Resource Management and Tuples in C∀

by

Robert Schluntz

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, 2017

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

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

Abstract

C$\forall$ is a modern, non-object-oriented extension of the C programming language. This thesis addresses several critical deficiencies of C, notably: resource management, a limited function-return mechanism, and unsafe variadic functions. To solve these problems, two fundamental language features are introduced: tuples and constructors/destructors. While these features exist in prior programming languages, the contribution of this work is engineering these features into a highly complex type system. C is an established language with a dedicated user-base. An important goal is to add new features in a way that naturally feels like C, to appeal to this core user-base, and due to huge amounts of legacy code, maintaining backwards compatibility is crucial.

Acknowledgements

I would like to thank my supervisor, Professor Peter Buhr, for all of his help, including reading the many drafts of this thesis and providing guidance throughout my degree. This work would not have been as enjoyable, nor would it have been as strong without Peter's knowledge, help, and encouragement.

I would like to thank my readers, Professors Gregor Richards and Patrick Lam for all of their helpful feedback.

Thanks to Aaron Moss and Thierry Delisle for many helpful discussions, both work-related and not, and for all of the work they have put into the C∀ project. This thesis would not have been the same without their efforts.

I thank Glen Ditchfield and Richard Bilson, for all of their help with both the design and implementation of C∀.

I thank my partner, Erin Blackmere, for all of her love and support. Without her, I would not be who I am today.

Thanks to my parents, Bob and Jackie Schluntz, for their love and support throughout my life, and for always encouraging me to be my best.

Thanks to my best friends, Travis Bartlett, Abraham Dubrisingh, and Kevin Wu, whose companionship is always appreciated. The time we've spent together over the past 4 years has always kept me entertained. An extra shout-out to Kaleb Alway, Max Bardakov, Ten Bradley, and Ed Lee, with whom I've shared many a great meal; thank you for being my friend.

Finally, I would like to acknowledge financial support in the form of a David R. Cheriton Graduate Scholarship and a corporate partnership with Huawei Ltd.

Table of Contents

List of Tables

Chapter 1

Introduction

1.1 C∀ Background

C∀[1] is a modern non-object-oriented extension to the C programming language. As it is an extension of C, there is already a wealth of existing C code and principles that govern the design of the language. Among the goals set out in the original design of C∀, four points stand out [3].

1. The behaviour of standard C code must remain the same when translated by a C∀ compiler as when translated by a C compiler.
2. Standard C code must be as fast and as small when translated by a C∀ compiler as when translated by a C compiler.
3. C∀ code must be at least as portable as standard C code.
4. Extensions introduced by C∀ must be translated in the most efficient way possible.

Therefore, these design principles must be kept in mind throughout the design and development of new language features. In order to appeal to existing C programmers, great care must be taken to ensure that new features naturally feel like C. These goals ensure existing C code-bases can be converted to C∀ incrementally with minimal effort, and C programmers can productively generate C∀ code without training beyond the features being used. Unfortunately, C++ is actively diverging from C, so incremental additions require significant effort and training, coupled with multiple legacy design-choices that cannot be updated.

The current implementation of C∀ is a source-to-source translator from C∀ to GNU C [6].

The remainder of this section describes some of the important features that currently exist in C∀, to give the reader the necessary context in which the new features presented in this thesis must dovetail.

1.1.1 C Background

In the context of this work, the term *object* refers to a region of data storage in the execution environment, the contents of which can represent values [14, p. 6].

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

One of the lesser-known features of standard C is *designations*. Designations are similar to named parameters in languages such as Python and Scala, except that they only apply to aggregate initializers. Note that in C∀, designations use a colon separator, rather than an equals sign as in C, because this syntax is one of the few places that conflicts with the new language features.

```
struct A {
  int w, x, y, z;
};
A a0 = { .x:4 .z:1, .x:8 };
A a1 = { 1, .y:7, 6 };
A a2[4] = { [2]:a0, [0]:a1, { .z:3 } };
// equivalent to
// A a0 = { 0, 8, 0, 1 };
// A a1 = { 1, 0, 7, 6 };
// A a2[4] = { a1, { 0, 0, 0, 3 }, a0, { 0, 0, 0, 0 } };
```

Designations allow specifying the field to initialize by name, rather than by position. Any field not explicitly initialized is initialized as if it had static storage duration [14, p. 141]. A designator specifies the current object for initialization, and as such any undesignated sub-objects pick up where the last initialization left off. For example, in the initialization of `a1`, the initializer of `y` is `7`, and the unnamed initializer `6` initializes the next sub-object, `z`. Later initializers override earlier initializers, so a sub-object for which there is more than one initializer is only initialized by its last initializer. These semantics can be seen in the initialization of `a0`, where `x` is designated twice, and thus initialized to `8`.

C also provides *compound literal* expressions, which provide a first-class mechanism for creating unnamed objects.

```
struct A { int x, y; };
int f(A, int);
int g(int *);

f((A){ 3, 4 }, (int){ 5 } = 10);
g((int[]){ 1, 2, 3 });
g(&(int){ 0 });
```

Compound literals create an unnamed object, and result in an lvalue, so it is legal to assign a value into a compound literal or to take its address [14, p. 86]. Syntactically, compound literals look like a cast operator followed by a brace-enclosed initializer, but semantically are different from a C cast, which only applies basic conversions and coercions and is never an lvalue.

The C∀ translator makes use of several GNU C extensions, including *nested functions* and *attributes*. Nested functions make it possible to access data that is lexically in scope in the nested function's body.

```
int f() {
  int x = 0;
  void g() {
    x++;
  }
  g();  // changes x
}
```

Nested functions come with the usual C caveat that they should not leak into the containing environment, since they are only valid as long as the containing function's stack frame is active.

Attributes make it possible to inform the compiler of certain properties of the code. For example, a function can be marked as deprecated, so that legacy APIs can be identified and slowly removed, or as *hot*, so that the compiler knows the function is called frequently and should be aggresively optimized.

```
__attribute__((deprecated("foo is deprecated, use bar instead")))
void foo();
__attribute__((hot)) void bar(); // heavily optimized

foo();  // warning
bar();
```

1.1.2 Overloading

Overloading is the ability to specify multiple entities with the same name. The most common form of overloading is function overloading, wherein multiple functions can be defined with the same name, but with different signatures. C provides a small amount of built-in overloading, *e.g.*, + is overloaded for the basic types. Like in C++, C∀ allows user-defined overloading based both on the number of parameters and on the types of parameters.

```
void f(void);  // (1)
void f(int);   // (2)
void f(char);  // (3)

f('A');        // selects (3)
```

In this case, there are three `f` procedures, where `f` takes either 0 or 1 arguments, and if an argument is provided then it may be of type **`int`** or of type **`char`**. Exactly which procedure is executed depends on the number and types of arguments passed. If there is no exact match available, C∀ attempts to find a suitable match by examining the C built-in conversion heuristics. The C∀ expression resolution algorithm uses a cost function to determine the interpretation that uses the fewest conversions and polymorphic type bindings.

```
void g(long long);

g(12345);
```

In the above example, there is only one instance of `g`, which expects a single parameter of type **`long long`**. Here, the argument provided has type **`int`**, but since all possible values of type **`int`** can be represented by a value of type **`long long`**, there is a safe conversion from **`int`** to **`long long`**, and so C∀ calls the provided `g` routine.

Overloading solves the problem present in C where there can only be one function with a given name, requiring multiple names for functions that perform the same operation but take in different types. This can be seen in the example of the absolute value functions C:

```
// stdlib.h
int abs(int);
long int labs(long int);
long long int llabs(long long int);
```

In C∀, the functions `labs` and `llabs` are replaced by appropriate overloads of `abs`.

In addition to this form of overloading, C∀ also allows overloading based on the number and types of *return* values. This extension is a feature that is not available in C++, but is available in other programming languages such as Ada [12].

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

int x = g();      // selects (1)
```

Here, the only difference between the signatures of the different versions of `g` is in the return values. The result context is used to select an appropriate routine definition. In this case, the result of `g` is assigned into a variable of type **`int`**, so C∀ prefers the routine that returns a single **`int`**, because it is an exact match.

Return-type overloading solves similar problems to parameter-list overloading, in that multiple functions that perform similar operations can have the same, but produce different values. One use case for this feature is to provide two versions of the `bsearch` routine:

```
forall(otype T | { int ?<?( T, T ); })
T * bsearch(T key, const T * arr, size_t dimension) {
  int comp(const void * t1, const void * t2) {
    return *(T *)t1 < *(T *)t2 ? -1 : *(T *)t2 < *(T *)t1 ? 1 : 0;
  }
  return (T *)bsearch(&key, arr, dimension, sizeof(T), comp);
}
forall(otype T | { int ?<?( T, T ); })
unsigned int bsearch(T key, const T * arr, size_t dimension) {
  T *result = bsearch(key, arr, dimension);
  // pointer subtraction includes sizeof(T)
  return result ? result - arr : dimension;
}
double key = 5.0;
double vals[10] = { /* 10 floating-point values */ };

double * val = bsearch( 5.0, vals, 10 ); // selection based on return type
int posn = bsearch( 5.0, vals, 10 );
```

The first version provides a thin wrapper around the C `bsearch` routine, converting untyped **`void`** `*` to the polymorphic type `T *`, allowing the C∀ compiler to catch errors when the type of `key`, `arr`, and the target at the call-site do not agree. The second version provides an alternate return of the index in the array of the selected element, rather than its address.

There are times when a function should logically return multiple values. Since a function in standard C can only return a single value, a programmer must either take in additional return values by address, or the function's designer must create a wrapper structure to package multiple return-values. For example, the first approach:

```
int f(int * ret) {          // returns a value through parameter ret
  *ret = 37;
  return 123;
}

int res1, res2;             // allocate return value
int res1 = g(&res2);        // explicitly pass storage
```

is awkward because it requires the caller to explicitly allocate memory for *n* result variables, even if they are only temporary values used as a subexpression, or even not used at all. The second approach:

```
struct A {
  int x, y;
};
struct A g() {              // returns values through a structure
  return (struct A) { 123, 37 };
}
struct A res3 = g();
... res3.x ... res3.y ... // use result values
```

is awkward because the caller has to either learn the field names of the structure or learn the names of helper routines to access the individual return values. Both approaches are syntactically unnatural.

In C∀, it is possible to directly declare a function returning multiple values. This extension provides important semantic information to the caller, since return values are only for output.

```
[int, int] f() {        // no new type
  return [123, 37];
}
```

However, the ability to return multiple values is useless without a syntax for accepting the results from the function.

In standard C, return values are most commonly assigned directly into local variables, or are used as the arguments to another function call. C∀ allows both of these contexts to accept multiple return values.

```
int res1, res2;
[res1, res2] = f();     // assign return values into local variables

void g(int, int);
g(f());                 // pass both return values of f to g
```

As seen in the example, it is possible to assign the results from a return value directly into local variables. These local variables can be referenced naturally, without requiring any unpacking as in structured return values. Perhaps more interesting is the fact that multiple return values can be passed to multiple parameters seamlessly, as in the call `g(f())`. In this call, the return values from `f` are linked to the parameters of `g` so that each of the return values is passed directly to the corresponding parameter of `g`, without any explicit storing, unpacking, or additional naming.

An extra quirk introduced by multiple return values is in the resolution of function calls.

```
int f();              // (1)
[int, int] f();       // (2)

void g(int, int);

int x, y;
[x, y] = f();         // selects (2)
g(f());               // selects (2)
```

In this example, the only possible call to `f` that can produce the two **`int`**s required for assigning into the variables `x` and `y` is the second option. A similar reasoning holds calling the function `g`.

This duality between aggregation and aliasing can be seen in the C standard library in the `div` and `remquo` functions, which return the quotient and remainder for a division of integer and floating-point values, respectively.

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

`div` aggregates the quotient/remainder in a structure, while `remquo` aliases a parameter to an argument. Alternatively, a programming language can directly support returning multiple values, *e.g.*, in C∀:

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

In C∀, overloading also applies to operator names, known as *operator overloading*. Similar to function overloading, a single operator is given multiple meanings by defining new versions of the operator with different signatures. In C++, this can be done as follows

```
struct A { int i; };
A operator+(A x, A y);
bool operator<(A x, A y);
```

In C∀, the same example can be written as follows.

```
struct A { int i; };
A ?+?(A x, A y);    // '?'s represent operands
int ?<?(A x, A y);
```

Notably, the only difference is syntax. Most of the operators supported by C++ for operator overloading are also supported in C∀. Of notable exception are the logical operators (*e.g.*, `||`), the sequence operator (*i.e.*, `,`), and the member-access operators (*e.g.*, `.` and `->`).

Finally, C∀ also permits overloading variable identifiers. This feature is not available in C++.

```
struct Rational { int numer, denom; };
int x = 3;                  // (1)
double x = 1.27;            // (2)
Rational x = { 4, 11 };     // (3)

void g(double);

x += 1;                     // chooses (1)
g(x);                       // chooses (2)
Rational y = x;             // chooses (3)
```

In this example, there are three definitions of the variable `x`. Based on the context, C∀ attempts to choose the variable whose type best matches the expression context. When used judiciously, this feature allows names like `MAX`, `MIN`, and `PI` to apply across many types.

Finally, the values `0` and `1` have special status in standard C. In particular, the value `0` is both an integer and a pointer literal, and thus its meaning depends on the context. In addition, several operations can be redefined in terms of other operations and the values `0` and `1`. For example,

```
int x;
if (x) {  // if (x != 0)
  x++;    //   x += 1;
}
```

Every if- and iteration-statement in C compares the condition with `0`, and every increment and decrement operator is semantically equivalent to adding or subtracting the value `1` and storing the result. Due to these rewrite rules, the values `0` and `1` have the types **`zero_t`** and **`one_t`** in C∀, which allow for overloading various operations that connect to `0` and `1` [2]. The types **`zero_t`** and **`one_t`** have special built-in implicit conversions to the various integral types, and a conversion to pointer types for `0`, which allows standard C code involving `0` and `1` to work as normal.

```
// lvalue is similar to returning a reference in C++
lvalue Rational ?+=?(Rational *a, Rational b);
Rational ?=?(Rational * dst, zero_t) {
  return *dst = (Rational){ 0, 1 };
}

Rational sum(Rational *arr, int n) {
  Rational r;
  r = 0;      // use rational-zero_t assignment
  for (; n > 0; n--) {
    r += arr[n-1];
  }
  return r;
}
```

This function takes an array of `Rational` objects and produces the `Rational` representing the sum of the array. Note the use of an overloaded assignment operator to set an object of type `Rational` to an appropriate `0` value.

[2] In the original design of C∀, `0` and `1` were overloadable names [8, p. 7].

1.1.3 Polymorphism

In its most basic form, polymorphism grants the ability to write a single block of code that accepts different types. In particular, C∀ supports the notion of parametric polymorphism. Parametric polymorphism allows a function to be written generically, for all values of all types, without regard to the specifics of a particular type. For example, in C++, the simple identity function for all types can be written as:

```
template<typename T>
T identity(T x) { return x; }
```

C++ uses the template mechanism to support parametric polymorphism. In C∀, an equivalent function can be written as:

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

Once again, the only visible difference in this example is syntactic. Fundamental differences can be seen by examining more interesting examples. In C++, a generic sum function is written as follows:

```
template<typename T>
T sum(T *arr, int n) {
  T t;  // default construct => 0
  for (; n > 0; n--) t += arr[n-1];
  return t;
}
```

Here, the code assumes the existence of a default constructor, assignment operator, and an addition operator over the provided type `T`. If any of these required operators are not available, the C++ compiler produces an error message stating which operators could not be found.

A similar sum function can be written in C∀ as follows:

```
forall(otype T | { T ?=?(T *, zero_t); T ?+=?(T *, T); })
T sum(T *arr, int n) {
  T t = 0;
  for (; n > 0; n--) t = t += arr[n-1];
  return t;
}
```

The first thing to note here is that immediately following the declaration of **otype** `T` is a list of *type assertions* that specify restrictions on acceptable choices of `T`. In particular, the assertions above specify that there must be an assignment from **zero_t** to `T` and an addition assignment operator from `T` to `T`. The existence of an assignment operator from `T` to `T` and the ability to create an object of type `T` are assumed implicitly by declaring `T` with the **otype** type-class. In addition to **otype**, there are currently two other type-classes.

dtype, short for *data type*, serves as the top type for object types; any object type, complete or incomplete, can be bound to a **dtype** type variable. To contrast, **otype**, short for *object type*, is a **dtype** with known size, alignment, and an assignment operator, and thus bind only to complete object types. With this extra information, complete objects can be used in polymorphic

code in the same way they are used in monomorphic code, providing familiarity and ease of use. The third type-class is **ftype**, short for *function type*, matching only function types. The three type parameter kinds are summarized in Table 1.1

name	object type	incomplete type	function type	can assign	can create	has size
otype	X			X	X	X
dtype	X	X				
ftype			X			

Table 1.1: The different kinds of type parameters in C∀

A major difference between the approaches of C++ and C∀ to polymorphism is that the set of assumed properties for a type is *explicit* in C∀. One of the major limiting factors of C++'s approach is that templates cannot be separately compiled. In contrast, the explicit nature of assertions allows C∀'s polymorphic functions to be separately compiled, as the function prototype states all necessary requirements separate from the implementation. For example, the prototype for the previous sum function is

```
forall(otype T | { T ?=?(T *, zero_t); T ?+=?(T *, T); })
T sum(T *arr, int n);
```

With this prototype, a caller in another translation unit knows all of the constraints on `T`, and thus knows all of the operations that need to be made available to `sum`.

In C∀, a set of assertions can be factored into a *trait*.

```
trait Addable(otype T) {
  T ?+?(T, T);
  T ++?(T);
  T ?++(T);
}
forall(otype T | Addable(T)) void f(T);
forall(otype T | Addable(T) | { T --?(T); }) T g(T);
forall(otype T, U | Addable(T) | { T ?/?(T, U); }) U h(T, U);
```

This capability allows specifying the same set of assertions in multiple locations, without the repetition and likelihood of mistakes that come with manually writing them out for each function declaration.

An interesting application of return-type resolution and polymorphism is a polymorphic version of `malloc`.

```
forall(dtype T | sized(T))
T * malloc() {
  return (T*)malloc(sizeof(T)); // call C malloc
}
int * x = malloc();      // malloc(sizeof(int))
double * y = malloc();   // malloc(sizeof(double))

struct S { ... };
S * s = malloc();        // malloc(sizeof(S))
```

The built-in trait `sized` ensures that size and alignment information for `T` is available in the body of `malloc` through **`sizeof`** and **`_Alignof`** expressions respectively. In calls to `malloc`, the type `T` is bound based on call-site information, allowing C∀ code to allocate memory without the potential for errors introduced by manually specifying the size of the allocated block.

1.1.4 Planned Features

One of the planned features C∀ is *reference types*. At a high level, the current proposal is to add references as a way to cleanup pointer syntax. With references, it will be possible to store any address, as with a pointer, with the key difference being that references are automatically dereferenced.

```
int x = 0;
int * p = &x;  // needs &
int & ref = x; // no &

printf("%d %d\n", *p, ref); // pointer needs *, ref does not
```

It is possible to add new functions or shadow existing functions for the duration of a scope, using normal C scoping rules. One application of this feature is to reverse the order of `qsort`.

```
forall(otype T | { int ?<?( T, T ); })
void qsort(const T * arr, size_t size) {
  int comp(const void * t1, const void * t2) {
    return *(T *)t1 < *(T *)t2 ? -1 : *(T *)t2 < *(T *)t1 ? 1 : 0;
  }
  qsort(arr, dimension, sizeof(T), comp);

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

Currently, there is no way to *remove* a function from consideration from the duration of a scope. For example, it may be desirable to eliminate assignment from a scope, to reduce accidental mutation. To address this desire, *deleted functions* are a planned feature for C∀.

```
forall(otype T) void f(T *);

int x = 0;
f(&x);  // might modify x
{
  int ?=?(int *, int) = delete;
  f(&x);   // error, no assignment for int
}
```

Now, if the deleted function is chosen as the best match, the expression resolver emits an error.

1.2 Invariants

An *invariant* is a logical assertion that is true for some duration of a program's execution. Invariants help a programmer to reason about code correctness and prove properties of programs.

In object-oriented programming languages, type invariants are typically established in a constructor and maintained throughout the object's lifetime. These assertions are typically achieved through a combination of access-control modifiers and a restricted interface. Typically, data which requires the maintenance of an invariant is hidden from external sources using the *private* modifier, which restricts reads and writes to a select set of trusted routines, including member functions. It is these trusted routines that perform all modifications to internal data in a way that is consistent with the invariant, by ensuring that the invariant holds true at the end of the routine call.

In C, the `assert` macro is often used to ensure invariants are true. Using `assert`, the programmer can check a condition and abort execution if the condition is not true. This powerful tool forces the programmer to deal with logical inconsistencies as they occur. For production, assertions can be removed by simply defining the preprocessor macro `NDEBUG`, making it simple to ensure that assertions are 0-cost for a performance intensive application.

```
struct Rational {
  int n, d;
};
struct Rational create_rational(int n, int d) {
  assert(d != 0);  // precondition
  if (d < 0) {
    n *= -1;
    d *= -1;
  }
  assert(d > 0);  // postcondition
  // rational invariant: d > 0
  return (struct Rational) { n, d };
}
struct Rational rat_abs(struct Rational r) {
  assert(r.d > 0); // check invariant, since no access control
  r.n = abs(r.n);
  assert(r.d > 0); // ensure function preserves invariant on return value
  return r;
}
```

Some languages, such as D, provide language-level support for specifying program invariants. In addition to providing a C-like `assert` expression, D allows specifying type invariants that are automatically checked at the end of a constructor, beginning of a destructor, and at the beginning and end of every public member function.

```
import std.math;
struct Rational {
  invariant {
    assert(d > 0, "d <= 0");
  }
  int n, d;
  this(int n, int d) {  // constructor
```

```
    assert(d != 0);
    this.n = n;
    this.d = d;
    // implicitly check invariant
  }
  Rational abs() {
    // implicitly check invariant
    return Rational(std.math.abs(n), d);
    // implicitly check invariant
  }
}
```

The D compiler is able to assume that assertions and invariants hold true and perform optimizations based on those assumptions. Note, these invariants are internal to the type's correct behaviour.

Types also have external invariants with the state of the execution environment, including the heap, the open-file table, the state of global variables, etc. Since resources are finite and shared (concurrency), it is important to ensure that objects clean up properly when they are finished, restoring the execution environment to a stable state so that new objects can reuse resources.

1.3 Resource Management

Resource management is a problem that pervades every programming language.

In standard C, resource management is largely a manual effort on the part of the programmer, with a notable exception to this rule being the program stack. The program stack grows and shrinks automatically with each function call, as needed for local variables. However, whenever a program needs a variable to outlive the block it is created in, the storage must be allocated dynamically with `malloc` and later released with `free`. This pattern is extended to more complex objects, such as files and sockets, which can also outlive the block where they are created, and thus require their own resource management. Once allocated storage escapes[3] a block, the responsibility for deallocating the storage is not specified in a function's type, that is, that the return value is owned by the caller. This implicit convention is provided only through documentation about the expectations of functions.

In other languages, a hybrid situation exists where resources escape the allocation block, but ownership is precisely controlled by the language. This pattern requires a strict interface and protocol for a data structure, consisting of a pre-initialization and a post-termination call, and all intervening access is done via interface routines. This kind of encapsulation is popular in object-oriented programming languages, and like the stack, it takes care of a significant portion of resource-management cases.

For example, C++ directly supports this pattern through class types and an idiom known as RAII [4] by means of constructors and destructors. Constructors and destructors are special

[3] In garbage collected languages, such as Java, escape analysis [7] is used to determine when dynamically allocated objects are strictly contained within a function, which allows the optimizer to allocate them on the stack.

[4] Resource Acquisition is Initialization

routines that are automatically inserted into the appropriate locations to bookend the lifetime of an object. Constructors allow the designer of a type to establish invariants for objects of that type, since it is guaranteed that every object must be initialized through a constructor. In particular, constructors allow a programmer to ensure that all objects are initially set to a valid state. On the other hand, destructors provide a simple mechanism for tearing down an object and resetting the environment in which the object lived. RAII ensures that if all resources are acquired in a constructor and released in a destructor, there are no resource leaks, even in exceptional circumstances. A type with at least one non-trivial constructor or destructor is henceforth referred to as a *managed type*. In the context of C∀, a non-trivial constructor is either a user defined constructor or an auto-generated constructor that calls a non-trivial constructor.

For the remaining resource ownership cases, a programmer must follow a brittle, explicit protocol for freeing resources or an implicit protocol enforced by the programming language.

In garbage collected languages, such as Java, resources are largely managed by the garbage collector. Still, garbage collectors typically focus only on memory management. There are many kinds of resources that the garbage collector does not understand, such as sockets, open files, and database connections. In particular, Java supports *finalizers*, which are similar to destructors. Unfortunately, finalizers are only guaranteed to be called before an object is reclaimed by the garbage collector [11, p. 373], which may not happen if memory use is not contentious. Due to operating-system resource-limits, this is unacceptable for many long running programs. Instead, the paradigm in Java requires programmers to manually keep track of all resources *except* memory, leading many novices and experts alike to forget to close files, etc. Complicating the picture, uncaught exceptions can cause control flow to change dramatically, leaking a resource that appears on first glance to be released.

```
void write(String filename, String msg) throws Exception {
  FileOutputStream out = new FileOutputStream(filename);
  FileOutputStream log = new FileOutputStream(filename);
  out.write(msg.getBytes());
  log.write(msg.getBytes());
  log.close();
  out.close();
}
```

Any line in this program can throw an exception, which leads to a profusion of finally blocks around many function bodies, since it is not always clear when an exception may be thrown.

```
public void write(String filename, String msg) throws Exception {
  FileOutputStream out = new FileOutputStream(filename);
  try {
    FileOutputStream log = new FileOutputStream("log.txt");
    try {
      out.write(msg.getBytes());
      log.write(msg.getBytes());
    } finally {
      log.close();
    }
  } finally {
    out.close();
  }
}
```

In Java 7, a new *try-with-resources* construct was added to alleviate most of the pain of working with resources, but ultimately it still places the burden squarely on the user rather than on the library designer. Furthermore, for complete safety this pattern requires nested objects to be declared separately, otherwise resources that can throw an exception on close can leak nested resources [5] [16].

```
public void write(String filename, String msg) throws Exception {
  try (  // try-with-resources
    FileOutputStream out = new FileOutputStream(filename);
    FileOutputStream log = new FileOutputStream("log.txt");
  ) {
    out.write(msg.getBytes());
    log.write(msg.getBytes());
  } // automatically closes out and log in every exceptional situation
}
```

Variables declared as part of a try-with-resources statement must conform to the `AutoClosable` interface, and the compiler implicitly calls `close` on each of the variables at the end of the block. Depending on when the exception is raised, both `out` and `log` are null, `log` is null, or both are non-null, therefore, the cleanup for these variables at the end is automatically guarded and conditionally executed to prevent null-pointer exceptions.

While Rust [17] does not enforce the use of a garbage collector, it does provide a manual memory management environment, with a strict ownership model that automatically frees allocated memory and prevents common memory management errors. In particular, a variable has ownership over its associated value, which is freed automatically when the owner goes out of scope. Furthermore, values are *moved* by default on assignment, rather than copied, which invalidates the previous variable binding.

```
struct S {
  x: i32
}
let s = S { x: 123 };
let z = s;           // move, invalidate s
println!("{}", s.x); // error, s has been moved
```

Types can be made copyable by implementing the `Copy` trait.

Rust allows multiple unowned views into an object through references, also known as borrows, provided that a reference does not outlive its referent. A mutable reference is allowed only if it is the only reference to its referent, preventing data race errors and iterator invalidation errors.

```
let mut x = 10;
{
  let y = &x;
  let z = &x;
  println!("{} {}", y, z); // prints 10 10
}
```

[5] Since close is only guaranteed to be called on objects declared in the try-list and not objects passed as constructor parameters, the `B` object may not be closed in `new A(new B())` if `A`'s close raises an exception.

```
{
  let y = &mut x;
  // let z1 = &x;      // not allowed, have mutable reference
  // let z2 = &mut x; // not allowed, have mutable reference
  *y = 5;
  println!("{}", y); // prints 5
}
println!("{}", x); // prints 5
```

Since references are not owned, they do not release resources when they go out of scope. There is no runtime cost imposed on these restrictions, since they are enforced at compile-time.

Rust provides RAII through the `Drop` trait, allowing arbitrary code to execute when the object goes out of scope, providing automatic clean up of auxiliary resources, much like a C++ program.

```
struct S {
  name: &'static str
}

impl Drop for S {  // RAII for S
  fn drop(&mut self) {  // destructor
    println!("dropped {}", self.name);
  }
}

{
  let x = S { name: "x" };
  let y = S { name: "y" };
} // prints "dropped y" "dropped x"
```

The programming language D also manages resources with constructors and destructors [4]. In D, **`struct`**s are stack allocatable and managed via scoping like in C++, whereas `class`es are managed automatically by the garbage collector. Like Java, using the garbage collector means that destructors are called indeterminately, requiring the use of finally statements to ensure dynamically allocated resources that are not managed by the garbage collector, such as open files, are cleaned up. Since D supports RAII, it is possible to use the same techniques as in C++ to ensure that resources are released in a timely manner. Finally, D provides a scope guard statement, which allows an arbitrary statement to be executed at normal scope exit with *success*, at exceptional scope exit with *failure*, or at normal and exceptional scope exit with *exit*. It has been shown that the *exit* form of the scope guard statement can be implemented in a library in C++ [1].

To provide managed types in C∀, new kinds of constructors and destructors are added to C∀ and discussed in Chapter 2.

1.4 Tuples

In mathematics, tuples are finite-length sequences which, unlike sets, are ordered and allow duplicate elements. In programming languages, tuples provide fixed-sized heterogeneous lists of

elements. Many programming languages have tuple constructs, such as SETL, K-W C, ML, and Scala.

K-W C, a predecessor of C∀, introduced tuples to C as an extension of the C syntax, rather than as a full-blown data type [21]. In particular, Till noted that C already contains a tuple context in the form of function parameter lists. The main contributions of that work were in the form of adding tuple contexts to assignment in the form of multiple assignment and mass assignment (discussed in detail in section 3.3), function return values (see section 3.1), and record field access (see section 3.4). Adding tuples to C∀ has previously been explored by Esteves [9].

The design of tuples in K-W C took much of its inspiration from SETL [19]. SETL is a high-level mathematical programming language, with tuples being one of the primary data types. Tuples in SETL allow a number of operations, including subscripting, dynamic expansion, and multiple assignment.

C++11 introduced `std::tuple` as a library variadic template struct. Tuples are a generalization of `std::pair`, in that they allow for arbitrary length, fixed-size aggregation of heterogeneous values.

```
tuple<int, int, int> triple(10, 20, 30);
get<1>(triple); // access component 1 => 20

tuple<int, double> f();
int i;
double d;
tie(i, d) = f(); // assign fields of return value into local variables

tuple<int, int, int> greater(11, 0, 0);
triple < greater; // true
```

Tuples are simple data structures with few specific operations. In particular, it is possible to access a component of a tuple using `std::get<N>`. Another interesting feature is `std::tie`, which creates a tuple of references, allowing assignment of the results of a tuple-returning function into separate local variables, without requiring a temporary variable. Tuples also support lexicographic comparisons, making it simple to write aggregate comparators using `std::tie`.

There is a proposal for C++17 called *structured bindings* [20], that introduces new syntax to eliminate the need to pre-declare variables and use `std::tie` for binding the results from a function call.

```
tuple<int, double> f();
auto [i, d] = f(); // unpacks into new variables i, d

tuple<int, int, int> triple(10, 20, 30);
auto & [t1, t2, t3] = triple;
t2 = 0; // changes middle element of triple

struct S { int x; double y; };
S s = { 10, 22.5 };
auto [x, y] = s; // unpack s
```

Structured bindings allow unpacking any structure with all public non-static data members into fresh local variables. The use of `&` allows declaring new variables as references, which is some-

thing that cannot be done with `std::tie`, since C++ references do not support rebinding. This extension requires the use of **`auto`** to infer the types of the new variables, so complicated expressions with a non-obvious type must be documented with some other mechanism. Furthermore, structured bindings are not a full replacement for `std::tie`, as it always declares new variables.

Like C++, D provides tuples through a library variadic-template structure. In D, it is possible to name the fields of a tuple type, which creates a distinct type.

```
Tuple!(float, "x", float, "y") point2D;
Tuple!(float, float) float2;  // different type from point2D

point2D[0]; // access first element
point2D.x;  // access first element

float f(float x, float y) {
  return x+y;
}

f(point2D.expand);
```

Tuples are 0-indexed and can be subscripted using an integer or field name, if applicable. The `expand` method produces the components of the tuple as a list of separate values, making it possible to call a function that takes *N* arguments using a tuple with *N* components.

Tuples are a fundamental abstraction in most functional programming languages, such as Standard ML [15]. A function in SML always accepts exactly one argument. There are two ways to mimic multiple argument functions: the first through currying and the second by accepting tuple arguments.

```
fun fact (n : int) =
  if (n = 0) then 1
  else n*fact(n-1)

fun binco (n: int, k: int) =
  real (fact n) / real (fact k * fact (n-k))
```

Here, the function `binco` appears to take 2 arguments, but it actually takes a single argument which is implicitly decomposed via pattern matching. Tuples are a foundational tool in SML, allowing the creation of arbitrarily-complex structured data-types.

Scala, like C++, provides tuple types through the standard library [18]. Scala provides tuples of size 1 through 22 inclusive through generic data structures. Tuples support named access and subscript access, among a few other operations.

```
val a = new Tuple3(0, "Text", 2.1) // explicit creation
val b = (6, 'a', 1.1f)       // syntactic sugar: Tuple3[Int, Char, Float]
val (i, _, d) = triple       // extractor syntax, ignore middle element

println(a._2)                // named access => print "Text"
println(b.productElement(0)) // subscript access => print 6
```

In Scala, tuples are primarily used as simple data structures for carrying around multiple values or for returning multiple values from a function. The 22-element restriction is an odd and arbitrary

choice, but in practice it does not cause problems since large tuples are uncommon. Subscript access is provided through the `productElement` method, which returns a value of the top-type `Any`, since it is impossible to receive a more precise type from a general subscripting method due to type erasure. The disparity between named access beginning at `_1` and subscript access starting at `0` is likewise an oddity, but subscript access is typically avoided since it discards type information. Due to the language's pattern matching facilities, it is possible to extract the values from a tuple into named variables, which is a more idiomatic way of accessing the components of a tuple.

C♯ also has tuples, but has similarly strange limitations, allowing tuples of size up to 7 components. The officially supported workaround for this shortcoming is to nest tuples in the 8th component. C♯ allows accessing a component of a tuple by using the field `Item`*N* for components 1 through 7, and `Rest` for the nested tuple.

In Python [22], tuples are immutable sequences that provide packing and unpacking operations. While the tuple itself is immutable, and thus does not allow the assignment of components, there is nothing preventing a component from being internally mutable. The components of a tuple can be accessed by unpacking into multiple variables, indexing, or via field name, like D. Tuples support multiple assignment through a combination of packing and unpacking, in addition to the common sequence operations.

Swift [2], like D, provides named tuples, with components accessed by name, index, or via extractors. Tuples are primarily used for returning multiple values from a function. In Swift, `Void` is an alias for the empty tuple, and there are no single element tuples.

Tuples comparable to those described above are added to C∀ and discussed in Chapter 3.

1.5 Variadic Functions

In statically-typed programming languages, functions are typically defined to receive a fixed number of arguments of specified types. Variadic argument functions provide the ability to define a function that can receive a theoretically unbounded number of arguments.

C provides a simple implementation of variadic functions. A function whose parameter list ends with `, ...` is a variadic function. Among the most common variadic functions is `printf`.

```
int printf(const char * fmt, ...);
printf("%d %g %c %s", 10, 3.5, 'X', "a string");
```

Through the use of a format string, C programmers can communicate argument type information to `printf`, allowing C programmers to print any of the standard C data types. Still, `printf` is extremely limited, since the format codes are specified by the C standard, meaning users cannot define their own format codes to extend `printf` for new data types or new formatting rules.

C provides manipulation of variadic arguments through the `va_list` data type, which abstracts details of the manipulation of variadic arguments. Since the variadic arguments are untyped, it is up to the function to interpret any data that is passed in. Additionally, the interface to manipulate `va_list` objects is essentially limited to advancing to the next argument, without any built-in facility to determine when the last argument is read. This limitation requires the use

of an *argument descriptor* to pass information to the function about the structure of the argument list, including the number of arguments and their types. The format string in `printf` is one such example of an argument descriptor.

```
int f(const char * fmt, ...) {
  va_list args;
  va_start(args, fmt);  // initialize va_list
  for (const char * c = fmt; *c != '\0'; ++c) {
    if (*c == '%') {
      ++c;
      switch (*c) {
        case 'd': {
          int i = va_arg(args, int);  // have to specify type
          // ...
          break;
        }
        case 'g': {
          double d = va_arg(args, double);
          // ...
          break;
        }
        ...
      }
    }
  }
  va_end(args);
  return ...;
}
```

Every case must be handled explicitly, since the `va_arg` macro requires a type argument to determine how the next set of bytes is to be interpreted. Furthermore, if the user makes a mistake, compile-time checking is typically restricted to standard format codes and their corresponding types. In general, this means that C's variadic functions are not type-safe, making them difficult to use properly.

C++11 added support for *variadic templates*, which add much needed type-safety to C's variadic landscape. It is possible to use variadic templates to define variadic functions and variadic data types.

```
void print(int);
void print(char);
void print(double);
...

void f() {}    // base case

template<typename T, typename... Args>
void f(const T & arg, const Args &... rest) {
  print(arg);  // print the current element
  f(rest...);  // handle remaining arguments recursively
}
```

Variadic templates work largely through recursion on the *parameter pack*, which is the argument with `...` following its type. A parameter pack matches 0 or more elements, which can be types

or expressions depending on the context. Like other templates, variadic template functions rely on an implicit set of constraints on a type, in this example a `print` routine. That is, it is possible to use the `f` routine on any type provided there is a corresponding `print` routine, making variadic templates fully open to extension, unlike variadic functions in C.

Recent C++ standards (C++14, C++17) expand on the basic premise by allowing variadic template variables and providing convenient expansion syntax to remove the need for recursion in some cases, amongst other things.

In Java, a variadic function appears similar to a C variadic function in syntax.

```
int sum(int... args) {
  int s = 0;
  for (int x : args) {
    s += x;
  }
  return s;
}

void print(Object... objs) {
  for (Object obj : objs) {
    System.out.print(obj);
  }
}

print("The sum from 1 to 10 is ", sum(1,2,3,4,5,6,7,8,9,10), ".\n");
```

The key difference is that Java variadic functions are type-safe, because they specify the type of the argument immediately prior to the ellipsis. In Java, variadic arguments are syntactic sugar for arrays, allowing access to length, subscripting operations, and for-each iteration on the variadic arguments, among other things. Since the argument type is specified explicitly, the top-type `Object` can be used to accept arguments of any type, but to do anything interesting on the argument requires a down-cast to a more specific type, landing Java in a similar situation to C in that writing a function open to extension is difficult.

The other option is to restrict the number of types that can be passed to the function by using a more specific type. Unfortunately, Java's use of nominal inheritance means that types must explicitly inherit from classes or interfaces in order to be considered a subclass. The combination of these two issues greatly restricts the usefulness of variadic functions in Java.

Type-safe variadic functions are added to C∀ and discussed in Chapter 4.

1.6 Contributions

No prior work on constructors or destructors had been done for C∀. I did both the design and implementation work. While the overall design is based on constructors and destructors in object-oriented C++, it had to be re-engineered into non-object-oriented C∀. I also had to make changes to the C∀ expression-resolver to integrate constructors and destructors into the type system.

Prior work on the design of tuples for C∀ was done by Till, and some initial implementation work by Esteves. I largely took the Till design but added tuple indexing, which exists in

a number of programming languages with tuples, simplified the implicit tuple conversions, and integrated with the C∀ polymorphism and assertion satisfaction model. I did a new implementation of tuples, and extensively augmented initial work by Bilson to incorporate tuples into the C∀ expression-resolver and type-unifier.

No prior work on variadic functions had been done for C∀. I did both the design and implementation work. While the overall design is based on variadic templates in C++, my design is novel in the way it is incorporated into the C∀ polymorphism model, and is engineered into C∀ so it dovetails with tuples.

Chapter 2

Constructors and Destructors

Since C∀ is a true systems language, it does not require a garbage collector. As well, C∀ is not an object-oriented programming language, *i.e.*, structures cannot have methods. While structures can have function pointer members, this is different from methods, since methods have implicit access to structure members and methods cannot be reassigned. Nevertheless, one important goal is to reduce programming complexity and increase safety. To that end, C∀ provides support for implicit pre/post-execution of routines for objects, via constructors and destructors.

This chapter details the design of constructors and destructors in C∀, along with their current implementation in the translator. Generated code samples have been edited for clarity and brevity.

2.1 Design Criteria

In designing constructors and destructors for C∀, the primary goals were ease of use and maintaining backwards compatibility.

In C, when a variable is defined, its value is initially undefined unless it is explicitly initialized or allocated in the static area.

```
int main() {
  int x;         // uninitialized
  int y = 5;     // initialized to 5
  x = y;         // assigned 5
  static int z; // initialized to 0
}
```

In the example above, `x` is defined and left uninitialized, while `y` is defined and initialized to 5. Next, `x` is assigned the value of `y`. In the last line, `z` is implicitly initialized to 0 since it is marked **`static`**. The key difference between assignment and initialization being that assignment occurs on a live object (*i.e.*, an object that contains data). It is important to note that this means `x` could have been used uninitialized prior to being assigned, while `y` could not be used uninitialized. Use of uninitialized variables yields undefined behaviour [14, p. 558], which is a common source of errors in C programs.

Initialization of a declaration is strictly optional, permitting uninitialized variables to exist. Furthermore, declaration initialization is limited to expressions, so there is no way to insert

arbitrary code before a variable is live, without delaying the declaration. Many C compilers give good warnings for uninitialized variables most of the time, but they cannot in all cases.

```
int f(int *);  // output parameter: never reads, only writes
int g(int *);  // input parameter: never writes, only reads,
               // so requires initialized variable

int x, y;
f(&x);  // okay - only writes to x
g(&y);  // uses y uninitialized
```

Other languages are able to give errors in the case of uninitialized variable use, but due to backwards compatibility concerns, this is not the case in C∀.

In C, constructors and destructors are often mimicked by providing routines that create and tear down objects, where the tear down function is typically only necessary if the type modifies the execution environment.

```
struct array_int {
  int * x;
};
struct array_int create_array(int sz) {
  return (struct array_int) { calloc(sizeof(int)*sz) };
}
void destroy_rh(struct resource_holder * rh) {
  free(rh->x);
}
```

This idiom does not provide any guarantees unless the structure is opaque, which then requires that all objects are heap allocated.

```
struct opqaue_array_int;
struct opqaue_array_int * create_opqaue_array(int sz);
void destroy_opaque_array(opaque_array_int *);
int opaque_get(opaque_array_int *);  // subscript

opaque_array_int * x = create_opaque_array(10);
int x2 = opaque_get(x, 2);
```

This pattern is cumbersome to use since every access becomes a function call, requiring awkward syntax and a performance cost. While useful in some situations, this compromise is too restrictive. Furthermore, even with this idiom it is easy to make mistakes, such as forgetting to destroy an object or destroying it multiple times.

A constructor provides a way of ensuring that the necessary aspects of object initialization is performed, from setting up invariants to providing compile- and run-time checks for appropriate initialization parameters. This goal is achieved through a *guarantee* that a constructor is called *implicitly* after every object is allocated from a type with associated constructors, as part of an object's *definition*. Since a constructor is called on every object of a managed type, it is *impossible* to forget to initialize such objects, as long as all constructors perform some sensible form of initialization.

In C∀, a constructor is a function with the name `?{}`. Like other operators in C∀, the name represents the syntax used to call the constructor, *e.g.*, **`struct`** `S = { ... };`. Every

constructor must have a return type of **void** and at least one parameter, the first of which is colloquially referred to as the *this* parameter, as in many object-oriented programming-languages (however, a programmer can give it an arbitrary name). The `this` parameter must have a pointer type, whose base type is the type of object that the function constructs. There is precedence for enforcing the first parameter to be the `this` parameter in other operators, such as the assignment operator, where in both cases, the left-hand side of the equals is the first parameter. There is currently a proposal to add reference types to C∀. Once this proposal has been implemented, the `this` parameter will become a reference type with the same restrictions.

Consider the definition of a simple type encapsulating a dynamic array of **int**s.

```
struct Array {
  int * data;
  int len;
}
```

In C, if the user creates an `Array` object, the fields `data` and `len` are uninitialized, unless an explicit initializer list is present. It is the user's responsibility to remember to initialize both of the fields to sensible values, since there are no implicit checks for invalid values or reasonable defaults. In C∀, the user can define a constructor to handle initialization of `Array` objects.

```
void ?{}(Array * arr){
  arr->len = 10;    // default size
  arr->data = malloc(sizeof(int)*arr->len);
  for (int i = 0; i < arr->len; ++i) {
    arr->data[i] = 0;
  }
}
Array x;  // allocates storage for Array and calls ?{}(&x)
```

This constructor initializes `x` so that its `length` field has the value 10, and its `data` field holds a pointer to a block of memory large enough to hold 10 **int**s, and sets the value of each element of the array to 0. This particular form of constructor is called the *default constructor*, because it is called on an object defined without an initializer. In other words, a default constructor is a constructor that takes a single argument: the `this` parameter.

In C∀, a destructor is a function much like a constructor, except that its name is `^?{}` [1] and it takes only one argument. A destructor for the `Array` type can be defined as:

```
void ^?{}(Array * arr) {
  free(arr->data);
}
```

The destructor is automatically called at deallocation for all objects of type `Array`. Hence, the memory associated with an `Array` is automatically freed when the object's lifetime ends. The exact guarantees made by C∀ with respect to the calling of destructors are discussed in section 2.1.5.

As discussed previously, the distinction between initialization and assignment is important.

[1] Originally, the name `~?{}` was chosen for destructors, to provide familiarity to C++ programmers. Unforunately, this name causes parsing conflicts with the bitwise-not operator when used with operator syntax (see section 2.1.1.)

Consider the following example.

```
1  Array x, y;
2  Array z = x;  // initialization
3  y = x;        // assignment
```

By the previous definition of the default constructor for `Array`, `x` and `y` are initialized to valid arrays of length 10 after their respective definitions. On line 2, `z` is initialized with the value of `x`, while on line 3, `y` is assigned the value of `x`. The key distinction between initialization and assignment is that a value to be initialized does not hold any meaningful values, whereas an object to be assigned might. In particular, these cases cannot be handled the same way because in the former case `z` has no array, while `y` does. A *copy constructor* is used to perform initialization using another object of the same type.

```
void ?{}(Array * arr, Array other) {  // copy constructor
  arr->len = other.len;               // initialization
  arr->data = malloc(sizeof(int)*arr->len)
  for (int i = 0; i < arr->len; ++i) {
    arr->data[i] = other.data[i];     // copy from other object
  }
}
Array ?=?(Array * arr, Array other) { // assignment
  ^?{}(arr);                          // explicitly call destructor
  ?{}(arr, other);                    // explicitly call constructor
  return *arr;
}
```

The two functions above handle the cases of initialization and assignment. The first function is called a copy constructor, because it constructs its argument by copying the values from another object of the same type. The second function is the standard copy-assignment operator. C∀ does not currently have the concept of reference types, so the most appropriate type for the source object in copy constructors and assignment operators is a value type. Appropriate care is taken in the implementation to avoid recursive calls to the copy constructor. The four functions (default constructor, destructor, copy constructor, and assignment operator) are special in that they safely control the state of most objects.

It is possible to define a constructor that takes any combination of parameters to provide additional initialization options. For example, a reasonable extension to the array type would be a constructor that allocates the array to a given initial capacity and initializes the elements of the array to a given `fill` value.

```
void ?{}(Array * arr, int capacity, int fill) {
  arr->len = capacity;
  arr->data = malloc(sizeof(int)*arr->len);
  for (int i = 0; i < arr->len; ++i) {
    arr->data[i] = fill;
  }
}
```

In C∀, constructors are called implicitly in initialization contexts.

```
Array x, y = { 20, 0xdeadbeef }, z = y;
```

Constructor calls look just like C initializers, which allows them to be inserted into legacy C code

with minimal code changes, and also provides a very simple syntax that veteran C programmers are familiar with. One downside of reusing C initialization syntax is that it is not possible to determine whether an object is constructed just by looking at its declaration, since that requires knowledge of whether the type is managed at that point in the program.

This example generates the following code

```
Array x;
?{}(&x);                    // implicit default construct
Array y;
?{}(&y, 20, 0xdeadbeef);    // explicit fill construct
Array z;
?{}(&z, y);                 // copy construct
^?{}(&z);                   // implicit destruct
^?{}(&y);                   // implicit destruct
^?{}(&x);                   // implicit destruct
```

Due to the way that constructor calls are interleaved, it is impossible for `y` to be referenced before it is initialized, except in its own constructor. This loophole is minor and exists in C++ as well. Destructors are implicitly called in reverse declaration-order so that objects with dependencies are destructed before the objects they are dependent on.

2.1.1 Calling Syntax

There are several ways to construct an object in C∀. As previously introduced, every variable is automatically constructed at its definition, which is the most natural way to construct an object.

```
struct A { ... };
void ?{}(A *);
void ?{}(A *, A);
void ?{}(A *, int, int);

A a1;              // default constructed
A a2 = { 0, 0 };   // constructed with 2 ints
A a3 = a1;         // copy constructed
// implicitly destruct a3, a2, a1, in that order
```

Since constructors and destructors are just functions, the second way is to call the function directly.

```
struct A { int a; };
void ?{}(A *);
void ?{}(A *, A);
void ^?{}(A *);

A x;                // implicitly default constructed: ?{}(&x)
A * y = malloc();   // copy construct: ?{}(&y, malloc())

^?{}(&x);   // explicit destroy x, in different order
?{}(&x);    // explicit construct x, second construction
^?{}(y);    // explicit destroy y
?{}(y, x);  // explit construct y from x, second construction
```

```
// implicit ^?{}(&y);
// implicit ^?{}(&x);
```

Calling a constructor or destructor directly is a flexible feature that allows complete control over the management of storage. In particular, constructors double as a placement syntax.

```
struct A { ... };
struct memory_pool { ... };
void ?{}(memory_pool *, size_t);

memory_pool pool = { 1024 };  // create an arena of size 1024

A * a = allocate(&pool);      // allocate from memory pool
?{}(a);                       // construct an A in place

for (int i = 0; i < 10; i++) {
  // reuse storage rather than reallocating
  ^?{}(a);
  ?{}(a);
  // use a ...
}
^?{}(a);
deallocate(&pool, a);         // return to memory pool
```

Finally, constructors and destructors support *operator syntax*. Like other operators in C∀, the function name mirrors the use-case, in that the question marks are placeholders for the first *N* arguments. This syntactic form is similar to the new initialization syntax in C++11, except that it is used in expression contexts, rather than declaration contexts.

```
struct A { ... };
struct B { A a; };

A x, y, * z = &x;
(&x){}          // default construct
(&x){ y }       // copy construct
(&x){ 1, 2, 3 } // construct with 3 arguments
z{ y };         // copy construct x through a pointer
^(&x){}         // destruct

void ?{}(B * b) {
  (&b->a){ 11, 17, 13 };  // construct a member
}
```

Constructor operator syntax has relatively high precedence, requiring parentheses around an address-of expression. Destructor operator syntax is actually an statement, and requires parentheses for symmetry with constructor syntax.

One of these three syntactic forms should appeal to either C or C++ programmers using C∀.

2.1.2 Constructor Expressions

In C∀, it is possible to use a constructor as an expression. Like other operators, the function name `?{}` matches its operator syntax. For example, `(&x){}` calls the default constructor on the variable `x`, and produces `&x` as a result. A key example for this capability is the use of constructor

expressions to initialize the result of a call to `malloc`.

```
struct X { ... };
void ?{}(X *, double);
X * x = malloc(){ 1.5 };
```

In this example, `malloc` dynamically allocates storage and initializes it using a constructor, all before assigning it into the variable `x`. Intuitively, the expression-resolver determines that `malloc` returns some type `T *`, as does the constructor expression since it returns the type of its argument. This type flows outwards to the declaration site where the expected type is known to be `X *`, thus the first argument to the constructor must be `X *`, narrowing the search space.

If this extension is not present, constructing dynamically allocated objects is much more cumbersome, requiring separate initialization of the pointer and initialization of the pointed-to memory.

```
X * x = malloc();
x{ 1.5 };
```

Not only is this verbose, but it is also more error prone, since this form allows maintenance code to easily sneak in between the initialization of `x` and the initialization of the memory that `x` points to. This feature is implemented via a transformation producing the value of the first argument of the constructor, since constructors do not themselves have a return value. Since this transformation results in two instances of the subexpression, care is taken to allocate a temporary variable to hold the result of the subexpression in the case where the subexpression may contain side effects. The previous example generates the following code.

```
struct X *_tmp_ctor;
struct X *x = ?{}(  // construct result of malloc
  _tmp_ctor=malloc_T( // store result of malloc
    sizeof(struct X),
    _Alignof(struct X)
  ),
  1.5
), _tmp_ctor; // produce constructed result of malloc
```

It should be noted that this technique is not exclusive to `malloc`, and allows a user to write a custom allocator that can be idiomatically used in much the same way as a constructed `malloc` call.

While it is possible to use operator syntax with destructors, destructors invalidate their argument, thus operator syntax with destructors is void-typed expression.

2.1.3 Function Generation

In C∀, every type is defined to have the core set of four special functions described previously. Having these functions exist for every type greatly simplifies the semantics of the language, since most operations can simply be defined directly in terms of function calls. In addition to simplifying the definition of the language, it also simplifies the analysis that the translator must perform. If the translator can expect these functions to exist, then it can unconditionally attempt

to resolve them. Moreover, the existence of a standard interface allows polymorphic code to interoperate with new types seamlessly. While automatic generation of assignment functions is present in previous versions of C∀, the the implementation has been largely rewritten to accomodate constructors and destructors.

To mimic the behaviour of standard C, the default constructor and destructor for all of the basic types and for all pointer types are defined to do nothing, while the copy constructor and assignment operator perform a bitwise copy of the source parameter (as in C++). This default is intended to maintain backwards compatibility and performance, by not imposing unexpected operations for a C programmer, as a zero-default behaviour would. However, it is possible for a user to define such constructors so that variables are safely zeroed by default, if desired.

```
void ?{}(int * i) { *i = 0; }
forall(dtype T) void ?{}(T ** p) { *p = 0; }  // any pointer type
void f() {
  int x;    // initialized to 0
  int * p;  // initialized to 0
}
```

There are several options for user-defined types: structures, unions, and enumerations. To aid in ease of use, the standard set of four functions is automatically generated for a user-defined type after its definition is completed. By auto-generating these functions, it is ensured that legacy C code continues to work correctly in every context where C∀ expects these functions to exist, since they are generated for every complete type. As well, these functions are always generated, since they may be needed by polymorphic functions. With that said, the generated functions are not called implicitly unless they are non-trivial, and are never exported, making it simple for the optimizer to strip them away when they are not used.

The generated functions for enumerations are the simplest. Since enumerations in C are essentially just another integral type, the generated functions behave in the same way that the built-in functions for the basic types work. For example, given the enumeration

```
enum Colour {
  R, G, B
};
```

The following functions are automatically generated.

```
void ?{}(enum Colour *_dst){
  // default constructor does nothing
}
void ?{}(enum Colour *_dst, enum Colour _src){
  *_dst=_src;  // bitwise copy
}
void ^?{}(enum Colour *_dst){
  // destructor does nothing
}
enum Colour ?=?(enum Colour *_dst, enum Colour _src){
  return *_dst=_src; // bitwise copy
}
```

In the future, C∀ will introduce strongly-typed enumerations, like those in C++, wherein enumerations create a new type distinct from **int** so that integral values require an explicit cast to be

stored in an enumeration variable. The existing generated routines are sufficient to express this restriction, since they are currently set up to take in values of that enumeration type. Changes related to this feature only need to affect the expression resolution phase, where more strict rules will be applied to prevent implicit conversions from integral types to enumeration types, but should continue to permit conversions from enumeration types to **int**. In this way, it is still possible to add an **int** to an enumeration, but the resulting value is an **int**, meaning it cannot be reassigned to an enumeration without a cast.

For structures, the situation is more complicated. Given a structure `S` with members $\texttt{M}_0$, $\texttt{M}_1$, ... $\texttt{M}_{N-1}$, each function `f` in the standard set calls $\texttt{f(s->M}_i\texttt{, ...)}$ for each i. That is, a default constructor for `S` default constructs the members of `S`, the copy constructor copy constructs them, and so on. For example, given the structure definition

```
struct A {
  B b;
  C c;
}
```

The following functions are implicitly generated.

```
void ?{}(A * this) {
  ?{}(&this->b);  // default construct each field
  ?{}(&this->c);
}
void ?{}(A * this, A other) {
  ?{}(&this->b, other.b);  // copy construct each field
  ?{}(&this->c, other.c);
}
A ?=?(A * this, A other) {
  ?=?(&this->b, other.b);  // assign each field
  ?=?(&this->c, other.c);
}
void ^?{}(A * this) {
  ^?{}(&this->c);  // destruct each field
  ^?{}(&this->b);
}
```

It is important to note that the destructors are called in reverse declaration order to prevent conflicts in the event there are dependencies among members.

In addition to the standard set, a set of *field constructors* is also generated for structures. The field constructors are constructors that consume a prefix of the structure's member-list. That is, N constructors are built of the form **void** $\texttt{?\{\}(S *, T}_{\texttt{M}_0}\texttt{)}$, **void** $\texttt{?\{\}(S *, T}_{\texttt{M}_0}\texttt{, T}_{\texttt{M}_1}\texttt{)}$, ..., **void** $\texttt{?\{\}(S *, T}_{\texttt{M}_0}\texttt{, T}_{\texttt{M}_1}\texttt{, ..., T}_{\texttt{M}_{N-1}}\texttt{)}$, where members are copy constructed if they have a corresponding positional argument and are default constructed otherwise. The addition of field constructors allows structures in C$\forall$ to be used naturally in the same ways as used in C (*i.e.*, to initialize any prefix of the structure), *e.g.*, `A a0 = { b }, a1 = { b, c }`. Extending the previous example, the following constructors are implicitly generated for `A`.

```
void ?{}(A * this, B b) {
  ?{}(&this->b, b);
  ?{}(&this->c);
}
```

```
void ?{}(A * this, B b, C c) {
  ?{}(&this->b, b);
  ?{}(&this->c, c);
}
```

For unions, the default constructor and destructor do nothing, as it is not obvious which member, if any, should be constructed. For copy constructor and assignment operations, a bitwise `memcpy` is applied. In standard C, a union can also be initialized using a value of the same type as its first member, and so a corresponding field constructor is generated to perform a bitwise `memcpy` of the object. An alternative to this design is to always construct and destruct the first member of a union, to match with the C semantics of initializing the first member of the union. This approach ultimately feels subtle and unsafe. Another option is to, like C++, disallow unions from containing members that are themselves managed types. This restriction is a reasonable approach from a safety standpoint, but is not very C-like. Since the primary purpose of a union is to provide low-level memory optimization, it is assumed that the user has a certain level of maturity. It is therefore the responsibility of the user to define the special functions explicitly if they are appropriate, since it is impossible to accurately predict the ways that a union is intended to be used at compile-time.

For example, given the union

```
union X {
  Y y;
  Z z;
};
```

The following functions are automatically generated.

```
void ?{}(union X *_dst){  // default constructor
}
void ?{}(union X *_dst, union X _src){  // copy constructor
  __builtin_memcpy(_dst, &_src, sizeof(union X ));
}
void ^?{}(union X *_dst){  // destructor
}
union X ?=?(union X *_dst, union X _src){  // assignment
  __builtin_memcpy(_dst, &_src, sizeof(union X));
  return _src;
}
void ?{}(union X *_dst, struct Y src){  // construct first field
  __builtin_memcpy(_dst, &src, sizeof(struct Y));
}
```

In C++11, unions may have managed members, with the caveat that if there are any members with a user-defined operation, then that operation is not implicitly defined, forcing the user to define the operation if necessary. This restriction could easily be added into C∀ once *deleted* functions are added.

2.1.4 Using Constructors and Destructors

Implicitly generated constructor and destructor calls ignore the outermost type qualifiers, *e.g.*, **`const`** and **`volatile`**, on a type by way of a cast on the first argument to the function. For

example,

```
struct S { int i; };
void ?{}(S *, int);
void ?{}(S *, S);

const S s = { 11 };
volatile S s2 = s;
```

Generates the following code

```
const struct S s;
?{}((struct S *)&s, 11);
volatile struct S s2;
?{}((struct S *)&s2, s);
```

Here, `&s` and `&s2` are cast to unqualified pointer types. This mechanism allows the same constructors and destructors to be used for qualified objects as for unqualified objects. This rule applies only to implicitly generated constructor calls. Hence, explicitly re-initializing qualified objects with a constructor requires an explicit cast.

As discussed in Section 1.1.1, compound literals create unnamed objects. This mechanism can continue to be used seamlessly in C∀ with managed types to create temporary objects. The object created by a compound literal is constructed using the provided brace-enclosed initializer-list, and is destructed at the end of the scope it is used in. For example,

```
struct A { int x; };
void ?{}(A *, int, int);
{
  int x = (A){ 10, 20 }.x;
}
```

is equivalent to

```
struct A { int x, y; };
void ?{}(A *, int, int);
{
  A _tmp;
  ?{}(&_tmp, 10, 20);
  int x = _tmp.x;
  ^?{}(&tmp);
}
```

Unlike C++, C∀ provides an escape hatch that allows a user to decide at an object's definition whether it should be managed or not. An object initialized with `@=` is guaranteed to be initialized like a C object, and has no implicit destructor call. This feature provides all of the freedom that C programmers are used to having to optimize a program, while maintaining safety as a sensible default.

```
struct A { int * x; };
// RAII
void ?{}(A * a) { a->x = malloc(sizeof(int)); }
void ^?{}(A * a) { free(a->x); }

A a1;           // managed
```

```
A a2 @= { 0 };  // unmanaged
```

In this example, `a1` is a managed object, and thus is default constructed and destructed at the start/end of `a1`'s lifetime, while `a2` is an unmanaged object and is not implicitly constructed or destructed. Instead, `a2->x` is initialized to `0` as if it were a C object, because of the explicit initializer.

In addition to freedom, `@=` provides a simple path for migrating legacy C code to C∀, in that objects can be moved from C-style initialization to C∀ gradually and individually. It is worth noting that the use of unmanaged objects can be tricky to get right, since there is no guarantee that the proper invariants are established on an unmanaged object. It is recommended that most objects be managed by sensible constructors and destructors, except where absolutely necessary, such as memory-mapped devices, trigger devices, I/O controllers, etc.

When a user declares any constructor or destructor, the corresponding intrinsic/generated function and all field constructors for that type are hidden, so that they are not found during expression resolution until the user-defined function goes out of scope. Furthermore, if the user declares any constructor, then the intrinsic/generated default constructor is also hidden, precluding default construction. These semantics closely mirror the rule for implicit declaration of constructors in C++, wherein the default constructor is implicitly declared if there is no user-declared constructor [13, p. 186].

```
struct S { int x, y; };

void f() {
  S s0, s1 = { 0 }, s2 = { 0, 2 }, s3 = s2;  // okay
  {
    void ?{}(S * s, int i) { s->x = i*2; } // locally hide autogen ctors
    S s4;  // error, no default constructor
    S s5 = { 3 };  // okay, local constructor
    S s6 = { 4, 5 };  // error, no field constructor
    S s7 = s5; // okay
  }
  S s8, s9 = { 6 }, s10 = { 7, 8 }, s11 = s10;  // okay
}
```

In this example, the inner scope declares a constructor from **`int`** to `S`, which hides the default constructor and field constructors until the end of the scope.

When defining a constructor or destructor for a structure `S`, any members that are not explicitly constructed or destructed are implicitly constructed or destructed automatically. If an explicit call is present, then that call is taken in preference to any implicitly generated call. A consequence of this rule is that it is possible, unlike C++, to precisely control the order of construction and destruction of sub-objects on a per-constructor basis, whereas in C++ sub-object initialization and destruction is always performed based on the declaration order.

```
struct A {
  B w, x, y, z;
};
void ?{}(A * a, int i) {
  (&a->x){ i };
  (&a->z){ a->y };
```

```
}
```

Generates the following

```
void ?{}(A * a, int i) {
  (&a->w){};   // implicit default ctor
  (&a->y){};   // implicit default ctor
  (&a->x){ i };
  (&a->z){ a->y };
}
```

Finally, it is illegal for a sub-object to be explicitly constructed for the first time after it is used for the first time. If the translator cannot be reasonably sure that an object is constructed prior to its first use, but is constructed afterward, an error is emitted. More specifically, the translator searches the body of a constructor to ensure that every sub-object is initialized.

```
void ?{}(A * a, double x) {
  f(a->x);
  (&a->x){ (int)x }; // error, used uninitialized on previous line
}
```

However, if the translator sees a sub-object used within the body of a constructor, but does not see a constructor call that uses the sub-object as the target of a constructor, then the translator assumes the object is to be implicitly constructed (copy constructed in a copy constructor and default constructed in any other constructor). To override this rule, `@=` can be used to force the translator to trust the programmer's discretion. This form of `@=` is not yet implemented.

```
void ?{}(A * a) {
  // default constructs all members
  f(a->x);
}

void ?{}(A * a, A other) {
  // copy constructs all members
  f(a->y);
}

void ?{}(A * a, int x) {
  // object forwarded to another constructor,
  // does not implicitly construct any members
  (&a){};
}

void ^?{}(A * a) {
  ^(&a->x){}; // explicit destructor call
} // z, y, w implicitly destructed, in this order
```

If at any point, the `this` parameter is passed directly as the target of another constructor, then it is assumed the other constructor handles the initialization of all of the object's members and no implicit constructor calls are added to the current constructor.

Despite great effort, some forms of C syntax do not work well with constructors in C∀. In particular, constructor calls cannot contain designations (see 1.1.1), since this is equivalent to allowing designations on the arguments to arbitrary function calls.

```
// all legal forward declarations in C
void f(int, int, int);
void f(int a, int b, int c);
void f(int b, int c, int a);
void f(int c, int a, int b);
void f(int x, int y, int z);

f(b:10, a:20, c:30);  // which parameter is which?
```

In C, function prototypes are permitted to have arbitrary parameter names, including no names at all, which may have no connection to the actual names used at function definition. Furthermore, a function prototype can be repeated an arbitrary number of times, each time using different names. As a result, it was decided that any attempt to resolve designated function calls with C's function prototype rules would be brittle, and thus it is not sensible to allow designations in constructor calls.

In addition, constructor calls do not support unnamed nesting.

```
struct B { int x; };
struct C { int y; };
struct A { B b; C c; };
void ?{}(A *, B);
void ?{}(A *, C);

A a = {
  { 10 },  // construct B? - invalid
};
```

In C, nesting initializers means that the programmer intends to initialize sub-objects with the nested initializers. The reason for this omission is to both simplify the mental model for using constructors, and to make initialization simpler for the expression resolver. If this were allowed, it would be necessary for the expression resolver to decide whether each argument to the constructor call could initialize to some argument in one of the available constructors, making the problem highly recursive and potentially much more expensive. That is, in the previous example the line marked as an error could mean construct using `?{}(A *, B)` or with `?{}(A *, C)`, since the inner initializer `{ 10 }` could be taken as an intermediate object of type `B` or `C`. In practice, however, there could be many objects that can be constructed from a given **`int`** (or, indeed, any arbitrary parameter list), and thus a complete solution to this problem would require fully exploring all possibilities.

More precisely, constructor calls cannot have a nesting depth greater than the number of array dimensions in the type of the initialized object, plus one. For example,

```
struct A;
void ?{}(A *, int);
void ?{}(A *, A, A);

A a1[3] = { { 3 }, { 4 }, { 5 } };
A a2[2][2] = {
  { { 9 }, { 10 } },  // a2[0]
  { {14 }, { 15 } }   // a2[1]
};
```

```
A a3[4] = { // 1 dimension => max depth 2
  { { 11 }, { 12 } },  // error, three levels deep
  { 80 }, { 90 }, { 100 }
}
```

The body of A has been omitted, since only the constructor interfaces are important.

It should be noted that unmanaged objects, i.e. objects that have only trivial constructors, can still make use of designations and nested initializers in C∀. It is simple to overcome this limitation for managed objects by making use of compound literals, so that the arguments to the constructor call are explicitly typed.

```
struct B { int x; };
struct C { int y; };
struct A { B b; C c; };
void ?{}(A *, B);
void ?{}(A *, C);

A a = {
  (C){ 10 } // disambiguate with compound literal
};
```

2.1.5 Implicit Destructors

Destructors are automatically called at the end of the block in which the object is declared. In addition to this, destructors are automatically called when statements manipulate control flow to leave a block in which the object is declared, *e.g.*, with return, break, continue, and goto statements. The example below demonstrates a simple routine with multiple return statements.

```
struct A;
void ^?{}(A *);

void f(int i) {
  A x;  // construct x
  {
    A y; // construct y
    {
      A z; // construct z
      {
        if (i == 0) return; // destruct x, y, z
      }
      if (i == 1) return; // destruct x, y, z
    } // destruct z
    if (i == 2) return; // destruct x, y
  } // destruct y
} // destruct x
```

The next example illustrates the use of simple continue and break statements and the manner that they interact with implicit destructors.

```
for (int i = 0; i < 10; i++) {
  A x;
```

```
  if (i == 2) {
    continue;  // destruct x
  } else if (i == 3) {
    break;     // destruct x
  }
} // destruct x
```

Since a destructor call is automatically inserted at the end of the block, nothing special needs to happen to destruct `x` in the case where control reaches the end of the loop. In the case where `i` is `2`, the continue statement runs the loop update expression and attempts to begin the next iteration of the loop. Since continue is a C statement, which does not understand destructors, it is transformed into a **`goto`** statement that branches to the end of the loop, just before the block's destructors, to ensure that `x` is destructed. When `i` is `3`, the break statement moves control to just past the end of the loop. Unlike the previous case, the destructor for `x` cannot be reused, so a destructor call for `x` is inserted just before the break statement.

C∀ also supports labeled break and continue statements, which allow more precise manipulation of control flow. Labeled break and continue allow the programmer to specify which control structure to target by using a label attached to a control structure.

```
L1: for (int i = 0; i < 10; i++) {
  A x;
  for (int j = 0; j < 10; j++) {
    A y;
    if (i == 1) {
      continue L1; // destruct y
    } else if (i == 2) {
      break L1;    // destruct x,y
    }
  } // destruct y
} // destruct X
```

The statement **`continue`** `L1` begins the next iteration of the outer for-loop. Since the semantics of continue require the loop update expression to execute, control branches to the end of the outer for loop, meaning that the block destructor for `x` can be reused, and it is only necessary to generate the destructor for `y`. Break, on the other hand, requires jumping out of both loops, so the destructors for both `x` and `y` are generated and inserted before the **`break`** `L1` statement.

Finally, an example which demonstrates goto. Since goto is a general mechanism for jumping to different locations in the program, a more comprehensive approach is required. For each goto statement G and each target label L, let S_G be the set of all managed variables alive at G, and let S_L be the set of all managed variables alive at L. If at any G, $S_L \setminus S_G = \emptyset$, then the translator emits an error, because control flow branches from a point where the object is not yet live to a point where it is live, skipping the object's constructor. Then, for every G, the destructors for each variable in the set $S_G \setminus S_L$ is inserted directly before G, which ensures each object that is currently live at G, but not at L, is destructed before control branches.

```
int i = 0;
{
  L0: ;      // S_L0 = { x }
    A y;
  L1: ;      // S_L1 = { x }
```

```
    A x;
  L2: ;      // S_L2 = { y, x }
    if (i == 0) {
      ++i;
      goto L1;     // S_G = { y, x }
      // S_G-S_L1 = { x } => destruct x
    } else if (i == 1) {
      ++i;
      goto L2;     // S_G = { y, x }
      // S_G-S_L2 = {} => destruct nothing
    } else if (i == 2) {
      ++i;
      goto L3;     // S_G = { y, x }
      // S_G-S_L3 = {}
    } else if (false) {
      ++i;
      A z;
      goto L3;     // S_G = { z, y, x }
      // S_G-S_L3 = { z } => destruct z
    } else {
      ++i;
      goto L4;     // S_G = { y, x }
      // S_G-S_L4 = { y, x } => destruct y, x
    }
  L3: ;     // S_L3 = { y, x }
    goto L2;       // S_G = { y, x }
    // S_G-S_L2 = {}
}
L4: ;  // S_L4 = {}
if (i == 4) {
  goto L0;         // S_G = {}
  // S_G-S_L0 = {}
}
```

All break and continue statements are implemented in C∀ in terms of goto statements, so the more constrained forms are precisely governed by these rules.

The next example demonstrates the error case.

```
{
    goto L1; // S_G = {}
    // S_L1-S_G = { y } => error
    A y;
  L1: ; // S_L1 = { y }
    A x;
  L2: ; // S_L2 = { y, x }
}
goto L2; // S_G = {}
// S_L2-S_G = { y, x } => error
```

While C∀ supports the GCC computed-goto extension, the behaviour of managed objects in combination with computed-goto is undefined.

```
void f(int val) {
  void * l = val == 0 ? &&L1 : &&L2;
```

```
  {
      A x;
    L1: ;
      goto *l;  // branches differently depending on argument
  }
  L2: ;
}
```

Likewise, destructors are not executed at scope-exit due to a computed-goto in C++, as of g++ version 6.2.

2.1.6 Implicit Copy Construction

When a function is called, the arguments supplied to the call are subject to implicit copy construction (and destruction of the generated temporary), and the return value is subject to destruction. When a value is returned from a function, the copy constructor is called to pass the value back to the call site. Exempt from these rules are intrinsic and built-in functions. It should be noted that unmanaged objects are subject to copy constructor calls when passed as arguments to a function or when returned from a function, since they are not the *target* of the copy constructor call. That is, since the parameter is not marked as an unmanaged object using `@=`, it is copy constructed if it is returned by value or passed as an argument to another function, so to guarantee consistent behaviour, unmanaged objects must be copy constructed when passed as arguments. These semantics are important to bear in mind when using unmanaged objects, and could produce unexpected results when mixed with objects that are explicitly constructed.

```
struct A { ... };
void ?{}(A *);
void ?{}(A *, A);
void ^?{}(A *);

A identity(A x) { // pass by value => need local copy
  return x;       // return by value => make call-site copy
}

A y, z @= {};
identity(y);  // copy construct y into x
identity(z);  // copy construct z into x
```

Note that unmanaged argument `z` is logically copy constructed into managed parameter `x`; however, the translator must copy construct into a temporary variable to be passed as an argument, which is also destructed after the call. A compiler could by-pass the argument temporaries since it is in control of the calling conventions and knows exactly where the called-function's parameters live.

This generates the following

```
struct A f(struct A x){
  struct A _retval_f;     // return value
  ?{}((&_retval_f), x);  // copy construct return value
  return _retval_f;
}
```

```
struct A y;
?{}(&y);                    // default construct
struct A z = { 0 };         // C default

struct A _tmp_cp1;          // argument 1
struct A _tmp_cp_ret0;      // return value
_tmp_cp_ret0=f(
  (?{}(&_tmp_cp1, y) , _tmp_cp1)  // argument is a comma expression
), _tmp_cp_ret0;            // return value for cascading
^?{}(&_tmp_cp_ret0);        // destruct return value
^?{}(&_tmp_cp1);            // destruct argument 1

struct A _tmp_cp2;          // argument 1
struct A _tmp_cp_ret1;      // return value
_tmp_cp_ret1=f(
  (?{}(&_tmp_cp2, z), _tmp_cp2)  // argument is a common expression
), _tmp_cp_ret1;            // return value for cascading
^?{}(&_tmp_cp_ret1);        // destruct return value
^?{}(&_tmp_cp2);            // destruct argument 1
^?{}(&y);
```

A special syntactic form, such as a variant of `@=`, can be implemented to specify at the call site that an argument should not be copy constructed, to regain some control for the C programmer.

```
identity(z@);  // do not copy construct argument
               // - will copy construct/destruct return value
A@ identity_nocopy(A @ x) {  // argument not copy constructed or destructed
  return x;  // not copy constructed
             // return type marked @ => not destructed
}
```

It should be noted that reference types will allow specifying that a value does not need to be copied, however reference types do not provide a means of preventing implicit copy construction from uses of the reference, so the problem is still present when passing or returning the reference by value.

Adding implicit copy construction imposes the additional runtime cost of the copy constructor for every argument and return value in a function call. This cost is necessary to maintain appropriate value semantics when calling a function. In the future, return-value-optimization (RVO) can be implemented for C∀ to elide unnecessary copy construction and destruction of temporary objects. This cost is not present for types with trivial copy constructors and destructors.

A known issue with this implementation is that the argument and return value temporaries are not guaranteed to have the same address for their entire lifetimes. In the previous example, since `_retval_f` is allocated and constructed in `f`, then returned by value, the internal data is bitwise copied into the caller's stack frame. This approach works out most of the time, because typically destructors need to only access the fields of the object and recursively destroy. It is currently the case that constructors and destructors that use the `this` pointer as a unique identifier to store data externally do not work correctly for return value objects. Thus, it is currently not safe to rely on an object's `this` pointer to remain constant throughout execution of the program.

```
A * external_data[32];
int ext_count;
struct A;
void ?{}(A * a) {
  // ...
  external_data[ext_count++] = a;
}
void ^?{}(A * a) {
  for (int i = 0; i < ext_count) {
    if (a == external_data[i]) { // may never be true
      // ...
    }
  }
}

A makeA() {
  A x;  // stores &x in external_data
  return x;
}
makeA();  // return temporary has a different address than x
// equivalent to:
//   A _tmp;
//   _tmp = makeA(), _tmp;
//   ^?{}(&_tmp);
```

In the above example, a global array of pointers is used to keep track of all of the allocated `A` objects. Due to copying on return, the current object being destructed does not exist in the array if an `A` object is ever returned by value from a function, such as in `makeA`.

This problem could be solved in the translator by changing the function signatures so that the return value is moved into the parameter list. For example, the translator could restructure the code like so

```
void f(struct A x, struct A * _retval_f){
  ?{}(_retval_f, x);  // construct directly into caller's stack frame
}

struct A y;
?{}(&y);
struct A z = { 0 };

struct A _tmp_cp1;     // argument 1
struct A _tmp_cp_ret0; // return value
f((?{}(&_tmp_cp1, y) , _tmp_cp1), &_tmp_cp_ret0), _tmp_cp_ret0;
^?{}(&_tmp_cp_ret0);   // return value
^?{}(&_tmp_cp1);       // argument 1
```

This transformation provides `f` with the address of the return variable so that it can be constructed into directly. It is worth pointing out that this kind of signature rewriting already occurs in polymorphic functions that return by value, as discussed in [3]. A key difference in this case is that every function would need to be rewritten like this, since types can switch between managed and unmanaged at different scope levels, *e.g.*,

```
struct A { int v; };
A x; // unmanaged, since only trivial constructors are available
{
  void ?{}(A * a) { ... }
  void ^?{}(A * a) { ... }
  A y; // managed
}
A z; // unmanaged
```

Hence there is not enough information to determine at function declaration whether a type is managed or not, and thus it is the case that all signatures have to be rewritten to account for possible copy constructor and destructor calls. Even with this change, it would still be possible to declare backwards compatible function prototypes with an **extern** "C" block, which allows for the definition of C-compatible functions within C∀ code, however this would require actual changes to the way code inside of an **extern** "C" function is generated as compared with normal code generation. Furthermore, it is not possible to overload C functions, so using **extern** "C" to declare functions is of limited use.

It would be possible to regain some control by adding an attribute to structures that specifies whether they can be managed or not (perhaps *manageable* or *unmanageable*), and to emit an error in the case that a constructor or destructor is declared for an unmanageable type. Ideally, structures should be manageable by default, since otherwise the default case becomes more verbose. This means that in general, function signatures would have to be rewritten, and in a select few cases the signatures would not be rewritten.

```
__attribute__((manageable)) struct A { ... };   // can declare ctors
__attribute__((unmanageable)) struct B { ... }; // cannot declare ctors
struct C { ... };                               // can declare ctors

A f();  // rewritten void f(A *);
B g();  // not rewritten
C h();  // rewritten void h(C *);
```

An alternative is to make the attribute *identifiable*, which states that objects of this type use the `this` parameter as an identity. This strikes more closely to the visible problem, in that only types marked as identifiable would need to have the return value moved into the parameter list, and every other type could remain the same. Furthermore, no restrictions would need to be placed on whether objects can be constructed.

```
__attribute__((identifiable)) struct A { ... };  // can declare ctors
struct B { ... };                                 // can declare ctors

A f();  // rewritten void f(A *);
B g();  // not rewritten
```

Ultimately, both of these are patchwork solutions. Since a real compiler has full control over its calling conventions, it can seamlessly allow passing the return parameter without outwardly changing the signature of a routine. As such, it has been decided that this issue is not currently a priority and will be fixed when a full C∀ compiler is implemented.

2.2 Implementation

2.2.1 Array Initialization

Arrays are a special case in the C type-system. Type checking largely ignores size information for C arrays, making it impossible to write a standalone C∀ function that constructs or destructs an array, while maintaining the standard interface for constructors and destructors. Instead, C∀ defines the initialization and destruction of an array recursively. That is, when an array is defined, each of its elements is constructed in order from element 0 up to element $n-1$. When an array is to be implicitly destructed, each of its elements is destructed in reverse order from element $n-1$ down to element 0. As in C, it is possible to explicitly provide different initializers for each element of the array through array initialization syntax. In this case, each of the initializers is taken in turn to construct a subsequent element of the array. If too many initializers are provided, only the initializers up to N are actually used. If too few initializers are provided, then the remaining elements are default constructed.

For example, given the following code.

```
struct X {
  int x, y, z;
};
void f() {
  X x[10] = { { 1, 2, 3 }, { 4 }, { 7, 8 } };
}
```

The following code is generated for `f`.

```
void f(){
  struct X x[((long unsigned int )10)];
  // construct x
  {
    int _index0 = 0;
    // construct with explicit initializers
    {
      if (_index0<10) ?{}(&x[_index0], 1, 2, 3);
      ++_index0;
      if (_index0<10) ?{}(&x[_index0], 4);
      ++_index0;
      if (_index0<10) ?{}(&x[_index0], 7, 8);
      ++_index0;
    }

    // default construct remaining elements
    for (;_index0<10;++_index0) {
      ?{}(&x[_index0]);
    }
  }
  // destruct x
  {
    int _index1 = 10-1;
    for (;_index1>=0;--_index1) {
      ^?{}(&x[_index1]);
    }
```

```
  }
}
```

Multidimensional arrays require more complexity. For example, a two dimensional array

```
void g() {
  X x[10][10] = {
    { { 1, 2, 3 }, { 4 } }, // x[0]
    { { 7, 8 } }            // x[1]
  };
}
```

Generates the following

```
void g(){
  struct X x[10][10];
  // construct x
  {
    int _index0 = 0;
    for (;_index0<10;++_index0) {
      {
        int _index1 = 0;
        // construct with explicit initializers
        {
          switch ( _index0 ) {
            case 0:
              // construct first array
              if ( _index1<10 ) ?{}(&x[_index0][_index1], 1, 2, 3);
              ++_index1;
              if ( _index1<10 ) ?{}(&x[_index0][_index1], 4);
              ++_index1;
              break;
            case 1:
              // construct second array
              if ( _index1<10 ) ?{}(&x[_index0][_index1], 7, 8);
              ++_index1;
              break;
          }
        }
        // default construct remaining elements
        for (;_index1<10;++_index1) {
            ?{}(&x[_index0][_index1]);
        }
      }
    }
  }
  // destruct x
  {
    int _index2 = 10-1;
    for (;_index2>=0;--_index2) {
      {
        int _index3 = 10-1;
        for (;_index3>=0;--_index3) {
            ^?{}(&x[_index2][_index3]);
        }
```

```
        }
      }
    }
}
```

2.2.2 Global Initialization

In standard C, global variables can only be initialized to compile-time constant expressions, which places strict limitations on the programmer's ability to control the default values of objects. In C∀, constructors and destructors are guaranteed to be run on global objects, allowing arbitrary code to be run before and after the execution of the main routine. By default, objects within a translation unit are constructed in declaration order, and destructed in the reverse order. The default order of construction of objects amongst translation units is unspecified. It is, however, guaranteed that any global objects in the standard library are initialized prior to the initialization of any object in a user program.

This feature is implemented in the C∀ translator by grouping every global constructor call into a function with the GCC attribute *constructor*, which performs most of the heavy lifting [6, 6.31.1]. A similar function is generated with the *destructor* attribute, which handles all global destructor calls. At the time of writing, initialization routines in the library are specified with priority *101*, which is the highest priority level that GCC allows, whereas initialization routines in the user's code are implicitly given the default priority level, which ensures they have a lower priority than any code with a specified priority level. This mechanism allows arbitrarily complicated initialization to occur before any user code runs, making it possible for library designers to initialize their modules without requiring the user to call specific startup or tear-down routines.

For example, given the following global declarations.

```
struct X {
  int y, z;
};
void ?{}(X *);
void ?{}(X *, int, int);
void ^?{}(X *);

X a;
X b = { 10, 3 };
```

The following code is generated.

```
__attribute__ ((constructor)) static void _init_global_ctor(void){
  ?{}(&a);
  ?{}(&b, 10, 3);
}
__attribute__ ((destructor)) static void _destroy_global_ctor(void){
  ^?{}(&b);
  ^?{}(&a);
}
```

GCC provides an attribute `init_priority` in C++, which allows specifying the relative priority for initialization of global objects on a per-object basis. A similar attribute can be imple-

mented in C∀ by pulling marked objects into global constructor/destructor-attribute functions with the specified priority. For example,

```
struct A { ... };
void ?{}(A *, int);
void ^?{}(A *);
__attribute__((init_priority(200))) A x = { 123 };
```

would generate

```
A x;
__attribute__((constructor(200))) __init_x() {
  ?{}(&x, 123);  // construct x with priority 200
}
__attribute__((destructor(200))) __destroy_x() {
  ?{}(&x);       // destruct x with priority 200
}
```

2.2.3 Static Local Variables

In standard C, it is possible to mark variables that are local to a function with the **`static`** storage class. Unlike normal local variables, a **`static`** local variable is defined to live for the entire duration of the program, so that each call to the function has access to the same variable with the same address and value as it had in the previous call to the function. Much like global variables, **`static`** variables can only be initialized to a *compile-time constant value* so that a compiler is able to create storage for the variable and initialize it at compile-time.

Yet again, this rule is too restrictive for a language with constructors and destructors. Since the initializer expression is not necessarily a compile-time constant and can depend on the current execution state of the function, C∀ modifies the definition of a **`static`** local variable so that objects are guaranteed to be live from the time control flow reaches their declaration, until the end of the program. Since standard C does not allow access to a **`static`** local variable before the first time control flow reaches the declaration, this change does not preclude any valid C code. Local objects with **`static`** storage class are only implicitly constructed and destructed once for the duration of the program. The object is constructed when its declaration is reached for the first time. The object is destructed once at the end of the program.

Construction of **`static`** local objects is implemented via an accompanying **`static`** `bool` variable, which records whether the variable has already been constructed. A conditional branch checks the value of the companion `bool`, and if the variable has not yet been constructed then the object is constructed. The object's destructor is scheduled to be run when the program terminates using `atexit` [2], and the companion `bool`'s value is set so that subsequent invocations of the function do not reconstruct the object. Since the parameter to `atexit` is a parameter-less function, some additional tweaking is required. First, the **`static`** variable must be hoisted up to global scope and uniquely renamed to prevent name clashes with other global objects. If necessary, a local structure may need to be hoisted, as well. Second, a function is built that calls the

[2] When using the dynamic linker, it is possible to dynamically load and unload a shared library. Since glibc 2.2.3 [10], functions registered with `atexit` within the shared library are called when unloading the shared library. As such, static local objects can be destructed using this mechanism even in shared libraries on Linux systems.

destructor for the newly hoisted variable. Finally, the newly generated function is registered with `atexit`, instead of registering the destructor directly. Since `atexit` calls functions in the reverse order in which they are registered, **`static`** local variables are guaranteed to be destructed in the reverse order that they are constructed, which may differ between multiple executions of the same program. Extending the previous example

```
int f(int x) {
  static X a;
  static X b = { x, x };  // depends on parameter value
  static X c = b;         // depends on local variable
}
```

Generates the following.

```
static struct X a_static_var0;
static void __a_dtor_atexit0(void){
  ((void)^?{}(((struct X *)(&a_static_var0))));
}
static struct X b_static_var1;
static void __b_dtor_atexit1(void){
  ((void)^?{}(((struct X *)(&b_static_var1))));
}
static struct X c_static_var2;
static void __c_dtor_atexit2(void){
  ((void)^?{}(((struct X *)(&c_static_var2))));
}
int f(int x){
  int _retval_f;
  __attribute__ ((unused)) static void *_dummy0;
  static _Bool __a_uninitialized = 1;
  if ( __a_uninitialized ) {
    ((void)?{}(((struct X *)(&a_static_var0))));
    ((void)(__a_uninitialized=0));
    ((void)atexit(__a_dtor_atexit0));
  }

  __attribute__ ((unused)) static void *_dummy1;
  static _Bool __b_uninitialized = 1;
  if ( __b_uninitialized ) {
    ((void)?{}(((struct X *)(&b_static_var1)), x, x));
    ((void)(__b_uninitialized=0));
    ((void)atexit(__b_dtor_atexit1));
  }

  __attribute__ ((unused)) static void *_dummy2;
  static _Bool __c_uninitialized = 1;
  if ( __c_uninitialized ) {
    ((void)?{}(((struct X *)(&c_static_var2)), b_static_var1));
    ((void)(__c_uninitialized=0));
    ((void)atexit(__c_dtor_atexit2));
  }
}
```

This implementation comes at the runtime cost of an additional branch for every **`static`**

local variable, each time the function is called. Since initializers are not required to be compile-time constant expressions, they can involve global variables, function arguments, function calls, etc. As a direct consequence, **static** local variables cannot be initialized with an attribute-constructor routines like global variables can. However, in the case where the variable is unmanaged and has a compile-time constant initializer, a C-compliant initializer is generated and the additional cost is not present. C++ shares the same semantics for its **static** local variables.

2.2.4 Polymorphism

As mentioned in section 1.1.3, C∀ currently has 3 type-classes that are used to designate polymorphic data types: **otype**, **dtype**, and **ftype**. In previous versions of C∀, **otype** was syntactic sugar for **dtype** with known size/alignment information and an assignment function. That is,

```
forall(otype T)
void f(T);
```

was equivalent to

```
forall(dtype T | sized(T) | { T ?=?(T *, T); })
void f(T);
```

This allows easily specifying constraints that are common to all complete object-types very simply.

Now that C∀ has constructors and destructors, more of a complete object's behaviour can be specified than was previously possible. As such, **otype** has been augmented to include assertions for a default constructor, copy constructor, and destructor. That is, the previous example is now equivalent to

```
forall(dtype T | sized(T) |
  { T ?=?(T *, T); void ?{}(T *); void ?{}(T *, T); void ^?{}(T *); })
void f(T);
```

These additions allow `f`'s body to create and destroy objects of type `T`, and pass objects of type `T` as arguments to other functions, following the normal C∀ rules. A point of note here is that objects can be missing default constructors (and eventually other functions through deleted functions), so it is important for C∀ programmers to think carefully about the operations needed by their function, as to not over-constrain the acceptable parameter types and prevent potential reuse.

These additional assertion parameters impose a runtime cost on all managed temporary objects created in polymorphic code, even those with trivial constructors and destructors. This cost is necessary because polymorphic code does not know the actual type at compile-time, due to separate compilation. Since trivial constructors and destructors either do not perform operations or are simply bit-wise copy operations, the imposed cost is essentially the cost of the function calls.

2.3 Summary

When creating a new object of a managed type, it is guaranteed that a constructor is be called to initialize the object at its definition point, and is destructed when the object's lifetime ends. Destructors are called in the reverse order of construction.

Every argument passed to a function is copy constructed into a temporary object that is passed by value to the functions and destructed at the end of the statement. Function return values are copy constructed inside the function at the return statement, passed by value to the call-site, and destructed at the call-site at the end of the statement.

Every complete object type has a default constructor, copy constructor, assignment operator, and destructor. To accomplish this, these functions are generated as appropriate for new types. User-defined functions shadow built-in and automatically generated functions, so it is possible to specialize the behaviour of a type. Furthermore, default constructors and aggregate field constructors are hidden when *any* constructor is defined.

Objects dynamically allocated with `malloc`, `@=` objects, and objects with only trivial constructors and destructors are unmanaged. Unmanaged objects are never the target of an implicit constructor or destructor call.

Chapter 3

Tuples

3.1 Multiple-Return-Value Functions

In standard C, functions can return at most one value. This restriction results in code which emulates functions with multiple return values by *aggregation* or by *aliasing*. In the former situation, the function designer creates a record type that combines all of the return values into a single type. For example, consider a function returning the most frequently occurring letter in a string, and its frequency. This example is complex enough to illustrate that an array is insufficient, since arrays are homogeneous, and demonstrates a potential pitfall that exists with aliasing.

```
struct mf_ret {
  int freq;
  char ch;
};

struct mf_ret most_frequent(const char * str) {
  char freqs [26] = { 0 };
  struct mf_ret ret = { 0, 'a' };
  for (int i = 0; str[i] != '\0'; ++i) {
    if (isalpha(str[i])) {        // only count letters
      int ch = tolower(str[i]);   // convert to lower case
      int idx = ch-'a';
      if (++freqs[idx] > ret.freq) {  // update on new max
        ret.freq = freqs[idx];
        ret.ch = ch;
      }
    }
  }
  return ret;
}

const char * str = "hello world";
struct mf_ret ret = most_frequent(str);
printf("%s -- %d %c\n", str, ret.freq, ret.ch);
```

Of note, the designer must come up with a name for the return type and for each of its fields. Unnecessary naming is a common programming language issue, introducing verbosity and a complication of the user's mental model. That is, adding another named type creates another

association in the programmer's mind that needs to be kept track of when reading and writing code. As such, this technique is effective when used sparingly, but can quickly get out of hand if many functions need to return different combinations of types.

In the latter approach, the designer simulates multiple return values by passing the additional return values as pointer parameters. The pointer parameters are assigned inside of the routine body to emulate a return. Using the same example,

```
int most_frequent(const char * str, char * ret_ch) {
  char freqs [26] = { 0 };
  int ret_freq = 0;
  for (int i = 0; str[i] != '\0'; ++i) {
    if (isalpha(str[i])) {        // only count letters
      int ch = tolower(str[i]);   // convert to lower case
      int idx = ch-'a';
      if (++freqs[idx] > ret_freq) {  // update on new max
        ret_freq = freqs[idx];
        *ret_ch = ch;   // assign to out parameter
      }
    }
  }
  return ret_freq;  // only one value returned directly
}

const char * str = "hello world";
char ch;                              // pre-allocate return value
int freq = most_frequent(str, &ch); // pass return value as out parameter
printf("%s -- %d %c\n", str, freq, ch);
```

Notably, using this approach, the caller is directly responsible for allocating storage for the additional temporary return values, which complicates the call site with a sequence of variable declarations leading up to the call. Also, while a disciplined use of **`const`** can give clues about whether a pointer parameter is going to be used as an out parameter, it is not immediately obvious from only the routine signature whether the callee expects such a parameter to be initialized before the call. Furthermore, while many C routines that accept pointers are designed so that it is safe to pass `NULL` as a parameter, there are many C routines that are not null-safe. On a related note, C does not provide a standard mechanism to state that a parameter is going to be used as an additional return value, which makes the job of ensuring that a value is returned more difficult for the compiler. Interestingly, there is a subtle bug in the previous example, in that `ret_ch` is never assigned for a string that does not contain any letters, which can lead to undefined behaviour. In this particular case, it turns out that the frequency return value also doubles as an error code, where a frequency of 0 means the character return value should be ignored. Still, not every routine with multiple return values should be required to return an error code, and error codes are easily ignored, so this is not a satisfying solution. As with the previous approach, this technique can simulate multiple return values, but in practice it is verbose and error prone.

In C∀, functions can be declared to return multiple values with an extension to the function declaration syntax. Multiple return values are declared as a comma-separated list of types in square brackets in the same location that the return type appears in standard C function declarations. The ability to return multiple values from a function requires a new syntax for the return statement. For consistency, the return statement in C∀ accepts a comma-separated list

of expressions in square brackets. The expression resolution phase of the C∀ translator ensures that the correct form is used depending on the values being returned and the return type of the current function. A multiple-returning function with return type `T` can return any expression that is implicitly convertible to `T`. Using the running example, the `most_frequent` function can be written using multiple return values as such,

```
[int, char] most_frequent(const char * str) {
  char freqs [26] = { 0 };
  int ret_freq = 0;
  char ret_ch = 'a';  // arbitrary default value for consistent results
  for (int i = 0; str[i] != '\0'; ++i) {
    if (isalpha(str[i])) {        // only count letters
      int ch = tolower(str[i]);   // convert to lower case
      int idx = ch-'a';
      if (++freqs[idx] > ret_freq) {  // update on new max
        ret_freq = freqs[idx];
        ret_ch = ch;
      }
    }
  }
  return [ret_freq, ret_ch];
}
```

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

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

```
const char * str = "hello world";
int freq;
char ch;
[freq, ch] = most_frequent(str);  // assign into multiple variables
printf("%s -- %d %c\n", str, freq, ch);
```

It is also common to use a function's output as the input to another function. C∀ also allows this case, without any new syntax. When a function call is passed as an argument to another call, the expression resolver attempts to find the best match of actual arguments to formal parameters given all of the possible expression interpretations in the current scope [3]. For example,

```
void process(int);       // (1)
void process(char);      // (2)
void process(int, char); // (3)
void process(char, int); // (4)

process(most_frequent("hello world"));  // selects (3)
```

In this case, there is only one option for a function named `most_frequent` that takes a string as input. This function returns two values, one **`int`** and one **`char`**. There are four options for a

function named `process`, but only two that accept two arguments, and of those the best match is (3), which is also an exact match. This expression first calls `most_frequent("hello world")`, which produces the values `3` and `'l'`, which are fed directly to the first and second parameters of (3), respectively.

3.2 Tuple Expressions

Multiple-return-value functions provide C∀ with a new syntax for expressing a combination of expressions in the return statement and a combination of types in a function signature. These notions can be generalized to provide C∀ with *tuple expressions* and *tuple types*. A tuple expression is an expression producing a fixed-size, ordered list of values of heterogeneous types. The type of a tuple expression is the tuple of the subexpression types, or a *tuple type*. In C∀, a tuple expression is denoted by a comma-separated list of expressions enclosed in square brackets. For example, the expression `[5, 'x', 10.5]` has type `[int, char, double]`. The previous expression has 3 *components*. Each component in a tuple expression can be any C∀ expression, including another tuple expression. The order of evaluation of the components in a tuple expression is unspecified, to allow a compiler the greatest flexibility for program optimization. It is, however, guaranteed that each component of a tuple expression is evaluated for side-effects, even if the result is not used. Multiple-return-value functions can equivalently be called *tuple-returning functions*.

3.2.1 Tuple Variables

The call-site of the `most_frequent` routine has a notable blemish, in that it required the preallocation of return variables in a manner similar to the aliasing example, since it is impossible to declare multiple variables of different types in the same declaration in standard C. In C∀, it is possible to overcome this restriction by declaring a *tuple variable*.

```
const char * str = "hello world";
[int, char] ret = most_frequent(str);  // initialize tuple variable
printf("%s -- %d %c\n", str, ret);
```

It is now possible to accept multiple values into a single piece of storage, in much the same way that it was previously possible to pass multiple values from one function call to another. These variables can be used in any of the contexts where a tuple expression is allowed, such as in the `printf` function call. As in the `process` example, the components of the tuple value are passed as separate parameters to `printf`, allowing very simple printing of tuple expressions. One way to access the individual components is with a simple assignment, as in previous examples.

```
int freq;
char ch;
[freq, ch] = ret;
```

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

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

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

3.2.2 Tuple Indexing

At times, it is desirable to access a single component of a tuple-valued expression without creating unnecessary temporary variables to assign to. Given a tuple-valued expression `e` and a compile-time constant integer i where $0 \leq i < n$, where n is the number of components in `e`, `e.i` accesses the i^{th} component of `e`. For example,

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

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

As seen above, tuple-index expressions can occur on any tuple-typed expression, including tuple-returning functions, square-bracketed tuple expressions, and other tuple-index expressions, provided the retrieved component is also a tuple. This feature was proposed for K-W C but never implemented [21, p. 45].

3.2.3 Flattening and Structuring

As evident in previous examples, tuples in C∀ do not have a rigid structure. In function call contexts, tuples support implicit flattening and restructuring conversions. Tuple flattening recursively expands a tuple into the list of its basic components. Tuple structuring packages a list of expressions into a value of tuple type.

```
int f(int, int);
int g([int, int]);
int h(int, [int, int]);
[int, int] x;
int y;

f(x);       // flatten
g(y, 10);   // structure
h(x, y);    // flatten & structure
```

In C∀, each of these calls is valid. In the call to `f`, `x` is implicitly flattened so that the components of `x` are passed as the two arguments to `f`. For the call to `g`, the values `y` and `10` are structured

into a single argument of type `[int, int]` to match the type of the parameter of `g`. Finally, in the call to `h`, `x` is flattened to yield an argument list of length 3, of which the first component of `x` is passed as the first parameter of `h`, and the second component of `x` and `y` are structured into the second argument of type `[int, int]`. The flexible structure of tuples permits a simple and expressive function-call syntax to work seamlessly with both single- and multiple-return-value functions, and with any number of arguments of arbitrarily complex structure.

In K-W C [5, 21], there were 4 tuple coercions: opening, closing, flattening, and structuring. Opening coerces a tuple value into a tuple of values, while closing converts a tuple of values into a single tuple value. Flattening coerces a nested tuple into a flat tuple, *i.e.*, it takes a tuple with tuple components and expands it into a tuple with only non-tuple components. Structuring moves in the opposite direction, *i.e.*, it takes a flat tuple value and provides structure by introducing nested tuple components.

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

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

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

3.3 Tuple Assignment

An assignment where the left side of the assignment operator has a tuple type is called tuple assignment. There are two kinds of tuple assignment depending on whether the right side of the

assignment operator has a tuple type or a non-tuple type, called *Multiple* and *Mass* Assignment, respectively.

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

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

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

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

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

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

Both kinds of tuple assignment have parallel semantics, such that each value on the left side and right side is evaluated *before* any assignments occur. As a result, it is possible to swap the values in two variables without explicitly creating any temporary variables or calling a function.

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

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

In C∀, tuple assignment is an expression where the result type is the type of the left side of the assignment, as in normal assignment. That is, a tuple assignment produces the value of the left-hand side after assignment. These semantics allow cascading tuple assignment to work out naturally in any context where a tuple is permitted. These semantics are a change from the original tuple design in K-W C [21], wherein tuple assignment was a statement that allows cascading assignments as a special case. Restricting tuple assignment to statements was an attempt to to fix what was seen as a problem with side-effects, wherein assignment can be used in many different locations, such as in function-call argument position. While permitting assignment as

an expression does introduce the potential for subtle complexities, it is impossible to remove assignment expressions from C∀ without affecting backwards compatibility. Furthermore, there are situations where permitting assignment as an expression improves readability by keeping code succinct and reducing repetition, and complicating the definition of tuple assignment puts a greater cognitive burden on the user. In another language, tuple assignment as a statement could be reasonable, but it would be inconsistent for tuple assignment to be the only kind of assignment that is not an expression. In addition, K-W C permits the compiler to optimize tuple assignment as a block copy, since it does not support user-defined assignment operators. This optimization could be implemented in C∀, but it requires the compiler to verify that the selected assignment operator is trivial.

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

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

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

3.3.1 Tuple Construction

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

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

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

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

It is possible to define constructors and assignment functions for tuple types that provide new semantics, if the existing semantics do not fit the needs of an application. For example, the

function `void ?{}([T, U] *, S);` can be defined to allow a tuple variable to be constructed from a value of type `S`.

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

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

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

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

3.4 Member-Access Tuple Expression

It is possible to access multiple fields from a single expression using a *Member-Access Tuple Expression*. The result is a single tuple-valued expression whose type is the tuple of the types of the members. For example,

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

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

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

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

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

It is possible for a member tuple expression to contain other member access expressions. For example,

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

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

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

In K-W C, member tuple expressions are known as *record field tuples* [21]. Since C∀ permits these tuple-access expressions using structures, unions, and tuples, *member tuple expression* or *field tuple expression* is more appropriate.

It is possible to extend member-access expressions further. Currently, a member-access expression whose member is a name requires that the aggregate is a structure or union, while a constant integer member requires the aggregate to be a tuple. In the interest of orthogonal design, C∀ could apply some meaning to the remaining combinations as well. For example,

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

s.x;  // access member
z.0;  // access component

s.1;  // ???
z.y;  // ???
```

One possibility is for `s.1` to select the second member of `s`. Under this interpretation, it becomes possible to not only access members of a struct by name, but also by position. Likewise, it seems natural to open this mechanism to enumerations as well, wherein the left side would be a type, rather than an expression. One benefit of this interpretation is familiarity, since it is extremely reminiscent of tuple-index expressions. On the other hand, it could be argued that this interpretation is brittle in that changing the order of members or adding new members to a structure becomes a brittle operation. This problem is less of a concern with tuples, since modifying a tuple affects only the code that directly uses the tuple, whereas modifying a structure has far reaching consequences for every instance of the structure.

As for `z.y`, one interpretation is to extend the meaning of member tuple expressions. That is, currently the tuple must occur as the member, *i.e.*, to the right of the dot. Allowing tuples to the left of the dot could distribute the member across the elements of the tuple, in much the same way that member tuple expressions distribute the aggregate across the member tuple. In this example, `z.y` expands to `[z.0.y, z.1.y]`, allowing what is effectively a very limited compile-time field-sections map operation, where the argument must be a tuple containing only aggregates having a member named `y`. It is questionable how useful this would actually be in practice, since structures often do not have names in common with other structures, and further this could cause maintainability issues in that it encourages programmers to adopt very simple naming conventions to maximize the amount of overlap between different types. Perhaps more useful would be to allow arrays on the left side of the dot, which would likewise allow mapping a field access across the entire array, producing an array of the contained fields. The immediate problem with this idea is that C arrays do not carry around their size, which would make it

impossible to use this extension for anything other than a simple stack allocated array.

Supposing this feature works as described, it would be necessary to specify an ordering for the expansion of member-access expressions versus member-tuple expressions.

```
struct { int x, y; };
[S, S] z;
z.[x, y];  // ???
// => [z.0, z.1].[x, y]
// => [z.0.x, z.0.y, z.1.x, z.1.y]
// or
// => [z.x, z.y]
// => [[z.0, z.1].x, [z.0, z.1].y]
// => [z.0.x, z.1.x, z.0.y, z.1.y]
```

Depending on exactly how the two tuples are combined, different results can be achieved. As such, a specific ordering would need to be imposed to make this feature useful. Furthermore, this addition moves a member-tuple expression's meaning from being clear statically to needing resolver support, since the member name needs to be distributed appropriately over each member of the tuple, which could itself be a tuple.

A second possibility is for C∀ to have named tuples, as they exist in Swift and D.

```
typedef [int x, int y] Point2D;
Point2D p1, p2;
p1.x + p1.y + p2.x + p2.y;
p1.0 + p1.1 + p2.0 + p2.1;  // equivalent
```

In this simpler interpretation, a tuple type carries with it a list of possibly empty identifiers. This approach fits naturally with the named return-value feature, and would likely go a long way towards implementing it.

Ultimately, the first two extensions introduce complexity into the model, with relatively little perceived benefit, and so were dropped from consideration. Named tuples are a potentially useful addition to the language, provided they can be parsed with a reasonable syntax.

3.5 Casting

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

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

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

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

```
1 int f();
2 void g();
3
4 (void)f();  // valid, ignore results
5 (int)g();   // invalid, void cannot be converted to int
6
7 struct A { int x; };
8 (struct A)f();  // invalid, int cannot be converted to A
```

In C, line 4 is a valid cast, which calls `f` and discards its result. On the other hand, line 5 is invalid, because `g` does not produce a result, so requesting an **`int`** to materialize from nothing is nonsensical. Finally, line 8 is also invalid, because in C casts only provide conversion between scalar types [14, p. 91]. For consistency, this implies that any case wherein the number of components increases as a result of the cast is invalid, while casts that have the same or fewer number of components may be valid.

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

For example,

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

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

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

3.6 Polymorphism

Due to the implicit flattening and structuring conversions involved in argument passing, **`otype`** and **`dtype`** parameters are restricted to matching only with non-tuple types. The integration

of polymorphism, type assertions, and monomorphic specialization of tuple-assertions are a primary contribution of this thesis to the design of tuples.

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

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

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

Tuples can contain otype and dtype components. For example, a plus operator can be written to add two triples of a type together.

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

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

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

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

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

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

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

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

```
f([0, 0,], [0, 0]); // cost for flattening
f([0, 0, 0], 0);    // cost for flattening and structuring
```

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

3.6.1 Assertion Inference

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

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

This relaxation is made possible by extending the existing thunk generation scheme, as described by Bilson [3]. Now, whenever a candidate's parameter structure does not exactly match the formal parameter's structure, a thunk is generated to specialize calls to the actual function.

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

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

3.7 Implementation

Tuples are implemented in the C∀ translator via a transformation into generic types. Generic types are an independent contribution developed at the same time. The transformation into generic types and the generation of tuple-specific code are primary contributions of this thesis to tuples.

The first time an *N*-tuple is seen for each *N* in a scope, a generic type with *N* type parameters is generated. For example,

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

is transformed into

```
forall(dtype T0, dtype T1 | sized(T0) | sized(T1))
struct _tuple2_ {  // generated before the first 2-tuple
  T0 field_0;
  T1 field_1;
};
_tuple2_(int, int) f() {
  _tuple2_(double, double) x;
  forall(dtype T0, dtype T1, dtype T2 | sized(T0) | sized(T1) | sized(T2))
  struct _tuple3_ {  // generated before the first 3-tuple
    T0 field_0;
    T1 field_1;
    T2 field_2;
  };
  _tuple3_(int, double, int) y;
}
```

Tuple expressions are then simply converted directly into compound literals

```
[5, 'x', 1.24];
```

becomes

```
(_tuple3_(int, char, double)){ 5, 'x', 1.24 };
```

Since tuples are essentially structures, tuple indexing expressions are just field accesses.

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

x.0+x.1;
printf("%d %g\n", x);
f(x, 'z');
```

is transformed into

```
void f(int, _tuple2_(double, char));
_tuple2_(int, double) x;

x.field_0+x.field_1;
printf("%d %g\n", x.field_0, x.field_1);
f(x.field_0, (_tuple2){ x.field_1, 'z' });
```

Note that due to flattening, `x` used in the argument position is converted into the list of its fields. In the call to `f`, the second and third argument components are structured into a tuple argument.

Expressions that may contain side effects are made into *unique expressions* before being expanded by the flattening conversion. Each unique expression is assigned an identifier and is guaranteed to be executed exactly once.

```
void g(int, double);
[int, double] h();
g(h());
```

Internally, this is converted to pseudo-C∀

```
void g(int, double);
```

```
[int, double] h();
lazy [int, double] unq0 = h(); // deferred execution
g(unq0.0, unq0.1);             // execute h() once
```

That is, the function `h` is evaluated lazily and its result is stored for subsequent accesses. Ultimately, unique expressions are converted into two variables and an expression.

```
void g(int, double);
[int, double] h();

_Bool _unq0_finished_ = 0;
[int, double] _unq0;
g(
  (_unq0_finished_ ? _unq0 : (_unq0 = h(), _unq0_finished_ = 1, _unq0)).0,
  (_unq0_finished_ ? _unq0 : (_unq0 = h(), _unq0_finished_ = 1, _unq0)).1,
);
```

Since argument evaluation order is not specified by the C programming language, this scheme is built to work regardless of evaluation order. The first time a unique expression is executed, the actual expression is evaluated and the accompanying boolean is set to true. Every subsequent evaluation of the unique expression then results in an access to the stored result of the actual expression.

Currently, the C∀ translator has a very broad, imprecise definition of impurity (side-effects), where every function call is assumed to be impure. This notion could be made more precise for certain intrinsic, auto-generated, and built-in functions, and could analyze function bodies, when they are available, to recursively detect impurity, to eliminate some unique expressions. It is possible that lazy evaluation could be exposed to the user through a lazy keyword with little additional effort.

Tuple-member expressions are recursively expanded into a list of member-access expressions.

```
[int, [double, int, double], int]] x;
x.[0, 1.[0, 2]];
```

becomes

```
[x.0, [x.1.0, x.1.2]];
```

Tuple-member expressions also take advantage of unique expressions in the case of possible impurity.

Finally, the various kinds of tuple assignment, constructors, and destructors generate GNU C statement expressions. For example, a mass assignment

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

[x, y, z] = 1.5;            // mass assignment
```

generates the following

```
// [x, y, z] = 1.5;
_tuple3_(int, double, int) _tmp_stmtexpr_ret0;
({ // GNU C statement expression
  // assign LHS address temporaries
  int *__massassign_L0 = &x;    // ?{}
  double *__massassign_L1 = &y; // ?{}
  int *__massassign_L2 = &z;    // ?{}

  // assign RHS value temporary
  double __massassign_R0 = 1.5; // ?{}

  ({ // tuple construction - construct statement expr return variable
    // assign LHS address temporaries
    int *__multassign_L0 = (int *)&_tmp_stmtexpr_ret0.0;       // ?{}
    double *__multassign_L1 = (double *)&_tmp_stmtexpr_ret0.1; // ?{}
    int *__multassign_L2 = (int *)&_tmp_stmtexpr_ret0.2;       // ?{}

    // assign RHS value temporaries and mass-assign to L0, L1, L2
    int __multassign_R0 = (*__massassign_L0=(int)__massassign_R0); // ?{}
    double __multassign_R1 = (*__massassign_L1=__massassign_R0);   // ?{}
    int __multassign_R2 = (*__massassign_L2=(int)__massassign_R0); // ?{}

    // perform construction of statement expr return variable using
    // RHS value temporary
    ((*__multassign_L0 = __multassign_R0 /* ?{} */),
     (*__multassign_L1 = __multassign_R1 /* ?{} */),
     (*__multassign_L2 = __multassign_R2 /* ?{} */));
  });
  _tmp_stmtexpr_ret0;
});
({ // tuple destruction - destruct assign expr value
  int *__massassign_L3 = (int *)&_tmp_stmtexpr_ret0.0;       // ?{}
  double *__massassign_L4 = (double *)&_tmp_stmtexpr_ret0.1; // ?{}
  int *__massassign_L5 = (int *)&_tmp_stmtexpr_ret0.2;       // ?{}
  ((*__massassign_L3 /* ^?{} */),
   (*__massassign_L4 /* ^?{} */),
   (*__massassign_L5 /* ^?{} */));
});
```

A variable is generated to store the value produced by a statement expression, since its fields may need to be constructed with a non-trivial constructor and it may need to be referred to multiple time, *e.g.*, in a unique expression. *N* LHS variables are generated and constructed using the address of the tuple components, and a single RHS variable is generated to store the value of the RHS without any loss of precision. A nested statement expression is generated that performs the individual assignments and constructs the return value using the results of the individual assignments. Finally, the statement expression temporary is destroyed at the end of the expression.

Similarly, a multiple assignment

```
[x, y, z] = [f(), 3];       // multiple assignment
```

generates the following

```
// [x, y, z] = [f(), 3];
_tuple3_(int, double, int) _tmp_stmtexpr_ret0;
({
  // assign LHS address temporaries
  int *__multassign_L0 = &x;    // ?{}
  double *__multassign_L1 = &y; // ?{}
  int *__multassign_L2 = &z;    // ?{}

  // assign RHS value temporaries
  _tuple2_(double, double) _tmp_cp_ret0;
  _Bool _unq0_finished_ = 0;
  double __multassign_R0 =
    (_unq0_finished_ ?
      _tmp_cp_ret0 :
      (_tmp_cp_ret0=f(), _unq0_finished_=1, _tmp_cp_ret0)).0; // ?{}
  double __multassign_R1 =
    (_unq0_finished_ ?
      _tmp_cp_ret0 :
      (_tmp_cp_ret0=f(), _unq0_finished_=1, _tmp_cp_ret0)).1; // ?{}
  ({ // tuple destruction - destruct f() return temporary
    // assign LHS address temporaries
    double *__massassign_L3 = (double *)&_tmp_cp_ret0.0;  // ?{}
    double *__massassign_L4 = (double *)&_tmp_cp_ret0.1;  // ?{}
    // perform destructions - intrinsic, so NOP
    ((*__massassign_L3 /* ^?{} */),
     (*__massassign_L4 /* ^?{} */));
  });
  int __multassign_R2 = 3; // ?{}

  ({ // tuple construction - construct statement expr return variable
    // assign LHS address temporaries
    int *__multassign_L3 = (int *)&_tmp_stmtexpr_ret0.0;       // ?{}
    double *__multassign_L4 = (double *)&_tmp_stmtexpr_ret0.1; // ?{}
    int *__multassign_L5 = (int *)&_tmp_stmtexpr_ret0.2;       // ?{}

    // assign RHS value temporaries and multiple-assign to L0, L1, L2
    int __multassign_R3 = (*__multassign_L0=(int)__multassign_R0);  // ?{}
    double __multassign_R4 = (*__multassign_L1=__multassign_R1);    // ?{}
    int __multassign_R5 = (*__multassign_L2=__multassign_R2);       // ?{}

    // perform construction of statement expr return variable using
    // RHS value temporaries
    ((*__multassign_L3=__multassign_R3 /* ?{} */),
     (*__multassign_L4=__multassign_R4 /* ?{} */),
     (*__multassign_L5=__multassign_R5 /* ?{} */));
  });
  _tmp_stmtexpr_ret0;
});
({  // tuple destruction - destruct assign expr value
  // assign LHS address temporaries
  int *__massassign_L5 = (int *)&_tmp_stmtexpr_ret0.0;       // ?{}
  double *__massassign_L6 = (double *)&_tmp_stmtexpr_ret0.1; // ?{}
  int *__massassign_L7 = (int *)&_tmp_stmtexpr_ret0.2;       // ?{}
```

```
  // perform destructions - intrinsic, so NOP
  ((*__massassign_L5 /* ^?{} */),
   (*__massassign_L6 /* ^?{} */),
   (*__massassign_L7 /* ^?{} */));
});
```

The difference here is that *N* RHS values are stored into separate temporary variables.

The use of statement expressions allows the translator to arbitrarily generate additional temporary variables as needed, but binds the implementation to a non-standard extension of the C language. There are other places where the C∀ translator makes use of GNU C extensions, such as its use of nested functions, so this is not a new restriction.

Chapter 4

Variadic Functions

4.1 Design Criteria

C provides variadic functions through the manipulation of `va_list` objects. In C, a variadic function is one which contains at least one parameter, followed by `...` as the last token in the parameter list. In particular, some form of *argument descriptor* or *sentinel value* is needed to inform the function of the number of arguments and their types. Two common argument descriptors are format strings or counter parameters. It is important to note that both of these mechanisms are inherently redundant, because they require the user to explicitly specify information that the compiler already knows [1]. This required repetition is error prone, because it is easy for the user to add or remove arguments without updating the argument descriptor. In addition, C requires the programmer to hard code all of the possible expected types. As a result, it is cumbersome to write a function that is open to extension. For example, a simple function to sum *N* **int**s,

```
int sum(int N, ...) {
  va_list args;
  va_start(args, N);
  int ret = 0;
  while(N) {
    ret += va_arg(args, int);  // have to specify type
    N--;
  }
  va_end(args);
  return ret;
}
sum(3, 10, 20, 30);  // need to keep counter in sync
```

The `va_list` type is a special C data type that abstracts variadic-argument manipulation. The `va_start` macro initializes a `va_list`, given the last named parameter. Each use of the `va_arg` macro allows access to the next variadic argument, given a type. Since the function signature does not provide any information on what types can be passed to a variadic function, the compiler does not perform any error checks on a variadic call. As such, it is possible to pass any value to the `sum` function, including pointers, floating-point numbers, and structures. In the case where the

[1] While format specifiers can convey some information the compiler does not know, such as whether to print a number in decimal or hexadecimal, the number of arguments is wholly redundant.

provided type is not compatible with the argument's actual type after default argument promotions, or if too many arguments are accessed, the behaviour is undefined [14, p. 81]. Furthermore, there is no way to perform the necessary error checks in the `sum` function at run-time, since type information is not carried into the function body. Since they rely on programmer convention rather than compile-time checks, variadic functions are unsafe.

In practice, compilers can provide warnings to help mitigate some of the problems. For example, GCC provides the `format` attribute to specify that a function uses a format string, which allows the compiler to perform some checks related to the standard format-specifiers. Unfortunately, this approach does not permit extensions to the format-string syntax, so a programmer cannot extend the attribute to warn for mismatches with custom types.

As a result, C's variadic functions are a deficient language feature. Two options were examined to provide better, type-safe variadic functions in C∀.

4.1.1 Whole Tuple Matching

Option 1 is to change the argument matching algorithm, so that type parameters can match whole tuples, rather than just their components. This option could be implemented with two phases of argument matching when a function contains type parameters and the argument list contains tuple arguments. If flattening and structuring fail to produce a match, a second attempt at matching the function and argument combination is made where tuple arguments are not expanded and structure must match exactly, modulo non-tuple implicit conversions. For example:

```
forall(otype T, otype U | { T g(U); })
void f(T, U);

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

f([1, 2], [3, 4, 5, 6]);
```

With flattening and structuring, the call is first transformed into `f(1, 2, 3, 4, 5, 6)`. Since the first argument of type `T` does not have a tuple type, unification decides that `T=int` and `1` is matched as the first parameter. Likewise, `U` does not have a tuple type, so `U=int` and `2` is accepted as the second parameter. There are now no remaining formal parameters, but there are remaining arguments and the function is not variadic, so the match fails.

With the addition of an exact matching attempt, `T=[int,int]` and `U=[int,int,int,int]`, and so the arguments type check. Likewise, when inferring assertion `g`, an exact match is found.

This approach is strict with respect to argument structure, by nature, which makes it syntactically awkward to use in ways that the existing tuple design is not. For example, consider a `new` function that allocates memory using `malloc`, and constructs the result using arbitrary arguments.

```
struct Array;
void ?{}(Array *, int, int, int);

forall(dtype T, otype Params | sized(T) | { void ?{}(T *, Params); })
T * new(Params p) {
```

```
  return malloc(){ p };
}
Array(int) * x = new([1, 2, 3]);
```

The call to `new` is not particularly appealing, since it requires the use of square brackets at the call-site, which is not required in any other function call. This shifts the burden from the compiler to the programmer, which is almost always wrong, and creates an odd inconsistency within the language. Similarly, in order to pass 0 variadic arguments, an explicit empty tuple must be passed into the argument list, otherwise the exact matching rule would not have an argument to bind against.

It should be otherwise noted that the addition of an exact matching rule only affects the outcome for polymorphic type-binding when tuples are involved. For non-tuple arguments, exact matching and flattening and structuring are equivalent. For tuple arguments to a function without polymorphic formal-parameters, flattening and structuring work whenever an exact match would have worked, since the tuple is flattened and implicitly restructured to its original structure. Thus there is nothing to be gained from permitting the exact matching rule to take effect when a function does not contain polymorphism and none of the arguments are tuples.

Overall, this option takes a step in the right direction, but is contrary to the flexibility of the existing tuple design.

4.1.2 A New Typeclass

A second option is the addition of another kind of type parameter, **`ttype`**. Matching against a **`ttype`** parameter consumes all remaining argument components and packages them into a tuple, binding to the resulting tuple of types. In a given parameter list, there should be at most one **`ttype`** parameter that must occur last, otherwise the call can never resolve, given the previous rule. This idea essentially matches normal variadic semantics, with a strong feeling of similarity to C++11 variadic templates. As such, **`ttype`** variables are also referred to as argument packs. This approach is the option that has been added to C∀.

Like variadic templates, the main way to manipulate **`ttype`** polymorphic functions is through recursion. Since nothing is known about a parameter pack by default, assertion parameters are key to doing anything meaningful. Unlike variadic templates, **`ttype`** polymorphic functions can be separately compiled.

For example, a simple translation of the C sum function using **`ttype`** is

```
int sum(void){ return 0; }         // (0)
forall(ttype Params | { int sum(Params); })
int sum(int x, Params rest) { // (1)
  return x+sum(rest);
}
sum(10, 20, 30);
```

Since (0) does not accept any arguments, it is not a valid candidate function for the call `sum(10, 20, 30)`. In order to call (1), `10` is matched with `x`, and the argument resolution moves on to the argument pack `rest`, which consumes the remainder of the argument list and `Params` is bound

to `[20, 30]`. In order to finish the resolution of `sum`, an assertion parameter that matches **`int`** `sum(`**`int, int`**`)` is required. Like in the previous iteration, (0) is not a valid candidate, so (1) is examined with `Params` bound to `[`**`int`**`]`, requiring the assertion **`int`** `sum(`**`int`**`)`. Next, (0) fails, and to satisfy (1) `Params` is bound to `[]`, requiring an assertion **`int`** `sum()`. Finally, (0) matches and (1) fails, which terminates the recursion. Effectively, this traces as `sum(10, 20, 30)` → `10+sum(20, 30)` → `10+(20+sum(30))` → `10+(20+(30+sum()))` → `10+(20+(30+0))`.

Interestingly, this version does not require any form of argument descriptor, since the C∀ type system keeps track of all of these details. It might be reasonable to take the `sum` function a step further to enforce a minimum number of arguments, which could be done simply

```
int sum(int x, int y){
  return x+y;
}
forall(ttype Params | { int sum(int, Params); })
int sum(int x, int y, Params rest) {
  return sum(x+y, rest);
}
sum(10);          // invalid
sum(10, 20);      // valid
sum(10, 20, 30);  // valid
...
```

One more iteration permits the summation of any summable type, as long as all arguments are the same type.

```
trait summable(otype T) {
  T ?+?(T, T);
};
forall(otype R | summable(R))
R sum(R x, R y){
  return x+y;
}
forall(otype R, ttype Params
  | summable(R)
  | { R sum(R, Params); })
R sum(R x, R y, Params rest) {
  return sum(x+y, rest);
}
sum(3, 10, 20, 30);
```

Unlike C, it is not necessary to hard code the expected type. This `sum` function is naturally open to extension, in that any user-defined type with a `?+?` operator is automatically able to be used with the `sum` function. That is to say, the programmer who writes `sum` does not need full program knowledge of every possible data type, unlike what is necessary to write an equivalent function using the standard C mechanisms.

Going one last step, it is possible to achieve full generality in C∀, allowing the summation of arbitrary lists of summable types.

```
trait summable(otype T1, otype T2, otype R) {
  R ?+?(T1, T2);
};
forall(otype T1, otype T2, otype R | summable(T1, T2, R))
```

```
R sum(T1 x, T2 y) {
  return x+y;
}
forall(otype T1, otype T2, otype T3, otype R, ttype Params
  | summable(T1, T2, T3)
  | { R sum(T3, Params); })
R sum(T1 x, T2 y, Params rest ) {
  return sum(x+y, rest);
}
sum(3, 10.5, 20, 30.3);
```

The C∀ translator requires adding explicit **double** ?+?(**int**, **double**) and **double** ?+?(**double**, **int**) functions for this call to work, since implicit conversions are not supported for assertions.

A notable limitation of this approach is that it heavily relies on recursive assertions. The C∀ translator imposes a limitation on the depth of the recursion for assertion satisfaction. Currently, the limit is set to 4, which means that the first version of the `sum` function is limited to at most 5 arguments, while the second version can support up to 6 arguments. The limit is set low due to inefficiencies in the current implementation of the C∀ expression resolver. There is ongoing work to improve the performance of the resolver, and with noticeable gains, the limit can be relaxed to allow longer argument lists to **ttype** functions.

C variadic syntax and **ttype** polymorphism probably should not be mixed, since it is not clear where to draw the line to decide which arguments belong where. Furthermore, it might be desirable to disallow polymorphic functions to use C variadic syntax to encourage a C∀ style. Aside from calling C variadic functions, it is not obvious that there is anything that can be done with C variadics that could not also be done with **ttype** parameters.

Variadic templates in C++ require an ellipsis token to express that a parameter is a parameter pack and to expand a parameter pack. C∀ does not need an ellipsis in either case, since the type class **ttype** is only used for variadics. An alternative design is to use an ellipsis combined with an existing type class. This approach was not taken because the largest benefit of the ellipsis token in C++ is the ability to expand a parameter pack within an expression, *e.g.*, in fold expressions, which requires compile-time knowledge of the structure of the parameter pack, which is not available in C∀.

```
template<typename... Args>
void f(Args &... args) {
  g(&args...);  // expand to addresses of pack elements
}
```

As such, the addition of an ellipsis token would be purely an aesthetic change in C∀ today.

It is possible to write a type-safe variadic print routine, which can replace `printf`

```
struct S { int x, y; };
forall(otype T, ttype Params |
  { void print(T); void print(Params); })
void print(T arg, Params rest) {
  print(arg);
  print(rest);
}
```

```
void print(char * x) { printf("%s", x); }
void print(int x) { printf("%d", x);  }
void print(S s) { print("{ ", s.x, ",", s.y, " }"); }
print("s = ", (S){ 1, 2 }, "\n");
```

This example routine showcases a variadic-template-like decomposition of the provided argument list. The individual `print` routines allow printing a single element of a type. The polymorphic `print` allows printing any list of types, as long as each individual type has a `print` function. The individual print functions can be used to build up more complicated `print` routines, such as for `S`, which is something that cannot be done with `printf` in C.

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

```
struct Array;
void ?{}(Array *, int, int, int);

forall(dtype T, ttype Params | sized(T) | { void ?{}(T *, Params); })
T * new(Params p) {
  return malloc(){ p }; // construct result of malloc
}
Array * x = new(1, 2, 3);
```

In the call to `new`, `Array` is selected to match `T`, and `Params` is expanded to match `[int, int, int, int]`. To satisfy the assertions, a constructor with an interface compatible with `void ?{}(Array *, int, int, int)` must exist in the current scope.

The `new` function provides the combination of polymorphic `malloc` with a constructor call, so that it becomes impossible to forget to construct dynamically-allocated objects. This approach provides the type-safety of `new` in C~~++~~, without the need to specify the allocated type, thanks to return-type inference.

4.2 Implementation

The definition of `new`

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

forall(dtype T, ttype Params | sized(T) | { void ?{}(T *, Params); })
T * new(Params p) {
  return malloc(){ p }; // construct result of malloc
}
```

generates the following

```
void *malloc(long unsigned int _sizeof_T, long unsigned int _alignof_T);

void *new(
  void (*_adapter_)(void (*)(), void *, void *),
  long unsigned int _sizeof_T,
  long unsigned int _alignof_T,
  long unsigned int _sizeof_Params,
```

```
  long unsigned int _alignof_Params,
  void (* _ctor_T)(void *, void *),
  void *p
){
  void *_retval_new;
  void *_tmp_cp_ret0;
  void *_tmp_ctor_expr0;
  _retval_new=
    (_adapter_(_ctor_T,
      (_tmp_ctor_expr0=(_tmp_cp_ret0=malloc(_sizeof_2tT, _alignof_2tT),
        _tmp_cp_ret0)),
      p),
    _tmp_ctor_expr0); // ?{}
  *(void **)&_tmp_cp_ret0; // ^?{}
  return _retval_new;
}
```

The constructor for `T` is called indirectly through the adapter function on the result of `malloc` and the parameter pack. The variable that is allocated and constructed is then returned from `new`.

A call to `new`

```
struct S { int x, y; };
void ?{}(S *, int, int);

S * s = new(3, 4);
```

Generates the following

```
struct _tuple2_ {  // _tuple2_(T0, T1)
  void *field_0;
  void *field_1;
};
struct _conc__tuple2_0 {  // _tuple2_(int, int)
  int field_0;
  int field_1;
};
struct _conc__tuple2_0 _tmp_cp1;  // tuple argument to new
struct S *_tmp_cp_ret1;           // return value from new
void _thunk0(  // ?{}(S *, [int, int])
  struct S *_p0,
  struct _conc__tuple2_0 _p1
){
  _ctor_S(_p0, _p1.field_0, _p1.field_1);  // restructure tuple parameter
}
void _adapter(void (*_adaptee)(), void *_p0, void *_p1){
  // apply adaptee to arguments after casting to actual types
  ((void (*)(struct S *, struct _conc__tuple2_0))_adaptee)(
    _p0,
    *(struct _conc__tuple2_0 *)_p1
  );
}
struct S *s = (struct S *)(_tmp_cp_ret1=
  new(
    _adapter,
```

```
    sizeof(struct S),
    __alignof__(struct S),
    sizeof(struct _conc__tuple2_0),
    __alignof__(struct _conc__tuple2_0),
    (void (*)(void *, void *))&_thunk0,
    (({ // copy construct tuple argument to new
      int *__multassign_L0 = (int *)&_tmp_cp1.field_0;
      int *__multassign_L1 = (int *)&_tmp_cp1.field_1;
      int __multassign_R0 = 3;
      int __multassign_R1 = 4;
      ((*__multassign_L0=__multassign_R0 /* ?{} */) ,
       (*__multassign_L1=__multassign_R1 /* ?{} */));
    }), &_tmp_cp1)
  ), _tmp_cp_ret1);
*(struct S **)&_tmp_cp_ret1; // ^?{}  // destroy return value from new
({  // destroy argument temporary
  int *__massassign_L0 = (int *)&_tmp_cp1.field_0;
  int *__massassign_L1 = (int *)&_tmp_cp1.field_1;
  ((*__massassign_L0 /* ^?{} */) , (*__massassign_L1 /* ^?{} */));
});
```

Of note, `_thunk0` is generated to translate calls to `?{}(S *, [int, int])` into calls to `?{}(S *, int, int)`. The call to `new` constructs a tuple argument using the supplied arguments.

The `print` function

```
forall(otype T, ttype Params |
  { void print(T); void print(Params); })
void print(T arg, Params rest) {
  print(arg);
  print(rest);
}
```

generates the following

```
void print_variadic(
  void (*_adapterF_7tParams__P)(void (*)(), void *),
  void (*_adapterF_2tT__P)(void (*)(), void *),
  void (*_adapterF_P2tT2tT__MP)(void (*)(), void *, void *),
  void (*_adapterF2tT_P2tT2tT_P_MP)(void (*)(), void *, void *, void *),
  long unsigned int _sizeof_T,
  long unsigned int _alignof_T,
  long unsigned int _sizeof_Params,
  long unsigned int _alignof_Params,
  void *(*_assign_TT)(void *, void *),
  void (*_ctor_T)(void *),
  void (*_ctor_TT)(void *, void *),
  void (*_dtor_T)(void *),
  void (*print_T)(void *),
  void (*print_Params)(void *),
  void *arg,
  void *rest
){
  void *_tmp_cp0 = __builtin_alloca(_sizeof_T);
  _adapterF_2tT__P(  // print(arg)
```

```
    ((void (*)())print_T),
    (_adapterF_P2tT2tT__MP( // copy construct argument
      ((void (*)())_ctor_TT),
      _tmp_cp0,
      arg
    ), _tmp_cp0)
  );
  _dtor_T(_tmp_cp0);  // destroy argument temporary
  _adapterF_7tParams__P(  // print(rest)
    ((void (*)())print_Params),
    rest
  );
}
```

The `print_T` routine is called indirectly through an adapter function with a copy constructed argument, followed by an indirect call to `print_Params`.

A call to print

```
void print(const char * x) { printf("%s", x); }
void print(int x) { printf("%d", x);  }

print("x = ", 123, ".\n");
```

generates the following

```
void print_string(const char *x){
  int _tmp_cp_ret0;
  (_tmp_cp_ret0=printf("%s", x)) , _tmp_cp_ret0;
  *(int *)&_tmp_cp_ret0; // ^?{}
}
void print_int(int x){
  int _tmp_cp_ret1;
  (_tmp_cp_ret1=printf("%d", x)) , _tmp_cp_ret1;
  *(int *)&_tmp_cp_ret1; // ^?{}
}

struct _tuple2_ {  // _tuple2_(T0, T1)
  void *field_0;
  void *field_1;
};
struct _conc__tuple2_0 {  // _tuple2_(int, const char *)
  int field_0;
  const char *field_1;
};
struct _conc__tuple2_0 _tmp_cp6;  // _tuple2_(int, const char *)
const char *_thunk0(const char **_p0, const char *_p1){
        // const char * ?=?(const char **, const char *)
  return *_p0=_p1;
}
void _thunk1(const char **_p0){ // void ?{}(const char **)
  *_p0; // ?{}
}
void _thunk2(const char **_p0, const char *_p1){
        // void ?{}(const char **, const char *)
```

```
  *_p0=_p1; // ?{}
}
void _thunk3(const char **_p0){ // void ^?{}(const char **)
  *_p0; // ^?{}
}
void _thunk4(struct _conc__tuple2_0 _p0){
        // void print([int, const char *])
  struct _tuple1_ { // _tuple1_(T0)
    void *field_0;
  };
  struct _conc__tuple1_1 { // _tuple1_(const char *)
    const char *field_0;
  };
  void _thunk5(struct _conc__tuple1_1 _pp0){ // void print([const char *])
    print_string(_pp0.field_0);  // print(rest.0)
  }
  void _adapter_i_pii_(
    void (*_adaptee)(),
    void *_ret,
    void *_p0,
    void *_p1
  ){
    *(int *)_ret=((int (*)(int *, int))_adaptee)(_p0, *(int *)_p1);
  }
  void _adapter_pii_(void (*_adaptee)(), void *_p0, void *_p1){
    ((void (*)(int *, int ))_adaptee)(_p0, *(int *)_p1);
  }
  void _adapter_i_(void (*_adaptee)(), void *_p0){
    ((void (*)(int))_adaptee)(*(int *)_p0);
  }
  void _adapter_tuple1_5_(void (*_adaptee)(), void *_p0){
    ((void (*)(struct _conc__tuple1_1 ))_adaptee)(
      *(struct _conc__tuple1_1 *)_p0
    );
  }
  print_variadic(
    _adapter_tuple1_5,
    _adapter_i_,
    _adapter_pii_,
    _adapter_i_pii_,
    sizeof(int),
    __alignof__(int),
    sizeof(struct _conc__tuple1_1),
    __alignof__(struct _conc__tuple1_1),
    (void *(*)(void *, void *))_assign_i,    // int ?=?(int *, int)
    (void (*)(void *))_ctor_i,               // void ?{}(int *)
    (void (*)(void *, void *))_ctor_ii,      // void ?{}(int *, int)
    (void (*)(void *))_dtor_ii,              // void ^?{}(int *)
    (void (*)(void *))print_int,             // void print(int)
    (void (*)(void *))&_thunk5,              // void print([const char *])
    &_p0.field_0,                            // rest.0
    &(struct _conc__tuple1_1 ){ _p0.field_1 }// [rest.1]
  );
}
```

```
struct _tuple1_ {  // _tuple1_(T0)
  void *field_0;
};
struct _conc__tuple1_6 {  // _tuple_1(const char *)
  const char *field_0;
};
const char *_temp0;
_temp0="x = ";
void _adapter_pstring_pstring_string(
  void (*_adaptee)(),
  void *_ret,
  void *_p0,
  void *_p1
){
  *(const char **)_ret=
    ((const char *(*)(const char **, const char *))_adaptee)(
      _p0,
      *(const char **)_p1
    );
}
void _adapter_pstring_string(void (*_adaptee)(), void *_p0, void *_p1){
  ((void (*)(const char **, const char *))_adaptee)(
    _p0,
    *(const char **)_p1
  );
}
void _adapter_string_(void (*_adaptee)(), void *_p0){
  ((void (*)(const char *))_adaptee)(*(const char **)_p0);
}
void _adapter_tuple2_0_(void (*_adaptee)(), void *_p0){
  ((void (*)(struct _conc__tuple2_0 ))_adaptee)(
    *(struct _conc__tuple2_0 *)_p0
  );
}
print_variadic(
  _adapter_tuple2_0_,
  _adapter_string_,
  _adapter_pstring_string_,
  _adapter_pstring_pstring_string_,
  sizeof(const char *),
  __alignof__(const char *),
  sizeof(struct _conc__tuple2_0 ),
  __alignof__(struct _conc__tuple2_0 ),
  &_thunk0,     // const char * ?=?(const char **, const char *)
  &_thunk1,     // void ?{}(const char **)
  &_thunk2,     // void ?{}(const char **, const char *)
  &_thunk3,     // void ^?{}(const char **)
  print_string, // void print(const char *)
  &_thunk4,     // void print([int, const char *])
  &_temp0,                              // "x = "
  ({  // copy construct tuple argument to print
    int *__multassign_L0 = (int *)&_tmp_cp6.field_0;
    const char **__multassign_L1 = (const char **)&_tmp_cp6.field_1;
    int __multassign_R0 = 123;
```

```
    const char *__multassign_R1 = ".\n";
    ((*__multassign_L0=__multassign_R0 /* ?{} */),
     (*__multassign_L1=__multassign_R1 /* ?{} */));
  }), &_tmp_cp6)                          // [123, ".\n"]
);
({  // destroy argument temporary
  int *__massassign_L0 = (int *)&_tmp_cp6.field_0;
  const char **__massassign_L1 = (const char **)&_tmp_cp6.field_1;
  ((*__massassign_L0 /* ^?{} */) , (*__massassign_L1 /* ^?{} */));
});
```

The type `_tuple2_` is generated to allow passing the `rest` argument to `print_variadic`. Thunks 0 through 3 provide wrappers for the **`otype`** parameters for **`const char`** `*`, while `_thunk4` translates a call to `print([`**`int, const char`** `*])` into a call to `print_variadic(`**`int,`** `[`**`const char`** `*])`. This all builds to a call to `print_variadic`, with the appropriate copy construction of the tuple argument.

Chapter 5

Conclusions

Adding resource management and tuples to C∀ has been a challenging design, engineering, and implementation exercise. On the surface, the work may appear as a rehash of similar mechanisms in C++. However, every added feature is different than its C++ counterpart, often with extended functionality, better integration with C and its programmers, and always supports separate compilation. All of these new features are being used extensively by the C∀ development-team to build the C∀ runtime system. In particular, the concurrency system is built on top of RAII, library functions `new` and `delete` are used to manage dynamically allocated objects, and tuples are used to provide uniform interfaces to C library routines such as `div` and `remquo`.

5.1 Constructors and Destructors

C∀ supports the RAII idiom using constructors and destructors. There are many engineering challenges in introducing constructors and destructors, partially since C∀ is not an object-oriented language. By making use of managed types, C∀ programmers are afforded an extra layer of safety and ease of use in comparison to C programmers. While constructors and destructors provide a sensible default behaviour, C∀ allows experienced programmers to declare unmanaged objects to take control of object management for performance reasons. Constructors and destructors as named functions fit the C∀ polymorphism model perfectly, allowing polymorphic code to use managed types seamlessly.

5.2 Tuples

C∀ can express functions with multiple return values in a way that is simple, concise, and safe. The addition of multiple-return-value functions naturally requires a way to use multiple return values, which begets tuple types. Tuples provide two useful notions of assignment: multiple assignment, allowing simple, yet expressive assignment between multiple variables, and mass assignment, allowing a lossless assignment of a single value across multiple variables. Tuples have a flexible structure that allows the C∀ type-system to decide how to restructure tuples, making it syntactically simple to pass tuples between functions. Tuple types can be combined with polymorphism and tuple conversions can apply during assertion inference to produce a cohesive feel.

5.3 Variadic Functions

Type-safe variadic functions, with a similar feel to variadic templates, are added to C∀. The new variadic functions can express complicated recursive algorithms. Unlike variadic templates, it is possible to write `new` as a library routine and to separately compile **`ttype`** polymorphic functions. Variadic functions are statically type checked and provide a user experience that is consistent with that of tuples and polymorphic functions.

5.4 Future Work

5.4.1 Constructors and Destructors

Both C++ and Rust support move semantics, which expand the user's control of memory management by providing the ability to transfer ownership of large data, rather than forcing potentially expensive copy semantics. C∀ currently does not support move semantics, partially due to the complexity of the model. The design space is currently being explored with the goal of finding an alternative to move semantics that provides necessary performance benefits, while reducing the amount of repetition required to create a new type, along with the cognitive burden placed on the user.

Exception handling is among the features expected to be added to C∀ in the near future. For exception handling to properly interact with the rest of the language, it must ensure all RAII guarantees continue to be met. That is, when an exception is raised, it must properly unwind the stack by calling the destructors for any objects that live between the raise and the handler. This can be accomplished either by augmenting the translator to properly emit code that executes the destructors, or by switching destructors to hook into the GCC `cleanup` attribute [6, 6.32.1].

The `cleanup` attribute, which is attached to a variable declaration, takes a function name as an argument and schedules that routine to be executed when the variable goes out of scope.

```
struct S { int x; };
void __dtor_S(struct S *);
{
  __attribute__((cleanup(__dtor_S))) struct S s;
} // calls __dtor_S(&s)
```

This mechanism is known and understood by GCC, so that the destructor is properly called in any situation where a variable goes out of scope, including function returns, branches, and built-in GCC exception handling mechanisms using libunwind.

A caveat of this approach is that the `cleanup` attribute only permits a function that consumes a single argument of type `T *` for a variable of type `T`. This restriction means that any destructor that consumes multiple arguments (*e.g.*, because it is polymorphic) or any destructor that is a function pointer (*e.g.*, because it is an assertion parameter) must be called through a local thunk. For example,

```
forall(otype T)
struct Box {
  T x;
```

```
};
forall(otype T) void ^?{}(Box(T) * x); // has implicit parameters

forall(otype T)
void f(T x) {
  T y = x;  // destructor is a function-pointer parameter
  Box(T) z = { x }; // destructor has multiple parameters
}
```

currently generates the following

```
void _dtor_BoxT(  // consumes more than 1 parameter due to assertions
  void (*_adapter_PTT)(void (*)(), void *, void *),
  void (*_adapter_T_PTT)(void (*)(), void *, void *, void *),
  long unsigned int _sizeof_T,
  long unsigned int _alignof_T,
  void *(*_assign_T_PTT)(void *, void *),
  void (*_ctor_PT)(void *),
  void (*_ctor_PTT)(void *, void *),
  void (*_dtor_PT)(void *),
  void *x
);

void f(
  void (*_adapter_PTT)(void (*)(), void *, void *),
  void (*_adapter_T_PTT)(void (*)(), void *, void *, void *),
  long unsigned int _sizeof_T,
  long unsigned int _alignof_T,
  void *(*_assign_TT)(void *, void *),
  void (*_ctor_T)(void *),
  void (*_ctor_TT)(void *, void *),
  void (*_dtor_T)(void *),
  void *x
){
  void *y = __builtin_alloca(_sizeof_T);
  // constructor call elided

  // generic layout computation elided
  long unsigned int _sizeof_BoxT = ...;
  void *z = __builtin_alloca(_sizeof_BoxT);
  // constructor call elided

  _dtor_BoxT(  // ^?{}(&z); -- _dtor_BoxT has > 1 arguments
    _adapter_PTT,
    _adapter_T_PTT,
    _sizeof_T,
    _alignof_T,
    _assign_TT,
    _ctor_T,
    _ctor_TT,
    _dtor_T,
    z
  );
  _dtor_T(y);  // ^?{}(&y); -- _dtor_T is a function pointer
}
```

Further to this point, every distinct array type will require a thunk for its destructor, where array destructor code is currently inlined, since array destructors hard code the length of the array.

For function call temporaries, new scopes have to be added for destructor ordering to remain consistent. In particular, the translator currently destroys argument and return value temporary objects as soon as the statement they were created for ends. In order for this behaviour to be maintained, new scopes have to be added around every statement that contains a function call. Since a nested expression can raise an exception, care must be taken when destroying temporary objects. One way to achieve this is to split statements at every function call, to provide the correct scoping to destroy objects as necessary. For example,

```
struct S { ... };
void ?{}(S *, S);
void ^?{}(S *);

S f();
S g(S);

g(f());
```

would generate

```
struct S { ... };
void _ctor_S(struct S *, struct S);
void _dtor_S(struct S *);

{
  __attribute__((cleanup(_dtor_S))) struct S _tmp1 = f();
  __attribute__((cleanup(_dtor_S))) struct S _tmp2 =
    (_ctor_S(&_tmp2, _tmp1), _tmp2);
  __attribute__((cleanup(_dtor_S))) struct S _tmp3 = g(_tmp2);
} // destroy _tmp3, _tmp2, _tmp1
```

Note that destructors must be registered after the temporary is fully initialized, since it is possible for initialization expressions to raise exceptions, and a destructor should never be called on an uninitialized object. This requires a slightly strange looking initializer for constructor calls, where a comma expression is used to produce the value of the object being initialized, after the constructor call, conceptually bitwise copying the initialized data into itself. Since this copy is wholly unnecessary, it is easily optimized away.

A second approach is to attach an accompanying boolean to every temporary that records whether the object contains valid data, and thus whether the value should be destructed.

```
struct S { ... };
void _ctor_S(struct S *, struct S);
void _dtor_S(struct S *);

struct _tmp_bundle_S {
  bool valid;
  struct S value;
};
```

```
void _dtor_tmpS(struct _tmp_bundle_S * ret) {
  if (ret->valid) {
    _dtor_S(&ret->value);
  }
}

{
  __attribute__((cleanup(_dtor_tmpS))) struct _tmp_bundle_S _tmp1 = { 0 };
  __attribute__((cleanup(_dtor_tmpS))) struct _tmp_bundle_S _tmp2 = { 0 };
  __attribute__((cleanup(_dtor_tmpS))) struct _tmp_bundle_S _tmp3 = { 0 };
  _tmp2.value = g(
    (_ctor_S(
      &_tmp2.value,
      (_tmp1.value = f(), _tmp1.valid = 1, _tmp1.value)
    ), _tmp2.valid = 1, _tmp2.value)
  ), _tmp3.valid = 1, _tmp3.value;
} // destroy _tmp3, _tmp2, _tmp1
```

In particular, the boolean is set immediately after argument construction and immediately after return value copy. The boolean is checked as a part of the `cleanup` routine, forwarding to the object's destructor if the object is valid. One such type and `cleanup` routine needs to be generated for every type used in a function parameter or return value.

The former approach generates much simpler code, however splitting expressions requires care to ensure that expression evaluation order does not change. Expression ordering has to be performed by a full compiler, so it is possible that the latter approach would be more suited to the C∀ prototype, whereas the former approach is clearly the better option in a full compiler. More investigation is needed to determine whether the translator's current design can easily handle proper expression ordering.

As discussed in Section 2.1.6, return values are destructed with a different `this` pointer than they are constructed with. This problem can be easily fixed once a full C∀ compiler is built, since it would have full control over the call/return mechanism. In particular, since the callee is aware of where it needs to place the return value, it can construct the return value directly, rather than bitwise copy the internal data.

Currently, the special functions are always auto-generated, except for generic types where the type parameter does not have assertions for the corresponding operation. For example,

```
forall(dtype T | sized(T) | { void ?{}(T *); })
struct S { T x; };
```

only auto-generates the default constructor for `S`, since the member `x` is missing the other 3 special functions. Once deleted functions have been added, function generation can make use of this information to disable generation of special functions when a member has a deleted function. For example,

```
struct A {};
void ?{}(A *) = delete;
struct S { A x; };  // does not generate void ?{}(S *);
```

Unmanaged objects and their interactions with the managed C∀ environment are an open problem that deserves greater attention. In particular, the interactions between unmanaged objects

and copy semantics are subtle and can easily lead to errors. It is possible that the compiler should mark some of these situations as errors by default, and possibly conditionally emit warnings for some situations. Another possibility is to construct, destruct, and assign unmanaged objects using the intrinsic and auto-generated functions. A more thorough examination of the design space for this problem is required.

Currently, the C∀ translator does not support any warnings. Ideally, the translator should support optional warnings in the case where it can detect that an object has been constructed twice. For example, forwarding constructor calls are guaranteed to initialize the entire object, so redundant constructor calls can cause problems such as memory leaks, while looking innocuous to a novice user.

```
struct B { ... };
struct A {
  B x, y, z;
};
void ?{}(A * a, B x) {
  // y, z implicitly default constructed
  (&a->x){ ... }; // explicitly construct x
} // constructs an entire A
void ?{}(A * a) {
  (&a->y){}; // initialize y
  a{ (B){ ... } }; // forwarding constructor call
                   // initializes entire object, including y
}
```

Finally, while constructors provide a mechanism for establishing invariants, there is currently no mechanism for maintaining invariants without resorting to opaque types. That is, structure fields can be accessed and modified by any block of code without restriction, so while it is possible to ensure that an object is initially set to a valid state, it is not possible to ensure that it remains in a consistent state throughout its lifetime. A popular technique for ensuring consistency in object-oriented programming languages is to provide access modifiers such as `private`, which provides compile-time checks that only privileged code accesses private data. This approach could be added to C∀, but it requires an idiomatic way of specifying what code is privileged and what data is protected. One possibility is to tie access control into an eventual module system.

The current implementation of implicit subobject-construction is currently an all-or-nothing check. That is, if a subobject is conditionally constructed, *e.g.*, within an if-statement, no implicit constructors for that object are added.

```
struct A { ... };
void ?{}(A * a) { ... }

struct B {
  A a;
};
void ?{}(B * b) {
  if (...) {
    (&b->a){};  // explicitly constructed
  } // does not construct in else case
}
```

This behaviour is unsafe and breaks the guarantee that constructors fully initialize objects. This situation should be properly handled, either by examining all paths and inserting implicit constructor calls only in the paths missing construction, or by emitting an error or warning.

5.4.2 Tuples

Named result values are planned, but not yet implemented. This feature ties nicely into named tuples, as seen in D and Swift.

Currently, tuple flattening and structuring conversions are 0-cost conversions in the resolution algorithm. This makes tuples conceptually very simple to work with, but easily causes unnecessary ambiguity in situations where the type system should be able to differentiate between alternatives. Adding an appropriate cost function to tuple conversions will allow tuples to interact with the rest of the programming language more cohesively.

5.4.3 Variadic Functions

Use of **`ttype`** functions currently relies heavily on recursion. C++ has opened variadic templates up so that recursion is not strictly necessary in some cases, and it would be interesting to see if any such cases can be applied to C∀.

C++ supports variadic templated data-types, making it possible to express arbitrary length tuples, arbitrary parameter function objects, and more with generic types. Currently, C∀ does not support **`ttype`**-parameter generic-types, though there does not appear to be a technical reason that it cannot. Notably, opening up support for this makes it possible to implement the exit form of scope guard (see section 1.3), making it possible to call arbitrary functions at scope exit in idiomatic C∀.

References

[1] Andrei Alexandrescu and Petru Marginean. Generic: Change the way you write exception-safe code - forever, 2000. http://www.drdobbs.com/cpp/generic-change-the-way-you-write-excepti/184403758.

[2] Apple Inc. *The Swift Programming Language (Swift 3.1)*, 2017. https://developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/AboutTheLanguageReference.html.

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

[4] Walter Bright and Andrei Alexandrescu. *D Programming Language*. Digital Mars, 2016. http://dlang.org/spec/spec.html.

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

[6] C Extensions. Extensions to the C language family, 2014. https://gcc.gnu.org/onlinedocs/-gcc-4.7.2/gcc/CExtensions.html.

[7] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for java. *SIGPLAN Not.*, 34(10):1–19, October 1999.

[8] Glen Ditchfield. *Cforall Reference Manual and Rationale*, revision 1.82 edition, January 1998. ftp://plg.uwaterloo.ca/pub/Cforall/refrat.ps.gz.

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

[10] The GNU Project. *The Linux Programmer's Manual*, 2017. http://man7.org/linux/man-pages/man3/atexit.3.html.

[11] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *Java Language Spec.* Oracle, java se8 edition, 2015.

[12] Intermetrics, Inc. *Ada Reference Manual*, international standard ISO/IEC 8652:1995(E) with COR.1:2000 edition, December 1995. Language and Standards Libraries.

[13] International Standard ISO/IEC 14882:1998 (E), www.ansi.org. *Programming Languages – C++*, 1998.

[14] American national standard information technology – programming languages – C. International standard, International Standard Organization, http://www.iso.org, 2012.

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

[16] Julien Ponge. Better resource management with java se 7: Beyond syntactic sugar, 2011. http://www.oracle.com/technetwork/articles/java/trywithresources-401775.html.

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

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

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

[20] Herb Sutter, Bjarne Stroustrup, and Gabriel Dos Reis. Structured bindings. pages 1–6, October 2015. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/p0144r0.pdf.

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

[22] Guido van Rossum. *Python Reference Manual, Release 2.5*. Python Software Foundation, September 2006. Fred L. Drake, Jr., editor.