

Project 1

CMSC 838Z, Spring 2004

Due Thursday, February 12 (e-mail to me)

For this assignment, you will massage your muscles in understanding formal descriptions of type systems for simple types, and get some experience in using quantified types in Cyclone. Finally, you will implement a smallish server in Cyclone, by porting it from C, to get a feel for the Cyclone language as a whole.

Exercises

Simple Type Systems

- 1.1 (a) Using the formal notation in the Cardelli paper, write the type for lists containing Natural numbers. Hint: you will need to combine pair or record types, union or variant types, and recursive types.
- (b) In the Cardelli formal syntax, write the function that calculates the length of the list.

Hint: You will have to use the fix-point operator Y to do this. You can assume this is present in your language as primitive. Recall that Y has the type $(\tau \rightarrow \tau) \rightarrow \tau$. That is, you provide it a function that takes itself as an argument for the recursive call, and it returns a version where that recursive call is “joined up” with the function itself. For example, to write the function that calculates factorial, we would write:

$$Y(\lambda \text{fact} : \text{Nat} \rightarrow \text{Nat} . \lambda x : \text{Nat} . \\ \text{if isZero } x \text{ then } 1 \text{ else } x * \text{fact } (x - 1))$$

(We assume the presence of subtraction and multiplication operators, which we could write ourselves.) You can see that the function passed to Y has type $(\text{Nat} \rightarrow \text{Nat}) \rightarrow \text{Nat} \rightarrow \text{Nat}$; the final result will have $\text{Nat} \rightarrow \text{Nat}$.

- 1.2 Assuming that we have $\Gamma = \text{sub} : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}, \text{mult} : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}$, write out the typing derivation that proves the following judgment:

$$\Gamma \vdash \lambda \text{fact} : \text{Nat} \rightarrow \text{Nat} . \lambda x : \text{Nat} . \\ \text{if isZero } x \text{ then } 1 \\ \text{else mult } x \text{ (fact (sub } x \text{ 1))} : (\text{Nat} \rightarrow \text{Nat}) \rightarrow \text{Nat} \rightarrow \text{Nat}$$

Quantified Types in Cyclone

- 2.1 Use Cyclone to write a simple library for lists. Use the following definition of the list type:

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

This definition differs from the one in the Cyclone list library in that it always uses the *heap region* 'H for allocating its memory. Implement the following functions. (These are implemented in the Cyclone list library; don't look there unless you get stuck!)

- `int length(list_t<'a> x)`: returns the length of the given list.
- `list_t<'b> map('b f('a), list_t<'a> x)`: If `x` has elements x_1 through x_n , then `map(f, x)` returns the list containing elements $f(x_1) \dots f(x_n)$.
- `'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 an exception `Not_found` if `k` is not paired with any value in `l`. `assoc` uses `==` to decide if `k` is a key in `l`.

- 2.2 Use Cyclone to implement a library for balanced trees which is parametrically polymorphic (universally quantified) with respect to the elements stored in the tree. Write functions to create a new tree, add an element to the tree, remove an element from the tree, and test for membership in the tree. You can adapt existing C code if you like. For example, an implementation of AVL trees can be found at [\[1\]](#), and an implementation of red-black trees can be found at [\[2\]](#) (I haven't fiddled with these myself, so use at your own risk). I recommend, as with lists above, you do everything in the heap region; that is, when in doubt (i.e. when a weird error message crops up), annotate pointers with 'H.
- 2.3 Use Cyclone to implement a small library that supports callbacks for event processing. Your datastructures should look like the following:

```
struct Event {
    int event_type;
};
typedef struct Event @'H event_t;

struct CB {
    <'env> : regions('env) > 'H
```

```

    void (@f)(event_t, 'env);
    'env env;
};
typedef struct CB @'H cb_t;

```

(You can ignore the `regions('env) > 'H` annotation here and in prototypes below; if you want to understand more, see the Cyclone manual, or ask me.) In particular, a callback is, in essence, a function `f`, which is called by the system when some event occurs. This function is called with a descriptor of the event (a `event_t` datastructure, which for our purposes will just contain an integer naming the event), as well as an *environment* that is specific to the function being called. Users use the following function to create callbacks (which you must implement):

```
cb_t make_cb(void f(event_t, 'env), 'env x : regions('env) > 'H);
```

For example, I might write a function

```

void print_event(event_t e, FILE *log) {
    fprintf(log, "Got event %d\n", e->event_type);
}

```

I can create a callback for this function by calling `make_cb` as follows:

```
cb_t cb = make_cb(print_event, stderr);
```

Now, I can register this callback with the event processing system, using the following function:

```
void register_cb(cb_t f, struct Event *e);
```

This indicates that callback `f` should be invoked whenever an event having the type `e->event_type` is signalled. If `e` is `NULL`, `f` should be called for all events. I can signal an event has occurred by using this routine:

```
void signal_event(event_t e);
```

This stores the event `e` in a queue of events. To process events, I can invoke the function:

```
void process_event();
```

This removes the first event from the queue and processes it (that is, invokes all the callbacks that are registered for it). We can also write the function

```
void process_all_events();
```

This function will not return until all events in the event queue have been processed. Finally, we have the function

```
void deregister_cb(struct CB *f);
```

This is used to deregister the callback `f`. If `f` is `NULL`, then it deregisters the currently executing callback.

You can test out your library by writing a little program that registers a few callbacks and loops, alternately signalling and processing events. Try writing a program that registers and deregisters callbacks within callbacks, and signals events within callbacks, to see how things can interact.

Project

Your final task is to write or port a small C program in Cyclone. You can use any program you like, but it should be at least 500 lines. There are a number of reasonable server programs on sourceforge.net or freshmeat.net. One program that I was able to port in about 45 minutes is the Cheetah webserver, available at: http://freshmeat.net/projects/cheetahd/?topic_id=250.

Here are some tips for porting:

- Use Cyclone's semi-automatic porting tool to get started. It doesn't work terribly well, but it will eliminate some of the tedium of converting `*` pointers to `?` pointers, for example. See the manual for details.
- If you are porting a program with a configure script, run the script first, and then make copies of the `.c` files to be `.cyc` files, and then run the porting tool using the `#define`'s (with `-D`) that the C code is normally compiled with. This may not work, though, since Cyclone does not necessarily have all of the headers available to C programs. So, you might have to modify the results of configure manually. I didn't have any problems with cheetah.
- The manual has lots of types for going from C to Cyclone, and for porting, which you should look at. One thing that is bound to come up is the use of `malloc()`. Cyclone treats this as a keyword in a very specialized way; its argument must either be
 - a `sizeof()` expression, so that Cyclone knows the size and type to be returned.
 - an integer-typed expression, in which case Cyclone assumes you are allocated character data.

Oftentimes, you want to remove the casts that precede calls to `malloc`, which only serve to confuse things in Cyclone. Finally, you will probably have to turn functions that act as wrappers for `malloc` into macros, so that the types are properly inferred.

Also, Cyclone is picky about uses of `bzero` and `memcpy`, which in C treat data as binary arrays.

- If you are porting a program that uses sockets, and in particular `sockaddr` structs, you may need to remove some casts, do to the weird way Cyclone handles these functions. For example, in cheetah you have to change the code

```
bind(sockfd, (struct sockaddr *)&my_addr, sizeof (struct sockaddr))
```

```
bind(sockfd, &my_addr, sizeof (struct sockaddr))
```

- Some issues with regions are bound to come up. When in doubt, assign region values to 'H, and hopefully that will help.
- I'm going to try to get a Wiki up over the weekend, so that you can share ideas and insights about the porting process. I'll try to monitor it during the week while I'm gone.