Enumerated Types in C∀

by

Jiada Liang

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

Waterloo, Ontario, Canada, 2024

Author's Declaration

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

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

Abstract

An *enumeration* is a type defining a (ordered) set of named constant values.

```
enum Week { Mon, Tue, Wed, Thu, Fri, Sat, Sun };
enum Math { PI = 3.14159, Tau = 6.28318, Phi = 1.61803 };
enum RGB { Red = 100b, Green = 010b, Blue = 001b };
```

Its purpose is for readability: replacing constant values in a program with symbolic names that are more meaningful to programmers in the context of the application. Thereafter, associating a name to a different value automatically distributes this rebinding, preventing errors. One of the key properties of an enumeration is the ability to enumerate (iterate) through the constants, and hence, access their values, if present. C restricts an enumeration to the integral type **signed int**, while C++ extends enumerations to all integral types, meaning enumeration names must bind to integer constants. Other modern programming languages provide bindings to any type and additional features to extend enumeration capabilities for better software engineering practices.

The C∀ (C-for-all) programming language is an evolutionary refinement of the C programming language. One of its distinctive features is a parametric-polymorphic generic type. However, legacy data types from C, such as enumerations, do not adapt well to the C∀ generic type-system.

This thesis extends the simple and unsafe enumeration type in the C programming language into a complex and safe enumeration type in the C∀ programming-language, while maintaining backwards compatibility with C. The major contribution is an adaptation of enumerated types with the C∀ type-system in a way that integrates naturally with the generic types. This thesis also presents several smaller refinements to the C∀ overload resolution rules for enumerated types, each of which improves the intuitive nature of enumeration name resolution by the compiler. Finally, this work adds other useful features to enumerations that better support software-engineering practices and simplify program development.

Acknowledgements

To begin, I would like to thank my supervisor Professor Peter Buhr. Thank you for your guidance and support throughout my study and research. I would not be here without you.

Thanks to Gregor Richards and Yizhou Zhang for reading my thesis.

Special thanks to Andrew James Beach for your insight on the theory development of the thesis.

Thanks to Michael Brooks, Fangran Yu, Colby Parsons, Thierry Delisle, Mubeen Zulifiqar, and the entire C∀ team for the development of the C∀ language, making it the best language it can be.

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

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

All basic types in a programming language have a set of constants (symbols), and these constants represent computable values, *e.g.*, integer types have constants −1, 17, 0xff representing whole numbers, floating-point types have constants 5.3, 2.3E−5, 0xff.ffp0 representing real numbers, character types have constants 'a', "abc\n", u8"«naïve»" representing (human readable) text, *etc*. Constants can be overloaded among types, *e.g.*, 0 is a null pointer for all pointer types, and the value zero for integer and floating-point types. (In C∀, the constants 0 and 1 can be overloaded for any type.) Higher-level types compose constants from the basic constants.

```
struct S { int i, j, k; } s;
s = (S){ 1, 2, 3 };              // structure constant
int x[5] = { 1, 2, 3, 4, 5 };    // array constant
```

A constant's symbolic name is dictated by language syntax related to types, *e.g.*, 5. (double), 5.0f (float), 5l (long double). In general, the representation of a constant's value is *opaque*, so the internal representation can be chosen arbitrarily, *e.g.*, two's complement, IEEE floating-point. In theory, there is an infinite set of constant names per type representing an infinite set of values.

It is common in mathematics, engineering, and computer science to alias new constants to existing constants so they have the same value, *e.g.*, π, τ (2π), ϕ (golden ratio), K(k), M, G, T for powers of 2[1] often prefixing bits (b) or bytes (B), *e.g.*, Gb, MB, and in general situations, *e.g.*, specific times (noon, New Years), cities (Big Apple), flowers (Lily), *etc*. An alias can bind to another alias, which transitively binds it to the specified constant. Multiple aliases can represent the same value, *e.g.*, eighth note and quaver, giving synonyms.

Many programming languages capture this important software-engineering capability through a mechanism called *constant* or *literal* naming, where a new constant is aliased to an existing constant. Its purpose is for readability: replacing constant values in a program with symbolic names that are more meaningful to programmers in the context of the application. Thereafter, associating a name to a different value automatically distributes this rebinding, preventing errors. Because an aliased name is a constant, it cannot appear in a mutable context, *e.g.*, π = 42 is meaningless, and a constant has no address, *i.e.*, it is an *rvalue*[2]. In theory, there is an infinite set of possible aliasing; in practice, the number of aliasing per program is finite and small.

[1] Overloaded with SI powers of 10.

[2] The term rvalue defines an expression that can only appear on the right-hand side of an assignment expression.

Aliased constants can form an (ordered) set, *e.g.*, days of a week, months of a year, floors of a building (basement, ground, 1st), colours in a rainbow, *etc*. In this case, the binding between a constant name and value can be implicit, where the values are chosen to support any set operations. Many programming languages capture the aliasing and ordering through a mechanism called an *enumeration*.

> enumerate (verb, transitive). To count, ascertain the number of; more usually, to mention (a number of things or persons) separately, as if for the purpose of counting; to specify as in a list or catalogue. [5]

Within an enumeration set, the enumeration names (aliases) must be unique, and instances of an enumerated type are *often* restricted to hold only these names.

It is possible to enumerate among set names without having an ordering among the set values. For example, the week, the weekdays, the weekend, and every second day of the week.

```
for ( cursor in Mon, Tue, Wed, Thu, Fri, Sat, Sun ) ...     // week
for ( cursor in Mon, Tue, Wed, Thu, Fri ) ...               // weekday
for ( cursor in Sat, Sun ) ...                              // weekend
for ( cursor in Mon, Wed, Fri, Sun ) ...                    // every second day of week
```

A set can have a partial or total ordering, making it possible to compare set elements, *e.g.*, Monday is before Tuesday and Tuesday is after. Ordering allows iterating among the enumeration set using relational operators and advancement, *e.g.*:

```
for ( cursor = Monday; cursor <= Friday; cursor = succ( cursor ) ) ... // weekdays
```

Here the values for the set names are logically *generated* rather than listing a subset of names.

Hence, the fundamental aspects of an enumeration are:

1. It provides a finite set of new constants, which are implicitly or explicitly assigned values that must be appropriate for any set operations, *e.g.*, increasing order. This aspect differentiates an enumeration from general types with an infinite set of constants.
2. The alias names are constants, which follow transitively from their binding to other constants.
3. Defines a type for generating instances (variables).
4. For safety, an enumeration instance should be restricted to hold only its constant names.
5. There is a mechanism for *enumerating* over the enumeration names, where the ordering can be implicit from the type, explicitly listed, or generated arithmetically.

1.1 Terminology

The term *enumeration* defines a type with a set of new constants, and the term *enumerator* represents an arbitrary alias name (see Section 2.1.2, p. 9 for the name derivations). An enumerated type can have the following properties: *label* (name), *order* (position), and *value* (payload).

```
   enumeration                    enumerators
        ↓                              ↓
enum Week { Mon,  Tue,  Wed,  Thu,  Fri,  Sat,  Sun = 42 };
      label  Mon   Tue   Wed   Thu   Fri   Sat    Sun
      order  0     1     2     3     4     5      6
      value  0     1     2     3     4     5      42
```

Here, the enumeration Week defines the enumerator constants Mon, Tue, Wed, Thu, Fri, Sat, and Sun. The implicit ordering implies the successor of Tue is Mon and the predecessor of Tue is Wed, independent of any associated enumerator values. The value is the implicitly/explicitly assigned constant to support any enumeration operations; the value may be hidden (opaque) or visible.

Specifying complex ordering is possible:

```
enum E1 { [1 {A, B}, [2 C ]1, {D, E} ]2 };      // overlapping square brackets
enum E2 { {A, {B, C} }, { {D, E}, F };          // nesting
```

For E1, there is the partial ordering among A, B and C, and C, D and E, but not among A, B and D, E. For E2, there is the total ordering A $<$ {B, C} $<$ {D, E} $<$ F. Only flat total-ordering among enumerators is considered in this work.

1.2 Motivation

Many programming languages provide an enumeration-like mechanism, which may or may not cover the previous five fundamental enumeration aspects. Hence, the term *enumeration* can be confusing and misunderstood. Furthermore, some languages conjoin the enumeration with other type features, making it difficult to tease apart which feature is being used. This section discusses some language features that are sometimes called an enumeration but do not provide all enumeration aspects.

1.2.1 Aliasing

Some languages provide simple aliasing (renaming).

```
const Size = 20, Pi = 3.14159, Name = "Jane";
```

The alias name is logically replaced in the program text by its matching constant. It is possible to compare aliases, if the constants allow it, *e.g.*, Size < Pi, whereas Pi < Name might be disallowed depending on the language.

Aliasing is *not* macro substitution, *e.g.*, **#define** Size 20, where a name is replaced by its value *before* compilation, so the name is invisible to the programming language. With aliasing, each new name is part of the language, and hence, participates fully, such as name overloading in the type system. Aliasing is not an immutable variable.

```
extern const int Size = 20;
extern void foo( const int & size );
foo( Size ); // take the address of (reference) Size
```

Taking the address of an immutable variable makes it an *lvalue*, which implies it has storage. With separate compilation, it is necessary to choose one translation unit to perform the initialization. If aliasing requires storage, its address and initialization are opaque (compiler only), similar to C++ rvalue reference &&.

Aliasing does provide readability and automatic resubstitution. It also provides simple enumeration properties, but with effort.

```
const Mon = 1, Tue = 2, Wed = 3, Thu = 4, Fri = 5, Sat = 6, Sun = 7;
```

Any reordering of the enumerators requires manual renumbering.

```
const Sun = 1, Mon = 2, Tue = 3, Wed = 4, Thu = 5, Fri = 6, Sat = 7;
```

For these reasons, aliasing is sometimes called an enumeration. However, there is no type to create a type-checked instance or iterator cursor, so there is no ability to enumerate. Hence, there are multiple enumeration aspects not provided by aliasing, justifying a separate enumeration type in a programming language.

1.2.2 Algebraic Data Type

An algebraic data type (ADT)[3] is another language feature often linked with enumeration, where an ADT conjoins an arbitrary type, possibly a **class** or **union**, and a named constructor. For example, in Haskell:

```
data S = S { i::Int, d::Double }          // structure
data Foo = A Int | B Double | C S         // ADT, composed of three types
foo = A 3;                                // type Foo is inferred
bar = B 3.5
baz = C S{ i = 7, d = 7.5 }
```

the ADT has three variants (constructors), A, B, C, with associated types Int, Double, and S. The constructors create an initialized value of the specific type that is bound to the immutable variables foo, bar, and baz. Hence, the ADT Foo is like a union containing values of the associated types, and a constructor name is used to initialize and access the value using dynamic pattern-matching.

```
prtfoo val = -- function                  main = do
   -- pattern match on constructor           prtfoo foo
   case val of                               prtfoo bar
    A a -> print a                           prtfoo baz
    B b -> print b                        3
    C (S i d) -> do                       3.5
      print i                             7
      print d                             7.5
```

For safety, most languages require all associated types to be listed or a default case with no field accesses.

A less frequent case is multiple constructors with the same type.

[3] ADT is overloaded with abstract data type.

```
struct S { char s[32]; };
variant< int, double, S > vd;
variant< int, int, int > vs;

// discrimination based on type
vd = 3;
if ( holds_alternative<int>(vd) ) cout << "int " << get<int>(vd ) << endl;
vd = 3.5;
if ( holds_alternative<double>(vd) ) cout << "double " << get<double>(vd) << endl;
vd = (S){ "abc" };
if ( holds_alternative<S>(vd) ) cout << "S.s " << get<S>(vd).s << endl;

// discrimination based on type and position within type
vs = (variant<int,int,int>){ in_place_index<0>, 12 };
if ( vs.index() == 0 ) cout << "posn 0 " << get<0>(vs) << endl;
vs = (variant<int,int,int>){ in_place_index<1>, 4 };
if ( vs.index() == 1 ) cout << "posn 1 " << get<1>(vs) << endl;
vs = (variant<int,int,int>){ in_place_index<2>, 5 };
if ( vs.index() == 2 ) cout << "posn 2 " << get<2>(vs) << endl;
```

Figure 1.1: C++ variant Discrimination Using RTTI/Position

```
data Bar = X Int | Y Int | Z Int;
foo = X 3;
bar = Y 3;
baz = Z 5;
```

Here, the constructor name gives different meanings to the values in the common **Int** type, *e.g.*, the value 3 has different interpretations depending on the constructor name in the pattern matching.

Note, the term *variant* is often associated with ADTs. However, there are multiple languages with a variant type that is not an ADT (see Algol68 [14] or C++ variant). Here, the type (and possibly the position for equivalent types) is used to discriminate the specific *variant* within the variant instance. For example, Figure 1.1 shows the C++ equivalent of the two Haskell ADT types using variant types. In these languages, the variant cannot be used to simulate an enumeration. Hence, in this work the term variant is not a synonym for ADT.

In terms of functional programming linguistics, enumerations often refer to a unit type ADT, which is a set with the nil value carrying no information. The unit type is different from type **void** in C, because **void** has no value, which is an empty set. Hence, **void** is a C annotation that nothing is expected in this place. For example, a function that takes a **void** parameter and returns a **void** is a function that expects no parameters and returns nothing.

```
void foo( void );
foo();                                  // no arguments and no result
```

Because of this distinction, it is impossible to have a variable of type **void**, to assign a **void** value, or have a function taking and returning multiple **void**s.

```
void v;                                 // disallowed
```

```
day = Tue                                          Tue
main = do                                          [Mon,Tue,Wed,Thu,Fri,Sat,Sun]
    if day == Tue then                             [Mon,Tue,Wed,Thu,Fri]
        print day                                  [Sat,Sun]
    else
        putStr "not Tue"
    print (enumFrom Mon)            – week
    print (enumFromTo Mon Fri)      – weekday
    print (enumFromTo Sat Sun)      – weekend
```

Figure 1.2: Haskell Enumeration

```
v = void;
[ void, void ] bar( void, void );
```

Programming languages often use an empty parameter list to imply no value and no return type for empty return.

```
[] bar();                                          // C∀ empty/empty prototype
```

However, C is saddled with an empty parameter list meaning a list of unknown type parameters, *i.e.*, var_arg, which is changed to **void** in C++/C∀. As a result, a function that returns **void** cannot be used as a parameter of a function that expects no parameter.

```
void foo( void );
foo( foo() );                                      // void argument does not match with void parameter
```

This issue arose when simulating an ADT using a C++ variant: **void** cannot be used as an empty variant. To solve this problem, C++ introduced std::monstate [4], a type that can be instantiated as a value but holds no information. A similar approximation in C is to define a **struct** type with no fields.

```
struct Unit {} e;                                  // empty type
Unit bar( Unit );
bar( bar( e ) );
```

Because std::monostate and Unit are user-defined types versus part of the type system, they are only an approximation to unit because other unit types can be defined.

In the Haskell ADT:

```
data Week = Mon | Tue | Wed | Thu | Fri | Sat | Sun deriving(Enum, Eq, Show)
```

the default type for each constructor is the unit type, and deriving from Enum enforces no other associated types. The Eq allows equality comparison, and Show is for printing. The nullary constructors for the unit types are numbered left-to-right from 0 to maxBound−1, and provide enumerating operations succ, pred, enumFrom, enumFromTo. Figure 1.2 shows enumeration comparison and iterating (enumerating).

The key observation is the dichotomy between an ADT and enumeration: the ADT uses the associated type resulting in a union-like data structure, and the enumeration does not use the associated type, and hence, is not a union. In contrast, an enumeration may be constructed using the ADT mechanism, but it is so restricted it is not an ADT. Furthermore, a general ADT

cannot be an enumeration because the constructors generate different values making enumerating meaningless. While functional programming languages regularly repurpose the ADT type into an enumeration type, this process seems contrived and confusing. Hence, there is only a weak equivalence between an enumeration and ADT, justifying a separate enumeration type in a programming language.

1.3 Contributions

The goal of this work is to to extend the simple and unsafe enumeration type in the C programming-language into a complex and safe enumeration type in the C∀ programming-language, while maintaining backwards compatibility with C. On the surface, enumerations seem like a simple type. However, when extended with advanced features, enumerations become complex for both the type system and the runtime implementation.

The contributions of this work are:

1. safety: Define a safe enumeration conversion scheme, both for C and C∀, and replace ad-hoc C idioms with safer software-engineering approaches.
2. overloading: Provide a pattern to overload functions, literals, and variables for polymorphic enumerations using the C∀ type system.
3. scoping: Add a namespace for enumerations and qualified access into the namespace to deal with the naming problem.
4. generalization: Support all language types for enumerators with associated values providing enumeration constants for any type.
5. reuse: Implement subset and containment inheritance for enumerations.
6. control flow: Extend control-flow structures making it safer and easier to enumerate over an enumeration.
7. I/O: Provide input and output of enumerations based on enumerator names.

Chapter 2

Background

This chapter covers background material for C enumerations and C∀ features used in later discussions.

2.1 C

As mentioned in Section 1.2.1, p. 3, it is common for C programmers to believe there are three equivalent forms of named constants.

```
#define Mon 0
static const int Mon = 0;
enum { Mon };
```

1. For **#define**, the programmer must explicitly manage the constant name and value. Furthermore, these C preprocessor macro names are outside the C type system and can unintentionally change program text.
2. The same explicit management is true for the **const** declaration, and the **const** variable cannot appear in constant-expression locations, like **case** labels, array dimensions,[1] immediate operands of assembler instructions and occupies storage.

   ```
   $ nm test.o
   0000000000000018 r Mon
   ```
3. Only the **enum** form is managed by the compiler, is part of the language type-system, works in all C constant-expression locations, and does not occupy storage.

2.1.1 C const

C can simulate the aliasing **const** declarations (see Section 1.2.1, p. 3), with static and dynamic initialization.

[1] C allows variable-length array declarations (VLA), so this case does work. Still, it fails in C++, which does not support VLAs, unless it is g++.

static initialization	dynamic initialization

static const int one = 0 + 1;
static const void * NIL = NULL;
static const double PI = 3.14159;
static const char Plus = '+';
static const char * Fred = "Fred";
static const int Mon = 0, Tue = Mon + 1, Wed = Tue + 1,
Thu = Wed + 1, Fri = Thu + 1, Sat = Fri + 1, Sun = Sat + 1;
	void foo() {
	// auto scope only
	const int r = random() % 100;
	int va[r];
	}

However, statically initialized identifiers cannot appear in constant-expression contexts, *e.g.*, **case**. Dynamically initialized identifiers may appear in initialization and array dimensions, which allows variable-sized arrays on the stack. Again, this form of aliasing is not an enumeration.

2.1.2 C Enumeration

The C enumeration has the following syntax [11, § 6.7.2.2].

enum–specifier:
enum *identifier*$_{opt}$ { *enumerator–list* }
enum *identifier*$_{opt}$ { *enumerator–list* , }
enum *identifier*
enumerator–list:
enumerator
enumerator–list , *enumerator*
enumerator:
enumeration–constant
enumeration–constant = *constant–expression*

The terms *enumeration* and *enumerator* used in this work (see Section 1.1, p. 2) come from the grammar. The C enumeration semantics are discussed using examples.

2.1.2.1 Type Name

An *unnamed* enumeration is used to provide aliasing (see Section 1.2.1, p. 3) exactly like a **const** declaration in other languages. However, it is restricted to integral values.

```
enum { Size = 20, Max = 10, MaxPlus10 = Max + 10, Max10Plus1, Fred = -7 };
```

Here, the aliased constants are 20, 10, 20, 21, and -7. Direct initialization is achieved by a compile-time expression that generates a constant value. Indirect initialization (without an initializer, Max10Plus1) is called *auto-initialization*, where enumerators are assigned values from left to right, starting at zero or the next explicitly initialized constant, incrementing by 1. Because multiple independent enumerators can be combined, enumerators with the same values can occur. The enumerators are rvalues, so the assignment is disallowed. Finally, enumerators are *unscoped*, *i.e.*, enumerators declared inside of an **enum** are visible (projected) outside into the enclosing scope of the **enum** type. This semantic is required for unnamed enumerations because there is no type name for scoped qualification.

As noted, this aliasing declaration is not an enumeration, even though it is declared using an **enum** in C. While the semantics is misleading, this enumeration form matches with aggregate types:

```
typedef struct /* unnamed */ { ... } S;
struct /* unnamed */ { ... } x, y, z;          // questionable
struct S {
    union /* unnamed */ {                      // unscoped fields
        int i; double d ; char ch;
    };
};
```

Hence, C programmers would expect this enumeration form to exist in harmony with the aggregate form.

A *named* enumeration is an enumeration:

```
enum Week { Mon, Tue, Wed, Thu = 10, Fri, Sat, Sun };
```

and adopts the same semantics as direct and auto initialization. For example, Mon to Wed are implicitly assigned with constants 0–2, Thu is explicitly set to constant 10, and Fri to Sun are implicitly assigned with constants 11–13. As well, initialization may occur in any order.

```
enum Week {
    Thu = 10, Fri, Sat, Sun,
    Mon = 0, Tue, Wed,                         // terminating comma
};
```

Note the comma in the enumerator list can be a terminator or a separator, allowing the list to end with a dangling comma.[2] This feature allows enumerator lines to be interchanged without moving a comma. Named enumerators are also unscoped.

2.1.2.2 Implementation

Theoretically, a C enumeration *variable* is an implementation-defined integral type large enough to hold all enumerator values. In practice, C defines **int** [11, § 6.4.4.3] as the underlying type for enumeration variables, restricting initialization to integral constants, which have type **int** (unless qualified with a size suffix). According to the C standard, type **int** is defined as the following:

> A "plain" **int** object has the natural size suggested by the architecture of the execution environment (large enough to contain any value in the range INT_MIN to INT_MAX as defined in the header <limits.h>). [11, § 6.2.5(5)]

Table 2.1 shows integer storage sizes. On UNIX systems (LP64), **int** means 4 bytes on both 32/64-bit architectures, which does not seem like the "natural" size for a 64-bit architecture. Whereas **long int** means 4 bytes on a 32-bit and 8 bytes on 64-bit architectures, and **long long int** means 8 bytes on both 32/64-bit architectures, where 64-bit operations are simulated on 32-bit architectures. On Windows systems (LLP64), **int** and **long** mean 4 bytes on both 32/64-bit

[2] A terminating comma appears in other C syntax, *e.g.*, the initializer list.

Table 2.1: Integer Storage Sizes (bytes)

Type	LP64	LLP64
char	1	1
short int	2	2
int	4	4
long int	8	4
long long int	8	8
pointer	8	8

```
enum E { IMin = INT_MIN, IMax = INT_MAX,
         ILMin = LONG_MIN, ILMax = LONG_MAX,
         ILLMin = LLONG_MIN, ILLMax = LLONG_MAX };
int main() {
    printf( "%zd %zd\n%zd %zd\n%zd %d %d\n%zd %ld %ld\n%zd %ld %ld\n",
            sizeof(enum E), sizeof(typeof(IMin)),
            sizeof(int), sizeof(long int),
            sizeof(IMin), IMin, IMax,
            sizeof(ILMin), ILMin, ILMax,
            sizeof(ILLMin), ILLMin, ILLMax );
}
8 4
4 8
4 -2147483648 2147483647
8 -9223372036854775808 9223372036854775807
8 -9223372036854775808 9223372036854775807
```

Figure 2.1: gcc/clang Enumeration Storage Size

architectures, which also does not seem like the "natural" size for a 64-bit architecture. Figure 2.1 shows both gcc and clang partially ignore this specification and type the integral size of an enumerator based on its initialization. Hence, initialization in the range INT_MIN..INT_MAX results in a 4-byte enumerator, and outside this range, the enumerator is 8 bytes. Note that **sizeof**(**typeof**(IMin)) != **sizeof**(E), making the size of an enumerator different than its containing enumeration type, which seems inconsistent.

2.1.2.3 Usage

C proves an implicit *bidirectional* conversion between an enumeration and its integral type and between different enumerations.

```
enum Week week = Mon;                   // week == 0
week = Fri;                             // week == 11
int i = Sun;                            // implicit conversion to int, i == 13
week = 10000;                           // UNDEFINED! implicit conversion to Week

enum Season { Spring, Summer, Fall, Winter };
week = Winter;                          // UNDEFINED! implicit conversion to Week
```

While converting an enumerator to its underlying type is sound, the implicit conversion from the base or another enumeration type to an enumeration is a common source of error.

Enumerators can appear in **switch** and looping statements.

```
enum Week { Mon, Tue, Wed, Thu, Fri, Sat, Sun };
switch ( week ) {
    case Mon ... Fri:                           // gcc case range
        printf( "weekday\n" );
    case Sat: case Sun:
        printf( "weekend\n" );
}
for ( enum Week day = Mon; day <= Sun; day += 1 ) { // step of 1
    printf( "day %d\n", day ); // 0-6
}
```

For iterating using arithmetic to make sense, the enumerator values *must* have a consecutive ordering with a fixed step between values. For example, a previous gap introduced by Thu = 10 results in iterating over the values 0–13, where values 3–9 are not Week values. Note that the bidirectional conversion allows incrementing day: day is converted to **int**, integer 1 is added, and the result is converted back to Week for the assignment to day. For safety, C++ does not support the bidirectional conversion, and hence, an unsafe cast is necessary to increment day: day = (Week)(day + 1).

There is a C idiom that computes the number of enumerators in an enumeration automatically.

```
enum E { A, B, C, D, N };  // N == 4
for ( enum E e = A; e < N; e += 1 ) ...
```

Serendipitously, the auto-incrementing counts the number of enumerators and puts the total into the last enumerator N. This N is often used as the dimension for an array associated with the enumeration.

```
E array[N];
for ( enum E e = A; e < N; e += 1 ) {
    array[e] = e;
}
```

However, for non-consecutive ordering and non-integral typed enumerations, (see Section 4.1, p. 24), this idiom fails.

This idiom is often used with another C idiom for matching companion information. For example, an enumeration may be linked with a companion array of printable strings.

```
enum Integral_Type { chr, schar, uschar, sshort, ushort, sint, usint, ..., NO_OF_ITYPES };
char * Integral_Name[NO_OF_ITYPES] = {
    "char", "signed char", "unsigned char",
    "signed short int", "unsigned short int",
    "signed int", "unsigned int", ...
};
enum Integral_Type integral_type = ...
printf( "%s\n", Integral_Name[integral_type] ); // human readable type name
```

However, the companion idiom results in the *harmonizing* problem because an update to the enumeration Integral_Type often requires a corresponding update to the companion array Integral_

Name. The requirement to harmonize is, at best, indicated by a comment before the enumeration. This issue is exacerbated if enumeration and companion array are in different translation units.

While C provides a true enumeration, it is restricted, has unsafe semantics, and does not provide helpful/advanced enumeration features in other programming languages.

2.2 C∀

C∀ in *not* an object-oriented programming language, *i.e.*, functions cannot be nested in aggregate types, and hence, there is no *receiver* notation for calling functions, *e.g.*, obj.method(...), where the first argument proceeds the call and becomes an implicit first (**this**) parameter. The following sections provide short descriptions of C∀ features needed further in the thesis. Other C∀ features are presented in situ with short or no explanation because the feature is obvious to C programmers.

2.2.1 Overloading

Overloading allows programmers to use the most meaningful names without fear of name clashes within a program or from external sources like included files.

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

Experience from C++ and C∀ developers is that the type system implicitly and correctly disambiguates the majority of overloaded names, *i.e.*, it is rare to get an incorrect selection or ambiguity, even among hundreds of overloaded (variables and) functions. In many cases, a programmer has no idea there are name clashes, as they are silently resolved, simplifying the development process. Depending on the language, ambiguous cases are resolved using some form of qualification and/or casting.

2.2.2 Operator Overloading

Virtually all programming languages overload the arithmetic operators across the basic types using the number and type of parameters and returns. Like C++, C∀ also allows these operators to be overloaded with user-defined types. The syntax for operator names uses the `'?'` character to denote a parameter, *e.g.*, unary operators: ?++, ++?, binary operator ?+?.

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

The type system examines each call size and selects the best matching overloaded function based on the number and types of arguments. If there are mixed-mode operands, 2 + 3.5, the type system, like in C/C++, attempts (safe) conversions, converting the argument type(s) to the parameter type(s).

2.2.3 Function Overloading

Both C∀ and C++ allow function names to be overloaded as long as their prototypes differ in the number and type of parameters and returns.

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

In this case, the name f is overloaded depending on the number and parameter types. The type system examines each call size and selects the best match based on the number and types of arguments. Here, the call f('A') is a perfect match for the number and parameter type of function (2).

Ada, Scala, and C∀ type-systems also use the return type to pinpoint the best-overloaded name in resolving a call.

```
int f( void );           // (4); overloaded on return type
double f( void );        // (5); overloaded on return type
int i = f();             // select (4)
double d = f();          // select (5)
```

2.2.4 Variable Overloading

Unlike almost all programming languages, C∀ has variable overloading within a scope, along with shadow overloading in nested scopes.

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

The C∀ type system treats overloaded variables as an overloaded function returning a value with no parameters. Hence, no significant effort is required to support this feature.

2.2.5 Constructor and Destructor

While C∀ is not object-oriented, it adopts many language features commonly used in object-oriented languages; these features are independent of object-oriented programming.

All objects in C∀ are initialized by constructors *after* allocation and de-initialized *before* deallocation. C++ cannot have constructors for basic types because they have no aggregate type **struct**/**class** in which to insert a constructor definition. Like C++, C∀ has multiple auto-generated constructors for every type.

```
struct Employee {
    char * name;
    double salary;
};
void ?{}( Employee & emp, char * nname, double nsalary ) with( emp ) { // auto qualification
    name = aalloc( sizeof(nname) );
    strcpy( name, nname );
    salary = nsalary;
}
void ^?{}( Employee & emp ) {
    free( emp.name );
}
{
    Employee emp = { "Sara Schmidt", 20.5 }; // initialize with implicit constructor call
    ... // use emp
    ^?{}( emp );                          // explicit de-initialize
    ?{}( emp, "Jack Smith", 10.5 );       // explicit re-initialize
    ... // use emp
}                                         // de-initialize with implicit destructor call
```

Figure 2.2: C∀ Constructor and Destructor

The prototype for the constructor/destructor are **void** ?{}(T &, ...) and **void** ^?{}(T &, ...), respectively. The first parameter is logically the **this** or self in other object-oriented languages and implicitly passed. Figure 2.2 shows an example of creating and using a constructor and destructor. Both constructor and destructor can be explicitly called to reuse a variable.

2.2.6 Special Literals

The C constants 0 and 1 have special meaning. 0 is the null pointer and is used in conditional expressions, where **if** (p) is rewritten **if** (p != 0); 1 is an additive identity in unary operators ++ and −−. Aware of their significance, C∀ provides a special type **zero_t** and **one_t** for custom types.

```
struct S { int i, j; };
void ?{}( S & this, zero_t ) { this.i = 0; this.j = 0; } // zero_t, no parameter name allowed
int ?!=?( S this, zero_t ) { return this.i != 0 && this.j != 0; }
S s = 0;
if ( s != 0 ) ...
```

Similarity, for **one_t**.

```
void ?{}( S & this, one_t ) { this.i = 1; this.j = 1; } // one_t, no parameter name allowed
S & ?++( S & this, one_t ) { return (S){ this.i++, this.j++ }; }
```

2.2.7 Polymorphic Functions

Polymorphic functions are the functions that apply to all types. C∀ provides *parametric polymorphism* written with the **forall** clause.

```
forall( T ) T identity( T v ) { return v; }
identity( 42 );
```

The identity function accepts a value from any type as an argument and returns that value. At the call size, the type parameter T is bounded to **int** from the argument 42.

For polymorphic functions to be useful, the **forall** clause needs *type assertion*s that restrict the polymorphic types it accepts.

```
forall( T | { void foo( T ); } ) void bar( T t ) { foo( t ); }
struct S { ... } s;
void foo( struct S );
bar( s );
```

The assertion on T restricts the range of types that can be manipulated by bar to only those that implement foo with the matching signature, allowing bar's call to foo in its body. Unlike templates in C++, which are macro expansions at the call site, C∀ polymorphic functions are compiled, passing the call-site assertion functions as hidden parameters.

2.2.8 Trait

A **forall** clause can assert many restrictions on multiple types. A common practice is refactoring the assertions into a named *trait*, similar to other languages like Go and Rust.

```
forall(T) trait Bird {
    int days_can_fly( T );
    void fly( T );
};
forall( B | Bird( B ) )
void bird_fly( int days_since_born, B bird ) {
    if ( days_since_born > days_can_fly( bird )) fly( bird );
}
struct Robin {} robin;
int days_can_fly( Robin robin ) { return 23; }
void fly( Robin robin ) {}
bird_fly( 23, robin );
```

Grouping type assertions into a named trait effectively creates a reusable interface for parametric polymorphic types.

2.3 Expression Resolution

Overloading poses a challenge for all expression-resolution systems. Multiple overloaded names give multiple candidates at a call site, and a resolver must pick a *best* match, where "best" is defined by a series of heuristics based on safety and programmer intuition/expectation. When multiple best matches exist, the resolution is ambiguous.

The C∀ resolver attempts to identify the best candidate based on: first, the number of parameters and types, and second, when no exact match exists, the fewest implicit conversions and polymorphic variables. Finding an exact match is not discussed here, because the mechanism is

fairly straightforward, even when the search space is ample; only finding a non-exact match is discussed in detail.

2.3.1 Conversion Cost

Most programming languages perform some implicit conversions among basic types to facilitate mixed-mode arithmetic; otherwise, the program becomes littered with many explicit casts which do not match the programmer's expectations. C is an aggressive language, providing conversions among almost all basic types, even when the conversion is potentially unsafe or not meaningful, *i.e.*, **float** to **bool**. C defines the resolution pattern as "usual arithmetic conversion" [11, § 6.3.1.8], in which C looks for a *common type* between operands, and converts one or both operands to the common type. A common type is the smallest type in terms of the size of representation that both operands can be converted into without losing their precision, called a *widening* or *safe conversion*.

C∀ generalizes "usual arithmetic conversion" to *conversion cost*. In the first design by Bilson [2], conversion cost is a 3-tuple, (unsafe, poly, safe) applied between each argument/parameter type, where:

1. unsafe is the number of precision losing (*narrowing* conversions),
2. poly is the number of polymorphic function parameters, and
3. safe is the sum of the degree of safe (widening) conversions.

Sum of degree is a method to quantify C's integer and floating-point rank. Every pair of widening conversion types is assigned a *distance*, and the distance between the two same types is 0. For example, the distance from **char** to **int** is 2, from **int** to **long** is 1, and from **int** to **long long int** is 2. This distance does not mirror C's rank system. For example, the **char** and **signed char** ranks are the same in C, but the distance from **char** to **signed char** is assigned 1. safe cost is summing all pairs of arguments to parameter safe conversion distances. Among the three costs in Bilson's model, unsafe is the most significant cost, and safe is the least significant, implying that C∀ always chooses a candidate with the lowest unsafe, if possible.

For example, assume the overloaded function foo is called with two **int** parameters. The cost for every overloaded foo has been listed along with the following:

```
void foo( char, char );                 // (1) (2, 0, 0)
void foo( char, int );                  // (2) (1, 0, 0)
forall( T, V ) void foo( T, V );        // (3) (0, 2, 0)
forall( T ) void foo( T, T );           // (4) (0, 2, 0)
forall( T ) void foo( T, int );         // (5) (0, 1, 0)
void foo( long long, long );            // (6) (0, 0, 3)
void foo( long, long );                 // (7) (0, 0, 2)
void foo( int, long );                  // (8) (0, 0, 1)
int i, j;
foo( i, j );                            // convert j to long and call (8)
```

The overloaded instances are ordered from the highest to the lowest cost, and C∀ selects the last candidate (8).

In the next iteration of C∀, Schluntz and Aaron [15] expanded conversion cost to a 7-tuple with 4 additional categories, (unsafe, poly, safe, sign, vars, specialization, reference), with the following interpretations:

- *Unsafe*
- *Poly*
- *Safe*
- *Sign* is the number of sign/unsign variable conversions.
- *Vars* is the number of polymorphic type variables.
- *Specialization* is the negative value of the number of type assertions.
- *Reference* is number of reference-to-rvalue conversion.

The extended conversion-cost model looks for candidates that are more specific and less generic. vars disambiguates **forall**(T, V) foo(T, V) and **forall**(T) **void** foo(T, T), where the extra type parameter V makes is more generic. A more generic type means fewer constraints on its parameter types. C∀ favours candidates with more restrictions on polymorphism, so **forall**(T) **void** foo(T, T) has lower cost. specialization is an arbitrary count-down value starting at zero. For every type assertion in the **forall** clause (no assertions in the above example), C∀ subtracts one from specialization. More type assertions mean more constraints on argument types, making the function less generic.

C∀ defines two special cost values: 0 and infinite. A conversion cost is 0 when the argument and parameter have an exact match, and a conversion cost is infinite when there is no defined conversion between the two types. For example, the conversion cost from **int** to a **struct** S is infinite.

In C∀, the meaning of a C-style cast is determined by its Cast Cost. For most cast-expression resolutions, a cast cost equals a conversion cost. Cast cost exists as an independent matrix for conversion that cannot happen implicitly while being possible with an explicit cast. These conversions are often defined as having an infinite conversion cost and a non-infinite cast cost.

Chapter 3

C Enumeration in C∀

C∀ supports legacy C enumeration using the same syntax for backward compatibility. A C-style enumeration in C∀ is called a *C Enum*. The semantics of the C Enum are mostly consistent with C with some restrictions. The following sections detail all of my new contributions to enumerations in C.

3.1 Visibility

In C, unscoped enumerators present a *naming problem* when multiple enumeration types appear in the same scope with duplicate enumerator names.

```
enum E1 { First, Second, Third, Fourth };
enum E2 { Fourth, Third, Second, First };    // same enumerator names
```

There is no mechanism in C to resolve these naming conflicts other than renaming one of the duplicates, which may be impossible if the conflict comes from system include-files.

The C∀ type-system allows extensive overloading, including enumerators. For example, enumerator First from E1 can exist at the scope as First from E2. Hence, most ambiguities among C enumerators are implicitly resolved by the C∀ type system, possibly without any programmer knowledge of the conflict. In addition, C Enum qualification is added, exactly like aggregate field-qualification, to disambiguate. Figure 3.1 shows how resolution, qualification, and casting are used to disambiguate situations for enumerations E1 and E2.

Aside, name shadowing in C∀ only happens when a name has been redefined with the *exact* same type. Because an enumeration define its type and enumerators in one definition, shadowing an enumerator is not possible, *i.e.*, it is impossible to have another First with same type E1.

3.2 Scoping

A C Enum can be scoped, using '!', so the enumerator constants are not projected into the enclosing scope.

```
enum Week ! { Mon, Tue, Wed, Thu = 10, Fri, Sat, Sun };
enum RGB ! { Red, Green, Blue };
```

Now, the enumerators *must* be qualified with the associated enumeration type.

```
E1 f() { return Third; }                    // overload functions with different return types
E2 f() { return Fourth; }
void g( E1 e );
void h( E2 e );
void foo() {                                // different resolutions and dealing with ambiguities
    E1 e1 = First;   E2 e2 = First;         // initialization
    e1 = Second;   e2 = Second;             // assignment
    e1 = f();   e2 = f();                   // function return
    g( First );   h( First );               // function argument
    int i = E1.First + E2.First;            // disambiguate with qualification
    int j = (E1)First + (E2)First;          // disambiguate with cast
}
```

Figure 3.1: Enumerator Visibility and Disambiguating

```
Week week = Week.Mon;
week = Week.Sat;
RGB rgb = RGB.Red;
rgb = RGB.Blue;
```

It is possible to introduce enumerators from a scoped enumeration to a block scope using the C∀ **with** auto-qualification clause/statement (see also C++ **using enum** in Section 6.3).

```
with ( Week, RGB ) {                        // type names
    week = Sun;                             // no qualification
    rgb = Green;
}
```

As in Section 3.1, opening multiple scoped enumerations in a **with** can result in duplicate enumeration names, but C∀ implicit type resolution and explicit qualification/casting handle this localized scenario.

3.3 Type Safety

As in Section 2.1.2.3, C's implicit bidirectional conversion between enumeration and integral type raises a safety concern. In C∀, the conversion is changed to unidirectional: an enumeration can be implicitly converted into an integral type, with an associated safe conversion cost. However, an integral type cannot be implicitly converted into a C enumeration because the conversion cost is set to infinity.

```
enum Bird { Penguin, Robin, Eagle };
enum Fish { Shark, Salmon, Whale };

int i = Robin;                              // allow, implicitly converts to 1
enum Bird bird = 1;                         // disallow
enum Bird bird = Shark;                     // disallow
```

It is now up to the programmer to insert an explicit cast to force the assignment.

```
enum Bird bird = (Bird)1;
enum Bird bird = (Bird)Shark
```

Note, C++ has the same safe restriction and provides the same workaround cast:

Change: C++ objects of enumeration type can only be assigned values of the same enumeration type. In C, objects of enumeration type can be assigned values of any integral type. Example:

```
enum color { red, blue, green };
color c = 1;                                    // valid C, invalid C++
```

Rationale: The type-safe nature of C++.
Effect on original feature: Deletion of semantically well-defined feature.
Difficulty of converting: Syntactic transformation. (The type error produced by the assignment can be automatically corrected by applying an explicit cast.)
How widely used: Common.

ISO/IEC 14882:1998 (C++ Programming Language Standard) [10, C.1.5.7.2.5]

Chapter 4

C∀ Enumeration

C∀ extends C-Style enumeration by adding a number of new features that bring enumerations in line with other modern programming languages. Any enumeration extensions must be intuitive to C programmers in syntax and semantics. The following sections detail my new contributions to enumerations in C∀.

4.1 Syntax

C∀ extends the C enumeration declaration (see Section 2.1.2, p. 9) by parameterizing with a type (like a generic type) and adding Plan-9 inheritance (see Section 4.9, p. 27) using an **inline** to another enumeration type.

enum−specifier:
    enum (type−specifier$_{opt}$) *identifier*$_{opt}$ { *cfa−enumerator−list* }
    enum (type−specifier$_{opt}$) *identifier*$_{opt}$ { *cfa−enumerator−list* , }
    enum (type−specifier$_{opt}$) *identifier*
cfa−enumerator−list:
    cfa−enumerator
    cfa−enumerator−list, *cfa−enumerator*
cfa−enumerator:
    enumeration−constant
    inline enum−type−name
    enumeration−constant = *constant−expression*

4.2 Operations

C∀ enumerations have access to the three enumerations properties (see Section 1.1, p. 2): label, order (position), and value via three overloaded functions label, posn, and value (see Section 5, p. 34 for details). C∀ auto-generates these functions for every C∀ enumeration.

```
enum(int) E { A = 3 } e = A;
sout | A | label( A ) | posn( A ) | value( A );
sout | e | label( e ) | posn( e ) | value( e );
A A 0 3
A A 0 3
```

For output, the default is to print the label. An alternate way to get an enumerator's position is to cast it to **int**.

```
sout | A | label( A ) | (int)A | value( A );
sout | A | label( A ) | (int)A | value( A );
A A 0 3
A A 0 3
```

Finally, C∀ introduces an additional enumeration pseudo-function **countof** (like **sizeof**, **typeof**) that returns the number of enumerators in an enumeration.

```
enum(int) E { A, B, C, D } e;
countof( E );  // 4, type argument
countof( e );  // 4, variable argument
```

This built-in function replaces the C idiom for automatically computing the number of enumerators (see Section 2.1.2.3, p. 11).

```
enum E { A, B, C, D, N };  // N == 4
```

4.3 Opaque Enumeration

When an enumeration type is empty. it is an *opaque* enumeration.

```
enum() Mode { O_RDONLY, O_WRONLY, O_CREAT, O_TRUNC, O_APPEND };
```

Here, the compiler chooses the internal representation, which is hidden, so the enumerators cannot be initialized. Compared to the C enum, opaque enums are more restrictive regarding typing and cannot be implicitly converted to integers.

```
Mode mode = O_RDONLY;
int www = mode;                         // disallowed
```

Opaque enumerations have only two attribute properties, label and posn.

```
char * s = label( O_TRUNC );            // "O_TRUNC"
int open = posn( O_WRONLY );            // 1
s = label( mode );                      // "O_RDONLY"
int open = posn( mode );                // 0
```

Equality and relational operations are available.

```
if ( mode == O_CREAT ) ...
bool b = mode < O_APPEND;
```

4.4 Typed Enumeration

When an enumeration type is specified, all enumerators have that type and can be initialized with constants of that type or compile-time convertible to that type. Figure 4.1 shows a series of examples illustrating that all C∀ types can be used with an enumeration, and each type's values are used to set the enumerator constants. Note the use of the synonyms Liz and Beth in the last declaration. Because enumerators are constants, the enumeration type is implicitly **const**, so all the enumerator types in Figure 4.1 are logically rewritten with **const**.

```
// integral
    enum( char ) Currency { Dollar = '$', Cent = '¢', Yen = '¥', Pound = '£', Euro = 'E' };
    enum( signed char ) srgb { Red = -1, Green = 0, Blue = 1 };
    enum( long long int ) BigNum { X = 123_456_789_012_345,  Y = 345_012_789_456_123 };
// non-integral
    enum( double ) Math { PI_2 = 1.570796, PI = 3.141597, E = 2.718282 };
    enum( _Complex ) Plane { X = 1.5+3.4i, Y = 7+3i, Z = 0+0.5i };
// 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 ] };            // new CA type
// function
    void f() {...}   void g() {...}
    enum( void (*)() ) funs { F = f,  G = g };
// aggregate
    struct Person { char * name; int age, height; };
    enum( Person ) friends { Liz = { "ELIZABETH", 22, 170 }, Beth = Liz,
                             Jon = { "JONATHAN", 35, 190 } };
```

Figure 4.1: Enumerator Typing

An advantage of the typed enumerations is eliminating the *harmonizing* problem between an enumeration and companion data (see Section 2.1.2.3, p. 11):

```
enum( char * ) integral_types {
    chr = "char", schar = "signed char", uschar = "unsigned char",
    sshort = "signed short int", ushort = "unsigned short int",
    sint = "signed int", usint = "unsigned int",
    ...
};
```

Note that the enumeration type can be a structure (see Person in Figure 4.1), so it is possible to have the equivalent of multiple arrays of companion data using an array of structures.

While the enumeration type can be any C aggregate, the aggregate's CA constructors are *not* used to evaluate an enumerator's value. CA enumeration constants are compile-time values (static); calling constructors happens at runtime (dynamic).

4.5 Implementation

CA-cc is a transpiler translating CA code into C, which is compiled by a C compiler. During transpiling, CA-cc breaks a CA enumeration definition into a definition of a C enumeration with the same name and auxiliary arrays: a label and value array for a typed enumeration. For example:

```
enum( T ) E { E1 = t1, E2 = t2, E3 = t3 };
```

is compiled into:

```
enum E { E1, E2, E3 };
const char * E_labels[3] = { "E1", "E2", "E3" };
const T E_values[3] = { t1, t2, t3 };
```

The generated C enumeration has enumerator values that match C∀ enumerator positions because of C's auto-initialization. A C∀ enumeration variable definition is the same in C∀ as C, *e.g.*:

```
enum E e = E1;
e = E2;
```

so these expressions remain unchanged by C∀-cc. Therefore, a C∀ enumeration variable has the same underlying representation as its generated C enumeration. This semantics implies a C∀ enumeration variable uses the same storage as a C enumeration variable, that posn can use as its underlying representation, and the label and value arrays take little storage. The arrays are annotated with linkonce, which are replaced later by gcc attribute section(".gnu.linkonce.NAME"), to prevent multiple definitions. It should be possible to eliminate the two arrays if unused, either by C∀ if local to a translation unit and unused, or by the linker if global but unreferenced. Also, the label and value arrays are declared **static** and initialized with constants, so the arrays are allocated in the .data section and initialized before program execution. Hence, there is no additional execution cost unless new enumeration features are used, and storage usage is minimal as the number of enumerations in a program is small as is the number of enumerators in an enumeration.

Along with the enumeration definition, C∀-cc generates definitions of the attribute functions, posn, label and value, for each enumeration:

```
inline int posn( E e ) { return (int) e; }
inline const * label( E e ) { return E_labels[ (int) e ]; }
inline const * E_value( E e ) { return E_values[ (int) e ]; }
```

where the function calls are normally inlined by the backend C compiler into a few instructions. These functions simplify the job of getting the enumerations types through the type system in the same way as normal functions and calls. Note, the cast to **int** is actually an internal reinterpreted cast added before type resolution to stop further reduction on the expression by the type resolver (see Section 4.6) and removed in code generation. Finally, to further mitigate C∀ enumeration costs, calls to label and value with an enumeration constant are unrolled into the appropriate constant expression, although this could be left to the backend C compiler. Hence, in space and time costs, C∀ enumerations follow the C philosophy of only paying for what is used, modulo some future work to convince the linker to remove un-accessed label and value arrays, possibly with weak attributes.

4.6 Value Conversion

C has an implicit type conversion from an enumerator to its base type **int**. Correspondingly, C∀ has an implicit conversion from a typed enumerator to its base type, allowing typed enumeration to be seamlessly used as the value of its base type For example, using type Currency in Figure 4.1:

```
char currency = Dollar;                 // implicit conversion to base type
void foo( char );
foo( Dollar );                          // implicit conversion to base type
```

The implicit conversion induces a *value cost*, which is a new category (8 tuple) in C∀'s conversion cost model (see Section 2.3.1, p. 17) to disambiguate function overloading over a C∀ enumeration and its base type.

```
void baz( char ch );                    // (1)
void baz( Currency cu );                // (2)
baz( Dollar );
```

While both baz functions are applicable to the enumerator Dollar, candidate (1) comes with a value cost for the conversion to the enumeration's base type, while candidate (2) has zero cost. Hence, C∀ chooses the exact match. Value cost is defined to be a more significant factor than an unsafe but less than the other conversion costs: (unsafe, value, poly, safe, sign, vars, specialization, reference).

```
void bar( int );
Math x = PI;                            // (1)
double x = 5.5;                         // (2)
bar( x );                               // costs (1, 0, 0, 0, 0, 0, 0, 0) or (0, 1, 0, 0, 0, 0, 0, 0)
```

Here, the candidate (1) has a value conversion cost to convert to the base type, while the candidate (2) has an unsafe conversion from **double** to **int**, which is a more expensive conversion. Hence, bar(x) resolves x as type Math.

4.7 Initialization

C∀ extends C's auto-initialization scheme to C∀ enumeration. For an enumeration type with base type T, the initialization scheme is the following:

1. the first enumerator is initialized with T's **zero_t**.
2. Every other enumerator is initialized with its previous enumerator's value "+1", where "+1" is defined in terms of overloaded operator ?+?(T, **one_t**).

```
struct S { int i; };
S ?+?( S & s, one_t ) { return s.i++; }
void ?{}( S & s, zero_t ) { s.i = 0; }
enum(S) E { A, B, C, D };
```

The restriction on C's enumeration initializers being constant expression is relaxed on C∀ enumeration. Therefore, an enumerator initializer allows function calls like ?+?(S & s, **one_t**) and ?{}(S & s, **zero_t**). It is because the values of C∀ enumerators are not stored in the compiled enumeration body but in the value array, which allows dynamic initialization.

4.8 Subset

An enumeration's type can be another enumeration.

```
enum( char ) Letter { A = 'A', ..., Z = 'Z' };
enum( Letter ) Greek { Alph = A, Beta = B, Gamma = G, ..., Zeta = Z }; // alphabet intersection
```

Enumeration Greek may have more or less enumerators than Letter, but its enumerator values *must* be from Letter. Therefore, the set of Greek enumerator values in a subset of the Letter

enumerator values. Letter is type compatible with enumeration Letter because value conversions are inserted whenever Letter is used in place of Greek.

```
Letter l = A;                           // allowed
Greek g = Alph;                         // allowed
l = Alph;                               // allowed, conversion to base type
g = A;                                  // disallowed
void foo( Letter );
foo( Beta );                            // allowed, conversion to base type
void bar( Greek );
bar( A );                               // disallowed
```

Hence, Letter enumerators are not type-compatible with the Greek enumeration, but the reverse is true.

4.9 Inheritance

C∀ Plan-9 inheritance may be used with C∀ enumerations, where Plan-9 inheritance is containment inheritance with implicit un-scoping (like a nested unnamed **struct/union** in C). Containment is nominative: an enumeration inherits all enumerators from another enumeration by declaring an **inline** statement in its enumerator lists.

```
enum( char * ) Names { /*(see Figure 4.1, p. 24) */ };
enum( char * ) Names2 { inline Names, Jack = "JACK", Jill = "JILL" };
enum( char * ) Names3 { inline Names2, Sue = "SUE", Tom = "TOM" };
```

In the preceding example, Names2 is defined with five enumerators, three of which are from Name through containment, and two are self-declared. Names3 inherits all five members from Names2 and declares two additional enumerators. Hence, enumeration inheritance forms a subset relationship. Specifically, the inheritance relationship for the example above is:

```
Names ⊂ Names2 ⊂ Names3                 // enum type of Names
```

Inheritance can be nested, and a C∀ enumeration can inline enumerators from more than one C∀ enumeration, forming a tree-like hierarchy. However, the uniqueness of the enumeration name applies to enumerators, including those from supertypes, meaning an enumeration cannot name an enumerator with the same label as its subtype's members or inherits from multiple enumeration that has overlapping enumerator labels. Consequently, a new type cannot inherit from an enumeration and its supertype or two enumerations with a common supertype (the diamond problem) since such would unavoidably introduce duplicate enumerator labels.

The base type must be consistent between subtype and supertype. When an enumeration inherits enumerators from another enumeration, it copies the enumerators' value and label, even if the value is auto-initialized. However, the underlying representation's position is the enumerator's order in the new enumeration.

```
enum() E1 { B };                        // B
enum() E2 { C, D };                     // C D
enum() E3 { inline E1, inline E2, E };  // [E1 B ] [E2 C D ] E
enum() E4 { A, inline E3, F};           // A [E3 [E1 B ] [E2 C D ] E ] F
```

In this example, B is at position 0 in E1 and E3, but position 1 in E4 as A takes position 0 in E4. C is at position 0 in E2, 1 in E3, and 2 in E4. D is at position 1 in E2, 2 in E3, and 3 in E4.

A subtype enumeration can be casted, or implicitly converted into its supertype, with a safe cost, called *enumeration conversion*.

```
enum E2 e2 = C;
posn( e2 );             // 0
enum E3 e3 = e2;        // Assignment with enumeration conversion E2 to E3
posn( e2 );             // 1 cost
void foo( E3 e );
foo( e2 );              // Type compatible with enumeration conversion E2 to E3
posn( (E3)e2 );         // Explicit cast with enumeration conversion E2 to E3
E3 e31 = B;             // No conversion: E3.B
posn( e31 );            // 0 cost
```

The last expression is unambiguous. While both E2.B and E3.B are valid candidates, E2.B has an associated safe cost and E3.B needs no conversion (zero cost). C∀ selects the lowest cost candidate E3.B.

For the given function prototypes, the following calls are valid.

```
void f( Names );          f( Fred );
void g( Names2 );         g( Fred );   g( Jill );
void h( Names3 );         h( Fred );   h( Jill );   h( Sue );
void j( const char * );   j( Fred );   j( Jill );   j( Sue );   j( "WILL" );
```

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.

4.9.1 Offset Calculation

As discussed in Section 4.3, p. 23, C∀ chooses position as a representation of a C∀ enumeration variable. When a cast or implicit conversion moves an enumeration from subtype to supertype, the position can be unchanged or increased. C∀ determines the position offset with an *offset calculation* function.

Figure 4.2 shows an outline of the offset calculation in C++. Structure CFAEnum represents the C∀ enumeration with a vector of variants of CFAEnum or Enumerator. The algorithm takes two CFAEnums parameters, src and dst, with src being the type of expression the conversion applies to, and dst being the type the expression is cast to. The algorithm iterates over the members in dst to find src. If a member is an enumerator of dst, the positions of all subsequent members are incremented by one. If the current member is dst, the function returns true, indicating *found* and the accumulated offset. Otherwise, the algorithm recurses into the current CFAEnum m to check if its src is convertible to m. If src is convertible to the current member m, this means src is a subtype-of-subtype of dst. The offset between src and dst is the sum of the offset of m in dst and the offset of src in m. If src is not a subtype of m, the loop continues but with the offset shifted by the size of m. If the loop ends, than src is not convertible to dst, and false is returned.

```
struct Enumerator;
struct CFAEnum { vector<variant<CFAEnum, Enumerator>> members; string name; };
inline static bool operator==(CFAEnum& lhs, CFAEnum& rhs) { return lhs.name == rhs.name; }
pair<bool, int> calculateEnumOffset(CFAEnum src, CFAEnum dst) {
    int offset = 0;
    if ( src == dst ) return make_pair(true, 0);
    for ( auto v : dst.members ) {
        if ( holds_alternative<Enumerator>(v) ) {
            offset++;
        } else {
            auto m = get<CFAEnum>(v);
            if ( m == src ) return make_pair( true, offset );
            auto dist = calculateEnumOffset( src, m );
            if ( dist.first ) {
                return make_pair( true, offset + dist.second );
            } else {
                offset += dist.second;
            }
        }
    }
    return make_pair( false, offset );
}
```

Figure 4.2: Compute Offset from Subtype Enumeration to a Supertype

4.10 Control Structures

Enumerators are used in various contexts. In most programming languages, an enumerator is implicitly converted to its value (like a typed macro substitution). However, in some contexts, enumerator synonyms and typed enumerations make this implicit conversion to value incorrect. A programmer's intuition assumes an implicit conversion to position in these contexts.

For example, an intuitive use of enumerations is with the C∀ **switch/choose** statement, where **choose** performs an implicit **break** rather than a fall-through at the end of a **case** clause. (For this discussion, ignore the fact that **case** requires a compile-time constant.)

```
enum Count { First, Second, Third, Fourth };
Count e;
```

```
choose( e ) {
    case First: ...;
    case Second: ...;
    case Third: ...;
    case Fourth: ...;
}
```

```
// rewrite
choose( value( e ) ) {
    case value( First ): ...;
    case value( Second ): ...;
    case value( Third ): ...;
    case value( Fourth ): ...;
}
```

Here, the intuitive code on the left is implicitly transformed into the standard implementation on the right, using the value of the enumeration variable and enumerators. However, this implementation is fragile, *e.g.*, if the enumeration is changed to:

```
enum Count { First, Second, Third = First, Fourth };
```

making Third == First and Fourth == Second, causing a compilation error because of duplicate **case** clauses. To better match with programmer intuition, C∀ toggles between value and position semantics depending on the language context. For conditional clauses and switch statements, C∀ uses the robust position implementation.

```
if ( posn( e ) < posn( Third ) ) ...
choose( posn( e ) ) {
    case posn( First ): ...;
    case posn( Second ): ...;
    case posn( Third ): ...;
    case posn( Fourth ): ...;
}
```

C∀ provides a special form of for-control for enumerating through an enumeration, where the range is a type.

```
for ( cx; Count ) { sout | cx | nonl; } sout | nl;
for ( cx; ~= Count ) { sout | cx | nonl; } sout | nl;
for ( cx; -~= Count ) { sout | cx | nonl; } sout | nl;
First Second Third Fourth
First Second Third Fourth
Fourth Third Second First
```

The enumeration type is syntax sugar for looping over all enumerators and assigning each enumerator to the loop index, whose type is inferred from the range type. The prefix +~= or −~= iterate forward or backward through the inclusive enumeration range, where no prefix defaults to +~=.

C has an idiom for **if** and loop predicates of comparing the predicate result "not equal to 0".

```
if ( x + y /* != 0 */ ) ...
while ( p /* != 0 */ ) ...
```

This idiom extends to enumerations because there is a boolean conversion regarding the enumeration value if and only if such a conversion is available. For example, such a conversion exists for all numerical types (integral and floating-point). It is possible to explicitly extend this idiom to any typed enumeration by overloading the != operator.

```
bool ?!=?( Name n, zero_t ) { return n != Fred; }
Name n = Mary;
if ( n ) ... // result is true
```

Specialize meanings are also possible.

```
enum(int) ErrorCode { Normal = 0, Slow = 1, Overheat = 1000, OutOfResource = 1001 };
bool ?!=?( ErrorCode ec, zero_t ) { return ec >= Overheat; }
ErrorCode code = ...;
if ( code ) { problem(); }
```

4.11 Dimension

Section 4.4, p. 23 introduces the harmonizing problem between an enumeration and secondary information. When possible, using a typed enumeration for the secondary information is the best

approach. However, there are times when combining these two types is not possible. For example, the secondary information might precede the enumeration and/or its type is needed directly to declare parameters of functions. In these cases, having secondary arrays of the enumeration size are necessary.

To support some level of harmonizing in these cases, an array dimension can be defined using an enumerator type, and the enumerators used as subscripts.

```
enum E1 { A, B, C, N }; // possibly predefined
enum(int) E2 { A, B, C };
float H1[N] = { [A] :[1] 3.4, [B] : 7.1, [C] : 0.01 }; // C
float H2[E2] = { [A] : 3.4, [B] : 7.1, [C] : 0.01 }; // CFA
```

This approach is also necessary for a predefined typed enumeration (unchangeable) when additional secondary information needs to be added. The array subscript operator, namely ?[?], is overloaded so that when a C∀ enumerator is used as an array index, it implicitly converts to its position over value to sustain data harmonization. This behaviour can be reverted by explicit overloading:

```
float ?[?]( float * arr, E2 index ) { return arr[ value( index ) ]; }
```

While enumerator labels A, B and C are being defined twice in different enumerations, they are unambiguous within the context. Designators in H1 are unambiguous becasue E2 has a value cost to **int**, which is more expensive than safe cost from C-Enum E1 to **int**. Designators in H2 are resolved as E2 because when a C∀ enumeration type is being used as an array dimension, C∀ adds the enumeration type to the initializer's resolution context.

4.12 I/O

As seen in multiple examples, C∀ enumerations can be printed and the default property printed is the enumerator's label, which is similar in other programming languages. However, very few programming languages provide a mechanism to read in enumerator values. Even the boolean type in many languages does not have a mechanism for input using the enumerators true or false. Figure 4.3 show C∀ enumeration input based on the enumerator labels. When the enumerator labels are packed together in the input stream, the input algorithm scans for the longest matching string. For basic types in C∀, the rule is that the same constants used to initialize a variable in a program are available to initialize a variable using input, where string constants can be quoted or unquoted.

4.13 Planet Example

Figure 4.4, p. 33 shows an archetypal enumeration example illustrating most of the C∀ enumeration features. Planet is an enumeration of type MR. Each planet enumerator is initialized to a specific mass/radius, MR, value. The unnamed enumeration provides the gravitational-constant enumerator G. Function surfaceGravity uses the **with** clause to remove p qualification from fields mass and radius. The program main uses the pseudo function **countof** to obtain the number

[1] C uses symbol '=' for designator initialization, but C∀ changes it to ':' because of problems with tuple syntax.

```
int main() {
    enum(int ) E { BBB = 3, AAA, AA, AB, B };
    E e;

    try {
        for () {
            try {
                sin | e;
            } catch( missing_data * ) {
                sout | "missing data";
                continue; // try again
            }
            sout | e | "=  " | value( e );
        }
    } catch( end_of_file ) {}
}
```

```
input
BBBABAAAAB
BBB AAA AA AB B

output
BBB = 3
AB = 6
AAA = 4
AB = 6
BBB = 3
AAA = 4
AA = 5
AB = 6
B = 7
```

Figure 4.3: Enumeration I/O

of enumerators in Planet, and safely converts the random value into a Planet enumerator using fromInt. The resulting random orbital-body is used in a **choose** statement. The enumerators in the **case** clause use the enumerator position for testing. The prints use label to print an enumerator's name. Finally, a loop enumerates through the planets computing the weight on each planet for a given Earth mass. The print statement does an equality comparison with an enumeration variable and enumerator (p == MOON).

```
struct MR { double mass, radius; };                     // planet definition
enum( MR ) Planet {                                     // typed enumeration
    //                mass (kg)   radius (km)
    MERCURY = { 0.330_E24, 2.4397_E6 },
    VENUS   = { 4.869_E24, 6.0518_E6 },
    EARTH   = { 5.976_E24, 6.3781_E6 },
    MOON    = { 7.346_E22, 1.7380_E6 },                 // not a planet
    MARS    = { 0.642_E24, 3.3972_E6 },
    JUPITER = { 1898._E24, 71.492_E6 },
    SATURN  = { 568.8_E24, 60.268_E6 },
    URANUS  = { 86.86_E24, 25.559_E6 },
    NEPTUNE = { 102.4_E24, 24.746_E6 },
    PLUTO   = { 1.303_E22, 1.1880_E6 },                 // not a planet
};
enum( double ) { G = 6.6743_E-11 };                     // universal gravitational constant (m3 kg-1 s-2)
static double surfaceGravity( Planet p ) with( p ) {
    return G * mass / ( radius \ 2 );                   // no qualification, exponentiation
}
static double surfaceWeight( Planet p, double otherMass ) {
    return otherMass * surfaceGravity( p );
}
int main( int argc, char * argv[] ) {
    if ( argc != 2 ) exit | "Usage:  " | argv[0] | "earth-weight"; // terminate program
    double earthWeight = convert( argv[1] );
    double earthMass = earthWeight / surfaceGravity( EARTH );
    Planet rp = fromInt( prng( countof( Planet ) ) );   // select random orbiting body
    choose( rp ) {                                      // implicit breaks
      case MERCURY, VENUS, EARTH, MARS:
        sout | rp | "is a rocky planet";
      case JUPITER, SATURN, URANUS, NEPTUNE:
        sout | rp | "is a gas-giant planet";
      default:
        sout | rp | "is not a planet";
    }
    for ( p; Planet ) {                                 // enumerate
        sout | "Your weight on" | ( p == MOON ? "the" : "  " ) | p
             | "is" | wd( 1,1, surfaceWeight( p, earthMass ) ) | "kg";
    }
}
$ planet 100
JUPITER is a gas-giant planet
Your weight on MERCURY is 37.7 kg
Your weight on VENUS is 90.5 kg
Your weight on EARTH is 100.0 kg
Your weight on the MOON is 16.6 kg
Your weight on MARS is 37.9 kg
Your weight on JUPITER is 252.8 kg
Your weight on SATURN is 106.6 kg
Your weight on URANUS is 90.5 kg
Your weight on NEPTUNE is 113.8 kg
Your weight on PLUTO is 6.3 kg
```

Figure 4.4: Planet Example

Chapter 5

Enumeration Traits

C++ introduced the std::is_enum trait in C++11 and concept feature in C++20. This combination makes it possible to write a polymorphic function over an enumerated type.

```
#include <type_traits>
template<typename T> concept Enumerable = std::is_enum<T>::value;
template<Enumerable E> E f( E e ) {          // constrained type
    E w = e;                                 // allocation and copy
    cout << e << ' ' << w << endl;           // value
    return w;                                // copy
}
int main() {
    enum E { A = 42, B, C } e = C;
    e = f( e );
}
44 44
```

The std::is_enum and other C++ traits are compile-time interfaces to query type information. While named the same as **trait** in other programming languages, it is orthogonal to the C∀ trait, with the latter being defined as a collection of assertions to be satisfied by a polymorphic type.

The following sections cover the underlying implementation features I created to generalize and restrict enumerations in the C∀ type-system using the **trait** mechanism.

5.1 Traits CfaEnum and TypedEnum

Traits CfaEnum and TypedEnum define the enumeration attributes: label, posn, value, and Countof. These traits support polymorphic functions for C∀ enumeration, *e.g.*:

```
forall( E ) | CfaEnum( E ) )
void f( E e ) {
    // access enumeration properties for e
}
```

```
forall( E, V | TypedEnum( E, V ) | { string str( V ); } ) // format any enumeration
string format_enum( E e ) {
    return label( e ) + '(' + str( value( e ) ) + ')'; // "label( value )"
}
enum(size_t) RGB { Red = 0xFF0000, Green = 0x00FF00, Blue = 0x0000FF };
// string library has conversion function str from size_t to string

struct color_code { int R, G, B; };
enum(color_code) Rainbow {
    Red = {255, 0, 0}, Orange = {255, 127, 0}, Yellow = {255, 255, 0}, Green = {0, 255, 0}, // ...
};
string str( color_code cc ) with( cc ) {          // format payload, "ddd,ddd,ddd"
    return str( R ) + ',' + str( G ) + ',' + str( B ); // "R,G,B"
}
int main() {
    sout | format_enum( RGB.Green );          // "Green(65280)"
    sout | format_enum( Rainbow.Green );      // "Green(0,255,0)"
}
```

Figure 5.1: Generalized Enumeration Formatter

Trait CfaEnum defines attribute functions label and posn for all C∀ enumerations, and internally C∀ enumerations fulfill this assertion.

```
forall( E ) trait CfaEnum {
    const char * label( E e );
    unsigned int posn( E e );
};
```

This trait covers opaque enumerations that do not have an explicit value.

The trait TypedEnum extends CfaEnum with the value assertion for typed enumerations.

```
forall( E, V | CfaEnum( E ) ) trait TypedEnum {
    V value( E e );
};
```

Here, the associate type-parameter V is the base type of the typed enumeration, and hence, the return type of value. These two traits provide a way to define functions over all C∀ enumerations.

For example, Figure 5.1 shows a generalized enumeration formatter for any enumeration type. The formatter prints an enumerator name and its value in the form `"label( value )"`. The trait for format_enum requires a function named str to print the value (payload) of the enumerator. Hence, enumeration defines how its value appears, and format_enum displays this value within the label name.

Other types may work with traits CfaEnum and TypedEnum, by supplying appropriate label, posn, and value functions. For example, Figure 5.2 extends a (possibly predefined) C enumeration to work with all the C∀ extensions.

```
enum Fruit { Apple, Banana, Cherry };        // C enum
const char * label( Fruit f ) {
    static const char * labels[] = { "Apple", "Banana", "Cherry" };
    return labels[f];
}
int posn( Fruit f ) { return f; }
int value( Fruit f ) {
    static const char values[] = { 'a', 'b', 'c' };
    return values[f];
}
sout | format_enum( Cherry );                // "Cherry(c)"
```

Figure 5.2: Extend C Enumeration to C∀ Enumeration

5.2 Discussion: Genericity

At the start of this chapter, the C++ concept is introduced to constrained template types, *e.g.*:

```
concept Enumerable = std::is_enum<T>::value;
```

Here, concept is referring directly to types with kind **enum**; other concepts can refer to all types with kind **int** with **long** or **long long** qualifiers, *etc*. Hence, the concept is the first level of restriction, allowing only the specified kinds of types and rejecting others. The template expansion is the second level of restriction verifying if the type passing the concept test provides the necessary functionality. Hence, a concept is querying precise aspects of the programming language set of types.

Alternatively, languages using traits, like C∀, Scala, Go, and Rust, are defining a restriction based on a set of operations, variables, or structure fields that must exist to match with usages in a function or aggregate type. Hence, the C∀ enumeration traits are never connected with the specific **enum** kind. Instead, anything that can look like the **enum** kind is considered an enumeration (static structural typing). However, Scala, Go, and Rust traits are nominative: a type explicitly declares a named trait to be of its type, while in C∀, any type implementing all requirements declared in a trait implicitly satisfy its restrictions.

One of the key differences between concepts and traits, which is leveraged heavily by C∀, is the ability to apply new C∀ features to C legacy code. For example, Figure 5.1 shows that pre-existing C enumerations can be upgraded to work and play with new C∀ enumeration facilities. Another example is adding constructors and destructors to pre-existing C types by simply declaring them for the old C type. C++ fails at certain levels of legacy extension because many of the new C++ features must appear *within* an aggregate definition due to the object-oriented nature of the type system, where it is impossible to change legacy library types.

5.3 Bounded and Serial

A bounded trait defines a lower and upper bound for a type.

```
forall( E ) trait Bounded {
    E lowerBound();
```

```
    E upperBound();
};
```

Both functions are necessary for the implementation of C∀ enumeration, with lowerBound returning the first enumerator and upperBound returning the last enumerator.

```
enum(int) Week { Mon, Tue, Wed, Thu, Fri, Sat, Sun };
enum(int) Fruit { Apple, Banana, Cherry };
Week first_day = lowerBound();          // Mon
Fruit last_fruit = upperBound();        // Cherry
```

The lowerBound and upperBound are functions overloaded on return type only, meaning their type resolution depends solely on the call-site context, such as the parameter type for a function argument or the left-hand side of an assignment expression. Calling either function without a context results in a type ambiguity, unless the type environment has only one type overloading the functions.

```
sout | lowerBound();                   // ambiguous as Week and Fruit implement Bounded
void foo( Fruit );
foo( lowerBound() );                   // parameter provides type Fruit
Week day = upperBound();               // day provides type Week
```

Trait Serial is a subset of Bounded, with functions mapping enumerators to integers, and implementing a sequential order between enumerators.

```
forall( E | Bounded( E ) ) trait Serial {
    int fromInstance( E e );
    E fromInt( unsigned int i );
    E pred( E e );
    E succ( E e );
    unsigned Countof( E );
};
```

Function fromInstance projects a Bounded member to a number and fromInt is the inverse. Function pred and succ are advancement functions: pred takes an enumerator and returns the previous enumerator, if there is one, in sequential order, and succ returns the next enumerator.

```
sout | fromInstance( Wed ) | fromInt( 2 ) | succ( Wed ) | pred( Wed );
2 Wed Thu Tue
```

Bound checking is provided for fromInt, pred, and succ, and the program is terminated if the lower or upper bound is exceeded, *e.g.*:

```
fromInt( 100 );
Cforall Runtime error: call to fromInt has index 100 outside of enumeration range 0−6.
```

Function fromInstance or a position cast using (**int**) is always safe, *i.e.*, within the enumeration range.

Function Countof is the generic counterpart to the built-in pseudo-function **countof**. **countof** only works on enumeration types and instances, so it is locked into the language type system; as such, **countof**(**enum**−type) becomes a compile-time constant. Countof works on any type that matches the Serial trait. Hence, Countof does not use its argument; only the parameter type is needed to compute the range size.

```
int Countof( E ) {
```

```
    E upper = upperBound();
    E lower = lowerBound();
    return fromInstance( upper ) − fromInstance( lower ) + 1;
}
```

countof also works for any type E that defines Countof and lowerBound, becoming a call to Countof(E). The resolution step on expression **countof**(E) are:

1. Look for an enumeration named E, such as **enum** E {... }. If such an enumeration E exists, replace **countof**(E) with the number of enumerators.
2. Look for a non-enumeration type named E that defines Countof and lowerBound, including E being a polymorphic type, such as **forall**(E). If type E exists, replace it with Countof(lowerBound()), where lowerBound is defined for type E.
3. Look for an enumerator A defined in enumeration E. If such an enumerator A exists, replace **countof**(A) with the number of enumerators in E.
4. Look for a name A in the lexical context with the type E. If the name A exists, replace **countof**(A) with a function call Countof(E).
5. If 1-4 fail, report a type error on expression **countof**(E).

5.4 Enumerating

The fundamental aspect of an enumeration type is the ability to enumerate over its enumerators. C∀ supports *for* loops, *while* loop, and *range* loop. This section covers **for** loops and range loops for enumeration, but the concept transitions to **while** loop.

5.4.1 For Loop

A for-loop consists of loop control and body. The loop control is often a 3-tuple: initializers, looping condition, and advancement. It is a common practice to declare one or more loop-index variables in initializers, whether the variables satisfy the loop condition, and update the variables in advancement. Such a variable is called an *index* and is available for reading and writing within the loop body. (Some languages make the index read-only in the loop body.) This style of iteration can be written for an enumeration using functions from the Bounded and Serial traits:

```
enum() E { A, B, C, D };
for ( unsigned int i = 0; i < countof(E); i += 1 ) // (1)
    sout | label( fromInt( i ) ) | nonl;
sout | nl;
for ( E e = lowerBound(); ; e = succ(e) ) {     // (2)
    sout | label(e) | nonl;
  if (e == upperBound()) break;
}
sout | nl;
A B C D
A B C D
```

A caveat in writing loop control using pred and succ is unintentionally exceeding the range.

```
for ( E e = upperBound(); e >= lowerBound(); e = pred( e ) ) {}
for ( E e = lowerBound(); e <= upperBound(); e = succ( e ) ) {}
```

Both of these loops look correct but fail because there is an additional bound check within the advancement *before* the conditional test to stop the loop, resulting in a failure at the endpoints of the iteration. These loops must be restructured by moving the loop test to the end of the loop (**do**–**while**), as in loop (2) above, which is safe because an enumeration always has at least one enumerator.

5.4.2 Range Loop

Instead of writing the traditional 3-tuple loop control, C∀ supports a *range loop*.

```
for ( E e; A ~= D ) { sout | label( e ) | nonl; } sout | nl;
for ( e; A ~= D ) { sout | label( e ) | nonl; } sout | nl;
for ( E e; A -~= D ) { sout | label( e ) | nonl; } sout | nl;
for ( e; A -~= D ~ 2 ) { sout | label( e ) | nonl; } sout | nl;
```

Every range loop above has an index declaration and a range bounded by *left bound* A and *right bound* D. If the index declaration-type is omitted, the index type is the type of the lower bound (**typeof**(A)). If a range is joined by ~= (range up equal) operator, the index variable is initialized by the left bound and advanced by 1 until it is greater than the right bound. If a range is joined by -~= (range down equal) operator, the index variable is initialized by the right bound and advanced by -1 until it is less than the left bound. (Note, functions pred and succ are not used for advancement, so the advancement problem does not occur.) A range can be suffixed by a positive *step*, *e.g.*, ~ 2, so advancement is incremented/decremented by step.

Finally, a shorthand for enumerating over the entire set of enumerators (the most common case) is using the enumeration type for the range.

```
for ( e; E ) sout | label( e ) | nonl; sout | nl;    // A B C D
for ( e; -~= E ) sout | label( e ) | nonl; sout | nl; // D C B A
```

For a C∀ enumeration, the loop enumerates over all enumerators of the enumeration. For a type matching the Serial trait: the index variable is initialized to lowerBound and loop control checks the index's value for greater than the upperBound. If the range type is not a C∀ enumeration or does not match trait Serial, it is compile-time error.

5.5 Overload Operators

C∀ overloads the comparison operators for C∀ enumeration satisfying traits Serial and CfaEnum. These definitions require the operand types to be the same, and the appropriate comparison is made using the the positions of the operands.

```
forall( E | CfaEnum( E ) | Serial( E ) ) {      // distribution block
    // comparison
    int ?==?( E l, E r );                       // true if l and r are same enumerators
    int ?!=?( E l, E r );                       // true if l and r are different enumerators
    int ?<?( E l, E r );                        // true if l is an enumerator before r
    int ?<=?( E l, E r );                       // true if l before or the same as r
```

```
    int ?>?( E l, E r );                    // true if l is an enumerator after r
    int ?>=?( E l, E r );                   // true if l after or the same as r
}
```

(Note, all the function prototypes are wrapped in a distribution block, where all qualifiers preceding the block are distributed to each declaration with the block, which eliminates tedious repeated qualification. Distribution blocks can be nested.)

C∀ implements a few arithmetic operators for CfaEnum. Bound checks are added to these operations to ensure the outputs fulfill the Bounded invariant.

```
forall( E | CfaEnum( E ) | Serial( E ) ) {      // distribution block
    // comparison
    E ++?( E & l );
    E --?( E & l );
    E ?+=? ( E & l, one_t );
    E ?-=? ( E & l, one_t );
    E ?+=? ( E & l, int i );
    E ?-=? ( E & l, int i );
    E ?++( E & l );
    E ?--( E & l );
}
```

Lastly, C∀ does not define **zero_t** for C∀ enumeration. Users can define the boolean false for C∀ enumerations on their own, *e.g.*:

```
forall( E | CfaEnum( E ) )
int ?!=?( E lhs, zero_t ) {
    return posn( lhs ) != 0;
}
```

which effectively turns the first enumeration into a logical false and true for the others.

Chapter 6

Related Work

Enumeration-like features exist in many popular programming languages, both past and present, *e.g.*, Pascal [12], Ada [1], C♯ [6], OCaml [16] C++, Go [8], Haskell [9] (see discussion in Section 1.2.2, p. 4), Java [7], Rust [18], Swift [13], Python [17]. Among these languages, there is a large set of overlapping features, but each language has its own unique extensions and restrictions.

6.1 Pascal

Pascal introduced the **const** aliasing declaration binding a name to a constant literal/expression.

```
const Three = 2 + 1;  NULL = NIL;  PI = 3.14159;  Plus = '+';  Fred = 'Fred';
```

As stated, this mechanism is not an enumeration because there is no specific type (pseudo enumeration). Hence, there is no notion of a (possibly ordered) set. The type of each constant name (enumerator) is inferred from the constant-expression type.

Pascal introduced the enumeration type characterized by a set of ordered, unscoped identifiers (enumerators), which are not overloadable.[1]

```
type Week = ( Mon, Tue, Wed, Thu, Fri, Sat, Sun );
```

Object initialization and assignment are restricted to the enumerators of this type. Enumerators are auto-initialized from left to right, starting at zero and incrementing by 1. Enumerators *cannot* be explicitly initialized. Pascal provides a predefined type **Boolean** defined as:

```
type Boolean = ( false, true );
```

The enumeration supports the relational operators =, <>, <, <=, >=, and >, interpreted as as comparison in terms of declaration order.

The following auto-generated pseudo-functions exist for all enumeration types:

```
succ( T )   succ( Tue ) = Wed
pred( T )   pred( Tue ) =  Mon
ord( T )    ord( Tue ) = 1
```

Pascal provides *consecutive* subsetting of an enumeration using a subrange type.

[1] Pascal is *case-insensitive* so identifiers may appear in multiple forms and still be the same, *e.g.*, Mon, moN, and MON (a questionable design decision).

```
type Week = ( Mon, Tue, Wed, Thu, Fri, Sat, Sun );
     Weekday = Mon..Fri;   { subtype }
     Weekend = Sat..Sun;
var day : Week;
    wday : Weekday;
    wend : Weekend;
```

Hence, declaration order of enumerators is crucial to provide the necessary ranges. There is a bidirectional assignment between the enumeration and its subranges.

```
day := Sat;
wday := day;        { check }
wend := day;        { maybe check }
day := Mon;
wday := day;        { maybe check }
wend := day;        { check }
day := wday;        { no check }
day := wend;        { no check }
```

A static/dynamic range check should be performed to verify the values assigned to subtypes. (Free Pascal does not check and aborts in certain situations, like writing an invalid enumerator.)

An enumeration can be used in the **if** and **case** statements or iterating constructs.

```
day := Mon;
if day = wday then
    Writeln( day );
if day <= Fri then
    Writeln( 'weekday');
Mon
weekday
```

```
case day of
  Mon..Fri :
    Writeln( 'weekday');
  Sat..Sun :
    Writeln( 'weekend')
end;
weekday
```

```
while day <= Sun do begin
    Write( day, ' ' );
    day := succ( day );
end;
Mon Tue Wed Thu Fri Sat Sun
```

```
for day := Mon to Sun do begin
    Write( day, ' ' );

end;
Mon Tue Wed Thu Fri Sat Sun
```

Note that subtypes Weekday and Weekend cannot be used to define a case or loop range.

An enumeration type can be used as an array dimension and subscript.

```
Lunch : array( Week ) of Time;
for day in Week loop
    Lunch( day ) := ... ;      { set lunch time }
end loop;
```

Free Pascal [3, § 3.1.1] is a modern, object-oriented version of Pascal, with a C-style enumeration type. Enumerators can be assigned explicit values assigned in ascending numerical order using a constant expression, and the range can be non-consecutive.

```
type Count = ( Zero, One, Two, Ten = 10, Eleven );
```

Pseudo-functions pred and succ can only be used if the range is consecutive. Enumerating gives extraneous values.

```
with Ada.Text_IO; use Ada.Text_IO;
procedure test is
    type RGB is ( Red, Green, Blue );
    type Traffic_Light is ( Red, Yellow, Green );      -- overload
    procedure Red( Colour : RGB ) is begin             -- overload
        Put_Line( "Colour is " & RGB'Image( Colour ) );
    end Red;
    procedure Red( TL : Traffic_Light ) is begin       -- overload
        Put_Line( "Light is " & Traffic_Light'Image( TL ) );
    end Red;
begin
    Red( Blue );              -- RGB
    Red( Yellow );            -- Traffic_Light
    Red( RGB'(Red) );         -- ambiguous without cast
end test;
```

Figure 6.1: Ada Enumeration Overload Resolution

```
for cnt := Zero to Eleven do begin
    Write( ord( cnt ), ' ' );
end;
0 1 2 3 4 5 6 7 8 9 10 11
```

The underlying type is an implementation-defined integral type large enough to hold all enumerated values; it does not have to be the smallest possible type. The integral size can be explicitly specified using compiler directive $PACKENUM N, where N is the number of bytes, *e.g.*:

```
Type {$PACKENUM 1} SmallEnum = ( one, two, three );
     {$PACKENUM 4} LargeEnum = ( BigOne, BigTwo, BigThree );
Var S : SmallEnum; { 1 byte }
    L : LargeEnum; { 4 bytes}
```

6.2 Ada

An Ada enumeration type is a set of ordered, unscoped identifiers (enumerators) bound to *unique literals*.[2]

```
type Week is ( Mon, Tue, Wed, Thu, Fri, Sat, Sun ); -- literals (enumerators)
```

Object initialization and assignment are restricted to the enumerators of this type. While Ada enumerators are unscoped, like C, Ada enumerators are overloadable.

```
type RGB is ( Red, Green, Blue );
type Traffic_Light is ( Red, Yellow, Green );
```

Like C∀, Ada uses a type-resolution algorithm, including the left-hand side of the assignment, to disambiguate among overloaded identifiers. Figure 6.1 shows how ambiguity is handled using a cast, *e.g.*, RGB'(Red).

[2]Ada is *case-insensitive* so identifiers may appear in multiple forms and still be the same, *e.g.*, Mon, moN, and MON (a questionable design decision).

Enumerators without initialization are auto-initialized from left to right, starting at zero and incrementing by 1. Enumerators with initialization must set *all* enumerators in *ascending* order, *i.e.*, there is no auto-initialization.

```
type Week is ( Mon, Tue, Wed, Thu, Fri, Sat, Sun );
    for Week use ( Mon => 0, Tue => 1, Wed => 2, Thu => 10, Fri => 11, Sat => 14, Sun => 15 );
```

The enumeration operators are the equality and relational operators, =, /=, <, <=, =, /=, >=, >, where the ordering relationship is given implicitly by the sequence of ascending enumerators.

Ada provides an alias mechanism, **renames**, for aliasing types, which is useful to shorten package identifiers.

```
OtherRed : RGB renames Red;
```

which suggests a possible C∀ extension to **typedef**.

```
typedef RGB.Red OtherRed;
```

There are three pairs of inverse enumeration pseudo-functions (attributes): 'Pos and 'Val, 'Enum_Rep and 'Enum_Val, and 'Image and 'Value,

```
RGB'Pos( Red ) = 0;             RGB'Val( 0 ) = Red
RGB'Enum_Rep( Red ) = 10;       RGB'Enum_Val( 10 ) = Red
RGB'Image( Red ) = "RED";       RGB'Value( "Red" ) = Red
```

These attributes are important for IO. An enumeration type T also has the following attributes: T'First, T'Last, T'Range, T'Pred, T'Succ, T'Min, and T'Max, producing an intuitive result based on the attribute name.

Ada allows the enumerator label to be a character constant.

```
type Operator is ( '+', '-', '*', '/' );
```

which is syntactic sugar for the label and not character literals from the predefined type Character. The purpose is strictly readability using character literals rather than identifiers.

```
Op : Operator := '+';
if Op = '+' or else Op = '-' then ... ;
elsif Op = '*' or else Op = '/' then ... ; end if;
```

Interestingly, arrays of character enumerators can be treated as strings.

```
Ops : array( 0..3 ) of Operator;
Ops := "+-*/";          -- string assignment to array elements
Ops := "+-" & "*/";     -- string concatenation and assignment
```

Ada's Character type is defined as a character enumeration across all Latin-1 characters.

Ada's boolean type is also a special enumeration, which can be used in conditions.

```
type Boolean is (False, True); -- False / True not keywords
Flag : Boolean;
if Flag then ...    -- conditional
```

Since only types derived from Boolean can be conditional, Boolean is essentially a builtin type.

Ada provides *consecutive* subsetting of an enumeration using **range**.

```
type Week is ( Mon, Tue, Wed, Thu, Fri, Sat, Sun );
subtype Weekday is Week range Mon .. Fri;
```

```
subtype Weekend is Week range Sat .. Sun;
Day : Week;
```

Hence, the ordering of the enumerators is crucial to provide the necessary ranges.

An enumeration type can be used in the Ada **case** (all enumerators must appear or a **default**) or iterating constructs.

```
case Day is
    when Mon .. Fri => ... ;
    when Sat .. Sun => ... ;
end case;
```

```
case Day is
    when Weekday => ... ;  -- subtype ranges
    when Weekend => ... ;
end case;
```

```
for Day in Mon .. Sun loop
    ...
end loop;
```

```
for Day in Weekday loop
    ...
end loop;
```

```
for Day in Weekend loop
    ...
end loop;
```

An enumeration type can be used as an array dimension and subscript.

```
Lunch : array( Week ) of Time;
for Day in Week loop
    Lunch( Day ) := ... ;      -- set lunch time
end loop;
```

6.3 C++

C++ enumeration is largely backward compatible with C, so it inherited C's enumerations with some modifications and additions.

C++ has aliasing using **const** declarations, like C (see Section 2.1.1, p. 8), with type inferencing, plus static/dynamic initialization. (Note, a C++ constexpr declaration is the same as **const** with the restriction that the initialization is a compile-time expression.)

```
const auto one = 0 + 1;                    // static initialization
const auto NIL = nullptr;
const auto PI = 3.14159;
const auto Plus = '+';
const auto Fred = "Fred";
const auto Mon = 0, Tue = Mon + 1, Wed = Tue + 1, Thu = Wed + 1, Fri = Thu + 1,
             Sat = Fri + 1, Sun = Sat + 1;
void foo() {
    const auto r = random();               // dynamic initialization
    int va[r];                             // VLA, auto scope only
}
```

Statically initialized identifiers may appear in any constant-expression context, *e.g.*, **case**. Dynamically initialized identifiers may appear as array dimensions in g++, which allows variable-sized arrays. Interestingly, global C++ **const** declarations are implicitly marked **static** (r, read-only local, rather than R, read-only external)

```
$ nm test.o
0000000000000018 r Mon
```

whereas C **const** declarations without **static** are marked R. This difference results from linking concerns that come from templates.

The following C++ non-backward compatible change is made, plus the safe-assignment change shown in Section 3.3, p. 20.

Change: In C++, the type of an enumerator is its enumeration. In C, the type of an enumerator is **int**. Example:

```
enum e { A };
sizeof(A) == sizeof(int)          // in C
sizeof(A) == sizeof(e)            // in C++
/* and sizeof(int) is not necessary equal to sizeof(e) */
```

Rationale: In C++, an enumeration is a distinct type.
Effect on original feature: Change to semantics of well-defined feature.
Difficulty of converting: Semantic transformation.
How widely used: Seldom. The only time this affects existing C code is when the size of an enumerator is taken. Taking the size of an enumerator is not a common C coding practice.

ISO/IEC 14882:1998 (C++ Programming Language Standard) [10, C.1.5.7.2.6]

Hence, the values in a C++ enumeration can only be its enumerators (without a cast).

While the storage size of an enumerator is up to the compiler, there is still an implicit cast to **int**.

```
enum E { A, B, C };
E e = A;
int i = A;   i = e;                // implicit casts to int
```

C++11 added a scoped enumeration, **enum class** (or **enum struct**)[3], where the enumerators are accessed using type qualification.

```
enum class E { A, B, C };
E e = E::A;                        // qualified enumerator
e = B;                             // error: B not in scope
```

C++20 supports explicit unscoping with a **using enum** declaration.

```
enum class E { A, B, C };
using enum E;
E e = A;   e = B;                  // direct access
```

C++11 added the ability to explicitly declare an underlying *integral* type for **enum class**.

```
enum class RGB : long { Red, Green, Blue };
enum class rgb : char { Red = 'r', Green = 'g', Blue = 'b' };
enum class srgb : signed char { Red = -1, Green = 0, Blue = 1 };
```

There is no implicit conversion from the **enum class** type to its declared type.

```
rgb crgb = rgb::Red;
char ch = rgb::Red;   ch = crgb;   // error
```

An enumeration can be used in the **if** and **switch** statements.

[3]The use of keyword **class** is reasonable because default visibility is **private** (scoped). However, default visibility for **struct** is **public** (unscoped) making it an odd choice.

```
if ( day <= Fri )
    cout << "weekday" << endl;
```

```
switch ( day ) {
  case Mon: case Tue: case Wed: case Thu: case Fri:
    cout << "weekday" << endl; break;
  case Sat: case Sun:
    cout << "weekend" << endl; break;
}
```

However, there is no mechanism to iterate through an enumeration. A common workaround is to iterate over enumerator as integral values, but it only works if enumerators resemble a sequence of natural, i.e., enumerators are auto-initialized. Otherwise, the iteration would have integers that are not enumeration values.

```
enum Week { Mon, Tue, Wed, Thu = 10, Fri, Sat, Sun };
for ( Week d = Mon; d <= Sun; d = (Week)(d + 1) ) cout << d << ' ';
0 1 2 3 4 5 6 7 8 9 10 11 12 13
```

As a consequence, there is no meaningful enumerating mechanism.

An enumeration type cannot declare an array dimension but an enumerator can be used as a subscript. There is no mechanism to subset or inherit from an enumeration.

6.4 C♯

C♯ is a programming language with a scoped, integral enumeration similar to C++ **enum class**.

```
enum Week : long { Mon, Tue, Wed, Thu = 10, Fri, Sat, Sun }
enum RGB { Red, Green, Blue }
```

The default underlying integral type is **int**, with auto-incrementing and implicit/explicit initialization. A method cannot be defined in an enumeration type (extension methods are possible). There is an explicit bidirectional conversion between an enumeration and its integral type, and an implicit conversion to the enumerator label in display contexts.

```
int iday = (int)Week.Fri;                   // day == 11
Week day = (Week)42;                        // day == 42, unsafe
string mon = Week.Mon.ToString();           // mon == "Mon"
RGB rgb = RGB.Red;                          // rgb == "Red"
day = (Week)rgb;                            // day == "Mon", unsafe
Console.WriteLine( Week.Fri );              // print label Fri
```

Like C++, C♯ defines enumeration relational and arithmetic operators in terms of value. Enumerators have no defined positional meaning.

```
day = day++ - 5;                            // value manipulation
day = day & day;
for ( Week d = Mon; d <= Sun; d += 1 ) {
    Console.Write( d + "  " );
}
Mon Tue Wed 3 4 5 6 7 8 9 Thu Fri Sat Sun
```

As a consequence, there is no direct meaningful enumerating mechanism.

An enumeration can be used in the **if** and **switch** statements.

```
if ( day <= Week.Fri )
    Console.WriteLine( "weekday" );
```

```
switch ( day ) {
  case Week.Mon: case Week.Tue: case Week.Wed:
  case Week.Thu: case Week.Fri:
    Console.WriteLine( "weekday" ); break;
  case Week.Sat: case Week.Sun:
    Console.WriteLine( "weekend" ); break;
}
```

To indirectly enumerate, C♯'s Enum library provides Enum.GetValues, a static member of abstract Enum type that return a reference to an array of all enumeration constants. Internally, an Enum type has a static member called fieldInfoHash – a Hashtable that stores information about enumerators. The field is populated on-demand: it only contains information if a reflection like GetValues is called. As an optimization, this information is cached, so the cost of reflection is paid once throughout the lifetime of a program. GetValues then converts a Hashtable to an Array, which supports enumerating.

```
foreach ( Week d in Enum.GetValues( typeof(Week) ) ) {
    Console.WriteLine( d + "   " + (int)d + "   " ); // label, position
}
Mon 0, Tue 1, Wed 2, Thu 10, Fri 11, Sat 12, Sun 13,
```

Hence, enumerating is not supplied directly by the enumeration, but indirectly through the expensive $O(N)$ creation of an enumerable array type, and recreating this array for each enumerating, versus direct arithmetic.

An enumeration type cannot declare an array dimension but an enumerator can be used as a subscript. There is no mechanism to subset or inherit from an enumeration.

The Flags attribute creates a bit-flags enumeration, making bitwise operators &, |, ~ (complement), ^ (xor) sensible.

```
[Flags] public enum Week {
    None = 0x0, Mon = 0x1, Tue = 0x2, Wed = 0x4,
    Thu = 0x8, Fri = 0x10, Sat = 0x20, Sun = 0x40,
    Weekdays = Mon | Tue | Wed | Thu | Fri  // Weekdays == 0x1f
    Weekend = Sat | Sun,                   // Weekend == 0x60
}
Week meetings = Week.Mon | Week.Wed;   // 0x5
```

6.5 Go

Go has **const** aliasing declarations, similar to C++ (see Section 6.3, p. 45), for basic types with type inferencing and static initialization (constant expression). The most basic form of constant definition is a **const** keyword, followed by the name of constant, an optional type declaration of the constant, and a mandatory initialize. For example:

```
const R int = 0;  const G uint = 1;  const B = 2; // explicit typing and type inferencing
const Fred = "Fred";  const Mary = "Mary";  const Jane = "Jane";
const S = 0;  const T = 0;
const USA = "USA";  const U = "USA";
const V = 3.1;  const W = 3.1;
```

These declarations defined immutable and unscoped variables, and Go has no naming overloading. If no type declaration is provided, Go infers type from the initializer expression.

These named constants can be grouped together in one **const** declaration block to introduce a form of auto-initialization.

```
const ( R = 0; G; B )                          // implicit initialization: 0 0 0
const ( Fred = "Fred"; Mary = "Mary"; Jane = "Jane" ) // explicit initialization: Fred Mary Jane
const ( S = 0; T; USA = "USA"; U; V = 3.1; W ) // implicit/explicit: 0 0 USA USA 3.1 3.1
```

The first identifier *must* be explicitly initialized; subsequent identifiers can be implicitly or explicitly initialized. Implicit initialization always uses the *previous* (predecessor) constant expression initializer. A constant block can still use explicit declarations, and the following constants inherit that type.

```
type BigInt int64
const ( R BigInt = 0; G; B )
const ( Fred string = "Fred"; Mary = "Mary"; Jane = "Jane" )
const ( S int = 0; T; USA string = "USA"; U; V float32 = 3.1; W )
```

Typing the first constant and implicit initializing is still not an enumeration because there is no unique type for the constant block; nothing stops other constant blocks from being of the same type.

Each **const** declaration provides an implicit *compile-time* integer counter starting at 0, called **iota**, which is post-incremented after each constant declaration.

```
const ( R = iota; G; B )                       // implicit: 0 1 2
const ( C = iota + B + 1; G; Y )               // implicit: 3 4 5
```

which are equivalent to:

```
const ( R = iota; G = iota; B = iota )         // implicit: 0 1 2
const ( C = iota + B + 1; G = iota + B + 1; Y = iota + B + 1 ) // implicit: 3 4 5
```

An underscore **const** identifier advances **iota**.

```
const ( O1 = iota + 1; _; O3; _; O5 )          // 1, 3, 5
```

Auto-initialization reverts from **iota** to the previous value after an explicit initialization, but auto-incrementing of **iota** continues.

```
const ( Mon = iota; Tue; Wed;                  // 0, 1, 2
        Thu = 10; Fri; Sat; Sun = iota )       // 10, 10, 10, 6
```

Auto-initialization from **iota** is restarted and **iota** reinitialized with an expression containing at most *one* **iota**.

```
const ( V1 = iota; V2; V3 = 7; V4 = iota + 1; V5 ) // 0 1 7 4 5
const ( Mon = iota; Tue; Wed; // 0, 1, 2
        Thu = 10; Fri = iota - Wed + Thu - 1; Sat; Sun ) // 10, 11, 12, 13
```

Here, V4 and Fri restart auto-incrementing from **iota** and reset **iota** to 4 and 11, respectively, because of the initialization expressions containing **iota**. Note, because **iota** is incremented for an explicitly initialized identifier or _, at Fri **iota** is 4 requiring the minus one to compute the value for Fri.

Basic switch and looping are possible.

```
day := Mon;   // := ⇒ type inferencing
switch day {
  case Mon, Tue, Wed, Thu, Fri:
    fmt.Println( "weekday" );
  case Sat, Sun:
    fmt.Println( "weekend" );
}
```

```
for i := Mon; i <= Sun; i += 1 {
    fmt.Println( i )
}
```

However, the loop in this example prints the values from 0 to 13 because there is no actual enumeration.

A constant variable can be used as an array dimension or a subscript.

```
var ar[Sun] int
ar[Mon] = 3
```

6.6 Java

Java provides an enumeration using a specialized class. A basic Java enumeration is an opaque enumeration, where the enumerators are constants.

```
enum Week { Mon, Tue, Wed, Thu, Fri, Sat, Sun; }
Week day = Week.Sat;
```

The enumerator's members are scoped, requiring qualification. The value of an enumeration instance is restricted to its enumerators.

The position (ordinal) and label (name) are accessible but there is no value property.

```
System.out.println( day.ordinal() + "  " + day + "  " + day.name() );
5 Sat Sat
```

Since day has no value, it prints its label (name). The member valueOf is the inverse of name converting a string to an enumerator.

```
day = Week.valueOf( "Wed" );
```

Extra members can be added to provide specialized operations.

```
public boolean isWeekday() { return ordinal() <= Fri.ordinal(); }
public boolean isWeekend() { return Sat.ordinal() <= ordinal(); }
```

Notice the unqualified calls to ordinal in the members implying a **this** to some implicit implementation variable, likely an **int**.

Enumerator values require an enumeration type (any Java type may be used) and implementation member.

```
enum Week {
    Mon(1), Tue(2), Wed(3), Thu(4), Fri(5), Sat(6), Sun(7); // must appear first
    private long day;                   // enumeration type and implementation member
    private Week( long d ) { day = d; }  // enumerator initialization
};
Week day = Week.Sat;
```

The position, value, and label are accessible.

```
System.out.println( day.ordinal() + "   " + day.day + "   " + day.name() );
5 6 Sat
```

If the implementation member is **public**, the enumeration is unsafe, as any value of the underlying type can be assigned to it, *e.g.*, day = 42. The implementation constructor must be private since it is only used internally to initialize the enumerators. Initialization occurs at the enumeration-type declaration.

Enumerations can be used in the **if** and **switch** statements but only for equality tests.

```
if ( day == Week.Fri )
    System.out.println( "Fri" );
```

```
switch ( day ) {
  case Mon: case Tue: case Wed: case Thu: case Fri:
    System.out.println( "weekday" );  break;
  case Sat: case Sun:
    System.out.println( "weekend" );  break;
}
```

Notice enumerators in the **switch** statement do not require qualification.

There are no arithmetic operations on enumerations, so there is no arithmetic way to iterate through an enumeration without making the implementation type **public**. Like C♯, enumerating is supplied indirectly through another enumerable type, not via the enumeration. Specifically, Java supplies a static method values, which returns an array of enumerator values. Unfortunately, values is an expensive O(n) operation, which is recreated each time it is called.

```
for ( Week d : Week.values() ) {
    System.out.print( d.ordinal() + d.day + "   " +  d.name() + ",     " );
}
0 1 Mon,  1 2 Tue,  2 3 Wed,  3 4 Thu,  4 5 Fri,  5 6 Sat,  6 7 Sun,
```

Java provides EnumSet, an auxiliary data structure that takes an enum class as parameter (Week.class) for its construction, and it contains members only with the supplied enum type. EnumSet is enumerable because it extends AbstractSet interfaces and thus supports direct enumerating via forEach. It also has subset operation range and it is possible to add to and remove from members of the set. EnumSet supports more enumeration features, but it is not an enumeration type; it is a set of enumerators from a pre-defined enum.

An enumeration type cannot declare an array dimension nor can an enumerator be used as a subscript. Enumeration inheritence is disallowed because an enumeration is **final**.

6.7 Rust

Rust **enum** provides two largely independent mechanisms from a single language feature: an ADT and an enumeration. When **enum** is an ADT, pattern matching is used to discriminate among the variant types.

```
struct S {
    i : isize,  j : isize
}
let mut s = S{ i : 3, j : 4 };
enum ADT {
    I( isize ),      // int
    F( f64 ),        // float
    S( S ),          // struct
}
```

```
let mut adt : ADT;
adt = ADT::I(3);  println!( "{:?}", adt );
adt = ADT::F(3.5);  println!( "{:?}", adt );
adt = ADT::S(s);  println!( "{:?}", adt );
match adt {
    ADT::I( i ) => println!( "{:}", i ),
    ADT::F( f ) => println!( "{:}", f ),
    ADT::S( s ) => println!( "{:}  {:}", s.i, s.j ),
}
```

```
I(3)
F(3.5)
S(S { i: 3, j: 4 })
3 4
```

Even when the variant types are the unit type, the ADT is still not an enumeration because there is no enumerating (see Section 1.2.2, p. 4).

```
enum Week { Mon, Tues, Wed, Thu, Fri, Sat, Sun, } // terminating comma
let mut week : Week = Week::Mon;
match week {
    Week::Mon => println!( "Mon" ),
    ...
    Week::Sun => println!( "Sun" ),
}
```

However, Rust allows direct setting of the ADT constructor, which means it is actually a tag.

```
enum Week {
    Mon, Tues, Wed, // start 0
    Thu = 10, Fri,
    Sat, Sun,
}
```

```
#[repr(u8)]
enum ADT {
    I(isize) = 5,
    F(f64) = 10,
    S(S) = 0,
}
```

Through this integral tag, it is possible to enumerate, and when all tags represent the unit type, it behaves like C++ **enum class**. When tags represent non-unit types, Rust largely precludes accessing the tag because the semantics become meaningless. Hence, the two mechanisms are largely disjoint, and only the enumeration component is discussed.

In detail, the **enum** type has an implicit integer tag (discriminant) with a unique value for each variant type. Direct initialization is achieved by a compile-time expression that generates a constant value. Indirect initialization (without initialization, Fri/Sun) is auto-initialized: from left to right, starting at zero or the next explicitly initialized constant, incrementing by 1. There is an explicit cast from the tag to integer.

```
let mut mon : isize = Week::Mon as isize;
```

An enumeration can be used in the **if** and **match** (**switch**) statements.

```
if week as isize == Week::Mon as isize {
    println!( "{:?}", week );
}
```

```
match week {
    Week::Mon | Week:: Tue | Week::Wed | Week::Thu
        | Week::Fri ⇒ println!( "weekday" ),
    Week::Sat | Week:: Sun => println!( "weekend" ),
}
```

Like C/C++, there is no mechanism to iterate through an enumeration. It can only be approximated by a loop over a range of enumerators and only works if the enumerator values are a sequence of natural numbers.

```
for d in Week::Mon as isize ..= Week::Sun as isize {
    print!( "{:?}  ", d );
}
0 1 2 3 4 5 6 7 8 9 10 11 12 13
```

There is no direct way to harmonize an enumeration and another data structure. For example, there is no mapping from an enumerated type to an array type. In terms of extensibility, there is no mechanism to subset or inherit from an enumeration.

6.8 Swift

Despite being named as enumeration, a Swift **enum** is in fact a ADT: cases (enumerators) of an **enum** can have heterogeneous types and be recursive. When **enum** is an ADT, pattern matching is used to discriminate among the variant types.

```
struct S {
    var i : Int,  j : Int
}
var s = S( i : 3, j : 5 )
enum ADT {
    case I(Int)      // int
    case F(Float)  // float
    case S(S)       // struct
}
```

```
var adt : ADT
adt = .I( 3 );  print( adt )
adt = .F( 3.5 );  print( adt )
adt = .S( s );  print( adt )
switch adt {  // pattern matching
    case .I(let i):  print( i )
    case .F(let f):  print( f )
    case .S(let s):  print( s.i, s.j )
}
```

```
I(3)
F(3.5)
S(S(i: 3, j: 5))
3 5
```

Note, after an adt's type is known, the enumerator is inferred without qualification, *e.g.*, .I(3).

Without type declaration for enumeration cases, the enumerators have unit-type, which is like a scoped, opaque enumeration.

```
enum Week { case Mon, Tue, Wed, Thu, Fri, Sat, Sun }; // unit−type
var week : Week = Week.Mon;
```

As well, it is possible to type associated values of enumeration cases with a common type. When enumeration cases are typed with a common integral type, Swift auto-initializes enumeration cases following the same initialization scheme as C language. If an enumeration is typed with string, its cases are auto-initialized to case names (labels).

```
enum WeekInt: Int {
    case Mon, Tue, Wed, Thu = 10, Fri,
         Sat = 4, Sun // auto−incrementing
};
```

```
enum WeekStr: String {
    case Mon = "MON", Tue, Wed, Thu, Fri,
         Sat = "SAT", Sun
};
```

An enumeration only supports equality comparison between enumerator values, unless it inherits from Comparable, adding relational operators <, <=, >, and >=.

An enumeration can have methods.

```
enum Week: Comparable {
    case Mon, Tue, Wed, Thu, Fri, Sat, Sun // unit−type
    func isWeekday() −> Bool { return self <= .Fri }  // methods
    func isWeekend() −> Bool { return .Sat <= self }
};
```

An enumeration can be used in the **if** and **switch** statements, where **switch** must be exhaustive or have a **default**.

```
if week <= .Fri {
    print( "weekday" );
}
```

```
switch week {
    case .Mon: print( "Mon" )
    ...
    case .Sun: print( "Sun" )
}
```

Enumerating is accomplished by inheriting from CaseIterable protocol, which has a static **enum**.allCases property that returns a collection of all the cases for looping over an enumeration type or variable. Like C∀, Swift's default enumerator output is the case name (label). An enumerator of a typed enumeration has an attribute rawValue that return its case value.

```
enum Week: Comparable, CaseIterable {
    case Mon, Tue, Wed, Thu, Fri, Sat, Sun // unit-type
};
for day in Week.allCases {
    print( day, terminator:"  " )
}
Mon Tue Wed Thu Fri Sat Sun
```

```
enum WeekInt: Int, CaseIterable {
    case Mon, Tue, Wed, Thu = 10, Fri,
         Sat = 4, Sun // auto-incrementing
};
for day in WeekInt.allCases {
    print( day.rawValue, terminator:"  " )
}
0 1 2 10 11 4 5
```

```
enum WeekStr: String, CaseIterable {
    case Mon = "MON", Tue, Wed, Thu, Fri,
         Sat = "SAT", Sun
};
for day in WeekStr.allCases {
    print( day.rawValue, terminator:"  " )
}
MON Tue Wed Thu Fri SAT Sun
```

There is a safe bidirectional conversion from typed enumerator to rawValue and vice versa.

```
if let opt = WeekInt( rawValue: 0 ) {  // test optional return value
    print( opt.rawValue, opt )  // 0 Mon
} else {
    print( "invalid weekday lookup" )
}
```

In the previous example, the initialization of opt fails if there is no enumeration value equal to 0, resulting in a nil value. Initialization from a raw value is considered an expensive operation because it requires a value lookup.

6.9 Python 3.13

Python is a dynamically-typed reflexive programming language with multiple incompatible versions. The generality of the language makes it possible to extend existing or build new language features. As a result, discussing Python enumerations is a moving target, because if a feature does not exist, it can often be created with varying levels of complexity within the language. Therefore, the following discussion is (mostly) restricted to the core enumeration features in Python 3.13.

A Python enumeration is not a basic type; it is a class inheriting from the Enum class. The Enum class presents a set of scoped enumerators, where each enumerator is a pair object with a *constant* string name and an arbitrary value. Hence, an enumeration instance is a fixed type (enumeration pair), and its value is the type of one of the enumerator pairs.

The enumerator value fields must be explicitly initialized and be *unique*.

```
class Week(Enum): Mon = 1; Tue = 2; Wed = 3; Thu = 4; Fri = 5; Sat = 6; Sun = 7
```

and/or explicitly auto-initialized with **auto** method, *e.g.*:

```
class Week(Enum): Mon = 1; Tue = 2; Wed = 3; Thu = 10; Fri = auto(); Sat = 4; Sun = auto()
Mon : 1 Tue : 2 Wed : 3 Thu : 10 Fri : 11 Sat : 4 Sun : 12
```

auto is controlled by member _generate_next_value_(), which by default returns one plus the highest value among enumerators, and can be overridden:

```
@staticmethod
def _generate_next_value_( name, start, count, last_values ):
    return name
```

There is no direct concept of restricting the enumerators in an enumeration *instance* because dynamic typing changes the type.

```
class RGB(Enum): Red = 1; Green = 2; Blue = 3
day : Week = Week.Tue;                  # type is Week
day = RGB.Red                           # type is RGB
day : Week = RGB.Red                    # type is RGB
```

The enumerators are constants and cannot be reassigned. Hence, while enumerators can be different types,

```
class Diff(Enum): Int = 1; Float = 3.5; Str = "ABC"
```

it is not an ADT because the enumerator names are not constructors.

An enumerator initialized with the same value is an alias and invisible at the enumeration level, *i.e.*, the alias is substituted for its aliases.

```
class WeekD(Enum): Mon = 1; Tue = 2; Wed = 3; Thu = 10; Fri = 10; Sat = 10; Sun = 10
```

Here, the enumeration has only 4 enumerators and 3 aliases. An alias is only visible by dropping down to the class level and asking for class members. Aliasing is prevented using the unique decorator.

```
@unique
class DupVal(Enum): One = 1; Two = 2; Three = 3; Four = 3
ValueError: duplicate values found in <enum 'DupVal'>: Four -> Three
```

There are bidirectional enumeration pseudo-functions for label and value, but there is no concept of access using ordering (position).[4]

[4]There is an $O(N)$ mechanism to access an enumerator's value by position.

```
def by_position(enum_type, position):
    for index, value in enumerate(enum_type):
        if position == index: return value
    raise Exception("by_position out of range")
```

```
Week.Thu.value == 4;            Week( 4 ) == Week.Thu
Week.Thu.name == "Thu";         Week["Thu"].value == 4
```

Enum only supports equality comparison between enumerator values. There are multiple library extensions to Enum, *e.g.*, OrderedEnum recipe class, adding relational operators <, <=, >, and >=.

An enumeration **class** can have methods.

```
class Week(OrderedEnum):
    Mon = 1; Tue = 2; Wed = 3; Thu = 4; Fri = 5; Sat = 6; Sun = 7
    def isWeekday(self):        # methods
        return Week(self.value) <= Week.Fri
    def isWeekend(self):
        return Week.Sat <= Week(self.value)
```

An enumeration can be used in the **if** and **switch** statements but only for equality tests, unless extended to OrderedEnum.

```
if day <= Week.Fri :            match day:
    print( "weekday" );             case Week.Mon | Week.Tue | Week.Wed | Week.Thu | Week.Fri:
                                        print( "weekday" );
                                    case Week.Sat | Week.Sun:
                                        print( "weekend" );
```

Looping is performed using the enumeration type or islice from itertools based on position.

```
for day in Week:                # Mon : 1 Tue : 2 Wed : 3 Thu : 4 Fri : 5 Sat : 6 Sun : 7
    print( day.name, " :", day.value, end="  " )
for day in islice(Week, 0, 5):  # Mon : 1 Tue : 2 Wed : 3 Thu : 4 Fri : 5
    print( day.name, " :", day.value, end="  " )
for day in islice(Week, 5, 7):  # Sat : 6 Sun : 7
    print( day.name, " :", day.value, end="  " )
for day in islice(Week,0, 7, 2): # Mon : 1 Wed : 3 Fri : 5 Sun : 7
    print( day.name, " :", day.value, end="  " )
```

Iterating that includes alias names only (strings) is done using attribute __members__.

```
for day in WeekD.__members__:
    print( day, " :", end="  " )
Mon : Tue : Wed : Thu : Fri : Sat : Sun
```

Enumeration subclassing is allowed only if the enumeration base-class does not define any members.

```
class WeekE(OrderedEnum): pass;  # no members
class WeekDay(WeekE): Mon = 1; Tue = 2; Wed = 3; Thu = 4; Fri = 5;
class WeekEnd(WeekE): Sat = 6; Sun = 7
```

Here, type WeekE is an abstract type because dynamic typing never uses it.

```
print( type(WeekE) )                        <class 'enum.EnumType'>
day : WeekE = WeekDay.Fri    # set type
print( type(day), day )                     <enum 'WeekDay'> WeekDay.Fri
day = WeekEnd.Sat            # set type
print( type(day), day )                     <enum 'WeekEnd'> WeekEnd.Sat
```

There are a number of supplied enumeration base-types: IntEnum, StrEnum, IntFalg, Flag,

which restrict the values in an enum using multi-inheritance. IntEnum is a subclass of **int** and Enum, allowing enumerator comparison to **int** and other enumerators of this type (like C enumerators). StrEnum is the same as IntEnum but a subclass of the string type **str**. IntFlag, is a restricted subclass of **int** where the enumerators can be combined using the bitwise operators (&, |, ^, ~) and the result is an IntFlag member. Flag is the same as IntFlag but cannot be combined with, nor compared against, any other Flag enumeration, nor **int**. Auto increment for IntFlag and Flag is by powers of 2. Enumerators that are combinations of single-bit enumerators are aliases and, hence, invisible. The following is an example for Flag.

```
class WeekF(Flag): Mon = 1; Tue = 2; Wed = 4; Thu = auto(); Fri = 16; Sat = 32; Sun = 64; \
    Weekday = Mon | Tue | Wed | Thu | Fri; \
    Weekend = Sat | Sun
print( f"0x{repr(WeekF.Weekday.value)} 0x{repr(WeekF.Weekend.value)}" )
0x31 0x96
```

It is possible to enumerate through a Flag enumerator (no aliases):

```
for day in WeekF:
    print( f"{day.name}: {day.value}", end="  ")
Mon: 1 Tue: 2 Wed: 4 Thu: 8 Fri: 16 Sat: 32 Sun: 64
```

and a combined alias enumerator for Flag.

```
weekday = WeekF.Weekday
for day in weekday:
    print( f"{day.name}:"
        f" {day.value}", end="  " )
Mon: 1 Tue: 2 Wed: 4 Thu: 8 Fri: 16
```

```
weekend = WeekF.Weekend
for day in weekend:
    print( f"{day.name}:"
        f" {day.value}", end="  " )
Sat: 32 Sun: 64
```

6.10 OCaml

Like Swift (Section 6.8, p. 53) and Haskell (Section 1.2.2, p. 4), OCaml **enum** provides two largely independent mechanisms from a single language feature: an ADT and an enumeration. When **enum** is an ADT, pattern matching is used to discriminate among the variant types.

```
type s = { i : int; j : int }
let sv : s = { i = 3; j = 5 }
type adt =
    I of int |      // int
    F of float |    // float
    S of s          // struct
```

```
let adtprt( adtv : adt ) =
    match adtv with (* pattern matching *)
        I i -> printf "%d\n" i |
        F f -> printf "%g\n" f |
        S sv -> printf "%d %d\n" sv.i sv.j
let adtv : adt = I(3)     let _ = adtprt( adtv )
let adtv : adt = F(3.5)   let _ = adtprt( adtv )
let adtv : adt = S(sv)    let _ = adtprt( adtv )
```

```
3
3.5
3 5
```

The type names are independent of the type value and mapped to an opaque, ascending, integral tag, starting from 0, supporting relational operators <, <=, >, and >=.

```
let silly( adtv : adt ) =
    if adtv <= F(3.5) then
        printf "<= F\n"
    else if adtv >= S(sv) then
        printf ">= S\n"
```

```
let adtv : adt = I(3)     let _ = silly( adtv )     <= F
let adtv : adt = F(3.5)   let _ = silly( adtv )     <= F
let adtv : adt = S(sv)    let _ = silly( adtv )     >= S
```

In the example, type values must be specified (any appropriate values work) but ignored in the relational comparison of the type tag.

An enumeration is created when *all* the enumerators are unit-type, which is like a scoped, opaque enumeration, where only the type tag is used.

```
type week = Mon | Tue | Wed | Thu | Fri | Sat | Sun
let day : week = Mon
```

Since the type names are opaque, a type-tag value cannot be explicitly set nor can it have a type other than integral.

As seen, a type tag can be used in the **if** and **match** statements, where **match** must be exhaustive or have a default case.

While OCaml enumerators have an ordering following the definition order, they are not enumerable. To iterate over all enumerators, an OCaml type needs to derive from the enumerate PPX (Pre-Preocessor eXtension), which appends a list of all enumerators to the program abstract syntax tree (AST). However, as stated in the documentation, enumerate PPX does not guarantee the order of the list. PPX is beyond the scope of OCaml native language and it is a preprocessor directly modifying a parsed AST. In conclusion, there is no enumerating mechanism within the scope of OCaml language.

New types can be formed as a composition of existing types.

```
type weekday = Mon | Tue | Wed | Thu | Fri
type weekend = Sat | Sun
type week = Weekday of weekday | Weekend of weekend
let day : week = Weekend Sun
```

The type week is the sum of weekday and weekend, *i.e.*, week has all the enumerators from the set weekday and weekend. The sum type construction resembles containment inheritance from non-functional programming discipline, with the sum type being a wrapper class that contains one of its parent types. The wrapper is unwrapped with pattern matching.

```
type weekday = Mon | Tue | Wed | Thu | Fri
type weekend = Sat | Sun
type week = Weekday of weekday |
            Weekend of weekend
let wd : weekday = Mon
let _ = match wd with
    Mon -> printf "Mon " | _ -> ()
let we : weekend = Sun
let _ = match we with
    Sun -> printf "Sun " | _ -> ()
let day : week = Weekend Sun
let _ = match day with
    Weekend Sun -> printf "Sun\n" | _ -> ()
```

```
enum() weekday { Mon, Tue, Wed, Thu, Fri };
enum() weekend { Sat, Sun };
enum() week { inline weekday, inline weekend };
int main() {
    weekday wd = Mon;

    printf( "%s ", label( wd ) );
    weekend we = Sun;

    printf( "%s ", label( we ) );
    week day = Sun;

    printf( "%s\n", label( day ) );
}
```

```
Mon Sun Sun
```

Table 6.1: Enumeration Feature / Language Comparison

	Pascal	Ada	C♯	OCaml	Java	Golang	Rust	Swift	Python	C	C++	C∀
enum	Dialect	✓	✓	ADT	✓	**const**	ADT/✓	ADT/✓	✓	✓	✓	✓
opaque	✓			✓	✓		✓	✓				✓
typed	Int	Int	Int	H	U	H	U/H	U/H	H	Int	Int	U
safety	✓	✓		✓	✓		✓	✓			✓	✓
posn ordered	Implied	Implied		✓								✓
unique values	✓	✓		✓				✓				
auto-init	✓	all or none	✓	N/A		✓	✓	✓	✓	✓	✓	✓
(Un)Scoped	U	U	S	S	S	U	S	S	S	U	U/S	U/S
overload		✓										✓
loop	✓	✓							✓			✓
arr. dim.	✓	✓										✓
subset	✓	✓										✓
superset				✓			✓	✓				✓

6.11 Comparison

Table 6.1 shows a comparison of enumeration features and programming languages with the explaination of categories below. The features are high-level and may not capture nuances within a particular language.

1. opaque: an enumerator cannot be used as its underlying representation or implemented in terms of an ADT.
2. typed: H ⇒ heterogeneous, *i.e.*, enumerator values may be different types.
   U ⇒ homogenous, *i.e.*, enumerator values have the same type.
3. safety: An enumeration variable can only hold a value from its defined enumerators.
4. posn ordered: enumerators have defined ordering based on enumerator declaration order. Position ordered is implied if the enumerator values must be strictly increasingly.
5. unique value: enumerators must have a unique value.
6. auto-init: Values are auto-initializable by language specification.
   It is not appliable to OCaml because OCaml enumeration has unit type.
7. (Un)Scoped: U ⇒ enumerators are projected into the containing scope. S ⇒ enumerators are contained in the enumeration scope and require qualification.
8. overload: An enumerator label can be used without type qualification in a context where multiple enumerations have defined the label.
9. loop: Enumerate without the need to convert an enumeration to another data structure.
10. arr. dim: An enumeration can be used directly as an array dimension, and enumerators can be mapped to an array element (not a conversion to integer type).
11. subset: Name a subset of enumerators as a new type.
12. superset: Create a new enumeration that contains all enumerators from pre-defined enumerations.

Chapter 7

Conclusion

This work aims to extend the simple and unsafe enumeration type in the C programming language into a complex and safe enumeration type in the C∀ programming language while maintaining backward compatibility with C. Within this goal, the new C∀ enumeration should align with the analogous enumeration features in other languages to match modern programming expectations. Hence, the C∀ enumeration features are borrowed from a number of programming languages, but engineered to work and play with C∀'s type system and feature set.

Strong type-checking of enumeration initialization and assignment provides additional safety, ensuring an enumeration only contains its enumerators. Overloading and scoping of enumerators significantly reduces the naming problem, providing a better software-engineering environment, with fewer name clashes and the ability to disambiguate those that cannot be implicitly resolved. Typed enumerations solve the data-harmonization problem, increasing safety through better software engineering. Moreover, integrating enumerations with existing control structures provides a consistent upgrade for programmers and a succinct and secure mechanism to enumerate with the new loop-range feature. Generalization and reuse are supported by incorporating the new enumeration type using the C∀ trait system. Enumeration traits define the meaning of an enumeration, allowing functions to be written that work on any enumeration, such as the reading and printing of an enumeration. With advanced structural typing, C enumerations can be extended so they work with all of the enumeration features, providing for legacy C code to be moved forward into the modern C∀ programming domain. Finally, the C∀ project's test suite has been expanded with multiple enumeration feature tests with respect to implicit conversions, control structures, inheritance, interaction with the polymorphic types, and the features built on top of enumeration traits. These tests ensure future C∀ work does not accidentally break the new enumeration system.

In summary, the new C∀ enumeration mechanisms achieve the initial goals, providing C programmers with an intuitive enumeration mechanism for handling modern programming requirements.

7.1 Future Work

The following are ideas to improve and extend the work in this thesis.

1. There are still corner cases being found in the current C∀ enumeration implementation. Fixing some of these corner cases requires changes to the C∀ resolver or extensions to C∀. When these changes are made, it should be straightforward to update the C∀ enumeration implementation to work with them.
2. Currently, some aspects of the enumeration trait system require explicitly including the file enum.hfa, which can lead to problems. It should be possible to have this file included implicitly by updating the C∀ prelude.
3. There are multiple C∀ features being developed in parallel with enumerations. Two closely related features are iterator and namespace. Enumerations may have to be modified to dovetail with these features. For example, enumerating with range loops does not align with the current iterator design, so some changes will be necessary.
4. C already provides **const**-style aliasing using the *unnamed* enumerator (see Section 2.1.2.1, p. 9), even if the name **enum** is misleading (**const** would be better). Given the existence of this form, it is conceivable to extend it with types other than **int**.

   ```
   enum { Size = 20u, PI = 3.14159L, Jack = L"John" };
   ```

   which matches with **const** aliasing in other programming languages. Here, the type of the enumerator is the type of the initialization constant, *e.g.*, **typeof**(20u) for Size implies **unsigned int**. Auto-initialization is restricted to the case where all constants are **int**, matching with C. As seen in Section 4.4, p. 23, this feature is just a shorthand for multiple typed-enumeration declarations.

   ```
   enum( unsigned int ) { Size = 20u };
   enum( long double ) { PI = 3.14159L };
   enum( wchar_t * ) { Jack = L"John" };
   ```
5. Currently, enumeration scoping is all or nothing. In some cases, it might be useful to increase the scoping granularity to individual enumerators.

   ```
   enum E1 { !A, ^B, C };
   enum E2 ! { !A, ^B, C };
   ```

   Here, '!' means the enumerator is scoped, and '^' means the enumerator is unscoped. For E1, A is scoped; B and C are unscoped. For E2, A, and C are scoped; B is unscoped. Finding a use case is important to justify this extension.
6. An extension mentioned in Section 6.2, p. 43 is using **typedef** to create an enumerator alias.

   ```
   enum( int ) RGB { Red, Green, Blue };
   enum( int ) Traffic_Light { Red, Yellow, Green };
   typedef RGB.Red OtherRed; // alias
   ```
7. Label and values arrays are auxiliary data structures that are always generated for C∀ enumeration, which is a program overhead when unused. It might be possible to provide a new syntax or annotation for a C∀ enumeration definition indicating these arrays are not used. Therefore, C∀ does not generated them. label can still be used on an enumeration constant, as the call reduces to a **char** * constant expression that holds the name of the enumerator. But calls on an enumeration variable, generate a compile-time error. The best alternative is for the linker to discard these arrays if unused.
8. The C∀ enumeration has limitations with separate compilation. Consider the following:

```
enum C_Codec {                          enum( int ) CFA_Codec {
    FIRST_VIDEO = 0,                        FIRST_VIDEO = 0,
    VP8 = 0,                                VP8 = 0,
    VP9,                                    VP9,
    LAST_VIDEO,                             LAST_VIDEO,

    FIRST_AUDIO = 64,                       FIRST_AUDIO = 64,
    VORBIS = 64,                            VORBIS = 64,
    OPUS,                                   OPUS,
    LAST_AUDIO                              LAST_AUDIO
};                                      };
C_Codec c_code = OPUS;
CFA_Codec cfa_code = OPUS;
```

c_code has value 65, the integral value of c_code.OPUS, while cfa_code has value 6, the position of CFA_Codec.OPUS.

If the enumerator AV1 is inserted in C_Codec and CFA_Codec,

```
enum C_Codec {                          enum( int ) CFA_Codec {
    FIRST_VIDEO = 0,                        FIRST_VIDEO = 0,
    VP8 = 0,                                VP8 = 0,
    VP9,                                    VP9,
    AV1,                                    AV1,
    LAST_VIDEO,                             LAST_VIDEO,

    FIRST_AUDIO = 64,                       FIRST_AUDIO = 64,
    VORBIS = 64,                            VORBIS = 64,
    OPUS,                                   OPUS,
    LAST_AUDIO                              LAST_AUDIO
};                                      };
```

the assignments still result in c_code with value 65, but cfa_code is now 7. For C∀, if all translation units including CFA_Codec are not recompiled, some could assign the old 6 and some the new 7, while partially compiled C translation units all continue to assign 65.

For C∀ to achieve the same behaviour for positions as C does with value for partial recompilation, enumeration positions could be represented as **const** declarations with corresponding **extern** declarations in the include file.

```
const int FIRST_VIDEO_posn = 0;         const int FIRST_VIDEO_posn = 0;
const int VP8_posn = 1;                 const int VP8_posn = 1;
const int VP9_posn = 2;                 const int VP9_posn = 2;
                                        const int AV1_posn = 3;
const int LAST_VIDEO_posn = 3;          const int LAST_VIDEO_posn = 4;
const int FIRST_AUDIO_posn = 4;         const int FIRST_AUDIO_posn = 5;
const int VORBIS_posn = 5;              const int VORBIS_posn = 6;
const int OPUS_posn = 6;                const int OPUS_posn = 7;
const int LAST_AUDIO_posn = 7;          const int LAST_AUDIO_posn = 8;
```

Then the linker always uses the most recent object file with the up-to-date positions. However, this implementation means the equivalent of a position array is generated using more storage.

References

[1] Ada. *The Programming Language Ada: Reference Manual*. United States Department of Defense, Springer, New York, ANSI/MIL-STD-1815A-1983 edition, February 1983. 41

[2] Richard C. Bilson. Implementing overloading and polymorphism in C∀. Master's thesis, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, 2003. http://plg.uwaterloo.ca/theses/BilsonThesis.pdf. 17

[3] Michaël Van Canneyt. *Free Pascal Reference Guide, version 3.2.2*, May 2021. http://downloads.freepascal.org/fpc/docs-pdf/ref.pdf. 42

[4] cppreference.com. std::monostate3. https://en.cppreference.com/w/cpp/utility/variant/monostate, September 2024. 6

[5] Oxford English Dictionary. enumerate (v.), sense 3. https://doi.org/10.1093/OED/1113960777, September 2023. 2

[6] ECMA International Standardizing Information and Communication Systems, Geneva, Switzerland. *C# Language Specification, Standard ECMA-334*, 4th edition, June 2006. 41

[7] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, Reading, 2nd edition, 2000. 41

[8] Robert Griesemer, Rob Pike, and Ken Thompson. *Go Programming Language*. Google, Mountain View, CA, USA, 2009. https://go.dev/ref/spec. 41

[9] Paul Hudak and Joseph H. Fasel. A gentle introduction to haskell. *SIGPLAN Not.*, 27(5):T1–53, May 1992. 41

[10] International Standard Organization, Geneva, Switzerland. *C++ Programming Language ISO/IEC 14882:1998*, 1st edition, 1998. https://www.iso.org/standard/25845.html. 21, 46

[11] International Standard Organization, Geneva, Switzerland. *C Programming Language ISO/IEC 9889:2011-12*, 3rd edition, December 2011. https://www.iso.org/standard/57853.html. 9, 10, 17

[12] 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. 41

[13] Chris Lattner, Doug Gregor, John McCall, Ted Kremenek, Joe Groff, and Apple Inc. *The Swift Programming Language*. Apple Inc., Cupertino, CA, USA, 5.9.2 edition, 2024. https://docs.swift.org/swift-book/documentation/the-swift-programming-language. 41

[14] C. H. Lindsey and S. G. van der Meulen. *Informal Introduction to ALGOL 68*. North-Holland, London, 1977. 5

[15] Aaron Moss, Robert Schluntz, and Peter A. Buhr. C∀ : Adding modern programming language features to C. *Softw. Pract. Exper.*, 48(12):2111–2146, December 2018. 18

[16] *The OCaml system, release 5.1*. Rust Project Developers, 2023. https://v2.ocaml.org/manual/. 41

[17] Python. *Python Language Reference, Release 3.7.2*. Python Software Foundation, https://docs.python.org/3/reference/index.html, 2018. 41

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