C∀ (Cforall) User Manual
Version 1.0

“describe not prescribe”

C∀ Team (past and present)

Andrew Beach, Richard Bilson, Michael Brooks, Peter A. Buhr, Thierry Delisle, Glen Ditchfield, Rodolfo G. Esteves, Jiada Liang, Aaron Moss, Colby Parsons Rob Schluntz, Fangren Yu, Mubeen Zulfiqar

February 13, 2024

Contents

1 Introduction

C∀[1] is a modern general-purpose concurrent programming-language, designed as an evolutionary step forward for the C programming language. The syntax of C∀ builds from C and should look immediately familiar to C/C++ programmers. C∀ adds many modern features that directly lead to increased *safety* and *productivity*, while maintaining interoperability with existing C programs and achieving similar performance. Like C, C∀ is a statically typed, procedural (non-object-oriented) language with a low-overhead runtime, meaning there is no global garbage-collection, but regional garbage-collection is possible. The primary new features include polymorphic routines and types, exceptions, concurrency, and modules.

One of the main design philosophies of C∀ is to "describe not prescribe", which means C∀ tries to provide a pathway from low-level C programming to high-level C∀ programming, but it does not force programmers to "do the right thing". Programmers can cautiously add C∀ extensions to their C programs in any order and at any time to incrementally move towards safer, higher-level programming. A programmer is always free to reach back to C from C∀, for any reason, and in many cases, new C∀ features can be locally switched back to their C counterpart. There is no notion or requirement for *rewriting* a legacy C program to C∀; instead, a programmer evolves a legacy program into C∀ by incrementally incorporating C∀ features. As well, new programs can be written in C∀ using a combination of C and C∀ features. In many ways, C∀ is to C as Scala [29] is to Java, providing a vehicle for new typing and control-flow capabilities on top of a highly popular programming language allowing immediate dissemination.

C++ [30] had a similar goal 30 years ago, allowing object-oriented programming to be incrementally added to C. However, C++ currently has the disadvantages of a strong object-oriented bias, multiple legacy design-choices that are difficult to update, and active divergence of the language model from C, requiring significant effort and training to incrementally add C++ to a C code-base. In contrast, C∀ has 30 years of hindsight and a clean starting point.

Like C++, there may be both old and new ways to achieve the same effect. For example, the following programs compare the C, C∀, and C++ I/O mechanisms, where the programs output the same result.

```
               C                                C∀                       C++
#include <stdio.h>                  #include <fstream>          #include <iostream>
                                                                using namespace std;
int main( void ) {                  int main( void ) {          int main() {
    int x = 0, y = 1, z = 2;          int x = 0, y = 1, z = 2;    int x = 0, y = 1, z = 2;
    printf( "%d %d %d\n", x, y, z );  sout | x | y | z;           cout << x << ' ' << y << ' ' << z << endl;
}                                   }                           }
```

While C∀ I/O (see Section 22, p. 49) looks similar to C++, there are important differences, such as automatic spacing between variables and an implicit newline at the end of the expression list, similar to Python [26]. In general, C∀ programs are 10% to 30% shorter than their equivalent C/C++ counterparts.

1.1 Background

This document is a programmer reference-manual for the C∀ programming language. The manual covers the core features of the language and runtime-system, with simple examples illustrating syntax and semantics of features. The manual does not teach programming, *i.e.*, how to combine the new constructs to build complex programs. The reader must have an intermediate knowledge of control flow, data structures, and concurrency issues to understand the ideas presented, as well as some experience programming in C/C++. Implementers should refer to the C∀ Programming Language Specification for details about the language syntax and semantics. Changes to the syntax and additional features are expected to be included in later revisions.

2 Why fix C?

The C programming language is a foundational technology for modern computing with billions 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. Even with all its problems, C continues to be popular because it allows writing software at virtually any level in a computer system without restriction. For system programming, where direct access to hardware, storage management, and real-time issues are a requirement, C is the only language of choice. The TIOBE index [32] for February

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

2023 ranks the top six most *popular* programming languages as C 17.4%, Java 12%, Python 12%, C++ 7.6%, C♯ 4%, Visual Basic 3.8% = 56.8%, where the next 50 languages are less than 2% each, with a long tail. The top 4 rankings over the past 35 years are:

	2023	2018	2013	2008	2003	1998	1993	1988
Python	1	4	8	7	12	25	18	-
C	**2**	**2**	**1**	**2**	**2**	**1**	**1**	**1**
C++	3	3	4	4	3	2	2	4
Java	4	1	2	1	1	18	-	-

Hence, C is still an extremely important programming language, with double the usage of C++; in many cases, C++ is often used solely as a better C. Love it or hate it, C has been an important and influential part of computer science for 40 years and its appeal is not diminishing. Nevertheless, C has many problems and omissions that make it an unacceptable programming language for modern needs.

As stated, the goal of the C∀ project is to engineer modern language-features into C in an evolutionary rather than revolutionary way. C++ [22, 12] is an example of a similar project; however, it largely extended the C language, and did not address many of C's existing problems.[2] Fortran [14], Cobol [6], and Ada [1] are examples of programming languages that took an evolutionary approach, where modern language-features (*e.g*., objects, concurrency) are added and problems fixed within the framework of the existing language. Java [18], Go [19], Rust [28] and D [3] are examples of the revolutionary approach for modernizing C/C++, resulting in a new language rather than an extension of the descendent. These languages have different syntax and semantics from C, do not interoperate directly with C, and are not systems languages because of restrictive memory-management or garbage collection. As a result, there is a significant learning curve to move to these languages, and C legacy-code must be rewritten. These costs can be prohibitive for many companies with a large software-base in C/C++, and a significant number of programmers require retraining in the new programming language.

The result of this project is a language that is largely backwards compatible with C11 [21], but fixes many of the well known C problems while adding modern language-features. To achieve these goals required a significant engineering exercise, *i.e*., "thinking *inside* the C box". Considering the large body of existing C code and programmers, there is significant impetus to ensure C is transformed into a modern language. While C11 made a few simple extensions to the language, nothing was added to address existing problems in the language or to augment the language with modern language-features. While some may argue that modern language-features may make C complex and inefficient, it is clear a language without modern capabilities is insufficient for the advanced programming problems existing today.

3 History

The C∀ project started with Dave Till's K-W C [5, 31], which extended C with new declaration syntax, multiple return values from routines, and advanced assignment capabilities using the notion of tuples (see [33] for similar work in C++). The first C∀ implementation of these extensions was by Rodolfo Esteves [13].

The signature feature of C∀ is *overloadable* parametric-polymorphic functions [7, 8, 11] with functions generalized using a **forall** clause (giving the language its name):

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

C∀'s polymorphism was originally formalized by Glen Ditchfield [9], and first implemented by Richard Bilson [2]. However, at that time, there was little interest in extending C, so work did not continue. As the saying goes, "What goes around, comes around.", and there is now renewed interest in the C programming language because of the legacy code-base, so the C∀ project was restarted in 2015.

4 Interoperability

C∀ is designed to integrate directly with existing C programs and libraries. The most important feature of interoperability is using the same calling conventions, so there is no complex interface or overhead to call existing C routines. This feature allows C∀ programmers to take advantage of the existing panoply of C libraries to access thousands of

[2]Two important existing problems addressed were changing the type of character literals from **int** to **char** and enumerator from **int** to the type of its enumerators.

software features. Language developers often state that adequate library support takes more work than designing and implementing the language itself. Fortunately, C∀, like C++, starts with immediate access to all exiting C libraries, and in many cases, can easily wrap library routines with simpler and safer interfaces, at zero or very low cost. Hence, C∀ begins by leveraging the large repository of C libraries, and than allows programmers to incrementally augment their C programs with modern backward-compatible features.

However, it is necessary to differentiate between C and C∀ code because of name overloading, as for C++. For example, the C math-library provides the following routines for computing the absolute value of the basic types: abs, labs, llabs, fabs, fabsf, fabsl, cabsf, cabs, and cabsl. Whereas, C∀ wraps these routines into one overloaded name abs:

```
unsigned char abs( signed char );                      // no C equivalent
extern "C" { int abs( int ); }                         // C abs
unsigned long int abs( long int );                     // C labs
unsigned long long int abs( long long int );           // C llabs
float abs( float );                                    // C fabsf
double abs( double );                                  // C fabs
long double abs( long double );                        // C fabsl
float _Complex abs( float _Complex );                  // C cabsf
double _Complex abs( double _Complex );                // C cabs
long double _Complex abs( long double _Complex );      // C cabsl
```

The problem is a name clash between the C name abs and the C∀ names abs, resulting in two name linkages: **extern** "C" and **extern** "Cforall" (default). Overloaded names must use *name mangling* to create unique names that are different from unmangled C names. Hence, there is the same need as in C++ to know if a name is a C or C∀ name, so it can be correctly formed. The only way around this problem is C's approach of creating unique names for each pairing of operation and type.

This example illustrates a core idea in C∀: *the power of a name*. The name "abs" evokes the notion of absolute value and many mathematical types provide the notion of absolute value. Hence, knowing the name abs is sufficient to apply it to any applicable type. The time savings and safety of using one name uniformly versus *N* unique names cannot be underestimated.

5 C∀ Compilation

C∀ is a *transpiler*, meaning it reads in a programming language (C∀) as input and generates another programming language (C) as output, whereas a *compiler* reads in a programming language and generates assembler/machine code. Hence, C∀ is like the C preprocessor modifying a program and sending it on to another step for further transformation. The order of transformation is C preprocessor, C∀, and finally GNU C compiler, which also has a number of transformation steps, such as assembler and linker.

The command cfa is used to compile a C∀ program and is based on the GNU gcc command, *e.g.*:

```
cfa [ gcc/C∀-options ] [ C/C∀ source-files ] [ assembler/loader files ]
```

There is no ordering among options (flags) and files, unless an option has an argument, which must appear immediately after the option possibly with or without a space separating option and argument.

C∀ has the following gcc flags turned on:

-std=gnu11 The 2011 C standard plus GNU extensions.

-fgnu89-inline Use the traditional GNU semantics for inline routines in C11 mode, which allows inline routines in header files.

C∀ has the following new options:

-CFA Only the C preprocessor (flag -E) and the C∀ translator steps are performed and the transformed program is written to standard output, which makes it possible to examine the code generated by the C∀ translator. The generated code starts with the standard C∀ prelude.

-XCFA Pass next flag as-is to the cfa-cpp translator (see details below).

-debug The program is linked with the debugging version of the runtime system. The debug version performs runtime checks to aid the debugging phase of a C∀ program, but can substantially slow program execution. The runtime checks should only be removed after a program is completely debugged. **This option is the default.**

-nodebug The program is linked with the non-debugging version of the runtime system, so the execution of the program is faster. ***However, no runtime checks or* asserts *are performed so errors usually result in abnormal***

program behaviour or termination.

–help Information about the set of C∀ compilation flags is printed.

–nohelp Information about the set of C∀ compilation flags is not printed. **This option is the default.**

–quiet The C∀ compilation message is not printed at the beginning of a compilation.

–noquiet The C∀ compilation message is printed at the beginning of a compilation. **This option is the default.**

The following preprocessor variables are available:

__CFA_MAJOR__ is available during preprocessing and its value is the major version number of C∀.[3]

__CFA_MINOR__ is available during preprocessing and its value is the minor version number of C∀.

__CFA_PATCH__ is available during preprocessing and its value is the patch level number of C∀.

__CFA__, __CFORALL__, and __cforall are always available during preprocessing and have no value.

These preprocessor variables allow conditional compilation of programs that must work differently in these situations. For example, to toggle between C and C∀ extensions, use the following:

```
#ifndef __CFORALL__
#include <stdio.h>                  // C header file
#else
#include <fstream>                  // C∀ header file
#endif
```

which conditionally includes the correct header file, if the program is compiled using gcc or cfa.

The C∀ transpiler has multiple internal steps. The following flags control how the C∀ transpiler works, the stages run, and printing within a stage. The majority of these flags are used by C∀ developers, but some are occasionally useful to programmers. Each option must be escaped with –XCFA to direct it to the C∀ compilation step, similar to the –Xlinker flag for the linker, *e.g.*:

```
cfa test.cfa –CFA –XCFA –p # print translated code without printing the standard prelude
cfa test.cfa –XCFA –P –XCFA parse –XCFA –n # show program parse without prelude
```

Alternatively, multiple flags can be specified separated with commas and *without* spaces.

```
cfa test.cfa –XCFA,–Pparse,–n # show program parse without prelude
```

–c, ––colors diagnostic color: never, always, auto

–g, ––gdb wait for gdb to attach

–h, ––help print transpiler help message

–i, ––invariant invariant checking during AST passes

–l, ––libcfa generate libcfa.c

–L, ––linemarks generate line marks

–m, ––no–main do not replace main

–N, ––no–linemarks do not generate line marks

–n, ––no–prelude do not read prelude

–p, ––prototypes do not generate prelude prototypes ⇒ prelude not printed

–d, ––deterministic–out only print deterministic output

–P, ––print one of:

- ascodegen print AST as codegen rather than AST
- asterr print AST on error
- declstats print code property statistics
- parse print yacc (parsing) debug information
- pretty prettyprint for ascodegen flag
- rproto resolver-proto instance
- rsteps print resolver steps
- ast print AST after parsing
- excpdecl print AST after translating exception decls
- symevt print AST after symbol table events

[3]The C preprocessor allows only integer values in a preprocessor variable so a value like "1.0.0 " is not allowed. Hence, the need to have three variables for the major, minor and patch version number.

```
// include file uses the CFA keyword "with".
#if ! defined( with )                     // nesting ?
#define with ``with                       // make keyword an identifier
#define __CFA_BFD_H__
#endif
#include_next <bfdlink.h>                 // must have internal check for multiple expansion
#if defined( with ) && defined( __CFA_BFD_H__ )   // reset only if set
#undef with
#undef __CFA_BFD_H__
#endif
```

Figure 1: Header-File Interposition

exprait print AST after expressions alternatives
valdecl print AST after declaration validation pass
bresolver print AST before resolver step
exprany print AST after expression analysis
ctordtor print AST after ctor/dtor are replaced
tuple print AST after tuple expansion
instgen print AST after instantiate generics
bbox print AST before box pass
bcodegen print AST before code generation
--prelude-dir <directory> prelude directory for debug/nodebug
-S, --statistics <option-list> enable profiling information: counters, heap, time, all, none
-t, --tree build in tree

6 Backquote Identifiers

C∀ introduces several new keywords (see Section C, p. 73) that can clash with existing C variable-names in legacy code. Keyword clashes are accommodated by syntactic transformations using the C∀ backquote escape-mechanism:

```
int ``coroutine = 3;                      // make keyword an identifier
double ``forall = 3.5;
```

Existing C programs with keyword clashes can be converted by prefixing the keyword identifiers with double backquotes, and eventually the identifier name can be changed to a non-keyword name. Figure 1 shows how clashes in existing C header-files (see Section D, p. 73) can be handled using preprocessor *interposition*: **#include_next** and command-line -I filename. Several common C header-files with keyword clashes are fixed in the standard C∀ header-library, so there is largely a seamless programming-experience.

7 Constant Underscores

Numeric constants are extended to allow underscores as a separator, *e.g.*:

```
2_147_483_648;                            // decimal constant
56_ul;                                    // decimal unsigned long constant
0_377;                                    // octal constant
0x_ff_ff;                                 // hexadecimal constant
0x_ef3d_aa5c;                             // hexadecimal constant
3.141_592_654;                            // floating constant
10_e_+1_00;                               // floating constant
0x_ff_ff_p_3;                             // hexadecimal floating
0x_1.ffff_ffff_p_128_l;                   // hexadecimal floating long constant
L_"\x_ff_ee";                             // wide character constant
```

The rules for placement of underscores are:

1. A sequence of underscores is disallowed, *e.g.*, 12__34 is invalid.

2. Underscores may only appear within a sequence of digits (regardless of the digit radix). In other words, an underscore cannot start or end a sequence of digits, *e.g.*, _1, 1_ and _1_ are invalid (actually, the 1st and 3rd examples are identifier names).
3. A numeric prefix may end with an underscore; a numeric infix may begin and/or end with an underscore; a numeric suffix may begin with an underscore. For example, the octal 0 or hexadecimal 0x prefix may end with an underscore 0_377 or 0x_ff; the exponent infix E may start or end with an underscore 1.0_E10, 1.0E_10 or 1.0_E_10; the type suffixes U, L, *etc*. may start with an underscore 1_U, 1_ll or 1.0E10_f.

It is significantly easier to read and enter long constants when they are broken up into smaller groupings (most cultures use comma and/or period among digits for the same purpose). This extension is backwards compatible, matches with the use of underscore in variable names, and appears in Ada and Java 8. C++ uses the single quote (') as a separator, restricted within a sequence of digits, *e.g.*, 0xaa'ff, 3.141'592E1'1. However, the drawback of the C++ approach is difficults parsing for IDEs between character and numeric constants, as quotes are no longer balanced ('x' and 3.14'159).

8 Exponentiation Operator

C, C++, and Java (and other programming languages) have *no* exponentiation operator, *i.e.*, x^y, and instead use a routine, like pow(x,y), to perform the exponentiation operation. C∀ extends the basic operators with the exponentiation operator ?\? and ?\=?, as in, x \ y and x \= y, which means x^y and $x \leftarrow x^y$. The priority of the exponentiation operator is between the cast and multiplicative operators, so that w * (**int**)x \ (**int**)y * z is parenthesized as (w * (((**int**)x) \ ((**int**)y))) * z.

There are exponentiation operators for integral and floating types, including the builtin complex types. Integral exponentiation is performed with repeated multiplication ($O(\log y)$ or shifting if the exponent is 2). Overflow for a large exponent or negative exponent returns zero. Floating exponentiation is performed using logarithms, so the exponent cannot be negative.

```
sout | 1 \ 0 | 1 \ 1 | 2 \ 8 | −4 \ 3 | 5 \ 3 | 5 \ 32 | 5L \ 32 | 5L \ 64 | −4 \ −3 | −4.0 \ −3 | 4.0 \ 2.1 | (1.0f+2.0fi) \ (3.0f+2.0fi);
1 1 256 −64 125 0 3273344365508751233 0 0 −0.015625 18.3791736799526 0.264715−1.1922i
```

Note, 5 \ 32 and 5L \ 64 overflow, and −4 \ −3 is a fraction but stored in an integer so all three computations generate an integral zero. Because exponentiation has higher priority than +, parenthesis are necessary for exponentiation of complex constants or the expression is parsed as 1.0f+(2.0fi \ 3.0f)+2.0fi, requiring (1.0f+2.0fi) \ (3.0f+2.0fi).

The exponentiation operator is available for all the basic types, but for user-defined types, only the integral-computation version is available.

```
forall( T | { void ?{}( T & this, one_t ); T ?*?( T, T ); } )
T ?\?( T ep, unsigned int y );
forall( T | { void ?{}( T & this, one_t ); T ?*?( T, T ); } )
T ?\?( T ep, unsigned long int y );
```

A user type T must define one (1), and multiplication (*) (see Section 26.4, p. 64).

9 Control Structures

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

9.1 if / while Statement

The **if** and **while** expressions are extended with declarations, similar to the **for** declaration expression.[4]

```
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
if ( struct S { int i; } x = { f() }; x.i < 4 ) // relational expression

while ( int x = f() ) ...                       // x != 0
while ( int x = f(), y = g() ) ...              // x != 0 && y != 0
```

[4]Declarations in the **do**-**while** condition are not useful because they appear after the loop body.

```
while ( int x = f(), y = g(); x < y ) ...       // relational expression
while ( struct S { int i; } x = { f() }; x.i < 4 ) ... // relational expression
```

Unless a relational expression is specified, each variable is compared not equal to 0, which is the standard semantics for the **if/while** expression, and the results are combined using the logical && operator. The scope of the declaration(s) is local to the **if/while** statement, *i.e.*, in both *then* and *else* clauses for **if**, and loop body for **while**. C++ only provides a single declaration always compared != to 0.

9.2 case Clause

C restricts the **case** clause in a **switch** statement to a single value. For multiple **case** clauses prefixing a statement within the **switch** statement, it is necessary to have multiple **case** clauses rather than multiple values. Requiring a **case** clause for each value is not in the spirit of brevity normally associated with C. Therefore, the **case** clause is extended with a list of values.

```
              C                           C∀
switch ( i ) {                    switch ( i ) {
  case 1: case 3 : case 5:          case 1, 3, 5:       // odd values
    ...                               ...
  case 2: case 4 : case 6:          case 2, 4, 6:       // even values
    ...                               ...
}                                 }
```

In addition, subranges are allowed to specify a contiguous set of case values.

```
                   C                                  C∀              gcc
switch ( i ) {                                  switch ( i ) {   switch ( i ) {
  case 1: case 2: case 3: case 4:                 case 1~4:        case 1␣...4:      // 1, 2, 3, 4
    ...                                             ...              ...
  case 10: case 11: case 12: case 13:             case 10~13:      case 10␣...13:    // 10, 11, 12, 13
    ...                                             ...              ...
}                                               }                }
```

While gcc has the same range mechanism, it has an awkward syntax, 2␣...42, because a space is required after a number, otherwise the period is a decimal point.

C∀ also allows lists of subranges.

```
case 1~5, 12~21, 35~42:
```

9.3 switch Statement

C allows a number of questionable forms for the **switch** statement:

1. By default, the end of a **case** clause[5] *falls through* to the next **case** clause in the **switch** statement; to exit a **switch** statement from a **case** clause requires explicitly terminating the clause with a transfer statement, most commonly **break**:

   ```
   switch ( i ) {
     case 1:
       ...
       // fall-through
     case 2:
       ...
       break;    // exit switch statement
   }
   ```

   The ability to fall-through to the next clause *is* a useful form of control flow, specifically when a sequence of case actions compound:

[5] In this section, the term *case clause* refers to either a **case** or **default** clause.

```
switch ( argc ) {
  case 3:                                 if ( argc == 3 ) {
    // open output file                       // open output file
    // fall−through                           // open input file
  case 2:                                 } else if ( argc == 2 ) {
    // open input file                        // open input file (duplicate)
    break;   // exit switch statement
  default:                                } else {
    // usage message                          // usage message
}                                         }
```

In this example, case 2 is always done if case 3 is done. This control flow is difficult to simulate with **if** statements or a **switch** statement without fall-through as code must be duplicated or placed in a separate routine. C also uses fall-through to handle multiple case-values resulting in the same action:

```
switch ( i ) {
  case 1: case 3: case 5:   // odd values
    // odd action
    break;
  case 2: case 4: case 6:   // even values
    // even action
    break;
}
```

This situation is better handled by a list of case values (see Section 9.2).

While fall-through itself is not a problem, the problem occurs when fall-through is the default, as this semantics is unintuitive for many programmers and is different from most programming languages with a **switch** statement. Hence, default fall-through semantics results in programming errors as programmers often *forget* the **break** statement at the end of a **case** clause, resulting in inadvertent fall-through.

2. It is possible to place **case** clauses on statements nested *within* the body of the **switch** statement:

```
switch ( i ) {
  case 0:
    if ( j < k ) {
      ...
     case 1:     // transfer into "if" statement
      ...
    } // if
```

This usage branches into control structures, which is known to cause both comprehension and technical difficulties. The comprehension problem results from the inability to determine how control reaches a particular point due to the number of branches leading to it. The technical problem results from the inability to ensure declaration and initialization of variables when blocks are not entered at the beginning. There are few arguments for this kind of control flow, and therefore, there is a strong impetus to eliminate it. This C idiom is known as "Duff's device" [10], from this example:

```
register int n = (count + 7) / 8;
switch ( count % 8 ) {
case 0: do{ *to = *from++;
case 7:     *to = *from++;
case 6:     *to = *from++;
case 5:     *to = *from++;
case 4:     *to = *from++;
case 3:     *to = *from++;
case 2:     *to = *from++;
case 1:     *to = *from++;
        } while ( −−n > 0 );
}
```

which unrolls a loop N times (N = 8 above) and uses the **switch** statement to deal with any iterations not a multiple of N. While efficient, this sort of special purpose usage is questionable:

Disgusting, no? But it compiles and runs just fine. I feel a combination of pride and revulsion at this discovery. [10]

3. It is possible to place the **default** clause anywhere in the list of labelled clauses for a **switch** statement, rather than only at the end. Most programming languages with a **switch** statement require the **default** clause to appear last in the case-clause list. The logic for this semantics is that after checking all the **case** clauses without success, the **default** clause is selected; hence, physically placing the **default** clause at the end of the **case** clause list matches with this semantics. This physical placement can be compared to the physical placement of an **else** clause at the end of a series of connected **if/else** statements.
4. It is possible to place unreachable code at the start of a **switch** statement, as in:

```
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 uninitialized
}
```

While the declaration of the local variable y is useful with a scope across all **case** clauses, the initialization for such a variable is defined to never be executed because control always transfers over it. Furthermore, any statements before the first **case** clause can only be executed if labelled and transferred to using a **goto**, either from outside or inside of the **switch**, where both are problematic. As well, the declaration of z cannot occur after the **case** because a label can only be attached to a statement, and without a fall-through to case 3, z is uninitialized. The key observation is that the **switch** statement branches into a control structure, *i.e.*, there are multiple entry points into its statement body.

Before discussing potential language changes to deal with these problems, it is worth observing that in a typical C program:

- the number of **switch** statements is small,
- most **switch** statements are well formed (*i.e.*, no Duff's device),
- the **default** clause is usually written as the last case-clause,
- and there is only a medium amount of fall-through from one **case** clause to the next, and most of these result from a list of case values executing common code, rather than a sequence of case actions that compound.

These observations put into perspective the C∀ changes to the **switch** statement.

1. Eliminating default fall-through has the greatest potential for affecting existing code. However, even if fall-through is removed, most **switch** statements would continue to work because of the explicit transfers already present at the end of each **case** clause, the common placement of the **default** clause at the end of the case list, and the most common use of fall-through, *i.e.*, a list of **case** clauses executing common code, *e.g.*:

```
case 1:  case 2:  case 3: ...
```

still works. Nevertheless, reversing the default action would have a non-trivial effect on case actions that compound, such as the above example of processing shell arguments. Therefore, to preserve backwards compatibility, it is necessary to introduce a new kind of **switch** statement, called **choose**, with no implicit fall-through semantics and an explicit fall-through if the last statement of a case-clause ends with the new keyword **fallthrough**/**fallthru**, *e.g.*:

```
choose ( i ) {
  case 1:  case 2:  case 3:
    ...
    // implicit end of switch (break)
  case 5:
    ...
    fallthru;                      // explicit fall through
  case 7:
    ...
    break                          // explicit end of switch (redundant)
```

```
      default:
        j = 3;
    }
```

Like the **switch** statement, the **choose** statement retains the fall-through semantics for a list of **case** clauses. An implicit **break** is applied only at the end of the *statements* following a **case** clause. An explicit **fallthru** is retained because it is a C-idiom most C programmers expect, and its absence might discourage programmers from using the **choose** statement. As well, allowing an explicit **break** from the **choose** is a carry over from the **switch** statement, and expected by C programmers.

2. Duff's device is eliminated from both **switch** and **choose** statements, and only invalidates a small amount of very questionable code. Hence, the **case** clause must appear at the same nesting level as the **switch/choose** body, as is done in most other programming languages with **switch** statements.
3. The issue of **default** at locations other than at the end of the cause clause can be solved by using good programming style, and there are a few reasonable situations involving fall-through where the **default** clause needs to appear is locations other than at the end. Therefore, no change is made for this issue.
4. Dealing with unreachable code in a **switch/choose** body is solved by restricting declarations and initialization to the start of statement body, which is executed *before* the transfer to the appropriate **case** clause[6] and precluding statements before the first **case** clause. Further declarations at the same nesting level as the statement body are disallowed to ensure every transfer into the body is sound.

```
switch ( x ) {
    int i = 0;                  // allowed only at start
  case 0:
    ...
    int j = 0;                  // disallowed
  case 1:
    {
        int k = 0;              // allowed at different nesting levels
        ...
      case 2:                   // disallow case in nested statements
    }
  ...
}
```

9.4 Non-terminating and Labelled fallthrough

The **fallthrough** clause may be non-terminating within a **case** clause or have a target label to common code from multiple case clauses.

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

```
choose ( ... ) {
  case 3:
    ... fallthrough common;
  case 4:
    ... fallthrough common;

  common: // below fallthrough
              // at case-clause level
    ...  // common code for cases 3/4
    // implicit break
  case 4:
```

```
choose ( ... ) {
  case 3:
    choose ( ... ) {
      case 4:
        for ( ... ) {
            // multi-level transfer
            ... fallthru common;
        }
        ...
    }
    ...
  common: // below fallthrough
              // at case-clause level
```

The target label must be below the **fallthrough** and may not be nested in a control structure, 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.

[6]Essentially, these declarations are hoisted before the **switch/choose** statement and both declarations and statement are surrounded by a compound statement.

9.5 Loop Control

Looping a fixed number of times, possibly with a loop index, occurs frequently. C∀ condenses simply looping to facilitate coding speed and safety (see examples in Figure 2).

The **for**, **while**, and **do** loop-control are extended to allow an empty conditional, which implies a comparison value of 1 (true).

```
while ( /* empty */ )               // while ( true )
for ( /* empty */ )                 // for ( ; true; )
do ... while ( /* empty */ )        // do ... while ( true )
```

The **for** control is extended with 4 new loop-control operators, which are not overloadable:

~ (tilde) up-to exclusive range,
~= (tilde equal) up-to inclusive range,
-~ (minus tilde) down-to exclusive range,
-~= (minus tilde equal) down-to inclusive range.

The **for** index components, low (L), high (H), and increment (I), are optional. If missing:

- 0 is the implicit low value.
- 1 is the implicit increment value.
- The up-to range uses operator += for increment.
- The down-to range uses operator -= for decrement.

If no type is specified for the loop index, it is the type of the high value H (when the low value is implicit) or the low value L.

```
for ( 5 )                           // typeof(5) anonymous-index; 5 is high value
for ( i; 1.5 ~ 5.5 )                // typeof(1.5) i; 1.5 is low value
for ( int i; 0 ~ 10 ~ 2 )           // int i; type is explicit
```

The following are examples for constructing different for-control:

- H is implicit up-to exclusive range [0,H).

```
    for ( 5 )                       // for ( typeof(5) i; i < 5; i += 1 )
```

- ~= H is implicit up-to inclusive range [0,H].

```
    for ( ~= 5 )                    // for ( typeof(5) i; i <= 5; i += 1 )
```

- L ~ H is explicit up-to exclusive range [L,H).

```
    for ( 1 ~ 5 )                   // for ( typeof(1) i = 1; i < 5; i += 1 )
```

- L ~= H is explicit up-to inclusive range [L,H].

```
    for ( 1 ~= 5 )                  // for ( typeof(1) i = 1; i <= 5; i += 1 )
```

- L -~ H is explicit down-to exclusive range [H,L), where L and H are implicitly interchanged to make the range down-to.

```
    for ( 1 -~ 5 )                  // for ( typeof(1) i = 5; i > 0; i -= 1 )
```

- L -~= H is explicit down-to inclusive range [H,L], where L and H are implicitly interchanged to make the range down-to.

```
    for ( 1 -~= 5 )                 // for ( typeof(1) i = 5; i >= 0; i -= 1 )
```

- @ means put nothing in this field.

```
    for ( i; 1 ~ @ ~ 2 )            // for ( typeof(1) i = 1; /* empty */; i += 2 )
    for ( i; 1 ~ 10 ~ @ )           // for ( typeof(1) i = 1; i < 10; /* empty */ )
    for ( i; 1 ~ @ ~ @ )            // for ( typeof(1) i = 1; /*empty*/; /* empty */ )
```

  L cannot be elided for the up-to range, @ ~ 5, and H for the down-to range, 1 -~ @, because then the loop index is uninitialized. Similarly, the high value cannot be elided is an anonymous loop index (1 ~ @), as there is no index to stop the loop.

- : means add another index.

```
    for ( i; 5 : j; 2 ~ 12 ~ 3 )    // for ( typeof(i) i = 1, j = 2; i < 5 && j < 12; i += 1, j += 3 )
```

Warning: specifying the down-to range maybe unexpected because the loop control *implicitly* switches the L and H

loop control	output
while () { sout | "empty"; **break**; }	empty
do { sout | "empty"; **break**; } **while** ();	empty
for () { sout | "empty"; **break**; } *sout | nl | nlOff;*	empty
for (0) { sout | "A"; } sout | "zero"; *sout | nl;*	zero
for (1) { sout | "A"; } *sout | nl;*	A
for (10) { sout | "A"; } *sout | nl;*	A A A A A A A A A A
for (~= 10) { sout | "A"; } *sout | nl;*	A A A A A A A A A A A
for (1 ~= 10 ~ 2) { sout | "B"; } *sout | nl;*	B B B B B
for (1 -~= 10 ~ 2) { sout | "C"; } *sout | nl;*	C C C C C
for (0.5 ~ 5.5) { sout | "D"; } *sout | nl;*	D D D D D
for (0.5 -~ 5.5) { sout | "E"; } *sout | nl;*	E E E E E
for (i; 10) { sout | i; } *sout | nl;*	0 1 2 3 4 5 6 7 8 9
for (i; ~= 10) { sout | i; } *sout | nl;*	0 1 2 3 4 5 6 7 8 9 10
for (i; 1 ~= 10 ~ 2) { sout | i; } *sout | nl;*	1 3 5 7 9
for (i; 1 -~= 10 ~ 2) { sout | i; } *sout | nl;*	10 8 6 4 2
for (i; 0.5 ~ 5.5) { sout | i; } *sout | nl;*	0.5 1.5 2.5 3.5 4.5
for (i; 0.5 -~ 5.5) { sout | i; } *sout | nl;*	5.5 4.5 3.5 2.5 1.5
for (ui; 2u ~= 10u ~ 2u) { sout | ui; } *sout | nl;*	2 4 6 8 10
for (ui; 2u -~= 10u ~ 2u) { sout | ui; } *sout | nl | nl | nl;*	10 8 6 4 2
enum { N = 10 };	
for (N) { sout | "N"; } *sout | nl;*	N N N N N N N N N N
for (i; N) { sout | i; } *sout | nl;*	0 1 2 3 4 5 6 7 8 9
for (i; -~ N) { sout | i; } *sout | nl | nl | nl;*	10 9 8 7 6 5 4 3 2 1
const int low = 3, high = 10, inc = 2;	
for (i; low ~ high ~ inc + 1) { sout | i; } *sout | nl;*	3 6 9
for (i; 1 ~ @) { **if** (i > 10) **break**; sout | i; } *sout | nl;*	1 2 3 4 5 6 7 8 9 10
for (i; @ -~ 10) { **if** (i < 0) **break**; sout | i; } *sout | nl;*	10 9 8 7 6 5 4 3 2 1 0
for (i; 2 ~ @ ~ 2) { **if** (i > 10) **break**; sout | i; } *sout | nl;*	2 4 6 8 10
for (i; 2.1 ~ @ ~ @) { **if** (i > 10.5) **break**; sout | i; i += 1.7; } *sout | nl;*	2.1 3.8 5.5 7.2 8.9
for (i; @ -~ 10 ~ 2) { **if** (i < 0) **break**; sout | i; } *sout | nl;*	10 8 6 4 2 0
for (i; 12.1 ~ @ ~ @) { **if** (i < 2.5) **break**; sout | i; i -= 1.7; } *sout | nl;*	12.1 10.4 8.7 7. 5.3 3.6
for (i; 5 : j; -5 ~ @) { sout | i | j; } *sout | nl;*	0 -5 1 -4 2 -3 3 -2 4 -1
for (i; 5 : j; @ -~ -5) { sout | i | j; } *sout | nl;*	0 -5 1 -6 2 -7 3 -8 4 -9
for (i; 5 : j; -5 ~ @ ~ 2) { sout | i | j; } *sout | nl;*	0 -5 1 -3 2 -1 3 1 4 3
for (i; 5 : j; @ -~ -5 ~ 2) { sout | i | j; } *sout | nl;*	0 -5 1 -7 2 -9 3 -11 4 -13
for (i; 5 : j; -5 ~ @) { sout | i | j; } *sout | nl;*	0 -5 1 -4 2 -3 3 -2 4 -1
for (i; 5 : j; @ -~ -5) { sout | i | j; } *sout | nl;*	0 -5 1 -6 2 -7 3 -8 4 -9
for (i; 5 : j; -5 ~ @ ~ 2) { sout | i | j; } *sout | nl;*	0 -5 1 -3 2 -1 3 1 4 3
for (i; 5 : j; @ -~ -5 ~ 2) { sout | i | j; } *sout | nl;*	0 -5 1 -7 2 -9 3 -11 4 -13
for (i; 5 : j; @ -~ -5 ~ 2 : k; 1.5 ~ @) { sout | i | j | k; } *sout | nl;*	0 -5 1.5 1 -7 2.5 2 -9 3.5 3 -11 4.5 4 -13 5.5
for (i; 5 : j; @ -~ -5 ~ 2 : k; 1.5 ~ @) { sout | i | j | k; } *sout | nl;*	0 -5 1.5 1 -7 2.5 2 -9 3.5 3 -11 4.5 4 -13 5.5
for (i; 5 : k; 1.5 ~ @ : j; @ -~ -5 ~ 2) { sout | i | j | k; } *sout | nl;*	0 -5 1.5 1 -7 2.5 2 -9 3.5 3 -11 4.5 4 -13 5.5

Figure 2: Loop Control Examples

```
{

    ForC: for ( ... ) {
        WhileC: while ( ... ) {
            DoC: do {
                if ( ... ) {
                    switch ( ... ) {
                        case 3:
                            goto Compound;
                            goto Try;
                            goto ForB;      /* or */  goto ForC;
                            goto WhileB;  /* or */  goto WhileC;
                            goto DoB;       /* or */  goto DoC;
                            goto If;
                            goto Switch;
                    } Switch: ;
                } else {
                    ... goto If; ...    // terminate if
                } If:;
            } while ( ... ); DoB: ;
        } WhileB: ;
    } ForB: ;


} Compound: ;
```

a) C

```
Compound: {
    Try: try {
        For: for ( ... ) {
            While: while ( ... ) {
                Do: do {
                    If: if ( ... ) {
                        Switch: switch ( ... ) {
                            case 3:
                                break Compound;
                                break Try;
                                break For;      /* or */  continue For;
                                break While;  /* or */  continue While;
                                break Do;       /* or */  continue Do;
                                break If;
                                break Switch;
                        } // switch
                    } else {
                        ... break If; ...    // terminate if
                    } // if
                } while ( ... ); // do
            } // while
        } // for
    } finally { // always executed
    } // try
} // compound
```

b) C∀

Figure 3: Multi-level Exit

values (and toggles the increment/decrement for I):

```
    for ( i; 1 ~ 10 )                    // up range
    for ( i; 1 -~ 10 )                   // down range
    for ( i; 10 -~ 1 )                   // WRONG down range!
```

The reason for this semantics is that the range direction can be toggled by adding/removing the minus, `'-'`, versus interchanging the L and H expressions, which has a greater chance of introducing errors.

9.6 Labelled **continue** / **break** Statement

C **continue** and **break** statements are restricted to one level of nesting for a particular control structure. 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 the **continue** and **break** with a target label to support static multi-level exit [4], 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 a comparison between labelled **continue** and **break** and the corresponding C equivalent using **goto** and labels. The innermost loop has 8 exit points, which cause continuation or termination of one or more of the 7 nested control-structures.

Both labelled **continue** and **break** are a **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 undefined behaviour.

The advantage of the labelled **continue**/**break** is allowing static multi-level 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 via a label. Furthermore, the location of the label at the *beginning* of the target control structure informs the reader (eye candy) that complex control-flow is 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 existing constructs. Otherwise, the implicit targets of the current **continue** and **break**, *i.e.*, the closest enclosing loop or **switch**, change as certain constructs are added or removed.

9.7 Extended else

The **if** statement has an optional **else** clause executed if the conditional is false. This concept is extended to the **while**, **for**, and **do** looping constructs (like Python). Hence, if the loop conditional becomes false, looping stops and the corresponding **else** clause is executed, if present.

The following example is a linear search for the key 3 in an array, where finding the key is handled with a **break** and not finding with the **else** clause on the loop construct.

```
int a[10];

                                                                  int i = 0;
while ( int i = 0; i < 10 ) {            for ( i; 10 ) {          do {
  if ( a[i] == 3 ) break; // found         if ( a[i] == 3 ) break; // found    if ( a[i] == 3 ) break; // found
     i += 1;                                                          i += 1;
} else { // i == 10                      } else { // i == 10      } while( i < 10 ) else { // i == 10
   sout | "not found";                      sout | "not found";      sout | "not found";
}                                        }                        }
```

Note, dangling else now occurs with **if**, **while**, **for**, **do**, and **waitfor**.

9.8 with Statement

Grouping heterogeneous data into an *aggregate* (structure/union) is a common programming practice, and aggregates may be nested:

```
struct Person {                          // aggregate
    struct Name {                        // nesting
        char first[20], last[20];
    } name;
    struct Address {                     // nesting
        ...
    } address;
    int sex;
};
```

Functions manipulating aggregates must repeat the aggregate name to access its containing fields.

```
Person p
p.name ...; p.address ...; p.sex ...;    // access containing fields
```

which extends to multiple levels of qualification for nested aggregates and multiple aggregates.

```
struct Ticket { ... } t;
p.name.first ...; p.address.street ...;  // access nested fields
t.departure ...; t.cost ...;             // access multiple aggregate
```

Repeated aggregate qualification is tedious and makes code difficult to read. Therefore, reducing aggregate qualification is a useful language design goal.

C partially addresses the problem by eliminating qualification for enumerated types and unnamed *nested* aggregates, which open their scope into the containing aggregate. This feature is used to group fields for attributes and/or with **union** aggregates.

```
struct S {
    struct /* unnamed */ { int g, h; } __attribute__(( aligned(64) ));
    int tag;
    union /* unnamed */ {
        struct { char c1, c2; } __attribute__(( aligned(128) ));
        struct { int i1, i2; };
        struct { double d1, d2; };
    };
} s;
enum { R, G, B };
s.g; s.h;   s.tag = R;   s.c1; s.c2;   s.i1 = G; s.i2 = B;   s.d1; s.d2;
```

Object-oriented languages reduce qualification for class variables within member functions, *e.g.*, C++:

```
struct S {
    char c;   int i;   double d;
    void f( /* S * this */ ) {              // implicit "this" parameter
        c;   i;   d;                         // this->c; this->i; this->d;
    }
}
```

In general, qualification is elided for the variables and functions in the lexical scopes visible from a member function. However, qualification is necessary for name shadowing and explicit aggregate parameters.

```
struct T {
    char m;   int i;   double n;            // derived class variables
};
struct S : public T {
    char c;   int i;   double d;            // class variables
    void g( double d, T & t ) {
        d;   t.m;   t.i;   t.n;             // function parameter
        c;   i;   this->d;   S::d;          // class S variables
        m;   T::i;   n;                     // class T variables
    }
};
```

Note the three different forms of qualification syntax in C++, ., −>, ::, which is confusing.

Since C∀ in not object-oriented, it has no implicit parameter with its implicit qualification. Instead C∀ introduces a general mechanism using the **with** statement (see Pascal [23, § 4.F]) to explicitly elide aggregate qualification by opening a scope containing the field identifiers. Hence, the qualified fields become variables with the side-effect that it is simpler to write, easier to read, and optimize field references in a block.

```
void f( S & this ) with ( this ) {          // with statement
    c;   i;   d;                             // this.c, this.i, this.d
}
```

with the generality of opening multiple aggregate-parameters:

```
void g( S & s, T & t ) with ( s, t ) {      // multiple aggregate parameters
    c;   s.i;   d;                           // s.c, s.i, s.d
    m;   t.i;   n;                           // t.m, t.i, t.n
}
```

where qualification is only necessary to disambiguate the shadowed variable i. In detail, the **with** statement may form a function body or be nested within a function body.

The **with** clause takes a list of expressions, where each expression provides an aggregate type and object. (Enumerations are already opened.) To open a pointer type, the pointer must be dereferenced to obtain a reference to the aggregate type.

```
S * sp;
with ( *sp ) { ... }
```

The expression object is the implicit qualifier for the open structure-fields.

C∀'s ability to overload variables (see Section 26.2, p. 63) and use the left-side of assignment in type resolution means most fields with the same name but different types are automatically disambiguated, eliminating qualification. All expressions in the expression list are open in parallel within the compound statement. This semantic is different from Pascal, which nests the openings from left to right. The difference between parallel and nesting occurs for fields with the same name and type:

```
struct Q { int i; int k; int m; } q, w;
struct R { int i; int j; double m; } r, w;
with ( r, q ) {
    j + k;                                   // unambiguous, r.j + q.k
    m = 5.0;                                 // unambiguous, q.m = 5.0
```

```
        m = 1;                          // unambiguous, r.m = 1
        int a = m;                      // unambiguous, a = r.i
        double b = m;                   // unambiguous, b = q.m
        int c = r.i + q.i;              // disambiguate with qualification
        (double)m;                      // disambiguate with cast
    }
```

For parallel semantics, both r.i and q.i are visible, so i is ambiguous without qualification; for nested semantics, q.i hides r.i, so i implies q.i. Pascal nested-semantics is possible by nesting **with** statements.

```
    with ( r ) {
        i;                              // unambiguous, r.i
        with ( q ) {
            i;                          // unambiguous, q.i
        }
    }
```

A cast or qualification can be used to disambiguate variables within a **with** *statement*. A cast can also be used to disambiguate among overload variables in a **with** *expression*:

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

Because there is no left-side in the **with** expression to implicitly disambiguate between the w variables, it is necessary to explicitly disambiguate by casting w to type Q or R.

Finally, there is an interesting problem between parameters and the function-body **with**, *e.g.*:

```
    void f( S & s, char c ) with ( s ) {
        s.c = c;  i = 3;  d = 5.5;      // initialize fields
    }
```

Here, the assignment s.c = c means s.c = s.c, which is meaningless, and there is no mechanism to qualify the parameter c, making the assignment impossible using the function-body **with**. To solve this problem, parameters *not* explicitly opened are treated like an initialized aggregate:

```
    struct Params {                     // s explicitly opened so S & s elided
        char c;
    } params;
```

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

```
    void f( S & s, char c ) with ( s ) with( params ) { // syntax not allowed, illustration only
        s.c = c;  i = 3;  d = 5.5;
    }
```

This implicit semantic matches with programmer expectation.

10 Exception Handling

Exception handling provides two mechanism: change of control flow from a raise to a handler, and communication from the raise to the handler. Transfer of control can be local, within a routine, or non-local, among routines. Non-local transfer can cause stack unwinding, *i.e.*, non-local routine termination, depending on the kind of raise.

Currently, C∀ uses macros ExceptionDecl and ExceptionInst to declare and instantiate an exception.

```
    #include <Exception.hfa>
    ExceptionDecl( E,       // must be global scope
        ... // exception fields
    );
    try {
        ...
        if ( ... ) throwResume ExceptionInst( E, /* intialization */ );
        if ( ... ) throw ExceptionInst( E, /* intialization */ );
        ...
    } catchResume( E * ) { // must be pointer
        ...
    } catch( E * ) {
        ...
```

```
}
exception_t E {};                         // exception type
void f(...) {
    ... throw E{}; ...                    // termination
    ... throwResume E{}; ...              // resumption
}
try {
    f(...);
} catch( E e ; boolean-predicate ) {      // termination handler
    // recover and continue
} catchResume( E e ; boolean-predicate ) { // resumption handler
    // repair and return
} finally {
    // always executed
}
```

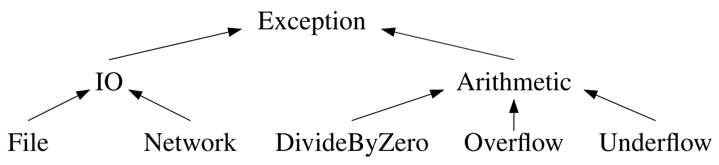
The kind of raise and handler match: **throw** with **catch** and **throwResume** with **catchResume**. Then the exception type must match along with any additional predicate must be true. The **catch** and **catchResume** handlers may appear in any oder. However, the **finally** clause must appear at the end of the **try** statement.

10.1 Non-local Exception

```
void main() {
    try {
        _Enable {
            ... resume(); ...
        }
    } catchResume( E & ) { // should be reference
        ...
    } catch( E & ) {
        ...
    }
}
```

10.2 Exception Hierarchy

An exception type can be derived from another exception type, just like deriving a subclass from a class, providing a kind of polymorphism among exception types. The exception-type hierarchy that is created is used to organize exception types, similar to a class hierarchy in object-oriented languages, *e.g.*:



A programmer can then choose to handle an exception at different degrees of specificity along the hierarchy; derived exception-types support a more flexible programming style. For example, higher-level code should catch general exceptions to reduce coupling to the specific implementation at the lower levels; unnecessary coupling may force changes in higher-level code when low-level code changes. A consequence of derived exception-types is that multiple exceptions may match, *e.g.*:

```
catch( Arithmetic )
```

matches all three derived exception-types: DivideByZero, Overflow, and Underflow. Because the propagation mechanisms perform a simple linear search of the handler clause for a guarded block, and selects the first matching handler, the order of catch clauses in the handler clause becomes important, *e.g.*:

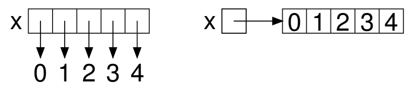
```
try {
    ...
} catch( Overflow ) {   // must appear first
    // handle overflow
} catch( Arithmetic )
    // handle other arithmetic issues
}
```

Multiple derivation among exception is not supported.

11 Alternative Declarations

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

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



Is this an array of 5 pointers to integers or a pointer to an array of 5 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 routine name and its parameters are embedded within the return type, mimicking the way the return value is used at the routine's call site. For example, a routine 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 routine 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, even though Dennis Richie believed otherwise:

> In spite of its difficulties, I believe that the C's approach to declarations remains plausible, and am comfortable with it; it is a useful unifying principle. [27, p. 12]

C∀ provides its own type, variable and routine declarations, using a different syntax. The new declarations place qualifiers to the left of the base type, while C declarations place qualifiers to the right of the base type. In the following example, red is the base type and **blue** is qualifiers. The C∀ declarations move the qualifiers to the left of the base type, *i.e.*, move the blue to the left of the red, while the qualifiers have the same meaning but are ordered left to right to specify a variable's type.

C	C∀
`int * x1 [5];`	`[5] * int x1;`
`int (*x2)[5];`	`* [5] int x2;`
`int (*f( int p ))[5];`	`[* [5] int] f( int p );`

The only exception is bit field specification, which always appear 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 follows:

C	C∀
`int *x, *y;`	`* int x, y;`

The downside of this semantics is the need to separate regular and pointer declarations:

C	C∀

`int *x, y;`	`* int x;`
	`int y;`

which is prescribing a safety benefit. Other examples are:

```
        C                      C∀
int z[ 5 ];              [ 5 ] int z;         // array of 5 integers
char * w[ 5 ];           [ 5 ] * char w;      // array of 5 pointers to char
double (* v)[ 5 ];       * [ 5 ] double v;    // pointer to array of 5 doubles
struct s {               struct s {
    int f0:3;                int f0:3;        // common bit field syntax
    int * f1;                * int f1;
    int * f2[ 5 ]            [ 5 ] * int f2;
};                       };
```

All type qualifiers, *e.g.*, **const**, **volatile**, *etc.*, are used in the normal way with the new declarations and also appear left to right, *e.g.*:

```
            C                         C∀
int const * const x;      const * const int x;        // const pointer to const integer
const int (* const y)[ 5 ]  const * [ 5 ] const int y;  // const pointer to array of 5 const integers
```

All declaration qualifiers, *e.g.*, **extern**, **static**, *etc.*, are used in the normal way with the new declarations but can only appear at the start of a C∀ routine declaration,[7] *e.g.*:

```
          C                        C∀
int extern x[ 5 ];          extern [ 5 ] int x;       // externally visible array of 5 integers
const int static * y;       static * const int y;     // internally visible pointer to constant int
```

The new declaration syntax can be used in other contexts where types are required, *e.g.*, casts and the pseudo-routine **sizeof**:

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

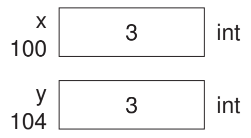
Finally, new C∀ declarations may appear together with C declarations in the same program block, but cannot be mixed within a specific declaration. Therefore, a programmer has the option of either continuing to use traditional C declarations or take advantage of the new style. Clearly, both styles need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

12 Pointer / Reference

C provides a *pointer type*; C∀ adds a *reference type*. These types may be derived from an object or routine type, called the *referenced type*. Objects of these types contain an *address*, which is normally a location in memory, but may also address memory-mapped registers in hardware devices. An integer constant expression with the value 0, or such an expression cast to type **void** *, is called a *null-pointer constant*.[8] An address is *sound*, if it points to a valid memory location in scope, *i.e.*, within the program's execution-environment and has not been freed. Dereferencing an *unsound* address, including the null pointer, is undefined, often resulting in a memory fault.

A program *object* is a region of data storage in the execution environment, the contents of which can represent values. In most cases, objects are located in memory at an address, and the variable name for an object is an implicit address to the object generated by the compiler and automatically dereferenced, as in:

```
int x;
x = 3;
int y;
y = x;
```



```
int * const x = (int *)100
*x = 3;          // implicit dereference
int * const y = (int *)104;
*y = *x;         // implicit dereference
```

where the right example is how the compiler logically interprets the variables in the left example. Since a variable name only points to one address during its lifetime, it is an immutable pointer; hence, the implicit type of pointer variables x and y are constant pointers in the compiler interpretation. In general, variable addresses are stored in

[7] The placement of a storage-class specifier other than at the beginning of the declaration specifiers in a declaration is an obsolescent feature. [21, § 6.11.5(1)]

[8] One way to conceptualize the null pointer is that no variable is placed at this address, so the null-pointer address can be used to denote an uninitialized pointer/reference object; *i.e.*, the null pointer is guaranteed to compare unequal to a pointer to any object or routine. In general, a value with special meaning among a set of values is called a *sentinel value*, *e.g.*, −1 as a return code value.

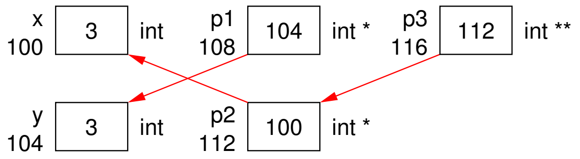
instructions instead of loaded from memory, and hence may not occupy storage. These approaches are contrasted in the following:

explicit variable address	implicit variable address
lda r1,100 // load address of x	
ld r2,(r1) // load value of x	ld r2,(100) // load value of x
lda r3,104 // load address of y	
st r2,(r3) // store x into y	st r2,(104) // store x into y

Finally, the immutable nature of a variable's address and the fact that there is no storage for the variable pointer means pointer assignment is impossible. Therefore, the expression x = y has only one meaning, *x = *y, *i.e.*, manipulate values, which is why explicitly writing the dereferences is unnecessary even though it occurs implicitly as part of instruction decoding.

A pointer/reference object is a generalization of an object variable-name, *i.e.*, a mutable address that can point to more than one memory location during its lifetime. (Similarly, an integer variable can contain multiple integer literals during its lifetime versus an integer constant representing a single literal during its lifetime, and like a variable name, may not occupy storage if the literal is embedded directly into instructions.) Hence, a pointer occupies memory to store its current address, and the pointer's value is loaded by dereferencing, *e.g.*:

```
int x, y, * p1, * p2, ** p3;
p1 = &x;     // p1 points to x
p2 = p1;     // p2 points to x
p1 = &y;     // p1 points to y
p3 = &p2;  // p3 points to p2
```



Notice, an address has a duality: a location in memory or the value at that location. In many cases, a compiler might be able to infer the best meaning for these two cases. For example, Algol68 [24] infers pointer dereferencing to select the best meaning for each pointer usage

```
p2 = p1 + x;                        // compiler infers *p2 = *p1 + x;
```

Algol68 infers the following dereferencing *p2 = *p1 + x, because adding the arbitrary integer value in x to the address of p1 and storing the resulting address into p2 is an unlikely operation. Unfortunately, automatic dereferencing does not work in all cases, and so some mechanism is necessary to fix incorrect choices.

Rather than inferring dereference, most programming languages pick one implicit dereferencing semantics, and the programmer explicitly indicates the other to resolve address-duality. In C, objects of pointer type always manipulate the pointer object's address:

```
p1 = p2;                            // p1 = p2  rather than  *p1 = *p2
p2 = p1 + x;                        // p2 = p1 + x  rather than  *p2 = *p1 + x
```

even though the assignment to p2 is likely incorrect, and the programmer probably meant:

```
p1 = p2;                            // pointer address assignment
*p2 = *p1 + x;                      // pointed-to value assignment / operation
```

The C semantics work well for situations where manipulation of addresses is the primary meaning and data is rarely accessed, such as storage management (malloc/free).

However, in most other situations, the pointed-to value is requested more often than the pointer address.

```
*p2 = ((*p1 + *p2) * (**p3 - *p1)) / (**p3 - 15);
```

In this case, it is tedious to explicitly write the dereferencing, and error prone when pointer arithmetic is allowed. It is better to have the compiler generate the dereferencing and have no implicit pointer arithmetic:

```
p2 = ((p1 + p2) * (p3 - p1)) / (p3 - 15);
```

To support this common case, a reference type is introduced in C∀, denoted by &, which is the opposite dereference semantics to a pointer type, making the value at the pointed-to location the implicit semantics for dereferencing (similar but not the same as C++ reference types).

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

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

Except for auto-dereferencing by the compiler, this reference example is the same as the previous pointer example. Hence, a reference behaves like the variable name for the current variable it is pointing-to. 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 a declaration, so the previous example becomes:

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

When a reference operation appears beside a dereference operation, *e.g.*, &*, they cancel out. However, in C, the cancellation always yields a value (rvalue).[9] For a C∀ reference type, the cancellation on the left-hand side of assignment leaves the reference as an address (lvalue):

```
(&*)r1 = &x;                    // (&*) cancel giving address in r1 not variable pointed-to by r1
```

Similarly, the address of a reference can be obtained for assignment or computation (rvalue):

```
(&(&*)*)r3 = &(&*)r2;           // (&*) cancel giving address in r2, (&(&*)*) cancel giving address in r3
```

Cancellation works to arbitrary depth.

Fundamentally, pointer and reference objects are functionally interchangeable because both contain addresses.

```
int x, *p1 = &x, **p2 = &p1, ***p3 = &p2,
      &r1 = x,    &&r2 = r1,   &&&r3 = r2;
***p3 = 3;                      // change x
r3 = 3;                         // change x, ***r3
**p3 = ...;                     // change p1
&r3 = ...;                      // change r1, (&*)**r3, 1 cancellation
*p3 = ...;                      // change p2
&&r3 = ...;                     // change r2, (&(&*)*)*r3, 2 cancellations
&&&r3 = p3;                     // change r3 to p3, (&(&(&*)*)*)r3, 3 cancellations
```

Furthermore, both types are equally performant, as the same amount of dereferencing occurs for both types. Therefore, the choice between them is based solely on whether the address is dereferenced frequently or infrequently, which dictates the amount of implicit dereferencing aid from the compiler.

As for a pointer type, a reference type may have qualifiers:

```
const int cx = 5;               // cannot change cx;
const int & cr = cx;            // cannot change what cr points to
&cr = &cx;                      // can change cr
cr = 7;                         // error, cannot change cx
int & const rc = x;             // must be initialized
&rc = &x;                       // error, cannot change rc
const int & const crc = cx;     // must be initialized
crc = 7;                        // error, cannot change cx
&crc = &cx;                     // error, cannot change crc
```

Hence, for type & **const**, there is no pointer assignment, so &rc = &x is disallowed, and *the address value cannot be the null pointer unless an arbitrary pointer is coerced into the reference*:

```
int & const cr = *0;            // where 0 is the int * zero
```

Note, constant reference-types do not prevent addressing errors because of explicit storage-management:

```
int & const cr = *malloc();
cr = 5;
free( &cr );
cr = 7;                         // unsound pointer dereference
```

The position of the **const** qualifier *after* the pointer/reference qualifier causes confuse for C programmers. The **const** qualifier cannot be moved before the pointer/reference qualifier for C style-declarations; C∀-style declarations (see Section 11, p. 18) attempt to address this issue:

C	C∀

`const int * const * const ccp;`	`const * const * const int ccp;`
	`const & const & const int ccr;`

[9]The unary & operator yields the address of its operand. If the operand has type "type", the result has type "pointer to type". If the operand is the result of a unary * operator, neither that operator nor the & operator is evaluated and the result is as if both were omitted, except that the constraints on the operators still apply and the result is not an lvalue. [21, § 6.5.3.2–3]

where the C∀ declaration is read left-to-right.

Finally, like pointers, references are usable and composable with other type operators and generators.

```
int w, x, y, z, & ar[3] = { x, y, z };      // initialize array of references
&ar[1] = &w;                                 // change reference array element
typeof( ar[1] ) p;                           // (gcc) is int, i.e., the type of referenced object
typeof( &ar[1] ) q;                          // (gcc) is int &, i.e., the type of reference
sizeof( ar[1] ) == sizeof( int );            // is true, i.e., the size of referenced object
sizeof( &ar[1] ) == sizeof( int *)           // is true, i.e., the size of a reference
```

In contrast to C∀ reference types, C++'s reference types are all **const** references, preventing changes to the reference address, so only value assignment is possible, which eliminates half of the address duality. Also, C++ does not allow arrays of reference[10] Java's reference types to objects (all Java objects are on the heap) are like C pointers, which always manipulate the address, and there is no (bit-wise) object assignment, so objects are explicitly cloned by shallow or deep copying, which eliminates half of the address duality.

12.1 Initialization

Initialization is different than assignment because initialization occurs on the empty (uninitialized) storage on an object, while assignment occurs on possibly initialized storage of an object. There are three initialization contexts in C∀: declaration initialization, argument/parameter binding, return/temporary binding. Because the object being initialized has no value, there is only one meaningful semantics with respect to address duality: it must mean address as there is no pointed-to value. In contrast, the left-hand side of assignment has an address that has a duality. Therefore, for pointer/reference initialization, the initializing value must be an address not a value.

```
int * p = &x;                                // assign address of x
int * p = x;                                 // assign value of x
int & r = x;                                 // must have address of x
```

Like the previous example with C pointer-arithmetic, it is unlikely assigning the value of x into a pointer is meaningful (again, a warning is usually given). Therefore, for safety, this context requires an address, so it is superfluous to require explicitly taking the address of the initialization object, even though the type is incorrect. Note, this is strictly a convenience and safety feature for a programmer. Hence, C∀ allows r to be assigned x because it infers a reference for x, by implicitly inserting a address-of operator, &, and it is an error to put an & because the types no longer match due to the implicit dereference. Unfortunately, C allows p to be assigned with &x (address) or x (value), but most compilers warn about the latter assignment as being potentially incorrect. Similarly, when a reference type is used for a parameter/return type, the call-site argument does not require a reference operator for the same reason.

```
int & f( int & r );                          // reference parameter and return
z = f( x ) + f( y );                         // reference operator added, temporaries needed for call results
```

Within routine f, it is possible to change the argument by changing the corresponding parameter, and parameter r can be locally reassigned within f. Since operator routine ?+? takes its arguments by value, the references returned from f are used to initialize compiler generated temporaries with value semantics that copy from the references.

```
int temp1 = f( x ), temp2 = f( y );
z = temp1 + temp2;
```

This implicit referencing is crucial for reducing the syntactic burden for programmers when using references; otherwise references have the same syntactic burden as pointers in these contexts.

When a pointer/reference parameter has a **const** value (immutable), it is possible to pass literals and expressions.

```
void f( const int & cr );
void g( const int * cp );
f( 3 );            g( &3 );
f( x + y );        g( &(x + y) );
```

Here, the compiler passes the address to the literal 3 or the temporary for the expression x + y, knowing the argument cannot be changed through the parameter. The & before the constant/expression for the pointer-type parameter (g) is a C∀ extension necessary to type match and is a common requirement before a variable in C (*e.g.*, scanf). Importantly,

[10]The reason for disallowing arrays of reference is unknown, but possibly comes from references being ethereal (like a textual macro), and hence, replaceable by the referent object.

&3 may not be equal to &3, where the references occur across calls because the temporaries maybe different on each call.

C∀ *extends* this semantics to a mutable pointer/reference parameter, and the compiler implicitly creates the necessary temporary (copying the argument), which is subsequently pointed-to by the reference parameter and can be changed.[11]

```
void f( int & r );
void g( int * p );
f( 3 );          g( &3 );              // compiler implicit generates temporaries
f( x + y );      g( &(x + y) );        // compiler implicit generates temporaries
```

Essentially, there is an implicit rvalue to lvalue conversion in this case.[12] The implicit conversion allows seamless calls to any routine without having to explicitly name/copy the literal/expression to allow the call.

Finally, C handles routine objects in an inconsistent way. A routine object is both a pointer and a reference (particle and wave).

```
void f( int i );
void (* fp)( int );             // routine pointer
fp = f;                         // reference initialization
fp = &f;                        // pointer initialization
fp = *f;                        // reference initialization
fp(3);                          // reference invocation
(*fp)(3);                       // pointer invocation
```

While C's treatment of routine objects has similarity to inferring a reference type in initialization contexts, the examples are assignment not initialization, and all possible forms of assignment are possible (f, &f, *f) without regard for type. Instead, a routine object should be referenced by a **const** reference:

```
const void (& fr)( int ) = f;   // routine reference
fr = ...;                       // error, cannot change code
&fr = ...;                      // changing routine reference
fr( 3 );                        // reference call to f
(*fr)(3);                       // error, incorrect type
```

because the value of the routine object is a routine literal, *i.e.*, the routine code is normally immutable during execution.[13] C∀ allows this additional use of references for routine objects in an attempt to give a more consistent meaning for them.

12.2 Address-of Semantics

In C, &E is an rvalue for any expression E. C∀ extends the & (address-of) operator as follows:

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

The following example shows the first rule applied to different rvalue contexts:

```
int x, * px, ** ppx, *** pppx, **** ppppx;
int & rx = x, && rrx = rx, &&& rrrx = rrx ;
x = rrrx;                // rrrx is an lvalue with type int &&& (equivalent to x)
px = &rrrx;              // starting from rrrx, &rrrx is an rvalue with type int *&&& (&x)
ppx = &&rrrx;            // starting from &rrrx, &&rrrx is an rvalue with type int **&& (&rx)
pppx = &&&rrrx;          // starting from &&rrrx, &&&rrrx is an rvalue with type int ***& (&rrx)
ppppx = &&&&rrrx;        // starting from &&&rrrx, &&&&rrrx is an rvalue with type int **** (&rrrx)
```

The following example shows the second rule applied to different lvalue contexts:

```
int x, * px, ** ppx, *** pppx;
```

[11] If whole program analysis is possible, and shows the parameter is not assigned, *i.e.*, it is **const**, the temporary is unnecessary.

[12] This conversion attempts to address the *const hell* problem, when the innocent addition of a **const** qualifier causes a cascade of type failures, requiring an unknown number of additional **const** qualifiers, until it is discovered a **const** qualifier cannot be added and all the **const** qualifiers must be removed.

[13] Dynamic code rewriting is possible but only in special circumstances.

```
int & rx = x, && rrx = rx, &&& rrrx = rrx ;
rrrx = 2;                               // rrrx is an lvalue with type int &&& (equivalent to x)
&rrrx = px;                             // starting from rrrx, &rrrx is an rvalue with type int *&&& (rx)
&&rrrx = ppx;                           // starting from &rrrx, &&rrrx is an rvalue with type int **&& (rrx)
&&&rrrx = pppx;                         // starting from &&rrrx, &&&rrrx is an rvalue with type int ***& (rrrx)
```

12.3 Conversions

C provides a basic implicit conversion to simplify variable usage:

0. lvalue to rvalue conversion: cv T converts to T, which allows implicit variable dereferencing.

   ```
   int x;
   x + 1;                       // lvalue variable (int) converts to rvalue for expression
   ```

   An rvalue has no type qualifiers (cv), so the lvalue qualifiers are dropped.

C∀ provides three new implicit conversion for reference types to simplify reference usage.

1. reference to rvalue conversion: cv T & converts to T, which allows implicit reference dereferencing.

   ```
   int x, &r = x, f( int p );
   x = r + f( r );              // lvalue reference converts to rvalue
   ```

   An rvalue has no type qualifiers (cv), so the reference qualifiers are dropped.

2. lvalue to reference conversion: lvalue−type cv1 T converts to cv2 T &, which allows implicitly converting variables to references.

   ```
   int x, &r = x, f( int & p );     // lvalue variable (int) convert to reference (int &)
   f( x );                          // lvalue variable (int) convert to reference (int &)
   ```

   Conversion can restrict a type, where cv1 $\leq$ cv2, *e.g.*, passing an **int** to a **const volatile int** &, which has low cost. Conversion can expand a type, where cv1 $>$ cv2, *e.g.*, passing a **const volatile int** to an **int** &, which has high cost (warning); furthermore, if cv1 has **const** but not cv2, a temporary variable is created to preserve the immutable lvalue.

3. rvalue to reference conversion: T converts to cv T &, which allows binding references to temporaries.

   ```
   int x, & f( int & p );
   f( x + 3 );                  // rvalue parameter (int) implicitly converts to lvalue temporary reference (int &)
   &f(...) = &x;                // rvalue result (int &) implicitly converts to lvalue temporary reference (int &)
   ```

   In both case, modifications to the temporary are inaccessible (warning). Conversion expands the temporary-type with cv, which is low cost since the temporary is inaccessible.

13 string Type

The C∀ string type is for manipulation of dynamically-size character-strings versus C **char** * type for manipulation of statically-size null-terminated character-strings. That is, the amount of storage for a C∀ string changes dynamically at runtime to fit the string size, whereas the amount of storage for a C string is fixed at compile time. Hence, a string declaration does not specify a maximum length; as a string dynamically grows and shrinks in size, so does its underlying storage. In contrast, a C string also dynamically grows and shrinks is size, but its underlying storage is fixed. The maximum storage for a C∀ string value is size_t characters, which is 2^{32} or 2^{64} respectively. A C∀ string manages its length separately from the string, so there is no null (`'\0'`) terminating value at the end of a string value. Hence, a C∀ string cannot be passed to a C string manipulation routine, such as strcat. Like C strings, the characters in a string are numbered starting from 0.

The following operations have been defined to manipulate an instance of type string. The discussion assumes the following declarations and assignment statements are executed.

```
#include <string.hfa>
string s, peter, digit, alpha, punctuation, ifstmt;
int i;
peter  = "PETER";
digit  = "0123456789";
punctuation = "().,";
ifstmt = "IF (A > B) {";
```

```
//  string s = 5;                 sout | s;
    string s;
    // conversion of char and char * to string
    s = 'x';                                     sout | s;      x
    s = "abc";                                   sout | s;      abc
    char cs[5] = "abc";
    s = cs;                                      sout | s;      abc
    // conversion of integral, floating-point, and complex to string
    s = 45hh;                                    sout | s;      45
    s = 45h;                                     sout | s;      45
    s = -(ssize_t)MAX - 1;                       sout | s;      -9223372036854775808
    s = (size_t)MAX;                             sout | s;      18446744073709551615
    s = 5.5;                                     sout | s;      5.5
    s = 5.5L;                                    sout | s;      5.5
    s = 5.5+3.4i;                                sout | s;      5.5+3.4i
    s = 5.5L+3.4Li;                              sout | s;      5.5+3.4i
```

Figure 4: Implicit Conversions to String

Note, the include file string.hfa to access type string.

13.1 Implicit String Conversions

The types **char**, **char** *, **int**, **double**, **_Complex**, including different signness and sizes, implicitly convert to type string. Figure 4 shows examples of implicit conversions between C strings, integral, floating-point and complex types to string. A conversions can be explicitly specified:

```
s = string( "abc" );            // converts char * to string
s = string( 5 );                // converts int to string
s = string( 5.5 );              // converts double to string
```

All conversions from string to **char** *, attempt to be safe: either by requiring the maximum length of the **char** * storage (strncpy) or allocating the **char** * storage for the string characters (ownership), meaning the programmer must free the storage. As well, a string is always null terminates, implying a minimum size of 1 character.

```
string s = "abcde";
char cs[3];
strncpy( cs, s, sizeof(cs) );          sout | cs;      ab
char * cp = s;                         sout | cp;      abcde
delete( cp );
cp = s + ' ' + s;                      sout | cp;      abcde abcde
delete( cp );
```

13.2 Size (length)

The size operation returns the length of a string.

```
i = size( "" );                 // i is assigned 0
i = size( "abc" );              // i is assigned 3
i = size( peter );              // i is assigned 5
```

13.3 Comparison Operators

The binary relational operators, <, <=, >, >=, and equality operators, ==, !=, compare strings using lexicographical ordering, where longer strings are greater than shorter strings.

13.4 Concatenation

The binary operators + and += concatenate two strings, creating the sum of the strings.

```
    s = peter + ' ' + digit;            // s is assigned "PETER 0123456789"
    s += peter;                         // s is assigned "PETER 0123456789PETER"
```

13.5 Repetition

The binary operators * and *= repeat a string N times. If $N = 0$, a zero length string, "" is returned.

```
    s = 'x' * 3;                        // s is assigned "PETER PETER PETER "
    s = (peter + ' ') * 3;              // s is assigned "PETER PETER PETER "
```

13.6 Substring

The substring operation returns a subset of the string starting at a position in the string and traversing a length.

```
    s = peter( 2, 3 );                  // s is assigned "ETE"
    s = peter( 4, -3 );                 // s is assigned "ETE", length is opposite direction
    s = peter( 2, 8 );                  // s is assigned "ETER", length is clipped to 4
    s = peter( 0, -1 );                 // s is assigned "", beyond string so clipped to null
    s = peter(-1, -1 );                 // s is assigned "R", start and length are negative
```

A negative starting position is a specification from the right end of the string. A negative length means that characters are selected in the opposite (right to left) direction from the starting position. If the substring request extends beyond the beginning or end of the string, it is clipped (shortened) to the bounds of the string. If the substring request is completely outside of the original string, a null string located at the end of the original string is returned. The substring operation can also appear on the left hand side of the assignment operator. The substring is replaced by the value on the right hand side of the assignment. The length of the right-hand-side value may be shorter, the same length, or longer than the length of the substring that is selected on the left hand side of the assignment.

```
    digit( 3, 3 ) = "";                 // digit is assigned "0156789"
    digit( 4, 3 ) = "xyz";              // digit is assigned "015xyz9"
    digit( 7, 0 ) = "***";              // digit is assigned "015xyz***9"
    digit(-4, 3 ) = "$$$";              // digit is assigned "015xyz$$$9"
```

A substring is treated as a pointer into the base (substringed) string rather than creating a copy of the subtext. As with all pointers, if the item they are pointing at is changed, then the pointer is referring to the changed item. Pointers to the result value of a substring operation are defined to always start at the same location in their base string as long as that starting location exists, independent of changes to themselves or the base string. However, if the base string value changes, this may affect the values of one or more of the substrings to that base string. If the base string value shortens so that its end is before the starting location of a substring, resulting in the substring starting location disappearing, the substring becomes a null string located at the end of the base string.

The following example illustrates passing the results of substring operations by reference and by value to a subprogram. Notice the side-effects to other reference parameters as one is modified.

```
    main() {
        string x = "xxxxxxxxxxxxx";
        test( x, x(1,3), x(3,3), x(5,5), x(9,5), x(9,5) );
    }

    // x, a, b, c, & d are substring results passed by reference
    // e is a substring result passed by value
    void test(string &x, string &a, string &b, string &c, string &d, string e) {
                                        // x a b c d e
        a( 1, 2 ) = "aaa";              // aaaxxxxxxxxxxx aaax axx xxxxx xxxxx xxxxx
        b( 2, 12 ) = "bbb";             // aaabbbxxxxxxxxx aaab abbb bbxxx xxxxx xxxxx
        c( 4, 5 ) = "ccc";              // aaabbbxcccxxxxxx aaab abbb bbxccc ccxxx xxxxx
        c = "yyy";                      // aaabyyyxxxxxx aaab abyy yyy xxxxx xxxxx
        d( 1, 3 ) = "ddd";              // aaabyyyxdddxx aaab abyy yyy dddxx xxxxx
        e( 1, 3 ) = "eee";              // aaabyyyxdddxx aaab abyy yyy dddxx eeexx
        x = e;                          // eeexx eeex exx x eeexx
    }
```

There is an assignment form of substring in which only the starting position is specified and the length is assumed to be the remainder of the string.

```
string operator () (int start);
```

For example:

```
s = peter( 2 );                    // s is assigned "ETER"
peter( 2 ) = "IPER";               // peter is assigned "PIPER"
```

It is also possible to substring using a string as the index for selecting the substring portion of the string.

```
string operator () (const string &index);
```

For example:

```
digit( "xyz$$$" ) = "678";         // digit is assigned "0156789"
digit( "234") = "***";             // digit is assigned "0156789***"
```

13.7 Searching

The index operation

```
int index( const string &key, int start = 1, occurrence occ = first );
```

returns the position of the first or last occurrence of the key (depending on the occurrence indicator occ that is either first or last) in the current string starting the search at position start. If the key does not appear in the current string, the length of the current string plus one is returned. A negative starting position is a specification from the right end of the string.

```
i = digit.index( "567" );          // i is assigned 3
i = digit.index( "567", 7 );       // i is assigned 11
i = digit.index( "567", -1, last ); // i is assigned 3
i = peter.index( "E", 5, last );   // i is assigned 4
```

The next two string operations test a string to see if it is or is not composed completely of a particular class of characters. For example, are the characters of a string all alphabetic or all numeric? Use of these operations involves a two step operation. First, it is necessary to create an instance of type strmask and initialize it to a string containing the characters of the particular character class, as in:

```
strmask digitmask = digit;
strmask alphamask = string( "abcdefghijklmnopqrstuvwxyz" );
```

Second, the character mask is used in the functions include and exclude to check a string for compliance of its characters with the characters indicated by the mask.

The include operation

```
int include( const strmask &, int = 1, occurrence occ = first );
```

returns the position of the first or last character (depending on the occurrence indicator, which is either first or last) in the current string that does not appear in the mask starting the search at position start; hence it skips over characters in the current string that are included (in) the mask. The characters in the current string do not have to be in the same order as the mask. If all the characters in the current string appear in the mask, the length of the current string plus one is returned, regardless of which occurrence is being searched for. A negative starting position is a specification from the right end of the string.

```
i = peter.include( digitmask );    // i is assigned 1
i = peter.include( alphamask );    // i is assigned 6
```

The exclude operation

```
int exclude( string &mask, int start = 1, occurrence occ = first )
```

returns the position of the first or last character (depending on the occurrence indicator, which is either first or last) in the current string that does appear in the mask string starting the search at position start; hence it skips over characters in the current string that are excluded from (not in) in the mask string. The characters in the current string do not have to be in the same order as the mask string. If all the characters in the current string do NOT appear in the mask string, the length of the current string plus one is returned, regardless of which occurrence is being searched for. A negative starting position is a specification from the right end of the string.

```
i = peter.exclude( digitmask );    // i is assigned 6
i = ifstmt.exclude( strmask( punctuation ) ); // i is assigned 4
```

The includeStr operation:

```
string includeStr( strmask &mask, int start = 1, occurrence occ = first )
```

returns the longest substring of leading or trailing characters (depending on the occurrence indicator, which is either first or last) of the current string that ARE included in the mask string starting the search at position start. A negative starting position is a specification from the right end of the string.

```
s = peter.includeStr( alphamask );      // s is assigned "PETER"
s = ifstmt.includeStr( alphamask );     // s is assigned "IF"
s = peter.includeStr( digitmask );      // s is assigned ""
```

The excludeStr operation:

```
string excludeStr( strmask &mask, int start = 1, occurrence = first )
```

returns the longest substring of leading or trailing characters (depending on the occurrence indicator, which is either first or last) of the current string that are excluded (NOT) in the mask string starting the search at position start. A negative starting position is a specification from the right end of the string.

```
s = peter.excludeStr( digitmask);       // s is assigned "PETER"
s = ifstmt.excludeStr( strmask( punctuation ) ); // s is assigned "IF "
s = peter.excludeStr( alphamask);       // s is assigned ""
```

13.8 Miscellaneous

The trim operation

```
string trim( string &mask, occurrence occ = first )
```

returns a string in that is the longest substring of leading or trailing characters (depending on the occurrence indicator, which is either first or last) which ARE included in the mask are removed.

```
// remove leading blanks
s = string( "    ABC" ).trim( " " );    // s is assigned "ABC",
// remove trailing blanks
s = string( "ABC    " ).trim( " ", last ); // s is assigned "ABC",
```

The translate operation

```
string translate( string &from, string &to )
```

returns a string that is the same length as the original string in which all occurrences of the characters that appear in the from string have been translated into their corresponding character in the to string. Translation is done on a character by character basis between the from and to strings; hence these two strings must be the same length. If a character in the original string does not appear in the from string, then it simply appears as is in the resulting string.

```
// upper to lower case
peter = peter.translate( "ABCDEFGHIJKLMNOPQRSTUVWXYZ", "abcdefghijklmnopqrstuvwxyz" );
        // peter is assigned "peter"
s = ifstmt.translate( "ABCDEFGHIJKLMNOPQRSTUVWXYZ", "abcdefghijklmnopqrstuvwxyz" );
        // ifstmt is assigned "if (a > b) {"
// lower to upper case
peter = peter.translate( "abcdefghijklmnopqrstuvwxyz", "ABCDEFGHIJKLMNOPQRSTUVWXYZ" );
        // peter is assigned "PETER"
```

The replace operation

```
string replace( string &from, string &to )
```

returns a string in which all occurrences of the from string in the current string have been replaced by the to string.

```
s = peter.replace( "E", "XX" );         // s is assigned "PXXTXXR"
```

The replacement is done left-to-right. When an instance of the from string is found and changed to the to string, it is NOT examined again for further replacement.

13.9 Returning N+1 on Failure

Any of the string search routines can fail at some point during the search. When this happens it is necessary to return indicating the failure. Many string types in other languages use some special value to indicate the failure. This value is often 0 or -1 (PL/I returns 0). This section argues that a value of N+1, where N is the length of the base string in the

char []	string
strcpy, strncpy	=
strcat, strncat	+
strcmp, strncmp	==, !=, <, <=, >, >=
strlen	size
[]	[]
strstr	find
strcspn	find_first_of, find_last_of
strspc	find_fist_not_of, find_last_not_of

Table 1: Companion Routines for C∀ string to C Strings

search, is a more useful value to return. The index-of function in APL returns N+1. These are the boundary situations and are often overlooked when designing a string type.

The situation that can be optimized by returning N+1 is when a search is performed to find the starting location for a substring operation. For example, in a program that is extracting words from a text file, it is necessary to scan from left to right over whitespace until the first alphabetic character is found.

```
line = line( line.exclude( alpha ) );
```

If a text line contains all whitespaces, the exclude operation fails to find an alphabetic character. If exclude returns 0 or -1, the result of the substring operation is unclear. Most string types generate an error, or clip the starting value to 1, resulting in the entire whitespace string being selected. If exclude returns N+1, the starting position for the substring operation is beyond the end of the string leaving a null string.

The same situation occurs when scanning off a word.

```
start = line.include(alpha);
word = line(1, start − 1);
```

If the entire line is composed of a word, the include operation will fail to find a non-alphabetic character. In general, returning 0 or -1 is not an appropriate starting position for the substring, which must substring off the word leaving a null string. However, returning N+1 will substring off the word leaving a null string.

13.10 C Compatibility

To ease conversion from C to C∀, there are companion string routines for C strings. Table 1 shows the C routines on the left that also work with string and the rough equivalent string opeation of the right. Hence, it is possible to directly convert a block of C string operations into @string@ just by changing the

For example, this block of C code can be converted to C∀ by simply changing the type of variable s from **char** [] to string.

```
char s[32];
//string s;
strcpy( s, "abc" );             PRINT( %s, s );
strncpy( s, "abcdef", 3 );      PRINT( %s, s );
strcat( s, "xyz" );             PRINT( %s, s );
strncat( s, "uvwxyz", 3 );      PRINT( %s, s );
PRINT( %zd, strlen( s ) );
PRINT( %c, s[3] );
PRINT( %s, strstr( s, "yzu" ) ) ;
PRINT( %s, strstr( s, 'y' ) ) ;
```

However, the conversion fails with I/O because printf cannot print a string using format code %s because C∀ strings are not null terminated.

13.11 Input/Output Operators

Both the C++ operators << and >> are defined on type string. However, input of a string value is different from input of a **char** * value. When a string value is read, *all* input characters from the current point in the input stream to either the end of line ('\n') or the end of file are read.

14 Enumeration

An *enumeration* is a compile-time mechanism to alias names to constants, like **typedef** is a mechanism to alias names to types. Its purpose is to define a restricted-value type providing code-readability and maintenance – changing an enum's value automatically updates all name usages during compilation.

An enumeration type is a set of names, each called an *enumeration constant* (shortened to *enum*) aliased to a fixed value (constant).

```
enum Days { Mon, Tue, Wed, Thu, Fri, Sat, Sun }; // enumeration type definition, set of 7 names & values
Days days = Mon; // enumeration type declaration and initialization
```

The set of enums is injected into the variable namespace at the definition scope. Hence, enums may be overloaded with variable, enum, and function names.

```
int Foo;                        // type/variable separate namespaces
enum Foo { Bar };
enum Goo { Bar };               // overload Foo.Bar
double Bar;                     // overload Foo.Bar, Goo.Bar
```

An anonymous enumeration injects enums with specific values into a scope.

```
enum { Prime = 103, BufferSize = 1024 };
```

An enumeration is better than using C preprocessor or constant declarations.

```
#define Mon 0           const int Mon = 0,
...                               ...,
#define Sun 6                     Sun = 6;
```

because the enumeration is succinct, has automatic numbering, can appear in **case** labels, does not use storage, and is part of the language type-system. Finally, the type of an enum is implicitly or explicitly specified and the constant value can be implicitly or explicitly specified. Note, enum values may be repeated in an enumeration.

14.1 Enum type

The type of enums can be any type, and an enum's value comes from this type. Because an enum is a constant, it cannot appear in a mutable context, *e.g.*, Mon = Sun is disallowed, and has no address (it is an rvalue). Therefore, an enum is automatically converted to its constant's base-type, *e.g.*, comparing/printing an enum compares/prints its value rather than the enum name; there is no mechanism to print the enum name.

The default enum type is **int**. Hence, Days is the set type Mon, Tue, ..., Sun, while the type of each enum is **int** and each enum represents a fixed integral value. If no values are specified for an integral enum type, the enums are automatically numbered by one from left to right starting at zero. Hence, the value of enum Mon is 0, Tue is 1, ..., Sun is 6. If an enum value is specified, numbering continues by one from that value for subsequent unnumbered enums. If an enum value is a *constant* expression, the compiler performs constant-folding to obtain a constant value.

C∀ allows other integral types with associated values.

```
enum( char ) Letter { A = 'A', B, C, I = 'I', J, K };
enum( long long int ) BigNum { X = 123_456_789_012_345, Y = 345_012_789_456_123 };
```

For enumeration Letter, enum A's value is explicitly set to `'A'`, with B and C implicitly numbered with increasing values from `'A'`, and similarly for enums I, J, and K.

Non-integral enum types must be explicitly initialized, *e.g.*, **double** is not automatically numbered by one.

```
// non−integral numeric
enum( double ) Math { PI_2 = 1.570796, PI = 3.141597, E = 2.718282 }
// pointer
enum( char * ) Name { Fred = "Fred", Mary = "Mary", Jane = "Jane" };
int i, j, k;
enum( int * ) ptr { I = &i, J = &j, K = &k };
enum( int & ) ref { I = i,  J = j,  K = k };
// tuple
enum( [int, int] ) { T = [ 1, 2 ] };
// function
void f() {...}  void g() {...}
enum( void (*)() ) funs { F = f, F = g };
```

```
// aggregate
struct S { int i, j; };
enum( S ) s { A = { 3,  4 }, B = { 7,  8 } };
// enumeration
enum( Letter ) Greek { Alph = A, Beta = B, /* more enums */ }; // alphabet intersection
```

Enumeration Greek may have more or less enums than Letter, but the enum values *must* be from Letter. Therefore, Greek enums are a subset of type Letter and are type compatible with enumeration Letter, but Letter enums are not type compatible with enumeration Greek.

The following examples illustrate the difference between the enumeration type and the type of its enums.

```
Math m = PI;                    // allowed
double d = PI;                  // allowed, conversion to base type
m = E;                          // allowed
m = Alph;                       // disallowed
m = 3.141597;                   // disallowed
d = m;                          // allowed
d = Alph;                       // disallowed
Letter l = A;                   // allowed
Greek g = Alph;                 // allowed
l = Alph;                       // allowed, conversion to base type
g = A;                          // disallowed
```

A constructor *cannot* be used to initialize enums because a constructor executes at runtime. A fallback is explicit C-style initialization using @=.

```
enum( struct vec3 ) Axis { Up @= { 1, 0, 0 }, Left @= { 0, 1, 0 }, Front @= { 0, 0, 1 } }
```

Finally, enumeration variables are assignable and comparable only if the appropriate operators are defined for its enum type.

14.2 Inheritance

Plan-9 inheritance may be used with enumerations.

```
enum( char * ) Name2 { inline Name, Jack = "Jack", Jill = "Jill" };
enum /* inferred */ Name3 { inline Name2, Sue = "Sue", Tom = "Tom" };
```

Enumeration Name2 inherits all the enums and their values from enumeration Name by containment, and a Name enumeration is a subtype of enumeration Name2. Note, enums must be unique in inheritance but enum values may be repeated. The enum type for the inheriting type must be the same as the inherited type; hence the enum type may be omitted for the inheriting enumeration and it is inferred from the inherited enumeration, as for Name3. When inheriting from integral types, automatic numbering may be used, so the inheritance placement left to right is important, *e.g.*, the placement of Sue and Tom before or after **inline** Name2.

Specifically, the inheritance relationship for Names is:

```
Name $\(\subseteq\)$ Name2 $\(\subseteq\)$ Name3 $\(\subseteq\)$ const char * // enum type of Name
```

Hence, given

```
void f( Name );
void g( Name2 );
void h( Name3 );
void j( const char * );
```

the following calls are valid

```
f( Fred );
g( Fred );   g( Jill );
h( Fred );   h( Jill );   h( Sue );
j( Fred );   j( Jill );   j( Sue );   j( 'W' );
```

Note, the validity of calls is the same for call-by-reference as for call-by-value, and **const** restrictions are the same as for other types.

Enums cannot be created at runtime, so inheritence problems, such as contra-variance do not apply. Only instances of the enum base-type may be created at runtime.

15 Routine Definition

C∀ supports a new syntax for routine definition, as well as C11 and K&R routine syntax. The point of the new syntax is to allow returning multiple values from a routine [16, 25], *e.g.*:

```
[ int o1, int o2, char o3 ] f( int i1, char i2, char i3 ) {
    routine body
}
```

where routine f has three output (return values) and three input parameters. Existing C syntax cannot be extended with multiple return types because it is impossible to embed a single routine name within multiple return type-specifications.

In detail, the brackets, [], enclose the result type, where each return value is named and that name is a local variable of the particular return type.[14] The value of each local return variable is automatically returned at routine termination. Declaration qualifiers can only appear at the start of a routine definition, *e.g.*:

```
extern [ int x ] g( int y ) {}
```

Lastly, if there are no output parameters or input parameters, the brackets and/or parentheses must still be specified; in both cases the type is assumed to be void as opposed to old style C defaults of int return type and unknown parameter types, respectively, as in:

```
[] g();                             // no input or output parameters
[ void ] g( void );                 // no input or output parameters
```

Routine f is called as follows:

```
[ i, j, ch ] = f( 3, 'a', ch );
```

The list of return values from f and the grouping on the left-hand side of the assignment is called a *return list* and discussed in Section 12.

C∀ style declarations cannot be used to declare parameters for K&R style routine definitions because of the following ambiguity:

```
int (*f(x))[ 5 ] int x; {}
```

The string "**int** (*f(x))[5]" declares a K&R style routine of type returning a pointer to an array of 5 integers, while the string "[5] **int** x" declares a C∀ style parameter x of type array of 5 integers. Since the strings overlap starting with the open bracket, [, there is an ambiguous interpretation for the string. As well, C∀-style declarations cannot be used to declare parameters for C-style routine-definitions because of the following ambiguity:

```
typedef int foo;
int f( int (* foo) );               // foo is redefined as a parameter name
```

The string "**int** (* foo)" declares a C-style named-parameter of type pointer to an integer (the parenthesis are superfluous), while the same string declares a C∀ style unnamed parameter of type routine returning integer with unnamed parameter of type pointer to foo. The redefinition of a type name in a parameter list is the only context in C where the character * can appear to the left of a type name, and C∀ relies on all type qualifier characters appearing to the right of the type name. The inability to use C∀ declarations in these two contexts is probably a blessing because it precludes programmers from arbitrarily switching between declarations forms within a declaration contexts.

C-style declarations can be used to declare parameters for C∀ style routine definitions, *e.g.*:

```
[ int ] f( * int, int * );          // returns an integer, accepts 2 pointers to integers
[ * int, int * ] f( int );          // returns 2 pointers to integers, accepts an integer
```

The reason for allowing both declaration styles in the new context is for backwards compatibility with existing preprocessor macros that generate C-style declaration-syntax, as in:

```
#define ptoa( n, d ) int (*n)[ d ]
int f( ptoa( p, 5 ) ) ...           // expands to int f( int (*p)[ 5 ] )
[ int ] f( ptoa( p, 5 ) ) ...       // expands to [ int ] f( int (*p)[ 5 ] )
```

Again, programmers are highly encouraged to use one declaration form or the other, rather than mixing the forms.

15.1 Named Return Values

Named return values handle the case where it is necessary to define a local variable whose value is then returned in a **return** statement, as in:

[14] Michael Tiemann, with help from Doug Lea, provided named return values in g++, circa 1989.

```
int f() {
    int x;
    ... x = 0; ... x = y; ...
    return x;
}
```

Because the value in the return variable is automatically returned when a C∀ routine terminates, the **return** statement *does not* contain an expression, as in:

```
[ int x, int y ] f() {
    int z;
    ... x = 0; ... y = z; ...
    return;                           // implicitly return x, y
}
```

When the return is encountered, the current values of x and y are returned to the calling routine. As well, "falling off the end" of a routine without a **return** statement is permitted, as in:

```
[ int x, int y ] f() {
    ...
}                                     // implicitly return x, y
```

In this case, the current values of x and y are returned to the calling routine just as if a **return** had been encountered.

Named return values may be used in conjunction with named parameter values; specifically, a return and parameter can have the same name.

```
[ int x, int y ] f( int, x, int y ) {
    ...
}                                     // implicitly return x, y
```

This notation allows the compiler to eliminate temporary variables in nested routine calls.

```
[ int x, int y ] f( int, x, int y );  // prototype declaration
int a, b;
[a, b] = f( f( f( a, b ) ) );
```

While the compiler normally ignores parameters names in prototype declarations, here they are used to eliminate temporary return-values by inferring that the results of each call are the inputs of the next call, and ultimately, the left-hand side of the assignment. Hence, even without the body of routine f (separate compilation), it is possible to perform a global optimization across routine calls. The compiler warns about naming inconsistencies between routine prototype and definition in this case, and behaviour is undefined if the programmer is inconsistent.

15.2 Routine Prototype

The syntax of the new routine prototype declaration follows directly from the new routine definition syntax; as well, parameter names are optional, *e.g.*:

```
[ int x ] f ();                       // returning int with no parameters
[ * int ] g (int y);                  // returning pointer to int with int parameter
[ ] 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 source text from the routine definition header (or vice versa). Like C, it is possible to declare multiple routine-prototypes in a single declaration, where the return type is distributed across *all* routine names in the declaration list (see Section 11, p. 18), *e.g.*:

```
C :    const double bar1(), bar2( int ), bar3( double );
C∀ :  [const double] foo(), foo( int ), foo( double ) { return 3.0; }
```

C∀ allows the last routine in the list to define its body.

Declaration qualifiers can only appear at the start of a C∀ routine declaration,[7] *e.g.*:

```
extern [ int ] f ( int );
static [ int ] g ( int );
```

16 Routine Pointers

The syntax for pointers to C∀ routines specifies the pointer name on the right, *e.g.*:

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

While parameter names are optional, *a routine name cannot be specified*; for example, the following is incorrect:

```
* [ int x ] f () fp;               // routine name "f" is not allowed
```

17 Named and Default Arguments

Named and default arguments [20][15] are two mechanisms to simplify routine call. Both mechanisms are discussed with respect to C∀.

Named (or Keyword) Arguments: provide the ability to specify an argument to a routine call using the parameter name rather than the position of the parameter. For example, given the routine:

```
void p( int x, int y, int z ) {...}
```

a positional call is:

```
p( 4, 7, 3 );
```

whereas a named (keyword) call may be:

```
p( z : 3, x : 4, y : 7 );          // rewrite ⇒ p( 4, 7, 3 )
```

Here the order of the arguments is unimportant, and the names of the parameters are used to associate argument values with the corresponding parameters. The compiler rewrites a named call into a positional call. The advantages of named parameters are:

- Remembering the names of the parameters may be easier than the order in the routine definition.
- Parameter names provide documentation at the call site (assuming the names are descriptive).
- Changes can be made to the order or number of parameters without affecting the call (although the call must still be recompiled).

Unfortunately, named arguments do not work in C-style programming-languages because a routine prototype is not required to specify parameter names, nor do the names in the prototype have to match with the actual definition. For example, the following routine prototypes and definition are all valid.

```
void p( int, int, int );           // equivalent prototypes
void p( int x, int y, int z );
void p( int y, int x, int z );
void p( int z, int y, int x );
void p( int q, int r, int s ) {}   // match with this definition
```

Forcing matching parameter names in routine prototypes with corresponding routine definitions is possible, but goes against a strong tradition in C programming. Alternatively, prototype definitions can be eliminated by using a two-pass compilation, and implicitly creating header files for exports. The former is easy to do, while the latter is more complex.

Furthermore, named arguments do not work well in a C∀-style programming-languages because they potentially introduces a new criteria for type matching. For example, it is technically possible to disambiguate between these two overloaded definitions of f based on named arguments at the call site:

```
int f( int i, int j );
int f( int x, double y );

f( j : 3, i : 4 );                 // 1st f
f( x : 7, y : 8.1 );               // 2nd f
f( 4, 5 );                         // ambiguous call
```

However, named arguments compound routine resolution in conjunction with conversions:

[15]Francez [15] proposed a further extension to the named-parameter passing style, which specifies what type of communication (by value, by reference, by name) the argument is passed to the routine.

```
    f( i : 3, 5.7 );                    // ambiguous call ?
```

Depending on the cost associated with named arguments, this call could be resolvable or ambiguous. Adding named argument into the routine resolution algorithm does not seem worth the complexity. Therefore, C∀ does *not* attempt to support named arguments.

Default Arguments provide the ability to associate a default value with a parameter so it can be optionally specified in the argument list. For example, given the routine:

```
    void p( int x = 1, int y = 2, int z = 3 ) {...}
```

the allowable positional calls are:

```
    p();                                // rewrite ⇒ p( 1, 2, 3 )
    p( 4 );                             // rewrite ⇒ p( 4, 2, 3 )
    p( 4, 4 );                          // rewrite ⇒ p( 4, 4, 3 )
    p( 4, 4, 4 );                       // rewrite ⇒ p( 4, 4, 4 )
    // empty arguments
    p(  , 4, 4 );                       // rewrite ⇒ p( 1, 4, 4 )
    p( 4,  , 4 );                       // rewrite ⇒ p( 4, 2, 4 )
    p( 4, 4,   );                       // rewrite ⇒ p( 4, 4, 3 )
    p( 4,  ,   );                       // rewrite ⇒ p( 4, 2, 3 )
    p(  , 4,   );                       // rewrite ⇒ p( 1, 4, 3 )
    p(  ,  , 4 );                       // rewrite ⇒ p( 1, 2, 4 )
    p(  ,  ,   );                       // rewrite ⇒ p( 1, 2, 3 )
```

Here the missing arguments are inserted from the default values in the parameter list. The compiler rewrites missing default values into explicit positional arguments. The advantages of default values are:

- Routines with a large number of parameters are often very generalized, giving a programmer a number of different options on how a computation is performed. For many of these kinds of routines, there are standard or default settings that work for the majority of computations. Without default values for parameters, a programmer is forced to specify these common values all the time, resulting in long argument lists that are error prone.
- When a routine's interface is augmented with new parameters, it extends the interface providing generalizability[16] (somewhat like the generalization provided by inheritance for classes). That is, all existing calls are still valid, although the call must still be recompiled.

The only disadvantage of default arguments is that unintentional omission of an argument may not result in a compiler-time error. Instead, a default value is used, which may not be the programmer's intent.

Default values may only appear in a prototype versus definition context:

```
    void p( int x, int y = 2, int z = 3 );   // prototype: allowed
    void p( int, int = 2, int = 3 );         // prototype: allowed
    void p( int x, int y = 2, int z = 3 ) {} // definition: not allowed
```

The reason for this restriction is to allow separate compilation. Multiple prototypes with different default values is an error.

Ellipse ("...") arguments present problems when used with default arguments. The conflict occurs because both named and ellipse arguments must appear after positional arguments, giving two possibilities:

```
p( /* positional */, ... , /* named */ );
p( /* positional */, /* named */, ... );
```

While it is possible to implement both approaches, the first possibly is more complex than the second, *e.g.*:

```
p( int x, int y, int z, ... );
p( 1, 4, 5, 6, z : 3, y : 2 );          // assume p( /* positional */, ... , /* named */ );
p( 1, z : 3, y : 2, 4, 5, 6 );          // assume p( /* positional */, /* named */, ... );
```

[16] "It should be possible for the implementor of an abstraction to increase its generality. So long as the modified abstraction is a generalization of the original, existing uses of the abstraction will not require change. It might be possible to modify an abstraction in a manner which is not a generalization without affecting existing uses, but, without inspecting the modules in which the uses occur, this possibility cannot be determined. This criterion precludes the addition of parameters, unless these parameters have default or inferred values that are valid for all possible existing applications." [8, p. 128]

In the first call, it is necessary for the programmer to conceptually rewrite the call, changing named arguments into positional, before knowing where the ellipse arguments begin. Hence, this approach seems significantly more difficult, and hence, confusing and error prone. In the second call, the named arguments separate the positional and ellipse arguments, making it trivial to read the call.

The problem is exacerbated with default arguments, *e.g.*:

```
void p( int x, int y = 2, int z = 3... );
p( 1, 4, 5, 6, z : 3 );             // assume p( /* positional */, ... , /* named */ );
p( 1, z : 3, 4, 5, 6 );             // assume p( /* positional */, /* named */, ... );
```

The first call is an error because arguments 4 and 5 are actually positional not ellipse arguments; therefore, argument 5 subsequently conflicts with the named argument z : 3. In the second call, the default value for y is implicitly inserted after argument 1 and the named arguments separate the positional and ellipse arguments, making it trivial to read the call. For these reasons, C∀ requires named arguments before ellipse arguments. Finally, while ellipse arguments are needed for a small set of existing C routines, like printf, the extended C∀ type system largely eliminates the need for ellipse arguments (see Section 26, p. 62), making much of this discussion moot.

Default arguments and overloading (see Section 26, p. 62) are complementary. While in theory default arguments can be simulated with overloading, as in:

default arguments	**overloading**

`void p( int x, int y = 2, int z = 3 ) {...}`	`void p( int x, int y, int z ) {...}`
	`void p( int x ) { p( x, 2, 3 ); }`
	`void p( int x, int y ) { p( x, y, 3 ); }`

the number of required overloaded routines is linear in the number of default values, which is unacceptable growth. In general, overloading should only be used over default arguments if the body of the routine is significantly different. Furthermore, overloading cannot handle accessing default arguments in the middle of a positional list, via a missing argument, such as:

```
p( 1, /* default */, 5 );           // rewrite ⇒ p( 1, 2, 5 )
```

Given the C∀ restrictions above, both named and default arguments are backwards compatible. C++ only supports default arguments; Ada supports both named and default arguments.

18 Unnamed Structure Fields

C requires each field of a structure to have a name, except for a bit field associated with a basic type, *e.g.*:

```
struct {
    int f1;                         // named field
    int f2 : 4;                     // named field with bit field size
    int : 3;                        // unnamed field for basic type with bit field size
    int ;                           // disallowed, unnamed field
    int *;                          // disallowed, unnamed field
    int (*)( int );                 // disallowed, unnamed field
};
```

This requirement is relaxed by making the field name optional for all field declarations; therefore, all the field declarations in the example are allowed. As for unnamed bit fields, an unnamed field is used for padding a structure to a particular size. A list of unnamed fields is also supported, *e.g.*:

```
struct {
    int , , ;                       // 3 unnamed fields
}
```

19 Nesting

Nesting of types and routines is useful for controlling name visibility (*name hiding*).

19.1 Type Nesting

C∀ allows type nesting, and type qualification of the nested types (see Figure 5), where as C hoists (refactors) nested types into the enclosing scope and has no type qualification. In the left example in C, types C, U and T are implicitly

```
C Type Nesting                  C Implicit Hoisting     | C∀
struct S {                      enum C { R, G, B };     | struct S {
    enum C { R, G, B };         union U { int i, j; };  |     enum C { R, G, B };
    struct T {                  struct T {              |     struct T {
        union U { int i, j; };      enum C c;           |         union U { int i, j; };
        enum C c;                   short int i, j;     |         enum C c;
        short int i, j;         };                      |         short int i, j;
    };                          struct S {              |     };
    struct T t;                     struct T t;         |     struct T t;
} s;                            } s;                    | } s;
                                                        |
int fred() {                                            | int fred() {
    s.t.c = R;                                          |     s.t.c = S.R;  // type qualification
    struct T t = { R, 1, 2 };                           |     struct S.T t = { S.R, 1, 2 };
    enum C c;                                           |     enum S.C c;
    union U u;                                          |     union S.T.U u;
}                                                       | }
```

Figure 5: Type Nesting / Qualification

hoisted outside of type S into the containing block scope. In the right example in C∀, the types are not hoisted and accessed using the field-selection operator "." for type qualification, as does Java, rather than the C++ type-selection operator "::".

19.2 Routine Nesting

While C∀ does not provide object programming by putting routines into structures, it does rely heavily on locally nested routines to redefine operations at or close to a call site. For example, the C quick-sort is wrapped into the following polymorphic C∀ routine:

```
forall( T | { int ?<?( T, T ); } )
void qsort( const T * arr, size_t dimension );
```

which can be used to sort in ascending and descending order by locally redefining the less-than operator into greater-than.

```
const unsigned int size = 5;
int ia[size];
...                                    // assign values to array ia
qsort( ia, size );                     // sort ascending order using builtin ?<?
{
    int ?<?( int x, int y ) { return x > y; } // nested routine
    qsort( ia, size );                 // sort descending order by local redefinition
}
```

Nested routines are not first-class, meaning a nested routine cannot be returned if it has references to variables in its enclosing blocks; the only exception is references to the external block of the translation unit, as these variables persist for the duration of the program. The following program in undefined in C∀ (and Indexcgcc)

```
[* [int]( int )] foo() {               // int (* foo())( int )
    int i = 7;
    int bar( int p ) {
        i += 1;                        // dependent on local variable
        sout | i;
    }
    return bar;                        // undefined because of local dependence
}
int main() {
    * [int]( int ) fp = foo();         // int (* fp)( int )
    sout | fp( 3 );
}
```

because

Currently, there are no lambda expressions, *i.e.*, unnamed routines because routine names are very important to properly select the correct routine.

20 Tuple

In C and C∀, lists of elements appear in several contexts, such as the parameter list of a routine call.

```
f( 2, x, 3 + i );                          // element list
```

A list of elements is called a *tuple*, and is different from a comma expression.

20.1 Multiple-Return-Value Functions

In C and most programming languages, functions return at most one value; however, many operations have multiple outcomes, some exceptional (see Section 10, p. 16). To emulate functions with multiple return values, *aggregation* and/or *aliasing* is used.

In the former approach, a record type is created combining all of the return values. For example, consider C's div function, which returns the quotient and remainder for a division of an integer value.

```
typedef struct { int quot, rem; } div_t; // from include stdlib.h
div_t div( int num, int den );
div_t qr = div( 13, 5 );                 // return quotient/remainder aggregate
printf( "%d %d\n", qr.quot, qr.rem );    // print quotient/remainder
```

This approach requires a name for the return type and fields, where naming is a common programming-language issue. That is, naming creates an association that must be managed when reading and writing code. While effective when used sparingly, this approach does not scale when functions need to return multiple combinations of types.

In the latter approach, additional return values are passed as pointer parameters. A pointer parameter is assigned inside the routine to emulate a return. For example, consider C's modf function, which returns the integral and fractional part of a floating value.

```
double modf( double x, double * i );    // from include math.h
double intp, frac = modf( 13.5, &intp ); // return integral and fractional components
printf( "%g %g\n", intp, frac );         // print integral/fractional components
```

This approach requires allocating storage for the return values, which complicates the call site with a sequence of variable declarations leading to the call. Also, while a disciplined use of **const** can give clues about whether a pointer parameter is used as an out parameter, it is not obvious from the routine signature whether the callee expects such a parameter to be initialized before the call. Furthermore, while many C routines that accept pointers are safe for a NULL argument, there are many C routines that are not null-safe. Finally, C does not provide a mechanism to state that a parameter is going to be used as an additional return value, which makes the job of ensuring that a value is returned more difficult for the compiler. Still, not every routine with multiple return values should be required to return an error code, and error codes are easily ignored, so this is not a satisfying solution. As with the previous approach, this technique can simulate multiple return values, but in practice it is verbose and error prone.

C∀ allows functions to return multiple values by extending the function declaration syntax. Multiple return values are declared as a comma-separated list of types in square brackets in the same location that the return type appears in standard C function declarations.

```
[ char, int, double ] f( ... );
```

The ability to return multiple values from a function requires a new syntax for the return statement. For consistency, the return statement in C∀ accepts a comma-separated list of expressions in square brackets.

```
return [ c, i, d ];
```

The expression resolution ensures the correct form is used depending on the values being returned and the return type of the current function. A multiple-returning function with return type T can return any expression that is implicitly convertible to T.

A common use of a function's output is input to another function. C∀ allows this case, without any new syntax; a multiple-returning function can be used in any of the contexts where an expression is allowed. When a function call is passed as an argument to another call, the best match of actual arguments to formal parameters is evaluated given all possible expression interpretations in the current scope.

```
void g( int, int );                   // 1
void g( double, double );             // 2
g( div( 13, 5 ) );                    // select 1
g( modf( 13.5 ) );                    // select 2
```

In this case, there are two overloaded g routines. Both calls to g expect two arguments that are matched by the two return values from div and modf. respectively, which are fed directly to the first and second parameters of g. As well, both calls to g have exact type matches for the two different versions of g, so these exact matches are chosen. When type matches are not exact, conversions are used to find a best match.

The previous examples can be rewritten passing the multiple returned-values directly to the printf function call.

```
[ int, int ] div( int x, int y );          // from include stdlib
printf( "%d %d\n", div( 13, 5 ) );         // print quotient/remainder

[ double, double ] modf( double x );       // from include math
printf( "%g %g\n", modf( 13.5 ) );         // print integral/fractional components
```

This approach provides the benefits of compile-time checking for appropriate return statements as in aggregation, but without the required verbosity of declaring a new named type.

Finally, the addition of multiple-return-value functions necessitates a syntax for retaining the multiple values at the call-site versus their temporary existence during a call. The simplest mechanism for retaining a return value in C is variable assignment. By assigning the multiple return-values into multiple variables, the values can be retrieved later. As such, C∀ allows assigning multiple values from a function into multiple variables, using a square-bracketed list of lvalue expressions on the left side.

```
int quot, rem;
[ quot, rem ] = div( 13, 5 );              // assign multiple variables
printf( "%d %d\n", quot, rem );            // print quotient/remainder
```

Here, the multiple return-values are matched in much the same way as passing multiple return-values to multiple parameters in a call.

20.2 Expressions

Multiple-return-value functions provide C∀ with a new syntax for expressing a combination of expressions in the return statement and a combination of types in a function signature. These notions are generalized to provide C∀ with *tuple expression*s and *tuple type*s. A tuple expression is an expression producing a fixed-size, ordered list of values of heterogeneous types. The type of a tuple expression is the tuple of the subexpression types, or a tuple type.

In C∀, a tuple expression is denoted by a comma-separated list of expressions enclosed in square brackets. For example, the expression [5, 'x', 10.5] has type [**int**, **char**, **double**]. The previous expression has 3 *components*. Each component in a tuple expression can be any C∀ expression, including another tuple expression. The order of evaluation of the components in a tuple expression is unspecified, to allow a compiler the greatest flexibility for program optimization. It is, however, guaranteed that each component of a tuple expression is evaluated for side-effects, even if the result is not used. Multiple-return-value functions can equivalently be called *tuple-returning functions*.

20.3 Variables

The previous call of div still requires the preallocation of multiple return-variables in a manner similar to the aliasing example. In C∀, it is possible to overcome this restriction by declaring a *tuple variable*.

```
[int, int] qr = div( 13, 5 );              // initialize tuple variable
printf( "%d %d\n", qr );                   // print quotient/remainder
```

It is now possible to match the multiple return-values to a single variable, in much the same way as aggregation. As well, the components of the tuple value are passed as separate parameters to printf, allowing direct printing of tuple variables. One way to access the individual components of a tuple variable is with assignment.

```
[ quot, rem ] = qr;                        // assign multiple variables
```

In addition to variables of tuple type, it is also possible to have pointers to tuples, and arrays of tuples. Tuple types can be composed of any types, except for array types, since array assignment is disallowed, which makes tuple assignment difficult when a tuple contains an array.

```
[ double, int ] di;
[ double, int ] * pdi
[ double, int ] adi[10];
```

This examples declares a variable of type [**double**, **int**], a variable of type pointer to [**double**, **int**], and an array of ten [**double**, **int**].

20.4 Indexing

It is also possible to access a single component of a tuple-valued expression without creating temporary variables. Given a tuple-valued expression *e*np and a compile-time constant integer i where $0 \leq i < n$, where n is the number of components in e, $e.i$ accesses the i^{th} component of e, *e.g.*:

```
[int, double] x;
[char *, int] f();
void g(double, int);
[int, double] * p;

int y = x.0;                    // access int component of x
y = f().1;                      // access int component of f
p->0 = 5;                       // access int component of tuple pointed-to by p
g( x.1, x.0 );                  // rearrange x to pass to g
double z = [ x, f() ].0.1;      // access second component of first component of tuple expression
```

Tuple-index expressions can occur on any tuple-typed expression, including tuple-returning functions, square-bracketed tuple expressions, and other tuple-index expressions, provided the retrieved component is also a tuple. This feature was proposed for K-W C but never implemented [31, p. 45].

20.5 Flattening and Structuring

As evident in previous examples, tuples in C∀ do not have a rigid structure. 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 & structure
```

In C∀, each of these calls is valid. In the call to f, x is implicitly flattened so that the components of x are passed as the two arguments to f. For the call to g, the values y and 10 are structured into a single argument of type [**int**, **int**] to match the type of the parameter 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 single- and multiple-return-value functions, and with any number of arguments of arbitrarily complex structure.

In K-W C [5, 31], there were 4 tuple coercions: opening, closing, flattening, and structuring. Opening coerces a tuple value into a tuple of values, while closing converts a tuple of values into a single tuple value. Flattening coerces a nested tuple into a flat tuple, *i.e.*, it takes a tuple with tuple components and expands it into a tuple with only non-tuple components. Structuring moves in the opposite direction, *i.e.*, it takes a flat tuple value and provides structure by introducing nested tuple components.

In C∀, the design has been simplified to require only the two conversions previously described, which trigger only in function call and return situations. This simplification is a primary contribution of this thesis to the design of tuples in C∀. Specifically, the expression resolution algorithm examines all of the possible alternatives for an expression to determine the best match. In resolving a function call expression, each combination of function value and list of argument alternatives is examined. Given a particular argument list and function value, the list of argument alternatives is flattened to produce a list of non-tuple valued expressions. Then the flattened list of expressions is compared with each value in the function's parameter list. If the parameter's type is not a tuple type, then the current argument value is unified with the parameter type, and on success the next argument and parameter are examined. If the parameter's type is a tuple type, then the structuring conversion takes effect, recursively applying the parameter matching algorithm using the tuple's component types as the parameter list types. Assuming a successful unification, eventually the algorithm gets to the end of the tuple type, which causes all of the matching expressions to be consumed and structured into a tuple expression. For example, in

```
int f(int, [double, int]);
f([5, 10.2], 4);
```

There is only a single definition of f, and 3 arguments with only single interpretations. First, the argument alternative list [5, 10.2], 4 is flattened to produce the argument list 5, 10.2, 4. Next, the parameter matching algorithm begins, with $P =$ **int** and $A =$ **int**, which unifies exactly. Moving to the next parameter and argument, $P =$ [**double**, **int**] and $A =$ **double**. This time, the parameter is a tuple type, so the algorithm applies recursively with $P' =$ **double** and $A =$ **double**, which unifies exactly. Then $P' =$ **int** and $A =$ **double**, which again unifies exactly. At this point, the end of P' has been reached, so the arguments 10.2, 4 are structured into the tuple expression [10.2, 4]. Finally, the end of the parameter list P has also been reached, so the final expression is f(5, [10.2, 4]).

20.6 Assignment

An assignment where the left side of the assignment operator has 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 non-tuple or tuple type, called *mass* and *multiple* assignment, respectively.

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

Let L_i for i in $[0,n)$ represent each component of the flattened left side, R_i represent each component of the flattened right side of a multiple assignment, and R represent the right side of a mass assignment.

For a multiple assignment to be valid, both tuples must have the same number of elements when flattened. For example, the following is invalid because the number of components on the left does not match the number of components on the right.

```
[ int, int ] x, y, z;
[ x, y ] = z;                     // multiple assignment, invalid 4 != 2
```

Multiple assignment assigns R_i to L_i for each i. That is, ?=?(&L_i, R_i) must be a well-typed expression. In the previous example, [x, y] = z, z is flattened into z.0, z.1, and the assignments x = z.0 and y = z.1 happen.

A mass assignment assigns the value R to each L_i. For a mass assignment to be valid, ?=?(&L_i, R) must be a well-typed expression. These semantics differ from C cascading assignment (*e.g.*, a=b=c) in that conversions are applied to R 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, which results in the value 3.14 in y and the value 3 in x. On the other hand, the C cascading assignment y = x = 3.14 performs the assignments x = 3.14 and y = x, which results in the value 3 in x, and as a result the value 3 in y as well.

Both kinds of tuple assignment have parallel semantics, such that each value on the left side and right side 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.

```
int x = 10, y = 20;
[ x, y ] = [ y, x ];
```

After executing this code, x has the value 20 and y has the value 10.

In C∀, tuple assignment is an expression where the result type is the type of the left side of the assignment, as in normal assignment. That is, a tuple assignment produces the value of the left-hand side after assignment. These semantics allow cascading tuple assignment to work out naturally in any context where a tuple is permitted. These semantics are a change from the original tuple design in K-W C [31], wherein tuple assignment was a statement that allows cascading assignments as a special case. Restricting tuple assignment to statements was an attempt to to fix what was seen as a problem with side-effects, wherein assignment can be used in many different locations, such as in function-call argument position. While permitting assignment as an expression does introduce the potential for subtle complexities, it is impossible to remove assignment expressions from C∀ without affecting backwards compatibility. Furthermore, there are situations where permitting assignment as an expression improves readability by keeping code succinct and reducing repetition, and complicating the definition of tuple assignment puts a greater cognitive burden on the user. In another language, tuple assignment as a statement could be reasonable, but it would be inconsistent for tuple assignment to be the only kind of assignment that is not an expression. In addition, K-W C permits the compiler to optimize tuple assignment as a block copy, since it does not support user-defined assignment operators. This optimization could be implemented in C∀, but it requires the compiler to verify that the selected assignment operator is trivial.

The following example shows multiple, mass, and cascading assignment used in one expression

```
    int a, b;
    double c, d;
    [ void ] f( [ int, int ] );
    f( [ c, a ] = [ b, d ] = 1.5 );           // assignments in parameter list
```

The tuple expression begins with a mass assignment of 1.5 into [b, d], which assigns 1.5 into b, which is truncated to 1, and 1.5 into d, producing the tuple [1, 1.5] as a result. That tuple is used as the right side of the multiple assignment (*i.e.*, [c, a] = [1, 1.5]) that assigns 1 into c and 1.5 into a, which is truncated to 1, producing the result [1, 1]. Finally, the tuple [1, 1] is used as an expression in the call to f.

20.7 Construction

Tuple construction and destruction follow the same rules and semantics as tuple assignment, except that in the case where there is no right side, the default constructor or destructor is called on each component of the tuple. As constructors and destructors did not exist in previous versions of C∀ or in K-W C, this is a primary contribution of this thesis to the design of tuples.

```
struct S;
void ?{}(S *);                    // (1)
void ?{}(S *, int);               // (2)
void ?{}(S * double);             // (3)
void ?{}(S *, S);                 // (4)

[S, S] x = [3, 6.28];             // uses (2), (3), specialized constructors
[S, S] y;                         // uses (1), (1), default constructor
[S, S] z = x.0;                   // uses (4), (4), copy constructor
```

In this example, x is initialized by the multiple constructor calls ?{}(&x.0, 3) and ?{}(&x.1, 6.28), while y is initialized by two default constructor calls ?{}(&y.0) and ?{}(&y.1). z is initialized by mass copy constructor calls ?{}(&z.0, x.0) and ?{}(&z.1, x.0). Finally, x, y, and z are destructed, *i.e.*, the calls ^?{}(&x.0), ^?{}(&x.1), ^?{}(&y.0), ^?{}(&y.1), ^?{}(&z.0), and ^?{}(&z.1).

It is possible to define constructors and assignment functions for tuple types that provide new semantics, if the existing semantics do not fit the needs of an application. For example, the function **void** ?{}([T, U] *, S); can be defined to allow a tuple variable to be constructed from a value of type S.

```
struct S { int x; double y; };
void ?{}([int, double] * this, S s) {
    this−>0 = s.x;
    this−>1 = s.y;
}
```

Due to the structure of generated constructors, it is possible to pass a tuple to a generated constructor for a type with a member prefix that matches the type of the tuple. For example,

```
struct S { int x; double y; int z };
[int, double] t;
S s = t;
```

The initialization of s with t works by default because t is flattened into its components, which satisfies the generated field constructor ?{}(S *, **int**, **double**) to initialize the first two values.

20.8 Member-Access Expression

Tuples may be used to select multiple fields of a record by field name. The result is a single tuple-valued expression whose type is the tuple of the types of the members. For example,

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

Here, the type of s.[x, y, z] is [**char**, **int**, **double**]. A member tuple expression has the form *e*.[x, y, z]; where *e* is an expression with type T, where T supports member access expressions, and x, y, z are all members of T with types T$_x$, T$_y$, and T$_z$ respectively. Then the type of *e*.[x, y, z] is [T$_x$, T$_y$, T$_z$].

A member-access tuple may be used anywhere a tuple can be used, *e.g.*:

```
s.[ y, z, x ] = [ 3, 3.2, 'x' ];          // equivalent to s.x = 'x', s.y = 3, s.z = 3.2
f( s.[ y, z ] );                          // equivalent to f( s.y, s.z )
```

Note, the fields appearing in a record-field tuple may be specified in any order; also, it is unnecessary to specify all the fields of a struct in a multiple record-field tuple.

Since tuple-index expressions are a form of member-access expression, it is possible to use tuple-index expressions in conjunction with member-access expressions to restructure a tuple (*e.g.*, rearrange components, drop components, duplicate components, *etc*.).

```
[ int, int, long, double ] x;
void f( double, long );

f( x.[ 0, 3 ] );                          // f( x.0, x.3 )
x.[ 0, 1 ] = x.[ 1, 0 ];                  // [ x.0, x.1 ] = [ x.1, x.0 ]
[ long, int, long ] y = x.[ 2, 0, 2 ];
```

It is possible for a member tuple expression to contain other member access expressions, *e.g.*:

```
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 ];
```

This expression is equivalent to [v.x, [v.y.i, v.y.j], v.z.k]. That is, the aggregate expression is effectively distributed across the tuple allowing simple and easy access to multiple components in an aggregate without repetition. It is guaranteed that the aggregate expression to the left of the . in a member tuple expression is evaluated exactly once. As such, it is safe to use member tuple expressions on the result of a function with side-effects.

```
[ int, float, double ] f();
[ double, float ] x = f().[ 2, 1 ];       // f() called once
```

In K-W C, member tuple expressions are known as *record field tuples* [31]. Since C∀ permits these tuple-access expressions using structures, unions, and tuples, *member tuple expression* or *field tuple expression* is more appropriate.

20.9 Casting

In C, the cast operator is used to explicitly convert between types. In C∀, the cast operator has a secondary use, which is type ascription, since it forces the expression resolution algorithm to choose the lowest cost conversion to the target type. That is, a cast can be used to select the type of an expression when it is ambiguous, as in the call to an overloaded function.

```
int f();                                  // (1)
double f();                               // (2)

f();                                      // ambiguous - (1),(2) both equally viable
(int)f();                                 // choose (2)
```

Since casting is a fundamental operation in C∀, casts need to be given a meaningful interpretation in the context of tuples. Taking a look at standard C provides some guidance with respect to the way casts should work with tuples.

```
int f();
void g();

(void)f();                                // valid, ignore results
```

```
5  (int)g();                              // invalid, void cannot be converted to int
6
7  struct A { int x; };
8  (struct A)f();                         // invalid, int cannot be converted to A
```

In C, line 4 is a valid cast, which calls f and discards its result. On the other hand, line 5 is invalid, because g does not produce a result, so requesting an **int** to materialize from nothing is nonsensical. Finally, line 8 is also invalid, because in C casts only provide conversion between scalar types [21, p. 91]. For consistency, this implies that any case wherein the number of components increases as a result of the cast is invalid, while casts that have the same or fewer number of components may be valid.

Formally, a cast to tuple type is valid when $T_n \leq S_m$, where T_n is the number of components in the target type and S_m is the number of components in the source type, and for each i in $[0,n)$, S_i can be cast to T_i. Excess elements (S_j for all j in $[n,m)$) are evaluated, but their values are discarded so that they are not included in the result expression. This discarding naturally follows the way that a cast to void works in C.

For example,

```
[int, int, int] f();
[int, [int, int], int] g();

([int, double])f();                  // (1) valid
([int, int, int])g();                // (2) valid
([void, [int, int]])g();             // (3) valid
([int, int, int, int])g();           // (4) invalid
([int, [int, int, int]])g();         // (5) invalid
```

(1) discards the last element of the return value and converts the second element to type double. Since **int** is effectively a 1-element tuple, (2) discards the second component of the second element of the return value of g. If g is free of side effects, this is equivalent to [(**int**)(g().0), (**int**)(g().1.0), (**int**)(g().2)]. Since **void** is effectively a 0-element tuple, (3) discards the first and third return values, which is effectively equivalent to [(**int**)(g().1.0), (**int**)(g().1.1)]). Note that a cast is not a function call in C∀, so flattening and structuring conversions do not occur for cast expressions. As such, (4) is invalid because the cast target type contains 4 components, while the source type contains only 3. Similarly, (5) is invalid because the cast ([**int**, **int**, **int**])(g().1) is invalid. That is, it is invalid to cast [**int**, **int**] to [**int**, **int**, **int**].

20.10 Polymorphism

Due to the implicit flattening and structuring conversions involved in argument passing, otype and dtype parameters are restricted to matching only with non-tuple types. The integration of polymorphism, type assertions, and monomorphic specialization of tuple-assertions are a primary contribution of this thesis to the design of tuples.

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

f([5, "hello"]);
```

In this example, [5, "hello"] is flattened, so that the argument list appears as 5, "hello". The argument matching algorithm binds T to **int** and U to **const char**, and calls the function as normal.

Tuples can contain polymorphic types. For example, a plus operator can be written to add two triples of a type together.

```
forall(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]);
```

Note that due to the implicit tuple conversions, this function is not restricted to the addition of two triples. A call to this plus operator type checks as long as a total of 6 non-tuple arguments are passed after flattening, and all of the arguments have a common type that can bind to T, with a pairwise ?+? over T. For example, these expressions also succeed and produce the same value.

```
([x.0, x.1]) + ([x.2, 10, 20, 30]);  // x + ([10, 20, 30])
```

```
x.0 + ([x.1, x.2, 10, 20, 30]);    // x + ([10, 20, 30])
```

This presents a potential problem if structure is important, as these three expressions look like they should have different meanings. Furthermore, these calls can be made ambiguous by introducing seemingly different functions.

```
forall(T | { T ?+?(T, T); })
[T, T, T] ?+?([T, T] x, [T, T, T, T]);
forall(T | { T ?+?(T, T); })
[T, T, T] ?+?(T x, [T, T, T, T, T]);
```

It is also important to note that these calls could be disambiguated if the function return types were different, as they likely would be for a reasonable implementation of ?+?, since the return type is used in overload resolution. Still, these semantics are a deficiency of the current argument matching algorithm, and depending on the function, differing return values may not always be appropriate. These issues could be rectified by applying an appropriate conversion cost to the structuring and flattening conversions, which are currently 0-cost conversions in the expression resolver. Care would be needed in this case to ensure that exact matches do not incur such a cost.

```
void f([int, int], int, int);

f([0, 0], 0, 0);                    // no cost
f(0, 0, 0, 0);                      // cost for structuring
f([0, 0,], [0, 0]);                 // cost for flattening
f([0, 0, 0], 0);                    // cost for flattening and structuring
```

Until this point, it has been assumed that assertion arguments must match the parameter type exactly, modulo polymorphic specialization (*i.e.*, no implicit conversions are applied to assertion arguments). This decision presents a conflict with the flexibility of tuples.

20.10.1 Assertion Inference

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

If assertion arguments must match exactly, then the call to g cannot be resolved, since the expected type of f is flat, while the only f in scope requires a tuple type. Since tuples are fluid, this requirement reduces the usability of tuples in polymorphic code. To ease this pain point, function parameter and return lists are flattened for the purposes of type unification, which allows the previous example to pass expression resolution.

This relaxation is made possible by extending the existing thunk generation scheme, as described by Bilson [2]. Now, whenever a candidate's parameter structure does not exactly match the formal parameter's structure, a thunk is generated to specialize calls to the actual function.

```
int _thunk(int _p0, double _p1, double _p2) {
    return f([_p0, _p1], _p2);
}
```

Essentially, this provides flattening and structuring conversions to inferred functions, improving the compatibility of tuples and polymorphism.

21 Tuples

In C and C∀, lists of elements appear in several contexts, such as the parameter list for a routine call. (More contexts are added shortly.) A list of such elements is called a *lexical list*. The general syntax of a lexical list is:

```
[ exprlist ]
```

where *exprlist* is a list of one or more expressions separated by commas. The brackets, [], allow differentiating between lexical lists and expressions containing the C comma operator. The following are examples of lexical lists:

```
[ x, y, z ]
[ 2 ]
[ v + w, x * y, 3.14159, f() ]
```

Tuples are permitted to contain sub-tuples (*i.e.*, nesting), such as [[14, 21], 9], which is a 2-element tuple whose first element is itself a tuple. Note, a tuple is not a record (structure); a record denotes a single value with substructure, whereas a tuple is multiple values with no substructure (see flattening coercion in Section 20.5, p. 40). In essence, tuples are largely a compile time phenomenon, having little or no runtime presence.

Tuples can be organized into compile-time tuple variables; these variables are of *tuple type*. Tuple variables and types can be used anywhere lists of conventional variables and types can be used. The general syntax of a tuple type is:

```
[ typelist ]
```

where *typelist* is a list of one or more legal C∀ or C type specifications separated by commas, which may include other tuple type specifications. Examples of tuple types include:

```
[ unsigned int, char ]
[ double, double, double ]
[ * int, int * ]                      // mix of CFA and ANSI
[ * [ 5 ] int, * * char, * [ [ int, int ] ] (int, int) ]
```

Like tuples, tuple types may be nested, such as [[**int**, **int**], **int**], which is a 2-element tuple type whose first element is itself a tuple type.

Examples of declarations using tuple types are:

```
[ int, int ] x;                       // 2 element tuple, each element of type int
* [ char, char ] y;                   // pointer to a 2 element tuple
[ [ int, int ] ] z ([ int, int ]);
```

The last example declares an external routine that expects a 2 element tuple as an input parameter and returns a 2 element tuple as its result.

As mentioned, tuples can appear in contexts requiring a list of value, such as an argument list of a routine call. In unambiguous situations, the tuple brackets may be omitted, *e.g.*, a tuple that appears as an argument may have its square brackets omitted for convenience; therefore, the following routine invocations are equivalent:

```
f( [ 1, x+2, fred() ] );
f( 1, x+2, fred() );
```

Also, a tuple or a tuple variable may be used to supply all or part of an argument list for a routine expecting multiple input parameters or for a routine expecting a tuple as an input parameter. For example, the following are all legal:

```
[ int, int ] w1;
[ int, int, int ] w2;
[ void ] f (int, int, int);           // three input parameters of type int
[ void ] g ([ int, int, int ]);       // 3 element tuple as input
f( [ 1, 2, 3 ] );
f( w1, 3 );
f( 1, w1 );
f( w2 );
g( [ 1, 2, 3 ] );
g( w1, 3 );
g( 1, w1 );
g( w2 );
```

Note, in all cases 3 arguments are supplied even though the syntax may appear to supply less than 3. As mentioned, a tuple does not have structure like a record; a tuple is simply converted into a list of components.

□ The present implementation of C∀ does not support nested routine calls when the inner routine returns multiple values; *i.e.*, a statement such as g(f()) is not supported. Using a temporary variable to store the results of the inner routine and then passing this variable to the outer routine works, however. □

A tuple can contain a C comma expression, provided the expression containing the comma operator is enclosed in parentheses. For instance, the following tuples are equivalent:

```
[ 1, 3, 5 ]
[ 1, (2, 3), 5 ]
```

The second element of the second tuple is the expression (2, 3), which yields the result 3. This requirement is the same as for comma expressions in argument lists.

Type qualifiers, *i.e.*, **const** and **volatile**, may modify a tuple type. The meaning is to distribute the qualifier across all of the types in the tuple, *e.g.*:

```
const volatile [ int, float, const int ] x;
```

is equivalent to:

```
[ const volatile int, const volatile float, const volatile int ] x;
```

Declaration qualifiers can only appear at the start of a C∀ tuple declaration4, *e.g.*:

```
extern [ int, int ] w1;
static [ int, int, int ] w2;
```

□ Unfortunately, C's syntax for subscripts precluded treating them as tuples. The C subscript list has the form [i][j]... and not [i, j, ...]. Therefore, there is no syntactic way for a routine returning multiple values to specify the different subscript values, *e.g.*, f[g()] always means a single subscript value because there is only one set of brackets. Fixing this requires a major change to C because the syntactic form M[i, j, k] already has a particular meaning: i, j, k is a comma expression. □

21.1 Tuple Coercions

There are four coercions that can be performed on tuples and tuple variables: closing, opening, flattening and structuring. In addition, the coercion of dereferencing can be performed on a tuple variable to yield its value(s), as for other variables. A *closing coercion* takes a set of values and converts it into a tuple value, which is a contiguous set of values, as in:

```
[ int, int, int, int ] w;
w = [ 1, 2, 3, 4 ];
```

First the right-hand tuple is closed into a tuple value and then the tuple value is assigned.

An *opening coercion* is the opposite of closing; a tuple value is converted into a tuple of values, as in:

```
[ a, b, c, d ] = w
```

w is implicitly opened to yield a tuple of four values, which are then assigned individually.

A *flattening coercion* coerces a nested tuple, *i.e.*, a tuple with one or more components, which are themselves tuples, into a flattened tuple, which is a tuple whose components are not tuples, as in:

```
[ a, b, c, d ] = [ 1, [ 2, 3 ], 4 ];
```

First the right-hand tuple is flattened and then the values are assigned individually. Flattening is also performed on tuple types. For example, the type [**int**, [**int**, **int**], **int**] can be coerced, using flattening, into the type [**int**, **int**, **int**, **int**].

A *structuring coercion* is the opposite of flattening; a tuple is structured into a more complex nested tuple. For example, structuring the tuple [1, 2, 3, 4] into the tuple [1, [2, 3], 4] or the tuple type [**int**, **int**, **int**, **int**] into the tuple type [**int**, [**int**, **int**], **int**]. In the following example, the last assignment illustrates all the tuple coercions:

```
[ int, int, int, int ] w = [ 1, 2, 3, 4 ];
int x = 5;
[ x, w ] = [ w, x ];                    // all four tuple coercions
```

Starting on the right-hand tuple in the last assignment statement, w is opened, producing a tuple of four values; therefore, the right-hand tuple is now the tuple [[1, 2, 3, 4], 5]. This tuple is then flattened, yielding [1, 2, 3, 4, 5], which is structured into [1, [2, 3, 4, 5]] to match the tuple type of the left-hand side. The tuple [2, 3, 4, 5] is then closed to create a tuple value. Finally, x is assigned 1 and w is assigned the tuple value using multiple assignment (see Section 20.6, p. 41).

□ A possible additional language extension is to use the structuring coercion for tuples to initialize a complex record with a tuple. □

21.2 Mass Assignment

C∀ permits assignment to several variables at once using mass assignment [25]. Mass assignment has the following form:

```
[ lvalue, ... , lvalue ] = expr;
```

The left-hand side is a tuple of *lvalues*, which is a list of expressions each yielding an address, *i.e.*, any data object that can appear on the left-hand side of a conventional assignment statement. *expr* is any standard arithmetic expression. Clearly, the types of the entities being assigned must be type compatible with the value of the expression.

Mass assignment has parallel semantics, *e.g.*, the statement:

```
[ x, y, z ] = 1.5;
```

is equivalent to:

```
x = 1.5; y = 1.5; z = 1.5;
```

This semantics is not the same as the following in C:

```
x = y = z = 1.5;
```

as conversions between intermediate assignments may lose information. A more complex example is:

```
[ i, y[i], z ] = a + b;
```

which is equivalent to:

```
t = a + b;
a1 = &i; a2 = &y[i]; a3 = &z;
*a1 = t; *a2 = t; *a3 = t;
```

The temporary t is necessary to store the value of the expression to eliminate conversion issues. The temporaries for the addresses are needed so that locations on the left-hand side do not change as the values are assigned. In this case, y[i] uses the previous value of i and not the new value set at the beginning of the mass assignment.

21.3 Multiple Assignment

C∀ also supports the assignment of several values at once, known as multiple assignment [25, 16]. Multiple assignment has the following form:

```
[ lvalue, ... , lvalue ] = [ expr, ... , expr ];
```

The left-hand side is a tuple of *lvalues*, and the right-hand side is a tuple of *expr*s. Each *expr* appearing on the right-hand side of a multiple assignment statement is assigned to the corresponding *lvalues* on the left-hand side of the statement using parallel semantics for each assignment. An example of multiple assignment is:

```
[ x, y, z ] = [ 1, 2, 3 ];
```

Here, the values 1, 2 and 3 are assigned, respectively, to the variables x, y and z. A more complex example is:

```
[ i, y[ i ], z ] = [ 1, i, a + b ];
```

Here, the values 1, i and a + b are assigned to the variables i, y[i] and z, respectively. Note, the parallel semantics of multiple assignment ensures:

```
[ x, y ] = [ y, x ];
```

correctly interchanges (swaps) the values stored in x and y. The following cases are errors:

```
[ a, b, c ] = [ 1, 2, 3, 4 ];
[ a, b, c ] = [ 1, 2 ];
```

because the number of entities in the left-hand tuple is unequal with the right-hand tuple.

As for all tuple contexts in C, side effects should not be used because C does not define an ordering for the evaluation of the elements of a tuple; both these examples produce indeterminate results:

```
f( x++, x++ );                    // C routine call with side effects in arguments
[ v1, v2 ] = [ x++, x++ ];        // side effects in right-hand side of multiple assignment
```

21.4 Cascade Assignment

As in C, C∀ mass and multiple assignments can be cascaded, producing cascade assignment. Cascade assignment has the following form:

```
tuple = tuple = ... = tuple;
```

and it has the same parallel semantics as for mass and multiple assignment. Some examples of cascade assignment are:

```
x1 = y1 = x2 = y2 = 0;
[ x1, y1 ] = [ x2, y2 ] = [ x3, y3 ];
[ x1, y1 ] = [ x2, y2 ] = 0;
[ x1, y1 ] = z = 0;
```

As in C, the rightmost assignment is performed first, *i.e.*, assignment parses right to left.

22 Stream I/O Library

The goal of C∀ stream input/output (I/O) is to simplify the common cases, while fully supporting polymorphism and user defined types in a consistent way. Stream I/O can be implicitly or explicitly formatted. Implicit formatting means C∀ selects an I/O format for values that matches a variable's type. Explicit formatting means additional I/O information is specified to control how a value is interpreted.

C∀ formatting incorporates ideas from C printf, C++ stream manipulators, and Python implicit spacing and newline. Specifically:

- printf/Python format codes are dense, making them difficult to read and remember. C∀/C++ format manipulators are named, making them easier to read and remember.
- printf/Python separate format codes from associated variables, making it difficult to match codes with variables. C∀/C++ co-locate codes with associated variables, where C∀ has the tighter binding.
- Format manipulators in printf/Python/C∀ have local effect, whereas C++ have global effect, except setw. Hence, it is common C++ programming practice to toggle manipulators on and then back to the default to prevent downstream side-effects. Without this programming style, errors occur when moving prints, as manipulator effects incorrectly flow into the new location. Furthermore, to guarantee no side-effects, manipulator values must be saved and restored across function calls. C++ programers never do any of this.
- C∀ has more sophisticated implicit value spacing than Python, plus implicit newline at the end of a print.

22.1 Basic I/O

The standard polymorphic I/O streams are stdin/sin (input), stdout/sout, and stderr/serr (output) (like C++ cin/cout/cerr). The standard I/O operator is the bit-wise (or) operator, `'|'`, which is used to cascade multiple I/O operations. The C∀ header file for the I/O library is fstream.hfa.

22.1.1 Stream Output

For implicit formatted output, the common case is printing a series of variables separated by whitespace.

C∀	C++	Python

`int x = 1, y = 2, z = 3;`
`sout | x | y | z;`
	`cout << x << " " << y << " " << z << endl;`	`x = 1; y = 2; z = 3`
		`print( x, y, z )`

`1␣2␣3`	`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 and newline. Similar simplification occurs for tuple I/O, which flattens the tuple and prints each value separated by ",␣" (comma space).

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

The bit-wise | operator is used for I/O, rather C++ shift-operators, << and >>, as it is the lowest-priority *overloadable* operator, other than assignment. (Operators || and && are not overloadable in C∀.) Therefore, fewer output expressions require parenthesis.

```
C∀:  sout | x * 3 | y + 1 | z << 2 | x == y | (x | y) | (x || y) | (x > z ? 1 : 2);
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 flows in the correct direction for input but the opposite direction for output. Input and output use a uniform operator, |, rather than C++'s << and >> input/output operators to prevent this common error in C++:

```
cin << i; // why is this generating a lot of error messages?
```

Streams exit and abort provide output with immediate program termination without and with generating a stack trace and core file. Stream exit implicitly returns EXIT_FAILURE to the shell.

```
exit  | "x (" | x | ") negative value.";   // print, terminate, and return EXIT_FAILURE to shell
abort | "x (" | x | ") negative value.";   // print, terminate, and generate stack trace and core file
```

Note, C∀ stream variables stdin, stdout, stderr, exit, and abort overload C variables stdin, stdout, stderr, and functions exit and abort, respectively.

22.1.2 Stream Input

For implicit formatted input, the common case is reading a sequence of values separated by whitespace, where the type of an input constant must match with the type of the input variable.

```
char c;   int i;   double d
        C∀                   C++                                Python
sin | c | i | d;          cin >> c >> i >> d;         c = input();   i = int(input());   d = float(input());
A␣1␣2.5                   A␣1␣2.5                     A
                                                      1
                                                      2.5
```

The format of numeric input values in the same as C constants without a trailing type suffix, as the input value-type is denoted by the input variable. For bool type, the constants are true and false. For integral types, any number of digits, optionally preceded by a sign (+ or –), where a

- 1-9 prefix introduces a decimal value (0-9),
- 0 prefix introduces an octal value (0-7), and
- 0x or 0X prefix introduces a hexadecimal value (0-f) with lower or upper case letters.

For floating-point types, any number of decimal digits, optionally preceded by a sign (+ or –), optionally containing a decimal point, and optionally followed by an exponent, e or E, with signed (optional) decimal digits. Floating-point values can also be written in hexadecimal format preceded by 0x or 0X with hexadecimal digits and exponent denoted by p or P. In all cases, whitespace characters are skipped until an appropriate value is found.

```
char ch;  int i;  float f; double d;  _Complex double cxd;
sin | ch | i | f | d | cxd;
X   42   1234.5     0xfffp−2    3.5+7.1i
```

It is also possible to scan and ignore specific strings and whitespace using a string format.

```
sin | "abc def";                          // space matches arbitrary whitespace (2 blanks, 2 tabs)
abc␣␣␣␣def
```

A non-whitespace format character reads the next input character, compares the format and input characters, and if equal, the input character is discarded and the next format character is tested. Note, a single whitespace in the format string matches **any** quantity of whitespace characters from the stream (including none).

For the C-string type, the default input format is any number of **non-whitespace** characters. There is no escape character supported in an input string, but any Latin-1 character can be typed directly in the input string. For example, if the following non-whitespace output is redirected into a file by the shell:

```
sout | "\n\t\f\0234\x23";
```

it can be read back from the file by redirecting the file as input using:

```
char s[64];
sin | wdi( sizeof(s), s );                // must specify string size
```

The input string is always null terminated '\0' in the input variable. Because of potential buffer overrun when reading C strings, strings are restricted to work with input manipulators (see Section 22.6, p. 56). As well, there are multiple input-manipulators for scanning complex input string formats, *e.g.*, a quoted character or string.

In all cases, if an invalid data value is not found for a type or format string, the exception missing_data is raised and the input variable is unchanged. For example, when reading an integer and the string "abc" is found, the exception missing_data is raised to ensure the program does not proceed erroneously. If a valid data value is found, but it is larger than the capacity of the input variable, such reads are undefined.

22.1.3 Stream Files

Figure 6 shows the I/O stream operations for interacting with files other than sin, sout, and cerr.

- fail tests the stream error-indicator, returning nonzero if it is set.
- clear resets the stream error-indicator.

```
// *********************************** ofstream ***********************************
bool fail( ofstream & );
void clear( ofstream & );
int flush( ofstream & );
void open( ofstream &, const char name[], const char mode[] = "w" );
void close( ofstream & );
ofstream & write( ofstream &, const char data[], size_t size );
void ?{}( ofstream & );
void ?{}( ofstream &, const char name[], const char mode[] = "w" );
void ^?{}( ofstream & );

// *********************************** ifstream ***********************************
bool fail( ifstream & is );
void clear( ifstream & );
bool eof( ifstream & is );
void open( ifstream & is, const char name[], const char mode[] = "r" );
void close( ifstream & is );
ifstream & read( ifstream & is, char data[], size_t size );
ifstream & ungetc( ifstream & is, char c );
void ?{}( ifstream & is );
void ?{}( ifstream & is, const char name[], const char mode[] = "r" );
void ^?{}( ifstream & is );
```

Figure 6: I/O Stream Functions

- flush (ofstream only) causes any unwritten data for a stream to be written to the file.
- eof (ifstream only) tests the end-of-file indicator for the stream pointed to by stream. Returns true if the end-of-file indicator is set, otherwise false.
- open binds the file with name to a stream accessed with mode (see fopen).
- close flushes the stream and closes the file.
- write (ofstream only) writes size bytes to the stream. The bytes are written lazily when an internal buffer fills. Eager buffer writes are done with flush
- read (ifstream only) reads size bytes from the stream.
- ungetc (ifstream only) pushes the character back to the input stream. Pushed-back characters returned by subsequent reads in the reverse order of pushing.

The constructor functions:

- create an unbound stream, which is subsequently bound to a file with open.
- create a bound stream to the associated file with given mode.

The destructor closes the stream.

Figure 7 demonstrates the file operations by showing the idiomatic C∀ command-line processing and copying an input file to an output file. Note, a stream variable may be copied because it is a reference to an underlying stream data-structures. **All I/O errors are handled as exceptions**, but end-of-file is not an exception as C programmers are use to explicitly checking for it.

22.2 Implicit Separator

The implicit separator character (space/blank) is a separator not a terminator for output. The rules for implicitly adding a separator are:

1. A separator does not appear at the start or end of a line.

```
sout | 1 | 2 | 3;
1␣2␣3
```

2. A separator does not appear before or after a character literal or variable.

```
sout | '1' | '2' | '3';
```

```
#include <fstream.hfa>

int main( int argc, char * argv[] ) {
    ifstream in  = stdin;                  // copy default files
    ofstream out = stdout;

    try {
        choose ( argc ) {
          case 3, 2:
            open( in, argv[1] );           // open input file first as output creates file
            if ( argc == 3 ) open( out, argv[2] ); // do not create output unless input opens
          case 1: ;                        // use default files
          default:
            exit | "Usage" | argv[0] | "[ input-file (default stdin) "
                   "[ output-file (default stdout) ] ]";
        } // choose
    } catch( open_failure * ex; ex->istream == &in ) { // input file errors
        exit | "Unable to open input file" | argv[1];
    } catch( open_failure * ex; ex->ostream == &out ) { // output file errors
        close( in );                       // optional
        exit | "Unable to open output file" | argv[2];
    } // try

    out | nlOff;                           // turn off auto newline
    in | nlOn;                             // turn on reading newline
    char ch;
    for () {                               // read/write characters
        in | ch;
      if ( eof( in ) ) break;              // eof ?
        out | ch;
    } // for
} // main
```

Figure 7: C∀ Command-Line Processing

```
123
```

3. A separator does not appear before or after a null (empty) C string, which is a local mechanism to disable insertion of the separator character.

```
sout | 1 | "" | 2 | "" | 3;
123
```

4. A separator does not appear before a C string starting with the (extended) ASCII characters: `, . ; ! ? ) ] } % ¢ »`, where » is a closing citation mark.

```
sout | 1 | ", x" | 2 | ". x" | 3 | "; x" | 4 | "! x" | 5 | "? x" | 6 | "% x"
     | 7 | "¢ x" | 8 | "» x" | 9 | ") x" | 10 | "] x" | 11 | "} x";
```

```
Input1,␣x␣2.␣x␣3;␣x␣4!␣x␣5?␣x␣6%␣x␣7¢␣x␣8»␣x␣9)␣x␣10]␣x␣11}␣x
```

5. A separator does not appear after a C string ending with the (extended) ASCII characters: `( [ { = $ £ ¥ ¡ ¿ «`, where ¡ ¿ are inverted opening exclamation and question marks, and « is an opening citation mark.

```
sout | "x (" | 1 | "x [" | 2 | "x {" | 3 | "x =" | 4 | "x $" | 5 | "x £" | 6 | "x ¥"
     | 7 | "x ¡" | 8 | "x ¿" | 9 | "x «" | 10;
```

```
x␣(1␣x␣[2␣x␣{3␣x␣=4␣x␣$5␣x␣£6␣x␣¥7␣x␣¡8␣x␣¿9␣x␣«10
```

6. A separator does not appear before/after a C string starting/ending with the ASCII quote or whitespace characters: `` `'": ␣\t\v\f\r\n ``

```
sout | "x`" | 1 | "`x'" | 2 | "'x\"" | 3 | "\"x:" | 4 | ":x " | 5 | " x\t" | 6 | "\tx";
```

```
    x`1`x'2'x"3"x:4:x␣5␣x␣␣6␣␣␣x
```

7. If a space is desired before or after one of the special string start/end characters, explicitly insert a space.

```
    sout | "x  (␣" | 1 | "␣)  x" | 2 | "␣,  x" | 3 | "␣:x:␣" | 4;
    x␣(␣1␣)␣x␣2␣,␣x␣3␣:x:␣4
```

22.3 Separation Manipulators

The following manipulators control implicit output separation. The effect of these manipulators is global for an output stream (except sep and nosep).

1. sepSet and sepVal/sepGet set and get the separator string. The separator string can be at most 16 characters including the '\0' string terminator (15 printable characters).

```
    sepSet( sout, ",  $" );             // set separator from " " to ", $"
    sout | 1 | 2 | 3 | "  \"" | sepVal | "\"";
    1,␣$2,␣$3␣",␣$"

    sepSet( sout, "  " );               // reset separator to " "
    sout | 1 | 2 | 3 | "  \"" | sepGet( sout ) | "\"";
    1␣2␣3␣"␣"
```

sepGet can be used to store a separator and then restore it:

```
    char store[sepSize];                // sepSize is the maximum separator size
    strcpy( store, sepGet( sout ) );    // copy current separator
    sepSet( sout, "_" );                // change separator to underscore
    sout | 1 | 2 | 3;
    1_2_3

    sepSet( sout, store );              // change separator back to original
    sout | 1 | 2 | 3;
    1␣2␣3
```

2. sepSetTuple and sepTupleVal/sepGetTuple get and set the tuple separator-string. The tuple separator-string can be at most 16 characters including the '\0' string terminator (15 printable characters).

```
    sepSetTuple( sout, "  " );          // set tuple separator from ", " to " "
    sout | t1 | t2 | "  \"" | sepTupleVal | "\"";
    1␣2␣3␣4␣5␣6␣"␣"

    sepSetTuple( sout, ",  " );         // reset tuple separator to ", "
    sout | t1 | t2 | "  \"" | sepGetTuple( sout ) | "\"";
    1,␣2,␣3␣4,␣5,␣6␣",␣"
```

As for sepGet, sepGetTuple can be use to store a tuple separator and then restore it.

3. sepOff and sepOn globally toggle printing the separator.

```
    sout | sepOff | 1 | 2 | 3;          // turn off implicit separator
    123

    sout | sepOn | 1 | 2 | 3;           // turn on implicit separator
    1␣2␣3
```

4. sep and nosep locally toggle printing the separator with respect to the next printed item, and then return to the global separator setting.

```
    sout | 1 | nosep | 2 | 3;           // turn off implicit separator for the next item
    12␣3

    sout | sepOff | 1 | sep | 2 | 3;    // turn on implicit separator for the next item
    1␣23
```

The tuple separator also responses to being turned on and off.

```
    sout | t1 | nosep | t2;             // turn off implicit separator for the next item
    1,␣2,␣34,␣5,␣6
```

sep *cannot* be used to start/end a line with a separator because separators do not appear at the start/end of a line. Use sep to accomplish this functionality.

```
sout | sep | 1 | 2 | 3 | sep;          // sep does nothing at start/end of line
1␣2␣3
sout | sepVal | 1 | 2 | 3 | sepVal ; // use sepVal to print separator at start/end of line
␣1␣2␣3␣
```

22.4 Newline Manipulators

The following manipulators control newline separation for input and output.

For input:

1. nlOn reads the newline character, when reading single characters.
2. nlOff does *not* read the newline character, when reading single characters.
3. nl scans characters until the next newline character, *i.e.*, ignore the remaining characters in the line. If nlOn is enabled, the nl is also consumed.

For example, in:

```
int i, j;
sin | i | nl | j;
1 2
3
```

variable i is assigned 1, the 2 is skipped, and variable j is assigned 3. For example, in:

```
char ch
                              sin | nlOn; // enable reading newlines
sin | ch; // read X           sin | ch; // read newline

X
```

the left example skips the newline and reads 'X' into ch, while the right example reads the newline into ch.

For output:

1. nlOn implicitly prints a newline at the end of each output expression.
2. nlOff does *not* implicitly print a newline at the end of each output expression.
3. nl inserts a newline.

```
sout | nl;                    // only print newline
sout | 2;                     // implicit newline
sout | 3 | nl | 4 | nl;       // terminating nl merged with implicit newline
sout | 5 | nl | nl;           // again terminating nl merged with implicit newline
sout | 6;                     // implicit newline

2
3
4
5

6
```

Note, a terminating nl is merged with (overrides) the implicit newline at the end of the sout expression, otherwise it is impossible to print a single newline

22.5 Output Manipulators

The following manipulators control formatting (printing) of the argument output values.

1. bin(integer) print value in base 2 preceded by 0b/0B.

```
sout | bin( 0 ) | bin( 27HH ) | bin( 27H ) | bin( 27 ) | bin( 27L );
0b0 0b11011 0b11011 0b11011 0b11011
sout | bin( -27HH ) | bin( -27H ) | bin( -27 ) | bin( -27L );
0b11100101 0b1111111111100101 0b11111111111111111111111111100101 0b(58 1s)100101
```

2. oct(integer) print value in base 8 preceded by 0.

```
sout | oct( 0 ) | oct( 27HH ) | oct( 27H ) | oct( 27 ) | oct( 27L );
0 033 033 033 033
sout | oct( -27HH ) | oct( -27H ) | oct( -27 ) | oct( -27L );
0345 0177745 037777777745 01777777777777777777745
```

Note, octal 0 is *not* preceded by 0 to prevent confusion.

3. hex(integer / floating-point) print value in base 16 preceded by 0x/0X.

```
sout | hex( 0 ) | hex( 27HH ) | hex( 27H ) | hex( 27 ) | hex( 27L );
0x0 0x1b 0x1b 0x1b 0x1b
sout | hex( -27HH ) | hex( -27H ) | hex( -27 ) | hex( -27L );
0xe5 0xffe5 0xffffffe5 0xffffffffffffffe5

sout | hex( 0.0 ) | hex( 27.5F ) | hex( 27.5 ) | hex( 27.5L );
0x0p+0 0x1.b8p+4 0x1.b8p+4 0xd.cp+1
sout | hex( -27.5F ) | hex( -27.5 ) | hex( -27.5L );
-0x1.b8p+4 -0x1.b8p+4 -0xd.cp+1
```

4. sci(floating-point) print value in scientific notation with exponent. Default is 6 digits of precision.

```
sout | sci( 0.0 ) | sci( 27.5 ) | sci( -27.5 );
0.000000e+00 2.750000e+01 -2.750000e+01
```

5. eng(floating-point) print value in engineering notation with exponent, which means the exponent is adjusted to a multiple of 3.

```
sout | eng( 0.0 ) | eng( 27000.5 ) | eng( -27.5e7 );
0e0 27.0005e3 -275e6
```

6. unit(engineering-notation) print engineering exponent as a letter between the range 10^{-24} and 10^{24}:

   y $\Rightarrow 10^{-24}$, z $\Rightarrow 10^{-21}$, a $\Rightarrow 10^{-18}$, f $\Rightarrow 10^{-15}$, p $\Rightarrow 10^{-12}$, n $\Rightarrow 10^{-9}$, u $\Rightarrow 10^{-6}$, m $\Rightarrow 10^{-3}$, K $\Rightarrow 10^{3}$, M $\Rightarrow 10^{6}$, G $\Rightarrow 10^{9}$, T $\Rightarrow 10^{12}$, P $\Rightarrow 10^{15}$, E $\Rightarrow 10^{18}$, Z $\Rightarrow 10^{21}$, Y $\Rightarrow 10^{24}$.

   For exponent 10^0, no decimal point or letter is printed.

```
sout | unit(eng( 0.0 )) | unit(eng( 27000.5 )) | unit(eng( -27.5e7 ));
0 27.0005K -275M
```

7. upcase(bin / hex / floating-point) print letters in a value in upper case. Lower case is the default.

```
sout | upcase( bin( 27 ) ) | upcase( hex( 27 ) ) | upcase( 27.5e-10 ) | upcase( hex( 27.5 ) );
0B11011 0X1B 2.75E-09 0X1.B8P+4
```

8. nobase(integer) do not precede bin, oct, hex with 0b/0B, 0, or 0x/0X. Printing the base is the default.

```
sout | nobase( bin( 27 ) ) | nobase( oct( 27 ) ) | nobase( hex( 27 ) );
11011 33 1b
```

9. nodp(floating-point) do not print a decimal point if there are no fractional digits. Printing a decimal point is the default, if there are no fractional digits.

```
sout | 0. | nodp( 0. ) | 27.0 | nodp( 27.0 ) | nodp( 27.5 );
0.0 0 27.0 27 27.5
```

10. sign(integer / floating-point) prefix with plus or minus sign (+ or −). Only printing the minus sign is the default.

```
sout | sign( 27 ) | sign( -27 ) | sign( 27. ) | sign( -27. ) | sign( 27.5 ) | sign( -27.5 );
+27 -27 +27.0 -27.0 +27.5 -27.5
```

11. wd(minimum, value), wd(minimum, precision, value) For all types, minimum is the number of printed characters. If the value is shorter than the minimum, it is padded on the right with spaces.

```
sout | wd( 4, 34) | wd( 3, 34 ) | wd( 2, 34 );
sout | wd( 10, 4.) | wd( 9, 4. ) | wd( 8, 4. );
sout | wd( 4, "ab" ) | wd( 3, "ab" ) | wd( 2, "ab" );
␣␣34␣␣34␣34
␣␣4.000000␣␣4.000000␣4.000000
␣␣ab␣␣ab␣ab
```

   If the value is larger, it is printed without truncation, ignoring the minimum.

```
sout | wd( 4, 34567 ) | wd( 3, 34567 ) | wd( 2, 34567 );
sout | wd( 4, 3456. ) | wd( 3, 3456. ) | wd( 2, 3456. );
sout | wd( 4, "abcde" ) | wd( 3, "abcde" ) | wd( 2,"abcde" );
34567␣34567␣34567
3456.␣3456.␣3456.
abcde␣abcde␣abcde
```

For integer types, precision is the minimum number of printed digits. If the value is shorter, it is padded on the left with leading zeros.

```
sout | wd( 4,3, 34 ) | wd( 8,4, 34 ) | wd( 10,10, 34 );
␣034␣␣␣␣␣0034␣0000000034
```

If the value is larger, it is printed without truncation, ignoring the precision.

```
sout | wd( 4,1, 3456 ) | wd( 8,2, 3456 ) | wd( 10,3, 3456 );
3456␣␣␣␣␣3456␣␣␣␣␣␣␣3456
```

If precision is 0, nothing is printed for zero. If precision is greater than the minimum, it becomes the minimum.

```
sout | wd( 4,0, 0 ) | wd( 3,10, 34 );
␣␣␣␣␣0000000034
```

For floating-point types, precision is the minimum number of digits after the decimal point.

```
sout | wd( 6,3, 27.5 ) | wd( 8,1, 27.5 ) | wd( 8,0, 27.5 ) | wd( 3,8, 27.5 );
27.500␣␣␣␣␣27.5␣␣␣␣␣␣28.␣27.50000000
```

For the C-string type, precision is the maximum number of printed characters, so the string is truncated if it exceeds the maximum.

```
sout | wd( 6,8, "abcd" ) | wd( 6,8, "abcdefghijk" ) | wd( 6,3, "abcd" ) | wd( 10, "" ) | 'X';
␣␣abcd␣abcdefgh␣␣␣␣abc␣␣␣␣␣␣␣␣␣␣␣X
```

Note, printing the null string with minimum width L pads with L spaces.

12. ws(minimum, significant, floating-point) For floating-point types, minimum is the same as for manipulator wd, but significant is the maximum number of significant digits to be printed for both the integer and fractions (versus only the fraction for wd). If a value's significant digits is greater than significant, the last significant digit is rounded up.

    ```
    sout | ws(6,6, 234.567) | ws(6,5, 234.567) | ws(6,4, 234.567) | ws(6,3, 234.567);
    234.567␣234.57␣␣234.6␣␣␣␣235
    ```

    If a value's magnitude is greater than significant, the value is printed in scientific notation with the specified number of significant digits.

    ```
    sout | ws(6,6, 234567.) | ws(6,5, 234567.) | ws(6,4, 234567.) | ws(6,3, 234567.);
    234567.␣2.3457e+05␣2.346e+05␣2.35e+05
    ```

    If significant is greater than minimum, it defines the number of printed characters.

    ```
    sout | ws(3,6, 234567.) | ws(4,6, 234567.) | ws(5,6, 234567.) | ws(6,6, 234567.);
    234567.␣234567.␣234567.␣234567.
    ```

13. left(field-width) left justify within the given field.

    ```
    sout | left(wd(4, 27)) | left(wd(10, 27.)) | left(wd(10, 27.5)) | left(wd(4,3, 27)) | left(wd(10,3, 27.5));
    27␣␣␣27.000000␣␣27.500000␣␣027␣␣27.500␣␣␣␣
    ```

14. pad0(field-width) left pad with zeroes (0).

    ```
    sout | pad0( wd( 4, 27 ) ) | pad0( wd( 4,3, 27 ) ) | pad0( wd( 8,3, 27.5 ) );
    0027  027 0027.500
    ```

22.6 Input Manipulators

A string variable *must* be large enough to contain the input sequence. To force programmers to consider buffer overruns for C-string input, C-strings may only be read with a width field, which should specify a size less than or equal to the C-string size, *e.g.*:

```
char line[64];
sin | wdi( sizeof(line), line );                // must specify string size
```

Certain input manipulators support a *scanset*, which is a simple regular expression, where the matching set contains any Latin-1 character (8-bits) or character ranges using minus. For example, the scanset `"a-zA-Z -/?§"` matches any number of characters between `'a'` and `'z'`, between `'A'` and `'Z'`, between space and `'/'`, and characters `'?'` and (Latin-1) `'§'`. The following string is matched by this scanset:

```
!&%$  abAA () ZZZ  ??§  xx§§
```

To match a minus, make it the first character in the set, *e.g.*, `"-0-9"`. Other complex forms of regular-expression matching are unsupported.

The following manipulators control scanning of input values (reading) and only affect the format of the argument.

1. skip(*scanset*), skip(*N*) consumes either the *scanset* or the next *N* characters, including newlines. If the match successes, the input characters are ignored, and input continues with the next character. If the match fails, the input characters are left unread.

   ```
   char scanset[ ] = "abc";
   sin | "abc␣" | skip( scanset ) | skip( 5 ); // match and skip input sequence
   abc␣␣␣abc␣␣xxx
   ```

   Again, the blank in the format string `"abc␣"` matches any number of whitespace characters.

2. wdi(*maximum*, T & v) For all types except **char** *, whitespace is skipped and the longest sequence of non-whitespace characters matching an appropriate typed (T) value is read, converted into its corresponding internal form, and written into the T variable. *maximum* is the maximum number of characters read for the current value rather than the longest sequence.

   ```
   char ch;   char ca[3];   int i;   double d;
   sin | wdi( sizeof(ch), ch ) | wdi( sizeof(ca), ca[0] ) | wdi( 3, i ) | wdi( 8, d );  // c == 'a', ca == "bcd", i == 123, d == 345.6
   abcd1233.456E+2
   ```

   Here, ca[0] is type **char**, so the width reads 3 characters **without** a null terminator. If an input value is not found for a variable, the exception missing_data is raised, and the input variable is unchanged.

   Note, input wdi cannot be overloaded with output wd because both have the same parameters but return different types. Currently, C∀ cannot distinguish between these two manipulators in the middle of an sout/sin expression based on return type.

3. wdi(*maximum size*, **char** s[]) For type **char** *, whitespace is skippped and the longest sequence of non-whitespace characters is read, without conversion, and written into the string variable (null terminated). *maximum size* is the maximum number of characters in the string variable. If the non-whitespace sequence of input characters is greater than *maximum size* − 1 (null termination), the exception cstring_length is raised.

   ```
   char cs[10];
   sin | wdi( sizeof(cs), cs );
   012345678
   ```

   Nine non-whitespace character are read and the null character is added to make ten.

4. wdi(*maximum size*, *maximum read*, **char** s[]) This manipulator is the same as the previous one, except *maximum read* is the maximum number of characters read for the current value rather than the longest sequence, where *maximum read* $\leq$ *maximum size*.

   ```
   char cs[10];
   sin | wdi( sizeof(cs), 9, cs );
   0123456789
   ```

   The exception cstring_length is not raised, because the read stops reading after nine characters.

5. getline(*wdi manipulator*, **const char** delimiter = `'\n'`) consumes the scanset `"[^D]D"`, where D is the delimiter character, which reads all characters from the current input position to the delimiter character into the string (null terminated), and consumes and ignores the delimiter. If the delimiter character is omitted, it defaults to `'\n'` (newline).

   ```
   char cs[10];
   sin | getline( wdi( sizeof(cs), cs ) );
   sin | getline( wdi( sizeof(cs), cs ), 'X' ); // X is the line delimiter
   abc␣??␣#@%
   abc␣??␣#@%X␣w
   ```

   The same value is read for both input strings.

6. quoted(**char** & ch, **const char** Ldelimiter = `'\''`, **const char** Rdelimiter = `'\0'`) consumes the string `"LCR"`, where L is the left delimiter character, C is the value in ch, and R is the right delimiter character, which skips whites-

pace, consumes and ignores the left delimiter, reads a single character into ch, and consumes and ignores the right delimiter (3 characters). If the delimit character is omitted, it defaults to '\'' (single quote).

```
char ch;
sin | quoted( ch );   sin | quoted( ch, '"' );   sin | quoted( ch, '[', ']' );
␣␣␣'a'␣␣"a"[a]
```

7. quoted(*wdi manipulator*, **const char** Ldelimiter = '\'', **const char** Rdelimiter = '\0') consumes the scanset "L[^R]R", where L is the left delimiter character and R is the right delimiter character, which skips whitespace, consumes and ignores the left delimiter, reads characters until the right-delimiter into the string variable (null terminated), and consumes and ignores the right delimiter. If the delimit character is omitted, it defaults to '\'' (single quote).

```
char cs[10];
sin | quoted( wdi( sizeof(cs), cs ) );                 // " is the start/end delimiter
sin | quoted( wdi( sizeof(cs), cs ), '\'' );       // ' is the start/end delimiter
sin | quoted( wdi( sizeof(cs), cs ), '[', ']' ); // [ is the start and ] is the end delimiter
␣␣␣"abc"␣␣'abc'[abc]
```

8. incl(scanset, *wdi manipulator*) consumes the scanset, which reads all the scanned characters into the string variable (null terminated).

```
char cs[10];
sin | incl( "abc", cs );
bcaxyz
```

9. excl(scanset, *wdi manipulator*) consumes the *not* scanset, which reads all the scanned characters into the string variable (null terminated).

```
char cs[10];
sin | excl( "abc", cs );
xyzbca
```

10. ignore(T & v or **const char** cs[] or *string manipulator*) consumes the appropriate characters for the type and ignores them, so the input variable is unchanged.

```
double d;
char cs[10];
sin | ignore( d );                         // d is unchanged
sin | ignore( cs );                        // cs is unchanged, no wdi required
sin | ignore( quoted( wdi( sizeof(cs), cs ) ) ); // cs is unchanged
␣␣−75.35e−4␣25␣"abc"
```

22.7 Concurrent Stream Access

When a stream is shared by multiple threads, input or output characters can be intermixed or cause failure. For example, if two threads execute the following:

```
thread₁ : sout | "abc  " | "def  ";
thread₂ : sout | "uvw  " | "xyz  ";
```

possible outputs are:

abc def uvw xyz	abc uvw xyz def	uvw abc xyz def	abuvwc dexf yz	uvw abc def xyz

Concurrent operations can even corrupt the internal state of the stream resulting in failure. As a result, some form of mutual exclusion is required for concurrent stream access.

A coarse-grained solution is to perform all stream operations via a single thread or within a monitor providing the necessary mutual exclusion for the stream. A fine-grained solution is to have a lock for each stream, which is acquired and released around stream operations by each thread. C∀ provides a fine-grained solution where a recursive lock is acquired and released indirectly via a manipulator acquire or instantiating an RAII type specific for the kind of stream: osacquire for output streams and isacquire for input streams.

The common usage is the short form of the mutex statement to lock a stream during a single cascaded I/O expression, *e.g.*:

```
thread₁ : mutex( sout ) sout | "abc " | "def ";
thread₂ : mutex( sout ) sout | "uvw " | "xyz ";
```

Now, the order of the thread execution is still non-deterministic, but the output is constrained to two possible lines in either order.

```
abc def | uvw xyz
uvw xyz | abc def
```

In summary, the stream lock is acquired by the acquire manipulator and implicitly released at the end of the cascaded I/O expression ensuring all operations in the expression occur atomically.

To lock a stream across multiple I/O operations, he long form of the mutex statement is used, *e.g.*:

```
mutex( sout ) {
    sout | 1;
    mutex( sout ) sout | 2 | 3;          // unnecessary, but ok because of recursive lock
    sout | 4;
} // implicitly release sout lock
```

Note, the unnecessary **mutex** in the middle of the mutex statement, works because the recursive stream-lock can be acquired/released multiple times by the owner thread. Hence, calls to functions that also acquire a stream lock for their output do not result in deadlock.

The previous values written by threads 1 and 2 can be read in concurrently:

```
mutex( sin ) {
    int x, y, z, w;
    sin | x;
    mutex( sin ) sin | y | z;            // unnecessary, but ok because of recursive lock
    sin | w;
} // implicitly release sin lock
```

Again, the order of the reading threads is non-deterministic. Note, non-deterministic reading is rare.

WARNING: The general problem of nested locking can occur if routines are called in an I/O sequence that block, *e.g.*:

```
mutex( sout ) sout | "data:" | rtn( mon );  // mutex call on monitor
```

If the thread executing the I/O expression blocks in the monitor with the sout lock, other threads writing to sout also block until the thread holding the lock is unblocked and releases it. This scenario can lead to deadlock, if the thread that is going to unblock the thread waiting in the monitor first writes to sout (deadly embrace). To prevent nested locking, a simple precaution is to factor out the blocking call from the expression, *e.g.*:

```
int data = rtn( mon );
mutex( sout ) sout | "data:" | data;
```

22.8 Locale

Cultures use different syntax, called a *locale*, for printing numbers so they are easier to read, *e.g.*:

```
12,345.123          // comma separator, period decimal-point
12.345,123          // period separator, comma decimal-point
12␣345,123.         // space separator, comma decimal-point, period terminator
```

A locale is selected with function setlocale, and the corresponding locale package *must* be installed on the underlying system; setlocale returns 0p if the requested locale is unavailable. Furthermore, a locale covers the syntax for many cultural items, *e.g.*, address, measurement, money, etc. This discussion applies to item LC_NUMERIC for formatting non-monetary integral and floating-point values. Figure 8 shows selecting different cultural syntax, which may be associated with one or more countries.

23 String Stream

The stream types ostrstream and istrstream provide all the stream formatting capabilities to/from a C string rather than a stream file. Figure 9, p. 61 shows writing (output) to and reading (input) from a C string. The only string stream operations different from a file stream are:

```
#include <fstream.hfa>
#include <locale.h>                         // setlocale
#include <stdlib.h>                         // getenv

int main() {
    void print() {
        sout | 12 | 123 | 1234 | 12345 | 123456 | 1234567;
        sout | 12. | 123.1 | 1234.12 | 12345.123 | 123456.1234 | 1234567.12345;
        sout | nl;
    }
    sout | "Default locale off";
    print();
    sout | "Locale on" | setlocale( LC_NUMERIC, getenv( "LANG" ) );  // enable local locale
    print();
    sout | "German" | setlocale( LC_NUMERIC, "de_DE.UTF-8" );  // enable German locale
    print();
    sout | "Ukraine" | setlocale( LC_NUMERIC, "uk_UA.utf8" );  // enable Ukraine locale
    print();
    sout | "Default locale off" | setlocale( LC_NUMERIC, "C" );  // disable locale
    print();
}
```

```
Default locale off
12 123 1234 12345 123456 1234567
12. 123.1 1234.12 12345.123 123456.1234 1234567.12345

Locale on en_US.UTF-8
12 123 1,234 12,345 123,456 1,234,567
12. 123.1 1,234.12 12,345.123 123,456.1234 1,234,567.12345

German de_DE.UTF-8
12 123 1.234 12.345 123.456 1.234.567
12. 123,1. 1.234,12 12.345,123 123.456,1234 1.234.567,12345

Ukraine uk_UA.utf8
12 123 1 234 12 345 123 456 1 234 567
12. 123,1. 1␣234,12. 12␣ 345,123. 123␣ 456,1234. 1␣ 234␣567,12345.

Default locale off C
12 123 1234 12345 123456 1234567
12. 123.1 1234.12 12345.123 123456.1234 1234567.12345
```

Figure 8: Stream Locale

- constructors to create a stream that writes to a write buffer (ostrstream) of size, or reads from a read buffer (istrstream) containing a C string terminated with `'\0'`.

  ```
  void ?{}( ostrstream &, char buf[], size_t size );
  void ?{}( istrstream & is, char buf[] );
  ```

- write (ostrstream only) writes all the buffered characters to the specified stream (stdout default).

  ```
  ostrstream & write( ostrstream & os, FILE * stream = stdout );
  ```

  There is no read for istrstream.

24 Structures

Structures in C∀ are basically the same as structures in C. A structure is defined with the same syntax as in C. When referring to a structure in C∀, users may omit the struct keyword.

```
#include <fstream.hfa>
#include <strstream.hfa>

int main() {
    enum { size = 256 };
    char buf[size];                     // output buffer
    ostrstream osstr = { buf, size };   // bind output buffer/size
    int i = 3, j = 5, k = 7;
    double x = 12345678.9, y = 98765.4321e-11;

    osstr | i | hex(j) | wd(10, k) | sci(x) | unit(eng(y)) | "abc";
    write( osstr );                     // write string to stdout
    printf( "%s", buf );                // same lines of output
    sout | i | hex(j) | wd(10, k) | sci(x) | unit(eng(y)) | "abc";

    char buf2[] = "12 14 15 3.5 7e4 abc"; // input buffer
    istrstream isstr = { buf2 };
    char s[10];
    isstr | i | j | k | x | y | s;
    sout  | i | j | k | x | y | s;
}
```

```
3␣0x5␣␣␣␣␣␣␣␣␣␣7␣1.234568e+07␣987.654n␣abc
3␣0x5␣␣␣␣␣␣␣␣␣␣7␣1.234568e+07␣987.654n␣abc
3␣0x5␣␣␣␣␣␣␣␣␣␣7␣1.234568e+07␣987.654n␣abc
12␣14␣15␣3.5␣70000.␣abc
```

Figure 9: String Stream Processing

```
struct Point {
    double x;
    double y;
};

Point p = {0.0, 0.0};
```

C∀ does not support inheritance among types, but instead uses composition to enable reuse of structure fields. Composition is achieved by embedding one type into another. When type A is embedded in type B, an object with type B may be used as an object of type A, and the fields of type A are directly accessible. Embedding types is achieved using anonymous members. For example, using Point from above:

```
void foo(Point p);

struct ColoredPoint {
    Point; // anonymous member (no identifier)
    int Color;
};
...
    ColoredPoint cp = ...;
    cp.x = 10.3; // x from Point is accessed directly
    cp.color = 0x33aaff; // color is accessed normally
    foo(cp); // cp can be used directly as a Point
```

25 Constructors and Destructors

C∀ supports C initialization of structures, but it also adds constructors for more advanced initialization. Additionally, C∀ adds destructors that are called when a variable is deallocated (variable goes out of scope or object is deleted). These functions take a reference to the structure as a parameter (see Section 12, p. 19 for more information).

```
struct Widget {
    int id;
    float size;
    int * optionalint;
};

// ?{} is the constructor operator identifier
// The first argument is a reference to the type to initialize
// Subsequent arguments can be specified for initialization

void ?{}( Widget & w ) {                    // default constructor
    w.id = -1;
    w.size = 0.0;
    w.optionalint = 0p;
}

// constructor with values (does not need to include all fields)
void ?{}( Widget & w, int id, float size ) {
    w.id = id;
    w.size = size;
    w.optionalint = 0p;
}

// ^?{} is the destructor operator identifier
void ^?{}( Widget & w ) {                   // destructor
    w.id = 0;
    w.size = 0.0;
    if ( w.optionalint != 0p ) {
        free( w.optionalint );
        w.optionalint = 0p;
    }
}

Widget baz;                                 // reserve space only
Widget foo{};                               // calls default constructor
Widget bar{ 23, 2.45 };                     // calls constructor with values
baz{ 24, 0.91 };                            // calls constructor with values
?{}( baz, 24, 0.91 );                       // explicit call to constructor
^?{} (bar );                                // explicit call to destructor
```

Figure 10: Constructors and Destructors

26 Overloading

Overloading refers to the capability of a programmer to define and use multiple objects in a program with the same name. In C∀, a declaration may overload declarations from outer scopes with the same name, instead of hiding them as is the case in C. This may cause identical C and C∀ programs to behave differently. The compiler selects the appropriate object (overload resolution) based on context information at the place where it is used. Overloading allows programmers to give functions with different signatures but similar semantics the same name, simplifying the interface to users. Disadvantages of overloading are that it can be used to give functions with different semantics the same name, causing confusion, or that the compiler may resolve to a different function from what the programmer expected. C∀ allows overloading of functions, operators, variables, and even the constants 0 and 1.

The compiler follows some overload resolution rules to determine the best interpretation of all of these overloads. The best valid interpretations are the valid interpretations that use the fewest unsafe conversions. Of these, the best are those where the functions and objects involved are the least polymorphic. Of these, the best have the lowest total conversion cost, including all implicit conversions in the argument expressions. Of these, the best have the highest total conversion cost for the implicit conversions (if any) applied to the argument expressions. If there is no single best valid interpretation, or if the best valid interpretation is ambiguous, then the resulting interpretation is ambiguous. For details about type inference and overload resolution, please see the C∀ Language Specification.

```
int foo(int a, int b) {
    float sum = 0.0;
    float special = 1.0;
    {
        int sum = 0;
        // both the float and int versions of sum are available
        float special = 4.0;
        // this inner special hides the outer version
        ...
    }
    ...
}
```

26.1 Constant

The constants 0 and 1 have special meaning. In C∀, as in C, all scalar types can be incremented and decremented, which is defined in terms of adding or subtracting 1. The operations &&, ||, and ! can be applied to any scalar arguments and are defined in terms of comparison against 0 (ex. (a && b) becomes (a != 0 && b != 0)).

In C, the integer constants 0 and 1 suffice because the integer promotion rules can convert them to any arithmetic type, and the rules for pointer expressions treat constant expressions evaluating to 0 as a special case. However, user-defined arithmetic types often need the equivalent of a 1 or 0 for their functions or operators, polymorphic functions often need 0 and 1 constants of a type matching their polymorphic parameters, and user-defined pointer-like types may need a null value. Defining special constants for a user-defined type is more efficient than defining a conversion to the type from bool.

Why just 0 and 1? Why not other integers? No other integers have special status in C. A facility that let programmers declare specific constants..const Rational 12., for instance. would not be much of an improvement. Some facility for defining the creation of values of programmer-defined types from arbitrary integer tokens would be needed. The complexity of such a feature does not seem worth the gain.

For example, to define the constants for a complex type, the programmer would define the following:

```
struct Complex {
    double real;
    double imaginary;
}

const Complex 0 = {0, 0};
const Complex 1 = {1, 0};
...

    Complex a = 0;
...

    a++;
...
    if (a) { // same as if (a == 0)
...
}
```

26.2 Variable

The overload rules of C∀ allow a programmer to define multiple variables with the same name, but different types. Allowing overloading of variable names enables programmers to use the same name across multiple types, simplifying naming conventions and is compatible with the other overloading that is allowed. For example, a developer may want to do the following:

```
int pi = 3;
float pi = 3.14;
```

```
char pi = .p.;
```

26.3 Function Overloading

Overloaded functions in C∀ are resolved based on the number and type of arguments, type of return value, and the level of specialization required (specialized functions are preferred over generic).

The examples below give some basic intuition about how the resolution works.

```
// Choose the one with less conversions
int doSomething(int value) {...} // option 1
int doSomething(short value) {...} // option 2

int a, b = 4;
short c = 2;

a = doSomething(b); // chooses option 1
a = doSomething(c); // chooses option 2

// Choose the specialized version over the generic

generic(type T)
T bar(T rhs, T lhs) {...} // option 3
float bar(float rhs, float lhs){...} // option 4
float a, b, c;
double d, e, f;
c = bar(a, b); // chooses option 4

// specialization is preferred over unsafe conversions

f = bar(d, e); // chooses option 5
```

26.4 Operator

C∀ also allows operators to be overloaded, to simplify the use of user-defined types. Overloading the operators allows the users to use the same syntax for their custom types that they use for built-in types, increasing readability and improving productivity. C∀ uses the following special identifiers to name overloaded operators:

?[?]	subscripting	?+?	addition	?=?	simple assignment
?()	function call	?−?	subtraction	?\=?	exponentiation assignment
?++	postfix increment	?<<?	left shift	?*=?	multiplication assignment
?−−	postfix decrement	?>>?	right shift	?/=?	division assignment
++?	prefix increment	?<?	less than	?%=?	remainder assignment
−−?	prefix decrement	?<=?	less than or equal	?+=?	addition assignment
*?	dereference	?>=?	greater than or equal	?−=?	subtraction assignment
+?	unary plus	?>?	greater than	?<<=?	left-shift assignment
−?	arithmetic negation	?==?	equality	?>>=?	right-shift assignment
~?	bitwise negation	?!=?	inequality	?&=?	bitwise AND assignment
!?	logical complement	?&?	bitwise AND	?^=?	exclusive OR assignment
?\?	exponentiation	?^?	exclusive OR	?|=?	inclusive OR assignment
?*?	multiplication	?|?	inclusive OR		
?/?	division				
?%?	remainder				

Table 2: Operator Identifiers

These identifiers are defined such that the question marks in the name identify the location of the operands. These operands represent the parameters to the functions, and define how the operands are mapped to the function call. For example, a + b becomes ?+?(a, b).

In the example below, a new type, myComplex, is defined with an overloaded constructor, + operator, and string operator. These operators are called using the normal C syntax.

```
type Complex = struct { // define a Complex type
    double real;
    double imag;
}

// Constructor with default values

void ?{}(Complex &c, double real = 0.0, double imag = 0.0) {
    c.real = real;
    c.imag = imag;
}

Complex ?+?(Complex lhs, Complex rhs) {
    Complex sum;
    sum.real = lhs.real + rhs.real;
    sum.imag = lhs.imag + rhs.imag;
    return sum;
}

String ()?(const Complex c) {
    // use the string conversions for the structure members
    return (String)c.real + . + . + (String)c.imag + .i.;
}
...

Complex a, b, c = {1.0}; // constructor for c w/ default imag
...
c = a + b;
print(.sum = . + c);
```

27 Auto Type-Inferencing

Auto type-inferencing occurs in a declaration where a variable's type is inferred from its initialization expression type.

```
         C++                  gcc
                      #define expr 3.0 * i
auto j = 3.0 * 4;     typeof(expr) j = expr;   // use type of initialization expression
int i;                int i;
auto k = i;           typeof(i) k = i;         // use type of primary variable
```

The two important capabilities are:

- not determining or writing long generic types,
- ensuring secondary variables, related to a primary variable, always have the same type.

In C∀, **typedef** provides a mechanism to alias long type names with short ones, both globally and locally, but not eliminate the use of the short name. gcc provides **typeof** to declare a secondary variable from a primary variable. C∀ also relies heavily on the specification of the left-hand side of assignment for type inferencing, so in many cases it is crucial to specify the type of the left-hand side to select the correct type of the right-hand expression. Only for overloaded routines *with the same return type* is variable type-inferencing possible. Finally, **auto** presents the programming problem of tracking down a type when the type is actually needed. For example, given

```
auto j = ...
```

and the need to write a routine to compute using j

```
void rtn( ... parm );
rtn( j );
```

A programmer must work backwards to determine the type of j's initialization expression, reconstructing the possibly long generic type-name. In this situation, having the type name or a short alias is very useful.

```
#include <fstream.hfa>
#include <coroutine.hfa>

coroutine Fibonacci {
    int fn;                              // used for communication
};

void main( Fibonacci & fib ) with( fib ) { // called on first resume
    int fn1, fn2;                        // retained between resumes
    fn = 0;  fn1 = fn;                   // 1st case
    suspend;                             // restart last resume
    fn = 1;  fn2 = fn1;  fn1 = fn;       // 2nd case
    suspend;                             // restart last resume
    for () {
        fn = fn1 + fn2;  fn2 = fn1;  fn1 = fn; // general case
        suspend;                         // restart last resume
    }
}
int next( Fibonacci & fib ) with( fib ) {
    resume( fib );                       // restart last suspend
    return fn;
}
int main() {
    Fibonacci f1, f2;
    for ( 10 ) {                         // print N Fibonacci values
        sout | next( f1 ) | next( f2 );
    }
}
```

Figure 11: Fibonacci Coroutine

There is also the conundrum in type inferencing of when to *brand* a type. That is, when is the type of the variable more important than the type of its initialization expression. For example, if a change is made in an initialization expression, it can cause cascading type changes and/or errors. At some point, a variable type needs to remain constant and the expression to be in error when it changes.

Given **typedef** and **typeof** in C∀, and the strong need to use the type of left-hand side in inferencing, auto type-inferencing is not supported at this time. Should a significant need arise, this feature can be revisited.

28 Concurrency

Concurrency support in C∀ is implemented on top of a highly efficient runtime system of light-weight, M:N, user level threads. The model integrates concurrency features into the language by making the structure type the core unit of concurrency. All communication occurs through method calls, where data is sent via method arguments, and received via the return value. This enables a very familiar interface to all programmers, even those with no parallel programming experience. It also allows the compiler to do static type checking of all communication, a very important safety feature. This controlled communication with type safety has some similarities with channels in Go, and can actually implement channels exactly, as well as create additional communication patterns that channels cannot. Mutex objects, monitors, are used to contain mutual exclusion within an object and synchronization across concurrent threads.

28.1 Coroutine

Coroutines are the precursor to threads. Figure 11 shows a coroutine that computes the Fibonacci numbers.

28.2 Monitors

A monitor is a structure in C∀ which includes implicit locking of its fields. Users of a monitor interact with it just like any structure, but the compiler handles code as needed to ensure mutual exclusion. An example of the definition of a

```
#include <fstream.hfa>
#include <thread.hfa>

monitor AtomicCnt { int counter; };
void ?{}( AtomicCnt & c, int init = 0 ) with(c) { counter = init; }
int inc( AtomicCnt & mutex c, int inc = 1 ) with(c) { return counter += inc; }
int dec( AtomicCnt & mutex c, int dec = 1 ) with(c) { return counter -= dec; }
forall( ostype & | ostream( ostype ) ) {  // print any stream
    ostype & ?|?( ostype & os, AtomicCnt c ) { return os | c.counter; }
    void ?|?( ostype & os, AtomicCnt c ) { (ostype &)(os | c.counter); ends( os ); }
}

AtomicCnt global;                    // shared

thread MyThread {};
void main( MyThread & ) {
    for ( i; 100_000 ) {
        inc( global );
        dec( global );
    }
}
int main() {
    enum { Threads = 4 };
    processor p[Threads - 1];        // + starting processor
    {
        MyThread t[Threads];
    }
    sout | global;                   // print 0
}
```

Figure 12: Atomic-Counter Monitor

monitor is shown here:

```
type Account = monitor {
    const unsigned long number; // account number
    float balance; // account balance
};
```

28.3 Threads

C∀ also provides a simple mechanism for creating and utilizing user level threads. A thread provides mutual exclusion like a monitor, and also has its own execution state and a thread of control. Similar to a monitor, a thread is defined like a structure:

29 Language Comparisons

C∀ is one of many languages that attempts to improve upon C. In developing C∀, many other languages were consulted for ideas, constructs, and syntax. Therefore, it is important to show how these languages each compare with Do. In this section, C∀ is compared with what the writers of this document consider to be the closest competitors of Do: C++, Go, Rust, and D.

29.1 C++

C++ is a general-purpose programming language. It is an imperative, object-oriented and generic programming language, while also providing facilities for low-level memory manipulation. The primary focus of C++ was adding object-oriented programming to C, and this is the primary difference between C++ and C∀. C++ uses classes to encapsulate data and the functions that operate on that data, and to hide the internal representation of the data. C∀ uses

```
#include <fstream>
#include <kernel>
#include <stdlib>
#include <thread>

thread First  { signal_once * lock; };
thread Second { signal_once * lock; };

void ?{}( First * this, signal_once* lock ) { this->lock = lock; }
void ?{}( Second * this, signal_once* lock ) { this->lock = lock; }

void main( First * this ) {
    for ( int i = 0; i < 10; i += 1 ) {
        sout | "First : Suspend No." | i + 1;
        yield();
    }
    signal( this->lock );
}

void main( Second * this ) {
    wait( this->lock );
    for ( int i = 0; i < 10; i += 1 ) {
        sout | "Second : Suspend No." | i + 1;
        yield();
    }
}

int main( void ) {
    signal_once lock;
    sout | "User main begin";
    {
        processor p;
        {
            First  f = { &lock };
            Second s = { &lock };
        }
    }
    sout | "User main end";
}
```

Figure 13: Simple Threads

modules instead to perform these same tasks. Classes in C++ also enable inheritance among types. Instead of inheritance, C∀ embraces composition and interfaces to achieve the same goals with more flexibility. There are many studies and articles comparing inheritance and composition (or is-a versus has-a relationships), so we will not go into more detail here (Venners, 1998) (Pike, Go at Google: Language Design in the Service of Software Engineering , 2012).

Overloading in C∀ is very similar to overloading in C++, with the exception of the additional use, in C∀, of the return type to differentiate between overloaded functions. References and exceptions in C∀ are heavily based on the same features from C++. The mechanism for interoperating with C code in C∀ is also borrowed from C++.

Both C∀ and C++ provide generics, and the syntax is quite similar. The key difference between the two, is that in C++ templates are expanded at compile time for each type for which the template is instantiated, while in C∀, function pointers are used to make the generic fully compilable. This means that a generic function can be defined in a compiled library, and still be used as expected from source.

29.2 Go

Go, also commonly referred to as golang, is a programming language developed at Google in 2007 [19]. It is a statically typed language with syntax loosely derived from that of C, adding garbage collection, type safety, some structural typing capabilities, additional built-in types such as variable-length arrays and key-value maps, and a large standard library. (Wikipedia)

Go and C∀ differ significantly in syntax and implementation, but the underlying core concepts of the two languages are aligned. Both Go and C∀ use composition and interfaces as opposed to inheritance to enable encapsulation and abstraction. Both languages (along with their tooling ecosystem) provide a simple packaging mechanism for building units of code for easy sharing and reuse. Both languages also include built-in light weight, user level threading concurrency features that attempt to simplify the effort and thought process required for writing parallel programs while maintaining high performance.

Go has a significant runtime which handles the scheduling of its light weight threads, and performs garbage collection, among other tasks. C∀ uses a cooperative scheduling algorithm for its tasks, and uses automatic reference counting to enable advanced memory management without garbage collection. This results in Go requiring significant overhead to interface with C libraries while C∀ has no overhead.

29.3 Rust

Rust is a general-purpose, multi-paradigm, compiled programming language developed by Mozilla Research. It is designed to be a "safe, concurrent, practical language", supporting pure-functional, concurrent-actor[dubious . discuss][citation needed], imperative-procedural, and object-oriented styles.

The primary focus of Rust is in safety, especially in concurrent programs. To enforce a high level of safety, Rust has added ownership as a core feature of the language to guarantee memory safety. This safety comes at the cost of a difficult learning curve, a change in the thought model of the program, and often some runtime overhead.

Aside from those key differences, Rust and C∀ also have several similarities. Both languages support no overhead interoperability with C and have minimal runtimes. Both languages support inheritance and polymorphism through the use of interfaces (traits).

29.4 D

The D programming language is an object-oriented, imperative, multi-paradigm system programming language created by Walter Bright of Digital Mars and released in 2001. [.] Though it originated as a re-engineering of C++, D is a distinct language, having redesigned some core C++ features while also taking inspiration from other languages, notably Java, Python, Ruby, C#, and Eiffel.

D and C∀ both start with C and add productivity features. The obvious difference is that D uses classes and inheritance while C∀ uses composition and interfaces. D is closer to C∀ than C++ since it is limited to single inheritance and also supports interfaces. Like C++, and unlike C∀, D uses garbage collection and has compile-time expanded templates. D does not have any built-in concurrency constructs in the language, though it does have a standard library for concurrency which includes the low-level primitives for concurrency.

A Syntax Ambiguities

C has a number of syntax ambiguities, which are resolved by taking the longest sequence of overlapping characters that constitute a token. For example, the program fragment `x+++++y` is parsed as `x␣++␣++␣+␣y` because operator tokens `++` and `+` overlap. Unfortunately, the longest sequence violates a constraint on increment operators, even though the parse `x␣++␣+␣++␣y` might yield a correct expression. Hence, C programmers are aware that spaces have to added to disambiguate certain syntactic cases.

In C∀, there are ambiguous cases with dereference and operator identifiers, *e.g.*, `int *?*?()`, where the string `*?*?` can be interpreted as:

```
*?␣*?                          // dereference operator, dereference operator
*␣?*?                          // dereference, multiplication operator
```

By default, the first interpretation is selected, which does not yield a meaningful parse. Therefore, C∀ does a lexical look-ahead for the second case, and backtracks to return the leading unary operator and reparses the trailing operator identifier. Otherwise a space is needed between the unary operator and operator identifier to disambiguate this common case.

A similar issue occurs with the dereference, `*?(...)`, and routine-call, `?()(...)` identifiers. The ambiguity occurs when the deference operator has no parameters:

```
*?()␣... ;
*?()␣...(...) ;
```

requiring arbitrary whitespace look-ahead for the routine-call parameter-list to disambiguate. However, the dereference operator *must* have a parameter/argument to dereference `*?(...)`. Hence, always interpreting the string `*?()` as `*␣?()` does not preclude any meaningful program.

The remaining cases are with the increment/decrement operators and conditional expression, *e.g.*:

```
i++?␣...(...);
i?++␣...(...);
```

requiring arbitrary whitespace look-ahead for the operator parameter-list, even though that interpretation is an incorrect expression (juxtaposed identifiers). Therefore, it is necessary to disambiguate these cases with a space:

```
i++␣? i : 0;
i?␣++i : 0;
```

B C Incompatibles

The following incompatibles exist between C∀ and C, and are similar to Annex C for C++ [22].

1. **Change:** add new keywords
   New keywords are added to C∀ (see Section C, p. 73).
   Rationale: keywords added to implement new semantics of C∀.
   Effect on original feature: change to semantics of well-defined feature.
   Any C11 programs using these keywords as identifiers are invalid C∀ programs.
   Difficulty of converting: keyword clashes are accommodated by syntactic transformations using the C∀ back-quote escape-mechanism (see Section 6, p. 5).
   How widely used: clashes among new C∀ keywords and existing identifiers are rare.
2. **Change:** drop K&R C declarations
   K&R declarations allow an implicit base-type of **int**, if no type is specified, plus an alternate syntax for declaring parameters. *e.g.*:

   ```
   x;                          // int x
   *y;                         // int * y
   f( p1, p2 );                // int f( int p1, int p2 );
   g( p1, p2 ) int p1, p2;     // int g( int p1, int p2 );
   ```

   C∀ continues to support K&R routine definitions:

   ```
   f( a, b, c )                // default int return
       int a, b; char c;       // K&R parameter declarations
   {
       ...
   }
   ```

   Rationale: dropped from C11 standard.[17]
   Effect on original feature: original feature is deprecated.
   Any old C programs using these K&R declarations are invalid C∀ programs.
   Difficulty of converting: trivial to convert to C∀.
   How widely used: existing usages are rare.
3. **Change:** type of character literal **int** to **char** to allow more intuitive overloading:

   ```
   int rtn( int i );
   int rtn( char c );
   ```

[17] At least one type specifier shall be given in the declaration specifiers in each declaration, and in the specifier-qualifier list in each structure declaration and type name [21, § 6.7.2(2)]

```
rtn( 'x' );                    // programmer expects 2nd rtn to be called
```

Rationale: it is more intuitive for the call to rtn to match the second version of definition of rtn rather than the first. In particular, output of **char** variable now print a character rather than the decimal ASCII value of the character.

```
sout | 'x' | " " | (int)'x';
x 120
```

Having to cast 'x' to **char** is non-intuitive.

Effect on original feature: change to semantics of well-defined feature that depend on:

```
sizeof( 'x' ) == sizeof( int )
```

no long work the same in C∀ programs.

Difficulty of converting: simple

How widely used: programs that depend upon **sizeof**('x') are rare and can be changed to **sizeof**(**char**).

4. **Change:** make string literals **const**:

```
char * p = "abc";              // valid in C, deprecated in C∀
char * q = expr ? "abc" : "de";    // valid in C, invalid in C∀
```

The type of a string literal is changed from [] **char** to **const** [] **char**. Similarly, the type of a wide string literal is changed from [] wchar_t to **const** [] wchar_t.

Rationale: This change is a safety issue:

```
char * p = "abc";
p[0] = 'w';                    // segment fault or change constant literal
```

The same problem occurs when passing a string literal to a routine that changes its argument.

Effect on original feature: change to semantics of well-defined feature.

Difficulty of converting: simple syntactic transformation, because string literals can be converted to **char** *.

How widely used: programs that have a legitimate reason to treat string literals as pointers to potentially modifiable memory are rare.

5. **Change:** remove *tentative definitions*, which only occurs at file scope:

```
int i;                         // forward definition
int *j = &i;                   // forward reference, valid in C, invalid in C∀
int i = 0;                     // definition
```

is valid in C, and invalid in C∀ because duplicate overloaded object definitions at the same scope level are disallowed. This change makes it impossible to define mutually referential file-local static objects, if initializers are restricted to the syntactic forms of C. For example,

```
struct X { int i; struct X *next; };
static struct X a;             // forward definition
static struct X b = { 0, &a }; // forward reference, valid in C, invalid in C∀
static struct X a = { 1, &b };    // definition
```

Rationale: avoids having different initialization rules for builtin types and user-defined types.

Effect on original feature: change to semantics of well-defined feature.

Difficulty of converting: the initializer for one of a set of mutually-referential file-local static objects must invoke a routine call to achieve the initialization.

How widely used: seldom

6. **Change:** have **struct** introduce a scope for nested types:

```
enum Colour { R, G, B, Y, C, M };
struct Person {
    enum Colour { R, G, B }; // nested type
    struct Face {            // nested type
        Colour Eyes, Hair;   // type defined outside (1 level)
    };
    .Colour shirt;           // type defined outside (top level)
    Colour pants;            // type defined same level
```

```
        Face looks[10];                 // type defined same level
    };
    Colour c = R;                       // type/enum defined same level
    Person.Colour pc = Person.R;  // type/enum defined inside
    Person.Face pretty;                 // type defined inside
```

In C, the name of the nested types belongs to the same scope as the name of the outermost enclosing structure, *i.e.*, the nested types are hoisted to the scope of the outer-most type, which is not useful and confusing. C∀ is C *incompatible* on this issue, and provides semantics similar to C++. Nested types are not hoisted and can be referenced using the field selection operator ".", unlike the C++ scope-resolution operator "::".

Rationale: **struct** scope is crucial to C∀ as an information structuring and hiding mechanism.
Effect on original feature: change to semantics of well-defined feature.
Difficulty of converting: Semantic transformation.
How widely used: C programs rarely have nest types because they are equivalent to the hoisted version.

7. **Change:** In C++, the name of a nested class is local to its enclosing class.
Rationale: C++ classes have member functions which require that classes establish scopes.
Difficulty of converting: Semantic transformation. To make the struct type name visible in the scope of the enclosing struct, the struct tag could be declared in the scope of the enclosing struct, before the enclosing struct is defined. Example:

```
struct Y;                           // struct Y and struct X are at the same scope
struct X {
    struct Y { /* ... */ } y;
};
```

All the definitions of C struct types enclosed in other struct definitions and accessed outside the scope of the enclosing struct could be exported to the scope of the enclosing struct. Note: this is a consequence of the difference in scope rules, which is documented in 3.3.
How widely used: Seldom.

8. **Change:** remove implicit conversion of **void** * to or from any T * pointer:

```
void foo() {
    int * b = malloc( sizeof(int) ); // implicitly convert void * to int *
    char * c = b;                   // implicitly convert int * to void *, and then void * to char *
}
```

Rationale: increase type safety
Effect on original feature: deletion of semantically well-defined feature.
Difficulty of converting: requires adding a cast (see Section E.1 for better alternatives):

```
    int * b = (int *)malloc( sizeof(int) );
    char * c = (char *)b;
```

How widely used: Significant. Some C translators already give a warning if the cast is not used.

9. **Change:** Types must be declared in declarations, not in expressions In C, a sizeof expression or cast expression may create a new type. For example,

```
p = (void*)(struct x {int i;} *)0;
```

declares a new type, struct x .
Rationale: This prohibition helps to clarify the location of declarations in the source code.
Effect on original feature: Deletion of a semantically welldefined feature.
Difficulty of converting: Syntactic transformation.
How widely used: Seldom.

10. **Change:** comma expression is disallowed as subscript
Rationale: safety issue to prevent subscripting error for multidimensional arrays: x[i,j] instead of x[i][j], and this syntactic form then taken by C∀ for new style arrays.
Effect on original feature: change to semantics of well-defined feature.
Difficulty of converting: semantic transformation of x[i,j] to x[(i,j)]
How widely used: Seldom.

C   C∀ Keywords

C∀ introduces the following new keywords, which cannot be used as identifiers.

basetypeof, **choose**, **coroutine**, **disable**, **enable**, **exception**, **fallthrough**, **fallthru**, **finally**, **fixup**, **forall**,**generator**, **int128**, **monitor**, **mutex**, **one_t**, **report**, **suspend**, **throw**, **throwResume**, **trait**, **try**, **virtual**, **waitfor**, **when**, **with**, **zero_t**

C∀ introduces the following new quasi-keywords, which can be used as identifiers.

catch, **catchResume**, **finally**, **fixup**, **or**, **timeout**

D   Standard Headers

C11 prescribes the following standard header-files [21, § 7.1.2]:

assert.h, complex.h, ctype.h, errno.h, fenv.h, float.h, inttypes.h, iso646.h, limits.h, locale.h, math.h, setjmp.h, signal.h, stdalign.h, stdarg.h, stdatomic.h, stdbool.h, stddef.h, stdint.h, stdio.h, stdlib.h, stdnoreturn.h, string.h, tgmath.h, threads.h, time.h, uchar.h, wchar.h, wctype.h

and C∀ adds to this list:

gmp.h, malloc.h, unistd.h

For the prescribed head-files, C∀ uses header interposition to wraps these includes in an **extern** `"C"`; hence, names in these include files are not mangled (see Section 4, p. 2). All other C header files must be explicitly wrapped in **extern** `"C"` to prevent name mangling. This approach is different from C++ where the name-mangling issue is handled internally in C header-files through checks for preprocessor variable __cplusplus, which adds appropriate **extern** `"C"` qualifiers.

E   Standard Library

The C∀ standard-library wraps explicitly-polymorphic C routines into implicitly-polymorphic versions.

E.1   Storage Management

The storage-management routines extend their C equivalents by overloading, alternate names, providing shallow type-safety, and removing the need to specify the allocation size for non-array types.

C storage management provides the following capabilities:

filled after allocation with a specified character or value.

resize an existing allocation to decreased or increased 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 is copied into the new allocation. For an increase in storage size, new storage after the copied data may be filled.

align an allocation on a specified memory boundary, *e.g.*, an address multiple of 64 or 128 for cache-line purposes.

array the allocation size is scaled to the specified number of array elements. An array may be filled, resized, or aligned.

Table 3 shows allocation routines supporting different combinations of storage-management capabilities.

C∀ memory management extends the type safety of all allocations by using the type of the left-hand-side type to determine the allocation size and return a matching type for the new storage. Type-safe allocation is provided for all C allocation routines and new C∀ allocation routines, *e.g.*, in

```
int * ip = (int *)malloc( sizeof(int) );      // C
int * ip = malloc();                          // C∀ type-safe version of C malloc
int * ip = alloc();                           // C∀ type-safe uniform alloc
```

the latter two allocations determine the allocation size from the type of p (**int**) and cast the pointer to the allocated storage to **int** *.

C∀ memory management extends allocation safety by implicitly honouring all alignment requirements, *e.g.*, in

```
struct S { int i; } __attribute__(( aligned( 128 ) )); // cache-line alignment
S * sp = malloc();                            // honour type alignment
```

		fill	resize	alignment	array
C	malloc	no	no	no	no
	calloc	yes (0 only)	no	no	yes
	realloc	copy	yes	no	no
	memalign	no	no	yes	no
	aligned_alloc[a]	no	no	yes	no
	posix_memalign	no	no	yes	no
	valloc	no	no	yes (page size)	no
	pvalloc[b]	no	no	yes (page size)	no
C∀	cmemalign	yes (0 only)	no	yes	yes
	realloc	copy	yes	yes	no
	alloc	no	yes	no	yes
	alloc_set	yes	yes	no	yes
	alloc_align	no	yes	yes	yes
	alloc_align_set	yes	yes	yes	yes

[a]Same as memalign but size is an integral multiple of alignment, which is universally ignored.
[b]Same as valloc but rounds size to multiple of page size.

Table 3: Allocation Routines versus Storage-Management Capabilities

the storage allocation is implicitly aligned to 128 rather than the default 16. The alignment check is performed at compile time so there is no runtime cost.

C∀ memory management extends the resize capability with the notion of *sticky properties*. Hence, initial allocation capabilities are remembered and maintained when resize requires copying. For example, an initial alignment and fill capability are preserved during a resize copy so the copy has the same alignment and extended storage is filled. Without sticky properties it is dangerous to use realloc, resulting in an idiom of manually performing the reallocation to maintain correctness.

C∀ memory management extends allocation to support constructors for initialization of allocated storage, *e.g.*, in

```
struct S { int i; };                    // cache-line alignment
void ?{}( S & s, int i ) { s.i = i; }
// assume ?|? operator for printing an S

S & sp = *new( 3 );                     // call constructor after allocation
sout | sp.i;
delete( &sp );

S * spa = anew( 10, 5 );                // allocate array and initialize each array element
for ( i; 10 ) sout | spa[i] | nonl;
sout | nl;
adelete( 10, spa );
```

Allocation routines new/anew allocate a variable/array and initialize storage using the allocated type's constructor. Note, the matching deallocation routines delete/adelete.

```
extern "C" {
    // C unsafe allocation
    void * malloc( size_t size );
    void * calloc( size_t dim, size_t size );
    void * realloc( void * ptr, size_t size );
    void * memalign( size_t align, size_t size );
    void * aligned_alloc( size_t align, size_t size );
    int posix_memalign( void ** ptr, size_t align, size_t size );
    void * cmemalign( size_t alignment, size_t noOfElems, size_t elemSize ); // CFA
```

```
	// C unsafe initialization/copy
	void * memset( void * dest, int c, size_t size );
	void * memcpy( void * dest, const void * src, size_t size );
}

void * realloc( void * oaddr, size_t nalign, size_t size ); // CFA heap

forall( dtype T | sized(T) ) {
	// C∀ safe equivalents, i.e., implicit size specification
	T * malloc( void );
	T * calloc( size_t dim );
	T * realloc( T * ptr, size_t size );
	T * memalign( size_t align );
	T * cmemalign( size_t align, size_t dim  );
	T * aligned_alloc( size_t align );
	int posix_memalign( T ** ptr, size_t align );

	// C∀ safe general allocation, fill, resize, alignment, array
	T * alloc( void );                                  // variable, T size
	T * alloc( size_t dim );                            // array[dim], T size elements
	T * alloc( T ptr[], size_t dim );                   // realloc array[dim], T size elements

	T * alloc_set( char fill );                         // variable, T size, fill bytes with value
	T * alloc_set( T fill );                            // variable, T size, fill with value
	T * alloc_set( size_t dim, char fill );             // array[dim], T size elements, fill bytes with value
	T * alloc_set( size_t dim, T fill );                // array[dim], T size elements, fill elements with value
	T * alloc_set( size_t dim, const T fill[] );        // array[dim], T size elements, fill elements with array
	T * alloc_set( T ptr[], size_t dim, char fill );    // realloc array[dim], T size elements, fill bytes with value

	T * alloc_align( size_t align );                    // aligned variable, T size
	T * alloc_align( size_t align, size_t dim );        // aligned array[dim], T size elements
	T * alloc_align( T ptr[], size_t align );           // realloc new aligned array
	T * alloc_align( T ptr[], size_t align, size_t dim ); // realloc new aligned array[dim]

	T * alloc_align_set( size_t align, char fill );     // aligned variable, T size, fill bytes with value
	T * alloc_align_set( size_t align, T fill );        // aligned variable, T size, fill with value
	T * alloc_align_set( size_t align, size_t dim, char fill ); // aligned array[dim], T size elements, fill bytes with value
	T * alloc_align_set( size_t align, size_t dim, T fill );    // aligned array[dim], T size elements, fill elements with value
	T * alloc_align_set( size_t align, size_t dim, const T fill[] ); // aligned array[dim], T size elements, fill elements with array
	T * alloc_align_set( T ptr[], size_t align, size_t dim, char fill ); // realloc new aligned array[dim], fill new bytes with value

	// C∀ safe initialization/copy, i.e., implicit size specification
	T * memset( T * dest, char fill );
	T * memcpy( T * dest, const T * src );

	// C∀ safe initialization/copy, i.e., implicit size specification, array types
	T * amemset( T dest[], char fill, size_t dim );
	T * amemcpy( T dest[], const T src[], size_t dim );
}

// C∀ allocation/deallocation and constructor/destructor, non-array types
forall( dtype T | sized(T), ttype Params | { void ?{}( T &, Params ); } ) T * new( Params p );
forall( dtype T | sized(T) | { void ^?{}( T & ); } ) void delete( T * ptr );
forall( dtype T, ttype Params | sized(T) | { void ^?{}( T & ); void delete( Params ); } )
  void delete( T * ptr, Params rest );

// C∀ allocation/deallocation and constructor/destructor, array types
forall( dtype T | sized(T), ttype Params | { void ?{}( T &, Params ); } ) T * anew( size_t dim, Params p );
forall( dtype T | sized(T) | { void ^?{}( T & ); } ) void adelete( size_t dim, T arr[] );
forall( dtype T | sized(T) | { void ^?{}( T & ); }, ttype Params | { void adelete( Params ); } )
```

```
 void adelete( size_t dim, T arr[], Params rest );
```

E.2 String to Value Conversion

```
int ato( const char * ptr );
unsigned int ato( const char * ptr );
long int ato( const char * ptr );
unsigned long int ato( const char * ptr );
long long int ato( const char * ptr );
unsigned long long int ato( const char * ptr );
float ato( const char * ptr );
double ato( const char * ptr );
long double ato( const char * ptr );
float _Complex ato( const char * ptr );
double _Complex ato( const char * ptr );
long double _Complex ato( const char * ptr );

int strto( const char * sptr, char ** eptr, int base );
unsigned int strto( const char * sptr, char ** eptr, int base );
long int strto( const char * sptr, char ** eptr, int base );
unsigned long int strto( const char * sptr, char ** eptr, int base );
long long int strto( const char * sptr, char ** eptr, int base );
unsigned long long int strto( const char * sptr, char ** eptr, int base );
float strto( const char * sptr, char ** eptr );
double strto( const char * sptr, char ** eptr );
long double strto( const char * sptr, char ** eptr );
float _Complex strto( const char * sptr, char ** eptr );
double _Complex strto( const char * sptr, char ** eptr );
long double _Complex strto( const char * sptr, char ** eptr );
```

E.3 Search / Sort

```
forall( T | { int ?<?( T, T ); } )                                  // location
T * bsearch( T key, const T * arr, size_t dim );

forall( T | { int ?<?( T, T ); } )                                  // position
unsigned int bsearch( T key, const T * arr, size_t dim );

forall( T | { int ?<?( T, T ); } )
void qsort( const T * arr, size_t dim );

forall( E | { int ?<?( E, E ); } ) {
    E * bsearch( E key, const E * vals, size_t dim );               // location
    size_t bsearch( E key, const E * vals, size_t dim );            // position
    E * bsearchl( E key, const E * vals, size_t dim );
    size_t bsearchl( E key, const E * vals, size_t dim );
    E * bsearchu( E key, const E * vals, size_t dim );
    size_t bsearchu( E key, const E * vals, size_t dim );
}

forall( K, E | { int ?<?( K, K ); K getKey( const E & ); } ) {
    E * bsearch( K key, const E * vals, size_t dim );
    size_t bsearch( K key, const E * vals, size_t dim );
    E * bsearchl( K key, const E * vals, size_t dim );
    size_t bsearchl( K key, const E * vals, size_t dim );
    E * bsearchu( K key, const E * vals, size_t dim );
    size_t bsearchu( K key, const E * vals, size_t dim );
}

forall( E | { int ?<?( E, E ); } ) {
    void qsort( E * vals, size_t dim );
}
```

E.4 Absolute Value

```
unsigned char abs( signed char );
int abs( int );
unsigned long int abs( long int );
unsigned long long int abs( long long int );
float abs( float );
double abs( double );
long double abs( long double );
float abs( float _Complex );
double abs( double _Complex );
long double abs( long double _Complex );
forall( T | { void ?{}( T *, zero_t ); int ?<?( T, T ); T -?( T ); } )
T abs( T );
```

E.5 C Random Numbers

```
void srandom( unsigned int seed );
char random( void );
char random( char u );                                      // [0,u)
char random( char l, char u );                              // [l,u]
int random( void );
int random( int u );                                        // [0,u)
int random( int l, int u );                                 // [l,u]
unsigned int random( void );
unsigned int random( unsigned int u );                      // [0,u)
unsigned int random( unsigned int l, unsigned int u );      // [l,u]
long int random( void );
long int random( long int u );                              // [0,u)
long int random( long int l, long int u );                  // [l,u]
unsigned long int random( void );
unsigned long int random( unsigned long int u );            // [0,u)
unsigned long int random( unsigned long int l, unsigned long int u ); // [l,u]
float random( void );                                       // [0.0, 1.0)
double random( void );                                      // [0.0, 1.0)
float _Complex random( void );                              // [0.0, 1.0)+[0.0, 1.0)i
double _Complex random( void );                             // [0.0, 1.0)+[0.0, 1.0)i
long double _Complex random( void );                        // [0.0, 1.0)+[0.0, 1.0)i
```

E.6 Algorithms

```
forall( T | { int ?<?( T, T ); } ) T min( T t1, T t2 );
forall( T | { int ?>?( T, T ); } ) T max( T t1, T t2 );
forall( T | { T min( T, T ); T max( T, T ); } ) T clamp( T value, T min_val, T max_val );
forall( T ) void swap( T * t1, T * t2 );
```

F Math Library

The C∀ math-library wraps explicitly-polymorphic C math-routines into implicitly-polymorphic versions.

F.1 General

```
float ?%?( float, float );
float fmod( float, float );
double ?%?( double, double );
double fmod( double, double );
long double ?%?( long double, long double );
long double fmod( long double, long double );
```

```
float remainder( float, float );
double remainder( double, double );
long double remainder( long double, long double );

float remquo( float, float, int * );
double remquo( double, double, int * );
long double remquo( long double, long double, int * );
[ int, float ] remquo( float, float );
[ int, double ] remquo( double, double );
[ int, long double ] remquo( long double, long double );

[ int, float ] div( float, float );
[ int, double ] div( double, double );
[ int, long double ] div( long double, long double );

float fma( float, float, float );
double fma( double, double, double );
long double fma( long double, long double, long double );

float fdim( float, float );
double fdim( double, double );
long double fdim( long double, long double );

float nan( const char * );
double nan( const char * );
long double nan( const char * );
```

F.2 Exponential

```
float exp( float );
double exp( double );
long double exp( long double );
float _Complex exp( float _Complex );
double _Complex exp( double _Complex );
long double _Complex exp( long double _Complex );

float exp2( float );
double exp2( double );
long double exp2( long double );
// float _Complex exp2( float _Complex );
// double _Complex exp2( double _Complex );
// long double _Complex exp2( long double _Complex );

float expm1( float );
double expm1( double );
long double expm1( long double );

float pow( float, float );
double pow( double, double );
long double pow( long double, long double );
float _Complex pow( float _Complex, float _Complex );
double _Complex pow( double _Complex, double _Complex );
long double _Complex pow( long double _Complex, long double _Complex );
```

F.3 Logarithm

```
float log( float );
double log( double );
long double log( long double );
```

```
float _Complex log( float _Complex );
double _Complex log( double _Complex );
long double _Complex log( long double _Complex );

int log2( unsigned int );
long int log2( unsigned long int );
long long int log2( unsigned long long int )
float log2( float );
double log2( double );
long double log2( long double );
// float _Complex log2( float _Complex );
// double _Complex log2( double _Complex );
// long double _Complex log2( long double _Complex );

float log10( float );
double log10( double );
long double log10( long double );
// float _Complex log10( float _Complex );
// double _Complex log10( double _Complex );
// long double _Complex log10( long double _Complex );

float log1p( float );
double log1p( double );
long double log1p( long double );

int ilogb( float );
int ilogb( double );
int ilogb( long double );

float logb( float );
double logb( double );
long double logb( long double );

float sqrt( float );
double sqrt( double );
long double sqrt( long double );
float _Complex sqrt( float _Complex );
double _Complex sqrt( double _Complex );
long double _Complex sqrt( long double _Complex );

float cbrt( float );
double cbrt( double );
long double cbrt( long double );

float hypot( float, float );
double hypot( double, double );
long double hypot( long double, long double );
```

F.4 Trigonometric

```
float sin( float );
double sin( double );
long double sin( long double );
float _Complex sin( float _Complex );
double _Complex sin( double _Complex );
long double _Complex sin( long double _Complex );

float cos( float );
double cos( double );
long double cos( long double );
```

```
float _Complex cos( float _Complex );
double _Complex cos( double _Complex );
long double _Complex cos( long double _Complex );

float tan( float );
double tan( double );
long double tan( long double );
float _Complex tan( float _Complex );
double _Complex tan( double _Complex );
long double _Complex tan( long double _Complex );

float asin( float );
double asin( double );
long double asin( long double );
float _Complex asin( float _Complex );
double _Complex asin( double _Complex );
long double _Complex asin( long double _Complex );

float acos( float );
double acos( double );
long double acos( long double );
float _Complex acos( float _Complex );
double _Complex acos( double _Complex );
long double _Complex acos( long double _Complex );

float atan( float );
double atan( double );
long double atan( long double );
float _Complex atan( float _Complex );
double _Complex atan( double _Complex );
long double _Complex atan( long double _Complex );

float atan2( float, float );
double atan2( double, double );
long double atan2( long double, long double );

float atan( float, float );                        // alternative name for atan2
double atan( double, double );
long double atan( long double, long double );
```

F.5 Hyperbolic

```
float sinh( float );
double sinh( double );
long double sinh( long double );
float _Complex sinh( float _Complex );
double _Complex sinh( double _Complex );
long double _Complex sinh( long double _Complex );

float cosh( float );
double cosh( double );
long double cosh( long double );
float _Complex cosh( float _Complex );
double _Complex cosh( double _Complex );
long double _Complex cosh( long double _Complex );

float tanh( float );
double tanh( double );
long double tanh( long double );
float _Complex tanh( float _Complex );
```

```
double _Complex tanh( double _Complex );
long double _Complex tanh( long double _Complex );

float asinh( float );
double asinh( double );
long double asinh( long double );
float _Complex asinh( float _Complex );
double _Complex asinh( double _Complex );
long double _Complex asinh( long double _Complex );

float acosh( float );
double acosh( double );
long double acosh( long double );
float _Complex acosh( float _Complex );
double _Complex acosh( double _Complex );
long double _Complex acosh( long double _Complex );

float atanh( float );
double atanh( double );
long double atanh( long double );
float _Complex atanh( float _Complex );
double _Complex atanh( double _Complex );
long double _Complex atanh( long double _Complex );
```

F.6 Error / Gamma

```
float erf( float );
double erf( double );
long double erf( long double );
float _Complex erf( float _Complex );
double _Complex erf( double _Complex );
long double _Complex erf( long double _Complex );

float erfc( float );
double erfc( double );
long double erfc( long double );
float _Complex erfc( float _Complex );
double _Complex erfc( double _Complex );
long double _Complex erfc( long double _Complex );

float lgamma( float );
double lgamma( double );
long double lgamma( long double );
float lgamma( float, int * );
double lgamma( double, int * );
long double lgamma( long double, int * );

float tgamma( float );
double tgamma( double );
long double tgamma( long double );
```

F.7 Nearest Integer

```
// n / align * align
signed char floor( signed char n, signed char align );
unsigned char floor( unsigned char n, unsigned char align );
short int floor( short int n, short int align );
unsigned short int floor( unsigned short int n, unsigned short int align );
int floor( int n, int align );
unsigned int floor( unsigned int n, unsigned int align );
```

```
long int floor( long int n, long int align );
unsigned long int floor( unsigned long int n, unsigned long int align );
long long int floor( long long int n, long long int align );
unsigned long long int floor( unsigned long long int n, unsigned long long int align );

// (n + (align − 1)) / align
signed char ceiling_div( signed char n, char align );
unsigned char ceiling_div( unsigned char n, unsigned char align );
short int ceiling_div( short int n, short int align );
unsigned short int ceiling_div( unsigned short int n, unsigned short int align );
int ceiling_div( int n, int align );
unsigned int ceiling_div( unsigned int n, unsigned int align );
long int ceiling_div( long int n, long int align );
unsigned long int ceiling_div( unsigned long int n, unsigned long int align );
long long int ceiling_div( long long int n, long long int align );
unsigned long long int ceiling_div( unsigned long long int n, unsigned long long int align );

// floor( n + (n % align != 0 ? align − 1 : 0), align )
signed char ceiling( signed char n, signed char align );
unsigned char ceiling( unsigned char n, unsigned char align );
short int ceiling( short int n, short int align );
unsigned short int ceiling( unsigned short int n, unsigned short int align );
int ceiling( int n, int align );
unsigned int ceiling( unsigned int n, unsigned int align );
long int ceiling( long int n, long int align );
unsigned long int ceiling( unsigned long int n, unsigned long int align );
long long int ceiling( long long int n, long long int align );
unsigned long long int ceiling( unsigned long long int n, unsigned long long int align );

float floor( float );
double floor( double );
long double floor( long double );

float ceil( float );
double ceil( double );
long double ceil( long double );

float trunc( float );
double trunc( double );
long double trunc( long double );

float rint( float );
long double rint( long double );
long int rint( float );
long int rint( double );
long int rint( long double );
long long int rint( float );
long long int rint( double );
long long int rint( long double );

long int lrint( float );
long int lrint( double );
long int lrint( long double );
long long int llrint( float );
long long int llrint( double );
long long int llrint( long double );

float nearbyint( float );
double nearbyint( double );
long double nearbyint( long double );
```

```

float round( float );
long double round( long double );
long int round( float );
long int round( double );
long int round( long double );
long long int round( float );
long long int round( double );
long long int round( long double );

long int lround( float );
long int lround( double );
long int lround( long double );
long long int llround( float );
long long int llround( double );
long long int llround( long double );
```

F.8 Manipulation

```
float copysign( float, float );
double copysign( double, double );
long double copysign( long double, long double );

float frexp( float, int * );
double frexp( double, int * );
long double frexp( long double, int * );

float ldexp( float, int );
double ldexp( double, int );
long double ldexp( long double, int );

[ float, float ] modf( float );
float modf( float, float * );
[ double, double ] modf( double );
double modf( double, double * );
[ long double, long double ] modf( long double );
long double modf( long double, long double * );

float nextafter( float, float );
double nextafter( double, double );
long double nextafter( long double, long double );

float nexttoward( float, long double );
double nexttoward( double, long double );
long double nexttoward( long double, long double );

float scalbn( float, int );
double scalbn( double, int );
long double scalbn( long double, int );

float scalbln( float, long int );
double scalbln( double, long int );
long double scalbln( long double, long int );
```

G Time Keeping

G.1 Duration

```
struct Duration {
    int64_t tn;                                        // nanoseconds
};

void ?{}( Duration & dur );
void ?{}( Duration & dur, zero_t );
void ?{}( Duration & dur, timeval t )
void ?{}( Duration & dur, timespec t )

Duration ?=?( Duration & dur, zero_t );
Duration ?=?( Duration & dur, timeval t )
Duration ?=?( Duration & dur, timespec t )

Duration +?( Duration rhs );
Duration ?+?( Duration & lhs, Duration rhs );
Duration ?+=?( Duration & lhs, Duration rhs );

Duration -?( Duration rhs );
Duration ?-?( Duration & lhs, Duration rhs );
Duration ?-=?( Duration & lhs, Duration rhs );

Duration ?*?( Duration lhs, int64_t rhs );
Duration ?*?( int64_t lhs, Duration rhs );
Duration ?*=?( Duration & lhs, int64_t rhs );

int64_t ?/?( Duration lhs, Duration rhs );
Duration ?/?( Duration lhs, int64_t rhs );
Duration ?/=?( Duration & lhs, int64_t rhs );
double div( Duration lhs, Duration rhs );

Duration ?%?( Duration lhs, Duration rhs );
Duration ?%=?( Duration & lhs, Duration rhs );

bool ?==?( Duration lhs, zero_t );
bool ?!=?( Duration lhs, zero_t );
bool ?<? ( Duration lhs, zero_t );
bool ?<=?( Duration lhs, zero_t );
bool ?>? ( Duration lhs, zero_t );
bool ?>=?( Duration lhs, zero_t );

bool ?==?( Duration lhs, Duration rhs );
bool ?!=?( Duration lhs, Duration rhs );
bool ?<? ( Duration lhs, Duration rhs );
bool ?<=?( Duration lhs, Duration rhs );
bool ?>? ( Duration lhs, Duration rhs );
bool ?>=?( Duration lhs, Duration rhs );

Duration abs( Duration rhs );

Duration ?`ns( int64_t nsec );
Duration ?`us( int64_t usec );
Duration ?`ms( int64_t msec );
Duration ?`s( int64_t sec );
Duration ?`s( double sec );
Duration ?`m( int64_t min );
Duration ?`m( double min );
Duration ?`h( int64_t hours );
Duration ?`h( double hours );
Duration ?`d( int64_t days );
Duration ?`d( double days );
Duration ?`w( int64_t weeks );
Duration ?`w( double weeks );
```

```
int64_t ?`ns( Duration dur );
int64_t ?`us( Duration dur );
int64_t ?`ms( Duration dur );
int64_t ?`s( Duration dur );
int64_t ?`m( Duration dur );
int64_t ?`h( Duration dur );
int64_t ?`d( Duration dur );
int64_t ?`w( Duration dur );

double ?`dns( Duration dur );
double ?`dus( Duration dur );
double ?`dms( Duration dur );
double ?`ds( Duration dur );
double ?`dm( Duration dur );
double ?`dh( Duration dur );
double ?`dd( Duration dur );
double ?`dw( Duration dur );

Duration max( Duration lhs, Duration rhs );
Duration min( Duration lhs, Duration rhs );

forall( ostype & | ostream( ostype ) ) ostype & ?|?( ostype & os, Duration dur );
```

G.2 timeval

```
void ?{}( timeval & t );
void ?{}( timeval & t, zero_t );
void ?{}( timeval & t, time_t sec, suseconds_t usec );
void ?{}( timeval & t, time_t sec );
void ?{}( timeval & t, Time time );

timeval ?=?( timeval & t, zero_t );
timeval ?+?( timeval & lhs, timeval rhs );
timeval ?-?( timeval & lhs, timeval rhs );
bool ?==?( timeval lhs, timeval rhs );
bool ?!=?( timeval lhs, timeval rhs );
```

G.3 timespec

```
void ?{}( timespec & t );
void ?{}( timespec & t, zero_t );
void ?{}( timespec & t, time_t sec, __syscall_slong_t nsec );
void ?{}( timespec & t, time_t sec );
void ?{}( timespec & t, Time time );

timespec ?=?( timespec & t, zero_t );
timespec ?+?( timespec & lhs, timespec rhs );
timespec ?-?( timespec & lhs, timespec rhs );
bool ?==?( timespec lhs, timespec rhs );
bool ?!=?( timespec lhs, timespec rhs );
```

G.4 itimerval

```
void ?{}( itimerval & itv, Duration alarm );
void ?{}( itimerval & itv, Duration alarm, Duration interval );
```

G.5 Time

```
struct Time {
    uint64_t tn;                                    // nanoseconds since UNIX epoch
};

void ?{}( Time & time );
void ?{}( Time & time, zero_t );
void ?{}( Time & time, timeval t );
void ?{}( Time & time, timespec t );

Time ?=?( Time & time, zero_t );
Time ?=?( Time & time, timeval t );
Time ?=?( Time & time, timespec t );

Time ?+?( Time & lhs, Duration rhs );
Time ?+?( Duration lhs, Time rhs );
Time ?+=?( Time & lhs, Duration rhs );

Duration ?-?( Time lhs, Time rhs );
Time ?-?( Time lhs, Duration rhs );
Time ?-=?( Time & lhs, Duration rhs );
bool ?==?( Time lhs, Time rhs );
bool ?!=?( Time lhs, Time rhs );
bool ?<?( Time lhs, Time rhs );
bool ?<=?( Time lhs, Time rhs );
bool ?>?( Time lhs, Time rhs );
bool ?>=?( Time lhs, Time rhs );

int64_t ?`ns( Time t );

char * yy_mm_dd( Time time, char * buf );
char * ?`ymd( Time time, char * buf ); // short form

char * mm_dd_yy( Time time, char * buf );
char * ?`mdy( Time time, char * buf ); // short form

char * dd_mm_yy( Time time, char * buf );
char * ?`dmy( Time time, char * buf ); // short form

size_t strftime( char * buf, size_t size, const char * fmt, Time time );

forall( ostype & | ostream( ostype ) ) ostype & ?|?( ostype & os, Time time );
```

H Clock

H.1 C time

```
char * ctime( time_t tp );
char * ctime_r( time_t tp, char * buf );
tm * gmtime( time_t tp );
tm * gmtime_r( time_t tp, tm * result );
tm * localtime( time_t tp );
tm * localtime_r( time_t tp, tm * result );
```

H.2 Clock

```
struct Clock {                                         // virtual clock
    Duration offset;                                   // offset from computer real-time
};

void ?{}( Clock & clk );                               // create no offset
void ?{}( Clock & clk, Duration adj );                 // create with offset
void reset( Clock & clk, Duration adj );               // change offset
```

```

Duration resolutionHi();                    // clock resolution in nanoseconds (fine)
Duration resolution();                      // clock resolution without nanoseconds (coarse)

Time timeHiRes();                           // real time with nanoseconds
Time time();                                // real time without nanoseconds
Time time( Clock & clk );                   // real time for given clock
Time ?()( Clock & clk );                    //    alternative syntax
timeval time( Clock & clk );                // convert to C time format
tm time( Clock & clk );
Duration processor();                       // non-monotonic duration of kernel thread
Duration program();                         // non-monotonic duration of program CPU
Duration boot();                            // monotonic duration since computer boot
```

I Pseudo Random Number Generator

Random numbers are values generated independently, i.e., new values do not depend on previous values (independent trials), *e.g.*, lottery numbers, shuffled cards, dice roll, coin flip. While a primary goal of programming is computing values that are *not* random, random values are useful in simulation, cryptography, games, etc. A random-number generator is an algorithm that computes independent values. If the algorithm uses deterministic computation (a predictable sequence of values), it generates *pseudo* random numbers versus *true* random numbers.

All *pseudo random-number generators* (*PRNG*) involve some technique to scramble bits of a value, *e.g.*, multiplicative recurrence:

```
rand = 33967 * (rand + 1063); // scramble bits
```

Multiplication of large values adds new least-significant bits and drops most-significant bits.

bits 63–32 (most)	bits 31–0 (least)
0x0	0x3e8e36
0x5f	0x718c25e1
0xad3e	0x7b5f1dbe
0xbc3b	0xac69ff19
0x1070f	0x2d258dc6

By dropping bits 63–32, bits 31–0 become scrambled after each multiply. The least-significant bits *appear* random but the same bits are always generated given a fixed starting value, called the *seed* (value 0x3e8e36 above). Hence, if a program uses the same seed, the same sequence of pseudo-random values is generated from the PRNG. Often the seed is set to another random value like a program's process identifier (getpid) or time when the program is run; hence, one random value bootstraps another. Finally, a PRNG usually generates a range of large values, *e.g.*, [0, UINT_MAX], which are scaled using the modulus operator, *e.g.*, prng() % 5 produces random values in the range 0–4.

C∀ provides 32/64-bit sequential PRNG type only accessible by a single thread (not thread-safe) and a set of global routines and companion thread PRNG functions accessible by multiple threads without contention. To use the PRNG interface **requires including** stdlib.hfa.

- The PRNG types for sequential programs, including coroutining, are:

```
    struct PRNG32 {};                                     // opaque type, no copy or assignment
    void ?{}( PRNG32 & prng, uint32_t seed );             // fixed seed
    void ?{}( PRNG32 & prng );                            // random seed
    void set_seed( PRNG32 & prng, uint32_t seed );        // set seed
    uint32_t get_seed( PRNG32 & prng );                   // get seed
    uint32_t prng( PRNG32 & prng );                       // [0,UINT_MAX]
    uint32_t prng( PRNG32 & prng, uint32_t u );           // [0,u)
    uint32_t prng( PRNG32 & prng, uint32_t l, uint32_t u ); // [l,u]
    uint32_t calls( PRNG32 & prng );                      // number of calls
    void copy( PRNG32 & dst, PRNG32 & src );              // checkpoint PRNG state

    struct PRNG64 {};                                     // opaque type, no copy or assignment
    void ?{}( PRNG64 & prng, uint64_t seed );             // fixed seed
    void ?{}( PRNG64 & prng );                            // random seed
```

```
PRNG sprng1, sprng2;                                        // select appropriate 32/64-bit PRNG
set_seed( sprng1, 1009 );   set_seed( sprng2, 1009 );
for ( 10 ) {
    // Do not cascade prng calls because side−effect functions called in arbitrary order.
    sout | nlOff | prng( sprng1 );  sout | prng( sprng1, 5 );  sout | prng( sprng1, 0, 5 ) | '\t';
    sout | prng( sprng2 );  sout | prng( sprng2, 5 );  sout | prng( sprng2, 0, 5 ) | nlOn;
}
```

```
37301721 2 2        37301721 2 2
1681308562 1 3      1681308562 1 3
290112364 3 2       290112364 3 2
1852700364 4 3      1852700364 4 3
733221210 1 3       733221210 1 3
1775396023 2 3      1775396023 2 3
123981445 2 3       123981445 2 3
2062557687 2 0      2062557687 2 0
283934808 1 0       283934808 1 0
672325890 1 3       672325890 1 3
```

Figure 14: Sequential PRNG

```
void set_seed( PRNG64 & prng, uint64_t seed );              // set seed
uint64_t get_seed( PRNG64 & prng );                         // get seed
uint64_t prng( PRNG64 & prng );                             // [0,UINT_MAX]
uint64_t prng( PRNG64 & prng, uint64_t u );                 // [0,u)
uint64_t prng( PRNG64 & prng, uint64_t l, uint64_t u );     // [l,u]
uint64_t calls( PRNG64 & prng );                            // number of calls
void copy( PRNG64 & dst, PRNG64 & src );                    // checkpoint PRNG state
```

The type PRNG is aliased to PRNG64 on 64-bit architectures and PRNG32 on 32-bit architectures. A PRNG object is used to randomize behaviour or values during execution, *e.g.*, in games, a character makes a random move or an object takes on a random value. In this scenario, it is useful to have multiple PRNG objects, *e.g.*, one per player or object. However, sequential execution is still repeatable given the same starting seeds for all PRNGs. Figure 14 shows an example that creates two sequential PRNGs, sets both to the same seed (1009), and illustrates the three forms for generating random values, where both PRNGs generate the same sequence of values. Note, to prevent accidental PRNG copying, the copy constructor and assignment are hidden. To copy a PRNG for checkpointing, use the explicit copy member.

- The PRNG global and companion thread functions are for concurrent programming, such as randomizing execution in short-running programs, *e.g.*, yield(prng() % 5).

```
void set_seed( size_t seed );                               // set global seed
size_t get_seed();                                          // get global seed
// SLOWER, global routines
size_t prng( void );                                        // [0,UINT_MAX]
size_t prng( size_t u );                                    // [0,u)
size_t prng( size_t l, size_t u );                          // [l,u]
// FASTER, thread members
size_t prng( thread$ & th );                                // [0,UINT_MAX]
size_t prng( thread$ & th, size_t u );                      // [0,u)
size_t prng( thread$ & th, size_t l, size_t u );            // [l,u]
```

The only difference between the two sets of prng routines is performance.

Because concurrent execution is non-deterministic, seeding the concurrent PRNG is less important, as repeatable execution is impossible. Hence, there is one system-wide PRNG (global seed) but each C∀ thread has its own non-contended PRNG state. If the global seed is set, threads start with this seed, until it is reset and then threads start with the reset seed. Hence, these threads generate the same sequence of random numbers from their specific starting seed. If the global seed is *not* set, threads start with a random seed, until the global seed is set. Hence, these threads generate different sequences of random numbers. If each thread needs its own seed, use a sequen-

```
thread T {};
void main( T & th ) {  // thread address
    for ( i; 10 ) {
        sout | nlOff | prng();  sout | prng( 5 );  sout | prng( 0, 5 ) | '\t';  // SLOWER
        sout | nlOff | prng( th );  sout | prng( th, 5 );  sout | prng( th, 0, 5 ) | nlOn;  // FASTER
    }
}
int main() {
    set_seed( 1009 );
    thread$ & th = *active_thread();  // program−main thread−address
    for ( i; 10 ) {
        sout | nlOff | prng(); sout | prng( 5 ); sout | prng( 0, 5 ) | '\t';  // SLOWER
        sout | nlOff | prng( th ); sout | prng( th, 5 ); sout | prng( th, 0, 5 ) | nlOn;  // FASTER
    }
    sout | nl;
    T t; // run thread
}
```

```
37301721 2 2        1681308562 1 3
290112364 3 2       1852700364 4 3
733221210 1 3       1775396023 2 3
123981445 2 3       2062557687 2 0
283934808 1 0       672325890 1 3
1414344101 1 3      873424536 3 4
871831898 3 4       866783532 0 1
2142057611 4 4      17310256 2 5
802117363 0 4       492964499 0 0
2346353643 1 3      2143013105 3 2
// same output as above from thread t
```

Figure 15: Concurrent PRNG

tial PRNG in each thread. The slower prng global functions, *i.e*., *without* a thread argument, call active_thread internally to indirectly access the current thread's PRNG state, while the faster prng functions, *i.e*., *with* a thread argument, directly access the thread through the thread parameter. If a thread pointer is available, *e.g*., in thread main, eliminating the call to active_thread significantly reduces the cost of accessing the thread's PRNG state. Figure 15 shows an example using the slower/faster concurrent PRNG in the program main and a thread.

J Multi-precision Integers

C∀ has an interface to the GMP multi-precision signed-integers [17], similar to the C++ interface provided by GMP. The C∀ interface wraps GMP routines into operator routines to make programming with multi-precision integers identical to using fixed-sized integers. The C∀ type name for multi-precision signed-integers is Int and the header file is gmp.

```
void ?{}( Int * this );                                         // constructor/destructor
void ?{}( Int * this, Int init );
void ?{}( Int * this, zero_t );
void ?{}( Int * this, one_t );
void ?{}( Int * this, signed long int init );
void ?{}( Int * this, unsigned long int init );
void ?{}( Int * this, const char * val );
void ^?{}( Int * this );

Int ?=?( Int * lhs, Int rhs );                                  // assignment
Int ?=?( Int * lhs, long int rhs );
Int ?=?( Int * lhs, unsigned long int rhs );
Int ?=?( Int * lhs, const char * rhs );

```

```
char ?=?( char * lhs, Int rhs );
short int ?=?( short int * lhs, Int rhs );
int ?=?( int * lhs, Int rhs );
long int ?=?( long int * lhs, Int rhs );
unsigned char ?=?( unsigned char * lhs, Int rhs );
unsigned short int ?=?( unsigned short int * lhs, Int rhs );
unsigned int ?=?( unsigned int * lhs, Int rhs );
unsigned long int ?=?( unsigned long int * lhs, Int rhs );

long int narrow( Int val );
unsigned long int narrow( Int val );

int ?==?( Int oper1, Int oper2 );                           // comparison
int ?==?( Int oper1, long int oper2 );
int ?==?( long int oper2, Int oper1 );
int ?==?( Int oper1, unsigned long int oper2 );
int ?==?( unsigned long int oper2, Int oper1 );

int ?!=?( Int oper1, Int oper2 );
int ?!=?( Int oper1, long int oper2 );
int ?!=?( long int oper1, Int oper2 );
int ?!=?( Int oper1, unsigned long int oper2 );
int ?!=?( unsigned long int oper1, Int oper2 );

int ?<?( Int oper1, Int oper2 );
int ?<?( Int oper1, long int oper2 );
int ?<?( long int oper2, Int oper1 );
int ?<?( Int oper1, unsigned long int oper2 );
int ?<?( unsigned long int oper2, Int oper1 );

int ?<=?( Int oper1, Int oper2 );
int ?<=?( Int oper1, long int oper2 );
int ?<=?( long int oper2, Int oper1 );
int ?<=?( Int oper1, unsigned long int oper2 );
int ?<=?( unsigned long int oper2, Int oper1 );

int ?>?( Int oper1, Int oper2 );
int ?>?( Int oper1, long int oper2 );
int ?>?( long int oper1, Int oper2 );
int ?>?( Int oper1, unsigned long int oper2 );
int ?>?( unsigned long int oper1, Int oper2 );

int ?>=?( Int oper1, Int oper2 );
int ?>=?( Int oper1, long int oper2 );
int ?>=?( long int oper1, Int oper2 );
int ?>=?( Int oper1, unsigned long int oper2 );
int ?>=?( unsigned long int oper1, Int oper2 );

Int +?( Int oper );                                         // arithmetic
Int −?( Int oper );
Int ~?( Int oper );

Int ?&?( Int oper1, Int oper2 );
Int ?&?( Int oper1, long int oper2 );
Int ?&?( long int oper1, Int oper2 );
Int ?&?( Int oper1, unsigned long int oper2 );
Int ?&?( unsigned long int oper1, Int oper2 );
Int ?&=?( Int * lhs, Int rhs );

Int ?|?( Int oper1, Int oper2 );
Int ?|?( Int oper1, long int oper2 );
Int ?|?( long int oper1, Int oper2 );
Int ?|?( Int oper1, unsigned long int oper2 );
Int ?|?( unsigned long int oper1, Int oper2 );
Int ?|=?( Int * lhs, Int rhs );

Int ?^?( Int oper1, Int oper2 );
Int ?^?( Int oper1, long int oper2 );
Int ?^?( long int oper1, Int oper2 );
Int ?^?( Int oper1, unsigned long int oper2 );
Int ?^?( unsigned long int oper1, Int oper2 );
Int ?^=?( Int * lhs, Int rhs );

Int ?+?( Int addend1, Int addend2 );
Int ?+?( Int addend1, long int addend2 );
Int ?+?( long int addend2, Int addend1 );
Int ?+?( Int addend1, unsigned long int addend2 );
Int ?+?( unsigned long int addend2, Int addend1 );
Int ?+=?( Int * lhs, Int rhs );
Int ?+=?( Int * lhs, long int rhs );
Int ?+=?( Int * lhs, unsigned long int rhs );
Int ++?( Int * lhs );
Int ?++( Int * lhs );

Int ?-?( Int minuend, Int subtrahend );
Int ?-?( Int minuend, long int subtrahend );
Int ?-?( long int minuend, Int subtrahend );
Int ?-?( Int minuend, unsigned long int subtrahend );
Int ?-?( unsigned long int minuend, Int subtrahend );
Int ?-=?( Int * lhs, Int rhs );
Int ?-=?( Int * lhs, long int rhs );
Int ?-=?( Int * lhs, unsigned long int rhs );
Int --?( Int * lhs );
Int ?--( Int * lhs );

Int ?*?( Int multiplicator, Int multiplicand );
Int ?*?( Int multiplicator, long int multiplicand );
Int ?*?( long int multiplicand, Int multiplicator );
Int ?*?( Int multiplicator, unsigned long int multiplicand );
Int ?*?( unsigned long int multiplicand, Int multiplicator );
Int ?*=?( Int * lhs, Int rhs );
Int ?*=?( Int * lhs, long int rhs );
Int ?*=?( Int * lhs, unsigned long int rhs );

Int ?/?( Int dividend, Int divisor );
Int ?/?( Int dividend, unsigned long int divisor );
Int ?/?( unsigned long int dividend, Int divisor );
Int ?/?( Int dividend, long int divisor );
Int ?/?( long int dividend, Int divisor );
Int ?/=?( Int * lhs, Int rhs );
Int ?/=?( Int * lhs, long int rhs );
Int ?/=?( Int * lhs, unsigned long int rhs );

[ Int, Int ] div( Int dividend, Int divisor );
[ Int, Int ] div( Int dividend, unsigned long int divisor );

Int ?%?( Int dividend, Int divisor );
Int ?%?( Int dividend, unsigned long int divisor );
Int ?%?( unsigned long int dividend, Int divisor );
Int ?%?( Int dividend, long int divisor );
Int ?%?( long int dividend, Int divisor );
Int ?%=?( Int * lhs, Int rhs );
Int ?%=?( Int * lhs, long int rhs );
Int ?%=?( Int * lhs, unsigned long int rhs );

Int ?<<?( Int shiften, mp_bitcnt_t shift );
Int ?<<=?( Int * lhs, mp_bitcnt_t shift );
Int ?>>?( Int shiften, mp_bitcnt_t shift );
Int ?>>=?( Int * lhs, mp_bitcnt_t shift );

Int abs( Int oper );                                          // number functions
Int fact( unsigned long int N );
Int gcd( Int oper1, Int oper2 );
Int pow( Int base, unsigned long int exponent );
Int pow( unsigned long int base, unsigned long int exponent );
void srandom( gmp_randstate_t state );
Int random( gmp_randstate_t state, mp_bitcnt_t n );
Int random( gmp_randstate_t state, Int n );
Int random( gmp_randstate_t state, mp_size_t max_size );
int sgn( Int oper );
Int sqrt( Int oper );

forall( dtype istype | istream( istype ) ) istype * ?|?( istype * is, Int * mp );  // I/O
forall( dtype ostype | ostream( ostype ) ) ostype * ?|?( ostype * os, Int mp );
```

Figure 16 shows C∀ and C factorial programs using the GMP interfaces. (Compile with flag -lgmp to link with the GMP library.)

K Rational Numbers

Rational numbers are numbers written as a ratio, *i.e.*, as a fraction, where the numerator (top number) and the denominator (bottom number) are whole numbers. When creating and computing with rational numbers, results are constantly reduced to keep the numerator and denominator as small as possible.

```
// implementation
struct Rational {
    long int numerator, denominator;                          // invariant: denominator > 0
}; // Rational

Rational rational();                                          // constructors
Rational rational( long int n );
Rational rational( long int n, long int d );
void ?{}( Rational * r, zero_t );
void ?{}( Rational * r, one_t );

long int numerator( Rational r );                             // numerator/denominator getter/setter
long int numerator( Rational r, long int n );
long int denominator( Rational r );
long int denominator( Rational r, long int d );

int ?==?( Rational l, Rational r );                           // comparison
int ?!=?( Rational l, Rational r );
int ?<?( Rational l, Rational r );
int ?<=?( Rational l, Rational r );
int ?>?( Rational l, Rational r );
int ?>=?( Rational l, Rational r );

Rational -?( Rational r );                                    // arithmetic
Rational ?+?( Rational l, Rational r );
Rational ?-?( Rational l, Rational r );
Rational ?*?( Rational l, Rational r );
```

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 );
    }
}
```

C∀

```
#include <gmp.hfa>
int main( void ) {
    sout | "Factorial Numbers";
    Int fact = 1;

    sout | 0 | fact;
    for ( i; 40 ) {
        fact *= i;
        sout | i | fact;
    }
}
```

```
Factorial Numbers
0 1
1 1
2 2
3 6
4 24
5 120
6 720
7 5040
8 40320
9 362880
10 3628800
11 39916800
12 479001600
13 6227020800
14 87178291200
15 1307674368000
16 20922789888000
17 355687428096000
18 6402373705728000
19 121645100408832000
20 2432902008176640000
21 51090942171709440000
22 1124000727777607680000
23 25852016738884976640000
24 620448401733239439360000
25 15511210043330985984000000
26 403291461126605635584000000
27 10888869450418352160768000000
28 304888344611713860501504000000
29 8841761993739701954543616000000
30 265252859812191058636308480000000
31 8222838654177922817725562880000000
32 263130836933693530167218012160000000
33 8683317618811886495518194401280000000
34 295232799039604140847618609643520000000
35 10333147966386144929666651337523200000000
36 371993326789901217467999448150835200000000
37 13763753091226345046315979581580902400000000
38 523022617466601111760007224100074291200000000
39 20397882081197443358640281739902897356800000000
40 815915283247897734345611269596115894272000000000
```

Figure 16: Multi-precision Factorials

```
Rational ?/?( Rational l, Rational r );

double widen( Rational r );                              // conversion
Rational narrow( double f, long int md );

forall( dtype istype | istream( istype ) ) istype * ?|?( istype *, Rational * ); // I/O
forall( dtype ostype | ostream( ostype ) ) ostype * ?|?( ostype *, Rational );
```

References

[1] Ada16. *Ada Reference Manual ISO/IEC 8652:2012(E) with COR.1:2016*. AXE Consultants, Madison WI, USA, 3rd with technical corrigendum 1 for ada 2012 edition, 2016. https://docs.adacore.com/live/wave/arm12/pdf/arm12/arm-12.pdf. 2

[2] Richard C. Bilson. Implementing overloading and polymorphism in C∀. Master's thesis, School of Computer Science, University of Waterloo, 2003. http://plg.uwaterloo.ca/theses/BilsonThesis.pdf. 2, 45

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

[4] P. A. Buhr. A case for teaching multi-exit loops to beginning programmers. *SIGPLAN Not.*, 20(11):14–22, November 1985. 13

[5] P. A. Buhr, David Till, and C. R. Zarnke. Assignment as the sole means of updating objects. *Softw. Pract. Exper.*, 24(9):835–870, September 1994. 2, 40

[6] Cobol14. *Programming Languages – Cobol ISO/IEC 1989:2014*. International Standard Organization, Geneva, Switzerland, 2nd edition, 2014. https://www.iso.org/standard/51416.html. 2

[7] G. V. Cormack and A. K. Wright. Polymorphism in the compiled language ForceOne. In *Proceedings of the 20th Hawaii International Conference on Systems Sciences*, pages 284–292, January 1987. 2

[8] G. V. Cormack and A. K. Wright. Type-dependent parameter inference. *SIGPLAN Not.*, 25(6):127–136, June 1990. Proceedings of the ACM Sigplan'90 Conference on Programming Language Design and Implementation June 20-22, 1990, White Plains, New York, U.S.A. 2, 35

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

[10] Tom Duff. Duff's device. http://www.lysator.liu.se/c/duffs-device.html, November 1983. 8, 9

[11] Dominic Duggan, Gordon V. Cormack, and John Ophel. Kinded type inference for parametric overloading. *Acta Infomatica*, 33(1):21–68, 1996. 2

[12] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Boston, 1st edition, 1990. 2

[13] Rodolfo Gabriel Esteves. C∀, a study in evolutionary design in programming languages. Master's thesis, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, 2004. http://plg.uwaterloo.ca/theses/EstevesThesis.pdf. 2

[14] Fortran08. *Programming Languages – Fortran Part 1:Base Language ISO/IEC 1539-1:2010*. International Standard Organization, Geneva, Switzerland, 3rd edition, 2010. https://www.iso.org/standard/50459.html. 2

[15] Nissim Francez. Another advantage of keyword notation for parameter communication with subprograms. *Communications of the ACM*, 20(8):604–605, August 1977. 34

[16] John Galletly. *OCCAM 2: Including OCCAM 2.1*. UCL (University College London) Press, London, 2nd edition, 1996. 32, 48

[17] *GNU Multiple Precision Arithmetic Library*. GNU, 2016. https://gmplib.org. 89

[18] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *Java Language Specification*, Java SE 8 edition, 2015. 2

[19] Robert Griesemer, Rob Pike, and Ken Thompson. *Go Programming Language*. Google, Mountain View, CA, USA, 2009. http://golang.org/ref/spec. 2, 69

[20] W. T. Hardgrave. Positional versus keyword parameter communication in programming languages. *SIGPLAN Not.*, 11(5):52–58, May 1976. 34

[21] International Standard Organization, Geneva, Switzerland. *C Programming Language ISO/IEC 9889:2011-12*, 3rd edition, 2012. https://www.iso.org/standard/57853.html. 2, 19, 21, 44, 70, 73

[22] International Standard Organization, Geneva, Switzerland. *C++ Programming Language ISO/IEC 14882:2014*, 4th edition, 2014. https://www.iso.org/standard/64029.html. 2, 70

[23] Kathleen Jensen and Niklaus Wirth. *Pascal User Manual and Report, ISO Pascal Standard*. Springer–Verlag, 4th edition, 1991. Revised by Andrew B. Mickel and James F. Miner. 15

[24] C. H. Lindsey and S. G. van der Meulen. *Informal Introduction to ALGOL 68*. North-Holland, London, 1977. 20

[25] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual*, volume 114 of *Lecture Notes in Computer Science*. Springer, New York, 1981. 32, 47, 48

[26] Python. *Python Language Reference, Release 3.7.2*. Python Software Foundation, https://docs.python.org/3/reference/index.html, 2018. 1

[27] Dennis M. Ritchie. The development of the C language. *SIGPLAN Not.*, 28(3):201–208, March 1993. 18

[28] *Rust Programming Language*, 2015. https://doc.rust-lang.org/reference.html. 2

[29] *Scala Language Specification, Version 2.11*. École Polytechnique Fédérale de Lausanne, 2016. http://www.scala-lang.org/files/archive/spec/2.11. 1

[30] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Series in Computer Science. Addison-Wesley, Boston, 1st edition, 1986. 1

[31] David W. Till. Tuples in imperative programming languages. Master's thesis, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, 1989. 2, 40, 41, 43

[32] TIOBE Index. http://www.tiobe.com/tiobe_index. 1

[33] Ben Werther and Damian Conway. A modest proposal: C++ resyntaxed. *SIGPLAN Not.*, 31(11):74–82, November 1996. 2

Index

Italic page numbers give the location of the main entry for the referenced term. Plain page numbers denote uses of the indexed term. Entries for grammar non-terminals are italicized. A typewriter font is used for grammar terminals and program identifiers.