediate types need to be involved in certain operations, which are neither input nor output types.

C∀ standard library provides a few polymorphic container types similar to those found in the C++ standard library. Like C++, C∀ would like to have a harmonized iterator interface for different container types. While this feature is still under development, the following trait outline is used to present the problem.

```
forall( Container, Elem, Iter ) trait iterable {
    Iter begin( Container );
    Iter end( Container );
    Elem & *?( Iter );
    Iter ++?( Iter );
    bool ?==?( Iter, Iter );
};
```

These are the C++ operators required for for-loop comprehension, *e.g.*, **for** (Elem & e : container). Note, the iterable trait is meant to describe a property for a container type, but involves two additional types, one for the element type and one for the iterator type. However, Iter is intended to be deduced from the begin and end operations on the container type; the same can be said for the element type Elem. The alternative is to make Elem and Iter unbound type-parameters by removing them from the **forall**. As mentioned, unbound type-parameters cause problems, but that is the only way to describe the iterable property.

To solve this problem, I propose adding associate-type declarations in addition to the existing unbounded **forall** type parameter. In the body of a trait declaration, new types can be introduced as the return type of an expression involving type parameters.

```
forall( Container ) trait iterable {
    type Iter begin( Container );
    Iter end( Container );
    type Elem & *?( Iter );
```

```
    Iter ++?( Iter );
    bool ?==?( Iter, Iter );
};
```

This matches conceptually with the iterable trait having only the container type, rather than three types. The iterator type and element type are now viewed as properties of the container type not independent type parameters.

There remains one design decision to be made: whether the expression that defines an associated type needs to be uniquely resolved. If the associated type does not appear in the parameter or return types, this requirement seems reasonable. The reason is that the expression cost does not consider any conversions that occur in assertion parameter matching, which essentially means having multiple ways to resolve an associated type always results in ambiguity. The interesting case is when the associated type also appears in the return type, where both resolving the return type overload based on context and resolving the assertion that defines the associate type can help; for the whole expression to resolve, they must agree on a common result.

Moss gave the following example to illustrate C∀ assertion system's expressiveness.

```
forall( Ptr, Elem ) trait pointer_like {
    Elem & *?( Ptr ); // Ptr can be dereferenced to Elem
};
struct list {
    int value;
    list * next;
};
typedef list * list_iterator;
int & *?( list_iterator it ) {
    return it->value;
}
```

This can be viewed as having an associated type, Elem, as return, which can potentially vary based on context. Note that the type list * satisfies both pointer_like(list *, **int**) and pointer_like(list *, list) (the latter by the built-in pointer dereference operator) and the expression *it can be either a **struct** list or an **int**. Requiring associated types to be unique makes the pointer_like trait not applicable to list *, which is undesirable. I have not attempted to implement associated types in the C∀ compiler, but based on the above discussions, one option is to make associated type resolution and return type overloading coexist: when the associated type appears in returns, it is deduced from the context and then verify the trait with ordinary assertion resolution; when it does not appear in the returns, the type is required to be uniquely determined by the expression that defines the associated type.

5.3 User-defined Conversions

A missing type-system feature in C∀ is a scheme for user-defined conversions. Conversion means one type goes through an arbitrary complex process of changing its value to some meaningful value in another type. Because the conversion process can be arbitrarily complex, it requires the power of a function. Arbitrary conversions are already possible through standalone named functions.

```
S convert( T );                 // change T to S
S s;   T t;
s = convert( t );
```

C has implicit/explicit conversions for the builtin types.

```
int i = 3.5;                    // implicit, change the floating-point bits to integral bits
int i = (int)3.5 + 17;          // explicit, change the floating-point bits to integral bits
```

When the conversion occurs in expression evaluation can significantly affect the result, where function call has higher precedence than cast. Overloading the cast operator would allow user-defined conversions to look and feel, like regular C casts, as is provided in C++.

```
S ?$( T ) { /* change T to S */ }
s = t;
```

Similarly, constructors can be used for implicit conversion, as in C++.

```
void ?{}( S s, T t ) { ... }    // convert T to S
s = t;                          // rewritten as s = ?( s, t )
```

To integrate properly with the C∀ conversion model, there would need to distinguish between safe and unsafe conversion, and possibly a way to denote conversions as explicit-only or non-chainable, otherwise conversion cycles are possible.

Chapter 6

Conclusion

In the past few years of development, C∀ has gone from a proof-of-concept prototype to an actual experimental language, with a few medium-sized projects written completely in C∀ included in the language's libraries ($\approx$45,000 lines of code).[1] The work done in this thesis is motivated by real needs arising from the development and testing of these projects, which often pushes the limits of C∀'s type system and compiler capabilities. While most of the previous C∀ language feature and compiler developments were done either in isolation or with limited testing, getting them to work together and with real projects is presenting significant new challenges. These challenges could have been foreseen before development and testing began in earnest.

This work aims to identify and fix a number of practical issues of multiple C∀ type-system features and their interactions. In particular, the inclusion of reference types, tuple types, and generic structures together with rich overloading in the language makes the complexity of expression resolution much higher than in other programming languages. I significantly reworked the abstract syntax-tree representation and resolution algorithm to push the C∀ compilation time down to a practical level. The expression-cost system was also revised multiple times to make overload selection more predictable and match programmer's intuition and expectation in the majority of cases. Still, fundamental problems remain and fixing them will require significant changes to the language type-system, possibly from the ground up.

As per the C∀ project motto "describe not prescribe," C∀'s type system is designed to have a lot of flexibility and give programmers freedom in the usage of overloading and polymorphism. With such a complex type system, it is very difficult (sometimes even impossible) to try to have the compiler accept all the intuitively valid C∀ programs. As has been demonstrated, the C∀ programming language is still far from complete, and the primary future goal is to expand C∀'s type-resolution capability while maintaining, expressibility, decent compile-time, and excellent run-time performance. Stealing some theoretical insights of parametric polymorphism from functional programming, may also prove to be useful.

[1] In Fall 2024, two amazing CS343 students completed all 6 concurrent assignments in C∀. Many small language problems were uncovered and missing features discovered; these issues are being fixed for Fall 2025.

References

[1] 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. 36

[2] Walter Bright and Andrei Alexandrescu. *D Programming Language*. Digital Mars, Vienna Virginia, U.S.A., 2016. http://dlang.org/spec/spec.html. 51

[3] 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. 22, 51

[4] Tom A. Cargill. Controversy: The case against multiple inheritance in C++. *Computer Systems*, 4(1):69–82, Winter 1991. 32

[5] G. V. Cormack and A. K. Wright. Polymorphism in the compiled language ForceOne. In *Proceedings of the 20th Hawaii International Conference on Systems Sciences*, pages 284–292, January 1987. 11

[6] G. V. Cormack and A. K. Wright. Type-dependent parameter inference. *SIGPLAN Not.*, 25(6):127–136, June 1990. Proceedings of the ACM Sigplan'90 Conference on Programming Language Design and Implementation June 20-22, 1990, White Plains, New York, U.S.A. 11

[7] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'82, pages 207–212, New York, NY, USA, 1982. Association for Computing Machinery. 9

[8] 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. 36

[9] Dominic Duggan, Gordon V. Cormack, and John Ophel. Kinded type inference for parametric overloading. *Acta Infomatica*, 33(1):21–68, 1996. 11

[10] Rodolfo Gabriel Esteves. C∀, a study in evolutionary design in programming languages. Master's thesis, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, 2004. http://plg.uwaterloo.ca/theses/EstevesThesis.pdf. 24, 29

[11] GObject reference manual. https://developer.gnome.org/gobject/stable/, 2014. 51

[12] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *Java Language Specification*, Java SE 8 edition, 2015. 51

[13] Robert Griesemer, Rob Pike, and Ken Thompson. *Go Programming Language*. Google, Mountain View, CA, USA, 2009. https://go.dev/ref/spec. 15

[14] Dan Grossman. Quantified types in an imperative language. *Transactions on Programming Languages and Systems*, 28(3):429–475, May 2006. 50

[15] Haskell. *Haskell 2010 Language Report*, Simon Marlow edition, 2010. https://haskell.org/definition/haskell2010.pdf. 50

[16] International Standard Organization, Geneva, Switzerland. *C++ Programming Language ISO/IEC 14882:1998*, 1st edition, 1998. https://www.iso.org/standard/25845.html. 31

[17] International Standard Organization, Geneva, Switzerland. *C Programming Language ISO/IEC 9889:2011-12*, 3rd edition, December 2011. https://www.iso.org/standard/57853.html. 40

[18] International Standard Organization, Geneva, Switzerland. *C++ Programming language – Extensions for concepts ISO/IEC TS 19217:2015*, 2015. https://www.iso.org/standard/64031.html. 50

[19] Brian W. Kernighan and Dennis M. Ritchie. The m4 macro processor. https://wolfram.schneider.org/bsd/7thEdManVol2/m4/m4.pdf, 1977. 14

[20] Chris Lattner, Doug Gregor, John McCall, Ted Kremenek, Joe Groff, and Apple Inc. *The Swift Programming Language*. Apple Inc., Cupertino, CA, USA, 5.9.2 edition, 2024. https://docs.swift.org/swift-book/documentation/the-swift-programming-language. 51

[21] Leandro T. C. Melo, Rodrigo G. Ribeiro, Marcus R. de Araújo, and Fernando Magno Quintão Pereira. Inference of static semantics for incomplete C programs. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. 52

[22] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978. 50

[23] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, 1990. 52

[24] Aaron Moss. *C∀ Type System Implementation*. PhD thesis, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, 2019. https://uwspace.uwaterloo.ca/handle/10012/14584. 14, 22, 25, 36, 42

[25] 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. 19

[26] Objective-C. https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC, 2014. 51

[27] 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. 53

[28] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, January 2000. 53

[29] *Rust Programming Language*. Rust Project Developers, 2015. https://doc.rust-lang.org/reference.html. 15, 51

[30] *Scala Language Specification, Version 2.11*. École Polytechnique Fédérale de Lausanne, 2016. http://www.scala-lang.org/files/archive/spec/2.11. 52

[31] Robert Schluntz. Resource management and tuples in C∀. Master's thesis, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, 2017. https://uwspace.uwaterloo.ca/handle/10012/11830. 25, 26

[32] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL*. Springer, New York, NY, USA, 1986. 51

[33] Mary Shaw, editor. *ALPHARD: Form and Content*. Springer, New York, 1981. 11

[34] Geoffrey Smith and Dennis Volpano. A sound polymorphic type system for a dialect of C. *Science of computer programming*, 32(1-3):49–72, 1998. 51

[35] Christopher Strachey. Fundamental concepts in programming languages. *Higher Order Symbol. Comput.*, 13(1-2):11–49, April 2000. 3

[36] Bjarne Stroustrup. Multiple inheritance for C++. *Computer Systems*, 2(4):367–395, Fall 1989. 32

[37] Herb Sutter, Bjarne Stroustrup, and Gabriel Dos Reis. Structured bindings. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0144r0.pdf, October 2015. 52

[38] Ken Thompson. A new C compiler. In *Proceedings of the Summer 1990 UKUUG Conference*, pages 41–51, 1990. https://doc.cat-v.org/bell_labs/new_c_compilers/new_c_compiler.pdf. 31

[39] David W. Till. Tuples in imperative programming languages. Master's thesis, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, 1989. 22, 23

[40] Vala reference manual. https://wiki.gnome.org/Projects/Vala/Manual, 2017. 51

[41] WikipediA. Function overloading. https://en.wikipedia.org/wiki/Function_overloading, 2025. 6

[42] WikipediA. Operator overloading. https://en.wikipedia.org/wiki/Operator_overloading, 2025. 6

[43] Fangren Yu. Optimization of C∀ compiler with case studies. Technical report, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, January 2021. https://cforall.uwaterloo.ca/doc/Fangren_Yu_Report_F20.pdf. 18, 34