Received: 12 March 2018 | Revised: 8 May 2018 | Accepted: 28 June 2018

DOI: 10.1002/spe.2624

RESEARCH ARTICLE

WILEY

C∀: Adding modern programming language features to C

Aaron Moss | Robert Schluntz | Peter A. Buhr

Cheriton School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada

Correspondence
Peter A. Buhr, Cheriton School of Computer Science, University of Waterloo, 200 University Avenue West, Waterloo, ON N2L 3G1, Canada.
Email: pabuhr@uwaterloo.ca

Funding information
Natural Sciences and Engineering Research Council of Canada

Summary

The C programming language is a foundational technology for modern computing with millions of lines of code implementing everything from hobby projects to commercial operating systems. This installation base and the programmers producing it represent a massive software engineering investment spanning decades and likely to continue for decades more. Nevertheless, C, which was first standardized almost 30 years ago, lacks many features that make programming in more modern languages safer and more productive. The goal of the C∀ project (pronounced "C for all") is to create an extension of C that provides modern safety and productivity features while still ensuring strong backward compatibility with C and its programmers. Prior projects have attempted similar goals but failed to honor the C programming style; for instance, adding object-oriented or functional programming with garbage collection is a nonstarter for many C developers. Specifically, C∀ is designed to have an orthogonal feature set based closely on the C programming paradigm, so that C∀ features can be added *incrementally* to existing C code bases, and C programmers can learn C∀ extensions on an as-needed basis, preserving investment in existing code and programmers. This paper presents a quick tour of C∀ features, showing how their design avoids shortcomings of similar features in C and other C-like languages. Experimental results are presented to validate several of the new features.

KEYWORDS

C, Cforall, generic types, polymorphic functions, tuple types, variadic types

1 | INTRODUCTION

The C programming language is a foundational technology for modern computing with millions of lines of code implementing everything from hobby projects to commercial operating systems. This installation base and the programmers producing it represent a massive software engineering investment spanning decades and likely to continue for decades more. The TIOBE index[1] ranks the top five most *popular* programming languages as Java 15%, **C 12%**, **C++ 5.5%**, and Python 5%, C♯ 4.5% = 42%, where the next 50 languages are less than 4% each with a long tail. The top three rankings over the past 30 years are as follows.

	2018	2013	2008	2003	1998	1993	1988
Java	1	2	1	1	18	–	–
C	**2**	**1**	**2**	**2**	**1**	**1**	**1**
C++	3	4	3	3	2	2	5

 © 2018 John Wiley & Sons, Ltd.

Love it or hate it, C is extremely popular, highly used, and one of the few systems languages. In many cases, C++ is often used solely as a better C. Nevertheless, C, which was first standardized almost 30 years ago,[2] lacks many features that make programming in more modern languages safer and more productive.

C∀ (pronounced "C for all" and written C∀ or Cforall) is an evolutionary extension of the C programming language that adds modern language features to C, while maintaining source and runtime compatibility in the familiar C programming model. The four key design goals for C∀[3] are as follows: (1) the behavior of the standard C code must remain the same when translated by a C∀ compiler as when translated by a C compiler; (2) the standard C code must be as fast and as small when translated by a C∀ compiler as when translated by a C compiler; (3) the C∀ code must be at least as portable as the standard C code; (4) extensions introduced by C∀ must be translated in the most efficient way possible. These goals ensure that the existing C code bases can be converted into C∀ incrementally with minimal effort, and C programmers can productively generate the C∀ code without training beyond the features being used. C++ is used similarly but has the disadvantages of multiple legacy design choices that cannot be updated and active divergence of the language model from C, requiring significant effort and training to incrementally add C++ to a C-based project.

All language features discussed in this paper are working, except for some advanced exception-handling features. Not discussed in this paper are the integrated concurrency constructs and user-level threading library.[4] C∀ is an *open-source* project implemented as a source-to-source translator from C∀ to the gcc-dialect of C,[5] allowing it to leverage the portability and code optimizations provided by gcc, meeting goals (1)-(3). The C∀ translator is 200+ files and 46 000+ lines of code written in C/C++. A translator versus a compiler makes it easier and faster to generate and debug the C object code rather than the intermediate, assembler, or machine code; ultimately, a compiler is necessary for advanced features and optimal performance. Two key translator components are expression analysis, determining expression validity and what operations are required for its implementation, and code generation, dealing with multiple forms of overloading, polymorphism, and multiple return values by converting them into the C code for a C compiler that supports none of these features. Details of these components are available in chapters 2 and 3 in the work of Bilson[3] and form the base for the current C∀ translator. The C∀ runtime system is 100+ files and 11 000+ lines of code, written in C∀. Currently, the C∀ runtime is the largest *user* of C∀, providing a vehicle to test the language features and implementation.

Finally, it is impossible to describe a programming language without usage before definition. Therefore, syntax and semantics appear before explanations; hence, patience is necessary until sufficient details are presented and discussed. Similarly, a detailed comparison with other programming languages is postponed until Section 10.

2 | POLYMORPHIC FUNCTIONS

C∀ introduces both ad hoc and parametric polymorphism to C, with a design originally formalized by Ditchfield[6] and first implemented by Bilson.[3] Shortcomings are identified in the existing approaches to generic and variadic data types in C-like languages and how these shortcomings are avoided in C∀. Specifically, the solution is both reusable and type checked, as well as conforming to the design goals of C∀ with ergonomic use of existing C abstractions. The new constructs are empirically compared with C and C++ approaches via performance experiments in Section 9.

2.1 | Name overloading

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

C already has a limited form of ad hoc polymorphism in its basic arithmetic operators, which apply to a variety of different types using identical syntax. C∀ extends the built-in operator overloading by allowing users to define overloads for any function, not just operators, and even any variable; Section 8 includes a number of examples of how this overloading simplifies C∀ programming relative to C. Code generation for these overloaded functions and variables is implemented by the usual approach of mangling the identifier names to include a representation of their type, while C∀ decides which overload to apply based on the same "usual arithmetic conversions" used in C to disambiguate operator overloads.

1097024x, 2018, 12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2624 by University Of Waterloo Dana Porter Library, Wiley Online Library on [22/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

```
int max = 2147483647;                                          // (1)
double max = 1.7976931348623157E+308;                          // (2)
int max( int a, int b ) { return a < b ? b : a; }              // (3)
double max( double a, double b ) { return a < b ? b : a; }     // (4)
max( 7, -max );                                  // uses (3) and (1), by matching int from constant 7
max( max, 3.14 );                                // uses (4) and (2), by matching double from constant 3.14
max( max, -max );                                // ERROR, ambiguous
int m = max( max, -max );                        // uses (3) and (1) twice, by matching return type
```

C∀ maximizes the ability to reuse names to aggressively address the naming problem. In some cases, hundreds of names can be reduced to tens, resulting in a significant cognitive reduction. In the above, the name max has a consistent meaning, and a programmer only needs to remember the single concept: maximum. To prevent significant ambiguities, C∀ uses the return type in selecting overloads, eg, in the assignment to m, the compiler uses m's type to unambiguously select the most appropriate call to function max (as does Ada). As is shown later, there are a number of situations where C∀ takes advantage of available type information to disambiguate, where other programming languages generate ambiguities.

C11 added **_Generic** expressions (see section 6.5.1.1 of the ISO/IEC 9899[7]), which is used with preprocessor macros to provide ad hoc polymorphism; however, this polymorphism is both functionally and ergonomically inferior to C∀ name overloading. The macro wrapping the generic expression imposes some limitations, for instance, it cannot implement the example above, because the variables max are ambiguous with the functions max. Ergonomic limitations of **_Generic** include the necessity to put a fixed list of supported types in a single place and manually dispatch to appropriate overloads, as well as possible namespace pollution from the dispatch functions, which must all have distinct names. C∀ supports **_Generic** expressions for backward compatibility, but it is an unnecessary mechanism.

2.2 | forall functions

The signature feature of C∀ is parametric-polymorphic functions[8-10] with functions generalized using a **forall** clause (giving the language its name).

```
forall( otype 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 any complete *object type* (or **otype**). The type variable T is transformed into a set of additional implicit parameters encoding sufficient information about T to create and return a variable of that type. The C∀ implementation passes the size and alignment of the type represented by an **otype** parameter, as well as an assignment operator, constructor, copy constructor, and destructor. If this extra information is not needed, for instance, for a pointer, the type parameter can be declared as a *data type* (or **dtype**).

In C∀, the polymorphic runtime cost is spread over each polymorphic call, because more arguments are passed to polymorphic functions; the experiments in Section 9 show this overhead is similar to C++ virtual function calls. A design advantage is that, unlike C++ template functions, C∀ polymorphic functions are compatible with C *separate compilation*, preventing compilation and code bloat.

Since bare polymorphic types provide a restricted set of available operations, C∀ provides a *type assertion*[11(pp37-44)] mechanism to provide further type information, where type assertions may be variable or function declarations that depend on a polymorphic type variable. For example, the function twice can be defined using the C∀ syntax for operator overloading.

```
forall( otype T | { T ?+?(T, T); } ) T twice( T x ) { return x + x; }  // ? denotes operands
int val = twice( twice( 3.7 ) );                                       // val == 14
```

This works for any type T with a matching addition operator. The polymorphism is achieved by creating a wrapper function for calling + with the T bound to **double** and then passing this function to the first call of twice. There is now the option of using the same twice and converting the result into **int** on assignment or creating another twice with the type parameter T bound to **int** because C∀ uses the return type[12-14] in its type analysis. The first approach has a late conversion from **double** to **int** on the final assignment, whereas the second has an early conversion to **int**. C∀ minimizes the number of conversions and their potential to lose information; hence, it selects the first approach, which corresponds with C programmer intuition.

1097024x, 2018, 12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2624 by University Of Waterloo Dana Porter Library, Wiley Online Library on [22/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Crucial to the design of a new programming language are the libraries to access thousands of external software features. Like C++, C∀ inherits a massive compatible library base, where other programming languages must rewrite or provide fragile interlanguage communication with C. A simple example is leveraging the existing type-unsafe (**void** *) C bsearch to binary search a sorted float array.

```
void * bsearch( const void * key, const void * base, size_t nmemb, size_t size,
                int (* compar)( const void *, const void * ));
int comp( const void * t1, const void * t2 ) {
    return *(double *)t1 < *(double *)t2 ? -1 : *(double *)t2 < *(double *)t1 ? 1 : 0;
}
double key = 5.0, vals[10] = { /* 10 sorted float values */ };
double * val = (double *)bsearch( &key, vals, 10, sizeof(vals[0]), comp ); // search sorted array
```

This can be augmented simply with generalized, type-safe, C∀-overloaded wrappers.

```
forall( otype T | { int ?<?( T, T ); } ) T * bsearch( T key, const T * arr, size_t size ) {
    int comp( const void * t1, const void * t2 ) { /* as above with double changed to T */ }
    return (T *)bsearch( &key, arr, size, sizeof(T), comp );
}
forall( otype T | { int ?<?( T, T ); } ) unsigned int bsearch( T key, const T * arr, size_t size ) {
    T * result = bsearch( key, arr, size );      // call first version
    return result ? result - arr : size;         // pointer subtraction includes sizeof(T)
}
double * val = bsearch( 5.0, vals, 10 );         // selection based on return type
int posn = bsearch( 5.0, vals, 10 );
```

The nested function comp provides the hidden interface from typed C∀ to untyped (**void** *) C, plus the cast of the result. Providing a hidden comp function in C++ is awkward as lambdas do not use C calling conventions and template declarations cannot appear at a block scope. In addition, an alternate kind of return is made available: position versus pointer to found element. C++'s type system cannot disambiguate between the two versions of bsearch because it does not use the return type in overload resolution, nor can C++ separately compile a template bsearch.

C∀ has replacement libraries condensing hundreds of existing C functions into tens of C∀ overloaded functions, all without rewriting the actual computations (see Section 8). For example, it is possible to write a type-safe C∀ wrapper malloc based on the C malloc, where the return type supplies the type/size of the allocation, which is impossible in most type systems.

```
forall( dtype T | sized(T) ) T * malloc( void ) { return (T *)malloc( sizeof(T) ); }
// select type and size from left-hand side
int * ip = malloc();  double * dp = malloc();  struct S {...} * sp = malloc();
```

Call site inferencing and nested functions provide a localized form of inheritance. For example, the C∀ qsort only sorts in ascending order using <. However, it is trivial to locally change this behavior.

```
forall( otype T | { int ?<?( T, T ); } ) void qsort( const T * arr, size_t size ) { /* use C qsort */ }
int main() {
    int ?<?( double x, double y ) { return x > y; }    // locally override behaviour
    qsort( vals, 10 );                                  // descending sort
}
```

The local version of ?<? performs ?>? overriding the built-in ?<?; hence, it is passed to qsort. Therefore, programmers can easily form local environments, adding and modifying appropriate functions, to maximize the reuse of other existing functions and types.

To reduce duplication, it is possible to distribute a group of **forall** (and storage-class qualifiers) over functions/types, such that each block declaration is prefixed by the group (see the example in Appendix A.2).

```
forall( otype T ) {                              // distribution block, add forall qualifier to declarations
    struct stack { stack_node(T) * head; };      // generic type
    inline {                                     // nested distribution block, add forall/inline to declarations
        void push( stack(T) & s, T value ) ...   // generic operations
    }
}
```

1097024x, 2018, 12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2624 by University Of Waterloo Dana Porter Library, Wiley Online Library on [22/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

2.3 | 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.

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

```
forall( otype T | sumable( T ) ) // use trait
T sum( T a[], size_t size ) {
    T total = { 0 }; // initialize by 0 constructor
    for ( size_t i = 0; i < size; i += 1 )
        total += a[i]; // select appropriate +
    return total;
}
```

Note that the sumable trait does not include a copy constructor needed for the right side of ?+=? and return; it is provided by **otype**, which is syntactic sugar for the following trait.

```
trait otype( dtype T | sized(T) ) { // sized is a pseudo-trait for types with known size and alignment
    void ?{}( T & );            // default constructor
    void ?{}( T &, T );         // copy constructor
    void ?=?( T &, T );         // assignment operator
    void ^?{}( T & );           // destructor
};
```

Given the information provided for an **otype**, variables of polymorphic type can be treated as if they were a complete type: stack allocatable, default or copy initialized, assigned, and deleted.

In summation, the C∀ type system uses *nominal typing* for concrete types, matching with the C type system, and *structural typing* for polymorphic types. Hence, trait names play no part in type equivalence; the names are simply macros for a list of polymorphic assertions, which are expanded at usage sites. Nevertheless, trait names form a logical subtype hierarchy with **dtype** at the top, where traits often contain overlapping assertions, eg, operator +. Traits are used 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 type needed for its execution (polymorphic type key), and this key is fulfilled at each call site from the lexical environment, which is similar to the Go[15] interfaces. Hence, new lexical scopes and nested functions are used extensively to create local subtypes, as in the qsort example, without having to manage a nominal inheritance hierarchy.

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. One approach is to write bespoke data structures for each context in which they are needed. While this approach is flexible and supports integration with the C type checker and tooling, it is also tedious and error prone, especially for more complex data structures. A second approach is to use **void** ∗-based polymorphism, eg, the C standard library functions bsearch and qsort, which allow for the reuse of code with common functionality. However, basing all polymorphism on **void** ∗ eliminates the type checker's ability to ensure that argument types are properly matched, often requiring a number of extra function parameters, pointer indirection, and dynamic allocation that is otherwise not needed. A third approach to generic code is to use preprocessor macros, which does allow the generated code to be both generic and type checked, but errors may be difficult to interpret. Furthermore, writing and using preprocessor macros is unnatural and inflexible.

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. However, for known concrete parameters, the generic-type definition can be inlined, like C++ templates.

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.

1097024x, 2018, 12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2624 by University Of Waterloo Dana Porter Library, Wiley Online Library on [22/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

```
forall( otype R, otype S ) struct pair {
    R first;  S second;
};
forall( otype T ) // dynamic
T value( pair(const char *, T) p ) { return p.second; }
forall( dtype F, otype T ) // dtype-static (concrete)
T value( pair(F *, T * ) p) { return *p.second; }
```

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

C∀ classifies generic types as either *concrete* or *dynamic*. Concrete types have a fixed memory layout regardless of type parameters, whereas dynamic types vary in memory layout depending on their type parameters. A *dtype-static* type has polymorphic parameters but is still concrete. Polymorphic pointers are an example of dtype-static types; given some type variable T, T is a polymorphic type, as is T *, but T * has a fixed size and can, therefore, be represented by **void** * in code generation.

C∀ generic types also allow checked argument constraints. For example, the following declaration of a sorted set type ensures that the set key supports equality and relational comparison.

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

3.1 | Concrete generic types

The C∀ translator template expands concrete generic types into new structure types, affording maximal inlining. To enable interoperation among equivalent instantiations of a generic type, the translator saves the set of instantiations currently in scope and reuses the generated structure declarations where appropriate. A function declaration that accepts or returns a concrete generic type produces a declaration for the instantiated structure in the same scope, which all callers may reuse. For example, the concrete instantiation for pair(**const char** *, **int**) is

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

A concrete generic type with dtype-static parameters is also expanded to a structure type, but this type is used for all matching instantiations. In the above example, the pair(F *, T *) parameter to value is such a type; its expansion is below, and it is used as the type of the variables q and r as well, with casts for member access where appropriate.

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

3.2 | Dynamic generic types

Though C∀ implements concrete generic types efficiently, it also has a fully general system for dynamic generic types. As mentioned in Section 2.2, **otype** function parameters (in fact, all sized polymorphic parameters) come with implicit size and alignment parameters provided by the caller. Dynamic generic types also have an *offset array* containing structure-member offsets. A dynamic generic **union** needs no such offset array, as all members are at offset 0, but size and alignment are still necessary. Access to members of a dynamic structure is provided at runtime via base displacement addressing the structure pointer and the member offset (similar to the offsetof macro), moving a compile-time offset calculation to runtime.

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

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

1097024x, 2018, 12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2624 by University Of Waterloo Dana Porter Library, Wiley Online Library on [22/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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

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

In some cases, the offset arrays cannot be statically generated. For instance, modularity is generally provided in C by including an opaque forward declaration of a structure and associated accessor and mutator functions in a header file, with the actual implementations in a separately compiled .c file. C∀ supports this pattern for generic types, but the caller does not know the actual layout or size of the dynamic generic type and only holds it by a pointer. The C∀ translator automatically generates *layout functions* for cases where the size, alignment, and offset array of a generic struct cannot be passed into a function from that function's caller. These layout functions take as arguments pointers to size and alignment variables and a caller-allocated array of member offsets, as well as the size and alignment of all sized parameters to the generic structure (unsized parameters are forbidden from being used in a context that affects layout). Results of these layout functions are cached so that they are only computed once per type per function. Layout functions also allow generic types to be used in a function definition without reflecting them in the function signature. For instance, a function that strips duplicate values from an unsorted vector(T) likely has a pointer to the vector as its only explicit parameter, but uses some sort of set(T) internally to test for duplicate values. This function could acquire the layout for set(T) by calling its layout function with the layout of T implicitly passed into the function.

Whether a type is concrete, dtype-static, or dynamic is decided solely on the **forall**'s type parameters. This design allows opaque forward declarations of generic types, eg, **forall(otype** T)**struct** Box; like in C, all uses of Box(T) can be separately compiled, and callers from other translation units know the proper calling conventions to use. If the definition of a structure type is included in deciding whether a generic type is dynamic or concrete, some further types may be recognized as dtype-static (eg, **forall**(**otype** T)**struct** unique_ptr { T * p } does not depend on T for its layout, but the existence of an **otype** parameter means that it *could*.); however, preserving separate compilation (and the associated C compatibility) in the existing design is judged to be an appropriate trade-off.

3.3 | Applications

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

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

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

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

```
forall( dtype Unit ) struct scalar { unsigned long value; };
struct metres {};
struct litres {};
forall( dtype U ) scalar(U) ?+?( scalar(U) a, scalar(U) b ) {
    return (scalar(U)){ a.value + b.value };
}
```

```
scalar(metres) half_marathon = { 21_098 };
scalar(litres) pool = { 2_500_000 };
scalar(metres) marathon = half_marathon +
                          half_marathon;
scalar(litres) two_pools = pool + pool;
marathon + pool;    // ERROR, mismatched types
```

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

1097024x, 2018, 12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2624 by University Of Waterloo Dana Porter Library, Wiley Online Library on [22/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

4 | TUPLES

In many languages, functions can return, at most, one value; however, many operations have multiple outcomes, some exceptional. Consider C's div and remquo functions, which return the quotient and remainder for a division of integer and float values, respectively.

```
typedef struct { int quo, rem; } div_t;            // from include stdlib.h
div_t div( int num, int den );
double remquo( double num, double den, int * quo );
div_t qr = div( 13, 5 );                            // return quotient/remainder aggregate
int q;
double r = remquo( 13.5, 5.2, &q );                 // return remainder, alias quotient
```

Here, div aggregates the quotient/remainder in a structure, whereas remquo aliases a parameter to an argument. Both approaches are awkward. Alternatively, a programming language can directly support returning multiple values, eg, C∀ provides the following

```
[ int, int ] div( int num, int den );               // return two integers
[ double, double ] div( double num, double den );   // return two doubles
int q, r;                                           // overloaded variable names
double q, r;
[ q, r ] = div( 13, 5 );                            // select appropriate div and q, r
[ q, r ] = div( 13.5, 5.2 );                        // assign into tuple
```

This approach is straightforward to understand and use; therefore, why do few programming languages support this obvious feature or provide it awkwardly? To answer, there are complex consequences that cascade through multiple aspects of the language, especially the type system. This section shows these consequences and how C∀ handles them.

4.1 | Tuple expressions

The addition of multiple-return-value functions (MRVFs) is *useless* without a syntax for accepting multiple values at the call site. The simplest mechanism for capturing the return values is variable assignment, allowing the values to be retrieved directly. As such, C∀ allows assigning multiple values from a function into multiple variables, using a square-bracketed list of lvalue expressions (as above), called a *tuple*.

However, functions also use *composition* (nested calls), with the direct consequence that MRVFs must also support composition to be orthogonal with single-returning-value functions (SRVFs). C∀ provides the following.

```
printf( "%d %d\n", div( 13, 5 ) );                 // return values seperated into arguments
```

Here, the values returned by div are composed with the call to printf by flattening the tuple into separate arguments. However, the C∀ type system must support significantly more complex composition.

```
[ 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, which involves unifying the possible foo functions with bar's parameter list. No combination of foo's is an exact match with bar's parameters; thus, the resolver applies C conversions. The minimal cost is bar(foo_1(3) foo_2(3)), giving (**int**, **int**, **double**) to (**int**, **double**, **double**) with one 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.

4.2 | Tuple variables

An important observation from function composition is that new variable names are not required to initialize parameters from an MRVF. C∀ also allows declaration of tuple variables that can be initialized from an MRVF, since it can be awkward to declare multiple variables of different types.

1097024x, 2018, 12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2624 by University Of Waterloo Dana Porter Library, Wiley Online Library on [22/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

```
[ int, int ] qr = div( 13, 5 );            // tuple-variable declaration and initialization
[ double, double ] qr = div( 13.5, 5.2 );
```

Here, the tuple variable name serves the same purpose as the parameter name(s). Tuple variables can be composed of any types, except for array types, since array sizes are generally unknown in C.

One way to access the tuple variable components is with assignment or composition.

```
[ q, r ] = qr;                             // access tuple-variable components
printf( "%d %d\n", qr );
```

C∀ also supports *tuple indexing* to access single components of a tuple expression.

```
[int, int] * p = &qr;                      // tuple pointer
int rem = qr.1;                            // access remainder
int quo = div( 13, 5 ).0;                  // access quotient
p->0 = 5;                                  // change quotient
bar( qr.1, qr );                           // pass remainder and quotient/remainder
rem = [div( 13, 5 ), 42].0.1;              // access 2nd component of 1st component
```

4.3 | Flattening and restructuring

In function call contexts, tuples support implicit flattening and restructuring conversions. Tuple flattening recursively expands a tuple into the list of its basic components. Tuple structuring packages a list of expressions into a value of tuple type.

```
int f( int, int );
[int] g( [int, int] );
[int] h( int, [int, int] );
[int, int] x;
int y;
f( x );                                    // flatten
g( y, 10 );                                // structure
h( x, y );                                 // flatten and structure
```

In the call to f, x is implicitly flattened so the components of x are passed as two arguments. In the call to g, the values y and 10 are structured into a single argument of type [int, int] to match the parameter type of g. Finally, in the call to h, x is flattened to yield an argument list of length 3, of which the first component of x is passed as the first parameter of h, and the second component of x and y are structured into the second argument of type [int, int]. The flexible structure of tuples permits a simple and expressive function call syntax to work seamlessly with both SRVFs and MRVFs with any number of arguments of an arbitrarily complex structure.

4.4 | Tuple assignment

An assignment where the left side is a tuple type is called *tuple assignment*. There are two kinds of tuple assignment depending on whether the right side of the assignment operator has a tuple type or a nontuple type, called *multiple* and *mass assignment*, respectively.

```
int x = 10;
double y = 3.5;
[int, double] z;
z = [x, y];                                // multiple assignment
[x, y] = z;                                // multiple assignment
z = 10;                                    // mass assignment
[y, x] = 3.14;                             // mass assignment
```

Both kinds of tuple assignment have parallel semantics, so that each value on the left and right sides is evaluated before any assignments occur. As a result, it is possible to swap the values in two variables without explicitly creating any temporary variables or calling a function, eg, [x, y] = [y, x]. This semantics means mass assignment differs from C cascading

1097024x, 2018, 12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2624 by University Of Waterloo Dana Porter Library, Wiley Online Library on [22/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

assignment (eg, a = b = c) in that conversions are applied in each individual assignment, which prevents data loss from the chain of conversions that can happen during a cascading assignment. For example, [y, x] = 3.14 performs the assignments y = 3.14 and x = 3.14, yielding y == 3.14 and x == 3, whereas C cascading assignment y = x = 3.14 performs the assignments x = 3.14 and y = x, yielding 3 in y and x. Finally, tuple assignment is an expression where the result type is the type of the left-hand side of the assignment, just like all other assignment expressions in C. This example shows mass, multiple, and cascading assignment used in one expression.

```
[void] f( [int, int] );
f( [x, y] = z = 1.5 );                    // assignments in parameter list
```

4.5 | Member access

It is also possible to access multiple members from a single expression using a *member access*. The result is a single tuple-valued expression whose type is the tuple of the types of the members.

```
struct S { int x; double y; char * z; } s;
s.[x, y, z] = 0;
```

Here, the mass assignment sets all members of s to zero. Since tuple-index expressions are a form of member-access expression, it is possible to use tuple-index expressions in conjunction with member-tuple expressions to manually restructure a tuple (eg, rearrange, drop, and duplicate components).

```
[int, int, long, double] x;
void f( double, long );
x.[0, 1] = x.[1, 0];                      // rearrange: [x.0, x.1] = [x.1, x.0]
f( x.[0, 3] );                            // drop: f(x.0, x.3)
[int, int, int] y = x.[2, 0, 2];          // duplicate: [y.0, y.1, y.2] = [x.2, x.0.x.2]
```

It is also possible for a member access to contain other member accesses.

```
struct A { double i; int j; };
struct B { int * k; short l; };
struct C { int x; A y; B z; } v;
v.[x, y.[i, j], z.k];                     // [v.x, [v.y.i, v.y.j], v.z.k]
```

4.6 | Polymorphism

Tuples also integrate with C∀ polymorphism as a kind of generic type. Due to the implicit flattening and structuring conversions involved in argument passing, **otype** and **dtype** parameters are restricted to matching only with nontuple types.

```
forall( otype T, dtype U ) void f( T x, U * y );
f( [5, "hello"] );
```

Here, [5, "hello"] is flattened, giving argument list 5, "hello", and T binds to **int** and U binds to **const char**. Tuples, however, may contain polymorphic components. For example, a plus operator can be written to sum two triples.

```
forall( otype T | { T ?+?( T, T ); } ) [T, T, T] ?+?( [T, T, T] x, [T, T, T] y ) {
    return [x.0 + y.0, x.1 + y.1, x.2 + y.2];
}
[int, int, int] x;
int i1, i2, i3;
[i1, i2, i3] = x + ([10, 20, 30]);
```

Flattening and restructuring conversions are also applied to tuple types in polymorphic type assertions.

```
[int] f( [int, double], double );
forall( otype T, otype U | { T f( T, U, U ); } ) void g( T, U );
g( 5, 10.21 );
```

1097024x, 2018, 12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2624 by University Of Waterloo Dana Porter Library, Wiley Online Library on [22/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Hence, function parameter and return lists are flattened for the purposes of type unification allowing the example to pass expression resolution. This relaxation is possible by extending the thunk scheme described by Bilson.[3]

4.7 | Variadic tuples

To define variadic functions, C∀ adds a new kind of type parameter, ie, **ttype** (tuple type). Matching against a **ttype** parameter consumes all the remaining argument components and packages them into a tuple, binding to the resulting tuple of types. In a given parameter list, there must be, at most, one **ttype** parameter that occurs last, which matches normal variadic semantics, with a strong feeling of similarity to C++11 variadic templates. As such, **ttype** variables are also called *argument packs*.

Like variadic templates, **ttype** polymorphic functions are primarily manipulated via recursion. Since nothing is known about a parameter pack by default, assertion parameters are key to doing anything meaningful. Unlike variadic templates, **ttype** polymorphic functions can be separately compiled. For example, the following is a generalized sum function.

```
int sum0() { return 0; }
forall( ttype Params | { int sum( Params ); } ) int sum1( int x, Params rest ) {
    return x + sum( rest );
}
sum( 10, 20, 30 );
```

Since sum_0 does not accept any arguments, it is not a valid candidate function for the call sum(10, 20, 30). In order to call sum_1, 10 is matched with x, and the argument resolution moves on to the argument pack rest, which consumes the remainder of the argument list, and Params is bound to [20, 30]. The process continues until Params is bound to [], requiring an assertion **int** sum(), which matches sum_0 and terminates the recursion. Effectively, this algorithm traces as sum(10, 20, 30) →10 + sum(20, 30) →10 + (20 + sum(30)) →10 + (20 + (30 + sum())) →10 + (20 + (30 + 0)).

It is reasonable to take the sum function a step further to enforce a minimum number of arguments.

```
int sum( int x, int y ) { return x + y; }
forall( ttype Params | { int sum( int, Params ); } ) int sum( int x, int y, Params rest ) {
    return sum( x + y, rest );
}
```

One more step permits the summation of any sumable type with all arguments of the same type.

```
trait sumable( otype T ) {
    T ?+?( T, T );
};
forall( otype R | sumable( R ) ) R sum( R x, R y ) {
    return x + y;
}
forall( otype R, ttype Params | sumable(R) | { R sum(R, Params); } ) R sum(R x, R y, Params rest) {
    return sum( x + y, rest );
}
```

Unlike C variadic functions, it is unnecessary to hard code the number and expected types. Furthermore, this code is extendable for any user-defined type with a ?+? operator. Summing arbitrary heterogeneous lists is possible with similar code by adding the appropriate type variables and addition operators.

It is also possible to write a type-safe variadic print function to replace printf.

```
struct S { int x, y; };
forall( otype T, ttype Params | { void print(T); void print(Params); } ) void print(T arg, Params rest) {
    print(arg);  print(rest);
}
void print( const char * x ) { printf( "%s", x ); }
void print( int x ) { printf( "%d", x ); }
void print( S s ) { print( "{ ", s.x, ",", s.y, " }" ); }
print( "s = ", (S){ 1, 2 }, "\n" );
```

1097024x, 2018, 12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2624 by University Of Waterloo Dana Porter Library, Wiley Online Library on [22/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

This example showcases a variadic-template–like decomposition of the provided argument list. The individual print functions allow printing a single element of a type. The polymorphic print allows printing any list of types, where each individual type has a print function. The individual print functions can be used to build up more complicated print functions, such as S, which cannot be done with printf in C. This mechanism is used to seamlessly print tuples in the C∀ I/O library (see Section 8.4).

Finally, it is possible to use **ttype** polymorphism to provide arbitrary argument forwarding functions. For example, it is possible to write new as a library function.

```
forall( otype R, otype S ) void ?{}( pair(R, S) *, R, S );
forall( dtype T, ttype Params | sized(T) | { void ?{}( T *, Params ); } ) T * new( Params p ) {
    return ((T *)malloc()){ p };                    // construct into result of malloc
}
pair( int, char ) * x = new( 42, '!' );
```

The new function provides the combination of type-safe malloc with a C∀ constructor call, making it impossible to forget constructing dynamically allocated objects. This function provides the type safety of new in C++, without the need to specify the allocated type again, due to return-type inference.

4.8 | Implementation

Tuples are implemented in the C∀ translator via a transformation into *generic types*. For each N, the first time an N-tuple is seen in a scope, a generic type with N type parameters is generated. For example, the following

```
[int, int] f() {
    [double, double] x;
    [int, double, int] y;
}
```

is transformed into

```
    forall( dtype T0, dtype T1 | sized(T0) | sized(T1) ) struct _tuple2 {
        T0 member_0;  T1 member_1;                    // generated before the first 2-tuple
    };
    _tuple2(int, int) f() {
    _tuple2(double, double) x;
    forall( dtype T0, dtype T1, dtype T2 | sized(T0) | sized(T1) | sized(T2) ) struct _tuple3 {
        T0 member_0;  T1 member_1;  T2 member_2;      // generated before the first 3-tuple
    };
    _tuple3(int, double, int) y;
}
```

Tuple expressions are then converted directly into compound literals, eg, [5, 'x', 1.24] becomes (_tuple3(**int**, **char**, **double**)){ 5, 'x', 1.24}.

5 | CONTROL STRUCTURES

C∀ identifies inconsistent, problematic, and missing control structures in C, as well as extends, modifies, and adds control structures to increase functionality and safety.

5.1 | if statement

The **if** expression allows declarations, similar to the **for** declaration expression.

```
if ( int x = f() ) ...                          // x != 0
if ( int x = f(), y = g() ) ...                 // x != 0 && y != 0
if ( int x = f(), y = g(); x < y ) ...          // relational expression
```

1097024x, 2018, 12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2624 by University Of Waterloo Dana Porter Library, Wiley Online Library on [22/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Unless a relational expression is specified, each variable is compared not equal to 0, which is the standard semantics for the **if** expression, and the results are combined using the logical && operator.* The scope of the declaration(s) is local to the **if** statement but exists within both the "then" and "else" clauses.

5.2 | switch statement

There are a number of deficiencies with the C **switch** statements: enumerating **case** lists, placement of **case** clauses, scope of the switch body, and fall through between case clauses.

C has no shorthand for specifying a list of case values, whether the list is noncontiguous or contiguous.† C∀ provides a shorthand for a noncontiguous list:

C∀	C
`case 2, 10, 34, 42:`	`case 2: case 10: case 34: case 42:`

for a contiguous list:‡

C∀	C
`case 2~42:`	`case 2: case 3: ... case 41: case 42:`

and a combination:

```
case -12~-4, -1~5, 14~21, 34~42:
```

C allows placement of **case** clauses *within* statements nested in the **switch** body (called Duff's device[16]);

```
switch ( i ) {
  case 0:
    for ( int i = 0; i < 10; i += 1 ) {
        ...
  case 1:       // no initialization of loop index
        ...
    }
}
```

C∀ precludes this form of transfer *into* a control structure because it causes an undefined behavior, especially with respect to missed initialization, and provides very limited functionality.

C allows placement of declaration within the **switch** body and unreachable code at the start, resulting in an undefined behavior.

```
switch ( x ) {
    int y = 1;                  // unreachable initialization
    x = 7;                      // unreachable code without label/branch
  case 0:
    ...
    int z = 0;                  // unreachable initialization, cannot appear after case
    z = 2;
  case 1:
    x = z;                      // without fall through, z is undefined
}
```

C∀ allows the declaration of local variables, eg, y, at the start of the **switch** with scope across the entire **switch** body, ie, all **case** clauses. C∀ disallows the declaration of local variable, eg, z, directly within the **switch** body, because a declaration cannot occur immediately after a **case** since a label can only be attached to a statement, and the use of z is undefined in **case**1 as neither storage allocation nor initialization may have occurred.

C **switch** provides multiple entry points into the statement body, but once an entry point is selected, control continues across *all* **case** clauses until the end of the **switch** body, called *fall through*; **case** clauses are made disjoint by the **break**

*C++ only provides a single declaration always compared not equal to 0.
†C provides this mechanism via fall through.
‡gcc has the same mechanism but awkward syntax, 2 ...42, as a space is required after a number; otherwise, the first period is a decimal point.

1097024x, 2018, 12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2624 by University Of Waterloo Dana Porter Library, Wiley Online Library on [22/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

C∀	C
```choose (day) {```	```switch (day) {```

```
choose ( day ) {
  case Mon~Thu:  // program

  case Fri:    // program
    wallet += pay;
    fallthrough;
  case Sat:   // party
    wallet -= party;

  case Sun:  // rest

  default:   // print error
}
```

```
switch ( day ) {
  case Mon: case Tue: case Wed: case Thu:  // program
    break;
  case Fri:    // program
    wallet += pay;

  case Sat:   // party
    wallet -= party;
    break;
  case Sun:  // rest
    break;
  default:   // print error
}
```

FIGURE 1 **choose** versus **switch** statements [Colour figure can be viewed at wileyonlinelibrary.com]

non-terminator

```
choose ( ... ) {
  case 3:
    if ( ... ) {
      ... fallthrough;  // goto case 4
    } else {
      ...
    }
    // implicit break
  case 4:
```

target label

```
choose ( ... ) {
  case 3:
    ... fallthrough common;
  case 4:
    ... fallthrough common;
  common: // below fallthrough at same level as case clauses
    ...   // common code for cases 3 and 4
    // implicit break
  case 4:
```

FIGURE 2 **fallthrough** statement [Colour figure can be viewed at wileyonlinelibrary.com]

statement. While fall through *is* a useful form of control flow, it does not match well with programmer intuition, resulting in errors from missing **break** statements. For backward compatibility, C∀ provides a *new* control structure, ie, **choose**, which mimics **switch**, but reverses the meaning of fall through (see Figure 1), similar to Go.

Finally, Figure 2 shows **fallthrough** may appear in contexts other than terminating a **case** clause and have an explicit transfer label allowing separate cases but common final code for a set of cases. The target label must be below the **fallthrough** and may not be nested in a control structure, ie, **fallthrough** cannot form a loop, and the target label must be at the same or higher level as the containing **case** clause and located at the same level as a **case** clause; the target label may be case **default**, but only associated with the current **switch**/**choose** statement.

5.3 | Labeled continue/break

While C provides **continue** and **break** statements for altering control flow, both are restricted to one level of nesting for a particular control structure. Unfortunately, this restriction forces programmers to use **goto** to achieve the equivalent control flow for more than one level of nesting. To prevent having to switch to the **goto**, C∀ extends **continue** and **break** with a target label to support a static multilevel exit,[17] as in Java. For both **continue** and **break**, the target label must be directly associated with a **for**, **while**, or **do** statement; for **break**, the target label can also be associated with a **switch**, **if** or compound ({}) statement. Figure 3 shows **continue** and **break** indicating the specific control structure and the corresponding C program using only **goto** and labels. The innermost loop has seven exit points, which cause a continuation or termination of one or more of the seven nested control structures.

With respect to safety, both labeled **continue** and **break** are **goto** restricted in the following ways.

- They cannot create a loop, which means only the looping constructs cause looping. This restriction means all situations resulting in repeated execution are clearly delineated.
- They cannot branch into a control structure. This restriction prevents missing declarations and/or initializations at the start of a control structure resulting in an undefined behavior.

The advantage of the labeled **continue**/**break** is allowing static multilevel exits without having to use the **goto** statement and tying control flow to the target control structure rather than an arbitrary point in a program. Furthermore, the location of the label at the *beginning* of the target control structure informs the reader (eye candy) that complex control flow is

1097024x, 2018, 12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2624 by University Of Waterloo Dana Porter Library, Wiley Online Library on [22/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

```
CA                                    C
LC: {                                 {
  ... declarations ...                  ... declarations ...
  LS: switch ( ... ) {                  switch ( ... ) {
    case 3:                               case 3:
      LIF: if ( ... ) {                     if ( ... ) {
        LF: for ( ... ) {                     for ( ... ) {
          ... break LC; ...                     ... goto LC; ...
          ... break LS; ...                     ... goto LS; ...    // terminate compound
          ... break LIF; ...                    ... goto LIF; ...   // terminate switch
          ... continue LF; ...                  ... goto LFC; ...   // terminate if
          ... break LF; ...                     ... goto LFB; ...   // continue loop
        } // for                              LFC: ; } LFB: ;       // terminate loop
      } else {                              } else {
        ... break LIF; ...                    ... goto LIF; ...
      } // if                               } LIF: ;
  } // switch                           } LS: ;                     // terminate if
} // compound                         } LC: ;
```

FIGURE 3 Multilevel exit [Colour figure can be viewed at wileyonlinelibrary.com]

```
Resumption                                          Termination
exception R { int fix; };                           exception T {};
void f() {                                          void f() {
    R r;
    ... resume( r ); ...                                ... throw( T{} ); ...
    ... r.fix // control returns here after handler     // control does NOT return here after handler
}                                                   }
try {                                               try {
    ... f(); ...                                        ... f(); ...
} catchResume( R r ) {                              } catch( T t ) {
    ... r.fix = ...; // return correction to raise      ... // recover and continue
} // dynamic return to _Resume                     } // static return to next statement
```

FIGURE 4 C∀ exception handling [Colour figure can be viewed at wileyonlinelibrary.com]

occurring in the body of the control structure. With **goto**, the label is at the end of the control structure, which fails to convey this important clue early enough to the reader. Finally, using an explicit target for the transfer instead of an implicit target allows new constructs to be added or removed without affecting the existing constructs. Otherwise, the implicit targets of the current **continue** and **break**, ie, the closest enclosing loop or **switch**, change as certain constructs are added or removed.

5.4 | Exception handling

The following framework for C∀ exception handling is in place, excluding some runtime type information and virtual functions. C∀ provides two forms of exception handling: *fix-up* and *recovery* (see Figure 4).[18,19] Both mechanisms provide a dynamic call to a handler using dynamic name lookup, where fix-up has dynamic return and recovery has static return from the handler. C∀ restricts exception types to those defined by aggregate type **exception**. The form of the raise dictates the set of handlers examined during propagation: *resumption propagation* (resume) only examines resumption handlers (**catchResume**); *terminating propagation* (**throw**) only examines termination handlers (**catch**). If resume or **throw** has no exception type, it is a reresume/rethrow, which means that the current exception continues propagation. If there is no current exception, the reresume/rethrow results in a runtime error.

The set of exception types in a list of catch clauses may include both a resumption and a termination handler.

```
try {
    ... resume( R{} ); ...
} catchResume( R r ) { ... throw( R{} ); ... }    // H1
  catch( R r ) { ... }                             // H2
```

1097024x, 2018, 12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2624 by University Of Waterloo Dana Porter Library, Wiley Online Library on [22/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

The resumption propagation raises R and the stack is not unwound; the exception is caught by the **catchResume** clause and handler H1 is invoked. The termination propagation in handler H1 raises R and the stack is unwound; the exception is caught by the **catch** clause and handler H2 is invoked. The termination handler is available because the resumption propagation did not unwind the stack.

An additional feature is conditional matching in a catch clause.

```
try {
    ... write( datafile, ... ); ...                  // may throw IOError
    ... write( logfile, ... ); ...
} catch ( IOError err; err.file == datafile ) { ... } // handle datafile error
  catch ( IOError err; err.file == logfile ) { ... }  // handle logfile error
  catch ( IOError err ) { ... }                       // handler error from other files
```

Here, the throw inserts the failing file handle into the I/O exception. Conditional catch cannot be trivially mimicked by other mechanisms because once an exception is caught, handler clauses in that **try** statement are no longer eligible.

The resumption raise can specify an alternate stack on which to raise an exception, called a *nonlocal raise*.

resume(*exception-type*, *alternate-stack*)

resume(*alternate-stack*)

These overloads of resume raise the specified exception or the currently propagating exception (reresume) at another C∀ coroutine or task.[4] Nonlocal raise is restricted to resumption to provide the exception handler the greatest flexibility because processing the exception does not unwind its stack, allowing it to continue after the handler returns.

To facilitate nonlocal raise, C∀ provides dynamic enabling and disabling of nonlocal exception propagation. The constructs for controlling propagation of nonlocal exceptions are the **enable** and **disable** blocks.

```
enable exception-type-list {        disable exception-type-list {
    // allow non-local raise            // disallow non-local raise
}                                   }
```

The arguments for **enable**/**disable** specify the exception types allowed to be propagated or postponed, respectively. Specifying no exception type is shorthand for specifying all exception types. Both **enable** and **disable** blocks can be nested; turning propagation on/off on entry and on exit, the specified exception types are restored to their prior state. Coroutines and tasks start with nonlocal exceptions disabled, allowing handlers to be put in place, before nonlocal exceptions are explicitly enabled.

```
void main( mytask & t ) {                  // thread starts here
    // non-local exceptions disabled
    try {                                  // establish handles for non-local exceptions
        enable {                           // allow non-local exception delivery
            // task body
        }
    // appropriate catchResume/catch handlers
    }
}
```

Finally, C∀ provides a Java-like **finally** clause after the catch clauses.

```
try {
    ... f(); ...
// catchResume or catch clauses
} finally {
    // house keeping
}
```

The **finally** clause is always executed, ie, if the try block ends normally or if an exception is raised. If an exception is raised and caught, the handler is run before the **finally** clause. Like a destructor (see Section 6.4), a **finally** clause can raise an exception but not if there is an exception being propagated. Mimicking the **finally** clause with mechanisms like Resource Aquisition Is Initialization (RAII) is nontrivial when there are multiple types and local accesses.

1097024x, 2018, 12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2624 by University Of Waterloo Dana Porter Library, Wiley Online Library on [22/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

5.5 | with statement

Heterogeneous data are often aggregated into a structure/union. To reduce syntactic noise, C∀ provides a **with** statement (see section 4.F in the Pascal User Manual and Report[20]) to elide aggregate member qualification by opening a scope containing the member identifiers.

```
struct S { char c; int i; double d; };
struct T { double m, n; };
// multiple aggregate parameters
void f( S & s, T & t ) {          | void f( S & s, T & t ) with ( s, t ) {
    s.c; s.i; s.d;                |     c; i; d;        // no qualification
    t.m; t.n;                     |     m; n;
}                                 | }
```

Object-oriented programming languages only provide implicit qualification for the receiver.

In detail, the **with** statement has the form

with-statement:
 'with' '(' *expression-list* ')' *compound-statement*

and may appear as the body of a function or nested within a function body. Each expression in the expression list provides a type and object. The type must be an aggregate type. (Enumerations are already opened.) The object is the implicit qualifier for the open structure members.

All expressions in the expression list are open in parallel within the compound statement, which is different from Pascal, which nests the openings from left to right. The difference between parallel and nesting occurs for members with the same name and type.

```
struct S { int i; int j; double m; } s, w;    // member i has same type in structure types S and T
struct T { int i; int k; int m; } t, w;
with ( s, t ) {                                // open structure variables s and t in parallel
    j + k;                                     // unambiguous, s.j + t.k
    m = 5.0;                                   // unambiguous, s.m = 5.0
    m = 1;                                     // unambiguous, t.m = 1
    int a = m;                                 // unambiguous, a = t.m
    double b = m;                              // unambiguous, b = s.m
    int c = s.i + t.i;                         // unambiguous, qualification
    (double)m;                                 // unambiguous, cast s.m
}
```

For parallel semantics, both s.i and t.i are visible and, therefore, i is ambiguous without qualification; for nested semantics, t.i hides s.i and, therefore, i implies t.i. C∀'s ability to overload variables means members with the same name but different types are automatically disambiguated, eliminating most qualification when opening multiple aggregates. Qualification or a cast is used to disambiguate.

There is an interesting problem between parameters and the function body **with**.

```
void ?{}( S & s, int i ) with ( s ) {          // constructor
    s.i = i;  j = 3;  m = 5.5;                 // initialize members
}
```

Here, the assignment s.i = i means s.i = s.i, which is meaningless, and there is no mechanism to qualify the parameter i, making the assignment impossible using the function body **with**. To solve this problem, parameters are treated like an initialized aggregate

```
struct Params {
    S & s;
    int i;
} params;
```

1097024x, 2018, 12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2624 by University Of Waterloo Dana Porter Library, Wiley Online Library on [22/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

and implicitly opened *after* a function body open, to give them higher priority

```
void ?{}( S & s, int i ) with ( s ) { with( params ) {
    s.i = i; j = 3; m = 5.5;
} }
```

Finally, a cast may be used to disambiguate among overload variables in a **with** expression

```
with ( w ) { ... }                  // ambiguous, same name and no context
with ( (S)w ) { ... }               // unambiguous, cast
```

and **with** expressions may be complex expressions with type reference (see Section 6.2) to aggregate

```
struct S { int i, j; } sv;
with ( sv ) {                       // implicit reference
    S & sr = sv;
    with ( sr ) {                   // explicit reference
        S * sp = &sv;
        with ( *sp ) {              // computed reference
            i = 3; j = 4;           // sp->i, sp->j
        }
        i = 2; j = 3;               // sr.i, sr.j
    }
    i = 1; j = 2;                   // sv.i, sv.j
}
```

Collectively, these control-structure enhancements reduce programmer burden and increase readability and safety.

6 | DECLARATIONS

Declarations in C have weaknesses and omissions. C∀ attempts to correct and add to C declarations, while ensuring C∀ subjectively "feels like" C. An important part of this subjective feel is maintaining C's syntax and procedural paradigm, as opposed to functional and object-oriented approaches in other systems languages such as C++ and Rust. Maintaining the C approach means that C coding patterns remain not only useable but idiomatic in C∀, reducing the mental burden of retraining C programmers and switching between C and C∀ development. Nevertheless, some features from other approaches are undeniably convenient; C∀ attempts to adapt these features to the C paradigm.

6.1 | Alternative declaration syntax

C declaration syntax is notoriously confusing and error prone. For example, many C programmers are confused by a declaration as simple as the following.

```
int * x[5]
```

x | | | | | | → 0 1 2 3 4   x | → 0 | 1 | 2 | 3 | 4

Is this an array of five pointers to integers or a pointer to an array of five integers? If there is any doubt, it implies productivity and safety issues even for basic programs. Another example of confusion results from the fact that a function name and its parameters are embedded within the return type, mimicking the way the return value is used at the function's call site. For example, a function returning a pointer to an array of integers is defined and used in the following way.

```
int (*f())[5] {...};                // definition
... (*f())[3] += 1;                 // usage
```

Essentially, the return type is wrapped around the function name in successive layers (like an onion). While attempting to make the two contexts consistent is a laudable goal, it has not worked out in practice.

1097024x, 2018, 12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2624 by University Of Waterloo Dana Porter Library, Wiley Online Library on [22/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

C∀ provides its own type, variable, and function declarations, using a different syntax.[21(pp856-859)] The new declarations place qualifiers to the left of the base type, whereas C declarations place qualifiers to the right. The qualifiers have the same meaning but are ordered left to right to specify a variable's type.

```
C∀                               C
[5] * int x1;                    int * x1 [5];                 // array of 5 pointers to int
* [5] int x2;                    int (*x2)[5];                 // pointer to array of 5 int
[* [5] int] f( int p );          int (*f( int p ))[5];         // function returning pointer to array of 5 int and taking int
```

The only exception is bit-field specification, which always appears to the right of the base type. However, unlike C, C∀ type declaration tokens are distributed across all variables in the declaration list. For instance, variables **x** and **y** of type pointer to integer are defined in C∀ as

```
C∀                 C
* int x, y;        int *x, *y, z;
int z;
```

The separation of regular and pointer declarations by C∀ declarations enforces greater clarity with only slightly more syntax.

All specifiers (**extern**, **static**, etc) and qualifiers (**const**, **volatile**, etc) are used in the normal way with the new declarations and also appear left to right.

```
C∀                                   C
extern const * const int x;          int extern const * const x;       // external const pointer to const int
static const * [5] const int y;      static const int (* const y)[5]   // internal const pointer to array of 5 const int
```

Specifiers must appear at the start of a C∀ function declaration.[§]

The new declaration syntax can be used in other contexts where types are required, eg, casts and the pseudo-function **sizeof**.

```
C∀                          C
y = (* int)x;               y = (int *)x;
i = sizeof([ 5 ] * int);    i = sizeof(int * [ 5 ]);
```

The syntax of the new function-prototype declaration follows directly from the new function-definition syntax; also, parameter names are optional.

```
[ int x ] f ( /* void */ );        // returning int with no parameters
[ int x ] f (...);                 // returning int with unknown parameters
[ * int ] g ( int y );             // returning pointer to int with int parameter
[ void ] h ( int, char );          // returning no result with int and char parameters
[ * int, int ] j ( int );          // returning pointer to int and int with int parameter
```

This syntax allows a prototype declaration to be created by cutting and pasting the source text from the function-definition header (or vice versa). Like C, it is possible to declare multiple function prototypes in a single declaration, where the return type is distributed across *all* function names in the declaration list.

```
C∀                                                C
[double] foo(), foo( int ), foo( double ) {...}   double foo1( void ), foo2( int ), foo3( double );
```

Here, C∀ allows the last function in the list to define its body.

The syntax for pointers to C∀ functions specifies the pointer name on the right.

```
* [ int x ] () fp;               // pointer to function returning int with no parameters
* [ * int ] ( int y ) gp;        // pointer to function returning pointer to int with int parameter
* [ ] ( int, char ) hp;          // pointer to function returning no result with int and char parameters
* [ * int, int ] ( int ) jp;     // pointer to function returning pointer to int and int with int parameter
```

[§] The placement of a storage-class specifier other than at the beginning of the declaration specifiers in a declaration is an obsolescent feature (see section 6.11.5(1) in ISO/IEC 9899[7]).

1097024x, 2018, 12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2624 by University Of Waterloo Dana Porter Library, Wiley Online Library on [22/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Note that the name of the function pointer is specified last, as for other variable declarations.

Finally, new C∀ declarations may appear together with C declarations in the same program block but cannot be mixed within a specific declaration. Therefore, a programmer has the option of either continuing to use traditional C declarations or taking advantage of the new style. Clearly, both styles need to be supported for some time due to existing C-style header files, particularly for UNIX-like systems.

6.2 | References

All variables in C have an *address*, a *value*, and a *type*; at the position in the program's memory denoted by the address, there exists a sequence of bits (the value), with the length and semantic meaning of this bit sequence defined by the type. The C type system does not always track the relationship between a value and its address; a value that does not have a corresponding address is called an *rvalue* (for "right-hand value"), whereas a value that does have an address is called an *lvalue* (for "left-hand value"). For example, in **int** x; x = 42, the variable expression x on the left-hand side of the assignment is an lvalue, whereas the constant expression 42 on the right-hand side of the assignment is an rvalue. Despite the nomenclature of "left-hand" and "right-hand," an expression's classification as an lvalue or an rvalue is entirely dependent on whether it has an address or not; in imperative programming, the address of a value is used for both reading and writing (mutating) a value, and as such, lvalues can be converted into rvalues and read from, but rvalues cannot be mutated because they lack a location to store the updated value.

Within a lexical scope, lvalue expressions have an *address interpretation* for writing a value or a *value interpretation* to read a value. For example, in x = y, x has an address interpretation, whereas y has a value interpretation. While this duality of interpretation is useful, C lacks a direct mechanism to pass lvalues between contexts, instead relying on *pointer types* to serve a similar purpose. In C, for any type T, there is a pointer type T *, the value of which is the address of a value of type T. A pointer rvalue can be explicitly *dereferenced* to the pointed-to lvalue with the dereference operator *?, whereas the rvalue representing the address of an lvalue can be obtained with the address-of operator &?.

```
int x = 1, y = 2, * p1, * p2, ** p3;
p1 = &x;                                    // p1 points to x
p2 = &y;                                    // p2 points to y
p3 = &p1;                                   // p3 points to p1
*p2 = ((*p1 + *p2) * (**p3 - *p1)) / (**p3 - 15);
```

Unfortunately, the dereference and address-of operators introduce a great deal of syntactic noise when dealing with pointed-to values rather than pointers, as well as the potential for subtle bugs because of pointer arithmetic. For both brevity and clarity, it is desirable for the compiler to figure out how to elide the dereference operators in a complex expression such as the assignment to *p2 above. However, since C defines a number of forms of *pointer arithmetic*, two similar expressions involving pointers to arithmetic types (eg, *p1 + x and p1 + x) may each have well-defined but distinct semantics, introducing the possibility that a programmer may write one when they mean the other and precluding any simple algorithm for elision of dereference operators. To solve these problems, C∀ introduces reference types T &; a T & has exactly the same value as a T *, but where the T * takes the address interpretation by default, a T & takes the value interpretation by default, as below.

```
int x = 1, y = 2, & r1, & r2, && r3;
&r1 = &x;                                   // r1 points to x
&r2 = &y;                                   // r2 points to y
&&r3 = &&r1;                                // r3 points to r2
r2 = ((r1 + r2) * (r3 - r1)) / (r3 - 15);   // implicit dereferencing
```

Except for auto-dereferencing by the compiler, this reference example is exactly the same as the previous pointer example. Hence, a reference behaves like a variable name—an lvalue expression that is interpreted as a value—but also has the type system track the address of that value. One way to conceptualize a reference is via a rewrite rule, where the compiler inserts a dereference operator before the reference variable for each reference qualifier in the reference variable declaration; thus, the previous example implicitly acts as in the following.

```
*r2 = ((*r1 + *r2) * (**r3 - *r1)) / (**r3 - 15);
```

References in C∀ are similar to those in C++, with important improvements, which can be seen in the example above. Firstly, C∀ does not forbid references to references. This provides a much more orthogonal design for library

1097024x, 2018, 12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2624 by University Of Waterloo Dana Porter Library, Wiley Online Library on [22/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

implementors, obviating the need for workarounds such as std::reference_wrapper. Secondly, C∀ references are rebindable, whereas C++ references have a fixed address. Rebinding allows C∀ references to be default initialized (eg, to a null pointer[¶]) and point to different addresses throughout their lifetime, like pointers. Rebinding is accomplished by extending the existing syntax and semantics of the address-of operator in C.

In C, the address of an lvalue is always an rvalue, as, in general, that address is not stored anywhere in memory and does not itself have an address. In C∀, the address of a T & is an lvalue T *, as the address of the underlying T is stored in the reference and can thus be mutated there. The result of this rule is that any reference can be rebound using the existing pointer assignment semantics by assigning a compatible pointer into the address of the reference, eg, &r1 = &x; above. This rebinding occurs to an arbitrary depth of reference nesting; loosely speaking, nested address-of operators produce a nested lvalue pointer up to the depth of the reference. These explicit address-of operators can be thought of as "cancelling out" the implicit dereference operators, eg, (&*)r1 = &x or (&(&*)*)r3 = &(&*)r1 or even (&*)r2 = (&*)*r3 for &r2 = &r3. More precisely, we have the following.

- If R is an rvalue of type T $\&_1 \cdots \&_r$, where $r \geq 1$ references (& symbols), then &R has type T*$\&@_2 \cdots \&_r$, ie, T pointer with $r - 1$ references (& symbols).
- If L is an lvalue of type T $\&_1 \cdots \&_l$, where $l \geq 0$ references (& symbols), then &L has type T*$\&_1 \cdots \&_l$, ie, T pointer with l references (& symbols).

Since pointers and references share the same internal representation, the code using either is equally performant; in fact, the C∀ compiler converts references into pointers internally, and the choice between them is made solely on convenience, eg, many pointer or value accesses.

By analogy to pointers, C∀ references also allow cv-qualifiers such as **const**.

```
const int cx = 5;                     // cannot change cx
const int & cr = cx;                  // cannot change cr's referred value
&cr = &cx;                            // rebinding cr allowed
cr = 7;                               // ERROR, cannot change cr
int & const rc = x;                   // must be initialized, like in C++
&rc = &x;                             // ERROR, cannot rebind rc
rc = 7;                               // x now equal to 7
```

Given that a reference is meant to represent an lvalue, C∀ provides some syntactic shortcuts when initializing references. There are three initialization contexts in C∀: declaration initialization, argument/parameter binding, and return/temporary binding. In each of these contexts, the address-of operator on the target lvalue is elided. The syntactic motivation is clearest when considering overloaded operator assignment, eg, **int** ?+=?(**int**&,**int**); given **int** x,y, the expected call syntax is x += y, not &x += y.

More generally, this initialization of references from lvalues rather than pointers is an instance of an "lvalue-to-reference" conversion rather than an elision of the address-of operator; this conversion is used in any context in C∀ where an implicit conversion is allowed. Similarly, use of the value pointed to by a reference in an rvalue context can be thought of as a "reference-to-rvalue" conversion, and C∀ also includes a qualifier-adding "reference-to-reference" conversion, analogous to the T * to **const** T * conversion in standard C. The final reference conversion included in C∀ is an "rvalue-to-reference" conversion, implemented by means of an implicit temporary. When an rvalue is used to initialize a reference, it is instead used to initialize a hidden temporary value with the same lexical scope as the reference, and the reference is initialized to the address of this temporary.

```
struct S { double x, y; };
int x, y;
void f( int & i, int & j, S & s, int v[] );
f( 3, x + y, (S){ 1.0, 7.0 }, (int [3]){ 1, 2, 3 } );    // pass rvalue to lvalue ⇒ implicit temporary
```

This allows complex values to be succinctly and efficiently passed to functions, without the syntactic overhead of the explicit definition of a temporary variable or the runtime cost of pass-by-value. C++ allows a similar binding, but only for **const** references; the more general semantics of C∀ are an attempt to avoid the *const poisoning* problem,[22] in which the addition of a **const** qualifier to one reference requires a cascading chain of added qualifiers.

[¶]While effort has been made into non-null reference checking in C++ and Java, the exercise seems moot for any nonmanaged languages (C/C++), given that it only handles one of many different error situations, eg, using a pointer after its storage is deleted.

1097024x, 2018, 12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2624 by University Of Waterloo Dana Porter Library, Wiley Online Library on [22/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

C Type Nesting

```
struct S {
    enum C { R, G, B };
    struct T {
        union U { int i, j; };
        enum C c;
        short int i, j;
    };
    struct T t;
} s;

int rtn() {
    s.t.c = R;
    struct T t = { R, 1, 2 };
    enum C c;
    union U u;
}
```

C Implicit Hoisting

```
enum C { R, G, B };
union U { int i, j; };
struct T {
    enum C c;
    short int i, j;
};
struct S {
    struct T t;
} s;
```

C∀

```
struct S {
    enum C { R, G, B };
    struct T {
        union U { int i, j; };
        enum C c;
        short int i, j;
    };
    struct T t;
} s;

int rtn() {
    s.t.c = S.R;   // type qualification
    struct S.T t = { S.R, 1, 2 };
    enum S.C c;
    union S.T.U u;
}
```

FIGURE 5 Type nesting/qualification [Colour figure can be viewed at wileyonlinelibrary.com]

6.3 | Type nesting

Nested types provide a mechanism to organize associated types and refactor a subset of members into a named aggregate (eg, subaggregates name, address, department, within aggregate employe). Java nested types are dynamic (apply to objects), C++ are static (apply to the **class**), and C hoists (refactors) nested types into the enclosing scope, which means there is no need for type qualification. Since C∀ in not object oriented, adopting dynamic scoping does not make sense; instead, C∀ adopts C++ static nesting, using the member-selection operator "." for type qualification, as does Java, rather than the C++ type-selection operator "::" (see Figure 5). In the C left example, types C, U, and T are implicitly hoisted outside of type S into the containing block scope. In the C∀ right example, the types are not hoisted and accessible.

6.4 | Constructors and destructors

One of the strengths (and weaknesses) of C is memory-management control, allowing resource release to be precisely specified versus unknown release with garbage-collected memory management. However, this manual approach is verbose, and it is useful to manage resources other than memory (eg, file handles) using the same mechanism as memory. C++ addresses these issues using RAII, implemented by means of *constructor* and *destructor* functions; C∀ adopts constructors and destructors (and **finally**) to facilitate RAII. While constructors and destructors are a common feature of object-oriented programming languages, they are an independent capability allowing C∀ to adopt them while retaining a procedural paradigm. Specifically, C∀ constructors and destructors are denoted by name and first parameter type versus name and nesting in an aggregate type. Constructor calls seamlessly integrate with existing C initialization syntax, providing a simple and familiar syntax to C programmers and allowing constructor calls to be inserted into legacy C code with minimal code changes.

In C∀, a constructor is named ?{} and a destructor is named ^?{}.[#] The name {} comes from the syntax for the initializer: **struct** S { **int** i, j; } s = { 2, 3 }. Like other C∀ operators, these names represent the syntax used to explicitly call the constructor or destructor, eg, s{...} or ^s{...}. The constructor and the destructor have return type **void**, and the first parameter is a reference to the object type to be constructed or destructed. While the first parameter is informally called the this parameter, as in object-oriented languages, any variable name may be used. Both constructors and destructors allow additional parameters after the this parameter for specifying values for initialization/deinitialization.[‖]

```
struct VLA { int size, * data; };                                     // variable length array of integers
void ?{}( VLA & vla ) with ( vla ) { size = 10; data = alloc( size ); }  // default constructor
void ^?{}( VLA & vla ) with ( vla ) { free( data ); }                  // destructor
{
    VLA x;                                                             // implicit: x{};
}                                                                      // implicit: ^x{};
```

[#] The symbol ^ is used for the destructor name because it was the last binary operator that could be used in a unary context.
[‖] Destruction parameters are useful for specifying storage-management actions, such as deinitialize but not deallocate.

1097024x, 2018, 12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2624 by University Of Waterloo Dana Porter Library, Wiley Online Library on [22/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

VLA is a *managed type*[**]: a type requiring a nontrivial constructor or destructor, or with a member of a managed type. A managed type is implicitly constructed at allocation and destructed at deallocation to ensure proper interaction with runtime resources, in this case, the data array in the heap. For details of the code-generation placement of implicit constructor and destructor calls among complex executable statements, see section 2.2 in the work of Schluntz.[23]

C∀ also provides syntax for *initialization* and *copy*.

```
void ?{}( VLA & vla, int size, char fill = '\0' ) {        // initialization
    vla.[ size, data ] = [ size, alloc( size, fill ) ];
}
void ?{}( VLA & vla, VLA other ) {                         // copy, shallow
    vla = other;
}
```

(Note that the example is purposely simplified using shallow-copy semantics.) An initialization constructor call has the same syntax as a C initializer, except that the initialization values are passed as arguments to a matching constructor (number and type of parameters).

```
VLA va = { 20, 0 },  * arr = alloc(){ 5, 0 };
```

Note of the use of a *constructor expression* to initialize the storage from the dynamic storage allocation. Like C++, the copy constructor has two parameters, the second of which is a value parameter with the same type as the first parameter; appropriate care is taken to not recursively call the copy constructor when initializing the second parameter.

C∀ constructors may be explicitly called, like Java, and destructors may be explicitly called, like C++. Explicit calls to constructors double as a C++-style *placement syntax*, useful for construction of members in user-defined constructors and reuse of existing storage allocations. Like the other operators in C∀, there is a concise syntax for constructor/destructor function calls.

```
{
    VLA  x,         y = { 20, 0x01 },    z = y;     // z points to y
    //   x{};       y{ 20, 0x01 };       z{ z, y };
    ^x{};                                           // deallocate x
    x{};                                            // reallocate x
    z{ 5, 0xff };                                   // reallocate z, not pointing to y
    ^y{};                                           // deallocate y
    y{ x };                                         // reallocate y, points to x
    x{};                                            // reallocate x, not pointing to y
}   // ^z{}; ^y{}; ^x{};
```

To provide a uniform type interface for **otype** polymorphism, the C∀ compiler automatically generates a default constructor, copy constructor, assignment operator, and destructor for all types. These default functions can be overridden by user-generated versions. For compatibility with the standard behavior of C, the default constructor and destructor for all basic, pointer, and reference types do nothing, whereas the copy constructor and assignment operator are bitwise copies; if default zero initialization is desired, the default constructors can be overridden. For user-generated types, the four functions are also automatically generated. **enum** types are handled the same as their underlying integral type, and unions are also bitwise copied and no-op initialized and destructed. For compatibility with C, a copy constructor from the first union member type is also defined. For **struct** types, each of the four functions is implicitly defined to call their corresponding functions on each member of the struct. To better simulate the behavior of C initializers, a set of *member constructors* is also generated for structures. A constructor is generated for each nonempty prefix of a structure's member list to copy-construct the members passed as parameters and default-construct the remaining members. To allow users to limit the set of constructors available for a type, when a user declares any constructor or destructor, the corresponding generated function and all member constructors for that type are hidden from expression resolution; similarly, the generated default constructor is hidden upon the declaration of any constructor. These semantics closely mirror the rule for implicit declaration of constructors in C++.[24(p186)]

In some circumstance, programmers may not wish to have implicit constructor and destructor generation and calls. In these cases, C∀ provides the initialization syntax `S x @= {}`, and the object becomes unmanaged; hence, implicit

[**]A managed type affects the runtime environment versus a self-contained type.

1097024x, 2018, 12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2624 by University Of Waterloo Dana Porter Library, Wiley Online Library on [22/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

constructor and destructor calls are not generated. Any C initializer can be the right-hand side of an @= initializer, eg, VLA a @= { 0, 0×0 }, with the usual C initialization semantics. The same syntax can be used in a compound literal, eg, a = (VLA)@{ 0, 0×0 }, to create a C-style literal. The point of @= is to provide a migration path from legacy C code to C∀, by providing a mechanism to incrementally convert into implicit initialization.

7 | LITERALS

C already includes limited polymorphism for literals–0 can be either an integer or a pointer literal, depending on context, whereas the syntactic forms of literals of the various integer and float types are very similar, differing from each other only in suffix. In keeping with the general C∀ approach of adding features while respecting the "C style" of doing things, C's polymorphic constants and typed literal syntax are extended to interoperate with user-defined types, while maintaining a backward-compatible semantics.

A simple example is allowing the underscore, as in Ada, to separate prefixes, digits, and suffixes in all C∀ constants, eg, 0x_1.ffff_ffff_p_128_l, where the underscore is also the standard separator in C identifiers. C++ uses a single quote as a separator, but it is restricted among digits, precluding its use in the literal prefix or suffix, eg, 0×1.ffff'ffffp128l, and causes problems with most integrated development environments (IDEs), which must be extended to deal with this alternate use of the single quote.

7.1 | 0/1

In C, 0 has the special property that it is the only "false" value; by the standard, any value that compares equal to 0 is false, whereas any value that compares unequal to 0 is true. As such, an expression x in any Boolean context (such as the condition of an **if** or **while** statement, or the arguments to &&, ||, or ?:) can be rewritten as x != 0 without changing its semantics. Operator overloading in C∀ provides a natural means to implement this truth-value comparison for arbitrary types, but the C type system is not precise enough to distinguish an equality comparison with 0 from an equality comparison with an arbitrary integer or pointer. To provide this precision, C∀ introduces a new type **zero_t** as the type of literal 0 (somewhat analagous to nullptr_t and nullptr in C++11); **zero_t** can only take the value 0, but has implicit conversions to the integer and pointer types so that C code involving 0 continues to work. With this addition, C∀ rewrites **if** (x) and similar expressions to **if** ((x) != 0) or the appropriate analogue, and any type T is "truthy" by defining an operator overload **int** ?!=?(T, **zero_t**). C++ makes types truthy by adding a conversion to bool; prior to the addition of explicit cast operators in C++11, this approach had the pitfall of making truthy types transitively convertible into any numeric type; C∀ avoids this issue.

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

7.2 | User literals

For readability, it is useful to associate units to scale literals, eg, weight (stone, pound, kilogram) or time (seconds, minutes, hours). The left of Figure 6 shows the C∀ alternative call syntax (postfix: literal argument before function name), using the backquote, to convert basic literals into user literals. The backquote is a small character, making the unit (function name) predominate. For examples, the multiprecision integer type in Section 8.5 has the following user literals.

```
y = 9223372036854775807L`mp * 18446744073709551615UL`mp;
y = "12345678901234567890123456789"`mp + "12345678901234567890123456789"`mp;
```

Because C∀ uses a standard function, all types and literals are applicable, as well as overloading and conversions, where ?` denotes a postfix-function name and ` denotes a postfix-function call.

1097024x, 2018, 12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2624 by University Of Waterloo Dana Porter Library, Wiley Online Library on [22/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

C∀

```
struct W {
    double stones;
};
void ?{}( W & w ) { w.stones = 0; }
void ?{}( W & w, double w ) { w.stones = w; }
W ?+?( W l, W r ) {
    return (W){ l.stones + r.stones };
}
W ?`st(double w) { return (W){ w }; }
W ?`lb(double w) { return (W){ w/14.0 }; }
W ?`kg(double w) { return (W){ w*0.16 }; }



int main() {
    W w, heavy = { 20 };
    w = 155`lb;
    w = 0b_1111`st;
    w = 0_233`lb;
    w = 0x_9b_u`kg;
    w = 5.5`st + 8`kg + 25.01`lb + heavy;
}
```

C++

```
struct W {
    double stones;
    W() { stones = 0.0; }
    W( double w ) { stones = w; }
};
W operator+( W l, W r ) {
    return W( l.stones + r.stones );
}
W operator""_st(unsigned long long int w) {return W(w); }
W operator""_lb(unsigned long long int w) {return W(w/14.0); }
W operator""_kg(unsigned long long int w) {return W(w*0.16); }
W operator""_st(long double w ) { return W( w ); }
W operator""_lb(long double w ) { return W( w / 14.0 ); }
W operator""_kg(long double w ) { return W( w * 0.16 ); }
int main() {
    W w, heavy = { 20 };
    w = 155_lb;
    // binary unsupported
    w = 0'233_lb;          // quote separator
    w = 0x9b_kg;
    w = 5.5d_st + 8_kg + 25.01_lb + heavy;
}
```

FIGURE 6 User literal [Colour figure can be viewed at wileyonlinelibrary.com]

postfix function	constant	variable/expression	postfix pointer
`int ?`h(int s);`	`0 `h;`	`int i = 7;`	`int (* ?`p)(int i);`
`int ?`h(double s);`	`3.5`h;`	`` i`h; ``	`` ?`p = ?`h; ``
`int ?`m(char c);`	`'1'`m;`	`(i + 3)`h;`	`` 3`p; ``
`int ?`m(const char * s);`	`"123" "456"`m;`	`(i + 3.5)`h;`	`` i`p; ``
`int ?`t(int a, int b, int c);`	`[1,2,3]`t;`		`(i + 3)`p;`

The right of Figure 6 shows the equivalent C++ version using the underscore for the call syntax. However, C++ restricts the types, eg, **unsigned long long int** and **long double** to represent integral and floating literals. After which, user literals must match (no conversions); hence, it is necessary to overload the unit with all appropriate types.

8 | LIBRARIES

As stated in Section 2.2, C∀ inherits a large corpus of library code, where other programming languages must rewrite or provide fragile interlanguage communication with C. C∀ has replacement libraries condensing hundreds of existing C names into tens of C∀ overloaded names, all without rewriting the actual computations. In many cases, the interface is an inline wrapper providing overloading during compilation but of zero cost at runtime. The following sections give a glimpse of the interface reduction to many C libraries. In many cases, **signed**/**unsigned char**, **short**, and **_Complex** functions are available (but not shown) to ensure that expression computations remain in a single type, as conversions can distort results.

8.1 | Limits

C library limits.h provides lower and upper bound constants for the basic types. C∀ name overloading is used to condense these typed constants.

Definition	Usage
`const short int MIN = -32768;`	`short int si = MIN;`
`const int MIN = -2147483648;`	`int i = MIN;`
`const long int MIN = -9223372036854775808L;`	`long int li = MIN;`

The result is a significant reduction in names to access typed constants.

1097024x, 2018, 12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2624 by University Of Waterloo Dana Porter Library, Wiley Online Library on [22/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

C∀	C
MIN	CHAR_MIN, SHRT_MIN, INT_MIN, LONG_MIN, LLONG_MIN, FLT_MIN, DBL_MIN, LDBL_MIN
MAX	UCHAR_MAX, SHRT_MAX, INT_MAX, LONG_MAX, LLONG_MAX, FLT_MAX, DBL_MAX, LDBL_MAX
PI	M_PI, M_PIl
E	M_E, M_El

8.2 | Math

C library math.h provides many mathematical functions. C∀ function overloading is used to condense these mathematical functions.

Definition	Usage
`float log( float x );`	`float f = log( 3.5 );`
`double log( double );`	`double d = log( 3.5 );`
`double _Complex log( double _Complex x );`	`double _Complex dc = log( 3.5+0.5I );`

The result is a significant reduction in names to access math functions.

C∀	C
log	logf, log, logl, clogf, clog, clogl
sqrt	sqrtf, sqrt, sqrtl, csqrtf, csqrt, csqrtl
sin	sinf, sin, sinl, csinf, csin, csinl

While C11 has type-generic math (see section 7.25 of the ISO/IEC 9899[7]) in tgmath.h to provide a similar mechanism, these macros are limited, matching a function name with a single set of floating type(s). For example, it is impossible to overload atan for both one and two arguments; instead, the names atan and atan2 are required (see Section 2.1). The key observation is that only a restricted set of type-generic macros is provided for a limited set of function names, which do not generalize across the type system, as in C∀.

8.3 | Standard

C library stdlib.h provides many general functions. C∀ function overloading is used to condense these utility functions.

Definition	Usage
`unsigned int abs( int );`	`unsigned int i = abs( -1 );`
`double abs( double );`	`double d = abs( -1.5 );`
`double abs( double _Complex );`	`double d = abs( -1.5+0.5I );`

The result is a significant reduction in names to access utility functions.

C∀	C
abs	abs, labs, llabs, fabsf, fabs, fabsl, cabsf, cabs, cabsl
strto	strtol, strtoul, strtoll, strtoull, strtof, strtod, strtold
random	srand48, mrand48, lrand48, drand48

In addition, there are polymorphic functions, like min and max, that work on any type with operator ?<? or ?>?.

The following shows one example where C∀ *extends* an existing standard C interface to reduce complexity and provide safety. C/C11 provide a number of complex and overlapping storage-management operations to support the following capabilities.

fill an allocation with a specified character.

resize an existing allocation to decrease or increase its size. In either case, new storage may or may not be allocated, and if there is a new allocation, as much data from the existing allocation are copied. For an increase in storage size, new storage after the copied data may be filled.

1097024x, 2018, 12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2624 by University Of Waterloo Dana Porter Library, Wiley Online Library on [22/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 1 Storage-management operations

		fill	resize	align	array
C	malloc	no	no	no	no
	calloc	yes (0 only)	no	no	yes
	realloc	no/copy	yes	no	no
	memalign	no	no	yes	no
	posix_memalign	no	no	yes	no
C11	aligned_alloc	no	no	yes	no
C∀	alloc	yes/copy	no/yes	no	yes
	align_alloc	yes	no	yes	yes

```
size_t  dim = 10;                              // array dimension
char fill = '\xff';                            // initialization fill value
int * ip;
```

C∀	C
`ip = alloc();`	`ip = (int *)malloc( sizeof(int) );`
`ip = alloc( fill );`	`ip = (int *)malloc( sizeof(int) ); memset( ip, fill, sizeof(int) );`
`ip = alloc( dim );`	`ip = (int *)malloc( dim * sizeof(int) );`
`ip = alloc( dim, fill );`	`ip = (int *)malloc( sizeof(int) ); memset( ip, fill, dim * sizeof(int) );`
`ip = alloc( ip, 2 * dim );`	`ip = (int *)realloc( ip, 2 * dim * sizeof(int) );`
`ip = alloc( ip, 4 * dim, fill );`	`ip = (int *)realloc( ip, 4 * dim * sizeof(int) ); memset( ip, fill, 4 * dim * sizeof(int));`
`ip = align_alloc( 16 );`	`ip = memalign( 16, sizeof(int) );`
`ip = align_alloc( 16, fill );`	`ip = memalign( 16, sizeof(int) ); memset( ip, fill, sizeof(int) );`
`ip = align_alloc( 16, dim );`	`ip = memalign( 16, dim * sizeof(int) );`
`ip = align_alloc( 16, dim, fill );`	`ip = memalign( 16, dim * sizeof(int) ); memset( ip, fill, dim * sizeof(int) );`

FIGURE 7 C∀ versus C storage allocation

align an allocation on a specified memory boundary, eg, an address multiple of 64 or 128 for cache-line purposes.
array allocation with a specified number of elements. An array may be filled, resized, or aligned.

Table 1 shows the capabilities provided by C/C11 allocation functions and how all the capabilities can be combined into two C∀ functions. C∀ storage-management functions extend the C equivalents by overloading, providing shallow type safety, and removing the need to specify the base allocation size. Figure 7 contrasts C∀ and C storage allocation performing the same operations with the same type safety.

Variadic new (see Section 4.7) cannot support the same overloading because extra parameters are for initialization. Hence, there are new and anew functions for single and array variables, and the fill value is the arguments to the constructor.

```
struct S { int i, j; };
void ?{}( S & s, int i, int j ) { s.i = i; s.j = j; }
S * s = new( 2, 3 );                   // allocate storage and run constructor
S * as = anew( dim, 2, 3 );            // each array element initialized to 2, 3
```

Note that C++ can only initialize array elements via the default constructor.

Finally, the C∀ memory allocator has *sticky properties* for dynamic storage: fill and alignment are remembered with an object's storage in the heap. When a realloc is performed, the sticky properties are respected, so that new storage is correctly aligned and initialized with the fill character.

8.4 | I/O

The goal of C∀ I/O is to simplify the common cases, while fully supporting polymorphism and user-defined types in a consistent way. The approach combines ideas from C++ and Python. The C∀ header file for the I/O library is fstream.

The common case is printing out a sequence of variables separated by whitespace.

1097024x, 2018, 12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2624 by University Of Waterloo Dana Porter Library, Wiley Online Library on [22/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

CV	C++
`int x = 1, y = 2, z = 3;`	
`sout	x	y	z	endl;`	`cout << x << " " << y << " " << z << endl;`
`1␣2␣3`	`1␣2␣3`

The C∀ form has half the characters of the C++ form and is similar to Python I/O with respect to implicit separators. Similar simplification occurs for tuple I/O, which prints all tuple values separated by ",␣".

```
[int, [ int, int ] ] t1 = [ 1, [ 2, 3 ] ], t2 = [ 4, [ 5, 6 ] ];
sout | t1 | t2 | endl;                                    // print tuples
1,␣2,␣3␣4,␣5,␣6
```

Finally, C∀ uses the logical-or operator for I/O as it is the lowest-priority overloadable operator, other than assignment. Therefore, fewer output expressions require parenthesis.

```
C∀:  sout | x * 3 | y + 1 | z << 2 | x == y | (x | y) | (x || y) | (x > z ? 1 : 2) | endl;
C++: cout << x * 3 << y + 1 << (z << 2) << (x == y) << (x | y) << (x || y) << (x > z ? 1 : 2) << endl;
     3␣3␣12␣0␣3␣1␣2
```

There is a weak similarity between the C∀ logical-or operator and the Shell pipe operator for moving data, where data flow in the correct direction for input but in the opposite direction for output. There are functions to set and get the separator string and manipulators to toggle separation on and off in the middle of output.

8.5 | Multiprecision integers

C∀ has an interface to the GNU multiple precision (GMP) signed integers,[25] similar to the C++ interface provided by GMP. The C∀ interface wraps GMP functions into operator functions to make programming with multiprecision integers identical to using fixed-sized integers. The C∀ type name for multiprecision signed integers is Int, and the header file is gmp. Figure 8 shows a multiprecision factorial program contrasting the GMP interface in C∀ and C.

9 | POLYMORPHISM EVALUATION

C∀ adds parametric polymorphism to C. A runtime evaluation is performed to compare the cost of alternative styles of polymorphism. The goal is to compare just the underlying mechanism for implementing different kinds of polymorphism. The experiment is a set of generic-stack microbenchmarks[26] in C, C∀, and C++ (see implementations in Appendix A). Since all these languages share a subset essentially comprising standard C, maximal-performance benchmarks should show little runtime variance, differing only in length and clarity of source code. A more illustrative comparison measures the costs of idiomatic usage of each language's features. Figure 9 shows the C∀ benchmark tests for a generic stack based on a singly linked list. The benchmark test is similar for the other languages. The experiment uses element types **int** and pair(**short**, **char**) and pushes $N = 40M$ elements on a generic stack, copies the stack, clears one of the stacks, and finds the maximum value in the other stack.

C∀

```
#include <gmp>
int main( void ) {
    sout | "Factorial Numbers" | endl;
    Int fact = 1;
    sout | 0 | fact | endl;
    for ( unsigned int i = 1; i <= 40; i += 1 ) {
        fact *= i;
        sout | i | fact | endl;
    }
}
```

C

```
#include <gmp.h>
int main( void ) {
    gmp_printf( "Factorial Numbers\n" );
    mpz_t fact;  mpz_init_set_ui( fact, 1 );
    gmp_printf( "%d %Zd\n", 0, fact );
    for ( unsigned int i = 1; i <= 40; i += 1 ) {
        mpz_mul_ui( fact, fact, i );
        gmp_printf( "%d %Zd\n", i, fact );
    }
}
```

FIGURE 8 GMP interface C∀ versus C [Colour figure can be viewed at wileyonlinelibrary.com]

1097024x, 2018, 12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2624 by University Of Waterloo Dana Porter Library, Wiley Online Library on [22/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

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

    REPEAT_TIMED( "push_int", N, push( si, val ); )
    TIMED( "copy_int", ti{ si }; )
    TIMED( "clear_int", clear( si ); )
    REPEAT_TIMED( "pop_int", N, int x = pop( ti ); if ( x > max ) max = x; )

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

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

FIGURE 9 C∀ benchmark test

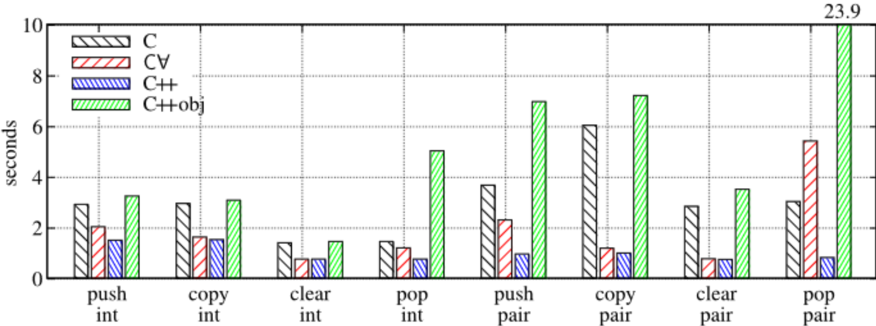

FIGURE 10 Benchmark timing results (smaller is better) [Colour figure can be viewed at wileyonlinelibrary.com]

TABLE 2 Properties of benchmark code

	C	C∀	C++	C++obj
maximum memory usage (MB)	10 001	2502	2503	11 253
source code size (lines)	201	191	125	294
redundant type annotations (lines)	27	0	2	16
binary size (KB)	14	257	14	37

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

Figure 10 and Table 2 show the results of running the benchmark in Figure 9 and its C, C++, and C++obj equivalents. The graph plots the median of five consecutive runs of each program, with an initial warm-up run omitted. All code is compiled at `-O2` by gcc or g++ 6.4.0, with all C++ code compiled as C++14. The benchmarks are run on an Ubuntu 16.04 workstation with 16 GB of RAM and a 6-core AMD FX-6300 CPU with 3.5 GHz maximum clock frequency.

The C and C++obj variants are generally the slowest with the largest memory footprint, due to their less-efficient memory layout and the pointer indirection necessary to implement generic types; this inefficiency is exacerbated by the second level of generic types in the pair benchmarks. By contrast, the C∀ and C++ variants run in roughly equivalent time for both the integer and pair because of the equivalent storage layout, with the inlined libraries (ie, no separate compilation) and greater maturity of the C++ compiler contributing to its lead. C++obj is slower than C largely due to the cost

1097024x, 2018, 12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2624 by University Of Waterloo Dana Porter Library, Wiley Online Library on [22/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

of runtime type checking of downcasts (implemented with dynamic_cast); the outlier for C∀, pop pair, results from the complexity of the generated-C polymorphic code. The gcc compiler is unable to optimize some dead code and condense nested calls; a compiler designed for C∀ could easily perform these optimizations. Finally, the binary size for C∀ is larger because of static linking with the C∀ libraries.

C∀ is also competitive in terms of source code size, measured as a proxy for programmer effort. The line counts in Table 2 include implementations of pair and stack types for all four languages for purposes of direct comparison, although it should be noted that C∀ and C++ have prewritten data structures in their standard libraries that programmers would generally use instead. Use of these standard library types has minimal impact on the performance benchmarks, but shrinks the C∀ and C++ benchmarks to 39 and 42 lines, respectively. The difference between the C∀ and C++ line counts is primarily declaration duplication to implement separate compilation; a header-only C∀ library would be similar in length to the C++ version. On the other hand, C does not have a generic collections library in its standard distribution, resulting in frequent reimplementation of such collection types by C programmers. C++obj does not use the C++ standard template library by construction and, in fact, includes the definition of object and wrapper classes for **char**, **short**, and **int** in its line count, which inflates this count somewhat, as an actual object-oriented language would include these in the standard library; with their omission, the C++obj line count is similar to C. We justify the given line count by noting that many object-oriented languages do not allow implementing new interfaces on library types without subclassing or wrapper types, which may be similarly verbose.

Line count is a fairly rough measure of code complexity; another important factor is how much type information the programmer must specify manually, especially where that information is not compiler checked. Such unchecked type information produces a heavier documentation burden and increased potential for runtime bugs and is much less common in C∀ than C, with its manually specified function pointer arguments and format codes, or C++obj, with its extensive use of un–type-checked downcasts, eg, object to integer when popping a stack. To quantify this manual typing, the "redundant type annotations" line in Table 2 counts the number of lines on which the type of a known variable is respecified, either as a format specifier, explicit downcast, type-specific function, or by name in a **sizeof**, struct literal, or new expression. The C++ benchmark uses two redundant type annotations to create a new stack nodes, whereas the C and C++obj benchmarks have several such annotations spread throughout their code. The C∀ benchmark is able to eliminate all redundant type annotations through the use of the polymorphic alloc function discussed in Section 8.

We conjecture that these results scale across most generic data types as the underlying polymorphism implement is constant.

10 | RELATED WORK

10.1 | Polymorphism

ML[27] was the first language to support parametric polymorphism. Like C∀, it supports universal type parameters, but not the use of assertions and traits to constrain type arguments. Haskell[28] combines ML-style polymorphism, polymorphic data types, and type inference with the notion of type classes, collections of overloadable methods that correspond in intent to traits in C∀. Unlike C∀, Haskell requires an explicit association between types and their classes that specifies the implementation of operations. These associations determine the functions that are assertion arguments for particular combinations of class and type, in contrast to C∀ where the assertion arguments are selected at function call sites based upon the set of operations in scope at that point. Haskell also severely restricts the use of overloading: an overloaded name can only be associated with a single class, and methods with overloaded names can only be defined as part of instance declarations.

C++ provides three disjoint polymorphic extensions to C: overloading, inheritance, and templates. The overloading is restricted because resolution does not use the return type, inheritance requires learning object-oriented programming and coping with a restricted nominal-inheritance hierarchy, templates cannot be separately compiled resulting in compilation/code bloat and poor error messages, and determining how these mechanisms interact and which to use is confusing. In contrast, C∀ has a single facility for polymorphic code supporting type-safe separate compilation of polymorphic functions and generic (opaque) types, which uniformly leverage the C procedural paradigm. The key mechanism to support separate compilation is C∀'s *explicit* use of assumed type properties. Until C++ concepts[29] are standardized (anticipated for C++20), C++ provides no way of specifying the requirements of a generic function beyond compilation errors during template expansion; furthermore, C++ concepts are restricted to template polymorphism.

1097024x, 2018, 12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2624 by University Of Waterloo Dana Porter Library, Wiley Online Library on [22/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

Cyclone[30] also provides capabilities for polymorphic functions and existential types, similar to C∀'s **forall** functions and generic types. Cyclone existential types can include function pointers in a construct similar to a virtual function table, but these pointers must be explicitly initialized at some point in the code, which is a tedious and potentially error-prone process. Furthermore, Cyclone's polymorphic functions and types are restricted to abstraction over types with the same layout and calling convention as **void**∗, ie, only pointer types and **int**. In C∀ terms, all Cyclone polymorphism must be dtype-static. While the Cyclone design provides the efficiency benefits discussed in Section 3.3 for dtype-static polymorphism, it is more restrictive than C∀'s general model. Smith and Volpano[31] present Polymorphic C, an ML dialect with polymorphic functions, C-like syntax, and pointer types; it lacks many of C's features, most notably structure types, and, hence, is not a practical C replacement.

Objective-C[32] is an industrially successful extension to C. However, Objective-C is a radical departure from C, using an object-oriented model with message passing. Objective-C did not support type-checked generics until recently,[33] historically using less-efficient runtime checking of object types. The GObject[34] framework also adds object-oriented programming with runtime type-checking and reference-counting garbage collection to C; these features are more intrusive additions than those provided by C∀, in addition to the runtime overhead of reference counting. Vala[35] compiles to GObject-based C, adding the burden of learning a separate language syntax to the aforementioned demerits of GObject as a modernization path for existing C code bases. Java[36] included generic types in Java 5, which are type checked at compilation and type erased at runtime, similar to C∀'s. However, in Java, each object carries its own table of method pointers, whereas C∀ passes the method pointers separately to maintain a C-compatible layout. Java is also a garbage-collected, object-oriented language, with the associated resource usage and C-interoperability burdens.

D,[37] Go, and Rust[38] are modern compiled languages with abstraction features similar to C∀ traits, *interfaces* in D and Go, and *traits* in Rust. However, each language represents a significant departure from C in terms of language model, and none has the same level of compatibility with C as C∀. D and Go are garbage-collected languages, imposing the associated runtime overhead. The necessity of accounting for data transfer between managed runtimes and the unmanaged C runtime complicates foreign-function interfaces to C. Furthermore, while generic types and functions are available in Go, they are limited to a small fixed set provided by the compiler, with no language facility to define more. D restricts garbage collection to its own heap by default, whereas Rust is not garbage collected and, thus, has a lighter-weight runtime more interoperable with C. Rust also possesses much more powerful abstraction capabilities for writing generic code than Go. On the other hand, Rust's borrow checker provides strong safety guarantees but is complex and difficult to learn and imposes a distinctly idiomatic programming style. C∀, with its more modest safety features, allows direct ports of C code while maintaining the idiomatic style of the original source.

10.2 | Tuples/variadics

Many programming languages have some form of tuple construct and/or variadic functions, eg, SETL, C, KW-C, C++, D, Go, Java, ML, and Scala. SETL[39] 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. 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. KW-C,[21] a predecessor of C∀, introduced tuples to C as an extension of the C syntax, taking much of its inspiration from SETL. The main contributions of that work were adding MRVF, tuple mass and multiple assignment, and record-member access. C++11 introduced std::tuple as a library variadic-template structure. Tuples are a generalization of std::pair, in that they allow 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*[40] to eliminate predeclaring 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. Like C++, D provides tuples through a library variadic-template structure. Go does not have tuples but supports MRVF. Java's variadic functions appear similar to C's but are type safe using homogeneous arrays, which are less useful than C∀'s heterogeneously typed variadic functions. Tuples are a fundamental abstraction in most functional programming languages, such as Standard ML,[41] Haskell, and Scala,[42] which decompose tuples using pattern matching.

10.3 | C extensions

C++ is the best known C-based language and is similar to C∀ in that both are extensions to C with source and runtime backward compatibility. Specific differences between C∀ and C++ have been identified in prior sections, with a final

1097024x, 2018, 12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2624 by University Of Waterloo Dana Porter Library, Wiley Online Library on [22/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

observation that C∀ has equal or fewer tokens to express the same notion in many cases. The key difference in design philosophies is that C∀ is easier for C programmers to understand by maintaining a procedural paradigm and avoiding complex interactions among extensions. C++, on the other hand, has multiple overlapping features (such as the three forms of polymorphism), many of which have complex interactions with its object-oriented design. As a result, C++ has a steep learning curve for even experienced C programmers, especially when attempting to maintain performance equivalent to C legacy code.

There are several other C extension languages with less usage and even more dramatic changes than C++. Objective-C and Cyclone are two other extensions to C with different design goals than C∀, as discussed above. Other languages extend C with more focused features. μC++,[43] CUDA,[44] ispc,[45] and Sierra[46] add concurrent or data-parallel primitives to C or C++; data-parallel features have not yet been added to C∀ but are easily incorporated within its design, whereas concurrency primitives similar to those in μC++ have already been added.[4] Finally, CCured[47] and Ironclad C++[48] attempt to provide a more memory-safe C by annotating pointer types with garbage collection information; type-checked polymorphism in C∀ covers several of C's memory-safety issues, but more aggressive approaches such as annotating all pointer types with their nullability or requiring runtime garbage collection are contradictory to C∀'s backward compatibility goals.

11 | CONCLUSION AND FUTURE WORK

The goal of C∀ is to provide an evolutionary pathway for large C development environments to be more productive and safer, while respecting the talent and skill of C programmers. While other programming languages purport to be a better C, they are, in fact, new and interesting languages in their own right, but not C extensions. The purpose of this paper is to introduce C∀ and showcase language features that illustrate the C∀ type system and approaches taken to achieve the goal of evolutionary C extension. The contributions are a powerful type system using parametric polymorphism and overloading, generic types, tuples, advanced control structures, and extended declarations, which all have complex interactions. The work is a challenging design, engineering, and implementation exercise. On the surface, the project may appear as a rehash of similar mechanisms in C++. However, every C∀ feature is different than its C++ counterpart, often with extended functionality, better integration with C and its programmers, and always supporting separate compilation. All of these new features are being used by the C∀ development team to build the C∀ runtime system. Finally, we demonstrate that C∀ performance for some idiomatic cases is better than C and close to C++, showing the design is practically applicable.

While all examples in the paper compile and run, there are ongoing efforts to reduce compilation time, provide better debugging, and add more libraries; when this work is complete in early 2019, a public beta release will be available at https://github.com/cforall/cforall. There is also new work on a number of C∀ features, including arrays with size, runtime type information, virtual functions, user-defined conversions, and modules. While C∀ polymorphic functions use dynamic virtual dispatch with low runtime overhead (see Section 9), it is not as low as C++ template inlining. Hence, it may be beneficial to provide a mechanism for performance-sensitive code. Two promising approaches are an `inline` annotation at polymorphic function call sites to create a template specialization of the function (provided the code is visible) or placing an `inline` annotation on polymorphic function definitions to instantiate a specialized version for some set of types (C++ template specialization). These approaches are not mutually exclusive and allow performance optimizations to be applied only when necessary, without suffering global code bloat. In general, we believe separate compilation, producing smaller code, works well with loaded hardware caches, which may offset the benefit of larger inlined code.

ACKNOWLEDGEMENTS

The authors would like to recognize the design assistance of Glen Ditchfield, Richard Bilson, Thierry Delisle, Andrew Beach, and Brice Dobry on the features described in this paper and thank Magnus Madsen for feedback on the writing. Funding for this project was provided by Huawei Ltd (http://www.huawei.com), and Aaron Moss and Peter Buhr were partially funded by the Natural Sciences and Engineering Research Council of Canada.

ORCID

Aaron Moss http://orcid.org/0000-0003-4569-976X
Peter A. Buhr http://orcid.org/0000-0003-3747-9281

1097024x, 2018, 12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2624 by University Of Waterloo Dana Porter Library, Wiley Online Library on [22/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

REFERENCES

1. TIOBE Index. http://www.tiobe.com/tiobe_index
2. American National Standards Institute. American National Standard for programming languages – C ANSI/ISO 9899-1990. New York, NY: American National Standards Institute; 1990.
3. Bilson RC. *Implementing Overloading and Polymorphism in C∀* [master's thesis]. Waterloo, Canada: School of Computer Science, University of Waterloo; 2003. http://plg.uwaterloo.ca/theses/BilsonThesis.pdf
4. Delisle T. *Concurrency in C∀* [thesis]. Waterloo, Canada: School of Computer Science, University of Waterloo; 2018. https://uwspace.uwaterloo.ca/handle/10012/12888
5. C Extensions. Extensions to the C language family. 2014. https://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/C-Extensions.html
6. Ditchfield GJ. *Contextual Polymorphism* [PhD thesis]. Waterloo, Canada: Department of Computer Science, University of Waterloo; 1992. http://plg.uwaterloo.ca/theses/DitchfieldThesis.pdf
7. International Organization for Standardization. ISO/IEC 9899:2011: information technology – programming languages – C. 3rd ed. 2012. https://www.iso.org/standard/57853.html
8. Cormack GV, Wright AK. Polymorphism in the compiled language force one. In: Proceedings of the 20th Hawaii International Conference on Systems Sciences (HICSS-22); 1987; Honolulu, HI.
9. Cormack GV, Wright AK. Type-dependent parameter inference. In: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation; 1990; White Plains, NY.
10. Duggan D, Cormack GV, Ophel J. Kinded type inference for parametric overloading. *Acta Infomatica*. 1996;33(1):21-68.
11. Shaw M. *ALPHARD: Form and Content*. New York, NY: Springer; 1981.
12. Cormack GV. An algorithm for the selection of overloaded functions in ADA. *ACM SIGPLAN Not*. 1981;16(2):48-52.
13. Baker TP. A one-pass algorithm for overload resolution in Ada. *ACM Trans Program Lang Syst*. 1982;4(4):601-614.
14. Hutchison D, Kanade T, Kittler J, et al. *The Programming Language Ada. Reference Manual: American National Standards Institute, Inc. ANSI/MIL-STD-1815A-1983. Approved 17 February 1983*. New York, NY: Springer; 1983.
15. Griesemer R, Pike R, Thompson K. Go programming language. Google; 2009. http://golang.org/ref/spec
16. Duff T. Duff's device. 1983. http://www.lysator.liu.se/c/duffs-device.html
17. Buhr PA. A case for teaching multi-exit loops to beginning programmers. *ACM SIGPLAN Not*. 1985;20(11):14-22.
18. Buhr PA, Macdonald HI, Zarnke CR. Synchronous and asynchronous handling of abnormal events in the μsystem. *Softw Pract Exp*. 1992;22(9):735-776.
19. Buhr PA, Mok WYR. Advanced exception handling mechanisms. *IEEE Trans Softw Eng*. 2000;26(9):820-836.
20. Jensen K, Wirth N. *Pascal User Manual and Report: ISO Pascal Standard*. 4th ed. New York, NY: Springer-Verlag; 1991. Revised by Andrew B. Mickel and James F. Miner.
21. Buhr PA, Till D, Zarnke CR. Assignment as the sole means of updating objects. *Softw Pract Exp*. 1994;24(9):835-870.
22. Taylor IL. const. 2010. https://www.airs.com/blog/archives/428
23. Schluntz R. *Resource Management and Tuples in C∀* [master's thesis]. Waterloo, Canada: School of Computer Science, University of Waterloo; 2017. https://uwspace.uwaterloo.ca/handle/10012/11830
24. International Organization for Standardization. ISO/IEC 14882:1998: programming language – C++. 1st ed. 1998. https://www.iso.org/standard/25845.html
25. GNU Multiple Precision Arithmetic Library. GNU. 2016. https://gmplib.org
26. Moss A. C∀ stack evaluation programs. 2018. https://cforall.uwaterloo.ca/CFAStackEvaluation.zip
27. Milner R. A theory of type polymorphism in programming. *J Comput Syst Sci*. 1978;17:348-375.
28. Simon Marlow. Haskell 2010: language report. 2010. https://haskell.org/definition/haskell2010.pdf
29. International Organization for Standardization. ISO/IEC TS 19217:2015: information technology – programming language –C++ Extensions for concepts. 2015. https://www.iso.org/standard/64031.html
30. Grossman D. Quantified types in an imperative language. *ACM Trans Program Lang Syst*. 2006;28(3):429-475.
31. Smith G, Volpano D. A sound polymorphic type system for a dialect of C. *Sci Comput Program*. 1998;32(1-3):49-72.
32. Objective-C. 2015. https://developer.apple.com/library/content/documentation/General/Conceptual/DevPedia-CocoaCore/ObjectiveC.html
33. Xcode 7 release notes. 2015. https://developer.apple.com/library/content/documentation/Xcode/Conceptual/RN-Xcode-Archive/Chapters/xc7_release_notes.html
34. GObject reference manual. 2014. https://developer.gnome.org/gobject/stable/
35. Vala reference manual. 2017. https://wiki.gnome.org/Projects/Vala/Manual
36. Gosling J, Joy B, Steele G, Bracha G, Buckley A. *Java Language Specification*. Java SE 8. Upper Saddle River, NJ: Addison-Wesley; 2015.
37. Bright W, Alexandrescu A. D programming language. Digital Mars. 2016. http://dlang.org/spec/spec.html
38. The Rust Programming Language. The rust project developers. 2015. https://doc.rust-lang.org/reference.html
39. Schwartz JT, Dewar RBK, Dubinsky E, Schonberg E. *Programming With Sets: An Introduction to SETL*. New York, NY: Springer; 1986.
40. Sutter H, Stroustrup B, Reis GD. Structured bindings. 2015. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0144r0.pdf

1097024x, 2018, 12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2624 by University Of Waterloo Dana Porter Library, Wiley Online Library on [22/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

41. Milner R, Tofte M, Harper R. *The Definition of Standard ML*. Cambridge, MA: The MIT Press; 1990.
42. Scala language specification: version 2.11. École Polytechnique Fédérale de Lausanne. 2016. http://www.scala-lang.org/files/archive/spec/2.11
43. Buhr PA. *Understanding Control Flow: Concurrent Programming Using μC++*. Cham, Switzerland: Springer; 2016.
44. Nickolls J, Buck I, Garland M, Skadron K. Scalable parallel programming with CUDA. *Queue*. 2008;6(2):40-53.
45. Pharr M, Mark WR. ispc: a SPMD compiler for high-performance CPU programming. Paper presented at: 2012 Innovative Parallel Computing (InPar); 2012; San Jose, CA.
46. Leißa R, Haffner I, Hack S. Sierra: a SIMD extension for C++. In: Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing; 2014; Orlando, FL.
47. Necula GC, McPeak S, Weimer W. CCured: type-safe retrofitting of legacy code. In: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages(POPL); 2002; New York, NY.
48. DeLozier C, Eisenberg R, Nagarakatte S, Osera P-M, Martin MMK, Zdancewic S. Ironclad C++: a library-augmented type-safe subset of c++. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA); 2013; Indianapolis, IN.

How to cite this article: Moss A, Schluntz R, Buhr PA. C∀: Adding modern programming language features to C. *Softw Pract Exper*. 2018;48:2111–2146. https://doi.org/10.1002/spe.2624

APPENDIX A

BENCHMARK STACK IMPLEMENTATIONS

Throughout, /***/ designates a counted redundant type annotation; code reformatted slightly for brevity.

A.1 | C

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

1097024x, 2018, 12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2624 by University Of Waterloo Dana Porter Library, Wiley Online Library on [22/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

A.2 | C∀

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

A.3 | C++

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

1097024x, 2018, 12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2624 by University Of Waterloo Dana Porter Library, Wiley Online Library on [22/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

A.4 | C++obj

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

1097024x, 2018, 12, Downloaded from https://onlinelibrary.wiley.com/doi/10.1002/spe.2624 by University Of Waterloo Dana Porter Library, Wiley Online Library on [22/03/2024]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License