Cyclone
User's Manual

16 November 2001

The current version of this manual should be available at `http://www.cs.cornell.edu/projects/cyclone/` and `http://www.research.att.com/projects/cyclone/`. The version here describes Cyclone Version 0.1.3, although minor changes may have occurred before the release.

Contents

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 at Cornell and AT&T. 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. 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 to Cyclone. 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 James Cheney, Fred Smith, Nathan Lutchansky, 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 or http://www.cs.cornell.edu/projects/cyclone 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 hello.cyc -o hello
```

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. However, it's not valid Cyclone code: it will be rejected by the Cyclone compiler. Cyclone requires a *definite return*: any function with a return type other than `void` must explicitly return a value of the correct type. Since `main` is declared with return type `int`, Cyclone requires that it explicitly return an integer.

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 (even if the return type is not a pointer type).

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.

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 fact, you can't use `(void *)0` as a pointer in Cyclone, even though this is how C typically defines `NULL`. Instead, Cyclone provides `NULL` as a keyword.
- 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 `?` 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 must use a second kind of pointer, called a *fat pointer* and indicated by ? (the question mark). For example, here is a program that echoes its command-line arguments:

```
#include <stdio.h>

int main(int argc, char ??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 ??`,

which is thought of in the same way: it is a (fat) pointer to a (fat) pointer to characters. The difference between a `*` pointer and a `?` pointer is that a `?` 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 not only a null check but a bounds check. This guarantees that a `?` pointer can never cause a buffer overflow.

Because of the bounds information contained in `?` pointers, `argc` is superfluous: you can get the size of `argv` by writing `argv.size`. 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 `?` 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.

Never-null pointers

There is one other kind of pointer in Cyclone: the never-null pointer. A never-null pointer is indicated by `@` (the at sign). An `@` pointer is like a `*` pointer, except that it is guaranteed not to be NULL. This means that when you dereference an `@` pointer or assign to its contents, a null check is unnecessary.

`@` 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 `@` pointers as arguments, or return `@` pointers as results. For example, the `getc` function that reads a character from a file is declared,

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

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 @)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 @` 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.

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

Initializing Pointers

Pointers must be initialized before they are used to ensure that random stack garbage does 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.

Other features of pointers

There's much more to Cyclone pointers than we've described here.

In particular, a pointer type can also specify that it points to a sequence of a particular (statically known) length. For instance, we can write:

```
void foo(int @{4} arr);
```

Here, the parameter `arr` is a pointer to a sequence of four integer values. Both the never-null and nullable 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 *{1}`".

We explain pointers in more detail in Section 3.

2.3 Regions

Another potential way to crash a program or violate security is to dereference a dangling pointer—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 re-used (*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 re-used 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 Cyclone tracks the lifetime of every object and ensures that a pointer to an object can only be dereferenced if that object has not been deallocated.

The way that Cyclone achieves this is by assigning each object a symbolic *region* that corresponds to the lexical block in which the object is declared, and each pointer type reflects the region into which a pointer points. For instance, the variable `result` 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`. If we like, we can write the type of `&result` with the explicit region as "`struct Point * 'newPoint`". Note that the region name comes after the `*` (or `?` or `@`).

When control flow exits a block, the storage (*i.e.*, the 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 *'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. 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 * 'f` reflecting 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. 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 * 'L` reflecting the fact 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

As we will see, Cyclone often figures out the region of a pointer without the programmer providing the information. This is called *region inference*. For instance, we can re-write the function `f` above without any region annotations, and without labelling 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 }
```

and 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 a little less cryptic.

You need to understand at least four more details about regions to be an effective Cyclone programmer: the heap region, dynamic regions, region polymorphism, and default region annotations for function parameters. The following sections give a brief overview of these details.

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

Dynamic 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 accurately track the lifetimes of individual objects within the heap, so it can't be sure whether dereferencing a pointer into the heap would cause problems. Instead, a conservative garbage collector reclaims the data allocated in the heap.

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 *dynamic regions*. A dynamic region is similar to the region associated with a code block. In particular, when you execute:

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

It is worth remarking that the heap is really just a dynamic region with global scope, and you can use the global variable `heap_region` as a handle on the heap. Indeed, `new` and `malloc(...)` are just abbreviations for `rnew(heap_region)` and `rmalloc(heap_region,...)` respectively.

Region Polymorphism

Another key concept you need to understand about regions 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 function making the regions explicit:

```
void foo(struct Point *‘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`. Note that `‘r` can be instantiated with any region you like, including the heap, or a region local to a function. So, for instance, we can write the following:

```
void g() {
  struct Point p = {0,1};
  struct Point *‘g ptr1 = &p;
  struct Point *‘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`.

Cyclone automatically inserts region parameters for function arguments, 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 `'r1` and `'r2`. To make this explicit, we would write:

```
void h(struct Point *'r1 p1, struct Point *'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. That is, Cyclone fills in the missing regions as follows:

```
void j(struct Point *'r1 p1, struct Point *'r2 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, `'r1` and `'r2` *might* be instantiated with different regions, in which case the code would be incorrect. But you can make them the same by putting in the same explicit region for each pointer. Thus, the following code does type-check:

```
void j(struct Point *'r1 p1, struct Point *'r1 p2) {
  p1 = p2;
}
```

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 about the results? 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:

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

type-checks since the `new` operator returns a pointer to the heap, and the default region for the return type is the heap.

This explains why the original bad code for allocating a new point does not type-check:

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

The value `&result` is a pointer into region `'newPoint` but the result type of the function needs to 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 immediately. To see why, let us rewrite the code with the default region annotations filled in. The argument is assumed to be in a region `‘r`, and the result is assumed to be in the heap, so the fully elaborated code is:

```
int *‘H id(int *‘r 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 *‘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 won't let you dereference a pointer into a deallocated region. The lexical blocks declared in functions correspond to one type of region, and simply declaring a variable within that block allocates storage within the region. The storage is deallocated upon exit of the block. Dynamic regions are similar, except that a dynamic number of objects can be allocated within the region using the region's handle. The heap is a special region that is garbage collected.

Region polymorphism makes it possible to omit many region annotations on types. Cyclone assumes that pointers passed to functions may live in distinct regions, and assumes that result pointers are in the heap. These assumptions are not perfect, but (a) programmers can fix the assumptions by providing explicit region annotations, (b) it permits Cyclone files to be separately compiled.

The region-based type system of Cyclone is perhaps the most 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 having to resort to garbage collection.

For more information about regions, see Section 8.

2.4 Tagged Unions and Pattern Matching

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 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.

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 `tunion` is a type that has several possible cases. Here's a simple example:

```
tunion t {
  Integer(int);
  String(const char ?);
};
tunion t x = new Integer(3);
tunion t y = new String("hello, world");
```

This declares a new type, `tunion t`, that can hold either an integer or a string (remember, a string is a `char ?` in Cyclone). `Integer` and `String` are *tags* for the two possibilities. The tags are used to build values of type `tunion t`, as in the declarations of `x` and `y`.

Pattern matching is used to determine the tag of a value of type `tunion t`, and to extract the underlying value. For example, here is a function that will print either an integer or a string:

```
void print(tunion t a) {
  switch (a) {
  case &Integer(i): printf("%d",i); return;
  case &String(s): printf("%s",s); return;
  }
}
```

The argument `a` has type `tunion t`, so it is either built with tag `Integer` or tag `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 only match values that have been built with the `Integer` tag. The variable `i` is bound to the underlying integer, and it can be used in the body of the case. For example, `print(x)` will print 3, since `x` was initialized by `new Integer(3)`, and `print(y)` will print `hello, world`.

The cases of a `tunion` can carry any number of values, including none, and they can be recursive. For example, we can define a tree datatype as follows.

```
tunion tree {
  Empty;
  Leaf(int);
  Node(tunion tree, tunion tree);
};
```

A tree can be empty, or it can be a single (leaf) node holding an integer, or it can be an internal node with a left and a right subtree. In other words, `tunion tree` is the type of possibly empty binary trees with integer leaves.

Here's a function, `sum`, that calculates the sum of the leaves of a tree:

```
int sum(tunion tree x) {
  switch (x) {
  case Empty: return 0;
  case &Leaf(i): return i;
  case &Node(y,z): return sum(y)+sum(z);
  }
}
```

It's written in a straightforward way, with a case for each possible tag in the type `tunion tree`. The `Empty` case is noticeably different than the other two cases: the pattern does not use the `&` character. The reason has to do with how `tunion` is implemented. Every value of `tunion` type must have the same size; for example, the `Node` case recursively calls `sum` on the subtrees `y` and `z`, *without knowing* whether they are empty, leaves, or internal nodes. The only way that it can extract `y` and `z` from `x` without knowing this is if all possible cases of `tunion tree` have the same size.

At the same time, each tag of a `tunion` can carry a different number of values, so obviously each can require a different amount of space. To make it all work, the value-carrying cases of a `tunion` are represented as pointers to structures containing a distinguishing integer plus the values, and the non-value-carrying cases of a `tunion` are represented just as distinguishing integers. Since integers and pointers have the same size in Cyclone, this achieves the goal.

The data representation is reflected both in how `tunion` values are constructed and in the patterns used to take them apart. Value-carrying cases are built using the `new` keyword, which performs a heap allocation and results in a pointer to the new storage. Non-value-carrying cases don't require any allocation, and so they don't use `new`. For example,

```
new Node(Empty,new Leaf(5))
```

builds a tree consisting of an internal node with an empty left subtree, and a right subtree consisting of a single leaf, 5. We use `new` for the value-carrying cases, `Node` and `Leaf`, but not for `Empty`.

In pattern matching, we use the `&` character to match a pointer. So in the function `sum`, since `Leaf` and `Node` are constructed as pointers, the `&` is required to match them. Since `Empty` is not built as a pointer, the `&` must not appear.

You might be wondering, "how does Cyclone tell whether a `tunion` comes from a value-carrying case or a non-value-carrying case?" In particular, how can Cyclone tell the integers used for non-value-carrying cases apart from the pointers used for the other cases? Here's how we do it in our current implementation: We reserve a space in the low part of memory where we will never allocate Cyclone objects using `new`. If a value of a `tunion` is an address in this space, then it represents a tag without values, and if it is an address outside of this space, it represents a pointer to a structure containing a tag plus the values that it carries.

You can find out more about patterns in Section 5; for more about `tunion` and memory management, see Section 8.

2.5 Exceptions

So far we've glossed over what happens when you try to dereference a null pointer, or assign to an out-of-bounds `?` 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 @)f);
```

Suppose that there is no file `/etc/passwd`; then `fopen` will return NULL, and when `f` is cast to `FILE @`, 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 `tunion`: it can carry along some values with it. For example, the standard exception `InvalidArg` carries a string. Exceptions can be handled in `try-catch` statements, using pattern matching:

```
FILE *f = fopen("/etc/passwd","r");
int c;
try {
  c = getc((FILE @)f);
}
catch {
case Null_Exception:
  printf("Error: can't open /etc/passwd\n");
  exit(1);
case &InvalidArg(s):
  printf("Error: InvalidArg(%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. 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 `tunion`: with `tunion`, 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* `tunion`, or `xtunion`. Here's how to declare a new exception:

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

The type `xtunion exn` is the type of exceptions, and this declaration introduces a new case for the `xtunion exn` type: `My_Exception`, which carries a single value (a string). Exception values are created just like `tunion` values—using `new` for value-carrying tags only—and are thrown with a `throw` statement. For example,

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

or

```
throw Null_Exception;
```

2.6 Additional Features of Cyclone

Thus far, we have mentioned a number of advanced features of Cyclone that provide facilities needed to avoid common bugs or security holes in C. But there are many other features in Cyclone that are aimed at making it easier to write code, ranging from convenient expression forms, to advanced typing constructs. For instance, like GCC and C99, Cyclone allows you declare variables just about anywhere, instead of at the top of a block. As another example, like Java, Cyclone lets you declare variables within the initializer of a `for`-statement.

In addition, Cyclone adds advanced typing support in the form of (a) parametric polymorphism, (b) structural subtyping, (c) some unification-based, local-type inference. These features are necessary to type-check or port a number of (potentially) unsafe C idioms, usually involving "`void*`" or the like. Similarly, `tunion` types can be used to code around many of the uses for C's `union` types – another potential source of unsoundness. In what follows, we give a brief overview of these added features.

2.7 GCC and C99 Additions

GCC and the ISO C99 standard have some useful new features that we have adopted for Cyclone. Some of the ones 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 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
- `//` 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 expect to follow the C99 standard fairly closely.

2.8 Tuples

Tuples are like lightweight structs. They need not be declared in advance, and 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 gensym one.)

2.9 Creating Arrays

There are about four ways to create arrays in Cyclone. 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.

To create heap-allocated arrays (or strings) within a Cyclone function, you should either use "`new`" 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 *{8}foo = new {1,2,3,4,5,6,7,8};

// s is a checked pointer to a heap-allocated string
char ?s = new "bar";

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

2.10 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.)

2.11 Let Declarations

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);
}
```

2.12 Polymorphic Functions

As mentioned above, Cyclone supports a limited amount of subtyping polymorphism. It also 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:

```
$(string_t,int) swap1($(int,string_t) 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 `string_t`s (`char *`) 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, `tunion`, and `xtunion` 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, `tunion` 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.

`Typedef`s cannot be made abstract. As in C, typedefs are type abbreviations and are expanded at compile time. If we chose to make them (potentially) abstract, then we'd have to enforce a "boxedness" restriction, similar to the restrictions on type variables. To simplify the language, we chose to make structs abstract.

It's also possible to code up "first-class" abstract data types using `tunion`s or `xtunion`s. Individual [x]tunion constructors can be parameterized by additional type variables that are local to the type-constructor. (From a type-theoretic point of view, these are existentially-quantified variables.) Our current approach is quite similar to the treatment of existential types in Haskell. Existential types are described in Section 4.

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 requires every function to declare a return type (the implicit int for the return type of a function is removed).
- 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 `*` or `@` pointers. 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 `*` or `@` value to a `?` value and then do the pointer arithmetic instead.
- Cyclone inserts a NULL check when a `*` pointer is dereferenced and it cannot determine statically that the pointer is not NULL.
- Cyclone requires any function that is supposed to return a non-`void` value to execute a return statement (or throw an exception) 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.
- Unions in Cyclone can only hold "bits." In particular, they can hold 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 unions (`tunion`s).
- Cyclone only supports a limited form of `malloc` which is baked in. Tuples and structs can be allocated via `malloc` but this requires writing explicitly: `malloc(sizeof(t))` where t is the type of the value that you are allocating. You cannot use `malloc` to allocate an array.
- Cyclone performs a static analysis to ensure that every variable and every `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. For instance, currently, Cyclone does not support initializing a struct 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`.

- In the near future, Cyclone will place some restrictions on linking for safety reasons. In particular, if you import a variable or function with one type, then it must be exported by another file with that type. In addition, access to C code will be restricted based on a notion of security roles.
- Cyclone has some new keywords (`let`, `abstract`, `region`, etc.) that can no longer be used as identifiers.
- 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`, `tunion`, `xtunion`, and `typedef` definitions must be at the top-level.
- Cyclone does not allow you to use a `struct`, `tunion`, `union`, `xtunion`, or `enum` type without first declaring it. We do support one special case of this where you embed a declaration within a typedef as in:

```
typedef struct Point {int x,y} *point_t;
```

- Cyclone does not allow a typedef declaration to be shadowed by another declaration.
- Cyclone does not allow 0 (zero) to be treated as the NULL pointer.

3 Pointers

As in C, Cyclone pointers are 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, so it may reject legal C operations on pointers.

In order to enforce memory safety, Cyclone pointer types contain more information than their C counterparts. In addition to the type of the object pointed to, pointer types indicate:

- Whether a value of the type may be `NULL`
- The number of objects pointed to
- The region into which the pointer points

For example, the type `int *{7}'H` is for possibly-null pointers to seven `int` objects in the heap. The syntax and semantics of all this additional pointer information is now explained. Then we introduce a new type for arrays of unknown size. Pointer arithmetic is allowed only on this last collection of types. Throughout, we mention planned improvements. We end with a summary.

Whether a value of the type may be `NULL`

Cyclone's type system distinguishes between pointers that may be `NULL` and those that may not.

Syntax and Semantics The syntax is straightforward: The `*` character is for pointers that may be `NULL` (as in C), and the `@` character is for pointers that may not be `NULL`. So "`int * x = NULL;`" is accepted, but "`int @ x = NULL;`" is not.

Subtyping For any type `t`, the type `t@` is a subtype of `t*`. The type of `malloc(sizeof(t))` is `t@`, as is `new e` where `e` has type `t`. Hence in the declaration, "`int *x = malloc(sizeof(int))`", there is an implicit

legal cast from `t@` to `t*`. Note that even when `t1` is a subtype of `t2`, the type `t1*` is not necessarily a subtype of `t2*`, nor is `t1@` necessarily a subtype of `t2@`. For example, `int@@` is not a subtype of `int*@`. This illegal code shows why:

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

You can explicitly cast a value of type `t*` to `t@`. Doing so will perform a run-time check. The cast can be omitted, but the compiler emits a warning and performs the run-time check. Because the current implementation does not consider tests to change a `t*` to `t@`, such casts are sometimes necessary to avoid spurious warnings, such as in this code:

```
extern void f(int @);

void g(int * x) {
  if (x != NULL)
     f((int @)x);
}
```

Implementation A run-time null check is a simple comparison. If it fails (i.e., the value is `NULL`), the exception `Null_Exception` is thrown. A check is inserted whenever a `t*` is (explicitly or implicitly) cast to a `t@`. Casting `t@` to `t*` has no run-time effect.

Safety demands that if `x` may be `NULL`, then `*e`, `e.f`, `e->f`, and `e[e2]` are translated such that we first check that `e` is not `NULL`. `e` is only evaluated once. The only way to guarantee there is no check at run-time is to use `@` instead of `*`. For example, the function on the left performs one check whereas the one on the right performs three (both throw `Null_Exception` if passed `NULL`):

```
int sum3(int *{3} x) {
  int @{3} y = x;
  return y[0]+y[1]+y[2];
}
```

```
int sum3(int *{3} x) {
  return x[0]+x[1]+x[2];
}
```

Note that `&e->f` and `&e[e2]` check (if necessary) that `e` is not `NULL` even though these constructs do not read through `e`.

Future

- We may use dataflow information to avoid spurious warning about implicit casts from `t*` to `t@` and to avoid inserting unnecessary checks. However, the analysis is non-trivial (due to the address-of operator, unstructured control flow, and undefined evaluation order), and the C compiler may be able to eliminate unnecessary checks for us.
- For debugging purposes, we may have `Null_Exception` carry source-position information.

The number of objects pointed to

Syntax and Semantics The type `t@{37}` (similarly `t*{37}`) describes pointers to 37 `t` values. In other words, if `x` has type `t@{37}`, then `x[e]` is safe so long as `e` is between 0 and 36, inclusive. If the `{n}` is omitted, it is implicitly `{1}`. Currently, the number must be a compile-time constant—see below for arrays of unknown size.

We are taking pains not to say `t@{37}` describes an array of 37 `t` values because C (and therefore Cyclone) distinguishes arrays and pointers in certain contexts. For example, a local declaration "`t@{37} x;`" allocates space on the stack for a pointer (which must hold a pointer to 37 `t` values) whereas "`t x[37];`" allocates space on the stack for 37 `t` values.

Subtyping Pointers to more objects are subtypes of pointers to fewer objects (of the same type). An explicit cast is not necessary. Put another way, we could say `t@{37}` describes pointers to at least 37 `t` values.

Implementation The length information is not present at run-time, except implicitly in run-time checks. That is, if `e` has type `t @{37}`, the compiler translates `e[e2]` to check that `e2` is less than 37. `e2` is evaluated only once. If `e2` is a constant expression, there is no run-time check. If `e2` is a constant expression not less than 37, it is a compile-time error.

Future In the future, the bounds information on a pointer will not have to be a compile-time constant. For example, you will be able to write

```
void f(int n) {}
  int *{n} arr = new {for i < n : 37};
  ...
}
```

This addition is non-trivial because, in terms of the above example, the variable n may be mutated later in the function. In general, we are developing a general system where the sizes of pointer bounds may be expressed in terms of expressions, yet the compiler can always insert the correct bounds check or verify that the check is unnecessary.

Currently, pointer arithmetic is only allowed on types of the form `t?`. Soon we will allow adding a compile-time constant `c` to `t@{n}` (for example), with the type of the result being `t@{n-c}`. It will be a compile-time error if `c > n`.

The region into which the pointer points

Syntax and Semantics The type `t@'r` describes pointers into region `'r`. All regions start with the `'` character so that they are not confused with identifiers. If the region is omitted, the compiler inserts one. The region inserted depends on where the type occurs, as described below.

The heap region (written `'H`) conceptually lives forever; in practice, it is garbage-collected.

Every block (i.e., local scope) is a region. If you label a block with `L:`, then the region's name is `'L`. Similarly, the parameters to a function `f` are in a region named `'f`. Thanks to region inference, you can point into regions without explicit names. For example, you can say `int *x = &y` if `y` is a local variable in an unlabeled block. Conceptually, the compiler creates a label for the block and fills in the corresponding region name for you. (The output need not actually have a label.)

Because every pointer has a type and every pointer type has a region, a pointer cannot be mutated so that it points into a different region than it did before the assignment. Often subtyping (see below) is sufficient, but in some cases it is necessary to rewrite C code to use different variables

for pointers into different regions. Note that there is no way for a global variable to hold a stack pointer.

Functions are implicitly polymorphic over the regions of their arguments. For example, `void f(int *'r);` is a prototype that can be passed a pointer into any *accessible* region. That is, it can be passed a stack pointer or a heap pointer, so long as it is not passed a dangling pointer. Note that our example function `f` could not possibly assign its argument to a global, whereas `void g(int *'H);` could. On the other hand, `g` cannot be passed a stack pointer.

The rules the compiler uses for filling in regions when they are omitted from pointer types are numerous, but they are designed to avoid clutter in the common case:

- In function-argument types, a fresh 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.

In the future, we intend to change the rule for `typedef` so that the meaning can be different at each *use* of the `typedef`, as dictated by the other rules. Until then, be warned that

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

is different than

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

Also, note that these rules are exactly the same as the rules for omitted regions in instantiations of parameterized types.

Subtyping `t *'r1` is a subtype of `t *'r2` if `'r1` is known to *outlive* `'r2`. In particular, you can always cast a heap pointer to a pointer into another region.

Implementation A pointer's region is not stored with the pointer at run-time. So there is no way to ask for the region into which a pointer points. For stack regions there is no region object at run-time *per se*, just the stack space for the objects. As is normal with region-based systems, Cyclone does not prevent dangling pointers. Rather, it prevents *dereferencing* dangling pointers. But this is a subtle point.

Pointers to an Unknown Number of Objects—The `t ?` Types

So far, we have not provided a way to point to a number of objects when the number is not known at compile-time.

Syntax and Semantics The type `t ?` describes such pointers to objects of type `t`. Such types may be assigned `NULL`. They may be annotated with a region, which (as with other pointer types) is the region into which the pointer points. Omitted region annotations are filled in by the compiler. Clearly, explicit bounds information makes no sense for these types. If `e` has type `t ?`, then `e.size` has type `int` and is the number of objects pointed to by type `t`. (Actually, `e.size` is allowed for any pointer type, but for other pointers it is evaluated at compile-time.) The meaning of operations on `t ?` objects is best explained in terms of the implementation.

Implementation Unlike with types like `t*{37}`, the implementation stores bounds information with objects of type `t ?`. Currently, a `t ?` object occupies three machine words. Conceptually, the object maintains the starting address of the collection of objects pointed to, the length of the collection, and the current value of the pointer used for accessing memory. Pointer arithmetic may cause the access pointer to be out-of-bounds; no error occurs at this point. On the other hand, a subscript operation `e1[e2]` where `e1` has type `t?` checks that the access-pointer of `e1` plus `e2` is within bounds of `e1`. Both `e1` and `e2` are evaluated once. If the bound is violated the exception `Null_Exception` is thrown.

When an object of type `t?` is assigned to, it gets the bounds information from the "right-hand side" of the assignment. So `x=y` copies all of `y`'s fields to the fields of `x`. Similarly `x = y + 17`, copies `y`'s fields and then adds 17 to `x`'s access pointer. Finally, `x++` just increments `x`'s access pointer. As in C, pointer arithmetic is limited to addition of constants and

subtraction. The result of pointer subtraction has type `unsigned int`, so there is no bounds information.

Even though, `t ?` types are implemented as multi-word values, comparison operations (e.g., `==`) are defined on them—the comparison is performed on the access pointers.

Conversions to/from `t ? 'r` types from/to `t*{n}'r` and `t@{n}'r` types exist. Converting to a `t?` type just uses the `t*` or `t@`'s static type information to initialize the bounds information. The cast may be implicit; no warning is given. Converting to a `t*` or `t@` type incurs a run-time check that the access pointer has a value such that the target type's bounds information is sound. If so, the access pointer is returned, else the exception `Null_Exception` is thrown. Implicit casts of this form cause the compiler to give a warning.

Future We may add a "cannot be NULL" version of these types for sake of completeness. More significantly, we intend to allow user-defined types to have certain fields describe the bounds information for other fields, rather than relying on types built into the language.

Summary and Discussion

A pointer type has one of the following forms, where `t` is a type and `n` is a constant unsigned expression:

- `t *{n}'r`, a possibly `NULL` pointer to `n` elements of type `t` in region `'r`
- `t @{n}'r`, a non-`NULL` pointer to `n` elements of type `t` in region `'r`
- `t ? 'r`, a pointer to an unknown number of elements of type `t` in region `'r`. Implemented as a multi-word object.

If `{n}` is omitted, it is `{1}`. If the region is omitted, the compiler inserts one. The region inserted depends on where the type is written.

The easiest way to port code is to replace uses of `t *` with `t ?`. Of course, this technique does not address region annotations. Functions that can take only heap pointers (because the pointers escape into data structures, for example) will need to add `'H` annotations for the relevant parameters.

Of course, using `t?` delays errors until run-time and is less efficient. Using `t@` is the most efficient and guarantees that `Null_Exception` will not be thrown.

Currently, code performing pointer arithmetic must use `t?`.

4 Tagged Unions

In addition to `struct`, `enum`, and `union`, Cyclone has `tunion` (for "tagged union") and `xtunion` (for "extensible tagged union") as ways to construct new aggregate types. Like a `union` type, each `tunion` and `xtunion` has a number of *variants* (or members). Unlike with `union`, an object of a `tunion` or `xtunion` 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 `tunion` and `xtunion` is that `tunion` is closed—a definition lists all possible variants. It is like the algebraic datatypes in ML. With `xtunion`, separately compiled files can add variants, so no code can be sure that it knows all the variants. There is a rough analogy with not knowing all the subclasses of a class in an object-oriented language.

For sake of specificity, we first explain how to create and use tunion types. We then explain xtunion by way of contrast with `tunion`. Because the only way to read parts of `tunion` and `xtunion` types is pattern-matching, it is hard to understand `tunion` without pattern-matching, but for sake of motivation and completeness, some of the examples in the explanation of pattern-matching use `tunion`! To resolve this circular dependency, we will informally explain pattern-matching as we use it here and we stick to its simplest uses.

4.1 tunion

Basic Type Declarations and Subtyping *[Warning: For expository purposes, this section contains a white lie that is exposed in the later section called "regions for tunion".]*

A `tunion` type declaration lists all of its variants. At its simplest, it looks just like an `enum` declaration. For example, we could say:

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

As with `enum`, the declaration creates a type (called `tunion Color`) and three constants `Red`, `Green`, and `Blue`. Unlike `enum`, these constants do not have type `tunion Color`. Instead, each variant has its *own type*, namely `tunion Color.Red`, `tunion Color.Green`, and `tunion Color.Blue`. Fortunately these are all subtypes of `tunion Color` and no explicit cast is necessary. So you can write, as expected:

```
tunion Color c = Red;
```

In this simple example, we are splitting hairs, but we will soon find all these distinctions useful. Unlike `enum`, `tunion` variants may carry any fixed number of values, as in this example:

```
tunion 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: `tunion Shape`, `tunion Shape.Point`, `tunion Shape.Circle`, `tunion Shape.Ellipse`, and `tunion Shape.Polygon`. Like in our previous example, `tunion Shape.Point` is a subtype of `tunion Shape` and `Point` is a constant of type `tunion 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 `tunion Shape.Circle`, for example `Circle(3.0)`. Similarly, `Ellipse(0,0)` has type `tunion Shape.Ellipse` (thanks to implicit casts from `int` to `float` for 0) and `Polygon(7,4.0)` has type `tunion Shape.Polygon`. The arguments to a constructor can be arbitrary expressions of the correct type, for example, `Ellipse(rand(), sqrt(rand()))`.

The second difference is that value-carrying variant types (e.g., `tunion Shape.Circle`) are not subtypes of the `tunion` type (e.g., `tunion Shape`).

Rather *non-null pointers* to the value-carrying variant types are (e.g., `tunion Shape.Circle @'H` is a subtype of `tunion Shape`). So the following are correct initializations that use implicit subtyping:

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

`tunion` types 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 };
tunion Exp {
  Int(int),
  Float(float),
  Unop(enum Unops, tunion Exp),
  Binop(enum Binops, tunion Exp, tunion Exp)
};
```

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

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

Accessing tunion Variants Given a value of a `tunion` type, such as `tunion Shape`, we do not know which variant it is.

For non-value variants, we can use a standard comparison. Continuing the example from above, "`s1 == Point`" would be true whereas "`s2 == Point`" would be false.

Analogous comparisons would not work for value-carrying variants because these variants are pointers. Rather than provide predicates (perhaps of the form `isCircle(s1)`), Cyclone requires pattern-matching. For example, here is how you could define `isCircle`:

```
bool isCircle(tunion Shape s) {
  switch(s) {
```

```
    case &Circle(r): return true;
    default: return false;
    }
  }
```

When a `switch` statement's argument has a `tunion` type, the cases describe variants. One variant of `tunion 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. Pattern-matching works for non-value variants too, but there is no `&` because they are not pointers.

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(tunion 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 `tunion` 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.

We emphasize that Cyclone has much richer pattern-matching support than we have used here.

Implementation Non-value variants are translated to distinct small integers. Because they are small, they cannot be confused with pointers to value-carrying variants. Value-carrying variants have a distinct integer tag field followed by fields for the values carried. Hence all values of a tunion type occupy one word, either with a small number or with a pointer.

Regions for tunion We have seen that non-null pointers to value-carrying variants are subtypes of the `tunion` type. For example, `tunion Shape.Circle @'H` is a subtype of `tunion Shape`. Because `tunion Shape.Circle @'H` is a pointer into the heap, it would seem that all values of type `tunion Shape` are either non-value variants or pointers into the heap. In fact, this is true, but only because `tunion Shape` is itself shorthand for `tunion 'H Shape`.

In other words, `tunion` types are region-polymorphic over the region into which the value-carrying variants point. An explicit region annotation goes after `tunion`, just like an explicit region annotation goes after `*` or `@`. Here is an example using a stack region:

```
tunion Shape.Circle c = Circle(3.0);
tunion _ Shape s = &c;
```

The _ is necessary because we did not give an explicit name to the stack region.

We can now correct the white lie from the "basic type declarations and subtyping" section. A declaration `tunion Foo {...}` creates a type constructor which given a region creates a type. For any region `'r`, `tunion`

`'r Foo` is a subtype of `tunion Foo.Bar @'r` if `tunion Foo.Bar` carries values. If `tunion Foo.Bar` does not carry values, then it is a subtype of `tunion 'r Foo` for all `'r`.

In the future, we may make the implied region for `tunion Foo` depend on context, as we do with pointer types. For now, `tunion Foo` is always shorthand `tunion 'H Foo`.

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

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

In the above example, the root may be in any region, but all children will be in the heap. This 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.)

```
tunion <'a,'r::R> Tree {
  Leaf('a);
  Node(tunion 'r Tree<'a,'r>, tunion 'r Tree<'a,'r>);
};
```

Existential Types *[This feature is independent of the rest of tunion's features and can be safely ignored when first learning Cyclone.]*

In addition to polymorphic `tunion` types, it is also possible to parameterize individual variants by additional type variables. (From a type-theoretic point of view, these are existentially-quantified variables.) Here is a useless example:

```
tunion T { Foo<'a>('a, 'a, int), Bar<'a,'b>('a, 'b), Baz(int) };
```

The constructors for variants with existential types are used the same way, for example `Foo("hi","mom",3)`, `Foo(8,9,3)`, and `Bar("hello",17)`

are all well-typed. The compiler checks that the type variables are used consistently—in our example, the first two arguments to `Foo` must have the same type. There is no need (and currently no way) to explicitly specify the types being used.

Once a value of an existential variant is created, there is no way to determine the types at which it was used. For example, `Foo("hi","mom",3)` and `Foo(8,9,3)` both have type, "there exists some `'a` such that the type is `Foo<'a>`". When pattern-matching an existential variant, you must give an explicit name to the type variables; the name can be different from the name in the type definition. Continuing our useless example, we can write:

```
void f(tunion T t) {
  switch(t) {
  case Foo<'a>(x,y,z): return;
  case Bar<'b,'c>(x,y): return;
  case Baz(x): return;
  }
}
```

The scope of the type variables is the body of the case clause. So in the first clause we could create a local variable of type `'a` and assign `x` or `y` to it. Our example is fairly "useless" because there is no way for code to use the values of existentially quantified types. In other words, given `Foo("hi","mom",3)`, no code will ever be able to use the strings `"hi"` or `"mom"`. 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 section on region and effects for an explanation of why the `'e` stuff is necessary.

```
int f1(int x, int y) { return x+y; }
int f2(string x, int y) {printf("%s",x); return y; }
tunion T<'e::E> { Foo<'a>('a, int f('a, int; 'e)); };
void g(bool b) {
  tunion T<{}> t;
  if(b)
    t = Foo(37,f1);
  else
    t = Foo("hi",f2);
```

```
    switch(t) {
    case Foo<'a>(arg,fun):
      'a x = arg;
      int (*f)('a,int;{}) = fun;
      f(arg,19);
      break;
    }
  }
```

The case clause could have just been `fun(arg)`—the compiler would figure out all the types for us. Similarly, all of 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.

Future

- Currently, given a value of a variant type (e.g., `tunion 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.
- If a `tunion` has only one value-carrying variant, it does not need a tag field in its implementation. We have not yet implemented this straightforward optimization.

4.2 xtunion

We now explain how an `xtunion` type differs from a `tunion` type. 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:

```
xtunion Food;
xtunion Food { Banana; Grape; Pizza(list_t<xtunion Food>) };
xtunion Food { Candy; Broccoli };
```

After these declarations, `Pizza(new List(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 `xtunion`. Hence all pattern-matches against a value of an `xtunion` type must end with a case that matches everything, typically `default`.

There is one built-in `xtunion` type: `xtunion exn` is the type of exceptions. Therefore, you declare new `xtunion exn` types like this:

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

The implementation of `xtunion` types is very similar to that of `tunion` types, but non-value variants cannot be represented as small integers because of separate compilation. Instead, these variants are represented as pointers to unique locations in static data. Creating a non-value variant still does not cause allocation.

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.

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. 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**
  - 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`.

- **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 `tunion` 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
  - Types for match: tuple types where pi matches the type of the tuple's ith field `i` between 1 and `n`.
  - 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 two 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` 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
  - 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`.
  - Expressions matched: structs where the value in fi matches pi for `i` between 1 and `n`.
  - Bindings introduced: bindings introduced by p1,...,pn

- **Pointer patterns**
  - Syntax: `&p` where p is a pattern

- Types for match: pointer types, including `?` types. Also `tunion Foo` (or instantiations of it) when the pattern is `&Bar(p1,...,pn)` and `Bar` is a value-carrying variant of `tunion 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 tunion 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

- **tunion and xtunion patterns**
  - Syntax: `X` if X is a variant that carries no values. Else `X(p1,...,pn)` where X is the name of a variant (that has no existential type parameters) and p1, ..., pn are patterns. If X has existential type parameters, the syntax is `X<'t1,...,'tm>(p1,...,pn)` for distinct `'t1`,...,`'tm`.
  - Types for match: If `X` is non-value-carrying variant of `tunion Foo`, then types `tunion Foo` and `tunion Foo.x` (or instantiations of them). If `X` carries values, then `tunion Foo.X` (or instantiations of it) where the pi matches the type of ith field. The number of existential type variables in the pattern must be the number of existential type variables for tunion `Foo.X`.
  - 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`, `tunion`, `xtunion`, 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 = (_ @) 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 casts (such as from `t@` to `t*`) 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 @ 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 @`, 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 @` 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 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. 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 three sorts 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.

Dynamic regions Cyclone also has *dynamic regions*, which are regions that you can add objects to over time. You create a dynamic region in Cyclone with a statement,

```
region identifier statement
```

This declares and allocates a new dynamic 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 Cyclone has a special region called the *heap*. There is only one heap, 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.

Objects outside of the heap live until their region is deallocated; there is no way to free such an object earlier. Objects in the heap can be garbage collected once they are unreachable (i.e., they cannot be reached by traversing pointers) from the program's variables. Objects in live non-heap regions always appear reachable to the garbage collector (so everything reachable from them appears reachable as well).

Cyclone forbids following 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) might be followed. 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 three 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 ? 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*`))`. This is the only use of `malloc` allowed in Cyclone; to enforce this, we have made `malloc` a keyword. This is much more restricted than in C, where `malloc` is just an identifier bound to a library function consuming an `int` and returning a `char *`.

  In Cyclone, you cannot even write `malloc(8)` if `sizeof(`*type*`)` is 8! So, `malloc` can't be used to create an array whose size depends on run-time data.

On the plus side, the type of `malloc(sizeof(`*type*`))` is *type* `@` (a subtype of *type* `*`), so there is no need to cast the result from `char *`.

Objects can be created in a dynamic region using the following analogous expressions.

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

`rnew` and `rmalloc` are keywords.

The Cyclone library has a global variable `Core::heap_region` which contains a handle for the heap region, so, for example, `new` *expr* is just `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 dynamic 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.

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 types 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 dynamic-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 to 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; the second one is more useful:

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

In type declarations (including `typedef` for now) 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, this 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`, `tunion`, and `xtunion` declarations can all be parameterized by regions, just as they can be parameterized by types. For example, here is part of the list library. Note that the "`::R`" is necessary.

```
struct List<‘a,‘r::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 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 region name.
- 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), the pointer's type's region 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 unique 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`.

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.) 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` and region `r s`, the type of `r` is `region_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 trickly code (such as existential region names) to make the check fail.

8.4.3 Assignment and Outlives

A pointer type's region name is part of the type. If `e1` and `e2` are pointers, then `e1 = e2` is well-typed only if the region name for `e2`'s type "outlives" the region name for `e1`'s type. By outlives, we intuitively mean the region corresponding to one region name will be deallocated after the region corresponding to the other region name. The rules for outlives are as follows:

- Every region outlives itself.
- `'H` outlives every region name.
- Region names for inner blocks outlive region names for outer blocks.
- For regions in function prototypes, you can provide explicit "outlives" as in this example:

  ```
  void f(int *'r1*'r2 x,int *'r3 y; 'r2 < 'r1, 'r3 < 'r2);
  ```

  This says that `'r1` outlives `'r2` and `'r2` outlives `'r3`. The body will be checked under these assumptions. Calls to `f` will type-check only if the compiler knows that the region names of the actual arguments obey the outlives assumptions.

For handlers, 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>`.

8.4.4 Type Declarations

A `struct`, `typedef`, `tunion`, or `xtunion` declaration may be parameterized by any number of region names. The region names are placed in the list of type parameters. They must be followed by "`::R`", except for `typedef` declarations (where the region name appears in the underlying type). For example, given

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

the type `struct List<int,'H>` is for a list of ints in the heap. 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.

`tunion` and `xtunion` declarations must also be instantiated with an additional region name. If an object of type `tunion 'r Foo` turns out to be a value-carrying variant, then the object is treated (capability-wise) as a pointer with region name `'r`. If the region name is omitted from a use of a `tunion` declaration, it is implicitly `'H`.

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*. 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). 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 more than "ignore those funny type variables of kind E".

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 an *effect variable* that is not existentially bound. An effect variable stands for a capability, that is, a set of region names.

If the contents of existential packages contain only heap pointers, this effect variable is unnecessary; it can just be the "empty effect".

We will provide more documentation for existential packages that contain region pointers in the near future.

9 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;
}
```

10 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 ? '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, `args.size` 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.

A very useful example of this style is in the list library:

```
list_t<'a> list(... 'a argv) {
  list_t result = NULL;
  for (int i = argv.size - 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 `tunion` in conjunction with the second style. The callee then uses an array subscript to access a vararg and a switch statement to determine its `tunion` variant. But this would not be very convenient for the caller—it would have to explicitly "wrap" each vararg in the `tunion` type. The third style makes this wrapping implicit. For example, the type of printf in Cyclone is:

```
extern tunion PrintArg<'r::R> {
  String_pa(const char ?'r);
  Int_pa(unsigned long);
  Double_pa(double);
  ShortPtr_pa(short @'r);
  IntPtr_pa(unsigned long @'r);
};
typedef tunion 'r PrintArg<'r> parg_t<'r>;
printf(const char ?'r fmt, ... inject parg_t<'r2>);
```

The special syntax "`inject`" is the syntactic distinction for the third style. The type must be a `tunion` type. In the body of the vararg function, the array holding the vararg elements has this `tunion` 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 `tunion` 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 `tunion` 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).

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 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:

- Use NULL instead of 0.
- Change pointer types to fat pointer types where necessary.
- Use comprehensions to heap-allocate arrays.

- Use tunions 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, realloc, memset, memcpy, etc.
- Use polymorphism or tunions 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.

port:null]Use NULL instead of 0. Use `NULL` instead of 0 for null-pointers.

port:pointers]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@`) 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?`). 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.

Use comprehensions to heap-allocate arrays. Cyclone provides limited support for `malloc` and separated

initialization but this really only works for `struct`s or

tuples. To heap- or region-allocate and initialize an array, use

`new` or `rnew` in conjunction with array comprehensions.

For example, to copy a string `s`, one might write:

```
char ?t = new {for i < s.size : s[i]};
```

Use tunions for unions with pointers. Cyclone only accepts 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 (`tunion`).

Note that this adds a level of indirection and requires pattern

matching to ensure type-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 @new_dict();

struct DICT @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 @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: const unsigned char ?#0  != unsigned char ?#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:

```
returns value of type int @#0  but requires int @
        #0 and ‘H failed to unify.
returns value of type int @#1  but requires int @
        #1 and ‘H failed to unify.
```

reflecting the fact 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.

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 = dup("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 = dup("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, realloc, memset, memcpy, 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` and `memcpy` are also not supported by Cyclone. You'll need to write code to manually copy one data structure to another. Fortunately, this isn't so bad since Cyclone supports structure assignment.

Use polymorphism or tunions 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`:

```
tunion IntOrString { Int(int), String(char ?) };
typedef tunion 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.2 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. 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);

}
```

The extern C approach works well enough that it covers many of the cases that you'll encounter. However, the situation is not so easy or straightforward when you start to take advantage of Cyclone's features. As a simple example, suppose you want to call a C function `int_to_string` that takes in an integer and returns a string representation of that integer. The C prototype for the function would be:

```
char *int_to_string(int i);
```

If we just “extern-C” it, then we can certainly call the function and pass it an integer. But we can’t really use the string that we get out, because we’ve asserted that the return type is not a string, but rather a (possibly-null) pointer to a single character. So, when we call `foo` below:

```
extern "C" char *int_to_string(int i);



void foo() {

  int i = 12345;

  printf(int_to_string(i));

}
```

we’ll only get “`1`” for the output instead of “`12345`”.
If we know that the function always returns a pointer to a buffer of some fixed constant size, say `MAX_NUM_STRING`, then we can change the prototype to:

```
extern "C" char *{MAX_NUM_STRING} int_to_string(int i);
```

and we’ll get the right behavior. However, this obviously isn’t going to work if the size of the buffer might be different for different calls.
Another solution is to somehow convert the “C string” to a “Cyclone

string" before handing it back to Cyclone. This is fundamentally an unsafe operation because we must rely upon the "C string" being properly zero-terminated. So, your best bet is to write a little wrapper function in C which can convert the C string to a Cyclone string and then use that as follows:

```
extern "C" char *int_to_string(int i);

extern "C" char ?Cstring_to_string(char *);



void foo() {

  int i = 12345;

  printf(Cstring_to_string(int_to_string(i)));

}
```

Fortunately, the Cyclone runtime (`lib/runtime_cyc.c`) provides the needed routine which looks as follows:

```
// struct definition for fat pointers

struct _tagged_arr {

  unsigned char *curr;  // current pointer

  unsigned char *base;  // base address of buffer

  unsigned char *last_plus_one; // last_plus_one - base = size

};
```

```
struct _tagged_arr Cstring_to_string(char *s) {
  struct _tagged_arr str;
  if (s == NULL) {
    // return Cyclone fat NULL
    str.base = str.curr = str.last_plus_one = NULL;
  }
  else {
    int sz = strlen(s)+1;  // calculate string length + 1 for 0
    str.base = (char *)GC_malloc_atomic(sz); // malloc a new buffer
    if (str.base == NULL) // check that mallloc succeeded
      _throw_badalloc();
    str.curr = str.base;  // set current to base
    str.last_plus_one = str.base + sz; // set the size

    // Copy the string in case the C code frees it or mangles it
    str.curr[--sz] = '\0';
    while(--sz>=0)
      str.curr[sz]=s[sz];
```

```
  }

  return str; // return the fat pointer

}
```

The `_tagged_arr` definition defines the struct type that Cyclone uses to represent all fat pointers. (It's actually defined in a header file that gets included.) Fat pointers are represented using a "current pointer" which is the real pointer, and two other pointers which represent the base address and maximum address (plus one) for the buffer of objects.
The second definition defines our wrapper function which returns a fat pointer (`struct _tagged_arr`) given a C string. You'll notice that the function is bullet-proofed to avoid a number of issues. For instance, we first check to see if the C string is actually `NULL` and if so, return a fat `NULL` (a struct where `curr`, `base`, and `last_plus_one` are all `NULL`.) If the C string is not `NULL`, we allocate a new buffer and copy the string over to the buffer. This ensures that if C re-uses the storage (or frees it), Cyclone won't get confused. Notice also that we call `GC_malloc_atomic` to allocate the storage. In this case, we can use the atomic malloc because we know the data do not contain pointers. After copying the string, we initialize a `struct _tagged_arr` appropriately and then return the struct.
If we could ensure that the storage passed back to us wasn't going to get recycled, then we could avoid the copy and simplify the code greatly:

```
struct _tagged_arr Cstring_to_string(char *s) {

  struct _tagged_arr str;

  if (s == NULL) {
```

```
    // return Cyclone fat NULL
    str.base = str.curr = str.last_plus_one = NULL;
  }
  else {
    int sz = strlen(s)+1;  // calculate string length + 1 for 0
    str.base = str.curr = s;
    str.last_plus_one = str.base + sz; // set the size
  }
  return str; // return the fat pointer
}
```

Of course, using this is a bit more risky. It's up to you to make sure that you get the code right.

In porting various C libraries to Cyclone, we have had to write a number of wrappers. Doing so is fraught with peril and in the future, we hope to provide tools that make this task easier and easier to get right. If you are planning to interface to C code and need to write interfaces or wrappers, we encourage you to look through the libraries to see how we have done things.

A particularly good example is the standard I/O library. The interface `lib/stdio.h` just includes `lib/cstdio.h` and opens up the `Std` namespace. (This makes it easier to port C code, but if you want to keep the namespace closed, you can directly include `lib/cstdio.h`.) The `cstdio.h` file is adapted from the BSD and Gnu `stdio.h` files and shares a lot in common with them. For instance, there is an abstract struct for files, definitions for stdout, stdin,

stderr, various macros, and various function prototypes. A typical example function is the one to remove a file which has the following prototype:

```
extern int remove(const char ?);
```

You'll notice that the function takes in a Cyclone string as an argument. Obviously, the "real" remove takes in a C string. What is going on here is that Cyclone defines a wrapper function which, when given a Cyclone string, converts it to a C string, and then calls C's remove. The wrapper function is defined in the file `stdio.cyc`. Here are a few excerpts from that file:

```
namespace Cstdio {

  extern "C" {

    extern struct __sFILE;

    typedef struct Cstdio::__sFILE __sFILE;

    int remove(char *);

    int fclose(__sFILE);

    ...

  }

}


namespace Std;
```

```
abstract struct __sFILE {
  Cstdio::__sFILE *file;
};

int remove(const char ? filename) {
  return Cstdio::remove(string_to_Cstring(filename));
}

int fclose(FILE @‘r f) {
  if (f->file == NULL) return -1;
  int r = Cstdio::fclose((Cstdio::__sFILE @) f->file);
  if (r == 0) {
    f->file = NULL;
  }
  return r;
}

...
```

At the top of the file, we have declared the external types
and functions that C uses. Notice that these definitions
are wrapped in their own namespace (`Cstdio`) so that
we can "redefine" them within the `Std` namespace.
Also notice that they are wrapped with an extern-C so that
when compiled, their names won't get mangled.
The Cyclone wrapper code starts after the `namespace Std`
declaration. The first thing we do is define a "wrapper"
type for C files. The wrapper includes a possibly null pointer
to a C file. We use this level of indirection to keep someone
from closing a file twice, or from reading or writing to a file
that has been closed. Of course, any operations on files will
need wrappers to strip off the level of indirection and check
that the file has not been closed already.
The wrapper function for `remove` calls the `string_to_Cstring`
function (defined in `runtime_cyc.c`) to conver the
argument to a C string and then passes the C string to the
real `remove` function, returning the error code.
The wrapper function for `fclose` checks to make sure that
the file has not already been closed. If so, it returns -1.
Otherwise, it pulls out the real C file and passes it to the
real `fclose` function. It then checks the return code
(to ensure that the close actually happened) and if it's 0,
sets the C file pointer to `NULL`, ensuring that we
don't call C's `fclose` on the file again.

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 through static or dynamic checks. For example,

```
int *x = NULL;
```

is not an error, but

```
int @x = NULL;
```

is an error

What does `int *{37}` mean? This is the type of pointers to a sequence of at least 37 integers. 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`. A region is just a group of objects with the same lifetime—all objects in a region are freed at once. Cyclone uses this region information to prevent dereferencing a pointer into a previously freed region. Regions can have a "nested" structure, for example, if the region for a function parameter is a variable, then the function may assume that the parameter points into a region whose lifetime includes the lifetime of the function.

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; conceptually, objects in the heap last "forever," since they are always available if needed; garbage collection is like an optimization that frees objects after they are no longer needed.

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 puts it in for you implicitly. The region added depends on where the pointer type occurs. In function arguments, a new region variable is used. In function results and type definitions (inlcuding `typedef`), the heap region (`'H`) is used. In function bodies, the compiler looks at the uses (using unification) to try to determine a region.

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`, pointers, `tunion` types, and `xtunion` types. 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.

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 _ is not allowed as part of top-level declarations.

What do `'a::B`, `'a::M`, `'a::A`, `'a::R`, and `'a::E` mean? Types are divided into different groups, which we call kinds. There are five different kinds: B (for Box), M (for Memory), A (for Any), R (for Region), and E (for Effect). 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`, pointers (except for questionable pointers) tagged unions, and extensible tagged unions. Memory types include all box types, tuples, `tunion` and `xtunion` variants, questionable pointers, and non-abstract structs. Any types include all types that don't have kind Region or Effect. Region types are regions, i.e., the heap and stack regions. 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 are `tunion` and `xtunion`? These are Cyclone's tagged union and extensible tagged union types. In C, when a value has `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 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`, `codegen`, `cut`, `fallthru`, `fill`, `let`, `malloc`, `namespace`, `new`, `NULL`, `region_t`, `regions`, `rmalloc`, `rnew`, `splice`, `throw`, `try`, `tunion`, `using`, `xtunion`. 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 `$(`$type_1$`, ..., `$type_n$`)`. A value of the type is constructed by `$(`$expr_1$`, ..., `$expr_n$`)`. where $expr_i$ has type $type_i$. If *expr* has type `$(`$type_1$`, ..., `$type_n$`)`, you can extract the component `i` using *expr*`[i]`. The expression in the brackets must be a compile-time constant. In short, tuples are like anonymous structs where you use *expr*`[i]` to extract fields instead of *expr*`.i`. There is no analogue of the `->` syntax that can be used with pointers of structs; if *expr* has type `$(`$type_1$`, ..., `$type_n$`) *`, 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`. i is an identifier. e1 is an unsigned int indicating the size of the array. e2 is evaluated e1 times, with i having values 0, 1, ..., e1-1 and the result initializes the ith element of the array. The form new {`for i < e1 : e2`} 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 < e1: e2`} is not an argument to new, e1 must be constant and e2 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

```
xtunion exn { MyExn };
```

The exception can be thrown with the statement

```
throw MyExn;
```

You can catch the expression with a `try/catch` statement:

`try` $statement_1$ `catch { case MyExn:` $statement_2$ `}`

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:

```
xtunion 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 `xtunion` type. Therefore, all the standard rules for extending, creating objects, and destructing objects of an xtunion type apply.

How efficient is exception handling? Entering a `try` block is implemented using `setjmp`. Throwing an exception is implemented with `longjmp`. Pattern-matching an `xtunion` 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 tagged unions. 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 (eg. 'a *), then an actual parameter can have any type, but the restrictions about using the same type still apply. In general, Cyclone has parametric polymorphism as a safe alternative to casts and void *.

Do functions with type variables get duplicated like C++ template functions?Is there run-tim
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*`}`, *type*`*{`*const*`}`, 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* objects. A cast

to *type*`@{`*const*`}` 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*`*{`$const_1$`}` can be cast to *type*`*{`$const_2$`}` provided $const_2 < const_1$, and similarly for *type*`@{`$const_1$`}` and *type*`@{`$const_2$`}`.

An object of type `tunion T.A` can be cast to `tunion T` if `A` does not carry values. An object of type tunion `T.A@` can be cast to `tunion T` if `A` does carry values. 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 cannot contain pointers. You can have numeric types (including bit fields and enumerations), structs and tuples of allowable union-member types, and other 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.

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 `argv.size`.

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 initializng the pointer variable with the local's address, rather than delaying the assignment until later.

How much pointer arithmetic can I do? On "questionable" pointers (pointers with type *type*?), 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.) 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 null-terminated? Cyclone follows C's lead on this. String literals like `"foo"` are null-terminated. Many of the library functions consider a null character to mark the end of a string. And library functions that return strings often ensure that they are null terminated. However, there is no guarantee that a string is null terminated. For one thing, as in C, the terminating null 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 null character is found, or when the bounds are exceeded.

Why can't I assign 0 to a pointer? While it handles many cases seamlessly, the type-checker sometimes cannot infer that 0 should be interpreted as a `NULL` pointer. This means that in some cases you have to assign `NULL` instead of 0.

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`? No, neither of these functions are currently provided and it is not possible to write them in Cyclone. Both features are hard to provide in a way that is guaranteed safe.

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? The easiest way to do this is to pass the flags `-save-c` and `-pp` to the compiler. This instructs 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 `typedef`s 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 gener-

ation, but we have reserved the keywords `codegen`, `splice`, `cut`, and `fill` in case we get a chance to add it.

What platforms are supported? You need a platform that has gcc 2.9, 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 have actively develop Cyclone on cygwin (Windows 98, NT, 2K), and Linux. We have code for versions on Solaris, OpenBSD, FreeBSD, and Mac OS X. The platform-specific parts of these non-development distributions, particularly system call interfaces, may not be correct. We are in the process of developing a tool to automatically generate system-dependent code that should be part of future releases.

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 (old last) cell, but `l` still points to the old first (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? Compile-time constants are `NULL`, integer and character constants, and arithmetic operations over compile-time constants. Constructs requiring compile-time constants are: tuple-subscript (e.g., `x[3]` for tuple `x`), case argument for `switch "C"` argument has a numeric type (e.g., `case 3+4`:), sizes in array declarations (e.g., `int y[37]`, and sizes in pointer bounds (e.g., `int * x{124}`). Unlike in C, `sizeof(t)` is not an integral constant expression because the Cyclone compiler does not know the actual size of aggregate types.

How can I get the size of an array? If *expr* is an array, then *expr*`.size` returns the array's size. Note that for `?` types, the size is retrieved at runtime from the object. For other array types, the size is determined at compile-time.

C Libraries

C.1 C Libraries

Cyclone provides partial support for the following C library headers:

```
<arpa/inet.h> <assert.h> <ctype.h> <dirent.h> <errno.h>
<fcntl.h> <getopt.h> <grp.h> <math.h> <netdb.h>
<netinet/in.h> <netinet/tcp.h> <pwd.h> <signal.h>
<stddef.h> <stdio.h> <stdlib.h> <string.h> <strings.h>
<sys/mman.h> <sys/resource.h> <sys/select.h> <sys/socket.h>
<sys/stat.h> <sys/time.h> <sys/types.h> <sys/wait.h>
<time.h> <unistd.h>
```

For each supported C library header `<XXX.h>`, we also provide a header `<cXXX.h>`, which has the same declarations as `<XXX.h>`, except that they are all contained in namespace Std. For example, `<cstdio.h>` declares `Std::printf`. Each file `<XXX.h>` is equivalent to

```
#include <cXXX.h>
using Std;
```

C.2 <array.h>

Defines namespace Array, implementing utility functions on arrays.

```
void qsort(cmpfn_t<'a, 'r, 'r>, 'a ?'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, , >, 'a ?'H 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 ?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 ?x);
```

`to_list(x)` returns a new heap-allocated list with the same elements as the array `x`.

```
'a ?copy('a ?x);
```

`copy(x)` returns a fresh copy of array `x`, allocated on the heap.

```
'b ?map('b(@'H f)('a), 'a ?x);
```

`map(f,x)` applies `f` to each element of `x`, returning the results in a new heap-allocated array.

```
'b ?map_c('b(@'H f)('c, 'a), 'c env, 'a ?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(@‘H f)(‘a), ‘a ?x);
```

imp_map(f,x) replaces each element xi of x with f(xi).

```
void imp_map_c(‘a(@‘H f)(‘b, ‘a), ‘b env, ‘a ?x);
```

imp_map_c is a version of imp_map where the function argument requires a closure as its first argument.

```
xtunion exn {
  Array_mismatch
};
```

Array_mismatch is thrown when two arrays don't have the same length.

```
‘c ?map2(‘c(@‘H f)(‘a, ‘b), ‘a ?x, ‘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 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(@‘H f)(‘a), ‘a ?‘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(@‘H f)(‘a, ‘b), ‘a env, ‘b ?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(@‘H f)(‘a), ‘a ?x);
```

iter(f,x) is like app(f,x), except that f returns void.

```
void iter_c(void(@‘H f)(‘b, ‘a), ‘b env, ‘a ?x);
```

iter_c(f,env,x) is like app_c(f,env,x) except that f returns void.

```
void app2(‘c(@‘H f)(‘a, ‘b), ‘a ?x, ‘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, Array_mismatch is thrown.

```
void app2_c(‘d(@‘H f)(‘a, ‘b, ‘c), ‘a env, ‘b ?x, ‘c ?y);
```

app2_c is a version of app where the function argument requires a closure as its first argument.

```
void iter2(void(@‘H f)(‘a, ‘b), ‘a ?x, ‘b ?y);
```

iter2 is a version of app2 where the function returns void.

```
void iter2_c(void(@‘H f)(‘a, ‘b, ‘c), ‘a env, ‘b ?x, ‘c ?y);
```

iter2_c is a version of app2_c where the function returns void.

```
‘a fold_left(‘a(@‘H f)(‘a, ‘b), ‘a accum, ‘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(@‘H f)(‘c, ‘a, ‘b), ‘c env, ‘a accum, ‘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(@‘H f)(‘a, ‘b), ‘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(@‘H f)(‘c, ‘a, ‘b), ‘c env, ‘a ?x, ‘b accum);
```

fold_right_c is a version of fold_right where the function argument requires a closure as its first argument.

```
‘a ?rev_copy(‘a ?x);
```

rev_copy(x) returns a new heap-allocated array whose elements are the elements of x in reverse order.

```
void imp_rev(‘a ?x);
```

imp_rev(x) reverses the elements of array x.

```
bool forall(bool (@‘H pred)(‘a), ‘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 (@‘H pred)(‘a, ‘b), ‘a env, ‘b ?x);
```

forall_c is a version of forall where the predicate argument requires a closure as its first argument.

```
bool exists(bool (@‘H pred)(‘a), ‘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 (@‘H pred)(‘a, ‘b), ‘a env, ‘b ?x);
```

exists_c is a version of exists where the predicate argument requires a closure as its first argument.

```
$(‘a, ‘b)?zip(‘a ?‘r1 x, ‘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, Array_mismatch is thrown.

```
$(‘a ?, ‘b ?)split($(‘a, ‘b)?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 ?l, ‘a x);
```

memq(l,x) returns true if x is == an element of array l, and returns false otherwise.

```
bool mem(int(@‘H cmp)(‘a, ‘a), ‘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.

```
‘a ?extract(‘a ?x, int start, int *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 ?'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 @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`.

```
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 reclaimed by the garbage collector.

```
void add_char(T , unsigned 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. These declarations are made outside of any namespace.

```
typedef const unsigned char ?‘r string_t<‘r>;
```

A string_t<‘r> is a constant array of characters allocated in region ‘r.

```
typedef unsigned char ?‘r mstring_t<‘r>;
```

An mstring_t<‘r> is a non-const (mutable) array of characters allocated in region ‘r.

```
typedef string_t<‘r1> @‘r2 stringptr_t<‘r1,‘r2>;
```

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> @‘r2 mstringptr_t<‘r1,‘r2>;
```

mstringptr is the mutable version of stringptr_t.

```
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.

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> @'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`.

```
xtunion exn {
  Present
};
```
`Present` is thrown when a key is present but not expected.

```
xtunion exn {
  Absent
};
```
`Absent` is thrown when a key is absent but should be present.

```
dict_t<'a, 'b> empty(int(@'H 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.

```
dict_t<'a, 'b, 'r> rempty('r, int(@'H cmp)('a, 'a));
```
`rempty(r,cmp)` is like `empty(cmp)` except that the dictionary is allocated in the region with handle `r`.

```
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, '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)@> l)
```
`inserts(d,l)` inserts each key, value pair into `d`, returning the resulting dictionary.

```
dict_t<‘a, ‘b> singleton(int(@‘H 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`.

```
dict_t<‘a, ‘b, ‘r> rsingleton(‘r, int(@‘H cmp)(‘a, ‘a), ‘a k, ‘b v);
```
`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 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.

```
‘b *‘r rlookup_opt(‘r, dict_t<‘a, ‘b> d, ‘a k);
```
`rlookup_opt(r,d,k)` is like `lookup_opt(d,k)` except that any option returned will be allocated in the region with handle `r`.

```
bool lookup_bool(dict_t<‘a, ‘b> d, ‘a k, ‘b @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(@‘H 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(@‘H f)(‘d, ‘a, ‘b, ‘c), ‘d env, dict_t<‘a, ‘b> d, ‘c accu
```
`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(@‘H 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(@‘H 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(@‘H 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(@‘H 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(@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(@f)(‘c, ‘b, ‘b), ‘c env, dict_t<‘a, ‘b> d1, dict_t<‘
```

`iter2_c` is like `iter` except that `f` takes a closure as its first argument.

```
‘c fold2_c(‘c(@f)(‘d, ‘a, ‘b1, ‘b2, ‘c), ‘d env, dict_t<‘a, ‘b1> d1, c
```

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(‘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(@‘H 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(‘r, ‘c(@‘H 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(@‘H 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(‘r, ‘c(@‘H f)(‘d, ‘b), ‘d env, dict_t<‘a, ‘b
```

`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(@f)(‘c, ‘a, ‘b, ‘b), ‘c env, dict_t<
```

`union_two(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 keyword in Cyclone.)

```
dict_t<‘a, ‘b, ‘r> intersect(‘b(@f)(‘a, ‘b, ‘b), dict_t<‘a, ‘b, ‘r> d1
```

`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(@f)(‘c, ‘a, ‘b, ‘b), ‘c env, dict_t<
```

`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 (@‘H 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 (@‘H f)(‘a, ‘b, ‘b), dict_t<‘a, ‘b> d1, dic
```

`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)@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.

```
$(‘a, ‘b)@‘r rchoose(‘r, dict_t<‘a, ‘b> d);
```

`rchoose(r,d)` is like `choose(d)`, except the resulting pair is allocated in the region with handle `r`.

```
list_t<$(‘a, ‘b)@> 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)@‘r, ‘r> rto_list(‘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 (@‘H 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(‘r, bool (@‘H 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 (@‘H f)(‘c, ‘a, ‘b), ‘c env, dict_t<‘a, ‘
```

`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(‘r, bool (@‘H f)(‘c, ‘a, ‘b), ‘c env, dic
```

`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(‘r, dict_t<‘a, ‘b> d1, dict_t<‘a, ‘b> d
```

`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(‘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`.

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 tunion`

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 `'eff` is the effect.

```
fn_t<'arg, 'res, 'e1> make_fn('res(@'H f)('env, 'arg;'e1+{}), 'env x;'
```

`make_fn(f,env)` builds a closure out of a function and an environment.

```
fn_t<'arg, 'res, 'e1> fp2fn('res(@'H f)('arg;'e1+{}));
```

`fp2fn(f)` converts a function pointer to a closure.

```
'res apply(fn_t<'arg, 'res, 'eff> f, 'arg x;'eff+{});
```

`apply(f,x)` applies closure `f` to argument `x` (taking care of the hidden environment in the process).

```
fn_t<'a, 'c, > compose<'a::?,'b::?,'c::?,'e1::?,'e2::?,'e3::?>(fn_t<'a
```

`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, 'e1>, > curry(fn_t<$('a, 'b)@'H, 'c, 'e1> f);
```

`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)@, 'c, > uncurry(fn_t<'a, fn_t<'b, 'c, 'e1>, 'e2> f);
```

`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> map_fn(fn_t<'a, 'b, 'e> 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 <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> @table_t<'a,'b>;
```

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(@'H cmp)('a, 'a), int(@'H 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)`.

```
void insert(table_t<'a, 'b> t, 'a key, 'b val);
```

`insert(t,key,val)` binds `key` to `value` in `t`.

```
'b lookup(table_t<'a, 'b> t, 'a key);
```

`lookup(t,key)` returns the value associated with `key` in `t`, or throws `Not_found` if there is no value associate with `key` in `t`.

```
void resize(table_t<'a, 'b> t);
```

`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> t, '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 s);
```

`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 p);
```

`hash_stringptr(p)` returns a hash of a string pointer `p`.

```
void iter(void(@‘H f)(‘a, ‘b), table_t<‘a, ‘b> t);
```
`iter(f,t)` applies `f` to each key/value pair in `t`.

```
void iter_c(void(@‘H 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.10 <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,‘r> {
  ‘a hd;
  struct List<‘a, ‘r> *‘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> *‘r list_t<‘a,‘r>;
```
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> @List_t<‘a,‘r>;
```
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('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 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`, if there is one, and throws `Failure("hd")` if `x` is NULL.

```
list_t<'a, 'r> tl(list_t<'a, 'r> x);
```

`tl(x)` returns the tail of list `x`, if there is one, and throws `Failure("tl")` if `x` is NULL.

```
list_t<'a> copy(list_t<'a> x);
```

`copy(x)` returns a new heap-allocated copy of list `x`.

```
list_t<'a, 'r> rcopy('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(@'H 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('r, 'b(@'H 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(@'H 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('r, 'b(@'H 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.

```
xtunion exn {
  List_mismatch
};
```

`List_mismatch` is thrown when two lists don't have the same length.

```
list_t<'c> map2('c(@'H 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('r, 'c(@'H 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`.

```
void app('b(@'H 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(@'H 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(@'H 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(@'H 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(@'H f)('a), list_t<'a> x);
```

`iter(f,x)` is like `app(f,x)`, except that `f` returns `void`.

```
void iter_c(void(@‘H 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(@‘H 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(@‘H f)(‘a, ‘b, ‘c), ‘a env, list_t<‘b> x, list_t<‘c>
```

iter2_c is a version of iter2 where the function argument requires a closure as its first argument.

```
‘a fold_left(‘a(@‘H 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(@‘H 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(@‘H 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(@‘H 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, > 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(‘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('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, > 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('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, >> 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('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(@'H 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('r, int(@'H 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(@'H 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(@'H cmp)('a, 'a), list_t<'a, > x, list_t<'a, > y)
```

`merge(cmp,x,y)` returns the merge of two sorted lists, according to the `cmp` function.

```
list_t<'a, 'r3> rmerge('r3, int(@'H cmp)('a, 'a), list_t<'a> a, list_t
```

`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(@'H cmp)('a, 'a), list_t<'a, 'r> a, list_
```

`imp_merge` is an imperative version of `merge`.

```
xtunion 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 (@'H 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 (@'H 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 (@'H 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 (@'H 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.

```
list_t<$('a, 'b)@'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)@'r2, 'r1> rzip('r1 r1, 'r2 r2, list_t<'a> x, list_t<'
```

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.

```
$(list_t<'a>, list_t<'b>)split(list_t<$('a, 'b)@> 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)@> x);
```

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`.

```
$(list_t<'a, 'r1>, list_t<'b, 'r2>)rsplit('r1 r1, 'r2 r2, list_t<$('a,
```

`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('r3 r3, 'r
```

`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(@'H 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)@> 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(@'H cmp)('a, 'c), list_t<$('a, 'b)@> l, 'c x);
```

assoc_cmp(cmp,l,k) is like assoc(l,k) except that the comparison 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)@> l, 'a x);
```

mem_assoc(l,k) returns true if k is a key in association list l (according to ==).

```
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.

```
Core::opt_t<'c> check_unique(int(@'H 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 ?'H to_array(list_t<'a> x);
```

to_array(x) returns a new heap-allocated array with the same elements as list x.

```
'a ?'r rto_array('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 ?arr);
```

from_array(x) returns a new heap-allocated list with the same elements as array x.

```
list_t<'a, 'r2> rfrom_array('r2 r2, 'a ?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(@'H cmp)('a, 'a), list_t<'a> l1, list_t<'a> 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(@‘H cmp)(‘a, ‘a), list_t<‘a> l1, list_t<‘a> 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 (@‘H 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 (@‘H 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(‘r r, bool (@‘H 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(‘r r, bool (@‘H f)(‘b, ‘a), ‘b env, list_t<‘a
```

`rfilter_c` is a version of `rfilter` where the function argument requires a closure as its first argument.

C.11 <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 @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 @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 )@>)@string_and_links(do
```

`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<> s);
```

`text(s)` returns a document containing exactly the string `s`.

```
doc_t textptr(stringptr_t<> p);
```

`textptr(p)` returns a documents containing exactly the string pointed to by `p`.

`doc_t` **`hyperlink`**`(string_t<> shrt, string_t<> 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 , > doclist);`

`cats(l)` returns a document containing all of the documents in list `l`, in order.

`doc_t` **`cats_arr`**`(doc_t ?'H 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<> sep, list_t<doc_t , > 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 (@'H pp)('a), string_t<> 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<> sep, list_t<doc_t , > l0);
```

`seql` is like `seq`, except that the resulting document has line breaks after each separator.

```
doc_t ppseql(doc_t (@‘H pp)(‘a), string_t<> 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<> start, string_t<> stop, string_t<> sep, list_t<
```

`group(start,stop,sep,l)` is like `cat(text(start), seq(sep,l), text(stop))`.

```
doc_t groupl(string_t<> start, string_t<> stop, string_t<> sep, list_t
```

`groupl` is like `group` but a line break is inserted after each separator.

```
doc_t egroup(string_t<> start, string_t<> stop, string_t<> sep, list_t
```

`egroup` is like `group`, except that the empty document is returned if the list is empty.

C.12 <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> @‘r queue_t<‘a,‘r>;
```

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 );
```

`is_empty(q)` returns true if `q` contains no elements, and returns false otherwise.

```
queue_t create();
```

`create()` allocates a new, empty queue on the heap and returns it.

```
void add(queue_t<'a, >, 'a x);
```

`add(q,x)` adds `x` to the end of `q` (by side effect).

```
void radd('r, queue_t<'a, 'r>, 'a x);
```

`radd(r,q,x)` is like `add(q,x)` except that the queue lives in the region with handle `r`.

```
xtunion 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(@'H 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(@'H 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`.

C.13 <rope.h>

Defines namespace Rope, which implements character arrays that can be concatenated in constant time.

```
typedef struct Rope_node @rope_t ;
```

A value of type `rope_t` is a character array that can be efficiently concatenated.

```
rope_t from_string(string_t<>);
```

`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 ?'H);
```

`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.14 <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.

```
typedef struct Set<'a, 'r> @'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(@'H cmp)('a, 'a));
```

`empty(cmp)` creates an empty set given comparison function `cmp`. The set is heap-allocated.

```
set_t<'a, 'r> rempty('r r, int(@'H cmp)('a, 'a));
```

`rempty(r,cmp)` creates an empty set in the region with handle `r`.

```
set_t<'a> singleton(int(@'H 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(@'H 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`.

```
set_t<'a> insert(set_t<'a, > s, 'a elt);
```

`insert(s,elt)` returns a set containing all the elements of `s`, plus `elt`. The set `s` is not modified.

```
set_t<'a, 'r> rinsert('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`.

```
set_t<'a> union_two(set_t<'a, > s1, set_t<'a, > 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, > s1, set_t<'a, > 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, > s1, set_t<'a, > 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, > s, 'a elt);
```

`delete(s,elt)` returns a set whose elements are the elements of `s`, minus `elt`.

```
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.

```
list_t<'a, 'r> elements(set_t<'a, 'r> s);
```

`elements(s)` returns a list of the elements of `s`, in no particular order. Note that the returned list is allocated in the same region as the set `s`.

```
'b fold('b(@'H 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(@'H 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(@'H 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 discared. Notice that `f` cannot return `void`; use `iter` instead of `app` for that.

```
void iter(void(@'H f)('a), set_t<'a> s);
```

`iter(f,s)` is like `app(f,s)`, except that `f` must return `void`.

```
void iter_c(void(@'H f)('c, 'a), 'c env, set_t<'a> s);
```

`iter_c` is a version of `iter` where the function argument `f` requires a closure.

```
xtunion 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`.

C.15 <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> @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`.

```
xtunion exn {
  Present
};
```

`Present` is thrown when a key is present but not expected.

```
xtunion exn {
  Absent
};
```

`Absent` is thrown when a key is absent but should be present.

```
dict_t<'a, 'b> empty(int(@'H 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)@> l);
```

`inserts(d,l)` inserts each key, value pair into `d`, returning the resulting dictionary.

```
dict_t<'a, 'b> singleton(int(@'H 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(@'H 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(@'H f)('d, 'a, 'b, 'c), 'd env, dict_t<'a, 'b> d, 'c accu
```

`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(@‘H 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(@‘H 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(@‘H 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(@‘H 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(@‘H 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(@‘H 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)@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)@> 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.16 <xarray.h>

Defines namespace Xarray, which implements a datatype of extensible arrays.

```
typedef struct Xarray<'a> @xarray_t<'a>;
```

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> create_empty();
```

`create_empty()` returns a new extensible array with starting size 0.

```
xarray_t<'a> singleton(int, 'a);
```

`singleton(n,v)` returns a new extensible array with a single element `v`.

```
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 ?to_array(xarray_t<'a>);
```

`to_array(a)` returns a normal (non-extensible) array with the same elements as `a`.

```
xarray_t<'a> from_array('a ?arr);
```

`from_array(a)` returns an extensible array with the same elements as the normal (non-extensible) array `a`.

```
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.

```
void app('b(@'H 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(@'H 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(@'H 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(@'H 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(@'H 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> map_c('b(@'H 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`.

```
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

The grammar of Cyclone is derived from ISO C99. It has the following additional keywords: `abstract`, `catch`, `codegen`, `cut`, `fallthru`, `fill`, `let`, `malloc`, `namespace`, `new`, `NULL`, `region_t`, `regions`, `rmalloc`, `rnew`, `splice`, `throw`, `try`, `tunion`, `using`, `xtunion`. As in gcc, `__attribute__` is reserved as well.

The non-terminals *character-constant*, *floating-constant*, *identifier*, *integer-constant*, *string*, *type-var*, and *typedef-name* are defined lexically as in C.

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` *string* `{` *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* `;`

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: one of
`auto register static extern typedef abstract`

type-specifier:

_
`void`
`char`
`short`
`int`
`long`
`float`
`double`
`signed`
`unsigned`
enum-specifier
struct-or-union-specifier
tunion-specifier
typedef-name type-params$_{opt}$
type-var
type-var `::` *kind*
`$(` *parameter-list* `)`
`region_t <` *any-type-name* `>`

kind:
identifier
typedef-name

type-qualifier: one of
`const restrict volatile`

enum-specifier:
`enum` *identifier* `{` *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: one of
    `struct 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*

tunion-specifier:
tunion-or-xtunion identifier type-params$_{opt}$ { *tunionfield-list* }
tunion-or-xtunion region$_{opt}$ *identifier type-params*$_{opt}$
tunion-or-xtunion identifier . *identifier type-params*$_{opt}$

tunion-or-xtunion: one of
`tunion xtunion`

tunionfield-list:
tunionfield
tunionfield ;
tunionfield , *tunionfield-list*
tunionfield ; *tunionfield-list*

tunionfield-scope: one of
`extern static`

tunionfield:
tunionfield-scope identifier
tunionfield-scope identifier type-params$_{opt}$ (*parameter-list*)

declarator:
pointer$_{opt}$ *direct-declarator*

direct-declarator:
identifier
(*declarator*)
direct-declarator [*assignment-expression*$_{opt}$]
direct-declarator (*parameter-type-list*)
direct-declarator (; *effect-set*)
direct-declarator (*identifier-list*$_{opt}$)
direct-declarator < *type-name-list* >

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:
    _
    'H
    type-var
    type-var :: *kind*

type-qualifier-list:
    type-qualifier
    type-qualifier-list type-qualifier

parameter-type-list:
    parameter-list
    parameter-list , ...

optional-effect:
    (empty)
    ; *effect-set*

optional-inject:
    (empty)
    identifier

effect-set:
    atomic-effect
    atomic-effect + *effect-set*

atomic-effect:
    { }
    { *region-set* }
    type-var
    type-var :: *kind*

region-set:
    type-var
    type-var `,` *region-set*
    type-var `::` *kind*
    type-var `::` *kind* `,` *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:
    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*$_{opt}$ `]`
    direct-abstract-declarator$_{opt}$ `(` *parameter-type-list*$_{opt}$ `)`
    direct-abstract-declarator$_{opt}$ `(` `;` *effect-set* `)`
    direct-abstract-declarator$_{opt}$ `[ ? ]`
    direct-abstract-declarator `<` *type-name-list* `>`

statement:
    labeled-statement
    expression-statement
    compound-statement
    selection-statement
    iteration-statement
    jump-statement
    `region` *identifier statement*
    `region` `<` *type-var* `>` *identifier statement*
    `cut` *statement*
    `splice` *statement*

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*)
$(*tuple-pattern-list*)
identifier type-params$_{opt}$ { }
identifier type-params$_{opt}$ { *field-pattern-list* }
& *pattern*
* *identifier*

tuple-pattern-list:
(empty)
pattern
tuple-pattern-list , *pattern*

field-pattern:
pattern
designation pattern

field-pattern-list:
field-pattern
field-pattern-list , *field-pattern*

expression:
assignment-expression
expression , *assignment-expression*

assignment-expression:
conditional-expression
unary-expression assignment-operator 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*)
expression `. size`

unary-operator: one of
`& * + - ~ !`

postfix-expression:
primary-expression
postfix-expression [*expression*]

postfix-expression `(` `)`
postfix-expression `(` *argument-expression-list* `)`
postfix-expression `.` *identifier*
postfix-expression `->` *identifier*
postfix-expression `++`
postfix-expression `--`
`(` *type-name* `)` `{` *initializer-list* `}`
`(` *type-name* `)` `{` *initializer-list* `,` `}`
`fill` `(` *expression* `)`
`codegen` `(` *function-definition* `)`

primary-expression:
identifier
constant
string
`(` *expression* `)`
identifier `<>`
identifier `@` `<` *type-name-list* `>`
`$(` *argument-expression-list* `)`
identifier `{` *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, and has only been tested on Win32 (Cygnus) and Linux (Red Hat 6.2) platforms. 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 some changes.

To install and use Cyclone, you'll need to use the Gnu utilities, including GCC (the Gnu C compiler) and Gnu-Make. For Win32, 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

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.

-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.

-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.

-S Stop after producing assembly code.

-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 currently only useful for compiler developers, as the debugging information reflects the C code that the Cyclone code is compiled to, and not the Cyclone code itself.

-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.

-use-cpp*path* Indicate which preprocessor to use.

-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.