Cyclone User's Manual

Version 1.0, mai 14, 2019

The latest version of this manual should be available at
http://www.eecs.harvard.edu/~greg/cyclone/
http://www.research.att.com/projects/cyclone/ and
http://www.cs.umd.edu/projects/cyclone/.

Contents

1 Introduction **6**
- 1.1 Acknowledgements . . . 7

2 Cyclone for C Programmers **7**
- 2.1 Getting Started . . . 7
- 2.2 Pointers . . . 9
- 2.3 Regions . . . 18
- 2.4 Arrays . . . 29
- 2.5 Structs . . . 30
- 2.6 Unions . . . 31
- 2.7 C++, GCC and C99 Additions . . . 35
- 2.8 Additional Features of Cyclone . . . 36
- 2.9 Exceptions . . . 37
- 2.10 Let Declarations and Pattern Matching . . . 39
- 2.11 Subtyping . . . 40
- 2.12 Polymorphic Functions . . . 41
- 2.13 Polymorphic Data Structures . . . 43
- 2.14 Abstract and Existential Types . . . 43
- 2.15 Restrictions . . . 44

3 Pointers **47**
- 3.1 Pointer Subtyping . . . 50
- 3.2 Pointer Coercions . . . 51
- 3.3 Default Region Qualifiers . . . 53
- 3.4 Static Expression Bounds . . . 53

4 Tagged Unions and Datatypes **54**
- 4.1 Tagged Unions . . . 55
- 4.2 Datatypes . . . 56
- 4.3 The Tagof Operator . . . 60
- 4.4 Extensible Datatypes . . . 61

5 Pattern Matching **62**
- 5.1 Let Declarations . . . 62
- 5.2 Pattern Forms . . . 64
- 5.3 Switch Statements . . . 68

6 Type Inference **73**

7 Polymorphism **76**

8 Memory Management Via Regions **76**
8.1 Introduction . . . 76
8.2 Allocation . . . 79
8.3 Common Uses . . . 81
8.4 Type-Checking Regions . . . 86
8.4.1 Region Names . . . 86
8.4.2 Capabilities . . . 87
8.4.3 Type Declarations . . . 88
8.4.4 Subtyping and Effect Qualifiers . . . 88
8.4.5 Function Calls . . . 90
8.4.6 Explicit and Default Effects . . . 90

9 Pointers with Restricted Aliasing **91**
9.1 Introduction . . . 91
9.2 Allocation and Freeing . . . 92
9.3 Unique Pointers . . . 94
9.3.1 Simple Unique Pointers . . . 95
9.3.2 Nested Unique Pointers . . . 97
9.3.3 Pattern Matching on Unique Pointers . . . 98
9.4 Reference-counted Pointers . . . 100
9.5 Qualifier Polymorphism . . . 101
9.6 Aliasing Unique Pointers . . . 103
9.7 Dynamic Regions . . . 105
9.8 Defaults and Shorthands . . . 108
9.9 Subtyping for Effect Qualifier and Reaps . . . 111
9.10 Reap Allocator Implementation . . . 112

10 Namespaces **113**

11 Varargs **115**

12 Definite Assignment **117**

13 Advanced Features **123**
13.1 Existential Types . . . 124
13.2 The Truth About Effects, Capabilities and Effect Subset Constraints . . . 125
13.3 Interprocedural Memory Initialization . . . 128

A Porting C code to Cyclone **129**
A.1 Semi-Automatic Porting . . . 129
A.2 Manually Translating C to Cyclone . . . 131
A.3 Interfacing to C . . . 140
A.3.1 Extern "C" . . . 140
A.3.2 Extern "C include" . . . 141

B Frequently Asked Questions **144**

C Libraries **162**
C.1 C Libraries . . . 162
C.2 `<array.h>` . . . 163
C.3 `<bitvec.h>` . . . 168
C.4 `<buffer.h>` . . . 169
C.5 `<core.h>` . . . 171
C.6 `<dict.h>` . . . 179
C.7 `<filename.h>` . . . 187
C.8 `<fn.h>` . . . 188
C.9 `<graph.h>` . . . 190
C.10 `<hashtable.h>` . . . 192
C.11 `<iter.h>` . . . 194
C.12 `<list.h>` . . . 194
C.13 `<pp.h>` . . . 206
C.14 `<queue.h>` . . . 209
C.15 `<rope.h>` . . . 211
C.16 `<set.h>` . . . 212
C.17 `<slowdict.h>` . . . 216
C.18 `<xarray.h>` . . . 219

D Grammar **222**

E Installing Cyclone **235**

F Tools **235**
F.1 The compiler 235
F.2 The lexer generator 238
F.3 The parser generator 238
F.4 The allocation profiler, `aprof` 238
F.5 The C interface tool, `buildlib` 239

1 Introduction

Cyclone is a language for C programmers who want to write secure, robust programs. It's a dialect of C designed to be *safe*: free of crashes, buffer overflows, format string attacks, and so on. Careful C programmers can produce safe C programs, but, in practice, many C programs are unsafe. Our goal is to make *all* Cyclone programs safe, regardless of how carefully they were written. All Cyclone programs must pass a combination of compile-time, link-time, and run-time checks designed to ensure safety.

There are other safe programming languages, including Java, ML, and Scheme. Cyclone is novel because its syntax, types, and semantics are based closely on C. This makes it easier to interface Cyclone with legacy C code, or port C programs to Cyclone. And writing a new program in Cyclone "feels" like programming in C: Cyclone tries to give programmers the same control over data representations, memory management, and performance that C has.

Cyclone's combination of performance, control, and safety make it a good language for writing systems and security software. Writing such software in Cyclone will, in turn, motivate new research into safe, low-level languages. For instance, originally, all heap-allocated data in Cyclone were reclaimed via a conservative garbage collector. Though the garbage collector ensures safety by preventing programs from accessing deallocated objects, it also kept Cyclone from being used in latency-critical or space-sensitive applications such as network protocols or device drivers. To address this shortcoming, we have added a region-based memory management system based on the work of Tofte and Talpin. The region-based memory manager allows you some real-time control over memory management and can significantly reduce space overheads when compared to a conventional garbage collector. Furthermore, the region type system ensures the same safety properties as a collector: objects cannot be accessed outside of their lifetimes.

This manual is meant to provide an informal introduction to Cyclone. We have tried to write the manual from the perspective of a C programmer who wishes either to port code from C to Cyclone, or develop a new system using Cyclone. Therefore, we assume a fairly complete understanding of C.

Obviously, Cyclone is a work in progress and we expect to make substantial changes to the design and implementation. Your feedback (and

patience) is greatly appreciated.

1.1 Acknowledgements

The people involved in the development of Cyclone are now at Harvard, AT&T, Maryland, and Washington; much work began at Cornell. Dan Grossman, Trevor Jim, and Greg Morrisett worked out the initial design and implementation, basing the language to some degree on Popcorn, a safe-C-like language that was developed at Cornell as part of the Typed Assembly Language (TAL) project. Mike Hicks ported a number of libraries and programs to Cyclone, helped with the configuration and installation procedures, and has been the lead on adding unique and reference-counted pointers to Cyclone, among other things. Mathieu Baudet contributed the bulk of the code for the link-checker. Matthew Harris did much of the hard work needed to wrap and import the necessary libraries. Yanling Wang ported bison and flex to Cyclone. James Cheney has added support for representation types, singleton ints, marshalling support, etc. Nikhil Swamy added support for reaps and the cyclone-inf mode. All of these people have also contributed by finding and fixing various bugs. A number of other people have also helped to find bugs and/or contributed key design ideas including Mujtaba Ali, Fred Smith, Nathan Lutchansky, Rajit Manohar, Bart Samwell, Emmanuel Schanzer, Frances Spalding, Jeff Vinocur, and David Walker.

2 Cyclone for C Programmers

We begin with a quick overview of Cyclone, suitable for those who already know how to program in C. We'll explain some of the ways that Cyclone differs from C and some of the reasons why; you should come away with enough knowledge to start writing, compiling, and running your own Cyclone programs. We assume that the Cyclone compiler is already installed on your system (see Appendix E if you need to install the compiler).

2.1 Getting Started

Here's a Cyclone program that prints the string "`hello, world`."

```
#include <stdio.h>

int main() {
  printf("hello, world\n");
  return 0;
}
```

It looks rather like a C program—in fact, a C compiler will happily compile it. The program uses `#include` to tell the preprocessor to import some standard definitions, it defines a distinguished function `main` that serves as the entry point of the program, and it uses the familiar `printf` function to handle the printing; all of this is just as in C.

To compile the program, put it into a file `hello.cyc`, and run the command

```
cyclone -o hello hello.cyc
```

This tells the Cyclone compiler (`cyclone`) to compile the file `hello.cyc`; the `-o` flag tells the compiler to leave the executable output in the file `hello` (or, in Windows, `hello.exe`). If all goes well you can execute the program by typing

```
hello
```

and it will print

```
hello, world
```

It's interesting to compare our program with a version that omits the return statement:

```
#include <stdio.h>

int main() {
  printf("hello, world\n");
}
```

A C compiler will compile and run this version without warning. In contrast, Cyclone will warn that you have failed to return an `int`. Cyclone only warns you when you fail to return an integral type (`char`, `short`, `int`, etc.) but it gives an error if you fail to return other types (e.g., pointer

types). This requirement of *definite return* ensures type safety while imposing minimal constraints on a programmer porting C code to Cyclone.

Definite return reflects Cyclone's concern with safety. The caller of the function expects to receive a value of the return type; if the function does not execute a `return` statement, the caller will receive some incorrect value instead. If the returned value is supposed to be a pointer, the caller might try to dereference it, and dereferencing an arbitrary address can cause the program to crash. So, Cyclone requires a return statement with a value of the return type whenever type safety can be compromised.

2.2 Pointers

Programs that use pointers properly in C can be both fast and elegant. But when pointers are used improperly in C, they cause core dumps and buffer overflows. To prevent this, Cyclone introduces different kinds of pointers and either (a) puts some restrictions on how you can use pointers of a given kind or (b) places no restrictions but may insert additional run-time checks. Here we present a basic overview of Cyclone pointers; they are summarized and covered in more detail in Section 3.

Nullable Pointers

The first kind of pointer is indicated with a `*`, as in C. For example, if we declare

```
int x = 3;
int *y = &x;
```

then `y` is a pointer to the integer 3 (the contents of `x`). The pointer, `y`, is represented by a memory address, namely, the address of `x`. To refer to the contents of `y`, you use `*y`, so, for example, you can increment the value of `x` with an assignment like

```
*y = *y + 1;
```

This much is just as in C. However, there are some differences in Cyclone:

- You can't cast an integer to a pointer. Cyclone prevents this because it would let you overwrite arbitrary memory locations. In Cyclone, `NULL` is a keyword suitable for situations where you would use a (casted) `0` in C. The compiler accepts `0` as a legal possibly-null pointer value, but using `NULL` is preferred.

- You can't do pointer arithmetic on a `*` pointer. Pointer arithmetic in C can take a pointer out of bounds, so that when the pointer is eventually dereferenced, it corrupts memory or causes a crash. (However, pointer arithmetic is possible using `@fat` and `@zeroterm` pointers.)

- There is one other way to crash a C program using pointers: you can dereference the `NULL` pointer or try to update the `NULL` location. Cyclone prevents this by inserting a null check whenever you dereference a `*` pointer (that is, whenever you use the `*`, `->`, or subscript operation on a pointer.)

These are drastic differences from C, particularly the restriction on pointer arithmetic. The benefit is that you can't cause a crash using `*` pointers in Cyclone.

Fat Pointers

If you need to do pointer arithmetic in Cyclone, you can use a second kind of pointer, called a *fat pointer* and indicated by writing the qualifier `@fat` after the `*`. For example, here is a program that echoes its command-line arguments:

```
#include <stdio.h>

int main(int argc, char *@fat *@fat argv) {
  argc--; argv++; /* skip command name */
  if (argc > 0) {
    /* print first arg without a preceding space */
    printf("%s",*argv);
    argc--; argv++;
  }
  while (argc > 0) {
    /* print other args with a preceding space */
```

```
    printf(" %s",*argv);
    argc--; argv++;
  }
  printf("\n");
  return 0;
}
```

Except for the declaration of `argv`, which holds the command-line arguments, the program looks just like you would write it in C: pointer arithmetic (`argv++`) is used to move `argv` to point to each argument in turn, so it can be printed.

In C, `argv` would typically be declared with type `char **`, a pointer to a pointer to a character, which is thought of as an array of an array of characters. In Cyclone, `argv` is instead declared with type `char *@fat*@fat`, which is thought of in the same way: it is a (fat) pointer to a (fat) pointer to characters. The difference between an unqualified pointer and a `@fat` pointer is that a `@fat` pointer comes with bounds information and is thus "fatter" than a traditional pointer. Each time a fat pointer is dereferenced or its contents are assigned to, Cyclone inserts both a null check and a bounds check. This guarantees that a `@fat` pointer can never cause a buffer overflow.

Because of the bounds information contained in `@fat` pointers, `argc` is superfluous: you can get the size of `argv` by writing `numelts(argv)`. We've kept `argc` as an argument of `main` for backwards compatibility.

It's worth remarking that you can always cast a `*` pointer to a `@fat` pointer (and vice-versa). So, it is possible to do pointer arithmetic on a value of type `*`, but only when you insert the appropriate casts to convert from one pointer type to another. Note that some of these casts can fail at run-time. For instance, if you try to cast a fat pointer that points to an empty sequence of characters to `char *`, then the cast will fail since the sequence doesn't contain at least one character.

Finally, `@fat` pointers are used so frequently in Cyclone, that there is special character, `?` (question mark) that you can use as an abbreviation for `*@fat`. For instance, we could write the prototype for main as:

```
int main(int argc, char ?? argv);
```

instead of the more verbose:

```
int main(int argc, char *@fat *@fat argv);
```

Non-NULL Pointers

Another kind of pointer in Cyclone is the non-NULL pointer. A non-NULL pointer is indicated by the qualifier `@notnull`. A `@notnull` pointer is like an unqualified pointer, except that it is guaranteed not to be NULL. This means that when you dereference a `@notnull` pointer or assign to its contents, a null check is sometimes unnecessary.

`@notnull` pointers are useful in Cyclone both for efficiency and as documentation. This can be seen at work in the standard library, where many functions take `@notnull` pointers as arguments, or return `@notnull` pointers as results. For example, the `getc` function that reads a character from a file is declared,

```
int getc(FILE *@notnull);
```

This says that `getc` expects to be called with a non-NULL pointer to a `FILE`. Cyclone guarantees that, in fact, when the `getc` function is entered, its argument is not `NULL`. This means that `getc` does not have to test whether it is `NULL`, or decide what to do if it is in fact `NULL`.

In C, the argument of `getc` is declared to have type `FILE *`, and programmers can call `getc` with `NULL`. So for safety, C's `getc` ought to check for `NULL`. In practice, many C implementations omit the check; `getc(NULL)` is an easy way to crash a C program.

In Cyclone, you can still call `getc` with a possibly-`NULL` `FILE` pointer (a `FILE *`). However, Cyclone insists that you insert a check before the actual call:

```
FILE *f = fopen("/etc/passwd","r");
int c = getc((FILE *@notnull)f);
```

Here `f` will be `NULL` if the file `/etc/passwd` doesn't exist or can't be read. So, in Cyclone `f` must be cast to `FILE *@notnull` before the call to `getc`. The cast causes a null check. If you try to call `getc` without the cast, Cyclone will insert one for you automatically, and warn you that it is doing so.

These warnings do not mean that your program is unsafe—after all, Cyclone has inserted the check for you. However, you should pay attention to the warnings because they indicate a place where your program could suddenly halt (if the check fails), and because the inserted checks can slow down your program. It's worth rewriting your code to handle

the error case better, or even eliminate the null check. For instance, if we rewrite the code above so that we explicitly test whether or not `fopen` succeeds in returning a non-`NULL` file descriptor:

```
FILE *f = fopen("/etc/passwd","r");
if (f == NULL) {
  fprintf(stderr,"cannot open passwd file!");
  exit(-1);
}
int c = getc(f);
```

then Cyclone no longer issues a warning at the call to `getc` and the resulting code does not have to do a null check.

If you call `getc` with a `FILE *@notnull`, of course, no check is required. For example, `stdin` is a `FILE *@notnull` in Cyclone, so you can simply call `getc(stdin)`. In Cyclone you will find that many functions return `*@notnull` pointers, so many of the pointers you deal with will already be `*@notnull` pointers, and neither the caller nor the called function needs to do null checks—and this is perfectly safe.

Like `@fat` pointers, `@notnull` pointers are so useful, Cyclone provides an abbreviation. Instead of writing `FILE *@notnull`, you can simply write `FILE @` when you want to write the type of a non-`NULL` pointer to a `FILE`.

Zero-Terminated Pointers

Fat pointers support arbitrary pointer arithmetic and subscripting, but they don't have the same representation as pointers in C. This is because we need extra information to determine the bounds and ensure that a subscript or dereference is in bounds. Unfortunately, this change in representations can make it difficult to interface with legacy C code where the representations might not be easily changed.

Fortunately, Cyclone supports one more pointer type where the representation matches C's and yet supports a limited form of pointer arithmetic and subscripting: the zero-terminated pointer. A zero-terminated pointer is a pointer to a sequence of elements that are guaranteed to be terminated with a zero. C's strings are a good example. In Cyclone, the type of C's strings can be written as `char *@zeroterm`. The `@zeroterm` qualifier indicates that the pointer points to a zero-terminated sequence.

The qualifier is orthogonal to other qualifiers, such as `@fat` or `@notnull`, so you can freely combine them.

Because C strings arise so frequently, the types `char *`, `char *@notnull`, and `char *@fat` are by default qualified with `@zeroterm`. You can override the `@zeroterm` qualifier on `char` pointers by putting in an explicit `@nozeroterm` qualifier (e.g., `char *@nozeroterm`). Pointers to other types (e.g., `int *`) have a default qualifier of `@nozeroterm`.

If `x` is a `* @zeroterm` pointer, you can use pointer arithmetic on it, as in `x+i`. However, the compiler inserts checks to ensure that (a) `i` is non-negative and (b) there is no zero between `x[0]` and `x[i-1]` inclusive. This ensures that you can't read past the terminating zero. In addition, when writing to a zero-terminated pointer, the compiler inserts checks to ensure that you don't replace the final zero with some other value. This is crucial for ensuring that a buffer overrun cannot occur. As in C, `x[i]` is equivalent to `x+i`, so subscripts come with the same checks.

Because of these checks, subscripts and pointer arithmetic on `* @zeroterm` can be fairly expensive. In particular, if you are not careful, you can turn what appears to be an *O(n)* algorithm into an *O(n-squared)* one. You can avoid this overhead by casting the pointer to a `@fat` zero-terminated pointer. This computes the length of the sequence once and then uses the bounds information associated with the fat pointer to do any bounds checks.

Cyclone's constraints on zero-terminated pointers mean that you have to be careful when porting code from C. For instance, consider the following function:

```
void foo(char *s, int offset) {
   unsigned int len = strlen(s);
   for (unsigned int i = 0; offset+i < len; i++)
      s[offset+i] = 'a';
}
```

This code can be quite expensive when `offset` is large because the compiler must check that there is no intervening zero between `s[0]` and `s[offset+i]` for each iteration of the loop. You can get rid of this overhead by rewriting the code as follows:

```
void foo(char *s, int offset) {
   unsigned int len = strlen(s);
   s = s + offset;
```

```
    for (unsigned int i = 0; offset+i < len; i++, s++)
      *s = 'a';
  }
```

Now the compiler is only checking that `*s` is not zero when it does the increment `s++`. In addition, however, the compiler is checking each time you do `*s = 'a'` that `*s` is not zero, because then you could overwrite the zero with an `'a'` and potentially step outside the bounds of the buffer.

One way to get rid of all of these checks is to cast `s` to a non-zero-terminated fat pointer before entering the loop. When you cast a zero-terminated pointer to a non-zero-terminated fat pointer, the compiler calculates the length of the sequence once, decrements it by one, and then builds an appropriate fat pointer with this bounds information. When you write using the fat pointer, bounds checks (not zero checks) keep you from writing any value over the zero. Furthermore, if you write the code in a straightforward fashion using subscripting, the compiler is more likely to eliminate the bounds checks. Here is an example:

```
  void foo(char *s, int offset) {
    char *@fat @nozeroterm fat_s = (char *@fat @nozeroterm)s;
    unsigned int len;
    fat_s += offset;
    len = numelts(fat_s);
    for (unsigned int i = 0; i < len; i++)
      fat_s[i] = 'a';
  }
```

The Cyclone compiler generates code that works like the following C code:

```
struct _tagged_arr {
  char *base;
  char *curr;
  char *last;
};

void Cyc_foo(char *s,int offset){
  struct _tagged_arr fat_s = {s, s, s+strlen(s)};
  unsigned int len;
  fat_s.curr += offset;
```

```
  if (fat_s.curr < fat_s.base || fat_s.curr >= fat_s.last)
    len = 0;
  else
    len = fat_s.last - fat_s.curr;
  { unsigned int i = 0;
    for(0; i < len; i++)
      fat_s.curr[i] = 'a';
  }
}
```

Notice that here, the compiler is able to eliminate all bounds checks within the loop and still ensure safety.

Bounded Pointers

A pointer type can also specify that it points to a sequence of a particular (statically known) length using the `@numelts` qualifier. For instance, we can write:

```
void foo(int *@numelts(4) arr);
```

Here, the parameter `arr` is a pointer to a sequence of four integer values. Both the non-null and nullable pointers (but not fat pointers) support explicit sequence bounds that are tracked statically. Indeed, both pointer kinds always have length information and when you write "`int *`" this is just short-hand for "`int *@numelts(1)`".

Bounded pointers are most often constructed from arrays. In particular, whenever you pass an array as a parameter to a function, it is *promoted* automatically to a pointer, following the rules of C. This pointer will have a sequence bound that is the same as the length of the array. Here is an example of calling `foo` above:

```
int x[4] = {1,2,3,4};
int y[8] = {1,2,3,4,5,6,7,8};
foo(x);
foo(y);
```

In the first call, the parameter `x` being passed to `foo` is automatically cast to type `int *@numelts(4)`, which is the type expected by `foo`. In the second case, the type of `y` is automatically cast to type `int *@numelts(8)`.

Since $8 \geq 4$, the call is safe and so Cyclone accepts it but emits a warning "implicit cast to shorter array." Finally, the following code will be rejected, because the pointer being passed is too short:

```
int bad[2] = {1,2};
foo(bad); // does not typecheck
```

Finally, bounded pointers can also be used to correlate a pointer to an array whose length is not known statically with a variable that defines it. For example, C programmers often write code like the following:

```
int sum(int num, int *p) {
  int a = 0;
  for (unsigned i = 0; i < num; i++)
    a += p[i];
}
```

Here, `num` is the length of the array pointed at by `p`. In Cyclone, this relationship can be expressed by giving `sum` the following type (the body of the function is the same):

```
int sum(tag_t<'n> num,
        int *@notnull @numelts(valueof('n)) p) {
```

The type of `num` is specified as `tag_t<'n>`. This simply means that `num` holds an integer value, called `'n`, and the number of elements of `p` is equal to `n`. This form of dependency is common enough that it can be abbreviated as follows:

```
int sum(tag_t num, int p[num]);
```

and the compiler will fill in the missing information.

A bounded pointer paired with a `tag_t` is quite similar to a fat pointer. Indeed, you can convert between the two using the library functions `mkfat` and `mkthin`. See Appendix C for a further description.

Initializing Pointers

Pointers must be initialized before they are used to ensure that unknown bits do not get used as a pointer. This requirement goes for variables that have pointer type, as well for arrays, elements of arrays, and for fields in

structures. Conversely, data that does not have pointer type need not be initialized before it is used, since doing so cannot result in a violation of safety. This decision adheres to the philosophy of C, but diverges from that of traditional type-safe languages like Java and ML. The rules for initialization of pointers used by Cyclone are described in Section 12.

2.3 Regions

Another potential way to crash a program or violate security is to dereference a *dangling pointer*, which is a pointer to storage that has been deallocated. These are particularly insidious bugs because the error might not manifest itself immediately. For example, consider the following C code:

```
struct Point {int x; int y;};

struct Point *newPoint(int x,int y) {
  struct Point result = {x,y};
  return &result;
}

void foo(struct Point *p) {
  p->y = 1234;
  return;
}

void bar() {
  struct Point *p = newPoint(1,2);
  foo(p);
}
```

The code has an obvious bug: the function `newPoint` returns a pointer to a locally-defined variable (`result`), even though the storage for that variable is deallocated upon exit from the function. That storage may be reused (*e.g.*, by a subsequent procedure call) leading to subtle bugs or security problems. For instance, in the code above, after `bar` calls `newPoint`, the storage for the point is reused to store information for the activation record of the call to `foo`. This includes a copy of the pointer `p` and the return address of `foo`. Therefore, it may be that `p->y` actually points to

the return address of `foo`. The assignment of the integer 1234 to that location could then result in `foo` "returning" to an arbitrary hunk of code in memory. Nevertheless, the C type-checker readily admits the code.

In Cyclone, this code would be rejected by the type-checker to avoid the kind of problems mentioned above. The reason the code is rejected is that the Cyclone compiler tracks object lifetimes and ensures that a pointer to an object can only be dereferenced if that object has not been deallocated.

Cyclone achieves this by assigning each object a symbolic *region* that corresponds to the area of memory in which the object is allocated. Cyclone also tracks, for every pointer, what region it points into. The region pointed to can be written as part of the pointer type, but usually the region can be omitted—the compiler is smart enough to discover the region automatically in many cases.

For example, the variable `result` in our code above lives within a region that corresponds to the invocation of the function `newPoint`. We write the name of the region explicitly using a back-quote, as in `'newPoint`. Because `result` lives in region `'newPoint`, the expression `&result` is a pointer into region `'newPoint`. The full Cyclone type of `&result`, with the explicit region, is `struct Point * @region('newPoint)`.

A region that corresponds to a lexical block, like an activation record, is referred to as a *stack region*, since it corresponds to a piece of the runtime stack. When control flow exits a block, the storage (*i.e.*, the stack region) for that block is deallocated. Cyclone keeps track of the set of regions that are allocated and deallocated at every control-flow point and ensures that you only dereference pointers to allocated regions. For example, consider the following fragment of (bad) Cyclone code:

```
1 int f() {
2    int x = 0;
3    int *@region('f) y = &x;
4    L:{ int a = 0;
5        y = &a;
6      }
7    return *y;
8 }
```

In the function `f` above, the variables `x` and `y` live within the region `'f` because they are declared in the outermost block of the function, and be-

cause the default region name for the block of a function is `'<function name>`. The storage for those variables will live as long as the invocation of the function. Note that since `y` is a pointer to `x`, the type of `y` is `int *@region('f)`, indicating that `y` points into region `'f`.

The variable `a` does *not* live in region `'f`, because it is declared in an inner block, which we have labeled with `L`. The storage for the inner block `L` may be deallocated upon exit of the block, before the function itself returns. To be more precise, the storage for `a` is deallocated at line 7 in the code. Thus, it is an error to try to access this storage in the rest of the computation, as is done on line 7.

Cyclone detects the error because it gives the expression `&a` the type `int *@region('L)`, meaning that the value is a pointer into region `'L`. So, the assignment `y = &a` fails to type-check because `y` expects to hold a pointer into region `'f`, not region `'L`. The restriction, compared to C, is that a pointer's type indicates *one* region instead of *all* regions.

Region Inference

If you had to write a `@region` qualifier on every pointer type, then writing code would be unpleasant. Fortunately, Cyclone provides a number of mechanisms to cut down on the region annotations you have to write.

First off, you can omit the `@region` qualifier keyword and simply write the region name (e.g., `'r`) as long as you put the region name after any other qualifiers. For instance, instead of writing "`int *@notnull @region('r)`" we can simply write "`int @'r`". For clarity, we will often use an explicit `@region` qualifier, but you'll find that the libraries and other example programs frequently use the abbreviations.

In addition, Cyclone often figures out the region of a pointer without the programmer providing the information. This is called *region inference*. For instance, we can rewrite the function `f` above without any region annotations, and without labeling the blocks:

```
1 int f() {
2    int x = 0;
3    int *y = &x;
4    { int a = 0;
5      y = &a;
6    }
```

```
7    return *y;
8 }
```

Cyclone can still figure out that `y` is a pointer into region `'f`, and `&a` is a pointer into a different (now anonymous) region, so the code should be rejected.

As we will show below, occasionally you will need to put explicit region annotations into the code to convince the type-checker that something points into a particular region, or that two things point into the same region. In addition, it is sometimes useful to put in the region annotations for documentation purposes, or to make type errors less cryptic. See Section 6 for more information about region inference.

You need to understand a few more details about regions to be an effective Cyclone programmer: the heap region, LIFO regions, region polymorphism, and default region annotations for function parameters. The following sections give a brief overview of these topics. Information about additional region-based constructs, including the unique and reference-counted regions, and dynamic regions, can be found in Section 8.

The Heap Region

There is a special region for the heap, written `'H`, that holds all of the storage for top-level variables, and for data allocated via `new` or `malloc`. For instance, if we write the following declarations at the top-level:

```
struct Point p = {0,1};
struct Point *ptr = &p;
```

then Cyclone figures out that `ptr` points into the heap region. To reflect this explicitly, we can put the region in the type of `ptr` if we like:

```
struct Point p = {0,1};
struct Point *@region('H) ptr = &p;
```

As another example, the following function heap-allocates a `Point` and returns it to the caller. We put the regions in here to be explicit:

```
struct Point *@region('H) good_newPoint(int x,int y) {
  struct Point *@region('H) p =
    malloc(sizeof(struct Point));
```

```
  p->x = x;
  p->y = y;
  return p;
}
```

Alternatively, we can use `new` to heap-allocate and initialize the result:

```
struct Point *@region(‘H) good_newPoint(int x,int y) {
  return new Point{x,y};
}
```

LIFO Regions

Storage on the stack is implicitly allocated and recycled when you enter and leave a block. Storage in the heap is explicitly allocated via `new` or `malloc`, but there is no support in Cyclone for explicitly freeing an object in the heap. The reason is that Cyclone cannot in general track the lifetimes of individual objects within the heap accurately, so it can't be sure whether dereferencing a pointer into the heap would cause problems. Instead, unless otherwise specified, a conservative garbage collector is used to reclaim the data allocated in the heap. We also support *unique pointers* and *reference-counted pointers* that programmers can safely free manually, but we defer discussion of these features until Section 8.

Using a garbage collector to recycle memory is the right thing to do for most applications. For instance, the Cyclone compiler uses heap-allocated data and relies upon the collector to recycle most objects it creates when compiling a program. But a garbage collector can introduce pauses in the program, and as a general purpose memory manager, might not be as space- or time-efficient as routines tailored to an application.

To address these applications, Cyclone provides support for *LIFO regions*. A LIFO region is similar to a stack region in that its lifetime is lexically-scoped (last-in-first-out, or LIFO) but permits dynamic allocation. Consider the following syntax:

```
{ region<‘r> h;
   ...
}
```

This declares a new region `‘r` along with a *region handle* `h`. The handle can be used for dynamically-allocating objects within the region `‘r`. Like

a stack region, all of the storage for the region is deallocated at the point of the closing brace. Unlike stack regions, the number (and size) of objects that you allocate into a LIFO region is not fixed at compile time. In this respect, LIFO regions are more like the heap. You can use the `rnew(h)` and `rmalloc(h,...)` operations to allocate objects within a growable region, where `h` is the handle for the region.

For instance, the following code takes an integer n, creates a new dynamic region and allocates an array of size n within the region using `rnew`.

```
int k(int n) {
  int result;
  { region<‘r> h;
    int ?arr = rnew(h) {for i < n : i};
    result = process(h, arr);
  }
  return result;
}
```

It then passes the handle for the region and the array to some processing function. Note that the processing function is free to allocate objects into the region `‘r` using the supplied handle. After processing the array, we exit the region which deallocates the array, and then return the calculated result.

LIFO regions are particularly useful in this circumstance; i.e. when you want the result of a function call to be allocated in the caller, but you don't know how much space you'll need.l An another example, consider the following usage of the Cyclone library function `rprintf`:

```
  { region<‘r> h;
    char ?‘H s = get_username();
    char ?‘r z = rprintf(h,"hello %s\n");
    emit(z);
  }
```

After allocating the region and acquring a user's name, the region handle is passed to the library function `rprintf`. `rprintf` is like `sprintf`, except that it does not print to a fixed-sized buffer; instead it allocates a buffer in a region, places the formatted output in the buffer, and returns a pointer to the buffer. In the example above, `z` is initialized with a pointer

to the string "hello" followed by the user's name; `z` is allocated in `h`'s region. Unlike `sprintf`, there is no risk of a buffer overflow, and unlike `snprintf`, there is no risk of passing a buffer that is too small. Moreover, the allocated buffer will be freed when the region goes out of scope, just as a stack-allocated buffer would be.

Finally, it is worth remarking that it may help to think of the heap as just a LIFO region with unbounded scope. Indeed, you can use the global variable `Core::heap_region` as a handle on the heap, and `new` and `malloc(...)` are just abbreviations for `rnew(Core::heap_region)` and `rmalloc(Core::heap_region,...)` respectively.

Region Polymorphism

Another key concept you need to understand is called *region polymorphism*. This is just a fancy way of saying that you can write functions in Cyclone that don't care which specific region a given object lives in, as long as it's still alive. For example, the function `foo` from the beginning of this section is a region-polymorphic function. To make this clear, let us re-write the `foo` function (page 18) making the region of its argument explicit:

```
void foo(struct Point *@region(‘r) p) {
  p->y = 1234;
  return;
}
```

The function is parameterized by a *region variable* `‘r`, and accepts a pointer to a `Point` that lives in region `‘r`. When calling `foo`, `‘r` can be *instantiated* with any region you like, including the heap `‘H`, or a region local to a function. So, for instance, we can write the following:

```
void g() {
  struct Point p = {0,1};
  struct Point *@region(‘g) ptr1 = &p;
  struct Point *@region(‘H) ptr2 = new Point{2,3};
  foo(ptr1);
  foo(ptr2);
}
```

Note that in the first call to `foo`, we are passing a pointer into region `‘g`, and in the second call to `foo`, we are passing in a pointer into the heap. In

the first call, `'r` is implicitly instantiated with `'g`, and in the second call, with `'H`.

Region polymorphism is a specific form of *paramteric polymorphism* which Cyclone supports in general, as we describe below.

Default Region Annotations

Cyclone automatically assigns region variables to function arguments that have pointer type, so you rarely have to write them. For instance, `foo` can be written simply as:

```
void foo(struct Point * p) {
  p->y = 1234;
  return;
}
```

As another example, if you write the following:

```
void h(struct Point * p1, struct Point * p2) {
  p1->x += p2->x;
  p2->x += p2->y;
}
```

then Cyclone fills in the region parameters for you by assuming that the points `p1` and `p2` can live in any two regions, and so it generates assigns fresh names for the region variables of `p1` and `p2`, e.g. something like `'r1` and `'r2`. To make this explicit, we would write:

```
void h(struct Point *@region('r1) p1,
       struct Point *@region('r2) p2) {
  p1->x += p2->x;
  p2->x += p2->y;
}
```

Now we can call `h` with pointers into any two regions, or even two pointers into the same region. This is because the code is type-correct *for all* regions `'r1` and `'r2`.

Occasionally, you will have to put region parameters in explicitly. This happens when you need to assert that two pointers point into the same region. Consider for instance the following function:

```
void j(struct Point * p1, struct Point * p2) {
  p1 = p2;
}
```

Cyclone will reject the code because it assumes that in general, `p1` and `p2` might point into *different* regions. The error is roughly the following:

```
foo.cyc:2: type mismatch:
     struct Point *‘GR0 != struct Point *‘GR1
  ‘GR1 and ‘GR0 are not compatible.
     (variable types are not the same)
```

Cyclone has picked the name `GR1` for `p1`'s region, and `GR2` for `p2`'s region. That is, Cyclone fills in the missing regions as follows:

```
void j(struct Point *@region(‘GR1) p1,
       struct Point *@region(‘GR2) p2) {
  p1 = p2;
}
```

Now it is clear that the assignment does not type-check because the types of `p1` and `p2` differ. In other words, `‘GR1` and `‘GR2` *might* be instantiated with different regions, in which case the code would be incorrect. For example, we could call `j` as follows:

```
void g() {
  struct Point p = {0,1};
  struct Point *@region(‘g) ptr1 = &p;
  struct Point *@region(‘H) ptr2 = new Point{2,3};
  j(ptr2,ptr1);
}
```

Doing this would effectively allow us to assign `ptr1` to `ptr2`, which is unsafe in general, since the heap outlives the stack region for `g`.

But you can require that `j`'s regions be instantiated with the same region by explicitly specifying the same explicit region variable for each pointer. Thus, the following code does type-check:

```
void j(struct Point *@region(‘r) p1,
       struct Point *@region(‘r) p2) {
  p1 = p2;
}
```

This would prevent the situation in function `g` above, since the arguments passed to `j` must be allocated in the same region.

So, Cyclone assumes that each pointer argument to a function is in a (potentially) different region unless you specify otherwise. The reason we chose this as the default is that (a) it is often the right choice for code, (b) it is the most general type in the sense that if it does work out, clients will have the most lattitude in passing arguments from different regions or the same region to the function.

What region variable is chosen for return-types that mention pointers? Here, there is no good answer because the region of the result of a function cannot be easily determined without looking at the body of the function, which defeats *separate compilation* of function definitions from their prototypes. Therefore, we have arbitrarily chosen the heap as the default region for function results. Consequently, the following code type-checks:

```
struct Point * good_newPoint(int x,int y) {
  return new Point{x,y};
}
```

This is because the `new` operator returns a pointer to the heap, and the default region for the return type is the heap.

This also explains why the `newPoint` function (page 18) for allocating a new `Point` does not type-check:

```
struct Point *newPoint(int x,int y) {
  struct Point result = {x,y};
  return &result;
}
```

The expression `&result` is a pointer into region `'newPoint` but the result type of the function must be a pointer into the heap (region `'H`).

If you want to return a pointer that is not in the heap region, then you need to put the region in explicitly. For instance, the following code:

```
int * id(int *x) {
  return x;
}
```

will not type-check. To see why, let us rewrite the code with the default region annotations filled in. The argument is assumed to be in a region

`'GR1`, and the result is assumed to be in the heap, so the fully elaborated code is:

```
int *@region(‘H) id(int *@region(‘GR1) x) {
  return x;
}
```

Now the type-error is manifest. To fix the code, we must put in explicit regions to connect the argument type with the result type. For instance, we might write:

```
int *@region(‘r) id(int *@region(‘r) x) {
  return x;
}
```

or using the abbreviation:

```
int *‘r id(int *‘r x) {
  return x;
}
```

Region Summary

In summary, each pointer in Cyclone points into a given region and this region is reflected in the type of the pointer. Cyclone will not let you dereference a pointer into a deallocated region. The lexical blocks declared in functions correspond to one type of region (a *stack region*), and simply declaring a variable within that block allocates storage within the region. The storage is deallocated upon exit of the block. LIFO regions are similar, except that a dynamic number of objects can be allocated within the region using the region's handle. Both stack and LIFO regions have structured lifetimes. Finally, the heap region `'H` is a special region whose dead objects are garbage collected.

Region polymorphism and region inference make it possible to omit many region annotations on types. By default, Cyclone assumes that unannotated pointer arguments in functions may live in distinct regions, and that unannotated result pointers are in the heap. These assumptions are not perfect, but programmers can fix the assumptions by providing explicit region annotations, and they permit Cyclone files to be separately compiled.

The region-based type system of Cyclone is perhaps the complicated aspect of the language. In large part, this is because memory management is a difficult and tricky business. We have attempted to make stack allocation and region polymorphic functions simple to use without sacrificing programmer control over the lifetimes of objects and without (always) having to resort to garbage collection. As more advanced features, we also provide even finer control over object lifetimes, but at the expense of more work from the programmer, using the *unique region* and *reference-counted* regions. In turn, these can be used to create *dynamic regions*, which support run-time allocation like LIFO regions, but have dynamic scope. For more information on these, and regions in general, see Section 8.

2.4 Arrays

Arrays in Cyclone are much as in C, and have a similar relationship to pointers, as we discussed earlier. However, there are more ways to create arrays in Cyclone, but with some restrictions on how they are initialized.

One can always declare an array and provide an initializer as in C. For instance:

```
int foo[8] = {1,2,3,4,5,6,7,8};
char s[4] = "bar";
```

are both examples from C for creating arrays. Note that Cyclone follows C's conventions here, so that if you declare arrays as above within a function, then the lifetime of the array coincides with the activation record of the enclosing scope. In other words, such arrays will be stack-allocated. Also note that by default, `char` arrays are *not* considered zero-terminated. To make them so, you must add the `@zeroterm` qualifier following the size of the array, as in

```
char s[4] @zeroterm = "bar";
```

In both cases, the size of the array must include the zero terminator.

To create heap-allocated arrays (or strings) within a Cyclone function, you should either use "`new`" or "`rnew`" operator with either an array initializer or an *array comprehension*. The following code demonstrates this:

```
// foo is a pointer to a heap-allocated array
int * @numelts(8) foo = new {1,2,3,4,5,6,7,8};
```

```
// s is a checked pointer to a heap-allocated string
char * @fat s = new {'b','a','r',0};

// a non-null pointer to the first 100 even numbers
int * @notnull @numelts(100) evens = new {for i < 100 : 2*i};
```

The last initializer is an array comprehension. The syntax is a simplified `for` loop: it declares the iterator variable `i` and its bound 100, indicating that `i` will iterate between 0 and 99. The part following the colon is the expression used to initialize the `i`th element of the array. In this example, the initializer is equivalent to writing

```
int * @notnull @numelts(100) evens = new {0,2,4,6,...,198};
```

Where the `...` represents the remaining even numbers in the sequence.

Finally, we note that it is not possible to create arrays that contain pointers without initializing them first. This is just as with normal pointers as discussed earlier. Moreover, Cyclone requires that pointerful arrays are initialized all at once, rather than one statement at a time. This is because Cyclone is not smart enough to know whether you have initialized an entire array, in general. For example, the following code would be rejected:

```
void f(int * p) {
  int *x[2];
  x[0] = p;
  x[1] = p;
}
```

Arrays that do not contain pointers need not be completely initialized in general to ensure safety, so how they are initilaized is up to the programmer.

2.5 Structs

Cyclone supports `struct` types just as in C. Quite often, a C `struct` declaration will be accepted without change by the Cyclone compiler. Cyclone supports two extensions to `struct` types in C: *tuples*, which are lightweight, "unnamed" structs, and *parameterization* for creating more

generic datastructures. We consider tuples below, and delay talking about parameterization until a bit later;

Tuples are like structs that need not be declared in advance; they have member or field names that are implicitly 0, 1, 2, 3, etc. For example, the following code declares `x` to be a 3-tuple of an integer, a character, and a boolean, initialized with the values 42, `'z'`, and `true` respectively. It then checks to see whether the third component in the tuple is `true` (it is) and if so, increments the first component in the tuple.

```
$(int,char,bool) x = $(42,'z',true)

if (x[2])
  x[0]++;
```

The above code would be roughly equivalent to writing:

```
struct {int f0; char f1; bool f2;} x = {42,'z',true};
if (x.f2)
  x.f1++;
```

Thus, tuple types are written `$(type1,...,typen)`, tuple constructor expressions are written `$(exp1,...,expn)`, and extracting the ith component of a tuple is written using subscript notation `exp[i-1]`. Note that, consistent with the rest of C, the members start with 0, not 1.

Unlike structs, tuple types are treated equivalent as long as they are structurally equivalent. As in C, struct types are equivalent only if they have the same tag or name. (Note that in C, all struct declarations have a tag, even if the compiler has to generate one.)

2.6 Unions

It's often necessary to write a function that accepts an argument with more than one possible type. For example, in

```
printf("%d",x);
```

`x` should be an integer, but in

```
printf("%s",x);
```

`x` should be a pointer to a sequence of characters.

If we call `printf("%s",x)` with an integer `x`, instead of a pointer `x`, the program will likely crash. To prevent this, most C compilers treat `printf` specially: they examine the first argument and require that the remaining arguments have the appropriate types. However, a compiler can't check this if `printf` isn't called with a literal format string:

```
printf(s,x);
```

where `s` is a string variable. This means that in C, programs that use `printf` (or `scanf`, or a number of related functions) are vulnerable to crashes and corrupted memory. In fact, it's possible for someone else to crash your program by causing it to call `printf` with arguments that don't match the format string. This is called a *format string attack*, and it's an increasingly common exploit.

Tagged Unions

Cyclone provides *tagged unions* so that you can safely write functions that accept an argument with more than one possible type. Like a C `union`, a Cyclone `@tagged union` is a type that has several possible cases. Here's a simple example:

```
@tagged union T {
  int Integer;
  const char *@fat String;
};
union T x = {.Integer = 3};
union T y = {.String = "hello, world"};
```

This declares a new tagged union type `T`, that can hold either an integer or a string (remember, a literal string can always be converted to a `char *@fat` in Cyclone). It also declares to `union T` values `x` and `y` and initializes them with an integer and string respectively.

Just as with C unions, you can read and write any member of a tagged union. However, to prevent security holes, Cyclone enforces the property that you can only read the last member written. This prevents you from accidentally treating an integer as if it's a string or some other kind of pointer.

Cyclone enforces this safety property by inserting a hidden tag into the union (hence the `@tagged` qualifier.) You can test the tag by using the built-in `tagcheck` function. For instance, here is a function that uses the real `printf` to safely print out the contents of a `union T` value, regardless of its contents:

```
bool printT(union T w) {
  if (tagcheck(w.Integer))
    printf("%d",w);
  else
    printf("%s",w);
}
```

Note that `tagcheck(w.Integer)` does not return the value of the `Integer` member, but rather returns true if and only if the `Integer` member was the last member written (and is thus safe to read.)

Each write to a tagged union member causes the hidden tag to be updated, and each read is preceded by a check to ensure that the member was the last one written. If you attempt to read some member other than the last one written, then the `Match` exception is thrown. For example, the following code writes the `String` member and then attempts to read the `Integer` member, so it will throw a `Match` exception:

```
union T a;
int x;
a.String = "hello, world";
/* Next line fails */
x = a.Integer + 3;
```

When you have a big union, it can be awkward to use `tagcheck` to test the hidden tag. You might accidentally test the wrong member or forget to cover a member. In these cases, its probably best to use *pattern matching* to determine the tag and to extract the underlying value. For example, here is the function `printT` coded with pattern matching:

```
void printT(union T a) {
  switch (a) {
  case {.Integer = i}: printf("%d",i); return;
  case {.String = s}: printf("%s",s); return;
  }
}
```

The argument `a` has type `union T`, so it is either an `Integer` or `String`. Cyclone extends switch statements with *patterns* that distinguish between the cases. The first case,

```
case {.Integer = i}: printf("%d",i); return;
```

contains a pattern, {`Integer = i`}, that will match only `T` values where the `Integer` member was the last one written. The variable `i` is bound to the underlying integer, and it can be used in the body of the case. For example, `printT(x)` will print 3, since `x` holds {`.Integer = 3`}, and `printT(y)` will print `hello, world`. You can find out more about patterns in Section 5.

Untagged Unions

Cyclone also supports untagged unions, but there are restrictions on how they may be used to ensure safety. In particular, you can write any value you like into a union, but you can only read out values that do not contain pointers. This ensures that you don't "spoof" a pointer with an integer or some other bogus value. So, the general rule is that you can use a normal C union if you aren't using pointers, but you must use a `@tagged` union if you are using pointers.

Datatypes

Cyclone provides another alternative to tagged unions for supporting heterogenous values called a *datatype*. Tagged unions require space proportional to the largest member (plus room for the tag.) In contrast, a datatype only requires space for the member being used. However, datatypes cannot be updated with a different member and require a level of indirection.

Here is our example type re-coded using a `datatype` declaration:

```
datatype T {
  Integer(int);
  String(const char *@fat);
};

datatype T.Integer x = Integer(3);
datatype T.String y = String("hello, world");
```

```
void printT(datatype T@ a) {
  switch (a) {
  case &Integer(i): printf("%d",i); return;
  case &String(s): printf("%s",s); return;
  }
}
```

In general, a `datatype` declaration includes a set of *constructors* which can be used to build datatype values. In this case, the constructors are `Integer` and `String`. The `Integer` constructor takes an `int` and returns a value of type `datatype T.Integer`. The `String` constructor takes a string and returns a `datatype T.String` value.

Note that the types of `x` and `y` are not the same so we can't interchange them, nor can we pass them directly to the `printT` function. In particular, their types reveal which constructor was used to build them. However, we can manipulate pointers to these values in an abstract way. In particular, we can pass a pointer to a `datatype T.Integer` value or a pointer to a `datatype T.String` value anywhere that expects a `datatype T`. For instance, we can write `printT(&x)` to print out the integer value in `x`, and we can write `printT(&y)` to print out the `"hello, world"` string in y.

We can use datatypes to implement a safe form of variable arguments (or *varargs*), as described in Section 11. More information on Cyclone unions is presented in Section 4.

2.7 C++, GCC and C99 Additions

C++, GCC and the ISO C99 standard have some useful new features that we have adopted for Cyclone. From C++ we borrow the idea of *namespaces* for avoiding conflicts among library and program definitions. In short, if a function `f` is defined in a namespace `Foo`, then we would access the function by referring to `Foo::f`. The Cyclone standard libraries, such as `Core` or `List` (covered in detail in Section C) are defined each in their own namespace. Cyclone also provides polymorphism similar to C++ templates.

Some of the GCC and C99 features that we currently support are:

- Statement expressions: There is a new expression form, `({` *statement expression* `})`. The statement is executed first, then the expression, and the value of the entire expression is the value of the expression
- Struct and Union expressions: If you've declared `struct point{int x; int y;};` then you can write `point{.x=`*`expression`*`, .y=`*`expression`*`}` to allocate and initialize a struct point. The same sort of constructors may be used for unions, tagged or not, as we showed above.
- `//` comments as in Java or C++
- Declarations can appear in any statement position. It is not necessary to wrap braces around the declaration of a local variable.
- For-statements can include a declaration. For instance:

```
for (int x=0; x < n; x++) {
  ...
}
```

We have attempted to follow the C99 standard fairly closely.

2.8 Additional Features of Cyclone

So far we have focused on features common to both Cyclone and GCC extensions of C. In the remainder of this tutorial, we overview some of the new language features that have been added to Cyclone, inspired from other programming languages. These include (a) exceptions (as in ML and Java), (b) type inference for local variables (as in ML), (c) parametric polymorphism (as in ML and Generic Java), (d) structural subtyping, to approximate object-oriented features, and (e) abstract and existential types.

In many cases, these features are useful for writing simpler code. Some features, like polymorphism and subtyping, are necessary to type-check or port a number of potentially-unsafe C idioms, usually involving "`void *`" or the like. We conclude the tutorial by enumerating some unsafe C idioms that are not supported in Cyclone.

2.9 Exceptions

So far we've glossed over what happens when you try to dereference a null pointer, or assign to an out-of-bounds `@fat` pointer. We've said that Cyclone inserts checks to make sure the operation is safe, but what if the checks fail? For safety, it would be sufficient to halt the program and print an error message—a big improvement over a core dump, or, worse, a program with corrupted data that keeps running.

In fact, Cyclone does something a bit more general than halting with an error message: it throws an *exception*. The advantage of exceptions is that they can be *caught* by the programmer, who can then take corrective action and perhaps continue with the program. If the exception is not caught, the program halts and prints an error message. Consider our earlier example:

```
FILE *f = fopen("/etc/passwd","r");
int c = getc((FILE *@notnull)f);
```

Suppose that there is no file `/etc/passwd`; then `fopen` will return NULL, and when `f` is cast to `FILE *@notnull`, the implied null check will fail. The program will halt with an error message,

```
Uncaught exception Null_Exception
```

`Null_Exception` is one of a handful of standard exceptions used in Cyclone. Each exception is like a case of a `datatype`: it can carry along some values with it. For example, the standard exception `Invalid_argument` carries a string (which is a zero-terminated fat pointer). Exceptions can be handled in `try-catch` statements, using pattern matching:

```
FILE *f = fopen("/etc/passwd","r");
int c;
try {
  c = getc((FILE *@notnull)f);
}
catch {
case &Null_Exception:
  printf("Error: can't open /etc/passwd\n");
  exit(1);
case &Invalid_argument(s):
  printf("Error: Invalid_argument(%s)\n",s);
```

```
  exit(1);
}
```

Here we've "wrapped" the call to `getc` in a `try-catch` statement. If `f` isn't `NULL` and the `getc` succeeds, then execution just continues, ignoring the `catch`. But if `f` is NULL, then the null check will fail and the exception `Null_Exception` will be thrown; execution immediately continues with the `catch` (the call to `getc` never happens). In the `catch`, the thrown exception is pattern-matched against the cases, in order. Since the thrown exception is `Null_Exception`, the first case is executed here.

There is one important difference between an exception and a case of a `datatype`: with a `datatype`, all of the cases have to be declared at once, while a new exception can be declared at any time. So, exceptions are an *extensible* `datatype`. You can specify that a datatype is extensible when you declare it, using the `@extensible` qualifier. For example, here's how to declare a new exception:

```
@extensible datatype exn {
  My_Exception(char *@fat);
};
```

The type `@extensible datatype exn` is the type of exceptions, and this declaration introduces a new case for the `@extensible datatype exn` type: `My_Exception`, which carries a single value (a string). Exception values are created just like `datatype` values, and are thrown with a `throw` statement. For example,

```
throw new My_Exception("some kind of error");
```

or

```
throw new Null_Exception;
```

In practice, "`@extensible datatype`" is quite a mouthful. So, Cyclone allows you abbreviate it with just `datatype`, as long as you've declared a datatype as `@extensible` once. So a more typical way to declare a new exception in Cyclone is

```
datatype exn {
  My_Exception(char ?);
};
```

2.10 Let Declarations and Pattern Matching

Sometimes, it's painful to declare a variable because you have to write down its type, and Cyclone types can be big when compared to their C counterparts since they may include bounds information, regions, *etc*. Therefore, Cyclone includes additional support for type inference using let declarations. In particular, you can write:

```
int foo(int x) {
  let y = x+3;
  let z = 3.14159;
  return (int)(y*z);
}
```

Here, we declared two variables `y` and `z` using "`let`." When you use `let`, you don't have to write down the type of the variable. Rather, the compiler infers the type from the expression that initializes the variable.

More generally, you can write "`let pattern = exp;`" to destructure a value into a bunch of variables. For instance, if you pass a tuple to a function, then you can extract the components as follows:

```
int sum($(int,int,int) args) {
  let $(x,y,z) = args;
  return (x+y+z);
}
```

This feature is called pattern matching, and is inspired from functional languages like ML and Haskell. Patterns can appear as part of `let` declarations, exception clauses (as we saw above), and `switch` statements. For example, we could rewrite the above code as follows using `switch`:

```
int sum($(int,int,int) args) {
  switch (args) {
  case $(x,y,z):
    return (x+y+z);
  }
}
```

Notice there is no need for a `default` case, since `args` will always be a valid 3-tuple. On the other hand, if we were to pass a pointer to the tuple, rather than a tuple itself, we would have code as follows:

```
int sum($(int,int,int) *argsp) {
  switch (argsp) {
  case &$(x,y,z):
    return (x+y+z);
  default:
    return 0;
  }
}
```

The `switch` statement first handles the situation that the argument is a pointer to a tuple, designated by putting an `&` in front of the tuple pattern. Recall that we did something similar when matching exceptions in `catch` clauses, above. The default case is for when `argsp` is NULL. Many more details about pattern matching are presented in Section 5.

2.11 Subtyping

Cyclone supports "extension on the right" and "covariant depth on `const`" subtyping for pointers. This simply means that you can cast a value `x` from having a type "pointer to a struct with 10 fields," to "pointer to a struct having only the first 5 fields." For example, if we have the following definitions:

```
typedef struct Point {float x,y;} *point;

typedef struct CPoint {float x,y; int color;} *cpoint;

float xcoord(point p) {
  return p->x;
}
```

then you can call `xcoord` with either a `point` or `cpoint` object. You can also cast a pointer to a tuple having 3 fields (e.g., `$(int,bool,double)*`) to a pointer to a tuple having only 2 fields (e.g., `$(int,bool)*`). In other words, you can forget about the "tail" of the object. This allows a degree of polymorphism that is useful when porting C code. In addition, you can do "deep" casts on pointer fields that are `const`. (It is unsafe to allow deep casts on non-const fields.) Also, you can cast a field from being non-const to being const. You can also cast a constant-sized array to an equivalent

pointer to a struct or tuple. In short, Cyclone attempts to allow you to cast one type to another as long as it is safe. Note, however, that these casts must be explicit.

We expect to add more support for subtyping in the future (e.g., subtyping on function pointers, bounded subtyping, etc.)

2.12 Polymorphic Functions

Cyclone supports a fairly powerful form of *parametric polymorphism*. Those of you coming from ML or Haskell will find this familiar. Those of you coming from C++ will also find it somewhat familiar. The basic idea is that you can write one function that abstracts the types of some of the values it manipulates. For instance, consider the following two functions:

```
$(char*,int) swap1($(int,char*) x) {
   return $(x[1], x[0]);
}
$(int,int) swap2($(int,int) x) {
   return $(x[1], x[0]);
}
```

The two functions are quite similar: They both take in a pair (i.e., a 2-tuple) and return a pair with the components swapped. At the machine-level, the code for these two functions will be exactly the same, assuming that `int`s and `char *`s are represented the same way. So it seems silly to write the code twice. Normally, a C programmer would replace the definition with simply:

```
$(void *,void *) swap1($(void *,void *) x) {
   return $(x[1], x[0]);
}
```

(assuming you added tuples to C). But of course, this isn't type-safe because once I cast the values to `void *`, then I can't be sure what type I'm getting out. In Cyclone, you can instead write something like this:

```
$(‘b,‘a) swap($(‘a,‘b) x) {
   return $(x[1],x[0]);
}
```

The code is the same, but it abstracts what the types are. The types `'a` and `'b` are type variables that can be instantiated with any word-sized, general-purpose register type. So, for instance, you can call swap on pairs of integers, pairs of pointers, pairs of an integer and a pointer, etc.:

```
let $(x,y) = swap($("hello",3));  // x is 3, y is hello
let $(w,z) = swap($(4,3));        // w is 3, z is 4
```

Note that when calling a polymorphic function, you need not tell it what types you're using to instantiate the type variables. Rather, Cyclone figures this out through unification.

C++ supports similar functionality with templates. However, C++ and Cyclone differ considerably in their implementation strategies. First, Cyclone only produces one copy of the code, whereas a C++ template is specialized and duplicated at each type that it is used. This approach requires that you include definitions of templates in interfaces and thus defeats separate compilation. However, the approach used by Cyclone does have its drawbacks: in particular, the only types that can instantiate type variables are those that can be treated uniformly. This ensures that we can use the same code for different types. The general rule is that values of the types that instantiate a type variable must fit into a machine word and must be passed in general-purpose (as opposed to floating-point) registers. Examples of such types include `int`, pointers, `datatype`, and `xdatatype` types. Other types, including `char`, `short`, `long long`, `float`, `double`, `struct`, and `tuple` types violate this rule and thus values of these types cannot be passed to a function like swap in place of the type variables. In practice, this means that you tend to manipulate a lot of pointers in Cyclone code.

The combination of parametric polymorphism and sub-typing means that you can cover a lot of C idioms where `void*` or unsafe casts were used without sacrificing type-safety. We use polymorphism a lot when coding in Cyclone. For instance, the standard library includes many container abstractions (lists, sets, queues, etc.) that are all polymorphic in the element type. This allows us to re-use a lot of code. In addition, unlike C++, those libraries can be compiled once and need not be specialized. On the downside, this style of polymorphism does not allow you to do any type-specific things (e.g., overloading or ad-hoc polymorphism.) Someday, we may add support for this, but in the short run, we're happy not to have it.

2.13 Polymorphic Data Structures

Just as function definitions can be parameterized by types, so can `struct` definitions, `datatype` definitions, and even `typedef`s. For instance, the following `struct` definition is similar to the one used in the standard library for lists:

```
struct List<'a> {'a hd; struct List<'a> * tl; };
typedef struct List<'a> *list_t<'a>;
```

Here, we've declared a `struct List` parameterized by a type `'a`. The `hd` field contains an element of type `'a` and the `tl` field contains a possibly-null pointer to a `struct List` with elements of type `'a`. We then define `list_t<'a>` as an abbreviation for `struct List<'a>*`. So, for instance, we can declare both integer and string lists like this:

```
list_t<int> ilist = new List{1,new List{2,null}};
list_t<string_t> slist = new List{.hd = "foo",
                                  .tl = new List{"bar",null}};
```

Note that we use "`new`" as in C++ to allocate a new `struct List` on the heap and return a pointer to the resulting (initialized) `List` object. Note also that the field designator ("`.hd`", "`.tl`") are optional.

Once you have polymorphic data structures, you can write lots of useful polymorphic code and use it over and over again. For instance, the standard list library (see lib/list.h) includes functions for mapping over a list, looking up items in a list, concatenating two lists, copying lists, sorting lists, etc.

2.14 Abstract and Existential Types

Suppose you want to declare an abstract type for implementing stacks. In Cyclone, the way this is accomplished is by declaring a struct that encapsulates the implementation type, and by adding the "`abstract`" qualifier to the struct definition. For instance, if we write:

```
abstract struct Queue<'a> { list_t<'a> front, rear; };
```

then this declares a polymorphic `Queue` implementation that is abstract. The definition of the struct is available within the unit that declares the

Queue, but will not be made available to the outside world. (This will be enforced by a link-time type-checker that we are currently putting together.) Typically, the provider of the `Queue` abstraction would write in an interface file:

```
extern struct Queue<'a>;
```

The abstract keyword in the implementation ensures that the definition does not leak to a client.

A `typedef` can be made abstract by writing:

```
typedef _ foo_t;
```

However, our current implementation does not support later redefining `foo_t` as a non-abstract typedef. The default kind for the type is B; you can write an explicit kind like this:

```
typedef _::A bar_t;
```

Generally abstract structs are sufficient. An abstract typedef is useful in some cases, though, such as when the abstracted type is actually `int`.

It's also possible to code up "first-class" abstract data types using `structs` with *existentially quantified type variables*. Existential types are described in Section 13.

For an example of the use of existential types, see the `fn.h` and `fn.cyc` files in the standard library, which implement first-class closures.

2.15 Restrictions

Though Cyclone adds many new features to C, there are also a number of restrictions that it must enforce to ensure code does not crash. Here is a list of the major restrictions:

- Cyclone enforces the evaluation order of subexpressions in contexts in which C is agnostic. For example, consider the following expression

  ```
  f(x=0,x=1);
  ```

In C, the resulting value of `x` could either be 1 or 0, depending on the compiler; the language makes no restriction on the order that argument expressions should be evaluated. This allows the compiler more freedom in generating optimized code. However, it often makes programs harder to reason about, and they can break in subtle ways when ported to different platforms. It also can foil forms of static analysis aimed at improving performance, such array-bounds- or null-check elimination.

For Cyclone, we implement the following policy:

- Cyclone follows C for all those places it has a requirement, e.g., for comma-separated expression lists.
- Cyclone enforces the evaluation order to be right-to-left for assignment expressions. For example:

  ```
  e1 = e2 = e3;
  ```

  In this expression, Cyclone evaluates first `e3`, then `e2`, then `e1`.
- In all contexts, Cyclone's evaluation order is left-to-right, e.g., for arguments passed to a function, and subexpressions in an arithmetic expression.

- Cyclone does not permit some of the casts that are allowed in C because incorrect casts can lead to crashes, and it is not always possible for us to determine what is safe. In general, you should be able to cast something from one type to another as long as the underlying representations are compatible. Note that Cyclone is very conservative about "compatible" because it does not know the size or alignment constraints of your C compiler.

- Cyclone does not support pointer arithmetic on thin pointers unless they are zero-terminated and even then, there are checks to make sure you can't go past a zero. Pointer arithmetic is not unsafe in itself, but it can lead to unsafe code when the resulting pointer is assigned or dereferenced. You can cast the thin pointer value to a `@fat` value and then do the pointer arithmetic instead.

- Cyclone inserts a NULL check when a possibly-NULL pointer is dereferenced and it cannot determine statically that the pointer is not NULL.

- If a function's return type is not "bits-only" (i.e., contains pointers), Cyclone requires that the function executes a return statement, throws an exception, or calls a `noreturn` function on every possible execution path. This is needed to ensure that the value returned from the function has the right type, and is not just a random value left in a register or on the stack.
- Untagged unions in Cyclone can hold arbitrary values, but you can only read out "bits." In particular, the members you can read out can only have combinations of chars, ints, shorts, longs, floats, doubles, structs of bits, or tuples of bits. Pointer types are not supported. This avoids the situation where an arbitrary bit pattern is cast to a pointer and then dereferenced. If you want to use multiple types, then use `@tagged union`s or `datatype`s.
- Cyclone only supports limited forms of `malloc` (and `calloc`). You must write `malloc(sizeof(t)*n)` and `t` must be a "bits-only" type. You can use `calloc` to allocate arrays of (possibly NULL) pointers (e.g., `calloc(sizeof(char*),34)`).
- Cyclone performs a static analysis to ensure that every non-numeric (i.e., pointer) variable and `struct` field is initialized before it is used. This prevents a random stack value from being used improperly. The analysis is somewhat conservative so you may need to initialize things earlier than you would do in C. There is only limited support for initializing memory in a procedure separate from the one that does the allocation.
- Cyclone does not permit `goto`s from one scope into another. C warns against this practice, as it can cause crashes; Cyclone rules it out entirely.
- Cyclone places some limitations on the form of switch statements that rule out crashes like those caused by unrestricted `goto`. Furthermore, Cyclone prevents you from accidentally falling through from one case to another. To fall through, you must explicitly use the `fallthru` keyword. Otherwise, you must explicitly `break`, `goto`, `continue`, `return`, or `throw` an exception. However, adjacent cases for a switch statement (with no intervening statement) do not require an explicit `fallthru`.

- Cyclone has some new keywords (beyond those of C99 and GCC) that can no longer be used as identifiers. The list includes: `abstract`, `calloc`, `datatype`, `dynregion_t`, `export`, `fallthru`, `__gen`, `let`, `malloc`, `namespace`, `numelts`, `__cyclone_port_on__`, `__cyclone_port_off__`, `region`, `regions`, `reset_region`, `rmalloc`, `rnew`, `tagcheck`, `tag_t`, `throw`, `try`, `using`, `valueof`, and `valueof_t`.

- Cyclone prevents you from using pointers to stack-allocated objects as freely as in C to avoid security holes. The reason is that each declaration block is placed in a conceptual "region" and the type system tracks the region into which a pointer points.

- Cyclone does not allow you to explicitly free a heap-allocated object. Instead, you can either use the region mechanism or rely upon the conservative garbage collector to reclaim the space.

In addition, there are a number of shortcomings of the current implementation that we hope to correct in the near future. For instance:

- Cyclone currently does not support nested type declarations within a function. All `struct`, `union`, `enum`, `datatype`, and `typedef` definitions must be at the top-level.

- Cyclone does not allow a typedef declaration to be shadowed by another declaration.

3 Pointers

As in C, one should think of Cyclone pointers as just addresses. Operations on pointers, such as `*x`, `x->f`, and `x[e]`, behave the same as in C, with the exception that run-time checks sometimes precede memory accesses. (Exactly when and where these checks occur is described below.) However, Cyclone prevents memory errors such as dereferencing dangling pointers or indexing outside an array's bounds, so it may reject some operations on pointers that C would accept.

In order to enforce memory safety and distinguish between different uses of pointers, Cyclone pointer types include additional qualifiers when

compared to their C counterparts. These qualifiers are described briefly below and in more detail throughout this section:

- `@nullable`: Pointers with this qualifier may be `NULL`. This qualifier is present by default and overridden by the `@notnull` qualifier. A dereference of a `@nullable` pointer will generally be preceded by a `NULL`-check.

- `@notnull`: Pointers with this qualifier may never be `NULL`, and thus never need to be checked for `NULL` upon dereference. This qualifier is not present by default and must be put in explicitly. The qualifier may be abbreviated by using "`@`" in place of the usual pointer "`*`". So, for instance, the type "`int *@notnull`" can be abbreviated by "`int @`". Currently, the `@notnull` qualifier cannot be used on pointers with the `@fat` qualifier.

- `@thin`: Pointers with this qualifier have the same representation as in C (i.e., a single machine word.) However, arithmetic on thin pointers is not supported except when the pointer is also qualified as `@zeroterm` (see below). This qualifier is present by default and overridden by the `@fat` qualifier.

- `@fat`: Pointers with this qualifier consist of a thin pointer plus additional information needed to support safe pointer arithmetic and dereferencing. (The current implementation uses three words in total.) Each dereference of a fat pointer incurs both a `NULL`-check and a bounds check to ensure that the pointer points to a valid object. The `@fat` qualifier cannot be used with the `@notnull` or `@numelts` qualifiers (though we expect to change this in the future.) The `numelts` operation may be applied to fat pointers to determine the number of elements in the (forward) sequence that may be safely dereferenced. Finally, the qualifier may be abbreviated by using "`?`" in place of the usual pointer "`*`". So, for instance, the type "`int *@fat`" can be abbreviated by "`int ?`".

- `@numelts(e)`: The term `e` must be a *static* expression (i.e., a constant expression or one involving `valueof`) and indicates an upper bound on the number of objects that that the pointer refers to. For example, if `p` has type `T *@numelts(42)`, then either `p` is `NULL`

or else for $0 \leq$ `i` $<$ `e`, the expression `p[i]` is guaranteed to contain a `T` object. This qualifier may not be used in conjunction with `@fat`. If omitted on a `@thin` pointer, then `@numelts(1)` is inserted by default. This qualifier can be abbreviated by writing the bounds expression `e` in curly braces. For instance, the type "`int *@numelts(42)`" can be abbreviated by "`int *{42}`".

- `@zeroterm`: This qualifier is used for zero-terminated sequences, such as C-style strings, and provides an alternative to fat pointers for doing safe pointer arithmetic without knowing bounds statically. This qualifier can only be used on pointers whose element type admits zero or `NULL` as a value, including integral types, and `@nullable` pointer types. Arithmetic in the forward direction is possible with zero-terminated pointers (e.g., `p++`) as is a subscript with a positive index (e.g., `p[i]`). However, the compiler inserts code to ensure that the index does not step over the final zero. When updating a zero-terminated array, the compiler also ensures that the final zero is not overwritten with a non-zero value. It is generally best to coerce a thin, zero-terminated pointer to a fat, zero-terminated pointer to avoid these overheads. This qualifier is only present by default for `char` pointers. It can be overridden with the `@nozeroterm` qualifier. This qualifier may also be used on array types.

- `@nozeroterm`: This qualifier is present by default on all pointer types except for `char` pointers. It is used to override the implicit `@zeroterm` qualfier for `char` pointers. This qualifier may also be used on array types.

- `@region('r)`: This qualifier is used to indicate the region into which a pointer points (in this case region `'r`). The qualifier may be abbreviated by simply writing the region name after any other Cyclone qualifiers. For instance, the type "`int *@notnull @region'r`" may be abbreviated as "`int @'r`". The rules about default region annotations are context-dependent and therefore described below.

- `@aqual(...)`: This qualifier is used to specify the aliasability of the pointer. Valid arguments to the @aqual(...) qualifier are `ALIASABLE, UNIQUE, REFCNT, RESTRICTED`, each of which impose a different aliasing discipline on the pointer. A type variable of kind `Q` may

also appear as an argument. See section 9 for more details.

3.1 Pointer Subtyping

Some pointer types may be safely used in contexts where another pointer type is expected. In particular, `T*@notnull` is a subtype of `T*@nullable` which means that a not-null pointer can be passed anywhere a possibly-null pointer is expected.

Similarly, a `T*@numelts(42)` pointer can be passed anywhere a `T*@numelts(30)` pointer is expected, because the former describes sequences that have at least 42 elements, which satisifes the constraint that it has at least 30 elements.

In addition, `T*@region(‘r)` is a subtype of `T*@region(‘s)` when region `‘r` *outlives* region `‘s`. The heap region (`‘H`) outlives every region so you can safely use a heap pointer anywhere another region is expected. Outer blocks and outer regions outlive inner blocks and regions. For example the following code is type-correct:

```
void foo(int x) {
  int *@region(‘foo) y = &x;
  L:{
    int *@region(‘L) z = y;
  }
}
```

because region `‘foo` outlives region `‘L`. By default, regions passed in to a function outlive any regions defined in the function (because they will live across the function call). Finally, you can specify outlives relations among region parameters within a function's prototype. The following code specifies that input region `‘r` outlives input region `‘s` so it's safe to treat `‘r` pointers as if they were `‘s` pointers:

```
void bar(int *@region(‘r) x,
         int *@region(‘s) y : {‘s} > ‘r);
```

In general, the outlives relation is specified after the function arguments by separating the relations with a colon (`:`) and giving a comma-separated list of primitive outlives relations. These outlives relations are of the form "`{‘r1,...,‘rn} > ‘r`" and specify that region `‘r` outlives all of the regions `‘r1` through `‘rn`.

Finally, when `T` is a subtype of `S`, then `T*` is a subtype of `const S*`. So, for instance, if we declare:

```
// nullable int pointers
typedef int * nintptr_t;
// not-nullable int pointers
typedef int *@notnull intptr_t;
```

then `intptr_t *` is a subtype of `const nintptr_t *`. Note, however, that "`const`" is important to get this kind of deep subtyping.

The following example shows what could go wrong if we allowed deep subtyping without the `const`:

```
void f(int *@notnull *@notnull x) {
  int *@nullable *@notnull y = x;
  // would be legal if int *@nullable *@notnull
  // was a subtype of int *@notnull *@notnull.
  *y = NULL;
  // legal because *y has type int *@nullable
  **x;
  // seg faults even though the type of *x is
  // int *@notnull
}
```

3.2 Pointer Coercions

In addition to pointer subtyping, Cyclone provides a number of *coercions* which allow you to convert a pointer value from one type to another. For instance, you can coerce a thin pointer with 42 elements to a fat pointer:

```
int arr[42];
int *@thin @numelts(42) p = arr;
int *@fat pfat = p;
```

As another example, you can coerce a thin, zero-terminated pointer to a fat, zero-terminated pointer:

```
int strlen(char *@zeroterm s) {
  char *@fat @zeroterm sfat = s;
  return numelts(s);
}
```

In both cases, the compiler inserts code to convert from the thin representation to an appropriate fat representation. In the former case, the bounds information can be calculated statically. In the latter case, the bounds information is calculated dynamically (by looking for the zero that terminates the sequence.) In both cases, the coercion is guaranteed to succeed, so the compiler does not emit a warning.

In other cases, a coercion can cause a run-time exception to be thrown. For instance, if you attempt to coerce a `@nullable` pointer to a `@notnull` pointer, and the value happens to be `NULL`, then the exception `Null_Exception` is thrown. In general, the compiler will warn you when you try to coerce from one pointer representation to another where a run-time check must be inserted, and that check might fail. A dataflow analysis is used to avoid some warnings, but in general, it's not smart enough to get rid of all of them. In these cases, you can explicitly cast the pointer from one representation to the other, and the compiler will not generate a warning (though it will still insert the run-time check to ensure safety.)

Here is a list of some of the coercions that are possible:

- `T` can be coerced to `S` when `T` is a subtype of `S`.
- `T*@nullable` can be coerced to `T*@notnull` but might throw an exception at run-time.
- `T*@thin@numelts(c)` can be coerced to `T*@fat` when `c` is a constant expression.
- `T*@fat` can be coerced to `T*@thin @numelts(c)` when `c` is a constant expression, but might throw an exception at run-time.
- `T*@thin@zeroterm` can be coerced to `T*@fat@zeroterm` and vice versa.
- `T*@thin@zeroterm` can be coerced to `const T*@fat@nozeroterm`.
- `T*@thin@zeroterm` can be coerced to `T*@fat@nozeroterm`, but access to the trailing zero is lost.

3.3 Default Region Qualifiers

The rules the compiler uses for filling in `@region` qualifiers when they are omitted from pointer types are a little complicated, but they are designed to avoid clutter in the common case:

- In function-argument types, a fresh (polymorphic) region name is used.
- In function-return types, `'H` is used.
- In type definitions, including `typedef`, `'H` is used.
- In function bodies, unification is used to infer the region based on how the location assigned the pointer type is used in the function.

Thus, be warned that

```
typedef int * foo_t;
void g(foo_t);
```

is different than

```
void g(int *);
```

The reason is clear when we fill in the default region qualifiers. In the first case, we have:

```
typedef int *@region('H) foo_t;
void g(foo_t);
```

whereas in the second case we have:

```
void g(int *@region('r));
```

3.4 Static Expression Bounds

The bound for the `@numelts` qualifier must be a *static* expression. A static expression is either a constant expression, or an expression involving `valueof(T)` for a type-level expression `T`. The `valueof` construct is used to connect the value of a run-time integer to the static bound on an array. For example, the following function takes in an integer `num` and pointer to a sequence of `num` integers and returns the sum of the sequence:

```
int sum(tag_t<'n> num,
        int *@notnull @numelts(valueof('n)) p) {
  int a = 0;
  for (unsigned i = 0; i < num; i++)
    a += p[i];
}
```

The type of `num` is specified as `tag_t<'n>`. This simply means that `num` holds an integer value, called `'n`, and the number of elements of `p` is equal to `n`. This form of dependency is common enough that it can be abbreviated as follows:

```
int sum(tag_t num, int p[num]);
```

and the compiler will fill in the missing information.

4 Tagged Unions and Datatypes

In addition to `struct`, `enum`, and `union`, Cyclone provides `@tagged union` and `datatype` declarations as ways to construct new aggregate types. Like a `union` type, each `@tagged union` and `datatype` has a number of *variants* (or members). Unlike conventional `union`s, an object of a `@tagged union` or `datatype` type is exactly one variant, we can detect (or discriminate) that variant at run-time, and the language prevents using an object as though it had a different variant.

The difference between `@tagged union`s and `datatype`s is that the former look and behave much like traditional unions, whereas the latter look and behave more like the algebraic datatypes found in functional languages such as ML. Furthermore, datatypes can be either closed or `@extensible`. A closed datatype's members are specified all together when the datatype is declared, whereas an `@extensible` datatype supports adding new members after the fact (much like adding a new subclass to a class-based OO language.)

In this section, we first discuss `@tagged` unions, then closed datatypes, and finally `@extensible` datatypes.

4.1 Tagged Unions

A tagged union declaration looks just like a C union, except that it you must specify the `@tagged` qualifier when declaring it. For example:

```
@tagged union Foo {
   int i;
   double d;
   char *@fat s;
};
```

The primary difference with C unions is that a tagged union includes a hidden tag. The tag indicates which member was last written. So, for example:

```
union Foo x;
x.i = 3;
x.s = "hello";
```

causes the hidden tag to first indicate that the `i` member was written, and then is updated to record that the `s` member was written.

When you attempt to read a member of a tagged union, a check is done on the hidden tag to ensure that this was the last member written, and thus the union contains a valid object of that member's type. If some other member was last updated, then a `Match_Exception` will be thrown.

You can test the hidden tag of a tagged union by using the `tagcheck` operation. For example:

```
void printFoo(union Foo x) {
  if (tagcheck(x.i))
    printf("%d",x.i);
  else if (tagcheck(x.d))
    printf("%g",x.d);
  else if (tagcheck(x.s))
    printf("%s",x.s);
}
```

Alternatively, you can use pattern matching (described in the next section) which will ensure that you cover all of the cases properly. For instance, the function above may be rewritten as:

```
void printFoo(union Foo x) {
  switch (x) {
  case {.i = i}: printf("%d",i); break;
  case {.d = d}: printf("%g",d); break;
  case {.s = s}: printf("%s",s); break;
  }
}
```

If we failed to leave out one of the cases in the pattern match, then the compiler would warn us. This is particularly helpful when you add new variants to a tagged union, for then the compiler pinpoints the spots that you need to update in your code. Therefore, we encourage the use of pattern matching where possible.

4.2 Datatypes

At its simplest, a `datatype` looks just like an `enum` declaration. For example, we could say:

```
datatype Color { Red, Green, Blue };
```

As with `enum`, the declaration creates a type (called `datatype Color`) and three constants `Red`, `Green`, and `Blue`. Unlike `enum`, these constants do not have type `datatype Color`. Instead, each variant has its *own type*, namely `datatype Color.Red`, `datatype Color.Green`, and `datatype Color.Blue`. However, a pointer to one of these values can be treated as a sub-type of a pointer to a `datatype Color`. So you can write:

```
datatype Color.Red red = Red;
datatype Color *c = &red;
```

In this simple example, we are splitting hairs, but we will soon find all these distinctions useful.

Unlike `enum`, `datatype` variants may carry any fixed number of values, as in this example:

```
datatype Shape {
  Point,
  Circle(float),
```

```
  Ellipse(float,float),
  Polygon(int,float),
};
```

A `Point` has no accompanying information, a `Circle` has a radius, an `Ellipse` has two axis lengths, and a (regular) `Polygon` has a number of sides and a radius. (The value fields do not have names, so it is often better style to have a variant carry one value of a struct type, which of course has named members.) This example creates five types: `datatype Shape`, `datatype Shape.Point`, `datatype Shape.Circle`, `datatype Shape.Ellipse`, and `datatype Shape.Polygon`. Like in our previous example, `datatype Shape.Point*` is a subtype of `datatype Shape*` and `Point` is a constant of type `datatype Shape.Point`.

Variants that carry one or more values are treated differently. `Circle` becomes a *constructor*; given a float it produces an object of type `datatype Shape.Circle`, for example `Circle(3.0)`. Similarly, `Ellipse(0,0)` has type `datatype Shape.Ellipse` (thanks to implicit casts from `int` to `float` for 0) and `Polygon(7,4.0)` has type `datatype Shape.Polygon`. The arguments to a constructor can be arbitrary expressions of the correct type, for example, `Ellipse(rand(), sqrt(rand()))`.

Here are some examples which allocate a `Point` and `Circle` respectively, but then use subtyping to treat the resulting values as if they are `Shape` pointers:

```
datatype Shape *s1 = new Point;
datatype Shape *s2 = new Circle(3.0);
```

Datatypes are particularly useful for building recursive structures. For example, a small language of arithmetic expressions might look like this:

```
enum Unops { Negate, Invert};
enum Binops { Add, Subtract, Multiply, Divide };
typedef datatype Exp *@notnull exp_t;
datatype Exp {
  Int(int),
  Float(float),
  Unop(enum Unops, exp_t),
  Binop(enum Binops, exp_t, exp_t)
};
```

A function returning an expression representing the multiplication of its parameter by two can be written like this:

```
exp_t double_exp(datatype Exp @e) {
  return new Binop(Multiply, e, new Int(2));
}
```

Accessing Datatype Variants Given a value of a `datatype` type, such as `datatype Shape`, we do not know which variant it is. It could be a `Circle` or `Shape`, etc. In Cyclone, we use *pattern matching* to determine which variant a given datatype value actually is, and to extract the arguments that were used to build the datatype value. For example, here is how you could define `isCircle`:

```
bool isCircle(datatype Shape *s) {
  switch(s) {
  case &Circle(r): return true;
  default: return false;
  }
}
```

When a `switch` statement's argument is a pointer to a datatype, the cases describe variants. One variant of `datatype Shape *` is a pointer to a `Circle`, which carries one value. The corresponding pattern has `&` for the pointer, `Circle` for the constructor name, and one identifier for each value carried by `Circle`. The identifiers are binding occurrences (declarations, if you will), and the initial values are the values of the fields of the `Circle` at which `s` points. The scope is the extent of the case clause.

Here is another example:

[The reader is asked to indulge compiler-writers who have forgotten basic geometry.]

```
extern area_of_ellipse(float,float);
extern area_of_poly(int,float);
float area(datatype Shape *s) {
  float ans;
  switch(s) {
  case &Point:
    ans = 0;
```

```
    break;
  case &Circle(r):
    ans = 3.14*r*r;
    break;
  case &Ellipse(r1,r2):
    ans = area_of_ellipse(r1,r2);
    break;
  case &Polygon(sides,r):
    ans = area_of_poly(sides,r);
    break;
  }
  return ans;
}
```

The cases are compared in order against `s`. The following are compile-time errors:

- It is possible that a member of the `datatype` type matches none of the cases. Note that `default` matches everything.
- A case is useless because it could only match if one of the earlier cases match. For example, a default case at the end of the `switch` in area would be an error.

As you can discover in Section 5, Cyclone has much richer pattern-matching support than we have used here.

Polymorphism and Datatypes A `datatype` declaration may be polymorphic over types and regions just like a `struct` or `union` definition (see the section on polymorphism). For example, here is a declaration for binary trees where the leaves can hold some `BoxKind 'a`:

```
datatype Tree<'a>  {
  Leaf('a);
  Node(datatype Tree<'a>*, datatype Tree<'a>*);
};
```

In the above example, the root may be in any region, but all children will be in the heap. The following version allows the children to be in any region, but they must all be in the same region. (The root can still be in a different region.)

```
datatype Tree<'a,'r>  {
  Leaf('a);
  Node(datatype Tree<'a,'r> *'r,
       datatype Tree<'a,'r> *'r);
};
```

4.3 The Tagof Operator

Each datatype constructor has an associated unique integer called the *tag*. The tag of a datatype or datatype field can be accessed using the `tagof` operator. For example, given the datatype `Exp` defined previously, we can write a structural comparison function which enforces an order on expressions as follows:

```
bool sameexp(datatype Exp @e1, datatype Exp @e2) {
  switch $(e1,e2) {
  case $(&Int(i),&Int(j)): return i - j;
  case $(&Float(f),&Float(g)):
      if (f == g) return 0;
      if (f < g) return -1;
      return 1;
  case $(&Unop(u1,ea),&Unop(u2,eb)):
      int c = u1 - u2;
      if (c != 0) return c;
      return sameexp(ea,eb);
  case $(&Binop(b1,ea1,ea2),&Binop(b2,eb1,eb2)):
      int c = b1 - b2;
      if (c != 0) return c;
      c = sameexp(ea1,eb1);
      if (c != 0) return c;
      return sameexp(ea2,eb2);
  default: return tagof(e1) - tagof(e2);
  }
}
```

This function compares the structure of two expressions `e1` and `e2` and returns a number less than zero if `e1` $<$ `e2`, zero if `e1` $==$ `e2`, and a number greater than zero otherwise. Notice that the default case needs to

handle all of the situtations where `e1` and `e2` are different constructors. To achieve this, we extract the tags from the two datatype values and return their difference. This keeps us from having to write a more tedious match that would include the following cases:

```
case $(&Int(_),_): return -1;
case $(_,&Int(_)): return 1;
case $(&Unop(_,_),_): return -1;
case $(_,&Unop(_)) return 1;
```

Future

- Currently, given a value of a variant type (e.g., `datatype Shape.Circle`), the only way to access the fields is with pattern-matching even though the variant is known. We may provide a tuple-like syntax in the future.
- We hope to provide symbolic definitions for the values corresponding to the tags of a datatype. For now, the current association is that the tags are assigned consecutively based on the order of the constructor declarations.

4.4 Extensible Datatypes

We now explain how an `@extensible datatype` type differs from a `datatype`. The main difference is that later declarations may continue to add variants. Extensible datatypes are useful for allowing clients to extend data structures in unforeseen ways. For example:

```
@extensible datatype Food;
datatype Food { Banana; Grape;
                Pizza(list_t<datatype Food*>) };
datatype Food { Candy; Broccoli };
```

After these declarations, `Pizza(new List(new Broccoli, NULL))` is a well-typed expression.

If multiple declarations include the same variants, the variants must have the same declaration (the number of values, types for the values, and the same existential type variables).

Because different files may add different variants and Cyclone compiles files separately, no code can know (for sure) all the variants of an `@extensible datatype`. Hence all pattern-matches against a value of an `@extensible datatype` type must end with a case that matches everything, typically `default`.

There is one built-in `@extensible datatype` type: `@extensible datatype exn` is the type of exceptions. Therefore, you declare new `exn` constructors like this:

```
datatype exn {BadFilename(string)};
```

The implementation of `@extensible datatype` types is very similar to that of `datatype` types, but variant tags cannot be represented as small integers because of separate compilation. Instead, these tags are represented as pointers to unique locations in static data.

5 Pattern Matching

Pattern matching provides a concise, convenient way to bind parts of large objects to new local variables. Two Cyclone constructs use pattern matching, `let` declarations and `switch` statements. Although the latter are more common, we first explain patterns with let declarations because they have fewer complications. Then we describe all the pattern forms. Then we describe switch statements.

You must use patterns to access values carried by tagged unions, including exceptions. In other situations, patterns make code more readable and less verbose.

Note that this section does not include rules for matching against *unique pointers*; this is explained in Section 9.3.3.

5.1 Let Declarations

In Cyclone, you can write

```
let x = e;
```

as a local declaration. The meaning is the same as `t x = e;` where `t` is the type of `e`. In other words, `x` is bound to the new variable. Patterns are

much more powerful because they can bind several variables to different parts of an aggregate object. Here is an example:

```
struct Pair {  int x; int y; };
void f(struct Pair pr) {
  let Pair(fst,snd) = pr;
  ...
}
```

The pattern has the same structure as a `struct Pair` with parts being variables. Hence the pattern is a match for `pr` and the variables are initialized with the appropriate parts of `pr`. Hence "`let Pair(fst,snd) = pr`" is equivalent to "`int fst =pr.x; int snd = pr.y`". A let-declaration's initializer is evaluated only once.

Patterns may be as structured as the expressions against which they match. For example, given type

```
struct Quad { struct Pair p1; struct Pair p2; };
```

patterns for matching against an expression of type struct Quad could be any of the following (and many more because of constants and wildcards—see below):

- `Quad(Pair(a,b),Pair(c,d))`
- `Quad(p1, Pair(c,d))`
- `Quad(Pair(a,b), p2)`
- `Quad(p1,p2)`
- `q`

In general, a let-declaration has the form "let p = e;" where p is a pattern and `e` is an expression. In our example, the match always succeeds, but in general patterns can have compile-time errors or run-time errors.

At compile-time, the type-checker ensures that the pattern makes sense for the expression. For example, it rejects "let Pair(fst,snd) = 0" because 0 has type int but the pattern only makes sense for type `struct Pair`.

Certain patterns are type-correct, but they may not match run-time values. For example, constants can appear in patterns, so "let Pair(17,snd) =

pr;" would match only when `pr.x` is 17. Otherwise the exception `Match_Exception` is thrown. Patterns that may fail are rarely useful and poor style in let-declarations; the compiler emits a warning when you use them. In switch statements, possibly-failing patterns are the norm—as we explain below, the whole point is that one of the cases' patterns should match.

5.2 Pattern Forms

So far, we have seen three pattern forms: variables patterns, struct patterns, and constant patterns. We now describe all the pattern forms.[1] For each form, you need to know:

- The syntax
- The types of expressions it can match against (to avoid a compile-time error)
- The expressions the pattern matches against (other expressions cause a match failure)
- The bindings the pattern introduces, if any.

There is one compile-time rule that is the same for all forms: All variables (and type variables) in a pattern must be distinct. For example, "let Pair(fst,fst) = pr;" is not allowed.

You may want to read the descriptions for variable and struct patterns first because we have already explained their use informally.

- **Variable patterns**
  - Syntax: an identifer
  - Types for match: all types
  - Expressions matched: all expressions
  - Bindings introduced: the identifier is bound to the expression being matched
- **Wildcard patterns**

[1] Actually, we defer description to `alias` variable patterns until Section 9.6, in the context of a discussion on Cyclone's non-aliasable pointers.

  - Syntax: _ (underscore, note this use is completely independent of _ for type inference)
  - Type for match: all types
  - Expressions matched: all expressions
  - Bindings introduced: none. Hence it is like a variable pattern that uses a fresh identifier. Using _ is better style because it indicates the value matched is not used. Notice that "`let _ = e;`" is equivalent to `e`.

- **As patterns**
  - Syntax: `x as p` where `x` is an identifier and `p` is a pattern.
  - Types for match: all types
  - Expressions matched: all expressions
  - Bindings introduced: if the expression matches the pattern `p`, then its value is bound to `x`. Thus, a variable pattern is simply shorthand for "`x as _`".

- **Reference patterns**
  - Syntax: `*x` (i.e., the * character followed by an identifier)
  - Types for match: all types
  - Expressions matched: all expressions. (Very subtle notes: Currently, reference patterns may only appear inside of other patterns so that the compiler can determine the region for the pointer type assigned to `x`. They also may not occur under a `datatype` pattern that has existential types unless there is a pointer pattern in-between.)
  - Bindings introduced: `x` is bound to *the address of* the expression being matched. Hence if matched against a value of type `t` in region `'r`, the type of `x` is `t@'r`.

- **Numeric constant patterns**
  - Syntax: An `int`, `char`, or `float` constant
  - Types for match: numeric types

  - Expressions matched: numeric values such that `==` applied to the value and the pattern yields true. (Standard C numeric promotions apply. Note that comparing floating point values for equality is usually a bad idea.)
  - Bindings introduced: none

- **`NULL` constant patterns**
  - Syntax: `NULL`
  - Types for match: nullable pointer types, including `?` types
  - Expressions matched: `NULL`
  - Bindings introduced: none

- **Enum patterns**
  - Syntax: an enum constant
  - Types for match: the enum type containing the constant
  - Expressions matched: the constant
  - Bindings introduced: none

- **Tuple patterns**
  - Syntax: `$(p1,...,pn[,...])` where p1,...,pn are patterns. The trailing comma and ellipses (`...`) are optional.
  - Types for match: if no ellipses, then tuple types with exactly n fields, where pi matches the type of the tuple's ith field. If the ellipses are present, then matches a tuple with at least n fields.
  - Expressions matched: tuples where the ith field matches `pi` for `i` between 1 and `n`.
  - Bindings introduced: bindings introduced by `p1`, ..., `pn`.

- **Struct patterns**
  - Syntax: There are three forms:
    * `X(p1,...,pn[,...])` where `X` is the name of a struct with `n` fields and p1,...,pn are patterns. This syntax is shorthand for `X{.f1 = p1, ..., .fn = pn [,...]}` where `fi` is the ith field in `X`.

    - `X{.f1 = p1, ..., .fn = pn [,...]}` where the fields of `X` are f1, ..., fn but not necessarily in that order
    - `{.f1 = p1, ..., .fn = pn [,...]}` which is the same as above except that struct name `X` has been omitted.
  - Types for match: `struct X` (or instantiations when `struct X` is polymorphic) such that pi matches the type of fi for `i` between 1 and `n`. If the ellipses are not present, then each member of the struct must have a pattern.
  - Expressions matched: structs where the value in fi matches pi for `i` between 1 and `n`.
  - Bindings introduced: bindings introduced by p1,...,pn

- **Tagged Union patterns**
  - Syntax: There are two forms:
    - `X{.fi = p}` where the members of `X` are f1, ..., fn and fi is one of those members.
    - `{{.f1 = p` which is the same as above except that union name `X` has been omitted.
  - Types for match: `union X` (or instantiations when `union X` is polymorphic) such that p matches the type of fi.
  - Expressions matched: tagged unions where the last member written was fi and the value of that member matches p.
  - Bindings introduced: bindings introduced by p.

- **Pointer patterns**
  - Syntax: `&p` where p is a pattern
  - Types for match: pointer types, including `?` types. Also `datatype Foo @` (or instantiations of it) when the pattern is `&Bar(p1,...,pn)` and `Bar` is a variant of `datatype Foo` and pi matches the type of the ith value carried by Bar.
  - Expressions matched: non-null pointers where the value pointed to matches p. Note this explanation includes the case where the expression has type datatype Foo @ and the pattern is `&Bar(p1,...,pn)` and the current variant of the expression is "pointer to `Bar`".

  - Bindings introduced: bindings introduced by p

- **Datatype patterns**
  - Syntax: `X` if X is a variant that carries no values. Else `X(p1,...,pn[,...])` where X is the name of a variant and p1, ..., pn are patterns. As with tuple and struct patterns, the ellipses are optional.
  - Types for match: `datatype Foo` (or instantiations of it).
  - Expressions matched: If `X` is non-value-carrying, then `X`. If `X` is value-carrying, then values created from the constructor `X` such that pi matches the ith field.
  - Bindings introduced: bindings introduced by p1,...,pn

5.3 Switch Statements

In Cyclone, you can switch on a value of any type and the case "labels" (the part between case and the colon) are patterns. The switch expression is evaluated and then matched against each pattern in turn. The first matching case statement is executed. Except for some restrictions, Cyclone's switch statement is therefore a powerful extension of C's switch statement.

Restrictions

- *You cannot implicitly "fall-through" to the next case*. Instead, you must use the `fallthru;` statement, which has the effect of transferring control to the beginning of the next case. There are two exceptions to this restriction: First, adjacent cases with no intervening statement do not require a fall-through. Second, the last case of a switch does not require a fall-through or break.

- The cases in a switch *must be exhaustive*; it is a compile-time error if the compiler determines that it could be that no case matches. The rules for what the compiler determines are described below.

- A *case cannot be unreachable*. It is a compile-time error if the compiler determines that a later case may be subsumed by an earlier one. The

rules for what the compiler determines are described below. (C almost has this restriction because case labels cannot be repeated, but Cyclone is more restrictive. For example, C allows cases after a default case.)

- The body of a switch statement must be a *sequence of case statements* and case statements can appear only in such a sequence. So idioms like Duff's device (such as "`switch(i%4) while(i-- >=0) { case 3: ... }`") are not supported.

- A constant case label must be a constant, *not a constant expression*. That is, case 3+4: is allowed in C, but not in Cyclone. Cyclone supports this feature with a separate construct: `switch "C" (e) { case 3+4: ... }`. This construct is much more like C's `switch`: The labels must be constant numeric expressions and `fallthru` is never required.

An Extension of C Except for the above restrictions, we can see Cyclone's switch is an extension of C's switch. For example, consider this code (which has the same meaning in C and Cyclone):

```
int f(int i) {
  switch(i) {
  case 0:  f(34); return 17;
  case 1:  return 17;
  default: return i;
  }
}
```

In Cyclone terms, the code tries to match against the constant 0. If it does not match (i is not 0), it tries to match against the pattern 1. Everything matches against default; in fact, default is just alternate notation for "`case _`", i.e., a case with a wildcard pattern. For performance reasons, switch statements that are legal C switch statements are translated to C switch statements. Other switch statements are translated to, "a mess of tests and gotos".

We now discuss some of the restrictions in terms of the above example. Because there is no "implicit fallthrough" in non-empty cases, the

return statement in case 0 cannot be omitted. However, we can replace the "return 17;" with "fallthru;" a special Cyclone statement that immediately transfers control to the next case. fallthru does not have to appear at the end of a case body, so it acts more like a goto than a fallthrough. As in our example, any case that matches all values of the type switched upon (e.g., `default:`, `case _:`, `case x:`) must appear last, otherwise later cases would be unreachable. (Note that other types may have even more such patterns. For example `Pair(x,y)` matches all values of type `struct Pair int x; int y;`).

Much More Powerful Because Cyclone case labels are patterns, a switch statement can match against any expression and bind parts of the expression to variables. Also, **fallthru can (in fact, must) bind values** to the next case's pattern variables. This silly example demonstrates all of these features:

```
extern int f(int);}
int g(int x, int y) {
  // return f(x)*f(y), but try to avoid using multiplication
  switch($(f(x),f(y))) {
  case $(0,_): fallthru;
  case $(_,0): return 0;
  case $(1,b): fallthru(b+1-1);
  case $(a,1): return a;
  case $(a,b): return a*b;
  }
}
```

The only part of this example using a still-unexplained feature is "fallthru(b)", but we explain the full example anyway. The switch expression has type `$(int,int)`, so all of the cases must have patterns that match this type. Legal case forms for this type not used in the example include "`case $(_,id):`", "`case $(id,_):`", "`case id:`", "`case _:`", and "default:".

The code does the following:

- It evaluates the pair `$(f(x), f(y))` and stores the result on the stack.

- If f(x) returned 0, the first case matches, control jumps to the second case, and 0 is returned.
- Else if f(y) returned 0, the second case matches and 0 is returned.
- Else if f(x) returned 1, the third case matches, b is assigned the value f(y) returned, control jumps to the fourth case after assigning b+1-1 to a, and a (i.e., b + 1 - 1, i.e., b, i.e., f(y)) is returned.
- Else if f(y) returned 1, the fourth case matches, a is assigned the value f(x) returned, and a is returned.
- Else the last case matches, a is assigned the value f(x) returned, b is assigned the value f(y) returned, and a*b is returned.

Note that the switch expression is evaluated only once. Implementation-wise, the result is stored in a compiler-generated local variable and the value of this variable is used for the ensuring pattern matches.

The general form of fallthrus is as follows: If the next case has no bindings (i.e., identifiers in its pattern), then you must write `fallthru;`. If the next case has n bindings, then you must write `fallthru(e1,...,en)` where each ei is an expression with the appropriate type for the ith binding in the next case's pattern, reading from left to right. (By appropriate type, we mean the type of the expression that would be bound to the ith binding were the next case to match.) The effect is to evaluate e1 through en, bind them to the identifiers, and then goto the body of the next case. `fallthru` is not allowed in the last case of a switch, not even if there is an enclosing switch.

We repeat that fallthru may appear anywhere in a case body, but it is usually used at the end, where its name makes the most sense. ML programmers may notice that fallthru with bindings is strictly more expressive than or-patterns, but more verbose.

Case Guards We have withheld the full form of Cyclone case labels. In addition to case p: where p is a pattern, you may write `case p && e:` where p is a pattern and e is an expression of type int. (And since `e1 && e2` is an expression, you can write `case p && e1 && e2:` and so on.) Let's call e the case's *guard*.

The case matches if p matches the expression in the switch and e evaluates to a non-zero value. e is evaluated only if p matches and only after the bindings caused by the match have been properly initialized. Here is a silly example:

```
extern int f(int);
int g(int a, int b) {
  switch ($(a,b-1)) {
  case $(0,y) && y > 1: return 1;
  case $(3,y) && f(x+y) == 7 : return 2;
  case $(4,72): return 3;
  default: return 3;
  }
}
```

The function g returns 1 if a is 0 and b is greater than 2. Else if x is 3, it calls the function f (which of course may do arbitrary things) with the sum of a and b. If the result is 7, then 2 is returned. In all other cases (x is not 3 or the call to f does not return 7), 3 is returned.

Case guards make constant patterns unnecessary (we can replace case 3: with `case x && x==3:`, for example), but constant patterns are better style and easier to use.

Case guards are not interpreted by the compiler when doing exhaustiveness and overlap checks, as explained below.

Exhaustiveness and Useless-Case Checking As mentioned before, it is a compile-time error for the type of the switch expression to have values that none of the case patterns match or for a pattern not to match any values that earlier patterns do not already match. Rather than explain the precise rules, we currently rely on your intuition. But there are two rules to guide your intuition:

- In terms of exhaustiveness checking, the compiler acts as if any case guard might evaluate to false.
- In terms of exhaustiveness checking, numeric constants cannot make patterns exhaustive. Even if you list out all 256 characters, the compiler will act as though there is another possibility you have not checked.

We emphasize that checking does not just involve the "top-level" of patterns. For example, the compiler rejects the switch below because the third case is redundant:

```
enum Color { Red, Green };
void f(enum Color c1, enum Color c2) {
  switch ($(c1,c2)) {
  case $(Red,x): return;
  case $(x,Green): return;
  case $(Red,Green): return;
  default: return;
  }
}
```

Rules for No Implicit Fall-Through As mentioned several times now, Cyclone differs from C in that a case body may not implicitly fall-through to the next case. It is a compile-time error if such a fall-through might occur. Because the compiler cannot determine exactly if an implicit fall-through could occur, it uses a precise set of rules, which we only sketch here. The exact same rules are used to ensure that a function (with return type other than void) does not "fall off the bottom." The rules are very similar to the rules for ensuring that Java methods do not "fall off the bottom."

The general intuition is that there must be a break, continue, goto, return, or throw along all control-flow paths. The value of expressions is not considered except for numeric constants and logical combinations (using `&&`, `||`, and `? :`) of such constants. The statement try s catch ... is checked as though an exception might be thrown at any point while s executes.

6 Type Inference

Cyclone allows many explicit types to be elided. In short, you write `_` (underscore) where a type should be and the compiler tries to figure out the type for you. Type inference can make C-like Cyclone code easier to write and more readable. For example,

```
_ x = malloc(sizeof(sometype_t));
```

is a fine substitute for

```
sometype_t * x = malloc(sizeof(sometype_t));
```

Of course, explicit types can make code more readable, so it is often better style not to use inference.

Inference is even more useful because of Cyclone's advanced typing constructs. For example, it is much easier to write down _ than a type for a function pointer.

We now give a rough idea of when you can elide types and how types get inferred. In practice, you tend to develop a sense of which idioms succeed, and, if there's a strange compiler-error message about a variable's type, you give more explicit information about the variable's type.

Syntax As far as the parser is concerned, _ is a legal type specifier. However, the type-checker will immediately reject _ in these places (or at least it should):

- As part of a top-level variable or function's type.
- As part of a `struct`, `union`, `datatype`, or `typedef` declaration.

Note that _ can be used for part of a type. A silly example is `$(_,int) = $(3,4)`; a more useful example is an explicit cast to a non-nullable pointer (to avoid a compiler warning). For example:

```
void f(some_big_type * x, some_big_type @ y) {
if(x != NULL) {
  y = (_ *@notnull) x;
}
```

Semantics Except for the subtleties discussed below, using _ should not change the meaning of programs. However, it may cause a program not to type-check because the compiler no longer has the type information it needs at some point in the program. For example, the compiler rejects `x->f` if it does not know the type of `x` because the different struct types can have members named `f`.

The compiler infers the types of expressions based on uses. For example, consider:

```
_ x = NULL;
x = g();
x->f;
```

This code will type-check provided the return type of `g` is a pointer to a struct with a field named `f`. If the two statements were in the other order, the code would not type-check. Also, if `g` returned an `int`, the code would not type-check, even without the `x->f` expression, because the `_ x = NULL` constrains `x` to have a pointer type.

However, the above discussion assumes that sequences of statements are type-checked in order. This is true, but *in general the type-checker's order is unspecified*.

Subtleties In general, inference has subtle interactions with implicit coercions (such as from `t*@notnull` to `t*@nullable`) and constants that have multiple types (such as numeric constants).

The following is a desirable property: If a program is modified by replacing some explicit types with _ and the program still type-checks, then its meaning is the same. *This property does not hold!* Here are two examples:

Numeric Types This program prints -24 1000:

```
int f() {
char c = 1000;
return c;
}
int g() {
_ c = 1000; // compiler infers int
return c;
}
int main() {
printf("%d %d", f(), g());
return 0;
}
```

Order Matters Here is an example where the function's meaning depends on the order the type-checker examines the function:

```
void h1(int *@notnull c, int maybe) {
  _ a;
  if(maybe)
    a = c;
  else
    a = NULL;
}
```

At first, the type of `a` is completely unconstrained. If we next consider `a = c`, we will give `a` the type of `c`, namely `int *@notnull`, an int pointer that cannot be `NULL`. Clearly that makes the assignment `a = NULL` problematic, but Cyclone allows assignment from nullable pointers to non-nullable pointers; it gives a compile-time warning and inserts a run-time check that the value is not `NULL`. Here the check will fail and an exception will be raised. That is, `h1(p,0)` is guaranteed to raise an exception.

But what if the type-checker examines `a = NULL` first? Then the type-checker will constrain `a`'s type to be a nullable pointer to an unconstrained type. Then the assignment `a = c` will constrain that type to be `int`, so the type of `a` is `int *`. An assignment from `int *@notnull` to `int *` is safe, so there is no warning. Moreover, the assignment `a = NULL` is not a run-time error.

The order of type-checking is left unspecified. In the future, we intend to move to a system that is order-independent.

7 Polymorphism

Use `'a` instead of `void *`.

8 Memory Management Via Regions

8.1 Introduction

C gives programmers complete control over how memory is managed. An expert programmer can exploit this to write very fast and/or space-efficient programs. However, bugs that creep into memory-management code can cause crashes and are notoriously hard to debug.

Languages like Java and ML use garbage collectors instead of leaving memory management in the hands of ordinary programmers. This makes memory management much safer, since the garbage collector is written by experts, and it is used, and, therefore, debugged, by every program. However, removing memory management from the control of the applications programmer can make for slower programs.

Safety is the main goal of Cyclone, so we provide a garbage collector. But, like C, we also want to give programmers as much control over memory management as possible, without sacrificing safety. Cyclone's region system is a way to give programmers more explicit control over memory management.

In Cyclone, objects are placed into *regions*. A region is simply an area of memory that is allocated and deallocated all at once (but not for our two special regions; see below). So to deallocate an object, you deallocate its region, and when you deallocate a region, you deallocate all of the objects in the region. Regions are sometimes called "arenas" or "zones."

Cyclone has four kinds of region:

Stack regions As in C, local variables are allocated on the runtime stack; the stack grows when a block is entered, and it shrinks when the block exits. We call the area on the stack allocated for the local variables of a block the *stack region* of the block. A stack region has a fixed size—it is just large enough to hold the locals of the block, and no more objects can be placed into it. The region is deallocated when the block containing the declarations of the local variables finishes executing. With respect to regions, the parameters of a function are considered locals—when a function is called, its actual parameters are placed in the same stack region as the variables declared at the start of the function.

Lexical regions Cyclone also has *lexical regions*, which are so named because, like stack regions, their lifetime is delimited by the surrounding scope. Unlike stack regions, however, you can can add new objects to a lexical region over time. You create a lexical region in Cyclone with a statement,

`region` *identifier*`;` *statement*

This declares and allocates a new lexical region, named *identifier*, and executes *statement*. After *statement* finishes executing, the region is

deallocated. Within *statement*, objects can be added to the region, as we will explain below.

Typically, *statement* is a compound statement:

```
{ region identifier;
  statement_1
  ...
  statement_n
}
```

The heap region Cyclone has a special region called the *heap*. There is only one heap, whose type is denoted `'H`, and it is never deallocated. New objects can be added to the heap at any time (the heap can grow). Cyclone uses a garbage collector to automatically remove objects from the heap when they are no longer needed. You can think of garbage collection as an optimization that tries to keep the size of the heap small. (Alternatively, you can avoid garbage collection all together by specifying the `-nogc` flag when building the executable.)

Dynamic regions Stack and lexical regions obey a strictly last-in-first-out (LIFO) lifetime discipline. This is often convenient for storing temporary data, but sometimes, the lifetime of data cannot be statically determined. Such data can be allocated in a *dynamic region*. A dynamic region supports deallocation at (essentially) any program point. However, before the data in a dynamic region may be accessed, the dynamic region must be *opened*. The open operation fails by throwing an exception if the dynamic region has already been freed. Note that each data access within a dynamic region does not require a check. Rather, you can open a given dynamic region once, access the data many times with no additional cost, and then exit the scope of the open. Thus, dynamic regions amortize the cost of checking whether or not data are still live and localize failure points. We describe dynamic regions in detail in Section 9.7.

Cyclone forbids dereferencing dangling pointers. This restriction is part of the type system: it's a compile-time error if a dangling pointer (a pointer into a deallocated region or to a deallocated object) might be dereferenced. There are no run-time checks of the form, "is this pointing

into a live region?" As explained below, each pointer type has a region and objects of the type may only point into that region.

8.2 Allocation

You can create a new object on the heap using one of a few kinds of expression:

- `new` *expr* evaluates *expr*, places the result into the heap, and returns a pointer to the result. It is roughly equivalent to

  ```
  t @ temp = malloc(sizeof(t));
  // where t is the type of expr
  *temp = expr;
  ```

  For example, `new 17` allocates space for an integer on the heap, initializes it to 17, and returns a pointer to the space. For another example, if we have declared

  ```
  struct Pair { int x; int y; };
  ```

  then `new Pair(7,9)` allocates space for two integers on the heap, initializes the first to 7 and the second to 9, and returns a pointer to the first.

- `new` *array-initializer* allocates space for an array, initializes it according to *array-initializer*, and returns a pointer to the first element. For example,

  ```
  let x = new { 3, 4, 5 };
  ```

  declares a new array containing 3, 4, and 5, and initializes `x` to point to the first element. More interestingly,

  `new { for` *identifier* `<` $expr_1$ `:` $expr_2$ `}`

  is roughly equivalent to

```
unsigned int sz = expr1;
t @ temp = malloc(sz * sizeof(t2)); // where t is the  type of e
for (int identifier = 0; identifier < sz; identifier++)
  temp[identifier] = expr2;
```

That is, $expr_1$ is evaluated first to get the size of the new array, the array is allocated, and each element of the array is initialized by the result of evaluating $expr_2$. $expr_2$ may use *identifier*, which holds the index of the element currently being initialized.

For example, this function returns an array containing the first `n` positive even numbers:

```
int *@fat n_evens(int n) {
  return new {for next < n :  2*(next+1)};
}
```

Note that:

- $expr_1$ is evaluated exactly once, while $expr_2$ is evaluated $expr_1$ times.
- $expr_1$ might evaluate to 0.
- $expr_1$ might evaluate to a negative number. If so, it is implicitly converted to a very large unsigned integer; the allocation is likely to fail due to insufficient memory. Currently, this will cause a crash!!
- Currently, `for` array initializers are the only way to create an object whose size depends on run-time data.

- `malloc(sizeof(`*type*`))`. Returns a `@notnull` pointer to an uninitialized value of type *type*.
- `malloc(n*sizeof(`*type*`))` or `malloc(sizeof(`*type*`)*n)`. The type must be a bits-only type (i.e., cannot contain pointers, tagged unions, zero-terminated values, etc.) If `n` is a compile-time constant expression, returns a `@thin` pointer with `@numelts(n)`. If `n` is not a compile-time constant, returns a `@fat` pointer to the sequence of `n` uninitialized values.

- `calloc(n,sizeof(`*type*`))`. Similar to the malloc case above, but returns memory that is zero'd. Therefore, `calloc` supports types that are bits-only or zero-terminated.
- `malloc(e)` where `e` is an expression not of one of the above forms. If `e` is constant, returns a `char *@numelts(e)@nozeroterm` otherwise returns a `char *@fat@nozeroterm`.

Objects within regions can be created using the following analogous expressions.

- `rnew(`*identifier*`)` *expr*
- `rnew(`*identifier*`)` *array-initializer*
- `rmalloc(`*identifier*`,sizeof(`*type*`))`
- `rmalloc(`*identifier*`,n*sizeof(`*type*`))`
- `rmalloc(`*identifier*`,sizeof(`*type*`)*n)`
- `rmalloc(`*identifier*`,e))`
- `rcalloc(`*identifier*`,n,sizeof(`*type*`))`

Note that `new`, `malloc`, `calloc`, `rnew`, `rmalloc` and `rcalloc` are keywords.

Here, the first argument specifies a region handle. The Cyclone library has a global variable `Core::heap_region` which is a handle for the heap region. So, for example, `rnew (heap_region)` *expr* allocates memory in the heap region which is initialized with *expr*. Moreover, `new` *expr* can be replaced with `rnew(heap_region)` *expr*.

The only way to create an object in a stack region is declaring it as a local variable. Cyclone does not currently support `salloc`; use a lexical region instead.

8.3 Common Uses

Although the type system associated with regions is complicated, there are some simple common idioms. If you understand these idioms, you should be able to easily write programs using regions, and port many

legacy C programs to Cyclone. The next subsection will explain the usage patterns of unique and reference-counted pointers, since they are substantially more restrictive than other pointers.

Remember that every pointer points into a region, and although the pointer can be updated, it must always point into that same region (or a region known to outlive that region). The region that the pointer points to is indicated in its type, but omitted regions are filled in by the compiler according to context.

When regions are omitted from pointer types in function bodies, the compiler tries to infer the region. However, it can sometimes be too "eager" and end up rejecting code. For example, in

```
void f1(int * x) {
  int * y = new 42;
  y = &x;
}
```

the compiler uses y's initializer to decide that y's type is `int * 'H`. Hence the assignment is illegal, the parameter's region (called `'f1`) does not outlive the heap. On the other hand, this function type-checks:

```
void f2(int x) {
  int * y = &x;
  y = new 42;
}
```

because y's type is inferred to be `int * 'f2` and the assignment makes y point into a region that outlives `'f2`. We can fix our first function by being more explicit:

```
void f1(int * x) {
  int *'f1 y = new 42;
  y = &x;
}
```

Function bodies are the only places where the compiler tries to infer the region by how a pointer is used. In function prototypes, type declarations, and top-level global declarations, the rules for the meaning of omitted region annotations are fixed. This is necessary for separate compilation: we often have no information other than the prototype or declaration.

In the absence of region annotations, function-parameter pointers are assumed to point into any possible region. Hence, given

```
void f(int * x, int * y);
```

we could call f with two stack pointers, a lexical-region pointer and a heap-pointer, etc. Hence this type is the "most useful" type from the caller's perspective. But the callee's body (f) may not type-check with this type. For example, x cannot be assigned a heap pointer because we do not know that x points into the heap. If this is necessary, we must give x the type `int *'H`. Other times, we may not care what region x and y are in so long as they are the *same* region. Again, our prototype for f does not indicate this, but we could rewrite it as

```
void f(int *'r x, int *'r y);
```

Finally, we may need to refer to the region for x or y in the function body. If we omit the names (relying on the compiler to make up names), then we obviously won't be able to do so.

Formally, omitted regions in function parameters are filled in by fresh region names and the function is "region polymorphic" over these names (as well as all explicit regions).

In the absence of region annotations, function-return pointers are assumed to point into the heap. Hence the following function will not type-check:

```
int * f(int * x) { return x; }
```

Both of these functions will type-check:

```
int * f(int *'H x) { return x; }
int *'r f(int *'r x) {return x; }
```

The second one is more useful because it can be called with any region.

In type declarations (including `typedef`) and top-level variables, omitted region annotations are assumed to point into the heap. In the future, the meaning of `typedef` may depend on where the `typedef` is used. In the meantime, the following code will type-check because it is equivalent to the first function in the previous example:

```
typedef int * foo_t;
foo_t f(foo_t x) { return x; }
```

If you want to write a function that creates new objects in a region determined by the caller, your function should take a region handle as one of its arguments. The type of a handle is `region_t<'r>`, where `'r` is the region information associated with pointers into the region. For example, this function allocates a pair of integers into the region whose handle is `r`:

```
$(int,int)*'r f(region_t<'r> r, int x, int y) {
   return rnew(r) $(x,y);
}
```

Notice that we used the same `'r` for the handle and the return type. We could have also passed the object back through a pointer parameter like this:

```
void f2(region_t<'r> r,int x,int y,$(int,int)*'r *'s p){
  *p = rnew(r) $(7,9);
}
```

Notice that we have been careful to indicate that the region where `*p` lives (corresponding to `'s`) may be different from the region for which `r` is the handle (corresponding to `'r`). Here's how to use `f2`:

```
{ region rgn;
  $(int,int) *'rgn x = NULL;
  f2(rgn,3,4,&x);
}
```

The `'s` and `'rgn` in our example are unnecessary because they would be inferred.

`typedef`, `struct`, and `datatype` declarations can all be parameterized by regions, just as they can be parameterized by types. For example, here is part of the list library.

```
struct List<'a,'r>{'a hd; struct List<'a,'r> *'r tl;};
typedef struct List<'a,'r> *'r list_t<'a,'r>;

// return a fresh copy of the list in r2
list_t<'a,'r2> rcopy(region_t<'r2> r2, list_t<'a> x) {
  list_t result, prev;
```

```
  if (x == NULL) return NULL;
  result = rnew(r2) List{.hd=x->hd,.tl=NULL};
  prev = result;
  for (x=x->tl; x != NULL; x=x->tl) {
    prev->tl = rnew(r2) List(x->hd,NULL);
    prev = prev->tl;
  }
  return result;
}
list_t<‘a> copy(list_t<‘a> x) {
  return rcopy(heap_region, x);
}

// Return the length of a list.
int length(list_t x) {
  int i = 0;
  while (x != NULL) {
    ++i;
    x = x->tl;
  }
  return i;
}
```

The type `list_t<`*`type`*`,`*`rgn`*`>` describes pointers to lists whose elements have type *type* and whose "spines" are in *rgn*.

The functions are interesting for what they *don't* say. Specifically, when types and regions are omitted from a type instantiation, the compiler uses rules similar to those used for omitted regions on pointer types. More explicit versions of the functions would look like this:

```
list_t<‘a,‘r2> rcopy(region_t<‘r2> r2, list_t<‘a,‘r1> x) {
  list_t<‘a,‘r2> result, prev;
  ...
}
list_t<‘a,‘H> copy(list_t<‘a,‘r> x) { ... }
int length(list_t<‘a,‘r> x) { ... }
```

8.4 Type-Checking Regions

Because of recursive functions, there can be any number of live regions at run time. The compiler uses the following general strategy to ensure that only pointers into live regions are dereferenced:

- Use compile-time *region names*. Syntactically these are just type variables, but they are used differently.
- Decorate each pointer type and handle type with one or more region names.
- Decorate each program point with a (finite) set of region names. We call the set the point's *capability*.
- To dereference a pointer (via `*`, `->`, or subscript), each region name that appears in the pointer's type name must be in the program point's capability. Similarly, to use a handle for allocation, the handle type's region name must be in the capability.
- Enforce a type system such that the following is impossible: A program point P's capability contains a region name `'r` that decorates a pointer (or handle) expression *expr* that, at run time, points into a region that has been deallocated and the operation at P dereferences *expr*.

This strategy is probably too vague to make sense at this point, but it may help to refer back to it as we explain specific aspects of the type system.

Note that in the rest of the documentation (and in common parlance) we abuse the word "region" to refer both to region names and to run-time collections of objects. Similarly, we confuse a block of declarations, its region-name, and the run-time space allocated for the block. (With loops and recursive functions, "the space allocated" for the block is really any number of distinct regions.) But in the rest of this section, we painstakingly distinguish region names, regions, etc.

8.4.1 Region Names

Given a function, we associate a distinct region name with each program point that creates a region, as follows:

- If a block (blocks create stack regions) has label `L`, then the region-name for the block is `'L`.
- If a block has no label, the compiler makes up a fresh region-name for the block.
- In `region r <'foo>` s, the region-name for the construct is `'foo`.
- In `region r` s, the region-name for the construct is `'r`.
- In region `h = open(k) s`, the region-name for the construct is `'r`, assuming `k` has type `region_key_t<'r,_>` (where _ is some other region name of no consequence).

The region name for the heap is `'H`. Region names associated with program points within a function should be distinct from each other, distinct from any region names appearing in the function's prototype, and should not be `'H`. (So you cannot use `H` as a label name, for example.) Because the function's return type cannot mention a region name for a block or region-construct in the function, it is impossible to return a pointer to deallocated storage.

In region `r <'r> s`, region `r s`, and region `r = open(k) s` the type of `r` is `region_t<'r>` (assuming, that `k` has type `region_key_t<'r,_->`). In other words, the handle is decorated with the region name for the construct. Pointer types' region names are explicit, although you generally rely on inference to put in the correct one for you.

8.4.2 Capabilities

In the absence of explicit effects (see below), the capability for a program point includes exactly:

- `'H`
- The effect corresponding to the function's prototype. Briefly, any region name in the prototype (or inserted by the compiler due to an omission) is in the corresponding effect. Furthermore, for each type variable `'a` that appears (or is inserted), "`regions('a)`" is in the corresponding effect. This latter effect roughly means, "I don't know what `'a` is, but if you instantiate with a type mentioning some

regions, then add those regions to the effect of the instantiated prototype." This is necessary for safely type-checking calls that include function pointers.

- The region names for the blocks and "`region r s`" statements that contain the program point

For each dereference or allocation operation, we simply check that the region name for the type of the object is in the capability. It takes extremely tricky code (such as existential region names) to make the check fail.

8.4.3 Type Declarations

Each pointer and region handle type is decorated with a set of region names. This set of region names is referred to as the store or effect qualifier of a type. For instance a pointer type might be `int *'r+'H`. This indicates the type of pointer to an integer that resides in the region named 'r or the heap region, 'H. Similarly, a region handle type might be `region_t<'r1+'r2>` which indicates a handle to region named 'r1 or 'r2.

A `struct`, `typedef`, or `datatype` declaration may be parameterized by any number of effect qualifiers. The store qualifiers are placed in the list of type parameters. They may be followed by their kind; i.e. "`::E`". For example, given

```
struct List<'a,'r>{'a hd; struct List<'a,'r> *'r tl;};
```

the type `struct List<int,'H>` is for a list of ints in the heap, while the type `struct List<int,'l>` is for a list of ints in some lexical region. Notice that all of the "cons cells" of the `List` will be in the same region (the type of the `tl` field uses the same region name `'r` that is used to instantiate the recursive instance of `struct List<'a,'r>`). However, we could instantiate `'a` with a pointer type that has a different region name.

8.4.4 Subtyping and Effect Qualifiers

A pointer type's effect qualifier is part of the type. If `e1` and `e2` are pointers, then `e1 = e2` is well-typed only if all the regions names mentioned in the the effect qualifier for `e2`'s type also appears in the effect qualifier

of e1's type. For instance, both assignments to b below is legal, while the assignment to a is not.

```
void foo(int *‘r a) {
  int *‘r+‘H b = a;
  if(!a) b = new 1;
  a = b;
}
```

The store qualifier in the type of b indicates that it can point into the region named ‘r or into the heap region. Therefore, initializing b with a pointer into ‘r is certainly consistent with its store qualifier. Similarly, the second assignment to b is legal since it is updated to point to the heap region. However, the assignment to a is not permitted since the declared type of a claims that it is pointer into the region named ‘r alone and b may actually point to the heap.

For handles, if ‘r is a region name, there is at most one value of type region_t<‘r> (there are 0 if ‘r is a block's name), so there is little use in creating variables of type region_t<‘r>. However, it could be useful to create variables of type region_t<‘r1+‘r2>. This is the type of handle to either the region named ‘r1 or to the region named ‘r2. This is illustrated by the following code:

```
void foo(int a, region_t<‘r> h) {
  region_t<‘r+‘H> rh = h;
  if(a) {
    rh = heap_region;
  }
}
```

As always, this form of subtyping for effect qualifiers is not permitted under a reference. Thus, the assignment in the program below is not legal.

```
void foo(int *‘H *‘r a) {
  int *‘r+‘H *‘r b = a;
}
```

8.4.5 Function Calls

If a function parameter or result has type `int *'r` or `region_t<'r>`, the function is polymorphic over the region name `'r`. That is, the caller can instantiate `'r` with any region *in the caller's current capability* as long as the region has the correct kind. This instantiation is usually implicit, so the caller just calls the function and the compiler uses the types of the actual arguments to infer the instantiation of the region names (just like it infers the instantiation of type variables).

The callee is checked knowing nothing about `'r` except that it is in its capability (plus whatever can be determined from explicit outlives assumptions), and that it has the given kind. For example, it will be impossible to assign a parameter of type `int*'r` to a global variable. Why? Because the global would have to have a type that allowed it to point into any region. There is no such type because we could never safely follow such a pointer (since it could point into a deallocated region).

8.4.6 Explicit and Default Effects

If you are not using existential types, you now know everything you need to know about Cyclone regions and memory management. Even if you are using these types and functions over them (such as the closure library in the Cyclone library), you probably don't need to know much more than "provide a region that the hidden types outlive".

The problem with existential types is that when you "unpack" the type, you no longer know that the regions into which the fields point are allocated. We are sound because the corresponding region names are not in the capability, but this makes the fields unusable. To make them usable, we do not hide the capability needed to use them. Instead, we use a *region bound* that is not existentially bound.

If the contents of existential packages contain only heap pointers, then `'H` is a fine choice for a region bound.

These issues are discussed in Section 13.

9 Pointers with Restricted Aliasing

9.1 Introduction

The main benefit of the regions described thus far is also their drawback: to free data you must free an entire region. This implies that to amortize the cost of creating a region, one needs to allocate into it many times. Furthermore, the objects allocated in a region should be mostly in use until the region is freed, or else memory will be wasted in the region that is unused by the program.

In an attempt to alleviate each of these problems Cyclone provides a mechanism by which individual objects in a region may be freed prior to freeing the entire region. Pointers to objects that may be freed early must obey an aliasing discipline to prevent dangling pointers. A static analysis ensures that such objects can only ever be accessed through one pointer at any time. At the time it is freed, this pointer is invalidated, thus preventing all future accesses to the object.

Pointer types in Cyclone can be qualified by their aliasability. As of now, there are four alias qualifiers. Each of these qualifiers must appear as arguments to the `@aqual(...)` pointer qualifier mentioned in section 3. The four alias qualifiers are:

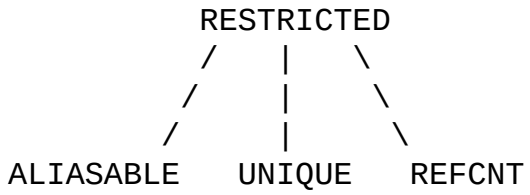
Aliasable Pointer types are by default qualified as `ALIASABLE`. These may be freely aliased, but can never be freed. For instance, `int @@aqual(ALIASABLE)` is an aliasable non-null pointer to an `int`.

Unique The `UNIQUE` qualifier on a pointer allows the object pointed to by that pointer to be deallocated individually, using the function `rufree`. For freeing objects to be safe, all accesses to such objects must be made through a single `UNIQUE` pointer. That is, only a single pointer may be used to access the object at any given time; this trivially guarantees that if the object is freed through its unique pointer, no other access to the object beyond that point is possible. Objects that become unreachable but are not freed manually will be freed by the garbage collector (assuming it's not removed with `-nogc`). For instance, `int ?@aqual(UNIQUE)` is a unique fat pointer to an `int`.

Reference-counted The `REFCNT` qualifier reference-counted qualifier also permits freeing individual objects. Unlike the `UNIQUE` qualifier, mul-

tiple pointers to a single object are permitted, the number of which is tracked dynamically via a hidden reference count stored with the object. Additional pointers to an object are created explicitly via a call to `alias_refptr`, which increases the reference count. Individual pointers are removed via a call to `rdrop_refptr`; when the last pointer is removed (i.e. the reference count is 0), the object is freed. Like the unique region, objects that become unreachable will be freed by the garbage collector. For instance, `int *@aqual(REFCNT)` is a nullable pointer to a reference counted `int`.

Restricted All the alias qualifiers are arranged in a subtyping hierarchy. The `RESTRICTED` qualifier is a super-type of all the other qualifiers. A `RESTRICTED` pointer may not be freed nor can any aliases of the pointer be created. For instance, `int *@aqual(RESTRICTED)` is a restricted nullable pointer to an `int`. The subtyping hierarchy is as below:



9.2 Allocation and Freeing

Unique and reference-counted pointers can be allocated in any region. However, to support freeing individual objects from a region we use a special allocator (see Section 9.10) that maintains additional metadata. We use the term "reap" to refer to regions from which individual objects can be deleted. Any region can contain objects that can be individually freed. Thus all regions are in fact reaps.

The allocation mechanism for unique and reference-counted pointers follows a pattern similar to that used for normal region allocation described in the previous section. The allocation functions rely on handles for alias qualifiers which are of type `aqual_t<'q::Q>`. Here `'q` is a type variable of alias qualifier kind and may be instantiated with the alias qualifiers above. The Cyclone libraries provide the following global variables:

```
aqual_t<ALIASABLE> Core::aliasable_qual
aqual_t<UNIQUE> Core::unique_qual
aqual_t<REFCNT> Core::refcnt_qual
```

which are handles for the aliasable, unique and reference-counted qualifiers respectively. Note that there is no handle for the restricted qualifier.

The following expressions are used for allocation

- `qnew(`*aq-identifier*`)` *expr*
- `qnew(`*aq-identifier*`)` *array-initializer*

  The `qnew` functions allocate in the `'H` region, i.e. the heap. They expect a alias qualifier handle. For instance, to allocate a unique integer in the heap, one might use

  ```
  int @@aqual(UNIQUE) a = qnew(Core::unique_qual) 0;
  ```

- `rnew(`*rgn-identifier, aq-identifier*`)`*expr*
- `rnew(`*rgn-identifier, aq-identifier*`)`*array-initializer*

  To allocate an alias-free object in a region other than the heap the `rnew` functions can be used. For instance,

  ```
  {
    region r;
    int @@aqual(UNIQUE) 'r a = rnew(r, Core::unique_qual) 0;
  }
  ```

  If *aq-identifier* is not specified it defaults to the `ALIASABLE` handle.

- `qmalloc(`*aq-identifier*`,sizeof(`*type*`))`
- `qcalloc(`*aq-identifier*`,n,sizeof(`*type*`))`

  Similar to `qnew` these functions allocate in the heap.

- `rmalloc(`*rgn-identifier, aq-identifier*`,sizeof(`*type*`))`

- `rcalloc(`*rgn-identifier, aq-identifier*`,n,sizeof(`*type*`))`

  Similar to `rnew` these functions can be used to allocate alias restricted pointers in the specified region. If *aq-identifier* is omitted, it defaults to `ALIASABLE`.

Since there is no handle for the restricted qualifier, `RESTRICTED` qualified pointers cannot be allocated. As with normal region allocation, we use very simple type inference to ease the burden of writing qualifiers on types. For instance,

```
int *a = rnew(r, Core::unique_qual) 0;
```

suffices to allocate a unique integer in the region `r`.

The Cyclone library provides the following functions to free alias-free pointers.

- `Core::ufree(`*ptr*`)` To free a heap directed unique pointer `ufree` is used.
- `Core::rufree(`*rgn-handle, ptr*`)` To free a unique pointer from a reap `rufree` is used. A constraint on the effect qualifiers that appear on the arguments of this function is described in Section 9.9.
- `Core::drop_refptr(`*ptr*`)` To reduce the reference count on a heap resident reference counted object `drop_refptr` is used. The *ptr* can no longer be dereferenced following this call. If the reference count on the object reaches zero then the object is freed automatically.
- `Core::rdrop_refptr(`*rgn-handle, ptr*`)` To reduce the reference count on a reference counted object in an arbitrary region `rdrop_refptr` is used.

9.3 Unique Pointers

To ease programming with unique pointers and allow reuse of library code, unique pointers can be aliased temporarily within a designated lexical scope using a special `alias` pattern. If this kind of aliasing is not sufficient, users can choose to allocate reference-counted objects; this idea is explained in the next subsection. We also define syntax *a* `:=:` *b* to

allow two unique pointers a and b to be atomically swapped. Careful use of the swap operator allows us to store unique pointers in objects that are not themselves uniquely pointed to. We also introduce bounded polymorphism over alias qualifiers and add an additional *kind* to specify type variables over these qualifiers. In practice, all of these mechanisms are necessary for writing useful and reusable code.

9.3.1 Simple Unique Pointers

Having a unique pointer ensures the object pointed to is not reachable by any other means. When pointers are first allocated, e.g. using `malloc`, they are unique. Such pointers are allowed to be *read through* (that is, dereferenced or indexed) but not copied, as the following example shows:

```
char @fat @aqual(UNIQUE) buf = malloc(3*sizeof(char));
buf[0] = 'h';
buf[1] = 'i';
buf[2] = '\0';
...
ufree(buf);
```

This piece of code allocates a unique buffer, and then indexes that buffer three times to initialize it. Because the process of storing to the buffer does not copy its unique pointer, it can be safely freed.

When a unique pointer is copied, e.g. when passed as a parameter to a function or stored in a datastructure, we say it has been *consumed*. We ensure that consumed pointers are not read through or copied via a dataflow analysis. When a consumed pointer is assigned to, very often it can be *unconsumed*, making it accessible again. Here is a simple example that initializes a datastructure with unique pointers:

```
1  struct pair { int *@aqual(UNIQUE) x; int *@aqual(UNIQUE) y; } p;
2  int *@aqual(UNIQUE) x = new 1;  // initializes x
3  p.x = x;            // consumes x
4  x = new 2;          // unconsumes x
5  p.y = x;            // consumes x
```

If an attempt was made to read through or copy `x` between lines 3 and 4 or after line 5, the flow analysis would reject the code, as in

```
int *@aqual(UNIQUE) x = new 1;  // initializes x
p.x = x;            // consumes x
p.y = x;            // rejected! x has been consumed already
```

When a multi-word data structure is assigned to another one, all of the unique pointers it contains are consumed. For example:

```
1  struct pair { int *@aqual(UNIQUE) x; int *@aqual(UNIQUE) y; } p, q,
2  p.x = new 1; p.y = new 2;
3  q = p;              // consumes p.x and p.y
```

By default, when a unique pointer is passed to a function, we expect that the function will not free the pointer, store it in a data structure, or otherwise make it unavailable to the caller. If a function violates any of thse assumptions its type must be augmented with the attribute `consume`, which indicates that a particular argument is no longer available to the caller after the call. For example:

```
void foo(int *@aqual(UNIQUE) x) __attribute__((consume(1))) {
  ufree(x);
}
void bar() {
  int *@aqual(UNIQUE) x = malloc(sizeof(int));
  foo(x);
  *x;//<--- this dereference is not allowed
}
```

Here, the `consume(1)` attribute in the definition of `foo` indicates that the first argument is consumed within the function body.

Note that if you fail to free a unique pointer, it will eventually be garbage collected.

Unique pointers have some restrictions. In particular:

- No pointer arithmetic is allowed on unique pointers. This ensures that all unique pointers point to the beginning of the object, so that the allocator is not confused when a pointer is passed to `ufree`.
- Take the address of a unique pointer is not allowed. This is because doing so effectively creates an alias to the original pointer that cannot be easily tracked statically.

- Unique pointers cannot be stored within datatypes (though they can be stored in tagged unions). This is a shortcoming of the current flow analysis.

9.3.2 Nested Unique Pointers

Directly reading a unique pointer is only allowed along a *unique path*. A unique path u has the form

$$u \texttt{ ::= } x \mid u\texttt{.m} \mid u\texttt{->m} \mid \texttt{*}u$$

where x is a local variable, and u is a unique pointer. To appreciate the unique-path restriction, consider this incorrect code:

```
int f(int *@aqual(UNIQUE) *‘r x) {
  int *@aqual(UNIQUE) *‘r y = x;  //x and y are aliases
  int *@aqual(UNIQUE) z = *y;
  ufree(z);
  return **x;  //accesses deallocated storage!
}
```

Here, `x` is a pointer into a conventional region `‘r` and thus its value can be freely copied to `y`. We then extract a unique pointer pointed to by `y` and free it. Then we attempt to access the deallocated storage through `x`.

If a unique pointer is not accessible via a unique path, it must be *swapped out* atomically to be used; in Cyclone this is expressed with syntax `:=:`. In particular, the code *a* `:=:` *b* will swap the contents of *a* and *b*. We can use this to swap out a nested unique pointer, and replace it with a different one; we will often swap in NULL, since this is a unique pointer that is always unconsumed. For example, in the code below, we define a queue type for queues that contain unique pointers, and a function `take` for removing the first element from the queue.

```
struct Queue<‘a,‘r> {
  list_t<‘a *@aqual(UNIQUE),‘r> front;
  list_t<‘a *@aqual(UNIQUE),‘r> rear;
};
typedef struct Queue<‘a,‘r> *‘r queue_t<‘a,‘r>;

‘a *@aqual(UNIQUE) take(queue_t<‘a> q) {
```

```
  if (q->front == NULL)
    throw &Empty_val;  // exception: def not shown
  else {
    let elem = NULL;
    elem :=: q->front->hd;
    q->front = q->front->tl;
    if (q->front == NULL) q->rear = NULL;
    return elem;
  }
}
```

Here, in order to extract the element stored in the queue (the `hd` portion of the underlying list), we need to use swap, because `q->front` is a non-unique pointer, and therefore `q->front->hd` is not a unique path.

Note that this code is not as polymorphic as it could be. In particular, the above queue definition requires its elements to be nullable unique pointers, when they could just as easily be non-unique pointers, or even reference-counted pointers (illustrated later), and the code for `take` would still work. This problem can be addressed, and its solution is described in Section 9.5.

9.3.3 Pattern Matching on Unique Pointers

As described in Section 5, Cyclone supports pattern matching on structured data with `let` declarations and `switch` statements. Unique pointers, or structures containing unique pointers, can be matched against, while still ensuring that only one legal pointer to a unique object exists at any given time.

In the simplest case, when a unique pointer to a structure is matched against, the matching operation is treated just like a dereference. Therefore, the pointer itself is *not* consumed. For example:

```
struct pair { int x; int y; };
void foo() {
  struct pair @@aqual(UNIQUE) p = new pair(1,2);
  let &pair{.x=x, .y=y} = p;
  ufree(p);
}
```

Here, we match against the unique pointer `p`'s two fields `x` and `y`. Because we don't make a copy of `p`, but rather only of its fields, `p` is not consumed. Therefore, `p` can be safely freed.

Because each of the fields matched against is assigned to the pattern variables, unique paths through the original pointer are consumed by virtue of being assigned. At the conclusion of the scope of the pattern, we can *unconsume* any location whose pattern variable has not been consumed or assigned to, as long as the parent pointer has not been consumed or assigned to. Here's an example:

```
struct Foo { int *@aqual(UNIQUE) x; int *@aqual(UNIQUE) y; };
void foo(struct Foo *@aqual(UNIQUE) p) {
  { let &Foo{.x=x, .y=y} = p; // consumes p->x and p->y
    ufree(x);                 // consumes x
  }                           // p->y is unconsumed
  ufree(p->y);                // p->y consumed
  ufree(p);                   // p consumed
}
```

The initial match against `p` consumes `p->x` and `p->y`, whose contents are copied to `x` and `y`, respectively. At the conclusion of the block, `p->y` is unconsumed because it did not change, whereas `p->x` is not, because `x` was freed within the block.

Note that the following code is illegal:

```
void foo(struct Foo *‘H p) {
 let &Foo{.x=x, .y=y} = p; // non-unique path!
 ...
}
```

To see why, notice that this is equivalent to

```
void foo(struct Foo *‘H p) {
 let x = p->x;
 let y = p->y;
 ...
}
```

This code is illegal because neither `p->x` nor `p->y` is a unique path. We also do not allow `*` patterns to create aliases of the original unique pointer,

for the same reason we forbid &e when e is a unique pointer. Unfortunately, this means we don't provide a way to assign to matched-against fields. However, in the case of the matched-against struct, we can just do this with regular paths. In the above example pattern block, we could do `p->y = new 1` or something like that (even within the scope of the pattern).

Matching against tagged unions is essentially like matching against structures, as just described. Since we do not allow unique pointers to be stored within datatypes, there is no change to how datatypes are matched.

9.4 Reference-counted Pointers

Cyclone also supports reference-counted pointers, which are treated quite similarly to unique pointers. Reference-counted objects may be allocated in any region. We define the constant `Core::refcnt_qual`, having type `aqual_t<REFCNT>`, for creating reference-counted pointers. The caveat here is that when you allocate something in this region, an extra word will be prepended to your data, which contains the reference count, initialized to 1.

As with unique pointers, no pointer arithmetic is allowed, for similar reasons: it can occlude where the "head" of the object is, and thus make it impossible to find the hidden reference count. The reference count can be accessed via the routine `Core::refptr_count`:

```
int refptr_count(‘a::TA ?@aqual(REFCNT) ptr);
```

The constant NULL is allowed to have type `‘a::A ?@aqual(REFCNT)`, and its reference count is always 0. Like unique pointers, implicit aliasing is not allowed. Aliases are created explicitly using the routine `Core::alias_refptr`:

```
‘a ?@aqual(REFCNT) alias_refptr(‘a::TA ?@aqual(REFCNT) ptr);
```

This routine returns a copy of its argument, which is itself not consumed. Furthermore, the reference count will be incremented by one. Reference counts are reduced explicitly by the routine `drop_refptr`:

```
void drop_refptr(‘a::TA ?@aqual(REFCNT) ptr)
                        __attribute((consume(1)));
```

In the case that the provided object's reference count is 1 (and is thus dropped to zero), the provided pointer is freed. The flow analysis will consume the passed pointer (as is always the case for function arguments), so you won't be able to use it afterwards. Just like unique pointers, you can "forget" reference-counted pointers without decrementing the count; this just means you'll never be able to free the pointer explicitly, but the GC will get it once it becomes unreachable.

Just like unique pointers, reference-counted pointers can be stored in normal, aliasable datastructures, and accessed using swap (e.g. `x :=: y`). Because NULL is a `'a::TA ?@aqual(REFCNT)` pointer, we can always cheaply construct a pointer to swap in.

A good example of the use of unique pointers and reference-counted pointers is in the Cyclone distribution's `tests` directory—the file `streambuff.cyc`. This is an implementation of a packet manipulation library with a representation for packets (called `streambuff_t`'s) that is similar to Linux's `skbuff_t`'s. It uses a combination of unique header structures and reference-counted data structures.

9.5 Qualifier Polymorphism

To allow the writing of reusable code we support both subtyping and bounded polymorphism over alias qualifiers. Type variables that range over the set of alias qualifiers are of kind `Q` are used in addition to the other kinds.

The list data structure in the Cyclone libraries illustrates many features of qualifier polymorphism. It has the following declaration:

```
struct List<'a::B,'r::R,'q::Q>{
  : RESTRICTED >= aquals('a), RESTRICTED >= 'q
  'a hd;
  struct List<'a,'r,'q> *@aqual('q) 'r tl;
};
typedef struct List<'a,'r,'q> *@aqual('q) 'r list_t<'a,'r,'q>;
```

Here, the structure is parameterized by three type variables. The first, of boxed-kind, admits instantiation by pointer types. Since pointer types may be qualified according to their aliasability, we require a bound on this

aliasability. For this purpose, we use the construct `aquals(‘a)`. Similar to the `regions(‘a)` construct, `aquals(‘a)` evaluates to the top-level alias qualifier of the type that instantiates the variable `‘a`. For instance,`aquals(int *@aqual(ALIASABLE) *@aqual(UNIQUE)) = UNIQUE`. The bound in the list declaration `RESTRICTED >= aquals(‘a)` states that `‘a` can be instantiated with a boxed kind with an aliasability that is a subtype of `RESTRICTED`. Since `RESTRICTED` is at the top of the subtyping hierarchy, this is the most general bound on the aliasability of the type.

The second type variable is of region kind. These types may not be qualified by aliasability and thus do not appear in the bounds at all.

Finally, we have the type variable `‘q::Q`, of alias qualifier kind. Thus it can be instantiated with any type that reduces to one of the alias qualifiers. That is, one of `ALIASABLE, UNIQUE, REFCNT, RESTRICTED` or `aquals(‘a)`. The bound on `‘q` also is the most general bound.

A list that uses unique pointers on the “spine” with reference counted elements might be constructed as follows:

```
int *rc_int = qnew(Core::refcnt_qual) 0;
int *rc_int2 = qnew(Core::refcnt_qual) 1;
list_t<int*@aqual(REFCNT),‘H,UNIQUE> l =
    rnew(Core::heap_region, Core::unique_qual)
        List{rc_int,
     rnew(Core::heap_region, Core::unique_qual)
        List{rc_int2, NULL}};
```

We can also quantify over alias qualifiers in function types. For instance, a function that copies a list can be defined as follows.

```
list_t<‘a,‘r,‘q> rqcopy(region_t<‘r> r,aqual_t<‘q> q,
                        list_t<‘a,‘r2,‘p> l
                        : RESTRICTED >= ‘q,
                          ALIASABLE >= aquals(‘a),
                          RESTRICTED >= ‘p) {
  if(l == NULL)
    return NULL;
  _ tl = NULL;
  tl :=: l->tl;
  list_t<‘a,‘r,‘q> result = rnew(r,q) List{l->hd, rqcopy(r,q,tl)};
```

```
  l->tl :=: tl;
  return result;
}
```

This function copies a list allocated in a region `r2` into a region `r`. As previously, the function is polymorphic in both the source and destination regions. Furthermore, the aliasability of the new list is specified by the handle `q` which has type `aqual_t<'q>`. As previously, since `'q` is of alias qualifier kind, a bound can be specified for it – this appears after the argument list along with any effects for this function. Note that the list that is being copied may also have a spine that is `RESTRICTED` as specified by the `RESTRICTED >= 'p` bound. This bound requires that we use the swap operator (`:=:`) to ensure that no aliases are manufactured. Finally, since we are copying the list, the elements of the list itself must be aliasable – this is specified by the `ALIASABLE >= aquals('a)` bound.

9.6 Aliasing Unique Pointers

Programmers often write code that aliases values temporarily, e.g. by storing them in loop iterator variables or by passing them to functions. Such reasonable uses would be severely hampered by "no alias" restrictions on unique pointers. To address this problem, we introduce a special `alias` pattern variable that permits temporary aliasing of a unique pointer. Here is a simple example:

```
char *@fat@aqual(UNIQUE) dst, *@fat@aqual(UNIQUE) src = ...
{ let alias <'r>char *@fat'r x = src; // consumes src
  memcpy(dst,x,numelts(x)); }
// src unconsumed
...
ufree(src);
```

In general, an alias pattern has form `alias <'r>`t `x`, where `'r` is a fresh region variable, and t is the type of `x`, which may mention `'r`. The `alias` pattern introduces a region `'r`, copies `src` to `x` which is treated as having the designated type `char *@fat'r`. Because `'r` is non-unique, `x` can be freely aliased. As such, we can pass `x` to the `memcpy` function. The matching operation consumes `src` during the block, and unconsumes it upon exit, so that `src` can be ultimately freed.

Alias pattern variables are similar to regular pattern variables. Like regular pattern variables, the matched-against expression (i.e. `src` in the above example) must be a unique path, and is consumed as a result of the match. As well, this expression can be unconsumed at the conclusion of the surrounding block as long as it hasn't been overwritten. However, in the case of regular pattern variables, unconsumption also requires that the pattern variable itself (i.e. `x` in the above example) hasn't changed within the block, while this requirement is unnecessary for alias patterns.

Intuitively, alias pattern variables are sound because we cast a unique pointer to instead point into a fresh region, for which there is no possibility of either creating new values or storing existing values into escaping data structures. As such we cannot create aliases that persist beyond the surrounding scope. However, we must take care when aliasing data having recursive type. For example, the following code is unsound:

```
void foo(list_t<‘a, ‘r1, UNIQUE> l) {
  let alias <‘r> x = (list_t<‘a, ‘r1, UNIQUE>)l;
  x->tl = x; // unsound: creates alias!
}
```

In this case, the `alias` effectively created many values in the fresh region `‘r`: one for each element of the list. This allows storing an alias in an element reachable from the original expression `l`, so that when the block is exited, this alias escapes.

For improved programmer convenience, the Cyclone typechecker optimistically inserts `alias` blocks around function-call arguments that are unique pointers when the formal-parameter type is polymorphic in the pointer's region. If this modified call does not type-check, we remove the inserted `alias`. For example, the `alias` pattern in the `foo` function above could be inferred, so we could instead write:

```
int foo() {
  list_t<int,‘H,UNIQUE> l = new List(1,new List(2,NULL));
  return length(l);
}
```

Right now, `alias` inference in Cyclone is fairly primitive, but could be extended to more contexts. We plan to improve this feature in future releases.

9.7 Dynamic Regions

Dynamic regions combine reference-counted or unique pointers and lexical regions together to essentially create reference-counted or unique *regions*; that is, the region is completely first class, and can be created or freed at conceptually any program point. This is done by representing a dynamic region as a unique (or reference-counted) pointer to an abstract struct `DynamicRegion` (which internally just contains the handle to a lexical region). The unique (or ref-counted) pointer is called the *key*. The key serves as a run-time *capability* that grants access to the region. At runtime, a key can be presented to a special `open` primitive, described later, that grants lexical access to the region.

The operation `new_ukey()` creates a fresh dynamic region and returns a unique key for the region; `new_rckey()` creates a fresh dynamic region and returns a ref-counted key for the region. The operations `free_ukey()` and `free_rckey()` are used to destroy unique and ref-counted keys respectively. The `free_ukey()` operation reclaims the key's region, as well as the storage for the key. The `free_rckey()` operation decrements the reference count, and if it's zero, reclaims the key's region as well as the storage for the key. Because ref-counted keys are pointers, you can use `alias_refptr` to make a copy of a ref-counted key. (Obviously, you can't make a copy of a unique key.) By the same token, you can pass a ref-counted key to `drop_refptr` (and you can pass a unique key to `ufree`), but doing so won't actually deallocate the region, but rather only the key. To obtain a dynamic *reap* (i.e. a dynamic region that supports the deletion of individual objects) use `new_reap_ukey` and `new_reap_rckey` instead.

Given a key k, a user can access the contents of its region by temporarily 'opening the region' within a lexical scope. This is done with the syntax `region` r `= open` k. That is, within the remainder of the current scope, the region handle r can be used to access k's region. The key k is temporarily consumed throughout the scope, and then unconsumed at its conclusion. This prevents you from opening up the dynamic region, and then freeing it while it's still in use. Note that `open` is very similar to `alias` in this way.

Here is a small example of the use of dynamic regions.

```
int main() {
  // Create a new dynamic region
```

```
  let NewDynamicRegion{<‘r> key} = new_ukey();

  // At this point, we refer to the region ‘r to
  // specify types, but we cannot actually access
  // ‘r (i.e. it’s not in our "static capability,"
  // a concept explained later)

  list_t<int,‘r> x = NULL;

  // We can access x by opening the region, which
  // temporarily consumes the key
  { region h = open(key);
    x = rnew(h) List(3,x);
  }

  // Now we can access the key again, but not x.
  // So we have to open the region to increment
  // its contents
  { region h = open(key);
    int i = x->hd + 1;
    x = rnew (h) List(i,x);
  }

  // Finally, destroy the key and the region
  free_ukey(key);
}
```

First, we allocate a new unique key and open it up, to reveal the name of the key's region (`‘r`), and the key itself. Because `‘r` is now in scope, we can declare a variable `x` that refers to it. However, because the key `key` must be opened before `‘r` becomes accessible, we cannot actually do anything with `x` yet (like dereference it).

Next, we open up the region using `key`, assigning its handle to the vairable `h`. Now, `key` is inaccessible (consumed) in the surrounding block, which prevents us from doing anything that might cause it to be freed while it's in use. We can use `h` to allocate into `‘r`, so we allocate a list element and store it in `x`.

At the conclusion of the `block`, the region `‘r` becomes inaccessible

again, so once again we cannot dereference `x`. However, `key` can now be accessed again, so we can open it again in the following block, to add a new list cell to `x`. At the conclusion of this block, `key` is unconsumed once again, so we legally call `free_ukey`. This frees the key and the region `'r`.

You can "share" a dynamic region key by placing it in some shared data structure, like a global variable. Of course, you'll then have to swap with NULL to get it in and out of the shared data structure, as the following code demonstrates:

```
struct MyContainer { <'r>
  uregion_key_t<'r> key;
  list_t<int,'r> data;
} *\U 'H global = NULL;

int main() {
  // allocate a dynamic region, and create a list
  let NewDynamicRegion{<'r> key} = new_ukey();
  list_t<int,'r> x = NULL;
  { region h = open(key);
    x = rnew(h) List(3,x);
  }

  // Stick the key and list in a global data
  // structure. We've now lost direct access to
  // the key and x.
  global = new MyContainer{key,x};

  // But we can regain it by swapping for the
  // container.
  struct MyContainer *@aqual(UNIQUE) p = NULL;
  global :=: p;

  // Now we can use it as above
  let MyContainer{<'r2> key2, data2} = *p;
  list_t<int,'r2> d = data2;
  { region h = open(key2);
    int i = d->hd + 1;
    d = rnew (h) List(i,d);
```

```
  }
}
```

Here, we define a global variable having type `MyContainer`, which consists of a key and some data into that key's region. The main function allocates a unique as before, and allocates some data into its region. Then we create a container for that key and data, and store it into the global variable; this consumes `key`, making it inaccessible, and effectively preventing access of `x` as well.

But we can then get the container back out of the global variable by swapping its contents with NULL. Then we can open up the container, and use the key and data as before. This way, a single dynamic region can be used by many different functions in the program. They simply swap out the global when they need it, operate on it, and then swap in the result.

One problem with using this technique with unique keys arises when you need to open the same region multiple times. The problem, of course, is that if you swap in NULL, then whoever tries to swap it out will fail. In other words, you can't really do recursive opens with UNIQUE keys. However, you can do this with REFCNT keys! Swap out the key, make a copy of it, swap it back in, and use the copy for the open (making sure to destroy the copy after the open).

One disadvantage of dynamic regions, which is inherited from unique and reference-counted pointers, is that if you put a key in some shared storage in a region 'r, then it is not the case that when 'r is deallocated that the key will be destroyed automatically. It's up to you to do the right thing or let the GC eventually collect it. In the long run, the right thing to do is add a finalizer interface for regions so that we can register a routine to deallocate a dynamic region whenever we put it in a shared data structure. The same goes for any unique pointer — we ought to have a way to register a finalizer. This is on our To-do list.

9.8 Defaults and Shorthands

As described so far, the notation for alias qualifiers is extremely verbose. The default qualifier annotations and bounds are intended to capture the most common cases and reduce the burden on the programmer. Where the

defaults do not suffice, shorthand versions allow explicit types to specified in a more compact manner.

The shorthand notation is as follows:

Alias Qualifier Specifier The strings `\A, \U, \RC, \T` can be used as substitutes for `@aqual(ALIASABLE)`, `@aqual(UNIQUE)`, `@aqual(REFCNT)`, and `@aqual(RESTRICTED)` respectively. For example

```
void cons(int *\T a, int *\U b)
```

is equivalent to

```
void cmp(int *@aqual(RESTRICTED) a, int *@aqual(UNIQUE) b)
```

Type Variable Bound The bound on a type variable, as shown previously, generally appears together with the effects in a function type, or with the outlives relations for an aggregate type. However, when the strings `\A, \U, \RC, \T` succeed a type variable they are interpreted as a bound on the type variable. This bound is only legal within an struct declaration or a function type; i.e. bounds may not appear within a local variable declaration, typedefs etc. If the type variable is of alias qualifier kind (kind Q) then this is interpreted as a qualifier bound. If the variable is of boxed, memory or abstract kind (::B, ::M, or ::A) then it is interpreted as an aquals bound. For instance, the function `rqcopy` above can be written as

```
list_t<‘a,‘r,‘q> rqcopy(region_t<‘r> r,aqual_t<‘q\T> q,
                        list_t<‘a\A,‘r2,‘p\T> l);
```

The same convention applies to aggregate types. For instance, we could define `List` as follows

```
struct List<‘a::B\T,‘r::R,‘q::Q\T>{
  ‘a hd;
  struct List<‘a,‘r,‘q> *@aqual(‘q) ‘r tl;
};
```

Note that if a type variable is used to specify the alias qualifier on a pointer, then the `@aqual(.)` notation must be used. That is, `int *@aqual(‘q\T)` cannot be substituted with `int *‘q\T`.

Heap Pointers For unique and reference counted pointers that reside in the heap we overload the region notation to provide a further shorthand. This is also for backward compatibility with previous versions of Cyclone where unique and reference counted pointers always resided in a separate region. Thus the type `T *@aqual(\U) ‘H` can be written more compactly as `T *‘U`; the type `T *@aqual(\RC) ‘H` can be written as `T *‘RC`.

The default qualifier bounds are as follows:

Function Types The aliasability of all parameters and return types in a function type is `ALIASABLE` by default. If a formal parameter is declared as `T *@aqual(\T)` then all subtypes of `T *@aqual(RESTRICTED)` may be passed as an argument. For parametricity `T *@aqual(‘q\T)` should be used instead.

Struct Types For pointers within a struct the default aliasability is `ALIASABLE`. For qualifier variables `‘q::Q` the default bound is `RESTRICTED`. The aquals bounds for type variables is `RESTRICTED` by default. Thus the following declarations are equivalent

```
struct PtrList<‘a::A, ‘q> {
  ‘a *hd;
  struct PtrList<‘a,‘q> *@aqual(‘q) tl;
};

struct PtrList<‘a::A\T,‘q\T> {
  ‘a *@aqual(\A) hd;
  struct PtrList<‘a,‘q> *@aqual(‘q) tl;
};
```

Type Instantiations When a type variable is instantiated with a pointer type we have to decide the aliasability of pointer. When instantiated in a function type or an aggregate type the aliasability is by default `ALIASABLE`. For instance, the following are equivalent:

```
void int_list(list_t<int*> l);
void int_list(list_t<int *@aqual(\A)> l);

struct Wrapper {
  list_t<int*> l;
}
struct Wrapper {
  list_t<int*@aqual(ALIASABLE)> l;
}
```

In variable declarations the default is `RESTRICTED` to allow for more aggressive unification. For instance

```
void int_list(list_t<int*@aqual(\T)> l) {
  list_t<int*> cp = l;
}
```

In the local variable declaration the bound defaults to `RESTRICTED` so that unification with the formal succeeds.

9.9 Subtyping for Effect Qualifier and Reaps

We mentioned in Section 9.2 that there was more to the type of `Core::rufree`. The full type of the function is given below.

```
void rufree(region_t<‘r> h, ‘a::A\T *\U ‘r ptr
            : single(‘r))  __attribute__((noliveunique(2)));
```

This type states that `rufree` expects a handle into region ‘r as its first argument. The ‘r effect qualifier on the second argument is meant to indicate that `ptr` is a pointer into the region ‘r, the handle of which is the first argument. The `\U` qualifier indicates that `ptr` must be a *unique* pointer into that region. The type of the element that is pointed to by `ptr` can be anything at all (`‘a::A`)‘, and may itself contain unique or reference-counted pointers, as indicated by the qualifier bound `\T`

The attribute `noliveunique` that appears in the type is a variant of the `consume` attribute shown earlier. This attribute states that after the call to the `rufree`, the pointer passed as the second parameter is consumed. Furthermore, if the element pointed to by `ptr` contains a live unique pointer, then Cyclone issues a warning. The reason is that since `ptr` is the only pointer to the underlying element, if this element contains a unique or reference-counted pointer, then the last reference to those pointers are lost leading to a memory leak.

The last component of the type of `rufree` is the `single(‘r)` constraint. The effect qualifier annotation of ‘r on both the handle as well as the pointer is meant to force the pointer to point into the region of the provided handle. However, in the presence of subtyping over effect qualifiers (as described in Section 8.4.4) the annotations on the parameters alone don't suffice to enforce this invariant. This is illustrated by the example below.

```
void bad_rufree(region_t<‘r>, ‘a *\U ‘r);
void foo() {
  region r;
  int *‘r ptr = rnew(r) 1;
  region_t<‘r+‘H> a = Core::heap_region;
  bad_rufree(a, ptr);
}
```

Subtyping of effect qualifiers means that it is permissble to treat `int *‘r` as `int*‘r+‘H` and so the call to `bad_rufree` typechecks even though pointer `ptr` does not point into the region for which `a` is the handle.

To avoid this behavior, the `single(‘r)` constraint is used to limit subtyping on the effect qualifier variable `‘r`. In particular, an effect qualifier variable that appears in a `single(‘r)` constraint is guaranteed to always be instantiated with a *single* region name, and not a set of region names.

9.10 Reap Allocator Implementation

To support deallocation of objects within a region the allocator must maintain some meta-data associated with each object. The current implementation uses a version of the bget allocator adapted for use with reaps. When

deallocation is rare, bget behaves much like a simple pointer bumping allocator and we expect performance to be competitive. However, the metadata does consume two additional header words for each object allocated.

For normal regions (i.e. those declared as `region r`, or those created using `new_ukey`, or `new_rckey`) the simple pointer-bumping allocator is used. Thus, if your application does not use reaps at all, then the overhead due to bget therefore does not apply.

10 Namespaces

As in C++, namespaces are used to avoid name clashes in code. For example:

```
namespace Foo {
  int x = 0;
  int f() { return x; }
}
```

declares an integer named `Foo::x` and a function named `Foo::f`. Note that within the namespace, you don't need to use the qualified name. For instance, `Foo::f` refers to `Foo::x` as simply `x`. We could also simply write "`namespace Foo;`" (note the trailing semi-colon) and leave out the enclosing braces. Every declaration (variables, functions, types, typedefs) following this namespace declaration would be placed in the Foo namespace.

As noted before, you can refer to elements of a namespace using the "`::`" notation. Alternatively, you can open up a namespace with a "`using`" declaration. For example, we could follow the above code with:

```
namespace Bar {
  using Foo {
    int g() { return f(); }
  }
  int h() { return Foo::f(); }
}
```

Here, we opened the `Foo` namespace within the definition of `Bar::g`. One can also write "`using Foo;`" to open a namespace for the remaining definitions in the current block.

Namespaces can nest as in C++.

Currently, namespaces are only supported at the top-level and you can't declare a qualified variable directly. Rather, you have to write a namespace declaration to encapsulate it. For example, you cannot write "`int Foo::x = 3;`."

The following subtle issues and **implementation bugs** may leave you scratching your head:

- The current implementation translates qualified Cyclone variables to C identifiers very naively: each `::` is translated to _ (underscore). This translation is wrong because it can introduce clashes that are not clashes in Cyclone, such as in the following:

  ```
  namespace Foo { int x = 7; }
  int Foo_x = 9;
  ```

  So avoid prefixing your identifiers with namespaces in your program. We intend to fix this bug in a future release.

- Because `#include` is defined as textual substitution, the following are usually very bad ideas: Having "`namespace Foo;`" or "`using Foo;`" at the top level of a header file. After all, you will be changing the identifiers produced or the identifiers available in every file that includes the header file. Having `#include` directives within the scope of namespace declarations. After all, you are changing the names of the identifiers in the header file by (further) qualifying them. Unfortunately, the current system uses the C pre-processor before looking at the code, so it cannot warn you of these probable errors.

  In short, you are advised to not use the "semicolon syntax" in header files and you are advised to put all `#include` directives at the top of files, before any namespace or using declarations.

- The translation of identifiers declared `extern "C"` is different. Given

  ```
  namespace Foo { extern "C" int x; }
  ```

  the Cyclone code refers to the global variable as `Foo::x`, but the translation to C will convert all uses to just `x`. The following code will therefore get compiled incorrectly (`f` will return 4):

```
namespace Foo { extern "C" int x; }
int f() {
  int x = 2;
  return x + Foo::x;
}
```

11 Varargs

C functions that take a variable number of arguments (vararg functions) are syntactically convenient for the caller, but C makes it very difficult to ensure safety. The callee has no fool-proof way to determine the number of arguments or even their types. Also, there is no type information for the compiler to use at call-sites to reject bad calls.

Cyclone provides three styles of vararg functions that provide different trade-offs for safety, efficiency, and convenience.

First, you can call C vararg functions just as you would in C:

```
extern "C" void foo(int x, ...);
void g() {
  foo(3, 7, "hi", 'x');
}
```

However, for the reasons described above, `foo` is almost surely unsafe. All the Cyclone compiler will do is ensure that the vararg arguments at the call site have some legal Cyclone type.

Actually, you can declare a Cyclone function to take C-style varargs, but Cyclone provides no way to access the vararg arguments for this style. That is why the example refers to a C function. (In the future, function subtyping could make this style less than completely silly for Cyclone functions.)

The second style is for a variable number of arguments of one type:

```
void foo(int x, ...string_t args);
void g() {
  foo(17, "hi", "mom");
}
```

The syntax is a type and identifer after the "`...`". (The identifier is optional in prototypes, as with other parameters.) You can use any identifier;

`args` is not special. At the call-site, Cyclone will ensure that each vararg has the correct type, in this case `string_t`.

Accessing the varargs is simpler than in C. Continuing our example, `args` has type `string_t *@fat ‘foo` in the body of `foo`. You retrieve the first argument (`"hi"`) with `args[0]`, the second argument (`"mom"`) with `args[1]`, and so on. Of course, `numelts(args)` tells you how many arguments there are.

This style is implemented as follows: At the call-site, the compiler generates a stack-allocated array with the array elements. It then passes a “fat pointer” to the callee with bounds indicating the number of elements in the array. Compared to C-style varargs, this style is less efficient because there is a bounds-check and an extra level of indirection for each vararg access. But we get safety and using vararg functions is just as convenient. No heap allocation occurs.

A useful example of this style is in the list library:

```
list_t<‘a> list(... ‘a argv) {
  list_t result = NULL;
  for (int i = numelts(argv) - 1; i >= 0; i--)
    result = new List{argv[i],result};
  return result;
}
```

Callers can now write `list(1,2,3,4,5)` and get a list of 5 elements.

The third style addresses the problem that it’s often desirable to have a function take a variable number of arguments of *different* types. For example, `printf` works this way. In Cyclone, we could use a `datatype` in conjunction with the second style. The callee then uses an array subscript to access a vararg and a switch statement to determine its `datatype` variant. But this would not be very convenient for the caller—it would have to explicitly “wrap” each vararg in the `datatype` type. The third style makes this wrapping implicit. For example, the type of printf in Cyclone is:

```
extern datatype PrintArg<‘r::R> {
  String_pa(const char ? *@notnull @nozeroterm‘r);
  Int_pa(unsigned long);
  Double_pa(double);
  LongDouble_pa(long double);
```

```
  ShortPtr_pa(short *@notnull ‘r);
  IntPtr_pa(unsigned long *@notnull ‘r);
};
typedef datatype PrintArg<‘r> *@notnull ‘r parg_t<‘r>;
int printf(const char *@fat fmt, ... inject parg_t);
```

The special syntax "`inject`" is the syntactic distinction for the third style. The type must be a `datatype` type. In the body of the vararg function, the array holding the vararg elements have this `datatype` type, with the function's region. (That is, the wrappers are stack-allocated just as the vararg array is.)

At the call-site, the compiler implicitly wraps each vararg by finding a `datatype` variant that has the expression's type and using it. The exact rules for finding the variant are as follows: Look in order for a variant that carries exactly the type of the expression. Use the first variant that matches. If none, make a second pass and find the first variant that carries a type to which the expression can be coerced. If none, it is a compile-time error.

In practice, the `datatype` types used for this style of vararg tend to be quite specialized and used only for vararg purposes.

Compared to the other styles, the third style is less efficient because the caller must wrap and the callee unwrap each argument. But everything is allocated on the stack and call sites do everything implicitly. A testament to the style's power is the library's implementation of printf and scanf entirely in Cyclone (except for the actual I/O system calls, of course).

12 Definite Assignment

It is unsafe to allow memory to be used as a value of a particular type just because the memory has been allocated at that type. In other words, you cannot use memory that has not been properly initialized. Most safe languages enforce this invariant by making allocation and initialization a single operation. This solution is undesirable in Cyclone for at least two reasons:

- Many idioms require declaring variables in a wider scope than is convenient for initializing the variable.

- C code, which we wish to port to Cyclone, is full of separated allocation and initialization, including all heap-allocated storage (i.e., `malloc`).

Inspired by Java's rules for separate declaration and initialization of local variables, Cyclone has a well-defined, sound system for checking that memory is written before it is used. The rules are more complicated than in Java because we support pointers to uninitialized memory, as is necessary for `malloc`, and because C's order-of-evaluation is not completely specified.

Here we begin with idioms that the analysis does and does not permit. With a basic sense of the idea, we expect programmers can generally not worry about the exact rules of the analysis. However, when the compiler rejects code because memory may be uninitialized, the programmer needs to know how to rewrite the code in order to pass the analysis. For this reason, we also give a more complete description of the rules.

We begin with examples not involving pointers. If you are familiar with Java's definite assignment, you can skip this part, but note that `struct` and tuple fields are tracked separately. So you can use an initialized field before another field of the same object is initialized. (Java does not allow separate allocation and initialization of object fields. Rather, it inserts null or 0 for you.)

Finally, we do allow uninitialized numeric values to be accessed. Doing so is dangerous and error-prone, but does not compromise type safety, so we allow it.

The following code is accepted:

```
extern int maybe();
int f() {
  int *x, *y, *z;
  if(maybe())
    x = new 3;
  else
    x = new 4;
  while(1) {
    y = x;
    break;
  }
  if(z = new maybe() && maybe() && q = new maybe())
```

```
    return q;
  else
    return z;
}
```

In short, the analysis checks that every control-flow path between a variable's declaration and use includes an assignment to the variable. More generally, the analysis works on memory locations, not just variables. The analysis knows that loop bodies and conditional branches are only executed if the value of certain expressions are 0 or not 0.

The following code is safe, but is not accepted:

```
extern int maybe();
int f() {
  int * x = new 1;
  int * y;
  int b = maybe();
  if(b)
    y = 2;
  if(b)
   return y;
 return 0;
}
```

The problem is that the analysis does not know that the second `if`-guard is true only if the first one is. General support for such "data correlation" would require reasoning about two different expressions at different times evaluating to the same value.

Unlike Java, Cyclone supports pointers to uninitialized memory. The following code is accepted:

```
extern int maybe();
int f() {
  int * x;
  int * z;
  int ** y;
  if(maybe()) {
    x = new 3;
    y = &x;
```

```
  } else {
    y = &z;
    z = new 3;
  }
  return *y;
}
```

The analysis does not know which branch of the if will be taken, so after the conditional it knows that either "`x` is initialized and `y` points to `x`" or "`z` is initialized and `y` points to `z`." It merges this information to "`y` points to somewhere initialized," so the function returns an initialized value, as required. (It is safe to return uninitialized ints, but we reject such programs anyway.)

However, this code is rejected even though it is safe:

```
extern int maybe();
int f() {
  int * x;
  int * z;
  int ** y;
  if(maybe()) {
    y = &x;
  } else {
    y = &z;
  }
  x = new 3;
  z = new 3;
  return *y;
}
```

The problem is that the analysis loses too much information after the conditional. Because `y` may allow (in fact, does allow) access to uninitialized memory and the analysis does not know exactly where `y` points, the conditional is rejected.

A compelling use of pointers to uninitialized memory is porting C code that uses `malloc`, such as the following (the cast is not necessary in Cyclone):

```
struct IntPair { int x; int y; };
```

```
struct IntPair * same(int z) {
  struct IntPair * ans =
        (struct IntPair *)malloc(sizeof(struct IntPair));
  ans->x = z;
  ans->y = z;
  return ans;
}
```

There is limited support for passing a pointer to uninitialized memory to a function that initializes it. See Section 13.

Certain expression forms require their arguments to be fully initialized (that is, everything reachable from the expression must be initialized) even though the memory is not all immediately used. These forms are the expression in "`let p = e`" and the argument to `switch`. We hope to relax these restrictions in the future.

You should now know enough to program effectively in Cyclone without immediately initializing all memory. For those wanting a more complete view of the language definition (i.e., what the analysis does and does not accept), we now go into the details. Note that the analysis is sound and well-specified—there is never a reason that the compiler rejects your program for unexplainable reasons.

For each local variable and for each program point that allocates memory, the analysis tracks information about each field. We call each such field a place. For example, in this code:

```
struct B { int * x; $(int*,int*) y;};
void f() {
  struct B b;
  struct B * bp = malloc(sizeof(B));
  ...
}
```

the places are `b.x`, `b.y[0]`, `b.y[1]`, `bp`, `<1>.x`, `<1>.y[0]`, and `<1>.y[1]` where we use `<1>` to stand for the `malloc` expression (a program point that does allocation). An initialization state can be "*must point to P*" where P is a path. For example, after the second declaration above, we have "`bp` must point to `<1>`." An ensuing assignment of the form "`bp->x = new 3`" would therefore change the initialization state of `<1>.x`. If there is not a unique path to which a place definitely points, then we keep track of the

place's *initialization level* and *escapedness*. A place is escaped if we do not know exactly all of the places that must point to it. For example, both of the following fragments would cause all the places starting with `<1>` to be escaped afterwards (assuming `bp` must point to `<1>`):

```
struct B * bp2;                 some_fun(bp);
if(maybe())
   bp2 = bp;
```

Note that if "p must point to P," then p is implicitly unescaped because we cannot know that p points to P if we don't know all the pointers to p. The initialization level is either None or All. All means p and everying reachable from p (following as many pointers as you want) is initialized.

Note that our choice of tracking "must point to" instead of "must alias" forces us to reject some safe programs, such as this one:

```
int f() {
  int * x, int *y;
  int **p1;
  if(maybe())
   p1 = &x;
  else
   p1 = &y;
  *p1 = new 7;
  return *p1;
}
```

Even though `p1` has not escaped, our analysis must give it initialization-level None. Moreover, `x` and `y` escape before they are initialized, so the conditional is rejected.

For safety reasons, once a place is escaped, any assignment to it must be a value that is fully initialized, meaning everything reachable from the value is initialized. This phenomenon is why the first function below is accepted but not the second (the `list_t` typedefs is defined in the `List` library):

```
list_t<'a,'H> copy(list_t<'a> x) {
  struct List *@notnull result, *@notnull prev;

  if (x == NULL) return NULL;
```

```
  result = new List{.hd=x->hd,.tl=NULL};
  prev = result;
  for (x=x->tl; x != NULL; x=x->tl) {
    struct List *@notnull temp = malloc(sizeof(struct List));
    temp->hd = x->hd;
    temp->tl = NULL;
    prev->tl = temp;
    prev = temp;
  }
  return result;
}

list_t<‘a,‘r2> rcopy(region_t<‘r2> r2, list_t<‘a> x) {
  struct List *@notnull result, *@notnull prev;

  if (x == NULL) return NULL;
  result = rnew(r2) List{.hd=x->hd,.tl=NULL};
  prev = result;
  for (x=x->tl; x != NULL; x=x->tl) {
    prev->tl = malloc(sizeof(struct List));
    prev->tl->hd = x->hd;
    prev->tl->tl = NULL;
    prev = prev->tl;
  }
  return result;
}
```

In the for body, we do not know where prev must point (on the first loop iteration it points to the first malloc site, but on ensuing iterations it points to the second). Hence `prev->tl` may be assigned only fully initialized objects.

13 Advanced Features

The features in this section are largely independent of the rest of Cyclone. It is probably safe to skip them when first learning the language, but it is

valuable to learn them at some point because they add significant expressiveness to the language.

13.1 Existential Types

The implementation of a `struct` type can have *existentially bound type variables* (as well as region variables, tag variables, and so on). Here is a useless example:

```
struct T { <'a> 'a f1; 'a f2; };
```

Values of type `struct T` have two fields with the same (boxed) type, but there is no way to determine what the type is. Different values can use different types. To create such a value, expressions of any appropriate type suffice:

```
struct T x = T{new 3, new 4};
```

Optionally, you can explicitly give the type being used for `'a`:

```
struct T x = T{<int*@notnull> new 3, new 4};
```

As with other lists of type variables, multiple existentially bound types should be comma-separated.

Once a value of an existential variant is created, there is no way to determine the types at which it was used. For example, `T("hi","mom")` and `T(8,3)` both have type `struct T`.

The only way to read fields of a `struct` with existentially bound type variables is pattern matching. That is, the field-projection operators (`.` and `->`) will *not* type-check. The pattern can give names to the correct number of type variables or have the type-checker generate names for omitted ones. Continuing our useless example, we can write:

```
void f(struct T t) {
  let T{<'b> x,y} = t;
  x = y;
}
```

We can now see why the example is useless; there is really nothing interesting that `f` can do with the fields of `t`. In other words, given `T("hi","mom")`,

no code will ever be able to use the strings "hi" or "mom". In any case, the scope of the type ‘b is the same as the scope of the variables x and y. There is one more restriction: For subtle reasons, you cannot use a reference pattern (such as *x) for a field of a struct that has existentially bound type variables.

Useful examples invariably use function pointers. For a realistic library, see fn.cyc in the distribution. Here is a smaller (and sillier) example; see the following two sections for an explanation of why the regions(‘a) <= ‘r stuff is necessary.

```
int f1(int x, int y) { return x+y; }
int f2(string x, int y) {printf("%s",x); return y; }
struct T<‘r::E> {<‘a> : regions(‘a) <= ‘r
  ‘a f1;
  int f(‘a, int);
};
void g(bool b) {
  struct T<‘H> t;
  if(b)
    t = Foo(37,f1);
  else
    t = Foo("hi",f2);
  let T{<‘b> arg,fun} = t;
  ‘b x = arg;
  int (*f)(‘b,int) = fun;
  f(arg,19);
}
```

We could replace the last three lines with fun(arg)—the compiler would figure out all the types for us. Similarly, the explicit types above are for sake of explanation; in practice, we tend to rely heavily on type inference when using these advanced typing constructs.

13.2 The Truth About Effects, Capabilities and Effect Subset Constraints

An *effect* or *capability* is a set of (compile-time) region names. We use *effect* to refer to the region names that must be "live" for some expression

to type-check and *capability* to refer to the region names that are "live" at some program point. A *effect subset constraint* indicates that all region names that appear in one effect qualifier also appear in another effect qualifier. Each program point has a set of "known effect subset relations".

The intuition is that a program point's capability and subset relations must imply that an expression's effect describes live regions, else the expression does not type-check. As we'll see, default effects for functions were carefully designed so that most Cyclone code runs no risk of such an "effect check" ever failing. But using existential types effectively requires a more complete understanding of the system, though perhaps not as complete as this section presents.

The form of effects or capabilities is described as follows:

- {} is the empty set. At most the heap region is accessed by an expression having this effect.
- `'r` is the set containing the indivisible effect `'r`. This effect variable can be isntantiated with a set consisting of one or more region names.
- `e1 + e2` is the set containing the effects e1 and e2. That is, we write + for set-union.
- `regions(t)`, where `t` is a type is the set containing all of the region names contained in `t` and `regions('a)` for all type variables `'a` contained in `t`.

The description of `regions(t)` appears circular, but in fact if we gave the definition for each form of types, it would not be. The point is that `regions('a)` is an "atomic effect" in the sense that it stands for a set of regions that cannot be further decomposed without knowing `'a`. The primary uses of `regions(t)` are descibed below.

The form of an effect subset constraint is `e1 <= e2` where `e1` and `e2` are both effects.

We now describe the capability at each program point. On function entry, the capability is the function's effect (typically the regions of the parameters and result, but fully described below). In a local block or a growable-region statement, the capability is the capability of the enclosing context plus the block/statement's region name.

The known effect subset relation at a program point are only those that are mentioned in the type of the function within which the program point occurs.

We can now describe an expression's effect: If it reads or writes to memory described by a region name `'r`, then the effect contains {`'r`}. If it calls a function with effect `e`, then the effect conatins `e`. Every function (type) has an effect, but programmers almost never write down an explicit effect. To do so, one puts "`; e`" at the end of the parameter list, wehre `e` is an effect. For example, we could write:

```
'a id('a x; {}) { return x; }
```

because the function does not access any memory.

If a function takes parameters of types `t1,...,tn` and returns a value of type `t`, the default effect is simply `regions(t1)+...+regions(tn)+regions(t)`. In English, the default assumption is that a function may dereference any pointers it is passed, so the corresponding regions must be live. In our example above, the default effect would have been `regions('a)`. If the caller had instantiated `'a` with `int*'r`, then with the default effect, the type-checker would require `'r` to be live, but with the explicit effect {} it would not. However, dangling pointers can be created only when using existential types, so the difference is rarely noticed.

By default, a function (type) has no effect subset constraints. That is, the function assumes no relationship between all the effect variables that appear in its type. Adding explicit subset relationships enables more subtyping in the callee and more stringent requirements at the call site (namely that the relationship holds).

We can describe when a capability `e` and a set of effect subset relations `s` imply an effect, although your intuition probably suffices. A "normalized effect" is either {} or unions of "atomic effects", where an atomic effect is either {`'r`} or `regions('a)`. For any effect `e1`, we can easily compute an equivalent normalized effect `e2`. Now, `e` and `s` imply `e1` if they imply every {`'r`} and `regions('a)` in `e2`. To imply {`'r`} (or `regions('a)`), we need {`'r`} (or `regions('a)`) to be in (a normalized effect of) `e`) or for `b` to contain some `'r <= 'r2` such that `e` and `b` imply `'r2`. Or something like that.

All of these complications are unnecessary except for existential types, to which we now return. Explicit bounds are usually necessary to make values of existential types usable. To see why, consider the example from

the previous section, with the `struct` declaration changed to remove the explicit bound:

```
struct T {<'a>
  'a f1;
  int f('a, int);
};
```

(So the declaration of `t` should just have type `struct T`.) Now the function call `f(arg,19)` at the end of `g` will not type-check because the (default) effect for `f` includes `regions('b)`, which cannot be established at the call site. But with the bound, we know that `regions('b) <='H`, which is sufficient to prove the call won't read through any dangling pointers.

13.3 Interprocedural Memory Initialization

We currently have limited support for functions that initialize parameters. if you have an *@notnulll1 parameter (pointing into any region), you can use an attribute __attribute__((initializes(1))) (where it's the first parameter, use a different number otherwise) to indicate that the function initializes through the parameter.

Obviously, this affects the definite-assignment analysis for the callee and the call-site. In the callee, we know the parameter is initialized, but not what it points to. The memory pointed to must be initialized before returning. Care must be taken to reject this code:

```
void f(int *@notnull*@notnull x) __attribute__((initializes(1))) {
  x = new (new 0);
  return x;
}
```

In the caller, the actual argument must point to a known location. Furthermore, this location must not be reachable from any other actual arguments, i.e., there must be no aliases available to the callee.

Two common idioms not yet supported are:

1. The parameter is initialized only if the return value satisfies some predicate; for example, it is 0.

2. The caller can pass NULL, meaning do not initialize through this parameter.

A Porting C code to Cyclone

Though Cyclone resembles and shares a lot with C, porting is not always straightforward. Furthermore, it's rare that you actually port an entire application to Cyclone. You may decide to leave certain libraries or modules in C and port the rest to Cyclone. In this chapter, we want to share with you the tips and tricks that we have developed for porting C code to Cyclone and interfacing Cyclone code against legacy C code.

A.1 Semi-Automatic Porting

The Cyclone compiler includes a simple porting mode which you can use to try to move your C code closer to Cyclone. The porting tool is not perfect, but it's a start and we hope to develop it more in the future.

When porting a file, say `foo.c`, you'll first need to copy the file to `foo.cyc` and then edit it to add `__cyclone_port_on__;` and `__cyclone_port_off__;` around the code that you want Cyclone to port. For example, if after copying `foo.c`, the file `foo.cyc` contains the following:

```
1.   #include <stdio.h>
2.
3.   void foo(char *s) {
4.     printf(s);
5.   }
6.
7.   int main(int argc, char **argv) {
8.     argv++;
9.     for (argc--; argc >= 0; argc--, argv++)
10.      foo(*argv);
11.  }
```

then you'll want to insert `__cyclone_port_on__;` at line 2 and `__cyclone_port_off__;` after line 11. You do not want to port standard include files such as `stdio`, hence the need for the delimiters.

Next compile the file with the `-port` flag:

```
cyclone -port foo.cyc > rewrites.txt
```

and pipe the output to a file, in this case `rewrites.txt`. If you edit the output file, you will see that the compiler has emitted a list of edits such as the following:

```
foo.cyc(5:14-5:15): insert '?' for '*'
foo.cyc(9:24-9:25): insert '?' for '*'
foo.cyc(9:25-9:26): insert '?' for '*'
```

You can apply these edits by running the `rewrite` program on the edits:

```
rewrite -port foo.cyc > rewrites.txt
```

(The `rewrite` program is written in Cyclone and included in the `tools` sub-directory.) This will produce a new file called `foo_new.cyc` which should look like this:

```
#include <stdio.h>

__cyclone_port_on__;

void foo(char ?s) {
  printf(s);
}

int main(int argc, char ??argv) {
  argv++;
  for (argc--; argc >= 0; argc--, argv++)
    foo(*argv);
}
__cyclone_port_off__;
```

Notice that the porting system has changed the pointers from thin pointers to fat pointers (`?`) to support the pointer arithmetic that is done in `main`, and that this constraint has flowed to procedures that are called (e.g., `foo`).

You'll need to strip out the port-on and port-off directives and then try to compile the file with the Cyclone compiler. In this case, the rewritten code in `foo_new.cyc` compiles with a warning that `main` might not return an integer value. In general, you'll find that the porting tool doesn't always produce valid Cyclone code. Usually, you'll have to go in and modify the code substantially to get it to compile. Nonetheless, the porting tool can take care of lots of little details for you.

A.2 Manually Translating C to Cyclone

To a first approximation, you can port a simple program from C to Cyclone by following these steps which are detailed below:

- Change pointer types to fat pointer types where necessary.
- Use comprehensions to heap-allocate arrays.
- Use tagged unions for unions with pointers.
- Initialize variables.
- Put breaks or fallthrus in switch cases.
- Replace one temporary with multiple temporaries.
- Connect argument and result pointers with the same region.
- Insert type information to direct the type-checker.
- Copy "const" code or values to make it non-const.
- Get rid of calls to free, calloc, etc.
- Use polymorphism or tagged unions to get rid of void*.
- Rewrite the bodies of vararg functions.
- Use exceptions instead of setjmp.

Even when you follow these suggestions, you'll still need to test and debug your code carefully. By far, the most common run-time errors you will get are uncaught exceptions for null-pointer dereference or array out-of-bounds. Under Linux, you should get a stack backtrace when you have an uncaught exception which will help narrow down where and why the exception occurred. On other architectures, you can use `gdb` to find the problem. The most effective way to do this is to set a breakpoint on the routines `_throw_null()` and `_throw_arraybounds()` which are defined in the runtime and used whenever a null-check or array-bounds-check fails. Then you can use `gdb`'s backtrace facility to see where the problem occurred. Of course, you'll be debugging at the C level, so you'll want to use the `-save-c` and `-g` options when compiling your code.

Change pointer types to fat pointer types where necessary. Ideally, you should examine the code and use thin pointers (e.g., `int*` or better `int*@notnull`) wherever possible as these require fewer run-time checks and less storage. However, recall that thin pointers do not support pointer arithmetic. In those situations, you'll need to use fat pointers (e.g., `int*@fat` which can also be written as `int?`). A particularly simple strategy when porting C code is to just change all pointers to fat pointers. The code is then more likely to compile, but will have greater overhead. After changing to use all fat pointers, you may wish to profile or reexamine your code and figure out where you can profitably use thin pointers.

Be careful with `char` pointers. By default, a `char ?` is treated as zero-terminated, i.e. a `char * @fat @zeroterm`. If you are using the `char` pointer as a buffer of bytes, then you may actually wish to change it to be a `char ? @nozeroterm` instead. Along these lines, you have to be careful that when you are using arrays that get promoted to pointers, that you correctly indicate the size of the array to account for the zero terminator. For example, say your original C code was

```
char line[MAXLINELEN];
while (fgets(line, MAXLINELEN, stdin)) ...
```

If you want your pointer to be zero-terminated, you would have to do the following:

```
char line[MAXLINELEN+1] @zeroterm;
while (fgets(line, MAXLINELEN, stdin)) ...
```

The `@zeroterm` qualifier is needed since `char` *arrays* are not zero-terminated by default. Adding the `+1` makes space for the extra zero terminator that Cyclone includes, ensuring that it won't be overwritten by `fgets`. If you don't do this, you could well get an array bounds exception at runtime. If you don't want your `char` array to be zero-terminated, you can simply leave the original C code as is.

Use comprehensions to heap-allocate arrays. Cyclone provides limited support for `malloc` and separated initialization but this really only works

for bits-only objects. To heap- or region-allocate and initialize an array that might contain pointers, use `new` or `rnew` in conjunction with array comprehensions. For example, to copy a vector of integers `s`, one might write:

```
int *@fat t = new {for i < numelts(s) : s[i]};
```

Use tagged unions for unions with pointers. Cyclone only lets you read members of unions that contain "bits" (i.e., ints; chars; shorts; floats; doubles; or tuples, structs, unions, or arrays of bits.) So if you have a C union with a pointer type in it, you'll have to code around it. One way is to simply use a `@tagged` union. Note that this adds hidden tag and associated checks to ensure safety.

Initialize variables. Top-level variables must be initialized in Cyclone, and in many situations, local variables must be initialized. Sometimes, this will force you to change the type of the variable so that you can construct an appropriate initial value. For instance, suppose you have the following declarations at top-level:

```
struct DICT;
struct DICT *@notnull new_dict();
struct DICT *@notnull d;
void init() {
  d = new_dict();
}
```

Here, we have an abstract type for dictionaries (`struct Dict`), a constructor function (`new_dict()`) which returns a pointer to a new dictionary, and a top-level variable (`d`) which is meant to hold a pointer to a dictionary. The `init` function ensures that `d` is initialized. However, Cyclone would complain that `d` is not initialized because `init` may not be called, or it may only be called after `d` is already used. Furthermore, the only way to initialize `d` is to call the constructor, and such an expression is not a valid top-level initializer. The solution is to declare `d` as a "possibly-null" pointer to a dictionary and initialize it with `NULL`:

```
struct DICT;
struct DICT *nonnull new_dict();
struct DICT *d;
void init() {
  d = new_dict();
}
```

Of course, now whenever you use `d`, either you or the compiler will have to check that it is not `NULL`.

Put breaks or fallthrus in switch cases. Cyclone requires that you either break, return, continue, throw an exception, or explicitly fallthru in each case of a switch.

Replace one temporary with multiple temporaries. Consider the following code:

```
void foo(char * x, char * y) {
  char * temp;
  temp = x;
  bar(temp);
  temp = y;
  bar(temp);
}
```

When compiled, Cyclone generates an error message like this:

```
type mismatch: char *@zeroterm #0  != char *@zeroterm #1
```

The problem is that Cyclone thinks that `x` and `y` might point into different regions (which it named `#0` and `#1` respectively), and the variable `temp` is assigned both the value of `x` and the value of `y`. Thus, there is no single region that we can say `temp` points into. The solution in this case is to use two different temporaries for the two different purposes:

```
void foo(char * x, char * y) {
  char * temp1;
```

```
  char * temp2;
  temp1 = x;
  bar(temp1);
  temp2 = y;
  bar(temp2);
}
```

Now Cyclone can figure out that `temp1` is a pointer into the region `#0` whereas `temp2` is a pointer into region `#1`.

Connect argument and result pointers with the same region. Remember that Cyclone assumes that pointer inputs to a function might point into distinct regions, and that output pointers, by default point into the heap. Obviously, this won't always be the case. Consider the following code:

```
int *foo(int *x, int *y, int b) {
  if (b)
    return x;
  else
    return y;
}
```

Cyclone complains when we compile this code:

```
foo.cyc:3: returns value of type int *`GR0 but requires int *
  `GR0 and `H are not compatible.
foo.cyc:5: returns value of type int *`GR1 but requires int *
  `GR1 and `H are not compatible.
```

The problem is that neither `x` nor `y` is a pointer into the heap. You can fix this problem by putting in explicit regions to connect the arguments and the result. For instance, we might write:

```
int *`r foo(int *`r x, int *`r y, int b) {
  if (b)
    return x;
  else
    return y;
}
```

and then the code will compile. Of course, any caller to this function must now ensure that the arguments are in the same region.

Insert type information to direct the type-checker. Cyclone is usually good about inferring types. But sometimes, it has too many options and picks the wrong type. A good example is the following:

```
void foo(int b) {
  printf("b is %s", b ? "true" : "false");
}
```

When compiled, Cyclone warns:

```
(2:39-2:40): implicit cast to shorter array
```

The problem is that the string `"true"` is assigned the type `const char ?{5}` whereas the string `"false"` is assigned the type `const char ?{6}`. (Remember that string constants have an implicit 0 at the end.) The type-checker needs to find a single type for both since we don't know whether `b` will come out true or false and conditional expressions require the same type for either case. There are at least two ways that the types of the strings can be promoted to a unifying type. One way is to promote both to `char?` which would be ideal. Unfortunately, Cyclone has chosen another way, and promoted the longer string (`"false"`) to a shorter string type, namely `const char ?{5}`. This makes the two types the same, but is not at all what we want, for when the procedure is called with false, the routine will print

```
b is fals
```

Fortunately, the warning indicates that there might be a problem. The solution in this case is to explicitly cast at least one of the two values to `const char ?`:

```
void foo(int b) {
  printf("b is %s", b ? ((const char ?)"true") : "false");
}
```

Alternatively, you can declare a temp with the right type and use it:

```
void foo(int b) {
  const char ? t = b ? "true" : "false"
  printf("b is %s", t);
}
```

The point is that by giving Cyclone more type information, you can get it to do the right sorts of promotions. Other sorts of type information you might provide include region annotations (as outlined above), pointer qualifiers, and casts.

Copy "const" code or values to make it non-const. Cyclone takes `const` seriously. C does not. Occasionally, this will bite you, but more often than not, it will save you from a core dump. For instance, the following code will seg fault on most machines:

```
void foo() {
  char ?x = "howdy"
  x[0] = 'a';
}
```

The problem is that the string `"howdy"` will be placed in the read-only text segment, and thus trying to write to it will cause a fault. Fortunately, Cyclone complains that you're trying to initialize a non-const variable with a const value so this problem doesn't occur in Cyclone. If you really want to initialize `x` with this value, then you'll need to copy the string, say using the `dup` function from the string library:

```
void foo() {
  char ?x = strdup("howdy");
  x[0] = 'a';
}
```

Now consider the following call to the `strtoul` code in the standard library:

```
extern unsigned long strtoul(const char ?‘r n,
                             const char ?‘r*‘r2 endptr,
                             int base);
unsigned long foo() {
  char ?x = strdup("howdy");
  char ?*e = NULL;
  return strtoul(x,e,0);
}
```

Here, the problem is that we're passing non-const values to the library function, even though it demands const values. Usually, that's okay, as `const char ?` is a super-type of `char ?`. But in this case, we're passing as the `endptr` a pointer to a `char ?`, and it is not the case that `const char ?*` is a super-type of `char ?*`. In this case, you have two options: Either make `x` and `e` const, or copy the code for `strtoul` and make a version that doesn't have const in the prototype.

Get rid of calls to free, calloc etc. There are many standard functions that Cyclone can't support and still maintain type-safety. An obvious one is `free()` which releases memory. Let the garbage collector free the object for you, or use region-allocation if you're scared of the collector. Other operations, such as `memset`, `memcpy`, and `realloc` are supported, but in a limited fashion in order to preserve type safety.

Use polymorphism or tagged unions to get rid of void*. Often you'll find C code that uses `void*` to simulate polymorphism. A typical example is something like swap:

```
void swap(void **x, void **y) {
  void *t = x;
  x = y;
  y = t;
}
```

In Cyclone, this code should type-check but you won't be able to use it in many cases. The reason is that while `void*` is a super-type of just about any pointer type, it's not the case that `void**` is a super-type of a pointer to a pointer type. In this case, the solution is to use Cyclone's polymorphism:

```
void swap(‘a *x, ‘a *y) {
  ‘a t = x;
  x = y;
  y = t;
}
```

Now the code can (safely) be called with any two (compatible) pointer types. This trick works well as long as you only need to “cast up” from a fixed type to an abstract one. It doesn’t work when you need to “cast down” again. For example, consider the following:

```
int foo(int x, void *y) {
  if (x)
   return *((int *)y);
  else {
    printf("%s\n",(char *)y);
    return -1;
  }
}
```

The coder intends for `y` to either be an int pointer or a string, depending upon the value of `x`. If `x` is true, then `y` is supposed to be an int pointer, and otherwise, it’s supposed to be a string. In either case, you have to put in a cast from `void*` to the appropriate type, and obviously, there’s nothing preventing someone from passing in bogus cominations of `x` and `y`. The solution in Cylcone is to use a tagged union to represent the dependency and get rid of the variable `x`:

```
@tagged union IntOrString {
  int Int;
  const char *@fat String;
};
typedef union IntOrString i_or_s;
int foo(i_or_s y) {
  switch (y) {
  case {.Int = i}:  return i;
  case {.String = s}:
```

```
      printf("%s\n",s);
      return -1;
    }
  }
```

Rewrite the bodies of vararg functions. See the section on varargs for more details.

Use exceptions instead of setjmp. Many uses of `setjmp/longjmp` can be replaced with a `try`-block and a `throw`. Of course, you can't do this for things like a user-level threads package, but rather, only for those situations where you're trying to "pop-out" of a deeply nested set of function calls.

A.3 Interfacing to C

When porting any large code from C to Cyclone, or even when writing a Cyclone program from scratch, you'll want to be able to access legacy libraries. To do so, you must understand how Cyclone represents data structures, how it compiles certain features, and how to write wrappers to make up for representation mismatches.

A.3.1 Extern "C"

Sometimes, interfacing to C code is as simple as writing an appropriate interface. For instance, if you want to call the `acos` function which is defined in the C Math library, you can simply write the following:

```
extern "C" double acos(double);
```

The `extern "C"` scope declares that the function is defined externally by C code. As such, it's name is not prefixed with any namespace information by the compiler. Note that you can still embed the function within a Cyclone namespace, it's just that the namespace is ignored by the time you get down to C code. If you have a whole group of functions then you can wrap them with a single `extern "C" { ... }`, as in:

```
extern "C" {
  double acos(double);
```

```
    float  acosf(float);
    double acosh(double);
    float  acoshf(float);
    double asin(double);
  }
```

You must be careful that the type you declare for the C function is its real type. Misdeclaring the type could result in a runtime error. Note that you can add Cyclonisms to the type that refine the meaning of the original C. For example, you could declare:

```
  extern "C" int strlen(const char * @notnull str);
```

Here we have refined the type of `strlen` to require that a non-NULL pointer is passed to it. Because this type is representation-compatible with the C type (that is, it has the same storage requirements and semantics), this is legal. However, the following would be incorrect:

```
  extern "C" int strlen(const char * @fat str);
```

Giving the function this type would probably lead to an error because Cyclone fat pointers are represented as three words, but the standard C library function expects a single pointer (one word).

The `extern "C"` approach works well enough that it covers many of the cases that you'll encounter. However, the situation is not so when you run into more complicated interfaces. Sometimes you will need to write some wrapper code to convert from Cyclone's representations to C's and back (so called *wrapper* code).

A.3.2 Extern "C include"

Another useful tool is the `extern "C include"` mechanism. It allows you to write C definitions within a Cyclone file. Here is a simple example:

```
extern "C include" {
  char peek(unsigned int i) {
    return *((char *)i);
  }

  void poke(unsigned int i, char c) {
```

```
    *((char *)i) = c;
  }
} export {
  peek, poke;
}
```

In this example, we've defined two C functions `peek` and `poke`. Cyclone will not compile or type-check their code, but rather pass them on to the C compiler. The `export` clause indicates which function and variable definitions should be exported to the Cyclone code. If we only wanted to export the `peek` function, then we would leave the `poke` function out of the export list. All all other definitions, like `typedef`s, `struct`s, etc., not to mention `#define`s and other preprocessor effects, are exported by default (but this may change in a later release).

Any top-level types you mention in the `extern "C include"` are interpreted by the Cyclone code that uses them as Cyclone types. If they are actually C types (as would be the case if you `#include`d some header in the C block), this will be safe, but possibly undesirable, since they may not communicate the right information to the Cyclone code. There are two ways around this. In many cases, you can actually declare Cyclone types within the C code, and they will be treated as such. For example, in `lib/core.cyc`, we have For example, you could do something like:

```
extern "C include" {
  ... Cyc_Core_mkthin(‘a ?‘r dyn, sizeof_t<‘a> sz) {
    unsigned bd = _get_dyneither_size(dyn,sz);
    return Cyc_Core_mktuple(dyn.curr,bd);
  }
} export {
  Cyc_Core_mkthin
}
```

In this case, we are able to include a `?` notation directly in the C type, but then manipulate it using the runtime system functions for fat pointers (see `cyc_include.h` for details).

In the case that you are `#include`ing a C header file, you may not be able to change its definitions to have a proper Cyclone type, or it may be that the Cyclone definitions will not parse for some reason. In this case, you can declare a block to *override* the definitions with Cyclone compatible versions. For example, we could change the above code to be instead:

```
extern "C include" {
  struct foo { int x; int y; };
  struct foo *cast_charbuf(char *buf, unsigned int n) {
    if (n >= sizeof(struct foo))
      return (struct foo *)buf;
    else
      return (void *)0;
  }
} cyclone_override {
  struct foo *cast_charbuf
    (char * @numelts(valueof(‘n)) @nozeroterm buf,tag_t<‘n> n);
} export {
  cast_charbuf
}
```

Now we have given `cast_charbuf` its original C type, but then provided the Cyclone type in the override block. The Cyclone type ensures the value of `n` correctly represents the length of the buffer, by using Cyclone's dependent types (see Section 3). Note that top-level `struct` and other type definitions can basically be entirely Cyclone syntax. If you try to declare a Cyclone overriding type that is representation-incompatible with the C version, the compiler will complain.

Here is a another example using an external header:

```
extern "C include" {  /* tell Cyclone that <pcre.h> is C code */
#include <pcre/pcre.h>
} cyclone_override {
  pcre *‘U pcre_compile(const char @pattern, int options,
                        const char *‘H *errptr, int *erroffset,
                        const unsigned char *tableptr);
  int pcre_exec(const pcre @code, const pcre_extra *extra,
                const char *subject, int length,
                int startoffset, int options,
                int *ovector, int ovecsize);
} export { pcre_compile, pcre_exec; }
```

In this case, we have included the Perl regular expression library C header, and then exported two of its functions, `pcre_compile` and `pcre_exec`.

Moreover, we have given these functions Cyclone types that are more expressive in the original C. Probably we would yet want to write wrappers around these functions to check other invariants of the arguments (e.g., that the `length` passed to `pcre_exec` is indeed the length of the `subject`). Take a look at `tests/pcredemo.cyc` for more information on this example. Another example that shows how you can override things is in `tests/cinclude.cyc`.

The goal of this example is to show how you can safely suck in a large C interface (in this case, the Perl Compatible Regular Expression interface), write wrappers around some of the functions to convert represenations and check properties, and then safely export these wrappers to Cyclone.

One word of warning: when you `#include` something within an `extern "C include"`, it will follow the normal include path, which is to say that it will look for *Cyclone* versions of the headers first. This means that if you do something like:

```
extern "C include" {
#include <string.h>
} export { ... }
```

It will actually include the Cyclone version of the `string.h` header! These easiest way around this is to use an absolute path, as in

```
extern "C include" {
#include "/usr/include/string.h"
} export { ... }
```

Even worse is when a C header you wish to include itself includes a header for which there exists a Cyclone version. In the `pcre.h` example above, this actually occurs in that `pcre.h` includes `stdlib.h`, and gets the Cyclone version. To avoid this, the `pcredemo.cyc` program includes the Cyclone versions of these headers first. Ultimately we will probably change the compiler so that header processing within `extern "C include"` searches the C header path but not the Cyclone one.

B Frequently Asked Questions

What does `$(`$type_1$`,`$type_2$`)` mean? What does `$(`$expr_1$`,` $expr_2$`)` mean? Cyclone has *tuples*, which are anonymous structs with fields numbered 0, 1,

2, ... For example, `$(int,string_t)` is a pair of an `int` and a `string_t`. An example value of this type is `$(4,"cyclone")`. To extract a field from a tuple, you use array-like notation: you write `x[0]`, not `x.0`.

What does `int @` mean? In Cyclone `@` is a pointer that is guaranteed not to be `NULL`. The Cyclone compiler guarantees this through static or dynamic checks. For example,

```
int *x = NULL;
```

is not an error, but

```
int @x = NULL;
```

is an error. Note that "`int @`" is shorthand for the more verbose "`int *@notnull`".

What does `int *{37}` mean? This is the type of (possibly-null) pointers to a sequence of at least 37 integers, which can also be written as "`int *@numelts(37)`". The extra length information is used by Cyclone to prevent buffer overflows. For example, Cyclone will compile `x[`*`expr`*`]` into code that will evaluate *expr*, and check that the result is less than 37 before accessing the element. Note that `int *` is just shorthand for `int *{1}`. Currently, the expression in the braces must be a compile-time constant.

What does `int *'r` mean? This is the type of a pointer to an `int` in region `'r`. This can also be written as "`int *@region('r)`". A region indicates conceptually where in memory an object is stored; different regions have different lifetimes and deallocation strategies, and the aliasing into certain regions may be restricted. Cyclone uses this region information to prevent dereferencing a pointer whose storage has been deallocated. See Section 8 for more information on regions.

What does `'H` mean? This is Cyclone's heap region: objects in this region cannot be explicitly freed, only garbage-collected. Effectively, this means that pointers into the heap region can always be safely dereferenced.

What does `int @{37}'r` mean? A pointer can come with all or none of the nullity, bound, and region annotation. This type is the type of non-`null` pointers to at least 37 consecutive integers in region `'r`. When the bound is omitted it default to 1.

What is a pointer type's region when it's omitted? Every pointer type has a region; if you omit it, the compiler chooses a region name you implicitly. The name chosen depends on where the pointer type occurs. In function arguments, a fresh region variable is used. In function results and type definitions (including `typedef`), the heap region (`'H`) is used. In function bodies, the compiler looks at the uses (using unification) to try to determine a region. See Section 2.3 and Section 8 for more information.

What does `int ?` mean? The `?` a special kind of pointer that carries along bounds information. It is a "questionable" pointer: it might be NULL or pointing out of bounds. An `int ?` is a pointer to an integer, along with some information that allows Cyclone to check whether the pointer is in bounds at run-time. These are the only kinds of pointers that you can use for pointer arithmetic in Cyclone.

What does `'a` mean? `'a` is a *type variable*. Type variables are typically used in polymorphic functions. For example, if a function takes a parameter of type `'a`, then the function can be called with a value of *any* suitable type. If there are two arguments of type `'a`, then any call will have to give values of the same type for those parameters. And if the function returns a type `'a`, then it must return a result of the same type as the the argument. Syntactically, a type variable is any identifier beginning with `'` (backquote).

What is a "suitable" type for a type variable? The last question said that a type variable can stand for a "suitable" type. Unfortunately, not all types are "suitable." Briefly, the "suitable" types are those that fit into a general-purpose machine register, typically including `int`, and pointers. Non-suitable types include `float`, `struct` types (which can be of arbitrary size), tuples, and questionable pointers. Technically, the suitable types are the types of "box kind," described below.

How do I cast from `void *`? You can't do this in Cyclone. A `void *` in C really does not point to `void`, it points to a value of some type. However, when you cast from a `void *` in C, there is no guarantee that the pointer actually points to a value of the expected type. This can lead to crashes, so Cyclone doesn't permit it. Cyclone's polymorphism and tagged unions can often be used in places where C needs to use `void *`, and they are safe. Note that you can generally cast to a `void *` in Cyclone, you just won't be able to cast back.

What does _ (underscore) mean in types? Underscore is a "wildcard" type. It stands for some type that the programmer doesn't want to bother writing out; the compiler is expected to fill in the type for the programmer. Sometimes, the compiler isn't smart enough to figure out the type (you will get an error message if so), but usually there is enough contextual information for the compiler to succeed. For example, if you write

```
_ x = new Pair(3,4);
```

the compiler can easily infer that the wildcard stands for `struct Pair @`. In fact, if `x` is later assigned `NULL`, the compiler will infer that `x` has type `struct Pair *` instead.

Note that only in restricted cases is _ allowed as part of top-level declarations.

What do `'a::B`, `'a::M`, `'a::A`, `'a::R`, `'a::Q` and `'a::E` mean? Types are divided into different groups, which we call *kinds*. There are six kinds: B (for Box), M (for Memory), A (for Any), E (for Effect), R (for Region), and Q (for alias-Qualifier). The notation *typevar*`::`*kind* says that a type variable belongs to a kind. A type variable can only be instantiated by types that belong to its kind.

Box types include `int`, `long`, `region_t`, `tag_t`, enums, and non-`@fat` pointers. Memory types include all box types, `void`, `char`, `short`, `long long`, `float`, `double`, arrays, tuples, `datatype` and `@extensible datatype` variants, `@fat` pointers, and non-abstract structs and unions. Any types include all types that don't have kind R, E or Q. For the region types, R indicates regions like the heap,

stack, and dynamic regions; Q indicates alias qualifiers like ALIASABLE, UNIQUE, REFCNT or RESTRICTED; Effect types are sets of regions (these are explained elsewhere).

What does it mean when type variables don't have explicit kinds? Every type variable has a kind, but usually the programmer doesn't have to write it down. In function prototypes, the compiler will infer the most permissive kind. For example,

```
void f(‘a *‘b x, ‘c * y, ‘a z);
```

is shorthand for

```
void f(‘a::B *‘b::R x, ‘c::M * y, ‘a::B z)
```

In type definitions, no inference is performed: an omitted kind is shorthand for `::B`. For example,

```
struct S<‘a,‘r::R> { ‘a *‘r x; };
```

is shorthand for

```
struct S<‘a::B,‘r::R> { ‘a *‘r x;};
```

but

```
struct S<‘a,‘r>{‘a *‘r x;};
```

is not.

What does `struct List<‘a,‘r::R>` mean? `struct List` takes a type of box kind and a region and produces a type. For example, `struct List<int, ‘H>` is a type, and `struct List<struct List<int,‘H>@, ‘H>` is a type. `struct List<‘a,‘r::R>` is a list whose elements all have type `‘a` and live in region `‘r`.

What is a `@tagged` union? In C, when a value has a `union` type, you know that in fact it has one of the types of the union's fields, but there is no guarantee which one. This can lead to crashes in C. Cyclone's @tagged unions are like C unions with some additional information (a tag) that lets the Cyclone compiler determine what type the underlying value actually has, thus helping to ensure safety.

What is `abstract`? `abstract` is a storage-class specifier, like `static` or `extern`. When attached to a top-level type declaration, it means that other files can use the type but cannot look at the internals of the type (e.g., other files cannot access the fields of an abstract struct). Otherwise, abstract has the same meaning as the `auto` (default) storage class. Hence `abstract` is a way to state within a Cyclone file that a type's representation cannot be exported.

What are the Cyclone keywords? In addition to the C keywords, the following have special meaning and cannot be used as identifiers: `abstract`, `catch`, `datatype`, `fallthru`, `let`, `malloc`, `namespace`, `new`, `NULL`, `region_t`, `regions`, `rmalloc`, `rnew`, `throw`, `try`, `using`. As in gcc, `__attribute__` is reserved as well.

What are `namespace` and `using`? These constructs provide a convenient way to help avoid name clashes. namespace X prepends X:: to the declarations in its body (rest of file in case of namespace X;) and using X makes the identifiers prepended with X:: available without having to write the X::.

What is `fallthru`? In Cyclone, you cannot implicitly fall through from one switch case to the next (a common source of bugs in C). Instead, you must explicitly fall through with a `fallthru` statement. So, to port C code, place `fallthru;` at the end of each case that implicitly falls through; note that `fallthru` may not appear in the last case of a `switch`.

`fallthru` is useful for more than just catching bugs. For instance, it can appear anywhere in a case; its meaning is to immediately goto the next case. Second, when the next case of the `switch` has pattern variables, a `fallthru` can (and must) be used to specify expressions that will be bound to those variables in the next case. Hence

`fallthru` is more powerful (but more verbose) than "or patterns" in ML.

What is `new`? `new` *expr* allocates space in the heap region, initializes it with the result of evaluating `expr`, and returns a pointer to the space. It is roughly equivalent to

```
type @temp = malloc(sizeof(type));
*temp = expr;
```

where `type` is the type of *expr*. You can also write

```
new { for i < expr1 : expr2 }
```

to heap-allocate an array of size $expr_1$ with the $\texttt{i}^{\text{th}}$ element initialized to $expr_2$ (which may mention `i`).

How do I use tuples? A tuple type is written $\texttt{\$(}type_0\texttt{, ..., }type_n\texttt{)}$. A value of the type is constructed by $\texttt{\$(}expr_0\texttt{, ..., }expr_n\texttt{)}$, where $expr_i$ has type $type_i$. If *expr* has type $\texttt{\$(}type_0\texttt{, ..., }type_n\texttt{)}$, you can extract the component `i` using *expr*`[i]`. The expression in the brackets must be a compile-time constant. In short, tuples are like anonymous structs where you use *expr*`[i]` to extract fields instead of *expr*`.i`. There is no analogue of the `->` syntax that can be used with pointers of structs; if *expr* has type $\texttt{\$(}type_1\texttt{, ..., }type_n\texttt{)*}$, you can extract component `i` by `(*`*expr*`)[i]`.

What is `{for i <` $expr_1$ `:` $expr_2$`}`? This is an array initializer. It can appear where array initializers appear in C, and it can appear as the argument to `new`. It declares an identifier (in this case, `i`) whose scope is $expr_2$. $expr_1$ is an expression which is evaluated to an unsigned integer giving the desired size of the array. The expression $expr_2$ is evaluated $expr_1$ times, with `i` ranging over 0, 1, ..., $expr_1$-1; the result of each evaluation initializes the $\texttt{i}^{\text{th}}$ element of the array.

The form `new {for i <` $expr_1$ `:` $expr_2$`}` allocates space for a new array and initializes it as just described. This form is the only way to create arrays whose size depends on run-time information. When `{for i <` $expr_1$ `:` $expr_2$`}` is not an argument to `new`, $expr_1$ must be constant and $expr_2$ may not mention `i`. This restriction includes all uses at top-level (for global variables).

How do I throw and catch exceptions? A new exception is declared as in

```
datatype exn { MyExn };
```

The exception can be thrown with the statement

```
throw MyExn;
```

You can catch the expression with a `try/catch` statement:

```
try statement1 catch { case MyExn: statement2 }
```

If *statement*$_1$ throws an `MyExn` and no inner `catch` handles it, control transfers to *statement*$_2$.

The `catch` body can have any number of `case` clauses. If none match, the exception is re-thrown.

Exceptions can carry values with them. For example, here's how to declare an exception that carries an integer:

```
datatype exn { MyIntExn(int) };
```

Values of such exceptions must be heap-allocated. For example, you can create and throw a `MyIntExn` exception with

```
throw new MyIntExn(42);
```

To catch such an exception you must use an &-pattern:

```
try statement1
catch {
  case &MyIntExn(x): statement2
}
```

When the exception is caught, the integer value is bound to `x`.

The `exn` type is just a pre-defined `@extensible datatype` type. Therefore, all the standard rules for extending, creating objects, and destructing objects of a datatype apply.

How efficient is exception handling? Entering a `try` block is implemented using `setjmp`. Throwing an exception is implemented with `longjmp`. Pattern-matching a `datatype` against each case variant in the catch clause is a pointer-comparsion. In short, exception handling is fairly lightweight.

What does `let` mean? In Cyclone, `let` is used to declare variables. For example,

```
let x,y,z;
```

declares the three variables `x`, `y`, and `z`. The types of the variables do not need to be filled in by the programmer, they are filled in by the compiler's type inference algorithm. The `let` declaration above is equivalent to

```
_ x;
_ y;
_ z;
```

There is a second kind of `let` declaration, with form

```
let pattern = expr;
```

It evaluates *expr* and matches it against *pattern*, initializing the pattern variables of *pattern* with values drawn from `expr`. For example,

```
let x = 3;
```

declares a new variable `x` and initializes it to 3, and

```
let $(y,z) = $(3,4);
```

declares new variables `y` and `z`, and initializes `y` to 3 and `z` to 4.

What is a pattern and how do I use it? Cyclone's patterns are a convenient way to destructure aggregate objects, such as structs and tuples. They are also the only way to destructure datatypes. Patterns are used in Cyclone's `let` declarations, `switch` statements, and `try/catch` statements.

What does _ mean in a pattern? It is a wildcard pattern, matching any value. For example, if `f` is a function that returns a pair, then

```
let $(_,y) = f(5);
```

is a way to extract the second element of the pair and bind it to a new variable `y`.

What does it mean when a function has an argument with type `'a`? Any type that looks like `'` (backquote) followed (without whitespace) by an identifier is a type variable. If a function parameter has a type variable for its type, it means the function can be called with any pointer or with an int. However, if two parameters have the same type variable, they must be instantiated with the same type. If all occurrences of `'a` appear directly under pointers (e.g., `'a *`), then an actual parameter can have any type, but the restrictions about using the same type still apply. This is called *parametric polymorphism*, and it's used in Cyclone as a safe alternative to casts and `void *`.

Do functions with type variables get duplicated like C++ template functions?Is there run-tir
No and no. Each Cyclone function gives rise to one function in the output, and types are not present at run-time. When a function is called, it does not need to know the types with which the caller is instantiating the type variables, so no instantiation actually occurs—the types are not present at run-time. We do not have to duplicate the code because we either know the size of the type or the size does not matter. This is why we don't allow type variables of memory kind as parameters—doing so would require code duplication or run-time types.

Can I use varargs? Yes, Cyclone has a way of supporting variable-argument functions. It is not quite the same as C's, but it is safe. For instance, we have written type-safe versions of printf and scanf all within Cyclone. See the documentation on varargs for more information.

Why can't I declare types within functions? We just haven't implemented this support yet. For now, you need to hoist type declarations and typedefs to the top-level.

What casts are allowed? Cyclone doesn't support all of the casts that C does, because incorrect casts can lead to crashes. Instead, Cyclone supports a safe subset of C's casts. Here are some examples.

All of C's numeric casts, conversions, and promotions are unchanged.

You can always cast between *type*`@{`*const-expr*`}`, *type*`*{`*const-expr*`}`, and *type*`?`. A cast from *type*`?` to one of the other types includes a run-time check that the pointer points to a sequence of at least *const-expr* objects. A cast to *type*`@{`*const-expr*`}`from one of the other types includes a run-time check that the pointer is not `NULL`. No other casts between these type have run-time checks. A failed run-time check throws `Null_Exception`. A pointer into the heap can be cast to a pointer into another region. A pointer to a `struct` or `tuple` can be cast to a pointer to another `struct` or `tuple` provided the "target type" is *narrower* (it has fewer fields after "flattening out" nested `structs` and `tuples`) and each (flattened out) field of the target type could be the target of a cast from the corresponding field of the source type. A pointer can be cast to `int`. The type *type*`*{`*const-expr*$_1$`}`can be cast to *type*`*{`*const-expr*$_2$`}`provided *const-expr*$_2$ $<$ *const-expr*$_1$, and similarly for *type*`@{`*const-expr*$_1$`}`and *type*`@{`*const-expr*$_2$`}`.

An object of type `datatype T.A @` can be cast to `datatype T @`. The current implementation isn't quite as lenient as it should be. For example, it rejects a cast from `int *{4}` to `$(int,int)*{2}`, but this cast is safe.

For all non-pointer-containing types *type*, you can cast from a *type* `?` to a `char ?`. This allows you to make frequent use of `memcpy`, `memset`, *etc.*

Why can't I implicitly fall-through to the next `switch` case? We wanted to add an explicit `fallthru` construct in conjunction with pattern matching, and we decided to enforce use of `fallthru` in all cases because this is a constant source of bugs in C code.

Do I have to initialize global variables? You currently must provide explicit initializers for global variables that may contain pointers, so that the compiler can be sure that uninitialized memory containing pointers is not read. In the future, we expect to provide some support for initializing globals in constructor functions.

Two techniques help with initializing global arrays. First, if an array element could be 0 or NULL, the compiler will insert 0 for any elements you do not specify. For example, you can write

```
int x[37];
```

to declare a global array `x` initialized with 37 elements, all 0. Second, you can use the comprehension form

```
int x[37] = { for i < expr1 : expr2 }
```

provided that $expr_1$ and $expr_2$ and constant expressions. Currently, $expr_2$ may not use the variable `i`, but in the future it will be able to. Note that it is not possible to have a global variable of an abstract type because it is impossible to know any constant expression of that type.

Are there threads? Cyclone does not yet have a threads library and some of the libraries are not re-entrant. In addition, because Cyclone uses unboxed structs of three words to represent fat pointers, and updating them is not an atomic operation, it's possible to introduce unsoundnesses by adding concurrent threads. However, in the future, we plan to provide support for threads and a static analysis for preventing these and other forms of data races.

Can I use `setjmp` and `longjmp`? No. However, Cyclone has exceptions, which can be used for non-local control flow. The problem with `setjmp` and `longjmp` is that safety demands we prohibit a `longjmp` to a place no longer on the stack. A future release may have more support for non-local control flow.

What types are allowed for union members? Currently, `union` members can be just about any type, other than those with kind (`A`) (see question on kinds, above). Examples include numeric types (including bit fields and enumerations), structs and tuples of allowable union-member types, and other unions. However, if a union contains a pointer type, you can only write the pointer, not read it. This prevents effectively casting an `int` to a pointer by writing an `int` member and then reading the pointer, for example. To use pointers as normal within a union, you must use `@tagged` unions.

Why can't I do anything with values of type `void *`? Because we cannot know the size of an object pointed to by a pointer of type void *, we prohibit derefencing the pointer or casting it to a different pointer type. To write code that works for all pointer types, use type variables and polymorphism. Tagged unions can also substitute in some cases where `void *` is used in C.

What is `aprintf`? The `aprintf` function is just like `printf`, but the output is placed in a new string allocated on the heap. Note that you can use the more general function `rprintf` to allocate the output in a region of your choosing.

How do I access command-line arguments? The type of `main` should be

```
int main(int argc, char ?? argv);
```

As in C, `argc` is the number of command-line arguments and `argv[i]` is a string with the $\texttt{i}^{\text{th}}$ argument. Unlike C, `argv` and each element of `argv` carry bounds information. Note that `argc` is redundant—it is always equal to `numelts(argv)`.

Why can't I pass a stack pointer to certain functions? If the type of a function parameter is a pointer into the heap region, it cannot be passed a stack parameter. Pointer types in typedef and struct definitions refer to the heap region unless there is an explicit region annotation.

Why do I get an incomprehensible error when I assign a local's address to a pointer variable
If the pointer variable has a type indicating that it points into the heap, then the assignment is illegal. Try initializing the pointer variable with the local's address, rather than delaying the assignment until later.

How much pointer arithmetic can I do? On fat pointers, you can add or subtract an int (including via increment/decrement), as in C. It is okay for the result to be outside the bounds of the object pointed to; it is a run-time error to dereference outside of the bounds. (The compiler inserts bounds information and a run-time check; an exception is thrown if the check fails.) You can also do pointer arithmetic on zero-terminated pointers. Currently, we do not support pointer arithmetic on the other pointer types. As in C, you can subtract two pointers of the same type; the type of the result is `unsigned int`.

What is the type of a literal string? The type of the string constant `"foo"` is `char @{4}` (remember the trailing null character). However, there are implicit casts from `char @{4}` to `char @{2}`, `char *{4}`, and `char ?`, so you shouldn't have to think too much about this.

Are strings NUL-terminated? Cyclone follows C's lead on this. String literals like `"foo"` are NUL-terminated. Many of the library functions consider a NUL character to mark the end of a string. And library functions that return strings often ensure that they are NUL terminated. However, there is no guarantee that a string is NUL terminated. For one thing, as in C, the terminating NUL may be overwritten by any character. In C this can be exploited to cause buffer overflows. To avoid this in Cyclone, strings generally have type `char ?`, that is, they carry bounds information. In Cyclone a string ends when a NUL character is found, or when the bounds are exceeded.

How do I use `malloc`? `malloc` is a Cyclone primitive, not a library function. Currently it has an extremely restricted syntax: You must write `malloc(sizeof(`*type*`))`. The result has type *type*`@`, so usually there is no need to explicitly cast the result (but doing so is harmless). Usually the construct `new` *expr* is more convenient than malloc followed by initialization, but `malloc` can be useful for certain idioms and when porting C code.

Notice that you cannot (yet) use `malloc` to allocate space for arrays (as in the common idiom, `malloc(n*sizeof(`*type*`))`. Also, the type-checker uses a conservative analysis to ensure that the fields of the allocated space are written before they are used.

Can I call free? Yes and no. Individual memory objects may not be freed. In future versions, we may support freeing objects for which you can prove that there are no other pointers to the object. Until then, you must rely on a garbage collector to reclaim heap objects or use regions (similar to "arenas" or "zones") for managing collections of objects.

For porting code, we have defined a `free` function that behaves as a no-op, having type

```
void free(‘a::A ?);
```

Is there a garbage collector? Yes, we use the Boehm-Demers-Weiser conservative collector. If you don't want to use the garbage collector (e.g., because you know that your program does little or no heap allocation), you can use the `-nogc` flag when linking your executable. This will make the executable smaller.

If you link against additional C code, that code must obey the usual rules for conservative garbage collection: no wild pointers and no calling `malloc` behind the collector's back. Instead, you should call `GC_malloc`. See the collector's documentation for more information.

Note that if you allocate all objects on the stack, garbage collection will never occur. If you allocate all objects on the stack or in regions, it is very unlikely collection will occur and nothing will actually get collected.

How can I make a stack-allocated array? As in C, you declare a local variable with an array type. Also as in C, all uses of the variable, except as an argument to `sizeof` and `&`, are promoted to a pointer. If your declaration is

```
int x[256];
```

then uses of `x` have type `int @'L{256}` where `L` is the name of the block in which `x` is declared. (Most blocks are unnamed and the compiler just makes up a name.)

Stack-allocated arrays must be initialized when they are declared (unlike other local variables). Use an array-initializer, as in

```
int y[] = { 0, 1, 2, 3 };
int z[] = { for i < 256 : i };
```

To pass (a pointer to) the array to another function, the function must have a type indicating it can accept stack pointers, as explained elsewhere.

Can I use `salloc` or `realloc`? Currently, we don't provide support for `salloc`. For `realloc`, we do provide support, but only on heap-allocated `char ?` buffers.

Why do I have to cast from * to @ if I've already tested for NULL? Our compiler is not as smart as you are. It does not realize that you have tested for NULL, and it insists on a check (the cast) just to be sure. You can leave the cast implicit, but the compiler will emit a warning. We are currently working to incorporate a flow analysis to omit spurious warning. Because of aliasing, threads, and undefined evaluation order, a sound analysis is non-trivial.

Why can't a function parameter or struct field have type 'a::M? Type variables of memory kind can be instantiated with types of any size. There is no straightforward way to compile a function with an argument of arbitrary size. The obvious way to write such a function is to manipulate a pointer to the arbitrary size value instead. So your parameter should have type 'a::M * or 'a::M @.

Can I see how Cyclone compiles the code? Just compile with flags -save-c and -pp. This tells the compiler to save the C code that it builds and passes to gcc, and print it out using the pretty-printer. You will have to work to make some sense out of the C code, though. It will likely contain many extern declarations (because the code has already gone through the preprocessor) and generated type definitions (because of tuples, tagged unions, and questionable pointers). Pattern-matching code gets translated to a mess of temporary variables and goto statements. Array-bounds checks and NULL checks can clutter array-intensive and pointer-intensive code. And all typedefs are expanded away before printing the output.

Can I use gdb on the output? You can run gdb, but debugging support is not all the way there yet. By default, source-level debugging operations within gdb will reference the C code generated by the Cyclone compiler, not the Cyclone source itself. In this case, you need to be aware of three things. First, you have to know how Cyclone translates top-level identifiers to C identifiers (it prepends Cyc_- and separates namespaces by _ instead of ::) so you can set breakpoints at functions. Second, it can be hard to print values because many Cyclone types get translated to void *. Third, we do not yet have source correlation, so if you step through code, you're stepping through C code, not Cyclone code.

To improve this situation somehwat, you can compile your files with the option `--lineno`. This will insert `#line` directives in the generated C code that refer to the original Cyclone code. This will allow you to step through the program and view the Cyclone source rather than the generated C. However, doing this has two drawbacks. First, it may occlude some operation in the generated C code that is causing your bug. Second, compilation with `--lineno` is significantly slower than without. Finally, the result is not bug-free; sometimes the debugger will fall behind the actual program point and print the wrong source lines; we hope to fix this problem soon.

Two more hints: First, on some architectures, the first memory allocation appears to seg fault in `GC_findlimit`. This is correct and documented garbage-collector behavior (it handles the signal but `gdb` doesn't know that); simply continue execution. Second, a common use of `gdb` is to find the location of an uncaught exception. To do this, set a breakpoint at `throw` (a function in the Cyclone runtime).

Can I use `gprof` on the output? Yes, just use the `-pg` flag. You should also rebuild the Cyclone libraries and the garbage collector with the `-pg` flag. The results of `gprof` make sense because a Cyclone function is compiled to a C function.

Notes for Cygwin users: First, the versions of `libgmon.a` we have downloaded from cygnus are wrong (every call gets counted as a self-call). We have modified libgmon.a to fix this bug, so download our version and put it in your cygwin/lib directory. Second, timing information should be ignored because `gprof` is only sampling 100 or 1000 times a second (because it is launching threads instead of using native Windows profiling). Neither of these problems are Cyclone-specific.

Is there an Emacs mode for Cyclone? Sort of. In the `doc/` directory of the distribution you will find a `font-lock.el` file and elisp code (in `cyclone_dot_emacs.el`) suitable for inclusion in your `.emacs` file. However, these files change C++ mode and use it for Cyclone rather than creating a new Cyclone mode. Of course, we intend to make our own mode rather than destroy C++-mode's ability to be good for C++. Note that we have not changed the C++ indentation rules at all; our elisp code is useful only for syntax highlighting.

Does Cyclone have something to do with runtime code generation? Cyclone has its roots in Popcorn, a language which was safe but not as compatible with C. An offshoot of Popcorn added safe runtime code generation, and was called Cyclone. The current Cyclone language is a merger of the two, refocused on safety and C compatibility. Currently, the language does not have support for runtime code generation.

What platforms are supported? You need a platform that has gcc, GNU make, ar, sed, either bash or ksh, and the ability to build the Boehm-Demers-Weiser garbage collector. Furthermore, the size of `int` and all C pointers must be the same. We actively develop Cyclone in Cygwin (a Unix emulation layer for Windows 98, NT, 2K), Linux, and Mac OS X. Versions have run on OpenBSD and FreeBSD.

Why aren't there more libraries? We are eager to have a wider code base, but we are compiler writers with limited resources. Let us know of useful code you write.

Why doesn't `List::imp_rev(l)` change `l` to its reverse? The library function `List::imp_rev` mutates its argument by reversing the `tl` fields. It returns a pointer to the new first cell (the old last cell), but `l` still points to the old first cell (the new last cell).

Can I inline functions? Functions can be declared inline as in ISO C99. You can get additional inlining by compiling the Cyclone output with the `-O2` flag. Whether a function is inlined or not has no effect on Cyclone type-checking.

If Cyclone is safe, why does my program crash? There are certain classes of errors that Cyclone does not attempt to prevent. Two examples are stack overflow and various numeric traps, such as division-by-zero. It is also possible to run out of memory. Other crashes could be due to compiler bugs or linking against buggy C code (or linking incorrectly against C code).

Note that when using `gdb`, it may appear there is a seg fault in GC_findlimit(). This behavior is correct; simply continue execution.

What are compile-time constants? Cyclone's compile-time constants are `NULL`, integer and character constants, and arithmetic operations over compile-time constants. Unlike C, `sizeof(t)` is not an integral constant expression in our current implementation of Cyclone because our compiler does not know the actual size of aggregate types; we hope to repair this in a future version. Constructs requiring compile-time constants are: tuple-subscript (e.g., `x[3]` for tuple `x`), sizes in array declarations (e.g., `int y[37]`), and sizes in pointer bounds (e.g., `int * x{124}`).

How can I get the size of an array? If *expr* is an array, then `numelts(`*expr*`)` returns the number of elements in the array. If *expr* is a pointer to an array, `numelts(`*expr*`)` returns the number of elements in the array pointed to. If *expr* is a fat pointer, then the number of elements is calculated at runtime from the bounds information contained in the fat pointer. For other types, the size is determined at compile-time.

C Libraries

C.1 C Libraries

Cyclone provides partial support for the following C library headers, at least on Linux. On other platforms (e.g., Cygwin), some of these headers are not available. Furthermore, not all definitions from these headers are available, but rather, those that we could easily make safe.

```
<aio.h>          <arpa/inet.h>     <assert.h>
<complex.h>      <cpio.h>          <ctype.h>
<dirent.h>       <dlfcn.h>         <errno.h>
<fcntl.h>        <fenv.h>          <float.h>
<fmtmsg.h>       <fnmatch.h>       <ftw.h>
<getopt.h>       <glob.h>          <grp.h>
<inttypes.h>     <iso646.h>        <langinfo.h>
<libgen.h>       <limits.h>        <locale.h>
<math.h>         <monetary.h>      <mqueue.h>
<ndbm.h>         <net/if.h>        <netdb.h>
<netinet/in.h>   <netinet/tcp.h>   <nl_types.h>
<poll.h>         <pthread.h>       <pwd.h>
<regexp.h>       <sched.h>         <search.h>
<semaphore.h>    <setjmp.h>        <signal.h>
<spawn.h>        <stdarg.h>        <stdbool.h>
<stddef.h>       <stdint.h>        <stdio.h>
<stdlib.h>       <string.h>        <strings.h>
<stropts.h>      <sys/dir.h>       <sys/file.h>
<sys/ioctl.h>    <sys/ipc.h>       <sys/mman.h>
<sys/msg.h>      <sys/resource.h>  <sys/select.h>
<sys/sem.h>      <sys/shm.h>       <sys/socket.h>
<sys/stat.h>     <sys/statvfs.h>   <sys/syslog.h>
<sys/time.h>     <sys/timeb.h>     <sys/times.h>
<sys/types.h>    <sys/uio.h>       <sys/un.h>
<sys/utsname.h>  <sys/wait.h>      <tar.h>
<termios.h>      <tgmath.h>        <time.h>
<trace.h>        <ucontext.h>      <ulimit.h>
<unistd.h>       <utime.h>         <utmpx.h>
<wchar.h>        <wctype.h>        <wordexp.h>
```

C.2 `<array.h>`

Defines namespace Array, implementing utility functions on arrays.

```
void qsort(cmpfn_t<'a,'r,'r>,'a ?@aqual('EQ265)'r x,
           int len);
```

`qsort(cmp,x,len)` sorts the first `len` elements of array `x` into ascending order (according to the comparison function `cmp`) by the Quick-

Sort algorithm. `cmp(a,b)` should return a number less than, equal to, or greater than 0 according to whether `a` is less than, equal to, or greater than `b`. `qsort` throws `Core::InvalidArg("Array::qsort")` if `len` is negative or specifies a segment outside the bounds of `x`.

`qsort` is not a stable sort.

```
void msort(cmpfn_t<'a,'H,'H>,'a ?@aqual('EQ268) x,int len);
```

`msort(cmp,x,len)` sorts the first `len` elements of array `x` into ascending order (according to the comparison function `cmp`), by the Merge-Sort algorithm. `msort` throws `Core::InvalidArg("Array::msort")` if `len` is negative or specifies a segment outside the bounds of `x`.

`msort` is a stable sort.

```
'a ?@aqual('EQ272) from_list(List::list_t<'a> l);
```

`from_list(l)` returns a heap-allocated array with the same elements as the list `l`.

```
List::list_t<'a> to_list('a ?@aqual('EQ276) x);
```

`to_list(x)` returns a new heap-allocated list with the same elements as the array `x`.

```
'a ?@aqual('EQ280) copy('a ?@aqual('EQ284) x);
```

`copy(x)` returns a fresh copy of array `x`, allocated on the heap.

```
'b ?@aqual('EQ288) map('b (@ f)('a),'a ?@aqual('EQ292) x);
```

`map(f,x)` applies `f` to each element of `x`, returning the results in a new heap-allocated array.

```
'b ?@aqual('EQ297) map_c('b (@ f)('c,'a),'c env,'a ?@aqual('EQ301) x);
```

`map_c(f,env,x)` is like `map(f,x)` except that `f` requires a closure `env` as its first argument.

```
void imp_map('a (@ f)('a),'a ?@aqual('EQ306) x);
```

`imp_map(f,x)` replaces each element `xi` of `x` with `f(xi)`.

```
void imp_map_c('a (@ f)('b,'a),'b env,'a ?@aqual('EQ311) x);
```

`imp_map_c` is a version of `imp_map` where the function argument requires a closure as its first argument.

```
datatype exn {
  Array_mismatch
};
```

`Array_mismatch` is thrown when two arrays don't have the same length.

```
'c ?@aqual('EQ316) map2('c (@ f)('a,'b),'a ?@aqual('EQ320) x,
                        'b ?@aqual('EQ324) y);
```

If `x` has elements `x1` through `xn`, and `y` has elements `y1` through `yn`, then `map2(f,x,y)` returns a new heap-allocated array with elements `f(x1,y1)` through `f(xn,yn)`. If `x` and `y` don't have the same number of elements, `Array_mismatch` is thrown.

```
void app('b (@ f)('a),'a ?@aqual('EQ328)'r x);
```

`app(f,x)` applies `f` to each element of `x`, discarding the results. Note that `f` must not return `void`.

```
void app_c('c (@ f)('a,'b),'a env,'b ?@aqual('EQ333) x);
```

`app_c(f,env,x)` is like `app(f,x)`, except that `f` requires a closure `env` as its first argument.

```
void iter(void (@ f)('a),'a ?@aqual('EQ338) x);
```

`iter(f,x)` is like `app(f,x)`, except that `f` returns `void`.

```
void iter_c(void (@ f)('b,'a),'b env,'a ?@aqual('EQ343) x);
```

`iter_c(f,env,x)` is like `app_c(f,env,x)` except that `f` returns `void`.

```
void app2('c (@ f)('a,'b),'a ?@aqual('EQ348) x,'b ?@aqual('EQ352) y);
```

If `x` has elements `x1` through `xn`, and `y` has elements `y1` through `yn`, then `app2(f,x,y)` performs `f(x1,y1)` through `f(xn,yn)` and discards the results. If `x` and `y` don't have the same number of elements, `Array_mismatch` is thrown.

```
void app2_c('d (@ f)('a,'b,'c),'a env,'b ?@aqual('EQ357) x,
            'c ?@aqual('EQ361) y);
```

`app2_c` is a version of `app` where the function argument requires a closure as its first argument.

```
void iter2(void (@ f)(‘a,‘b),‘a ?@aqual(‘EQ366) x,‘b ?@aqual(‘EQ370) y
```

iter2 is a version of app2 where the function returns void.

```
void iter2_c(void (@ f)(‘a,‘b,‘c),‘a env,‘b ?@aqual(‘EQ375) x,
             ‘c ?@aqual(‘EQ379) y);
```

iter2_c is a version of app2_c where the function returns void.

```
‘a fold_left(‘a (@ f)(‘a,‘b),‘a accum,‘b ?@aqual(‘EQ384) x);
```

If x has elements x1 through xn, then fold_left(f,accum,x) returns f(f(...(f(x2,f(x1,accum))),xn-1),xn).

```
‘a fold_left_c(‘a (@ f)(‘c,‘a,‘b),‘c env,‘a accum,‘b ?@aqual(‘EQ389) x
```

fold_left_c is a version of fold_left where the function argument requires a closure as its first argument.

```
‘b fold_right(‘b (@ f)(‘a,‘b),‘a ?@aqual(‘EQ394) x,
              ‘b accum);
```

If x has elements x1 through xn, then fold_right(f,accum,x) returns f(x1,f(x2,...,f(xn-1,f(xn,a))...)).

```
‘b fold_right_c(‘b (@ f)(‘c,‘a,‘b),‘c env,‘a ?@aqual(‘EQ399) x,
                ‘b accum);
```

fold_right_c is a version of fold_right where the function argument requires a closure as its first argument.

```
‘a ?@aqual(‘EQ404) rev_copy(‘a ?@aqual(‘EQ408) x);
```

rev_copy(x) returns a new heap-allocated array whose elements are the elements of x in reverse order.

```
void imp_rev(‘a ?@aqual(‘EQ412) x);
```

imp_rev(x) reverses the elements of array x.

```
bool forall(bool (@ pred)(‘a),‘a ?@aqual(‘EQ416) x);
```

forall(pred,x) returns true if pred returns true when applied to every element of x, and returns false otherwise.

```
bool forall_c(bool (@ pred)(‘a,‘b),‘a env,‘b ?@aqual(‘EQ421) x);
```

forall_c is a version of forall where the predicate argument requires a closure as its first argument.

```
bool exists(bool (@ pred)(‘a),‘a ?@aqual(‘EQ426) x);
```

exists(pred,x) returns true if pred returns true when applied to some element of x, and returns false otherwise.

```
bool exists_c(bool (@ pred)(‘a,‘b),‘a env,‘b ?@aqual(‘EQ431) x);
```

exists_c is a version of exists where the predicate argument requires a closure as its first argument.

```
$(‘a,‘b) ?@aqual(‘EQ436) zip(‘a ?@aqual(‘EQ439)‘r1 x,
                             ‘b ?@aqual(‘EQ443) y);
```

If x has elements x1 through xn, and y has elements y1 through yn, then zip(x,y) returns a new heap-allocated array with elements $(x1,y1) through $(xn,yn). If x and y don't have the same number of elements, Array_mismatch is thrown.

```
$(‘a ?@aqual(‘EQ447),‘b ?@aqual(‘EQ451)) split($(‘a,
                                                ‘b) ?@aqual(‘EQ455) x
```

If x has elements $(a1,b1) through $(an,bn), then split(x) returns a pair of new heap-allocated arrays with elements a1 through an, and b1 through bn.

```
bool memq(‘a ?@aqual(‘EQ459) l,‘a x);
```

memq(l,x) returns true if x is == an element of array l, and returns false otherwise.

```
bool mem(int (@ cmp)(‘a,‘a),‘a ?@aqual(‘EQ463) l,‘a x);
```

mem(cmp,l,x) is like memq(l,x) except that the comparison function cmp is used to determine if x is an element of l. cmp(a,b) should return 0 if a is equal to b, and return a non-zero number otherwise.

```
‘a ?@aqual(‘EQ468) extract(‘a ?@aqual(‘EQ472) x,int start,
                          int *@aqual(‘EQ476) len_opt);
```

extract(x,start,len_opt) returns a new array whose elements

are the elements of `x` beginning at index `start`, and continuing to the end of `x` if `len_opt` is NULL; if `len_opt` points to an integer `n`, then `n` elements are extracted. If `n<0` or there are less than `n` elements in `x` starting at `start`, then `Core::InvalidArg("Array::extract")` is thrown.

C.3 <bitvec.h>

Defines namespace Bitvec, which implements bit vectors. Bit vectors are useful for representing sets of numbers from 0 to `n`, where `n` is not too large.

```
typedef int ?@aqual('EQ1197)'r bitvec_t<'r>;
```

`bitvec_t` is the type of bit vectors.

```
bitvec_t new_empty(int);
```

`new_empty(n)` returns a bit vector containing `n` bits, all set to 0.

```
bitvec_t new_full(int);
```

`new_full(n)` returns a bit vector containing `n` bits, all set to 1.

```
bitvec_t new_copy(bitvec_t);
```

`new_copy(v)` returns a copy of bit vector `v`.

```
bool get(bitvec_t,int);
```

`get(v,n)` returns the `n`th bit of `v`.

```
void set(bitvec_t,int);
```

`set(v,n)` sets the `n`th bit of `v` to 1.

```
void clear(bitvec_t,int);
```

`clear(v,n)` sets the `n`th bit of `v` to 0.

```
bool get_and_set(bitvec_t,int);
```

`get_and_set(v,n)` sets the `n`th bit of `v` to 1, and returns its value before the set.

`void clear_all(bitvec_t);`

`clear_all(v)` sets every bit in `v` to 0.

`void set_all(bitvec_t);`

`set_all(v)` sets every bit in `v` to 1.

`bool all_set(bitvec_t bvec,int sz);`

`all_set(v)` returns true if every bit in `v` is set to 1, and returns false otherwise.

`void union_two(bitvec_t dest,bitvec_t src1,bitvec_t src2);`

`union_two(dest,src1,src2)` sets `dest` to be the union of `src1` and `src2`: a bit of `dest` is 1 if either of the corresponding bits of `src1` or `src2` is 1, and is 0 otherwise.

`void intersect_two(bitvec_t dest,bitvec_t src1,bitvec_t src2);`

`intersect_two(dest,src1,src2)` sets `dest` to be the intersection of `src1` and `src2`: a bit of `dest` is 1 if both of the corresponding bits of `src1` and `src2` are 1, and is 0 otherwise.

`void diff_two(bitvec_t dest,bitvec_t src1,bitvec_t src2);`

`diff_two(dest,src1,src2)` sets `dest` to be the difference of `src1` and `src2`: a bit of `dest` is 1 if the corresponding bit of `src1` is 1, and the corresponding bit of `src2` is 0; and is 0 otherwise.

`bool compare_two(bitvec_t src1,bitvec_t src2);`

`compare_two(src1,src2)` returns true if `src1` and `src2` are equal, and returns false otherwise.

C.4 <buffer.h>

Defines namespace Buffer, which implements extensible character arrays. It was ported from Objective Caml.

`typedef struct t @@aqual('EQ1678) T;`

`T` is the type of buffers.

```
T create(unsigned int n);
```

`create(n)` returns a fresh buffer, initially empty. `n` is the initial size of an internal character array that holds the buffer's contents. The internal array grows when more than `n` character have been stored in the buffer; it shrinks back to the initial size when `reset` is called. If `n` is negative, no exception is thrown; a buffer with a small amount of internal storage is returned instead.

```
mstring_t contents(T);
```

`contents(b)` heap allocates and returns a string whose contents are the contents of buffer `b`.

```
mstring_t<'H,UNIQUE> extract(T);
```

`extract(b)` is like `contents(b)` except that returns a unique pointer to the internal array itself, adding a zero terminator at the current position. This avoids making a copy. However, the buffer `b` is unusable until this array is restored.

```
bool restore(T,mstring_t<'H,UNIQUE>) __attribute__((consume(2)));
```

`restore(b,s)` restores the string `s` to be the internal array of buffer `b`. If `b` already has valid contents, the operation fails and `b` is freed, returning false. Otherwise, true is returned.

```
size_t length(T);
```

`length(b)` returns the number of characters in buffer `b`.

```
void clear(T);
```

`clear(b)` makes `b` have zero characters. Internal storage is not released.

```
void reset(T);
```

`reset(b)` sets the number of characters in `b` to zero, and sets the internal storage to the initial string. This means that any storage used to grow the buffer since the last create or reset can be freed.

```
void add_char(T,char);
```

`add_char(b,c)` appends character `c` to the end of `b`.

```
void add_substring(T,string_t,int offset,int len);
```

add_substring(b,s,ofs,len) takes len characters starting at offset ofs in string s and appends them to the end of b. If ofs and len do not specify a valid substring of s, then the function throws InvalidArg("Buffer::add_substring"). Note, the substring specified by offset and len may contain NUL (0) characters; in any case, the entire substring is appended to b, not just the substring up to the first NUL character.

```
void add_string(T,string_t);
```

add_string(b,s) appends the string s to the end of b.

```
void add_buffer(T buf_dest,T buf_source);
```

add_buffer(b1,b2) appends the current contents of b2 to the end of b1. b2 is not modified.

C.5 <core.h>

The file <core.h> defines some types and functions outside of any namespace, and also defines a namespace Core.

The following declarations are made outside of any namespace.

```
typedef const char ?@aqual(‘q)@nozeroterm‘r string_t<‘r,
                                                     ‘q>;
```

A string_t<‘r> is a constant array of characters allocated in region ‘r.

```
typedef char ?@aqual(‘q)@nozeroterm‘r mstring_t<‘r,
                                                  ‘q>;
```

An mstring_t<‘r> is a non-const (mutable) array of characters allocated in region ‘r.

```
typedef string_t<‘r1,‘q1> @@aqual(‘q2)‘r2 stringptr_t<‘r1,
                                                      ‘r2,
                                                      ‘q1,
                                                      ‘q2>;
```

A stringptr_t<‘r1,‘r2> is used when a “boxed” string is needed, for example, you can have a list of string pointers, but not a list of strings.

```
typedef mstring_t<‘r1,‘q1> @@aqual(‘q2)‘r2 mstringptr_t<‘r1,
                                                        ‘r2,
                                                        ‘q1,
                                                        ‘q2>;
```

`mstringptr_t` is the mutable version of `stringptr_t`.

```
typedef char *@aqual(‘q)@nozeroterm‘r Cbuffer_t<‘r,
                                                ‘q>;
```

`Cbuffer_t` is a possibly-NULL, non-zero-terminated C buffer

```
typedef char @@aqual(‘q)@nozeroterm‘r CbufferNN_t<‘r,
                                                  ‘q>;
```

`CbufferNN_t` is a non-NULL, non-zero-terminated C buffer

```
typedef const char ?@aqual(‘q)@nozeroterm‘r buffer_t<‘r,
                                                    ‘q>;
```

`buffer_t` is a non-zero-terminated dynamically sized buffer

```
typedef int bool;
```

In Cyclone, we use `bool` as a synonym for `int`. We also define macros `true` and `false`, which are 1 and 0, respectively.

The rest of the declarations are in namespace Core.

```
typedef tag_t<valueof_t(sizeof(‘a))> sizeof_t<‘a>;
```

`sizeof_typ<T>` is the singleton type of `sizeof(T)`.

```
struct Opt<‘a::B> {
  ‘a v;
};
```

A `struct Opt` is a cell with a single field, `v` (for value).

```
typedef struct Opt<‘a> *@aqual(‘q)‘r opt_t<‘a,‘r,‘q>;
```

An `opt_t` is a pointer to a `struct Opt`. An `opt_t` can be used to pass an optional value to a function, or return an optional result. For example, to return no result, return NULL; to return a result `x`, return `new Opt(x)`.

Another way to return an option result of type `t` would be to return a pointer to `t`. The `opt_t` type is useful primarily when porting Objective Caml code, which has a corresponding type.

```
opt_t<‘b,‘H,UNIQUE> opt_map(‘b (@ f)(‘a),opt_t<‘a> x);
```

`opt_map(f,x)` applies `f` to the value contained in option `x`, if any, and returns the result as an option; if `x` is NULL, `opt_map(f,x)` returns NULL.

```
mstring_t<‘H> new_string(unsigned int x) @ensures(x <= numelts(return_
```

`new_string(n)` allocates space for `n` characters on the heap and returns a pointer to the space. All of the characters are set to NUL (0).

```
mstring_t<‘r> rnew_string(region_t<‘r>,unsigned int x) @ensures(x <= r
```

`rnew_string(r,n)` allocates space for `n` characters in the region with handle `r`, and returns a pointer to the space. All of the characters are set to NUL (0).

```
mstring_t<‘r,‘q> rqnew_string(region_t<‘r>,aqual_t<‘q>,
                              unsigned int x) @ensures(x <= numelts(re
```

`rqnew_string(r,q,n)` same as above ...except allocates with aqual(‘q)

```
bool true_f(‘a);
```

`true_f` is the constant `true` function: `true_f(x)` returns `true` regardless of the value of `x`.

```
bool false_f(‘a);
```

`false_f` is the constant `false` function.

```
‘a fst($(‘a,‘b) @@aqual(‘EQ1705));
```

`fst(x)` returns the first element of the pair pointed to by `x`.

```
‘b snd($(‘a,‘b) @@aqual(‘EQ1709));
```

`snd(x)` returns the second element of the pair pointed to by `x`.

```
‘c third($(‘a,‘b,‘c) @@aqual(‘EQ1713));
```

`third(x)` returns the third element of the triple pointed to by `x`.

```
'a identity('a);
```
identity is the identity function: identity(x) returns x.

```
int intcmp(int,int);
```
intcmp is a comparison function on integers: intcmp(i1,i2) returns a number less than, equal to, or greater than 0 according to whether i1 is less than, equal to, or greater than i2.

```
int charcmp(char,char);
```
charcmp is a comparison function on char.

```
int ptrcmp('a @@aqual('EQ1716)'r,'a @@aqual('EQ1719)'r);
```
ptrcmp is a comparison function on pointers.

```
region_t<'C> current_handle();
```
current_handle() returns the region handle on the top of the LIFO region stack. If the region stack is empty, then this will be the heap region.

```
datatype exn {
  Invalid_argument(string_t)
};
```
Invalid_argument is an exception thrown by library functions when one of their arguments is inappropriate.

```
datatype exn {
  Failure(string_t)
};
```
Failure is an exception that's thrown by library functions when they encounter an unexpected error.

```
datatype exn {
  Impossible(string_t)
};
```
The Impossible exception is thrown when a supposedly impossible situation occurs (whether in a library or in your own code). For example, you might throw Impossible if an assertion fails.

```
datatype exn {
  Not_found
};
```

The `Not_found` exception is thrown by search functions to indicate failure. For example, a function that looks up an entry in a table can throw `Not_found` if the entry is not found.

```
int region_used_bytes(region_t<‘r>);
```

`region_used_bytes` returns the number of bytes currently allocated for region pages for Cyclone objects; i.e., does not account for overhead of region page data structures.

```
int region_free_bytes(region_t<‘r>);
```

`region_free_bytes` returns the number of bytes currently free in the current region page.

```
int region_alloc_bytes(region_t<‘r>);
```

`region_alloc_bytes` returns the number of bytes of allocated Cyclone objects in the region.

```
region_t<‘H> heap_region;
```

`heap_region` is the region handle of the heap.

```
aqual_t<REFCNT> refcnt_qual;
```

these handles are used for allocating pointers of each kind; RESTRICTED is abstract

```
void ufree(‘a *@aqual(UNIQUE) ptr) __attribute__((noliveunique(1)));
```

`ufree` frees a unique pointer.

```
void rufree(region_t<‘r> h,‘a *@aqual(UNIQUE)‘r ptr:single(‘r)) __attr
```

`refcnt_region` is the region handle of the reference-counted region. Data allocated in this region contains an additional reference count for managing aliases.

```
int refptr_count(‘a ?@aqual(REFCNT)‘r ptr);
```

`refptr_count(p)` returns the current reference count for `p` (always ¿= 1); `p` is not consumed.

```
'a ?@aqual(REFCNT)'r alias_refptr('a ?@aqual(REFCNT)'r ptr);
```

`alias_refptr(p)` returns an alias to `p`, and increments the reference count by 1. `p` is not consumed.

```
void drop_refptr('a ?@aqual(REFCNT) ptr) __attribute__((noliveunique(1
```

`drop_refptr(p)` decrements the reference count on `p` by 1. If the reference count goes to 0, it frees p. This will not recursively decrement reference counts to embedded pointers, meaning that those pointers will have to get GC'ed if `p` ends up being freed.

```
'a ?@aqual('EQ1742)@autoreleased'r autorelease_handle(region_t<'r> h,
                                                      'a ?@aqual('EQ17
```

`autorelease(h,p)` attaches the given reference-counted pointer to the region h; the count on `p` will be decremted when the region is freed. An alias into the pool is returned, and `p` is consumed.

```
'a ?@aqual('EQ1748)@autoreleased'C autorelease('a ?@aqual('EQ1751)'RC
```

`autorelease(p)` is the same as `autorelease_handle(current_-handle(),p)`.

```
'a ?@aqual('EQ1754)'RC inc_refptr('a ?@aqual('EQ1757)@autoreleased'r p
```

`inc_refptr(p)` increments the reference count of the given autoreleased pointer, returning a reference-counted pointer to that storage. This pointer will outlive the current autorelease pool.

```
struct DynamicRegion<'r::E>;
```

`struct DynamicRegion<'r>` is an abstract type for the dynamic region named `'r`. Dynamic regions can be created and destroyed at will, but access to them must be done through the open_region function.

```
typedef struct DynamicRegion<'r1> @@aqual('q)'r2 region_key_t<'r1,
                                                              'q,
                                                              'r2>;
```

A `region_key_t<'r1,'r2>` is a pointer (in `'r2`) to a `DynamicRegion<'r1>`. Keys are used as capabilities for accessing a dynamic region. You have to present a key to the open procedure to access the region.

```
typedef region_key_t<‘r,UNIQUE,‘r2> uregion_key_t<‘r,
                                                  ‘r2>;
```

A `uregion_key_t<‘r>` is a unique pointer to a `DynamicRegion<‘r>`. You can't make copies of the key, but if you call `free_ukey`, then you are assured that the region `‘r` is reclaimed.

```
typedef region_key_t<‘r,REFCNT,‘r2> rcregion_key_t<‘r,
                                                   ‘r2>;
```

A `rcregion_key_t<‘r>` is a reference-counted pointer to a `DynamicRegion<‘r>`. You can make copies of the key using `alias_refptr` which increments the reference count. You can call `free_rckey` to destroy the key, which will decrement the reference count. If the count reaches zero, then the region will be reclaimed.

```
struct NewDynamicRegion<‘q::Q> {<‘r>:single(‘r)
  region_key_t<‘r,‘q> key;
};
```

A `struct NewDynamicRegion<‘r2>` is used to return a new dynamic region `‘r`. The struct hides the name of the region and must be opened, guaranteeing that the type-level name is unique.

```
struct NewDynamicRegion<UNIQUE> _new_ukey(unsigned int dis_reaps,
                                          const char *@aqual(‘EQ1763)@
                                          const char *@aqual(‘EQ1767)@
                                          int lineno);
```

`new_ukey()` creates a fresh dynamic region `‘r` and returns a unique key for that region.

```
void free_ukey(uregion_key_t<‘r> k) __attribute__((consume(1)));
```

`free_ukey(k)` takes a unique key for the region `‘r` and deallocates the region `‘r` and destroys the key `k`.

```
void free_rckey(rcregion_key_t<‘r> k) __attribute__((consume(1)));
```

`free_rckey(k)` takes a reference-counted key for the region `‘r`, decrements the reference count and destroyes the key `k`. If the reference count becomes zero, then all keys have been destroyed and the region `‘r` is deallocated.

```
‘result open_region(region_key_t<‘r,‘q> key,‘arg arg,
                    ‘result (@ body)(region_t<‘r> h,
                                     ‘arg arg):RESTRICTED >= ‘q);
```

open_region(k,arg,body) extracts a region handle h for the region ‘r which the k provides access to. The handle and value arg are passed to the function pointer body and the result is returned. Note that k can be either a uregion_key_t or an rcregion_key_t. The caller does not need to have static access to region ‘r when calling open_region but that capability is allowed within body. In essence, the key k provides dynamic evidence that ‘r is still live.

```
void rethrow(datatype exn @@aqual(‘EQ1780)) __attribute__((noreturn));
```

throws the exception without updating the source or line number information. Useful for try ... catch case e: ... rethrow(e);

```
const char *@aqual(‘EQ1784) get_exn_name(datatype exn @@aqual(‘EQ1788)
```

returns the name of the exception as a string

```
const char *@aqual(‘EQ1792) get_exn_filename();
```

if an exception is thrown, then you can use @get_exn_filename@ to determine what source file caused the exception.

```
void set_uncaught_exn_fun(int (*@aqual(‘EQ1796))());
```

if an exception is thrown and no handler is installed, the function passed as an argument is executed _before_ the exception is actually thrown. (A useful argument is Execinfo::bt.)

```
int get_exn_lineno();
```

if an exception is thrown, then you can use @get_exn_lineno@ to determine what source line caused the exception.

```
__cyclone_internal_array_t<‘a,‘i,‘r> arrcast(‘a ?@aqual(‘EQ1805)‘r dyn
                                             __cyclone_internal_single
                                             sizeof_t<‘a> sz);
```

Converts dyn to a thin pointer with length bd, assuming that bd is less than numelts(dyn); sz is the size of the elements in dyn. This routine is useful for eliminating bounds checks within loops.

```
‘a ?@aqual(‘EQ1808)‘r mkfat(__nn_cyclone_internal_array_t<‘a,
                                                        ‘i,
                                                        ‘r> arr,
                                sizeof_t<‘a> s,__cyclone_internal_singleto
```

mkfat can be used to convert a thin pointer (@) of elements of type ‘a to a fat pointer (?). It requires that you pass in the size of the element type, as well as the number of elements.

```
struct ThinRes<‘a::A,‘r> {<‘i::I>
  __cyclone_internal_array_t<‘a,‘i,‘r> arr;
  __cyclone_internal_singleton<‘i> nelts;
};
```

`ThinRes<‘a,‘r>` is used as the return type of mkthin. It includes an abstract integer `‘i`, and a thin pointer `arr` to a vector of `‘i` values of type `‘a`, living in region `‘r`, and a `tag_t<‘i>` value `nelts` which has the value of `‘i`.

```
struct ThinRes<‘a,‘r> mkthin(‘a ?@aqual(‘EQ1811)‘r dyn,
                             sizeof_t<‘a> sz);
```

mkthin is a special case of arrcast, which converts a fat pointer to a thin pointer and its bound. It requires that you pass in the size of the element type.

```
unsigned int arr_prevsize(‘a ?@aqual(‘EQ1815) arr,sizeof_t<‘a> elt_sz)
```

Returns the distance, in terms of elements of size `elt_sz`, to the start of the buffer pointed to by `arr`.

C.6 <dict.h>

Defines namespace Dict, which implements dictionaries: mappings from keys to values. The dictionaries are implemented functionally: adding a mapping to an existing dictionary produces a new dictionary, without affecting the existing dictionary. To enable an efficient implementation, you are required to provide a total order on keys (a comparison function).

We follow the conventions of the Objective Caml Dict library as much as possible.

Namespace Dict implements a superset of namespace SlowDict, except that `delete_present` is not supported.

```
typedef struct Dict<‘a,‘b,‘r> dict_t<‘a,‘b,‘r>;
```

A value of type `dict_t<‘a,‘b,‘r>` is a dictionary that maps keys of type `‘a` to values of type `‘b`; the dictionary datatypes live in region `‘r`.

```
datatype exn {
  Present
};
```

`Present` is thrown when a key is present but not expected.

```
datatype exn {
  Absent
};
```

`Absent` is thrown when a key is absent but should be present.

```
dict_t<‘a,‘b> empty(int (@@aqual(‘EQ2427) cmp)(‘a,‘a)) @throws(0);
```

`empty(cmp)` returns an empty dictionary, allocated on the heap. `cmp` should be a comparison function on keys: `cmp(k1,k2)` should return a number less than, equal to, or greater than 0 according to whether `k1` is less than, equal to, or greater than `k2` in the ordering on keys.

```
dict_t<‘a,‘b,‘r> rempty(region_t<‘r>,int (@@aqual(‘EQ2430) cmp)(‘a,
                                                                ‘a)) @
```

`rempty(r,cmp)` is like `empty(cmp)` except that the dictionary is allocated in the region with handle `r`.

```
dict_t<‘a,‘b,‘r+‘r2> rshare(region_t<‘r>,dict_t<‘a,
                                                ‘b,
                                                ‘r2>);
```

`rshare(r,d)` creates a virtual copy in region `‘r` of the dictionary `d` that lives in region `‘r2`. The internal data structures of the new dictionary share with the old one.

```
bool is_empty(dict_t d) @throws(0);
```

`is_empty(d)` returns true if `d` is empty, and returns false otherwise.

```
int cardinality(dict_t d);
```

`cardinality(d)` returns the number of keys in the dictionary.

```
bool member(dict_t<‘a> d,‘a k);
```

member(d,k) returns true if k is mapped to some value in d, and returns false otherwise.

```
dict_t<‘a,‘b,‘r> insert(dict_t<‘a,‘b,‘r> d,‘a k,‘b v);
```

insert(d,k,v) returns a dictionary with the same mappings as d, except that k is mapped to v. The dictionary d is not modified.

```
dict_t<‘a,‘b,‘r> insert_new(dict_t<‘a,‘b,‘r> d,‘a k,
                            ‘b v);
```

insert_new(d,k,v) is like insert(d,k,v), except that it throws Present if k is already mapped to some value in d.

```
dict_t<‘a,‘b,‘r> inserts(dict_t<‘a,‘b,‘r> d,list_t<$(‘a,
                                                    ‘b) @@aqual(‘EQ24
```

inserts(d,l) inserts each key, value pair into d, returning the resulting dictionary.

```
dict_t<‘a,‘b> singleton(int (@@aqual(‘EQ2437) cmp)(‘a,
                                                   ‘a),
                        ‘a k,‘b v) @throws(0);
```

singleton(cmp,k,v) returns a new heap-allocated dictionary with a single mapping, from k to v.

```
dict_t<‘a,‘b,‘r> rsingleton(region_t<‘r>,int (@@aqual(‘EQ2440) cmp)(‘a
                                                                    ‘a
                            ‘a k,‘b v) @throws(0);
```

rsingleton(r,cmp,k,v) is like singleton(cmp,k,v), except the resulting dictionary is allocated in the region with handle r.

```
‘b lookup(dict_t<‘a,‘b> d,‘a k);
```

lookup(d,k) returns a pointer to the value associated with key k in d, or throws Absent if k is not mapped to any value.

```
‘b lookup_other(dict_t<‘a,‘b> d,int (@ cmp)(‘c,‘a),
                ‘c k);
```

lookup_other(d,cmp,k) returns a pointer to the value associated

with key k in d, or throws Absent if k is not mapped to any value. The comparison function associated with the dictionary is ignored and instead, the cmp argument is used. Note that cmp must respect the same ordering constraints as the dictionary's built-in comparison in order to succeed. This is useful when the dictionary has keys that are pointers into one region, but you want to look up with a key that is a pointer into another region.

```
'b *@aqual('EQ2444)'r lookup_opt(dict_t<'a,'b,'r> d,
                                 'a k);
```

lookup_opt(d,k) returns NULL if k is not mapped to any value in d, and returns a non-NULL, heap-allocated option containing the value k is mapped to in d otherwise.

```
bool lookup_bool(dict_t<'a,'b> d,'a k,'b @@aqual('EQ2448) ans);
```

If d maps k to a value, then lookup_bool(d,k,ans) assigns that value to *ans and returns true; otherwise, it returns false.

```
'c fold('c (@ f)('a,'b,'c),dict_t<'a,'b> d,'c accum);
```

If d has keys k1 through kn mapping to values v1 through vn, then fold(f,d,accum) returns f(k1,v1,...f(kn,vn,accum)...).

```
'c fold_c('c (@ f)('d,'a,'b,'c),'d env,dict_t<'a,'b> d,
          'c accum);
```

fold_c(f,env,d,accum) is like fold(f,d,accum) except that f takes closure env as its first argument.

```
void app('c (@ f)('a,'b),dict_t<'a,'b> d);
```

app(f,d) applies f to every key/value pair in d; the results of the applications are discarded. Note that f cannot return void.

```
void app_c('c (@ f)('d,'a,'b),'d env,dict_t<'a,'b> d);
```

app_c(f,env,d) is like app(f,d) except that f takes closure env as its first argument.

```
void iter(void (@ f)('a,'b),dict_t<'a,'b> d);
```

iter(f,d) is like app(f,d) except that f returns void.

```
void iter_c(void (@ f)(‘c,‘a,‘b),‘c env,dict_t<‘a,‘b> d);
```

`iter_c(f,env,d)` is like `app_c(f,env,d)` except that `f` returns `void`.

```
void iter2(void (@@aqual(‘EQ2458) f)(‘b,‘b),dict_t<‘a,
                                                  ‘b> d1,
           dict_t<‘a,‘b> d2);
```

For every key `k` in the domain of both `d1` and `d2`, `iter2(f,d1,d2)` performs `f(lookup(d1,k), lookup(d2,k))`. If there is any key present in `d1` but not `d2`, then `Absent` is thrown.

```
void iter2_c(void (@@aqual(‘EQ2462) f)(‘c,‘b,‘b),‘c env,
             dict_t<‘a,‘b> d1,dict_t<‘a,‘b> d2);
```

`iter2_c` is like `iter` except that `f` takes a closure as its first argument.

```
‘c fold2_c(‘c (@@aqual(‘EQ2466) f)(‘d,‘a,‘b1,‘b2,‘c),
           ‘d env,dict_t<‘a,‘b1> d1,dict_t<‘a,‘b2> d2,
           ‘c accum);
```

If `k1` through `kn` are the keys of `d1`, then `fold2_c(f,env,d1,d2,accum)` returns `f(env,k1,lookup(k1,d1),lookup(k1,d2), ... f(env,kn,lookup(kn,`
If there is any key present in `d1` but not `d2`, then `Absent` is thrown.

```
dict_t<‘a,‘b,‘r> rcopy(region_t<‘r>,dict_t<‘a,‘b>);
```

`rcopy(r,d)` returns a copy of `d`, newly allocated in the region with handle `r`.

```
dict_t<‘a,‘b> copy(dict_t<‘a,‘b>);
```

`copy(r,d)` returns a copy of `d`, newly allocated on the heap.

```
dict_t<‘a,‘c> map(‘c (@ f)(‘b),dict_t<‘a,‘b> d);
```

`map(f,d)` applies `f` to each value in `d`, and returns a new dictionary with the results as values: for every binding of a key `k` to a value `v` in `d`, the result binds `k` to `f(v)`. The returned dictionary is allocated on the heap.

```
dict_t<‘a,‘c,‘r> rmap(region_t<‘r>,‘c (@ f)(‘b),dict_t<‘a,
                                                  ‘b> d);
```

`rmap(r,f,d)` is like `map(f,d)`, except the resulting dictionary is allocated in the region with handle `r`.

```
dict_t<‘a,‘c> map_c(‘c (@ f)(‘d,‘b),‘d env,dict_t<‘a,
                                                ‘b> d);
```

`map_c(f,env,d)` is like `map(f,d)` except that `f` takes `env` as its first argument.

```
dict_t<‘a,‘c,‘r> rmap_c(region_t<‘r>,‘c (@ f)(‘d,‘b),
                        ‘d env,dict_t<‘a,‘b> d);
```

`rmap_c(r,f,env,d)` is like `map_c(f,env,d)` except that the resulting dictionary is allocated in the region with handle `r`.

```
dict_t<‘a,‘b,‘r> union_two_c(‘b (@@aqual(‘EQ2474) f)(‘c,
                                                    ‘a,
                                                    ‘b,
                                                    ‘b),
                             ‘c env,dict_t<‘a,‘b,‘r> d1,
                             dict_t<‘a,‘b,‘r> d2);
```

`union_two_c(f,env,d1,d2)` returns a new dictionary with a binding for every key in `d1` or `d2`. If a key appears in both `d1` and `d2`, its value in the result is obtained by applying `f` to the two values, the key, and env. Note that the resulting dictionary is allocated in the same region as `d2`. (We don't use `union` as the name of the function, because `union` is a Cyclone keyword.)

```
dict_t<‘a,‘b,‘r> intersect(‘b (@@aqual(‘EQ2478) f)(‘a,
                                                   ‘b,
                                                   ‘b),
                           dict_t<‘a,‘b,‘r> d1,dict_t<‘a,
                                                      ‘b,
                                                      ‘r> d2);
```

`intersect(f,d1,d2)` returns a new dictionary with a binding for every key in both `d1` and `d2`. For every key appearing in both `d1` and `d2`, its value in the result is obtained by applying `f` to the key and the two values. Note that the input dictionaries and result must be allocated in the same region.

```
dict_t<‘a,‘b,‘r> intersect_c(‘b (@@aqual(‘EQ2482) f)(‘c,
                                                    ‘a,
                                                    ‘b,
                                                    ‘b),
                             ‘c env,dict_t<‘a,‘b,‘r> d1,
                             dict_t<‘a,‘b,‘r> d2);
```

`intersect_c(f,env,d1,d2)` is like `intersect(f,d1,d2)`, except that `f` takes `env` as its first argument.

```
bool forall_c(bool (@ f)(‘c,‘a,‘b),‘c env,dict_t<‘a,
                                                ‘b> d);
```

`forall_c(f,env,d)` returns true if `f(env,k,v)` returns true for every key `k` and associated value `v` in `d`, and returns false otherwise.

```
bool forall_intersect(bool (@ f)(‘a,‘b,‘b),dict_t<‘a,
                                                 ‘b> d1,
                      dict_t<‘a,‘b> d2);
```

`forall_intersect(f,d1,d2)` returns true if `f(k,v1,v2)` returns true for every key `k` appearing in both `d1` and `d2`, where `v1` is the value of `k` in `d1`, and `v2` is the value of `k` in `d2`; and it returns false otherwise.

```
$(‘a,‘b) @@aqual(‘EQ2487)‘r rchoose(region_t<‘r>,dict_t<‘a,
                                                      ‘b> d);
```

`rchoose(r,d)` returns a key/value pair from `d`, allocating the pair in region `r`; if `d` is empty, `Absent` is thrown.

```
list_t<$(‘a,‘b) @@aqual(‘EQ2491)> to_list(dict_t<‘a,
                                                ‘b> d);
```

`to_list(d)` returns a list of the key/value pairs in `d`, allocated on the heap.

```
list_t<$(‘a,‘b) @@aqual(‘EQ2494)‘r,‘r> rto_list(region_t<‘r>,
                                                dict_t<‘a,
                                                       ‘b> d);
```

`rto_list(r,d)` is like `to_list(d)`, except that the resulting list is allocated in the region with handle `r`.

```
dict_t<‘a,‘b> filter(bool (@ f)(‘a,‘b),dict_t<‘a,‘b> d);
```

`filter(f,d)` returns a dictionary that has a binding of `k` to `v` for every binding of `k` to `v` in `d` such that `f(k,v)` returns true. The resulting dictionary is allocated on the heap.

```
dict_t<‘a,‘b,‘r> rfilter(region_t<‘r>,bool (@ f)(‘a,
                                                 ‘b),
                         dict_t<‘a,‘b> d);
```

`rfilter(r,f,d)` is like `filter(f,d)`, except that the resulting dictionary is allocated in the region with handle `r`.

```
dict_t<‘a,‘b> filter_c(bool (@ f)(‘c,‘a,‘b),‘c env,
                       dict_t<‘a,‘b> d);
```

`filter_c(f,env,d)` is like `filter(f,d)` except that `f` takes a closure `env` as its first argument.

```
dict_t<‘a,‘b,‘r> rfilter_c(region_t<‘r>,bool (@ f)(‘c,
                                                   ‘a,
                                                   ‘b),
                           ‘c env,dict_t<‘a,‘b> d);
```

`rfilter_c(r,f,env,d)` is like `filter_c(f,env,d)`, except that the resulting dictionary is allocated in the region with handle `r`.

```
dict_t<‘a,‘b> difference(dict_t<‘a,‘b> d1,dict_t<‘a,
                                                 ‘b> d2);
```

`difference(d1,d2)` returns a dictionary that has a binding of `k` to `v` for every binding of `k` to `v` in `d1` where `k` is not in `d2`. (Note that the values of `d2` are not relevant to `difference(d1,d2)`.) The resulting dictionary is allocated on the heap.

```
dict_t<‘a,‘b,‘r> rdifference(region_t<‘r>,dict_t<‘a,
                                                 ‘b> d1,
                             dict_t<‘a,‘b> d2);
```

`rdifference(d1,d2)` is like `difference(d1,d2)`, except that the resulting dictionary is allocated in the region with handle `r`.

```
dict_t<‘a,‘b> delete(dict_t<‘a,‘b>,‘a);
```

`delete(d,k)` returns a dictionary with the same bindings as `d`, except that any binding of `k` is removed. The resulting dictionary is allocated on the heap.

```
dict_t<‘a,‘b,‘r> rdelete(region_t<‘r>,dict_t<‘a,‘b>,
                         ‘a);
```

`rdelete(r,d,k)` is like `delete(d,k)` except that the result is allocated in the region with handle `r`.

```
dict_t<‘a,‘b,‘r> rdelete_same(dict_t<‘a,‘b,‘r>,‘a);
```

`rdelete_same(d,k)` is like `delete(d,k)`, except that the resulting dictionary is allocated in the same region as the input dictionary `d`. This can be faster than `delete(d,k)` because it avoids a copy when `k` is not a member of `d`.

```
Iter::iter_t<$(‘a,‘b),regions($(‘a,‘b))+‘r1+‘r2> make_iter(region_t<‘r
                                                           dict_t<‘a,
                                                                  ‘b,
                                                                  ‘r2>
```

`make_iter(rgn,d)` returns an iterator over the dictionary `d`; O(log n) space is allocated in `rgn` where n is the number of elements in `d`.

C.7 <filename.h>

Defines a namespace Filename, which implements some useful operations on file names that are represented as strings.

```
mstring_t concat(string_t,string_t);
```

Assuming that `s1` is a directory name and `s2` is a file name, `concat(s1,s2)` returns a new (heap-allocated) file name for the child `s2` of directory `s1`.

```
mstring_t chop_extension(string_t);
```

`chop_extension(s)` returns a copy of `s` with any file extension removed. A file extension is a period (‘.’) followed by a sequence of non-period characters. If `s` does not have a file extension, `chop_extension(s)` throws `Core::Invalid_argument("chop_extension")`.

```
mstring_t dirname(string_t);
```

`dirname(s)` returns the directory part of `s`. For example, if `s` is `"foo/bar/baz"`, `dirname(s)` returns `"foo/bar"`.

```
mstring_t basename(string_t);
```

`basename(s)` returns the file part of `s`. For example, if `s` is `"foo/bar/baz"`, `basename(s)` returns `"baz"`.

```
bool check_suffix(string_t,string_t);
```

`check_suffix(filename,suffix)` returns true if `filename` ends in `suffix`, and returns false otherwise.

```
mstring_t gnuify(string_t);
```

`gnuify(s)` forces `s` to follow Unix file name conventions: any Windows separator characters (backslashes) are converted to Unix separator characters (forward slashes).

C.8 <fn.h>

Defines namespace Fn, which implements closures: a way to package up a function with some hidden data (an environment). Many of the library functions taking function arguments have versions for functions that require an explicit environment; the closures of namespace Fn are different, they combine the function and environment, and the environment is hidden. They are useful when two functions need environments of different type, but you need them to have the same type; you can do this by hiding the environment from the type of the pair.

```
typedef struct Function<‘arg,‘res,‘bd> @@aqual(‘EQ2930) fn_t<‘arg,
                                                             ‘res,
                                                             ‘bd>;
```

A value of type `fn_t<‘arg,‘res,‘eff>` is a function and its closure; `‘arg` is the argument type of the function, `‘res` is the result type, and `‘bd` is a region that `regions(‘arg)` outlive.

```
fn_t<‘arg,‘res,‘bd> make_fn(‘res (@@aqual(‘EQ2933) f)(‘env,
                                                      ‘arg),
                            ‘env x:regions(‘env) <= ‘bd);
```

`make_fn(f,env)` builds a closure out of a function and an environment.

```
fn_t<‘arg,‘res,‘bd> fp2fn(‘res (@@aqual(‘EQ2936) f)(‘arg):regions(‘res
```

`fp2fn(f)` converts a function pointer to a closure.

```
‘res apply(fn_t<‘arg,‘res> f,‘arg x);
```

`apply(f,x)` applies closure `f` to argument `x` (taking care of the hidden environment in the process).

```
fn_t<‘a,‘c,‘bd2> compose(fn_t<‘a,‘b,‘bd> g,fn_t<‘b,
                                              ‘c,
                                              ‘bd> f:regions($(‘a,
                                                              ‘b,
                                                              ‘c) @
```

`compose(g,f)` returns the composition of closures `f` and `g`; `apply(compose(g,f),x)` has the same effect as `apply(f,apply(g,x))`.

```
fn_t<‘a,fn_t<‘b,‘c,‘bd2>,‘bd2> curry(fn_t<$(‘a,‘b) @@aqual(‘EQ2945),
                                        ‘c,‘bd> f:regions($(‘a,
                                                           ‘b,
                                                           ‘c) @@ac
```

`curry(f)` curries a closure that takes a pair as argument: if `x` points to a pair `$(x1,x2)`, then `apply(f,x)` has the same effect as `apply(apply(curry(f),x1),`

```
fn_t<$(‘a,‘b) @@aqual(‘EQ2952),‘c,‘bd3> uncurry(fn_t<‘a,
                                                 fn_t<‘b,
                                                      ‘c,
                                                      ‘bd2>,
                                                 ‘bd> f:regions($(
```

`uncurry(f)` converts a closure that takes two arguments in sequence into a closure that takes the two arguments as a pair: if `x` points to a

pair `$(x1,x2)`, then `apply(uncurry(f),x)` has the same effect as `apply(apply(f,x1),x2)`.

```
List::list_t<‘b,‘H> map_fn(fn_t<‘a,‘b,‘bd> f,List::list_t<‘a> x);
```

`map_fn(f,x)` maps the closure `f` on the list `x`: if `x` has elements `x1` through `xn`, then `map_fn(f,x)` returns a new heap-allocated list with elements `apply(f,x1)` through `apply(f,xn)`.

C.9 <graph.h>

Defines namespace Graph, which implements functional graphs that are parameterized over the type of the nodes of the graph. To create a graph you must supply a comparison function on nodes (the comparison function should return a number less than, equal to, or greater than 0 according to whether its first argument is less than, equal to, or greater than its second argument).

A node `s` that is the source of an edge (`s`,`t`) in a graph is called a source node of the graph; a node that appears only as a target is not a source node. Whether a node is a source node matters for some of the graph functions, notably, `scc` and `tsort`. Use `add_node` to ensure that a node is a source node.

```
typedef Dict::dict_t<‘a,Set::set_t<‘a>> graph_t<‘a>;
```

A value of type `graph_t<‘a>` is a graph with nodes of type `‘a`.

```
void print(FILE @@aqual(‘EQ3696)‘r f,graph_t<‘a> g,
           void (@@aqual(‘EQ3700) nodeprint)(FILE @@aqual(‘EQ3703)‘r,
                                             ‘a));
```

`print(f,g,nodeprint)` prints the graph `g` to file `f`. `nodeprint` must be a printing function for nodes. The output is in the DOT language, suitable for display by Graphviz.

```
graph_t<‘a> empty(int (@@aqual(‘EQ3706) cmp)(‘a,‘a));
```

`empty(cmp)` creates an empty graph given comparison function `cmp`. The graph is heap-allocated.

```
graph_t<‘a> add_node(graph_t<‘a> g,‘a s);
```

`add_node(g,s)` returns a graph identical to `g`, except that `s` is added as a source node of `g` (if necessary).

```
graph_t<'a> add_edge(graph_t<'a> g,'a s,'a t);
```

`add_edge(g,s,t)` returns a graph identical to `g` except that there is an edge from `s` to `t`. `t` is not added as a source node of the result.

```
graph_t<'a> add_edges(graph_t<'a> g,'a s,Set::set_t<'a,
                                                    'H> T);
```

`add_edges(g,s,T)` returns a graph identical to `g` except that there is an edge from `s` to every node in the set `T`. The nodes of `T` are not added as source nodes of the result.

```
graph_t<'a> remove_edge(graph_t<'a> g,'a s,'a t);
```

`remove_edge(g,s,t)` returns a graph identical to `g` except that any edge from `s` to `t` is removed.

```
graph_t<'a> remove_edges(graph_t<'a> g,'a s,Set::set_t<'a,
                                                      'H> T);
```

`remove_edge(g,s,T)` returns a graph identical to `g` except that any edge from `s` to a node in the set `T` is removed.

```
int is_edge(graph_t<'a> g,'a s,'a t);
```

`is_edge(g,s,t)` returns true if there is an edge from `s` to `t` in `g`, and returns false otherwise.

```
Set::set_t<'a> get_targets(graph_t<'a>,'a s);
```

`get_targets(g,s)` returns the set of targets of node `s` in graph `g`.

```
graph_t<'a> tc(graph_t<'a> g);
```

`tc(g)` returns the transitive closure of graph `g`: the nodes of `tc(g)` are the nodes of `g`, and there is an edge from a node `s` to a node `t` in `tc(g)` iff `s` can reach `t` by one or more edges in `g`. Note that `tc(g)` is not necessarily reflexive.

```
graph_t<'a> tkernel(graph_t<'a> g);
```

`tkernel(g)` returns the transitive kernel of graph `g`: the nodes of `tkernel(g)` are the nodes of `g`, and the edges `tkernel(g)` are a minimal subset of the edges of `g` such that `tc(tkernel(g))` is the same as `tc(g)`.

```
List::list_t<'a> tsort(graph_t<'a> g);
```

`tsort(g)` returns a list of the source nodes of `g`, in topological order. That is, if `s` can reach `t` by one or more edges in `g`, then `s` appears before `t` in `tsort(g)`, unless `t` can also reach `s` by one or more edges in `g`, in which case `s` and `t` will appear in some unspecified order in the output.

```
graph_t<'a> scc(graph_t<'a> g);
```

`scc(g)` returns the strongly connected components of graph `g`: the source nodes of `scc(g)` are the source nodes of `g`, and the targets of any node `s` in `scc(g)` are the members of the strongly connected component of `s` in `g`. We consider every `s` to be a member of its own strongly connected component.

C.10 <hashtable.h>

Defines namespace Hashtable, which implements mappings from keys to values. These hashtables are imperative—values are added and deleted destructively. (Use namespace Dict or SlowDict if you need functional (non-destructive) mappings.) To enable an efficient implementation, you are required to provide a total order on keys (a comparison function).

```
typedef struct Table<'a,'b,'r> @@aqual('EQ3846)'r table_t<'a,
                                                          'b,
                                                          'r>;
```

A `table_t<'a,'b>` is a hash table with keys of type `'a` and values of type `'b`.

```
table_t<'a,'b> create(int sz,int (@@aqual('EQ3849) cmp)('a,
                                                        'a),
                      int (@@aqual('EQ3852) hash)('a));
```

`create(sz,cmp,hash)` returns a new hash table that starts out with `sz` buckets. `cmp` should be a comparison function on keys: `cmp(k1,k2)` should return a number less than, equal to, or greater than 0 according to whether `k1` is less than, equal to, or greater than `k2`. `hash` should be a hash function on keys. `cmp` and `hash` should satisfy the following property: if `cmp(k1,k2)` is 0, then `hash(k1)` must equal `hash(k2)`.

```
table_t<‘a,‘b,‘r> rcreate(region_t<‘r>,int sz,int (@@aqual(‘EQ3855) cm

                          int (@@aqual(‘EQ3858) hash)(‘a));
```

`rcreate(r,sz,cmp,hash)` is similar to create but allocates its result in the region r instead of the heap.

```
void insert(table_t<‘a,‘b>,‘a key,‘b val);
```

`insert(t,key,val)` binds `key` to `value` in `t`.

```
‘b lookup(table_t<‘a,‘b>,‘a key);
```

`lookup(t,key)` returns the value associated with `key` in `t`, or throws `Not_found` if there is no value associated with `key` in `t`.

```
‘b *@aqual(‘EQ3861)‘r lookup_opt(table_t<‘a,‘b,‘r>,
                                 ‘a key);
```

`lookup_opt(t,key)` returns a pointer to the value associated with `key` in `t`, or NULL if there is no value associated with `key`.

```
bool try_lookup(table_t<‘a,‘b>,‘a key,‘b @@aqual(‘EQ3870) data);
```

`try_lookup(t,key,p)` tries to find the `key` in the table `t`. If successful, it sets `*p` to the value associated with `key` and returns true. If the `key` is not found, then try_lookup returns false.

```
void resize(table_t<‘a,‘b>);
```

`resize(t)` increases the size (number of buckets) in table `t`. `resize` is called automatically by functions like `insert` when the buckets of a hash table get large, however, it can also be called by the programmer explicitly.

```
void remove(table_t<‘a,‘b>,‘a key);
```

`remove(t,key)` removes the most recent binding of `key` from `t`; the next-most-recent binding of `key` (if any) is restored. If there is no value associated with `key` in `t`, `remove` returns silently.

```
int hash_string(string_t);
```

`hash_string(s)` returns a hash of a string `s`. It is provided as a convenience for making hash tables mapping strings to values.

```
int hash_stringptr(stringptr_t);
```
`hash_stringptr(p)` returns a hash of a string pointer `p`.

```
void iter(void (@ f)('a,'b),table_t<'a,'b>);
```
`iter(f,t)` applies `f` to each key/value pair in `t`.

```
void iter_c(void (@ f)('a,'b,'c),table_t<'a,'b> t,'c env);
```
`iter_c(f,t,e)` calls `f(k,v,e)` for each key/value pair `(k,v)`.

C.11 <iter.h>

Defines namespace Iter, which implements imperative iterators over sets/sequences of elements.

```
typedef struct Iter<'a,'bd> iter_t<'a,'bd>;
```
A value of type `iter_t<'a,'bd>` is an iterator over elements of type `'a`.

```
bool next(iter_t<'a>,'a @@aqual('EQ4022));
```
If there is a next element, `next(i,p)` returns `true` and assigns the next element to `*p`. If there is no next element, `next(i,p)` returns `false` without assigning anything to `*p`.

C.12 <list.h>

Defines namespace List, which implements generic lists and various operations over them, following the conventions of the Objective Caml list library as much as possible.

```
struct List<'a::B,'r,'q::Q> {
  'a hd;
  struct List<'a,'r,'q> *@aqual('q)'r tl;
};
```
A `struct List` is a memory cell with a head field containing an element and a tail field that points to the rest of the list. Such a structure is traditionally called a cons cell. Note that every element of the list must have the same type `'a`, and every cons cell in the list must be allocated in the same region `'r`.

```
typedef struct List<‘a,‘r,‘q> *@aqual(‘q)‘r list_t<‘a,
                                                   ‘r,
                                                   ‘q>;
```

A `list_t` is a possibly-NULL pointer to a `struct List`. Most of the functions in namespace List operate on values of type `list_t` rather than `struct List`. Note that a `list_t` can be empty (NULL) but a `struct List` cannot.

```
typedef struct List<‘a,‘r,‘q> @@aqual(‘q)‘r List_t<‘a,
                                                   ‘r,
                                                   ‘q>;
```

A `List_t` is a non-NULL pointer to a `struct List`. This is used much less often than `list_t`, however it may be useful when you want to emphasize that a list has at least one element.

```
list_t<‘a> list(...  ‘a);
```

`list(x1,...,xn)` builds a heap-allocated list with elements `x1` through `xn`.

```
list_t<‘a,‘r> rlist(region_t<‘r>,...  ‘a);
```

`rlist(r, x1,...,xn)` builds a list with elements `x1` through `xn`, allocated in the region with handle `r`.

```
int length(list_t<‘a> x);
```

`length(x)` returns the number of elements in list `x`.

```
‘a hd(List_t<‘a> x);
```

`hd(x)` returns the first element of list `x`.

```
list_t<‘a,‘r> tl(List_t<‘a,‘r> x);
```

`tl(x)` returns the tail of list `x`.

```
list_t<‘a> copy(list_t<‘a> x);
```

`copy(x)` returns a new heap-allocated copy of list `x`.

```
list_t<‘a,‘r> rcopy(region_t<‘r>,list_t<‘a> x);
```

`rcopy(r,x)` returns a new copy of list `x`, allocated in the region with handle `r`.

```
list_t<'b> map('b (@ f)('a),list_t<'a> x);
```

If x has elements x1 through xn, then map(f,x) returns list(f(x1),...,f(xn)).

```
list_t<'b,'r> rmap(region_t<'r>,'b (@ f)('a),list_t<'a> x);
```

If x has elements x1 through xn, then rmap(r,f,x) returns rlist(r,f(x1),...,f(xn)

```
list_t<'b> map_c('b (@ f)('c,'a),'c env,list_t<'a> x);
```

map_c is a version of map where the function argument requires a closure as its first argument.

```
list_t<'b,'r> rmap_c(region_t<'r>,'b (@ f)('c,'a),'c env,
                     list_t<'a> x);
```

rmap_c is a version of rmap where the function argument requires a closure as its first argument.

```
datatype exn {
  List_mismatch
};
```

List_mismatch is thrown when two lists don't have the same length.

```
list_t<'c> map2('c (@ f)('a,'b),list_t<'a> x,list_t<'b> y);
```

If x has elements x1 through xn, and y has elements y1 through yn, then map2(f,x,y) returns a new heap-allocated list with elements f(x1,y1) through f(xn,yn). If x and y don't have the same number of elements, List_mismatch is thrown.

```
list_t<'c,'r> rmap2(region_t<'r>,'c (@ f)('a,'b),list_t<'a> x,
                    list_t<'b> y);
```

rmap2(r,f,x,y) is like map2(f,x,y), except that the resulting list is allocated in the region with handle r.

```
list_t<'d> map3('d (@ f)('a,'b,'c),list_t<'a> x,list_t<'b> y,
                list_t<'c> z);
```

If x has elements x1 through xn, y has elements y1 through yn, and z has elements z1 through zn, then map3(f,x,y,z) returns a new heap-allocated list with elements f(x1,y1,z1) through f(xn,yn,zn). If x, y, and z don't have the same number of elements, List_mismatch is thrown.

```
list_t<'d,'r> rmap3(region_t<'r>,'d (@ f)('a,'b,'c),
                    list_t<'a> x,list_t<'b> y,list_t<'c> z);
```

`rmap3(r,f,x,y,z)` is like `map3(f,x,y,z)`, except that the resulting list is allocated in the region with handle `r`.

```
void app('b (@ f)('a),list_t<'a> x);
```

`app(f,x)` applies `f` to each element of `x`, discarding the results. Note that `f` must not return `void`.

```
void app_c('c (@ f)('a,'b),'a,list_t<'b> x);
```

`app_c` is a version of `app` where the function argument requires a closure as its first argument.

```
void app2('c (@ f)('a,'b),list_t<'a> x,list_t<'b> y);
```

If `x` has elements `x1` through `xn`, and `y` has elements `y1` through `yn`, then `app2(f,x,y)` performs `f(x1,y1)` through `f(xn,yn)` and discards the results. If `x` and `y` don't have the same number of elements, `List_mismatch` is thrown.

```
void app2_c('d (@ f)('a,'b,'c),'a env,list_t<'b> x,
            list_t<'c> y);
```

`app2_c` is a version of `app2` where the function argument requires a closure as its first argument.

```
void iter(void (@ f)('a),list_t<'a> x);
```

`iter(f,x)` is like `app(f,x)`, except that `f` returns `void`.

```
void iter_c(void (@ f)('b,'a),'b env,list_t<'a> x);
```

`iter_c` is a version of `iter` where the function argument requires a closure as its first argument.

```
void iter2(void (@ f)('a,'b),list_t<'a> x,list_t<'b> y);
```

`iter2` is a version of `app2` where the function returns `void`.

```
void iter2_c(void (@ f)('a,'b,'c),'a env,list_t<'b> x,
             list_t<'c> y);
```

`iter2_c` is a version of `iter2` where the function argument requires a closure as its first argument.

```
'a fold_left('a (@ f)('a,'b),'a accum,list_t<'b> x);
```

If x has elements x1 through xn, then fold_left(f,accum,x) returns f(f(...(f(x2,f(x1,accum))),xn-1),xn).

```
'a fold_left_c('a (@ f)('c,'a,'b),'c,'a accum,list_t<'b> x);
```

fold_left_c is a version of fold_left where the function argument requires a closure as its first argument.

```
'b fold_right('b (@ f)('a,'b),list_t<'a> x,'b accum);
```

If x has elements x1 through xn, then fold_left(f,accum,x) returns f(x1,f(x2,...,f(xn-1,f(xn,a))...)).

```
'b fold_right_c('b (@ f)('c,'a,'b),'c,list_t<'a> x,
                'b accum);
```

fold_right_c is a version of fold_right where the function argument requires a closure as its first argument.

```
list_t<'a> revappend(list_t<'a,'r> x,list_t<'a,'H> y);
```

If x has elements x1 through xn, revappend(x,y) returns a list that starts with elements xn through x1, then continues with y. Cons cells for the first n elements are newly-allocated on the heap, and y must be allocated on the heap.

```
list_t<'a,'r> rrevappend(region_t<'r>,list_t<'a> x,
                         list_t<'a,'r> y);
```

rrevappend(r,x,y) is like revappend(x,y), except that y must be allocated in the region with handle r, and the result is allocated in the same region.

```
list_t<'a> rev(list_t<'a> x);
```

rev(x) returns a new heap-allocated list whose elements are the elements of x in reverse.

```
list_t<'a,'r> rrev(region_t<'r>,list_t<'a> x);
```

rrev(r,x) is like rev(x), except that the result is allocated in the region with handle r.

```
list_t<'a,'r> imp_rev(list_t<'a,'r> x);
```

`imp_rev(x)` imperatively reverses list `x` (the list is side-effected). Note that `imp_rev` returns a list. This is because the first cons cell of the result is the last cons cell of the input; a typical use is therefore `x = imp_rev(x)`.

```
list_t<'a> append(list_t<'a> x,list_t<'a,'H> y);
```

If `x` has elements `x1` through `xn`, `append(x,y)` returns a list that starts with elements `x1` through `xn`, then continues with `y`. Cons cells for the first `n` elements are newly-allocated on the heap, and `y` must be allocated on the heap.

```
list_t<'a,'r> rappend(region_t<'r>,list_t<'a> x,list_t<'a,
                                                 'r> y);
```

`rappend(r,x,y)` is like `append(x,y)`, except that `y` must be allocated in the region with handle `r`, and the result is allocated in the same region.

```
list_t<'a,'r> imp_append(list_t<'a,'r> x,list_t<'a,
                                               'r> y);
```

`imp_append(x,y)` modifies `x` to append `y` to it, destructively. Note that `imp_append` returns a list. This is because `x` might be NULL, in which case, `imp_append(x,y)` returns `y`; so a typical use would be `x = imp_append(x,y)`.

```
list_t<'a> flatten(list_t<list_t<'a,'H>> x);
```

In `flatten(x)`, `x` is a list of lists, and the result is a new heap-allocated list with elements from each list in `x`, in sequence. Note that `x` must be allocated on the heap.

```
list_t<'a,'r> rflatten(region_t<'r>,list_t<list_t<'a,
                                                 'r>> x);
```

`rflatten(r,x)` is like `flatten(x)`, except that the result is allocated in the region with handle `r`, and each element of `x` must be allocated in `r`.

```
list_t<'a> merge_sort(int (@ cmp)('a,'a),list_t<'a> x);
```

merge_sort(cmp,x) returns a new heap-allocated list whose elements are the elements of x in ascending order (according to the comparison function cmp), by the MergeSort algorithm.

```
list_t<'a,'r> rmerge_sort(region_t<'r>,int (@ cmp)('a,
                                                   'a),
                          list_t<'a> x);
```

rmerge_sort(r,x) is like merge_sort(x), except that the result is allocated in the region with handle r.

```
list_t<'a,'r> rimp_merge_sort(int (@ cmp)('a,'a),list_t<'a,
                                                        'r> x);
```

rimp_merge_sort is an imperative version of rmerge_sort: the list is sorted in place. rimp_merge_sort returns a list because the first cons cell of the sorted list might be different from the first cons cell of the input list; a typical use is x = rimp_merge_sort(cmp,x).

```
list_t<'a> merge(int (@ cmp)('a,'a),list_t<'a,'H> x,
                 list_t<'a,'H> y);
```

merge(cmp,x,y) returns the merge of two sorted lists, according to the cmp function.

```
list_t<'a,'r3> rmerge(region_t<'r3>,int (@ cmp)('a,
                                                'a),
                      list_t<'a> a,list_t<'a> b);
```

rmerge(r,cmp,x,y) is like merge(cmp,x,y), except that x, y, and the result are allocated in the region with handle r.

```
list_t<'a,'r> imp_merge(int (@ cmp)('a,'a),list_t<'a,
                                                  'r> a,
                        list_t<'a,'r> b);
```

imp_merge is an imperative version of merge.

```
datatype exn {
  Nth
};
```

Nth is thrown when nth doesn't have enough elements in the list.

```
'a nth(list_t<'a> x,int n);
```

If x has elements x0 through xm, and 0<=n<=m, then nth(x,n) returns xn. If n is out of range, Nth is thrown. Note that the indexing is 0-based.

```
list_t<'a,'r> nth_tail(list_t<'a,'r> x,int i);
```

If x has elements x0 through xm, and 0<=n<=m, then nth(x,n) returns the list with elements xn through xm. If n is out of range, Nth is thrown.

```
bool forall(bool (@ pred)('a),list_t<'a> x);
```

forall(pred,x) returns true if pred returns true when applied to every element of x, and returns false otherwise.

```
bool forall_c(bool (@ pred)('a,'b),'a env,list_t<'b> x);
```

forall_c is a version of forall where the function argument requires a closure as its first argument.

```
bool exists(bool (@ pred)('a),list_t<'a> x);
```

exists(pred,x) returns true if pred returns true when applied to some element of x, and returns false otherwise.

```
bool exists_c(bool (@ pred)('a,'b),'a env,list_t<'b> x);
```

exists_c is a version of exists where the function argument requires a closure as its first argument.

```
'c *@aqual('EQ4198)'r find_c('c *@aqual('EQ4201)'r (@ pred)('a,
                                                            'b),
                             'a env,list_t<'b> x);
```

find_c iterates over the given list and returns the first element for which pred does not return NULL. Otherwise it returns NULL.

```
list_t<$('a,'b) @@aqual('EQ4205),'H> zip(list_t<'a> x,
                                         list_t<'b> y);
```

If x has elements x1 through xn, and y has elements y1 through yn, then zip(x,y) returns a new heap-allocated array with elements &$(x1,y1) through &$(xn,yn). If x and y don't have the same number of elements, List_mismatch is thrown.

```
list_t<$(‘a,‘b) @@aqual(‘EQ4208)‘r2,‘r1> rzip(region_t<‘r1> r1,
                                              region_t<‘r2> r2,
                                              list_t<‘a> x,
                                              list_t<‘b> y);
```

`rzip(r1,r2,x,y)` is like `zip(x,y)`, except that the list returned is allocated in the region with handle `r1`, and the pairs of that list are allocated in the region with handle `r2`.

```
list_t<$(‘a,‘b,‘c) @@aqual(‘EQ4211),‘H> zip3(list_t<‘a> x,
                                             list_t<‘b> y,
                                             list_t<‘c> z);
```

If `x` has elements `x1` through `xn`, and `y` has elements `y1` through `yn`, and `z` has elements `z1` through `zn`, then `zip3(x,yz)` returns a new heap-allocated array with elements `&$(x1,y1,z1)` through `&$(xn,yn,zn)`. If `x` and `y` don't have the same number of elements, `List_mismatch` is thrown.

```
list_t<$(‘a,‘b,‘c) @@aqual(‘EQ4214)‘r2,‘r1> rzip3(region_t<‘r1> r1,
                                                  region_t<‘r2> r2,
                                                  list_t<‘a> x,
                                                  list_t<‘b> y,
                                                  list_t<‘c> z);
```

`rzip3(r1,r2,x,y)` is like `zip3(x,y)`, except that the list returned is allocated in the region with handle `r1`, and the pairs of that list are allocated in the region with handle `r2`.

```
$(list_t<‘a>,list_t<‘b>) split(list_t<$(‘a,‘b) @@aqual(‘EQ4218)> x);
```

If `x` has elements `&$(a1,b1)` through `&$(an,bn)`, then `split(x)` returns a pair of new heap-allocated arrays with elements `a1` through `an`, and `b1` through `bn`.

```
$(list_t<‘a>,list_t<‘b>,list_t<‘c>) split3(list_t<$(‘a,
                                                    ‘b,
                                                    ‘c) @@aqual(‘EQ422
```

If `x` has elements `&$(a1,b1,c1)` through `&$(an,bn,cn)`, then `split(x)` returns a triple of new heap-allocated arrays with elements `a1` through `an`, and `b1` through `bn`, and `c1` through `cn`.

```
$(list_t<‘a,‘r1>,list_t<‘b,‘r2>) rsplit(region_t<‘r1> r1,
                                        region_t<‘r2> r2,
                                        list_t<$(‘a,
                                                 ‘b) @@aqual(‘EQ4226)>
```

`rsplit(r1,r2,x)` is like `split(x)`, except that the first list returned is allocated in the region with handle `r1`, and the second list returned is allocated in the region with handle `r2`.

```
$(list_t<‘a,‘r3>,list_t<‘b,‘r4>,list_t<‘c,‘r5>) rsplit3(region_t<‘r3>
                                                        region_t<‘r4>
                                                        region_t<‘r5>
                                                        list_t<$(‘a,
                                                                 ‘b,
                                                                 ‘c) @
```

`rsplit(r1,r2,r3,x)` is like `split3(x)`, except that the first list returned is allocated in the region with handle `r1`, the second list returned is allocated in the region with handle `r2`, and the third list returned is allocated in the region with handle `r3`.

```
bool memq(list_t<‘a> l,‘a x);
```

`memq(l,x)` returns true if `x` is == an element of list `l`, and returns false otherwise.

```
bool mem(int (@ compare)(‘a,‘a),list_t<‘a> l,‘a x);
```

`mem(cmp,l,x)` is like `memq(l,x)` except that the comparison function `cmp` is used to determine if `x` is an element of `l`. `cmp(a,b)` should return 0 if `a` is equal to `b`, and return a non-zero number otherwise.

```
‘b assoc(list_t<$(‘a,‘b) @@aqual(‘EQ4235)> l,‘a k);
```

An association list is a list of pairs where the first element of each pair is a key and the second element is a value; the association list is said to map keys to values. `assoc(l,k)` returns the first value paired with key `k` in association list `l`, or throws `Core::Not_found` if `k` is not paired with any value in `l`. `assoc` uses == to decide if `k` is a key in `l`.

```
‘b assoc_cmp(int (@ cmp)(‘a,‘c),list_t<$(‘a,‘b) @@aqual(‘EQ4239)> l,
             ‘c x);
```

`assoc_cmp(cmp,l,k)` is like `assoc(l,k)` except that the compari-

son function `cmp` is used to decide if `k` is a key in `l`. `cmp` should return 0 if two keys are equal, and non-zero otherwise.

```
bool mem_assoc(list_t<$(‘a,‘b) @@aqual(‘EQ4244)> l,
               ‘a x);
```

`mem_assoc(l,k)` returns true if `k` is a key in association list `l` (according to ==).

```
bool mem_assoc_cmp(int (@ cmp)(‘a,‘c),list_t<$(‘a,‘b) @@aqual(‘EQ4248)
                   ‘c x);
```

Same as `mem_assoc`, but uses comparison function `cmp` rather than pointer equality ==.

```
list_t<‘a,‘r> delete(list_t<‘a,‘r> l,‘a x);
```

`delete(l,k)` returns the list with the first occurence of `x` removed from it, if `x` was in the list; otherwise raises `Core::Not_found`. Side-effects original list `l`.

```
list_t<‘a,‘r> delete_cmp(int (@ cmp)(‘a,‘a),list_t<‘a,
                                                   ‘r> l,
                         ‘a x);
```

`delete(l,k)` returns the list with the first `e` in the list such that `cmp(x,e)` == 0. If no such `e` exists, raises `Core::Not_found`. Side-effects original list `l`.

```
Core::opt_t<‘c> check_unique(int (@ cmp)(‘c,‘c),list_t<‘c> x);
```

`check_unique(cmp,x)` checks whether the sorted list `x` has duplicate elements, according to `cmp`. If there are any duplicates, one will be returned; otherwise, NULL is returned.

```
‘a ?@aqual(‘EQ4254) to_array(list_t<‘a> x);
```

`to_array(x)` returns a new heap-allocated array with the same elements as list `x`.

```
‘a ?@aqual(‘EQ4257)‘r rto_array(region_t<‘r> r,list_t<‘a> x);
```

`rto_array(r,x)` is like `to_array(x)`, except that the resulting array is allocated in the region with handle `r`.

```
list_t<'a> from_array('a ?@aqual('EQ4261) arr);
```

`from_array(x)` returns a new heap-allocated list with the same elements as array `x`.

```
list_t<'a,'r> rfrom_array(region_t<'r> r,'a ?@aqual('EQ4265) arr);
```

`rfrom_array(r,x)` is like `from_array(x)`, except that the resulting list is allocated in the region with handle `r`.

```
int list_cmp(int (@ cmp)('a,'b),list_t<'a> l1,list_t<'b> l2);
```

`list_cmp(cmp,l1,l2)` is a comparison function on lists, parameterized by a comparison function `cmp` on list elements.

```
bool list_prefix(int (@ cmp)('a,'b),list_t<'a> l1,list_t<'b> l2);
```

`list_prefix(cmp,l1,l2)` returns true if `l1` is a prefix of `l2`, using `cmp` to compare the elements of `l1` and `l2`.

```
list_t<'a> filter(bool (@ f)('a),list_t<'a> x);
```

`filter(f,x)` returns a new heap-allocated list whose elements are the elements of `x` on which `f` returns true, in order.

```
list_t<'a> filter_c(bool (@ f)('b,'a),'b env,list_t<'a> x);
```

`filter_c` is a version of `filter` where the function argument requires a closure as its first argument.

```
list_t<'a,'r> rfilter(region_t<'r> r,bool (@ f)('a),
                      list_t<'a> x);
```

`rfilter_c(r,f,x)` is like `filter_c(f,x)`, except that the resulting list is allocated in the region with handle `r`.

```
list_t<'a,'r> rfilter_c(region_t<'r> r,bool (@ f)('b,
                                                'a),
                        'b env,list_t<'a> x);
```

`rfilter_c` is a version of `rfilter` where the function argument requires a closure as its first argument.

C.13 <pp.h>

Defines a namespace PP that has functions for implementing pretty printers. Internally, PP is an implementation of Kamin's version of Wadler's pretty printing combinators, with some extensions for doing hyperlinks in Tk text widgets.

All of the internal data structures used by PP are allocated on the heap.

```
typedef struct Doc @@aqual('EQ4879) doc_t;
```

A value of type `doc_t` is a "document" that can be combined with other documents, formatted at different widths, converted to strings or files.

```
void file_of_doc(doc_t d,int w,FILE @@aqual('EQ4883) f);
```

`file_of_doc(d,w,f)` formats `d` to width `w`, and prints the formatted output to `f`.

```
string_t string_of_doc(doc_t d,int w);
```

`string_of_doc(d,w)` formats `d` to width `w`, and returns the formatted output in a heap-allocated string.

```
$(string_t,list_t<$(int,int,int,string_t) @@aqual('EQ4887)>) @@aqual(
```

`string_and_links(d,w)` formats `d` to width `w`, returns the formatted output in a heap-allocated string, and returns in addition a list of hyperlinks. Each hyperlink has the form `$(line,char,length,contents)`, where `line` and `char` give the line and character in the formatted output where the hyperlink starts, `length` gives the number of characters of the hyperlink, and `contents` is a string that the hyperlink should point to. The `line`, `char`, and `length` are exactly what is needed to create a hyperlink in a Tk text widget.

```
doc_t nil_doc();
```

`nil_doc()` returns an empty document.

```
doc_t blank_doc();
```

`blank_doc()` returns a document consisting of a single space character.

`doc_t` **`line_doc`**`();`

`line_doc()` returns a document consisting of a single line break.

`doc_t` **`oline_doc`**`();`

`oline_doc()` returns a document consisting of an optional line break; when the document is formatted, the pretty printer will decide whether to break the line.

`doc_t` **`text`**`(string_t<'H> s);`

`text(s)` returns a document containing exactly the string `s`.

`doc_t` **`textptr`**`(stringptr_t<'H> p);`

`textptr(p)` returns a documents containing exactly the string pointed to by `p`.

`doc_t` **`text_width`**`(string_t<'H> s,int w);`

`text_width(s,w)` returns a document containing exactly the string `s`, which is assumed to have `w` characters. This is useful when `s` contains markup character that don't take up space when printed, e.g., instructions for making text bold.

`doc_t` **`hyperlink`**`(string_t<'H> shrt,string_t<'H> full);`

`hyperlink(shrt,full)` returns a document that will be formatted as the string `shrt` linked to the string `full`.

`doc_t` **`nest`**`(int k,doc_t d);`

`nest(k,d)` returns a document that will be formatted like document `d`, but indented by `k` spaces.

`doc_t` **`cat`**`(... doc_t);`

`cat(d1, d2, ..., dn)` returns a document consisting of document `d1` followed by `d2`, and so on up to `dn`.

`doc_t` **`cats`**`(list_t<doc_t,'H> doclist);`

`cats(l)` returns a document containing all of the documents in list `l`, in order.

```
doc_t cats_arr(doc_t ?@aqual('EQ4894) docs);
```

cats_arr(a) returns a document containing all of the documents in array a, in order.

```
doc_t doc_union(doc_t d1,doc_t d2);
```

doc_union(d1,d2) does ?? FIX.

```
doc_t tab(doc_t d);
```

tab(d) returns a document formatted like d but indented by a tab stop.

```
doc_t seq(string_t<'H> sep,list_t<doc_t,'H> l);
```

seq(sep,l) returns a document consisting of each document of l, in sequence, with string sep between each adjacent element of l.

```
doc_t ppseq(doc_t (@ pp)('a),string_t<'H> sep,list_t<'a> l);
```

ppseq is a more general form of seq: in ppseq(pp,sep,l), l is a list of values to pretty print in sequence, pp is a function that knows how to pretty print a value, and sep is a string to print between each value.

```
doc_t seql(string_t<'H> sep,list_t<doc_t,'H> l0);
```

seql is like seq, except that the resulting document has line breaks after each separator.

```
doc_t ppseql(doc_t (@ pp)('a),string_t<'H> sep,list_t<'a> l);
```

ppseql is like ppseq, except that the resulting document has line breaks after each separator.

```
doc_t group(string_t<'H> start,string_t<'H> stop,string_t<'H> sep,
            list_t<doc_t,'H> l);
```

group(start,stop,sep,l) is like cat(text(start), seq(sep,l), text(stop)).

```
doc_t groupl(string_t<'H> start,string_t<'H> stop,string_t<'H> sep,
             list_t<doc_t,'H> l);
```

groupl is like group but a line break is inserted after each separator.

```
doc_t egroup(string_t<‘H> start,string_t<‘H> stop,string_t<‘H> sep,
             list_t<doc_t,‘H> l);
```

egroup is like group, except that the empty document is returned if the list is empty.

C.14 <queue.h>

Defines namespace Queue, which implements generic imperative queues and various operations following the conventions of the Objective Caml queue library as much as possible.

```
typedef struct Queue<‘a,‘r,‘q> @@aqual(‘q)‘r queue_t<‘a,
                                                    ‘r,
                                                    ‘q>;
```

A value of type queue_t<‘a,‘r> is a first-in, first-out queue of elements of type ‘a; the queue data structures are allocated in region ‘r.

```
bool is_empty(queue_t<‘a>);
```

is_empty(q) returns true if q contains no elements, and returns false otherwise.

```
queue_t<‘a> create();
```

create() allocates a new, empty queue on the heap and returns it.

```
void add(queue_t<‘a,‘H>,‘a x) __attribute__((consume(2)));
```

add(q,x) adds x to the end of q (by side effect).

```
void radd(region_t<‘r>,queue_t<‘a,‘r>,‘a x:RESTRICTED >= aquals(‘a)) _
```

radd(r,q,x) is like add(q,x) except that the queue lives in the region with handle r.

```
void push(queue_t<‘a,‘H> q,‘a x) __attribute__((consume(2)));
```

push(q,x) adds x to the front of q (by side effect).

```
void rpush(region_t<‘r> r,queue_t<‘a,‘r> q,‘a x) __attribute__((consum
```

rpush(r,q,x) is like push(q,x) except that the queue lives in the region with handle r.

```
datatype exn {
  Empty
};
```

`Empty` is an exception raised by `take` and `peek`.

```
'a take(queue_t<'a>);
```

`take(q)` removes the element from the front on `q` and returns it; if `q` is empty, exception `Empty` is thrown.

```
'a peek(queue_t<'a>);
```

`peek(q)` returns the element at the front of `q`, without removing it from `q`. If `q` is empty, exception `Empty` is thrown.

```
void clear(queue_t<'a>);
```

`clear(q)` removes all elements from `q`.

```
int length(queue_t<'a>);
```

`length(q)` returns the number of elements in `q`.

```
void iter(void (@ f)('a),queue_t<'a>);
```

`iter(f,q)` applies `f` to each element of `q`, from first to last. Note that `f` must return `void`.

```
void app('b (@ f)('a),queue_t<'a>);
```

`app(f,q)` applies `f` to each element of `q`, from first to last. Note that `f` must return a value of kind `M`.

The following procedures are specialized to work with no-aliasable and/or unique pointers.

```
'a *@aqual('EQ5156)'U take_match(region_t<'r> r,queue_t<'a *@aqual('EQ
                                                        'r> q,
                                 bool (@@aqual('EQ5163) f)('b,
                                                           'a *@aqual(
                                 'b env);
```

`take_match(r,q,f,c)` looks through the queue (starting from the front) and returns the element `x` for which `f(x,c)` returns true.

```
'a noalias_take(queue_t<'a> q,'a null_elem) __attribute__((consume(2)
```

`noalias_take(q)` is as take, above, but works when the queue contains potentially-unique elements; the caller needs to supply a 'null' element to swap with the element in the first spot in the queue.

```
'a *@aqual('EQ5169)'U ptr_take(queue_t<'a *@aqual('EQ5172)'U> q);
```

`ptr_take(q)` is a wrapper for `noalias_take(q,NULL)`.

C.15 <rope.h>

Defines namespace Rope, which implements character arrays that can be concatenated in constant time.

```
typedef struct Rope_node @@aqual('EQ5429) rope_t;
```

A value of type `rope_t` is a character array that can be efficiently concatenated.

```
rope_t from_string(string_t<'H>);
```

`from_string(s)` returns a rope that has the same characters as string `s`. Note that `s` must be heap-allocated.

```
mstring_t to_string(rope_t);
```

`to_string(r)` returns a new, heap-allocated string with the same characters as rope `r`.

```
rope_t concat(rope_t,rope_t);
```

`concat(r1,r2)` returns a rope whose characters are the characters of `r1` followed by the characters of `r2`.

```
rope_t concata(rope_t ?@aqual('EQ5432));
```

`concata(a)` returns a rope that contains the concatenation of the characters in the array `a` of ropes.

```
rope_t concatl(List::list_t<rope_t>);
```

`concata(l)` returns a rope that contains the concatenation of the characters in the list `l` of ropes.

```
unsigned int length(rope_t);
```

`length(r)` returns the number of characters in the rope `r`, up to but not including the first NUL character.

```
int cmp(rope_t,rope_t);
```

`cmp(r1,r2)` is a comparison function on ropes: it returns a number less than, equal to, or greater than 0 according to whether the character array of `r1` is lexicographically less than, equal to, or greater than the character array of `r2`.

C.16 <set.h>

Defines namespace Set, which implements polymorphic, functional, finite sets over elements with a total order, following the conventions of the Objective Caml set library as much as possible. Sets can also be used imperatively, but choosing the imp_ variations of functions, but unioning and intersecting imperative sets should be done with caution.

```
typedef struct Set<'a,'r> @@aqual('EQ5699)'r set_t<'a,
                                                   'r>;
```

A value of type `set_t<'a,'r>` is a set with elements of type `'a`. The data structures used to implement the set (not the elements of the set!) are in region `'r`.

The set creation functions require a comparison function as an argument. The comparison function should return a number less than, equal to, or greater than 0 according to whether its first argument is less than, equal to, or greater than its second argument.

```
set_t<'a> empty(int (@@aqual('EQ5702) cmp)('a,'a));
```

`empty(cmp)` creates an empty set given comparison function `cmp`. The set is heap-allocated.

```
set_t<'a,'r> rempty(region_t<'r> r,int (@@aqual('EQ5705) cmp)('a,
                                                          'a));
```

`rempty(r,cmp)` creates an empty set in the region with handle `r`.

```
set_t<‘a> singleton(int (@@aqual(‘EQ5708) cmp)(‘a,‘a),
                    ‘a x);
```

`singleton(cmp,x)` creates a set on the heap with a single element, `x`.

```
set_t<‘a> from_list(int (@@aqual(‘EQ5711) cmp)(‘a,‘a),
                    list_t<‘a> l);
```

`from_list(cmp,l)` creates a set on the heap; the elements of the set are the elements of the list `l`.

```
list_t<‘a,‘r> to_list(set_t<‘a,‘r>);
```

`to_list(s)` returns a list in sorted order of the elements.

```
set_t<‘a> insert(set_t<‘a,‘H> s,‘a elt);
```

`insert(s,elt)` returns a set containing all the elements of `s`, plus `elt`. The set `s` is not modified.

```
void imp_insert(set_t<‘a,‘H> s,‘a elt);
```

`imp_insert(s,elt)` returns modifies `s` to additionally contain `elt`, if not already present.

```
set_t<‘a,‘r> rinsert(region_t<‘r> r,set_t<‘a,‘r> s,
                     ‘a elt);
```

`rinsert(r,s,elt)` is like `insert(s,elt)`, except that it works on sets allocated in the region with handle `r`.

```
void imp_rinsert(region_t<‘r> r,set_t<‘a,‘r> s,‘a elt);
```

`imp_rinsert(r,s,elt)` is like `imp_insert(s,elt)`, except that it works on sets allocated in the region with handle `r`.

```
set_t<‘a> union_two(set_t<‘a,‘H> s1,set_t<‘a,‘H> s2);
```

`union_two(s1,s2)` returns a set whose elements are the union of the elements of `s1` and `s2`. (We use the name `union_two` because `union` is a keyword in Cyclone.)

```
set_t<‘a> intersect(set_t<‘a,‘H> s1,set_t<‘a,‘H> s2);
```

`intersect(s1,s2)` returns a set whose elements are the intersection of the elements of `s1` and `s2`.

```
set_t<'a> diff(set_t<'a,'H> s1,set_t<'a,'H> s2);
```

`diff(s1,s2)` returns a set whose elements are the elements of `s1` that are not members of `s2`.

```
set_t<'a> delete(set_t<'a,'H> s,'a elt);
```

`delete(s,elt)` returns a set whose elements are the elements of `s`, minus `elt`.

```
'a imp_delete(set_t<'a,'H> s,'a elt);
```

`imp_delete(s,elt)` imperatively deletes `elt` from `s`, if present. returns the element (in case the element in the set differs from `elt` due to how the comparison function was specified).

```
int cardinality(set_t s);
```

`cardinality(s)` returns the number of elements in the set `s`.

```
bool is_empty(set_t s);
```

`is_empty(s)` returns true if `s` has no members, and returns false otherwise.

```
bool member(set_t<'a> s,'a elt);
```

`member(s,elt)` returns true if `elt` is a member of `s`, and returns false otherwise.

```
bool subset(set_t<'a> s1,set_t<'a> s2);
```

`subset(s1,s2)` returns true if `s1` is a subset of `s2`, and returns false otherwise.

```
int setcmp(set_t<'a> s1,set_t<'a> s2);
```

`setcmp(s1,s2)` returns a number less than, equal to, or greater than 0 according to whether `s1` is less than, equal to, or greater than `s2` in the subset order.

```
bool equals(set_t<'a> s1,set_t<'a> s2);
```

`equals(s1,s2)` returns true if `s1` equals `s2` have the same elements, and returns false otherwise.

```
'b fold('b (@ f)('a,'b),set_t<'a> s,'b accum);
```

If `s` is a set with elements `x1` through `xn`, then `fold(f,s,accum)` returns `f(x1,f(x2,f(...,f(xn,accum)...)))`.

```
'b fold_c('b (@ f)('c,'a,'b),'c env,set_t<'a> s,'b accum);
```

`fold_c(f,env,s,accum)` is like `fold`, except that the function `f` takes an extra (closure) argument, `env`.

```
void app('b (@ f)('a),set_t<'a> s);
```

`app(f,s)` applies `f` to each element of `s`, in no particular order; the result of the application is discarded. Notice that `f` cannot return `void`; use `iter` instead of `app` for that.

```
void iter(void (@ f)('a),set_t<'a> s);
```

`iter(f,s)` is like `app(f,s)`, except that `f` must return `void`.

```
void iter_c(void (@ f)('c,'a),'c env,set_t<'a> s);
```

`iter_c` is a version of `iter` where the function argument `f` requires a closure.

```
datatype exn {
  Absent
};
```

`Absent` is an exception thrown by the `choose` function.

```
'a choose(set_t<'a> s);
```

`choose(s)` returns some element of the set `s`; if the set is empty, `choose` throws `Absent`.

```
Iter::iter_t<'a,regions('a)+'r1+'r2> make_iter(region_t<'r1> rgn,
                                               set_t<'a,
                                                     'r2> s);
```

`make_iter(s)` returns an iterator over the set `s`; a constant amount of space is allocated in `rgn`.

C.17 <slowdict.h>

Defines namespace SlowDict, which implements polymorphic, functional, finite maps whose domain must have a total order. We follow the conventions of the Objective Caml Dict library as much as possible.

The basic functionality is the same as Dict, except that SlowDict supports `delete_present`; but region support still needs to be added, and some functions are missing, as well.

```
typedef struct Dict<'a,'b> @@aqual('EQ5973) dict_t<'a,
                                                   'b>;
```

A value of type `dict_t<'a,'b>` is a dictionary that maps keys of type `'a` to values of type `'b`.

```
datatype exn {
  Present
};
```

`Present` is thrown when a key is present but not expected.

```
datatype exn {
  Absent
};
```

`Absent` is thrown when a key is absent but should be present.

```
dict_t<'a,'b> empty(int (@@aqual('EQ5976) cmp)('a,'a));
```

`empty(cmp)` returns an empty dictionary, allocated on the heap. `cmp` should be a comparison function on keys: `cmp(k1,k2)` should return a number less than, equal to, or greater than 0 according to whether `k1` is less than, equal to, or greater than `k2` in the ordering on keys.

```
bool is_empty(dict_t d);
```

`is_empty(d)` returns true if `d` is empty, and returns false otherwise.

```
bool member(dict_t<'a> d,'a k);
```

`member(d,k)` returns true if `k` is mapped to some value in `d`, and returns false otherwise.

```
dict_t<'a,'b> insert(dict_t<'a,'b> d,'a k,'b v);
```

`insert(d,k,v)` returns a dictionary with the same mappings as `d`, except that `k` is mapped to `v`. The dictionary `d` is not modified.

```
dict_t<‘a,‘b> insert_new(dict_t<‘a,‘b> d,‘a k,‘b v);
```

`insert_new(d,k,v)` is like `insert(d,k,v)`, except that it throws `Present` if `k` is already mapped to some value in `d`.

```
dict_t<‘a,‘b> inserts(dict_t<‘a,‘b> d,list_t<$(‘a,‘b) @@aqual(‘EQ5980)
```

`inserts(d,l)` inserts each key, value pair into `d`, returning the resulting dictionary.

```
dict_t<‘a,‘b> singleton(int (@@aqual(‘EQ5983) cmp)(‘a,
                                                   ‘a),
                        ‘a k,‘b v);
```

`singleton(cmp,k,v)` returns a new heap-allocated dictionary with a single mapping, from `k` to `v`.

```
‘b lookup(dict_t<‘a,‘b> d,‘a k);
```

`lookup(d,k)` returns the value associated with key `k` in `d`, or throws `Absent` if `k` is not mapped to any value.

```
Core::opt_t<‘b> lookup_opt(dict_t<‘a,‘b> d,‘a k);
```

`lookup_opt(d,k)` returns `NULL` if `k` is not mapped to any value in `d`, and returns a non-NULL, heap-allocated option containing the value `k` is mapped to in `d` otherwise.

```
dict_t<‘a,‘b> delete(dict_t<‘a,‘b> d,‘a k);
```

`delete(d,k)` returns a dictionary with the same bindings as `d`, except that any binding of `k` is removed. The resulting dictionary is allocated on the heap.

```
dict_t<‘a,‘b> delete_present(dict_t<‘a,‘b> d,‘a k);
```

`delete_present(d,k)` is like `delete(d,k)`, except that `Absent` is thrown if `k` has no binding in `d`.

```
‘c fold(‘c (@ f)(‘a,‘b,‘c),dict_t<‘a,‘b> d,‘c accum);
```

If `d` has keys `k1` through `kn` mapping to values `v1` through `vn`, then `fold(f,d,accum)` returns `f(k1,v1,...f(kn,vn,accum)...)`.

```
'c fold_c('c (@ f)('d,'a,'b,'c),'d env,dict_t<'a,'b> d,
          'c accum);
```

`fold_c(f,env,d,accum)` is like `fold(f,d,accum)` except that `f` takes closure `env` as its first argument.

```
void app('c (@ f)('a,'b),dict_t<'a,'b> d);
```

`app(f,d)` applies `f` to every key/value pair in `d`; the results of the applications are discarded. Note that `f` cannot return `void`.

```
void app_c('c (@ f)('d,'a,'b),'d env,dict_t<'a,'b> d);
```

`app_c(f,env,d)` is like `app(f,d)` except that `f` takes closure `env` as its first argument.

```
void iter(void (@ f)('a,'b),dict_t<'a,'b> d);
```

`iter(f,d)` is like `app(f,d)` except that `f` returns `void`.

```
void iter_c(void (@ f)('c,'a,'b),'c env,dict_t<'a,'b> d);
```

`iter_c(f,env,d)` is like `app_c(f,env,d)` except that `f` returns `void`.

```
dict_t<'a,'c> map('c (@ f)('b),dict_t<'a,'b> d);
```

`map(f,d)` applies `f` to each value in `d`, and returns a new dictionary with the results as values: for every binding of a key `k` to a value `v` in `d`, the result binds `k` to `f(v)`. The returned dictionary is allocated on the heap.

```
dict_t<'a,'c> map_c('c (@ f)('d,'b),'d env,dict_t<'a,
                                                'b> d);
```

`map_c(f,env,d)` is like `map(f,d)` except that `f` takes a closure `env` as its first argument.

```
$('a,'b) @@aqual('EQ5995) choose(dict_t<'a,'b> d);
```

`choose(d)` returns a key/value pair from `d`; if `d` is empty, `Absent` is thrown. The resulting pair is allocated on the heap.

```
list_t<$('a,'b) @@aqual('EQ5999)> to_list(dict_t<'a,
                                              'b> d);
```

`to_list(d)` returns a list of the key/value pairs in `d`, allocated on the heap.

C.18 <xarray.h>

Defines namespace Xarray, which implements a datatype of extensible arrays.

```
typedef struct Xarray<'a> @@aqual('EQ6005)'r xarray_t<'a,
                                                      'r>;
```

An `xarray_t` is an extensible array.

```
int length(xarray_t<'a>);
```

`length(a)` returns the length of extensible array `a`.

```
'a get(xarray_t<'a>,int);
```

`get(a,n)` returns the `n`th element of `a`, or throws `Invalid_argument` if `n` is out of range.

```
void set(xarray_t<'a>,int,'a);
```

`set(a,n,v)` sets the `n`th element of `a` to `v`, or throws `Invalid_argument` if `n` is out of range.

```
xarray_t<'a> create(int,'a);
```

`create(n,v)` returns a new extensible array with starting size `n` and default value `v`.

```
xarray_t<'a,'r> rcreate(region_t<'r>,int,'a);
```

`rcreate(r,n,v)` returns a new extensible array with starting size `n` and default value `v` in region `r`.

```
xarray_t<'a> create_empty();
```

`create_empty()` returns a new extensible array with starting size 0.

```
xarray_t<'a,'r> rcreate_empty(region_t<'r>);
```

`rcreate_empty(r)` returns a new extensible array with starting size 0 in region `r`.

```
xarray_t<'a> singleton(int,'a);
```

`singleton(n,v)` returns a new extensible array with a single element `v`.

```
xarray_t<‘a,‘r> rsingleton(region_t<‘r>,int,‘a);
```
rsingleton(r,n,v) returns a new extensible array with a single element v in region r.

```
void add(xarray_t<‘a>,‘a);
```
add(a,v) makes the extensible array larger by adding v to the end.

```
int add_ind(xarray_t<‘a>,‘a);
```
add_ind(a,v) makes a larger by adding v to the end, and returns v.

```
‘a ?@aqual(‘EQ6009) to_array(xarray_t<‘a>);
```
to_array(a) returns a normal (non-extensible) array with the same elements as a.

```
‘a ?@aqual(‘EQ6012)‘r rto_array(region_t<‘r>,xarray_t<‘a>);
```
rto_array(a,r) returns a normal (non-extensible) array with the same elements as a allocated in region r.

```
xarray_t<‘a> from_array(‘a ?@aqual(‘EQ6016) arr);
```
from_array(a) returns an extensible array with the same elements as the normal (non-extensible) array a.

```
xarray_t<‘a,‘r> rfrom_array(region_t<‘r>,‘a ?@aqual(‘EQ6020) arr);
```
rfrom_array(r,a) returns an extensible array with the same elements as the normal (non-extensible) array a, allocated in region r.

```
xarray_t<‘a> append(xarray_t<‘a>,xarray_t<‘a>);
```
append(a1,a2) returns a new extensible array whose elements are the elements of a1 followed by a2. The inputs a1 and a2 are not modified.

```
xarray_t<‘a,‘r> rappend(region_t<‘r>,xarray_t<‘a>,xarray_t<‘a>);
```
rappend(r,a1,a2) returns a new extensible array whose elements are the elements of a1 followed by a2, allocated in region r. The inputs a1 and a2 are not modified.

```
void app('b (@ f)('a),xarray_t<'a>);
```

app(f,a) applies f to each element of a, in order from lowest to highest. Note that f returns 'a, unlike with iter.

```
void app_c('b (@ f)('c,'a),'c,xarray_t<'a>);
```

app_c(f,e,a) applies f to e and each element of a, in order from lowest to highest.

```
void iter(void (@ f)('a),xarray_t<'a>);
```

iter(f,a) applies f to each element of a, in order from lowest to highest. Note that f returns void, unlike with app.

```
void iter_c(void (@ f)('b,'a),'b,xarray_t<'a>);
```

iter_c(f,e,a) applies f to e and each element of a, in order from lowest to highest.

```
xarray_t<'b> map('b (@ f)('a),xarray_t<'a>);
```

map(f,a) returns a new extensible array whose elements are obtained by applying f to each element of a.

```
xarray_t<'b,'r> rmap(region_t<'r>,'b (@ f)('a),xarray_t<'a>);
```

rmap(r,f,a) returns a new extensible array whose elements are obtained by applying f to each element of a, and allocated in region r.

```
xarray_t<'b> map_c('b (@ f)('c,'a),'c,xarray_t<'a>);
```

map_c(f,e,a) returns a new extensible array whose elements are obtained by applying f to e and each element of a.

```
xarray_t<'b,'r> rmap_c(region_t<'r>,'b (@ f)('c,'a),
                       'c,xarray_t<'a>);
```

rmap_c(r,f,e,a) returns a new extensible array whose elements are obtained by applying f to e and each element of a. The result is allocated in region r.

```
void reuse(xarray_t<'a> xarr);
```

reuse(a) sets the number of elements of a to zero, but does not free the underlying array.

`void delete(xarray_t<'a> xarr,int num);`

`delete(a,n)` deletes the last `n` elements of `a`.

`void remove(xarray_t<'a> xarr,int i);`

`remove(a,i)` removes the element at position `i` from `a`; elements at positions greater than `i` are moved down one position.

D Grammar

This section is out-of-date.

The grammar of Cyclone is derived from ISO C99. It has the following additional keywords: `abstract`, `alias`, `as`, `calloc`, `catch`, `datatype`, `dynregion_t`, `export`, `fallthru`, `inject`, `let`, `malloc`, `namespace`, `new`, `NULL`, `numelts`, `offsetof`, `rcalloc`, `region_t`, `region`, `regions`, `reset_region`, `rmalloc`, `rnew`, `tagcheck`, `tag_t`, `throw`, `try`, `using`, `valueof`, `valueof_t`. As in gcc, `__attribute__` is reserved as well.

The non-terminals *character-constant*, *floating-constant*, *identifier*, *integer-constant*, *string*, and *typedef-name* are defined lexically as in C. A *type-var* is defined as a C *identifier* preceded by a `'` (backquote), optionally followed by `::`*kind*.

The start symbol is *translation-unit*.

translation-unit:
- (empty)
- *external-declaration translation-unit*$_{opt}$
- `using` *identifier* `;` *translation-unit*
- `namespace` *identifier* `;` *translation-unit*
- `using` *identifier* `{` *translation-unit* `}` *translation-unit*
- `namespace` *identifier* `{` *translation-unit* `}` *translation-unit*
- `extern "C" {` *translation-unit* `}` *translation-unit*
- `extern "C include" {` *translation-unit* `}` *translation-unit*

external-declaration:
- *function-definition*
- *declaration*

function-definition:
    declaration-specifiers$_{opt}$ *declarator*
    declaration-list$_{opt}$ *compound-statement*

declaration:
    declaration-specifiers init-declarator-list$_{opt}$ `;`
    `let` *pattern* `=` *expression* `;`
    `let` *identifier-list* `;`
    `resetable`$_{opt}$ `region` `<` *type-var* `>` *identifier* `;`
    `resetable`$_{opt}$ `region` *identifier* `;`
    `resetable`$_{opt}$ `region` *identifier* `=` `(` *expression* `)` `;`
    `alias` `<` *type-var* `>` *identifier* `=` *expression* `;`

declaration-list:
    declaration
    declaration-list declaration

declaration-specifiers:
    storage-class-specifier declaration-specifiers$_{opt}$
    type-specifier declaration-specifiers$_{opt}$
    type-qualifier declaration-specifiers$_{opt}$
    function-specifier declaration-specifiers$_{opt}$

storage-class-specifier:
    `auto`
    `register`
    `static`
    `extern`
    `extern "C"`
    `typedef`
    `abstract`

type-specifier:
    `_`
    `_::`*kind*
    `void`
    `char`
    `short`
    `int`

`long`
`float`
`double`
`signed`
`unsigned`
enum-specifier
struct-or-union-specifier
datatype-specifier
type-var
`$(` *parameter-list* `)`
`region_t`
`region_t <` *any-type-name* `>`
`dynregion_t <` *any-type-name* `>`
`dynregion_t <` *any-type-name*`,` *any-type-name* `>`
`tag_t`
`tag_t <` *any-type-name* `>`
`valueof_t (` *expression* `)`
typedef-name *type-params*$_{opt}$

kind:
identifier
typedef-name

type-qualifier:
`const`
`restrict`
`volatile`
`@numelts (` *assignment-expression* `)`
`@region (` *any-type-name* `)`
`@thin`
`@fat`
`@zeroterm`
`@nozeroterm`
`@notnull`
`@nullable`

enum-specifier:
`enum` *identifier*$_{opt}$ `{` *enum-declaration-list* `}`
`enum` *identifier*

enum-field:
    identifier
    identifier `=` *constant-expression*

enum-declaration-list:
    enum-field
    enum-field `,` *enum-declaration-list*

function-specifier:
    `inline`

struct-or-union-specifier:
    struct-or-union `{` *struct-declaration-list* `}`
    struct-or-union identifier type-params$_{opt}$ `{` *struct-declaration-list* `}`
    struct-or-union identifier type-params$_{opt}$

type-params:
    `<` *type-name-list* `>`

struct-or-union:
    `struct`
    `union`
    `@tagged union`

struct-declaration-list:
    struct-declaration
    struct-declaration-list struct-declaration

init-declarator-list:
    init-declarator
    init-declarator-list `,` *init-declarator*

init-declarator:
    declarator
    declarator `=` *initializer*

struct-declaration:
    specifier-qualifier-list struct-declarator-list `;`

specifier-qualifier-list:
    type-specifier specifier-qualifier-list$_{opt}$
    type-qualifier specifier-qualifier-list$_{opt}$

struct-declarator-list:
    struct-declarator
    struct-declarator-list `,` *struct-declarator*

struct-declarator:
    declarator
    declarator$_{opt}$ `:` *constant-expression*

datatype-specifier:
    `@extensible`$_{opt}$ `datatype` *identifier type-params*$_{opt}$ `{` *datatypefield-list* `}`
    `@extensible`$_{opt}$ `datatype` *identifier type-params*$_{opt}$
    `@extensible`$_{opt}$ `datatype` *identifier* `.` *identifier type-params*$_{opt}$

datatypefield-list:
    datatypefield
    datatypefield `;`
    datatypefield `,` *datatypefield-list*
    datatypefield `;` *datatypefield-list*

datatypefield-scope: one of
    `extern static`

datatypefield:
    datatypefield-scope$_{opt}$ *identifier*
    datatypefield-scope$_{opt}$ *identifier* `(` *parameter-list* `)`

declarator:
    pointer$_{opt}$ *direct-declarator*

direct-declarator:
    identifier
    `(` *declarator* `)`
    direct-declarator `[` *assignment-expression*$_{opt}$ `]` *zeroterm-qualifier*$_{opt}$
    direct-declarator `(` *parameter-type-list* `)`
    direct-declarator `(` *effect*$_{opt}$ *region-order*$_{opt}$ `)`
    direct-declarator `(` *identifier-list*$_{opt}$ `)`
    direct-declarator `<` *type-name-list* `>`

effect:
`;` *effect-set*

region-order:
`:` *region-order-list*

region-order-list:
atomic-effect `>` *type-var*
atomic-effect `>` *type-var*`,` *region-order-list*

zeroterm-qualifier: one of
`@zeroterm` `@nozeroterm`

pointer:
`*` *range*$_{opt}$ *region*$_{opt}$ *type-qualifier-list*$_{opt}$ *pointer*$_{opt}$
`@` *range*$_{opt}$ *region*$_{opt}$ *type-qualifier-list*$_{opt}$ *pointer*$_{opt}$
`?` *region*$_{opt}$ *type-qualifier-list*$_{opt}$ *pointer*$_{opt}$

range:
`{` *assignment-expression* `}`

region:
`_`
type-var

type-qualifier-list:
type-qualifier
type-qualifier-list type-qualifier

parameter-type-list:
parameter-list effect$_{opt}$ *region-order*$_{opt}$
parameter-list `, ...` *effect*$_{opt}$ *region-order*$_{opt}$
`...` `inject`$_{opt}$ *parameter-declaration effect*$_{opt}$ *region-order*$_{opt}$
parameter-list `, ...` `inject`$_{opt}$ *parameter-declaration effect*$_{opt}$ *region-order*$_{opt}$

effect-set:
atomic-effect
atomic-effect `+` *effect-set*

atomic-effect:
`{ }`
`{` *region-set* `}`
type-var
`regions (` *any-type-name* `)`

region-set:
type-var
type-var `,` *region-set*

parameter-list:
parameter-declaration
parameter-list `,` *parameter-declaration*

parameter-declaration:
specifier-qualifier-list declarator
specifier-qualifier-list abstract-declarator$_{opt}$

identifier-list:
identifier
identifier-list `,` *identifier*

initializer:
assignment-expression
array-initializer

array-initializer:
`{` *initializer-list*$_{opt}$ `}`
`{` *initializer-list* `, }`
`{ for` *identifier* `<` *expression* `:` *expression* `}`

initializer-list:
designation$_{opt}$ *initializer*
initializer-list `,` *designation*$_{opt}$ *initializer*

designation:
designator-list `=`

designator-list:
designator
designator-list designator

designator:
`[` *constant-expression* `]`
`.` *identifier*

type-name:
specifier-qualifier-list abstract-declarator$_{opt}$

any-type-name:
type-name
`{ }`
`{` *region-set* `}`
any-type-name `+` *atomic-effect*

type-name-list:
any-type-name
type-name-list `,` *type-name*

abstract-declarator:
pointer
pointer$_{opt}$ *direct-abstract-declarator*

direct-abstract-declarator:
`(` *abstract-declarator* `)`
direct-abstract-declarator$_{opt}$ `[` *assignment-expression*$_{opt}$ `]` *zeroterm-qualifier*$_{opt}$
direct-abstract-declarator$_{opt}$ `(` *parameter-type-list*$_{opt}$ `)`
direct-abstract-declarator$_{opt}$ `(` *effect*$_{opt}$ *region-order*$_{opt}$ `)`
direct-abstract-declarator `<` *type-name-list* `>`

statement:
labeled-statement
expression-statement
compound-statement
selection-statement
iteration-statement
jump-statement
`reset_region (` *expression* `) ;`

labeled-statement:
    identifier `:` *statement*

expression-statement:
    expression$_{opt}$ `;`

compound-statement:
    `{` *block-item-list*$_{opt}$ `}`

block-item-list:
    block-item
    block-item block-item-list

block-item:
    declaration
    statement

selection-statement:
    `if (` *expression* `)` *statement*
    `if (` *expression* `)` *statement* `else` *statement*
    `switch (` *expression* `) {` *switch-clauses* `}`
    `try` *statement* `catch {` *switch-clauses* `}`

switch-clauses:
    (empty)
    `default :` *block-item-list*
    `case` *pattern* `:` *block-item-list*$_{opt}$ *switch-clauses*
    `case` *pattern* `&&` *expression* `:` *block-item-list*$_{opt}$ *switch-clauses*

iteration-statement:
    `while (` *expression* `)` *statement*
    `do` *statement* `while (` *expression* `) ;`
    `for (` *expression*$_{opt}$ `;` *expression*$_{opt}$ `;` *expression*$_{opt}$ `)` *statement*
    `for (` *declaration expression*$_{opt}$ `;` *expression*$_{opt}$ `)` *statement*

jump-statement:
    `goto` *identifier* `;`
    `continue ;`
    `break ;`

`return ;`
`return` *expression* `;`
`fallthru ;`
`fallthru (` *argument-expression-list*$_{opt}$ `) ;`

pattern:

`_`
`(` *pattern* `)`
integer-constant
`-` *integer-constant*
floating-constant
character-constant
`NULL`
identifier
identifier type-params$_{opt}$ `(` *tuple-pattern-list* `)`
`$ (` *tuple-pattern-list* `)`
identifier `(` *tuple-pattern-list* `)`
identifier$_{opt}$ `{` *type-params*$_{opt}$ *field-pattern-list*$_{opt}$ `}`
`&` *pattern*
`*` *identifier*
identifier `as` *pattern*
identifier `<` *type-var* `>`
identifier `< _ >`

tuple-pattern-list:

`...`
pattern
pattern `,` *tuple-pattern-list*

field-pattern:

pattern
designation pattern

field-pattern-list:

`...`
field-pattern
field-pattern `,` *field-pattern-list*

expression:
    assignment-expression
    expression `,` *assignment-expression*

assignment-expression:
    conditional-expression
    unary-expression assignment-operator assignment-expression
    unary-expression `:=:` *assignment-expression*

assignment-operator: one of
    `=  *=  /=  %=  +=  -=  <<=  >>=  &=  ^=  |=`

conditional-expression:
    logical-or-expression
    logical-or-expression `?` *expression* `:` *conditional-expression*
    `throw` *conditional-expression*
    `new` *array-initializer*
    `new` *logical-or-expression*
    `rnew (` *expression* `)` *array-initializer*
    `rnew (` *expression* `)` *logical-or-expression*

constant-expression:
    conditional-expression

logical-or-expression:
    logical-and-expression
    logical-or-expression `||` *logical-and-expression*

logical-and-expression:
    inclusive-or-expression
    logical-and-expression `&&` *inclusive-or-expression*

inclusive-or-expression:
    exclusive-or-expression
    inclusive-or-expression `|` *exclusive-or-expression*

exclusive-or-expression:
    and-expression
    exclusive-or-expression `^` *and-expression*

and-expression:
equality-expression
and-expression & *equality-expression*

equality-expression:
relational-expression
equality-expression == *relational-expression*
equality-expression != *relational-expression*

relational-expression:
shift-expression
relational-expression < *shift-expression*
relational-expression > *shift-expression*
relational-expression <= *shift-expression*
relational-expression >= *shift-expression*

shift-expression:
additive-expression
shift-expression << *additive-expression*
shift-expression >> *additive-expression*

additive-expression:
multiplicative-expression
additive-expression + *multiplicative-expression*
additive-expression - *multiplicative-expression*

multiplicative-expression:
cast-expression
multiplicative-expression * *cast-expression*
multiplicative-expression / *cast-expression*
multiplicative-expression % *cast-expression*

cast-expression:
unary-expression
(*type-name*) *cast-expression*

unary-expression:
postfix-expression
++ *unary-expression*

`--` *unary-expression*
unary-operator cast-expression
`sizeof` *unary-expression*
`sizeof (` *type-name* `)`
`offsetof (` *type-name* `,` *identifier* `)`
`offsetof (` *type-name* `,` *integer-constant* `)`
`malloc (` *assignment-expression* `)`
`rmalloc (` *assignment-expression* `,` *assignment-expression* `)`
`calloc (` *assignment-expression* `, sizeof (` *type-name* `) )`
`rcalloc (` *assignment-expression* `,` *assignment-expression* `, sizeof (` *type-name* `) )`
`numelts (` *assignment-expression* `)`
`tagcheck (` *postfix-expression* `.` *identifier* `)`
`tagcheck (` *postfix-expression* `->` *identifier* `)`
`valueof (` *type-name* `)`

unary-operator: one of
`&` `*` `+` `-` `~` `!`

postfix-expression:
primary-expression
postfix-expression `[` *expression* `]`
postfix-expression `(` *argument-expression-list*$_{opt}$ `)`
postfix-expression `.` *identifier*
postfix-expression `->` *identifier*
postfix-expression `++`
postfix-expression `--`
`(` *type-name* `) {` *initializer-list* `}`
`(` *type-name* `) {` *initializer-list* `, }`

primary-expression:
identifier
constant
string
`(` *expression* `)`
primary-expression `<>`
primary-expression `@ <` *type-name-list* `>`

$(*argument-expression-list*)
identifier { *type-params*$_{opt}$ *initializer-list* }
({ *block-item-list* })

argument-expression-list:
assignment-expression
argument-expression-list , *assignment-expression*

constant:
integer-constant
character-constant
floating-constant
NULL

E Installing Cyclone

Cyclone currently only runs on 32-bit machines. It has been tested on Linux, Windows 98/NT/2K/XP using the Cygwin environment, and on Mac OS X. Other platforms might or might not work. Right now, there are a few 32-bit dependencies in the compiler, so the system will probably not work on a 64-bit machine without major changes.

To install and use Cyclone, you'll need to use the Gnu utilities, including gcc (the Gnu C compiler) and Gnu-Make. For Windows, you should first install the latest version of the Cygwin utilities to do the build, and make sure that the Cygwin `bin` directory is on your path. We use some features of gcc extensively, so Cyclone definitely will not build with another C compiler.

Cyclone is distributed as a compressed archive (a .tar.gz file). Unpack the distribution into a directory; if you are installing Cyclone on a Windows system, we suggest you choose `c:/cyclone`.

From here, follow the instructions in the INSTALL file included in the distribution.

F Tools

F.1 The compiler

This section is out-of-date.

General options

The Cyclone compiler has the following command-line options:

-help Print a short description of the command-line options.

-v Print compilation stages verbosely.

–version Print version number and exit.

-o *file* Set the output file name to *file*.

-D*name* Define a macro named *name* for preprocessing.

-D*name*=*defn* Give macro *name* the definition *defn* in preprocessing.

-B*dir* Add *dir* to the list of directories to search for special compiler files.

-I*dir* Add *dir* to the list of directories to search for include files.

-L*dir* Add *dir* to the list of directories to search for libraries.

-l*lib* Link library *lib* into the final executable.

-c Produce an object (`.o`) file instead of an executable; do not link.

-x language Specify *language* for the following input files

-s Remove all symbol table and relocation information from the executable.

-O Optimize.

-O2 A higher level of optimization.

-O3 Even more optimization.

-p Compile for profiling with the `prof` tool.

-pg Compile for profiling with the `gprof` tool.

-pa Compile for profiling with the `aprof` tool.

-S Stop after producing assembly code.

-M Produce dependencies for inclusion in a makefile.

-MG When producing dependencies assume missing files are generated. Must be used with `-M`.

-MT *file* Make *file* be the target of any dependencies generated using the `-M` flag.

-E Stop after preprocessing.

-nogc Don't link in the garbage collector.

Developer options

In addition, the compiler has some options that are primarily of use to its developers:

-g Compile for debugging. This is bulletproof for compiler developers, as the debugging information reflects the C code that the Cyclone code is compiled to, and not the Cyclone code itself. To have a look at Cyclone code during debugging (but not very cleanly as of yet), also pass in *–lineno* (see below).

-stopafter-parse Stop after parsing.

-stopafter-tc Stop after type checking.

-stopafter-toc Stop after translation to C.

-ic Activate the link-checker.

-pp Pretty print.

-up Ugly print.

-tovc Avoid gcc extensions in the C output.

-save-temps Don't delete temporary files.

-save-c Don't delete temporary C files.

–lineno Insert `#line` directives in generated C code. This slows down compilation, but helps debugging. Works best when also using `-pp`.

–nochecks Disable all null and array bounds checks (still uses "fat" representation of `?` pointers).

–nonullchecks Disable null checks.

–noboundschecks Disable array bounds checks (still uses "fat" representation of `?` pointers).

-use-cpp*path* Indicate which preprocessor to use.

-no-cpp-precomp Disable smart preprocessing (mac only).

-nocyc Don't add the implicit namespace Cyc to variable names in the C output.

-noremoveunused Don't remove externed variables that aren't used.

-noexpandtypedefs Don't expand typedefs in pretty printing.

-printalltvars Print all type variables (even implicit default effects).

-printallkinds Always print kinds of type variables.

-printfullevars Print full information for evars (type debugging).

F.2 The lexer generator

F.3 The parser generator

F.4 The allocation profiler, `aprof`

To get a profile of the allocation behavior of a Cyclone program, follow these steps:

1. Compile the program with the flag `-pa`. The resulting executable will be compiled to record allocation behavior. It will also be linked with a version of the standard library that records its allocation behavior. (If you get the message, "can't find internal compiler file `libcyc_a.a`," then ask your system administrator to install the special version of the library.)

2. Execute the program as normal. As it executes, it will write to a file `amon.out` in the current working directory; if the file exists before execution, it will be overwritten.

3. Run the program `aprof`. This will examine `amon.out` and print a report on the allocation behavior of the program.

F.5 The C interface tool, `buildlib`

`buildlib` is a tool that semi-automatically constructs a Cyclone interface to C code. It scans C header files and builds Cyclone header files and stub code so that Cyclone programs can call the C code. We use it to build the Cyclone interface to the C standard library (in much the same way that `gcc` uses the `fixincludes` program).

To use `buildlib`, you must construct a *spec file* that tells it what C headers to scan, and what functions and constants to extract from the headers. By convention, the names of spec files end in `.cys`. If `spec.cys` is a spec file, then `buildlib` is invoked by

```
buildlib spec.cys
```

The output of `buildlib` is placed in a directory, `BUILDLIB.OUT`. The output consists of Cyclone header files and the stub files `cstubs.c` and `cycstubs.cyc`.

Spec files

The form of a spec file is given by the following grammar.

spec-file:
(empty)
spec spec-file

spec:
header-name `:` *directives* `;`

directives:
(empty)
directive directives

directive:

`cpp` { *balanced-braces* }
`include` { *ids* }
`hstub` id_{opt} { *balanced-braces* }
`cycstub` id_{opt} { *balanced-braces* }
`cstub` id_{opt} { *balanced-braces* }

ids:

(empty)
*id balanced-braces** *ids*

The non-terminal *id* refers to C identifiers, and *header-name* ranges over C header names (e.g., `stdio.h`, `sys/types.h`). We use *balanced-braces* to refer to any sequence of C tokens with balanced braces, ignoring braces inside of comments, strings, and character constants.

Directives

include The include directive is used to extract constants and type definitions from C header files and put them into the equivalent Cyclone header file. For example, here is part of the spec that we use to interface to C's `errno.h`:

```
errno.h:
include { E2BIG EACCES EADDRINUSE ... }
```

The spec says that the Cyclone version of `errno.h` should use the C definitions of error constants like `E2BIG`. These are typically macro-defined as integers, but the integers can differ from system to system. We ensure that Cyclone uses the right constants by running buildlib on each system.

For another example, our spec for `sys/types.h` reads, in part:

```
sys/types.h:
include { id_t mode_t off_t pid_t ... }
```

Here the symbols are typedef names, and the result will be that the Cyclone header file contains the typedefs that define `id_t`, etc. Again, these can differ from system to system.

You can use `include` to obtain not just constants (macros) and typedefs, but struct and union definitions as well. Furthermore, if a definition you `include` requires any other definitions that you do not explicitly `include`, those other definitions will be placed into the Cyclone header too. Moreover, for all such definitions, you can include an optional, expected Cyclonedefinition that is "equivalent" to the C definition on your system. By "equivalent," we mean that your definition defines all of the same elements as the system definition (but possibly fewer), and each of these elements is "representation-compatible" in the sense that they use the same amount of storage when compiled. As example, here is our spec for `grp.h`:

```
include {
  gid_t
  group {
    struct group {
      char @gr_name;
      char @gr_passwd;
      gid_t gr_gid;
      char ** @zeroterm gr_mem;
    };
  }
}
```

This provides richer information than the compatible definition on most systems. Here is the Linux definition:

```
struct group {
  char *gr_name;
  char *gr_passwd;
  gid_t gr_gid;
  char **gr_mem;
};
```

The user definition refines the system definition by indicating that for `group` strings `gr_name` and `gr_passwd` must be non-NULL, and indicates that the array of strings `gr_mem`, is null-terminated. But note that the two definitions are representation-compatible in

that they have the same run-time storage requirements. The Cyclone version provides more precise type information. You can provide user definitions for enumerated types and `typedef`'s as well.

Some refinements (such as polymorphism), are not yet supported for user definitions. Also, `include` does not work for variable or function declarations. You have to use the `hstub` directive to add variable and function declarations to your Cyclone header.

cstub The `cstub` directive adds code (the *balanced-braces*) to the C stub file. If an optional *id* is used, then the code will be added to the stub file only if the *id* is declared by the C header. This is useful because every system defines a different subset of the C standard library.

cycstub The `cycstub` directive is like the `cstub` directive, except that the code is added to the Cyclone stub file.

hstub The `hstub` directive is like the `cstub` directive, except that the code is added to the Cyclone header file.

cpp The `cpp` directive is used to tell `buildlib` to scan some extra header files before scanning the header file of the spec. This is useful when a header file can't be parsed in isolation. For example, the standard C header `sys/resource.h` is supposed to define `struct timeval`, but on some systems, this is defined in `sys/types.h`, which must be included before `sys/resource.h` for that file to parse. This can be handled with a spec like the following:

```
sys/resource.h:
cpp {
  #include <sys/types.h>
}
...
```

This will cause `sys/types.h` to be scanned by `buildlib` before `sys/resource.h`.

You can also use the `cpp` directive to directly specify anything that might appear in a C include file (e.g., macros).

Options

`buildlib` has the following options.

-d *directory* Use *directory* as the output directory instead of the default `BUILDLIB.OUT`.

-gather and -finish `buildlib` works in two phases. In the gather phase, `buildlib` grabs the C headers listed in the spec file from their normal locations in the C include tree, and stores them in a special format in the output directory. In the finish phase, `buildlib` uses the specially formatted C headers to build the Cyclone headers and stub files. The `-gather` flag tells `buildlib` to perform just the gather phase, and the `-finish` flag tells it to perform just the finish phase.

`buildlib`'s two-phase strategy is intended to support cross compilation. A Cyclone compiler on one architecture can compile to a second architecture provided it has the other architecture's Cyclone header files. These headers can be generated on the first architecture from the output of the gather phase on the second architecture. This is more general than just having the second architecture's Cyclone headers, because it permits works even in the face of some changes in the spec file or `buildlib` itself (which would change the other architecture's Cyclone headers).

-gatherscript The `-gatherscript` flag tells `buildlib` to output a shell script that, when executed, performs `buildlib`'s gather phase. This is useful when porting Cyclone to an unsupported architecture, where `buildlib` itself does not yet work. The script can be executed on the unsupported architecture, and the result can be moved to a supported architecture, which can then cross-compile itself to the new architecture.