Announcements

- Prelim #2 on Tuesday 11/13 7:30-9:00PM in B17 Upson
  - Graded tonight, back in section
  - Coverage through Monday (which really means today plus some details in section)
- You can take the exam early if you have a conflict. Come to my office, 4158 Upson, at 5:30PM. Email me in advance so I have a head count.

Abstraction

- You’ve seen modules, structures, signatures
    - The support in OCaml for abstraction, for hiding implementation details, and for namespace management
    - Other languages have similar tools, some better, some worse

- Today we are going to talk about a small abstraction example, give a more interesting example, then return to the general issue.

- Small example:
  - Graphics package in C, or a similar language
  - Points package for 2D points
  - TA implements, Ramin uses

- Spec contains the obvious functions:
  - Predicate (is this a point?)
  - Constructor (make a point from x y coordinates)
  - Accessors (get the x or y coordinates from a point)
  - Invariants, like point_x(make_point(*x*,*y*)) = *x*, point_y(make_point(*x*,*y*))=*y*

- TA will create an implementation of the spec

- The user (Ramin) is only supposed to rely on the documented properties of the implementation
  - And NOT on the accidental ones

- In real life, this often goes awry and results in a catastrophic software failure
  - such as Windows

- Let’s suppose that TA’s initial implementation is the obvious one,
    - she uses 2 consecutive memory locations to represent the X and Y coordinates

- In a language like C, this information “leaks” very easily
    - Partly because it is not strongly typed
    - OCaml prevents this (hard to make a good example in ML!)

- Ramin can take advantage of TA’s implementation to do “clever” things
    - For example, suppose he has a bunch of points in an array and wants to check if any lie on an axis (X or Y)
        - Just multiply them all together, check for 0
    - Or to determine quadrant, multiply locations and check the sign
        - Without calling point_x or point_y

- This can be genuinely faster, in terms of execution time, and sometimes this really matters
- But usually it leads to a disaster
    - Premature emphasis on performance!

- TA changes her implementation, for example by using polar coordinates, or switching X and Y
    - Or adding a “magic number” to the beginning of each point, as a primitive form of type checking
    - After all, her default implementation of point is too weak
        - any pair of numbers is a point!
- Why do this? For example, determining that a pair of points lie along a line through the origin is trivial in polar coordinates, and perhaps TA wants to add this to the spec since she can support it efficiently

- A bigger disaster can occur if the change TA makes is more subtle
- Suppose she keeps most points in (x,y) order, but some in (y,x)? Or suppose that on some computers they are in (x,y) and some they are in (y,x)?
    - “big-endian” vs “small-endian” hardware
- Ramin’s code might not break, but his users (or his users users) might suddenly become flaky
- Tracking down this kind of bug is a total nightmare

- Examples: In Windows 3.0, a message was sent to a window if it was being resized or moved that its size was potentially changing.
    - In Windows 3.1, they fixed this so the message was only sent on resizing, not on moving
    - But the most popular application (Lotus 123) depended on getting this message even on a move, so it broke
    - See: “Make Compatible” on Wikipedia
- It is well-known within Microsoft that certain system calls would try to figure out what program was calling them (by looking up the call stack) and change their behavior to try to do what was expected
- Maintaining compatibility is hard is a huge, painful effort
- It’s no wonder that Intel and Microsoft are huge, and sometimes lumbering, companies!

- Typically failures come from:
    - Eagerness to get to work writing code
    - Too early focus on performance
        - There is no Moore’s law for programmers!
    - Inexperience with writing code big enough to need specs

- Spec design itself is something of an art form
- Remove any un-needed details, only describe what the implementation is required to do
    - What it does versus how it does it
    - Leave implementor (TA) freedom to improve and innovate!
    - Potentially might specify resources needed (time, space)
        - This is a somewhat controversial point, but there are situations where it is clearly needed
        - Such as most data structures (consider finite maps)
- You want these to be designed by both sides
    - Just like a good business contract

- You need a spec to decouple the user of (something) from the builder of (something)
    - User and builder agree on the spec
    - User can freely assume the implementation meets the spec
    - Builder can implement the spec as they see fit!
- Good examples of complex specs: standards
    - 802.11b/bluetooth, HTML/XML, MP3/JPEG, TCP/IP
        - Also: electricity, construction, drugs, DICOM, etc.
    - Complex design process w/ competing commercial interests
    - Reference implementation for debugging
- Implementation is not the spec!
    - Huge issue for, e.g., Windows, HTML, Office, etc
    - Retrofitting a spec is a huge pain
        - For example, Office file formats switched to XML

Assurance

So, you have implemented your program using the specification methodology outlined in the last couple of lectures. But does it really work?

You need **assurance**: confidence that the program works.

There are two main strategies for gaining assurance: **verification** and **testing**, and many variations on each.

Ideally we would like to show convincingly that a program works correctly on all possible inputs that it could be provided. (SOUNDNESS, as with threads and concurrency.)

This is often too strong a goal; we must settle for a assurance process that increases our confidence that the program works correctly, or perhaps that proves definitively that it avoids only certain kinds of errors.

Verification

In the *verification* approach, we use the program and its specifications to argue either formally or informally that the program satisfies all its specification and therefore works correctly on all possible inputs.

The value of verification is that if carried out thoroughly, it produces a convincing demonstration that the program really does work on all possible inputs.

For example, the ML type checker is a limited verifier that ensures that the program does not contain run-time type errors no matter what the inputs are. But it doesn't guarantee that the program does not contain other kinds of errors.

Note that there are limits on what ML checks (e.g., no positive integers), and in fact most compilers are much “weaker” than they could be. This is partly due to efficiency and partly to make it possible for programmers to model the type checker in their heads.

There are even tools available to help do program verification, based on automated theorem provers.

However, these tools haven't really caught on; one problem is that it is too hard for most programmers to write down specifications precise enough for these tools to work. Informal verification is often more effective.

But even informal verification can be a difficult process to carry out; it is most effective when applied to small data abstractions and algorithms. In subsequent lectures we will see some more involved informal arguments for correctness.

Testing

In the *testing* approach, we actually run the program or parts of the program on various inputs and see whether the behavior of the code is as expected.

By comparing the actual results of the program with the expected results, we find out whether the program really works on the particular inputs we try it on.

One weakness of testing is that unless we try all possible inputs to the program, we can't be sure that it works on all of them.

Another weakness is that some programs are nondeterministic (particularly if they use concurrency) and the same test case may give different results on different runs.

This contrasts with verification, in which we attempt to prove that the program always works. If carried out carefully, however, testing can give us useful assurance that the program works, at lower cost than formal or even informal verification.

Coverage

We would like to know that the program works on all possible inputs. The problem with testing is that it is usually infeasible to try all the possible inputs. For example, suppose that we are implementing a module that provides an abstract data type for rational numbers. One of its operations might be an addition function `plus`, e.g.:

```
type rational = int*int
(* AF: (p,q) represents the rational number p/q
   RI: q is not 0 and either p=0 or gcd(p,q)=1 *)

(* Creates the rational number p/q.
 * Checks: q is not 0 *)
let create(p:int, q:int) : rational = ...
let toReal(r: rational): real = ...
...
(* plus(r1,r2) is r1 + r2. *)
let plus(r1: rational, r2: rational) : rational = ...
```

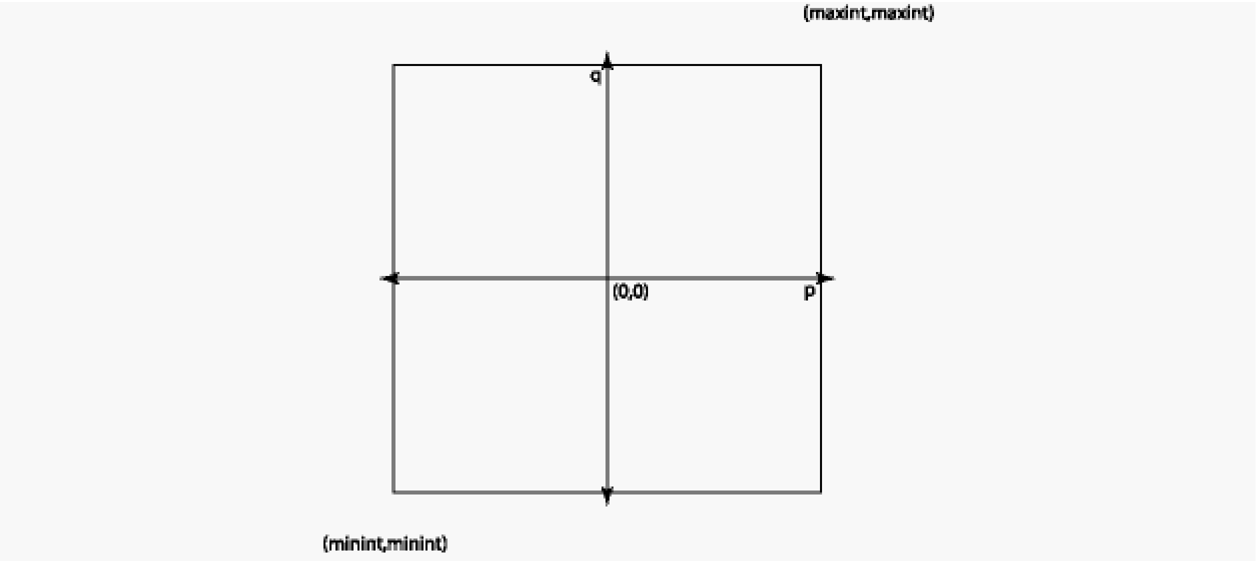
What would it take to exhaustively test just this one routine? We'd want to try all possible rationals as both the `r1` and `r2` arguments.

A rational is formed from two ints, and there are 2^{32}ints on most machines. Therefore there are $2^{32}×2^{32}×2^{32}×2^{32} = 2^{128}$ possible inputs fo the `plus`routine. Even if we test one addition every nanosecond (perhaps by using a lot of machines in parallel), it will take about 10^{29} years to finish testing this one routine.

Clearly we can't test software exhaustively. But that doesn't mean we should give up on testing. It just means that we need to think carefully about what our test cases should be so that they are as effective as possible at convincing us that the code works.

Consider our `create` routine, above. It takes in two integers `p` and `q` as arguments.

How should we go about selecting a relatively small number of test cases that will convince us that the function works correctly on all possible inputs? We can visualize the space of all possible inputs as a large square:



There are about 2^{64} points in this square, so we can't afford to test them all. And testing them all is going to mostly be a waste of time—most of the possible inputs provide nothing new.

We need a way to find a set of points in this space to test that are interesting and will give a good sense of the behavior of the program across the whole space.

Input spaces generally comprise a number of subsets in which the behavior of the code is similar in some essential fashion across the entire subset. We don't get any additional information by testing more than one input from each such subset.

If we test all the interesting regions of the input space, we have achieved **coverage**. We want tests that in some useful sense **cover** the space of possible program inputs.

Testing against the specification

In selecting our test cases for good coverage, we might want to consider both the specification and the implementation of the program or program module being tested. It turns out that we can often do a pretty good job of picking test cases by just looking at the specification and ignoring the implementation. This is known as **black-box testing**.

The idea is that we think of the code as a black box about which all we can see is its surface: its specification. We pick test cases by looking at how the specification implicitly introduces boundaries that divide the space of possible inputs into different regions.

When writing black-box test cases, we ask ourselves what set of test cases that will produce distinctive behavior as predicted by the specification. It is important to try out both "typical inputs" and inputs that are "corner cases".

A common error is to only test typical inputs, with the result that the program usually works but fails in less frequent situations. Sometimes those are the situations where we really need the code to work!

Here are some ideas for how to test the `create` function:

- Looking at the square above, we see that it has boundaries at `minint` (usually -2^{31}) and `maxint` (usually $2^{31}-1$). We want to try to construct rationals at the corners and along the sides of the square, e.g. `create(minint, minint)`, `create(maxint, 2)`, etc.
- The line p=0 is important because p/q is zero all along it. We should try (0,q) for various values of q.
- We should try some typical (p,q) pairs in all four quadrants of the space.
- We should try both (p,q) pairs in which q divides evenly into p, and pairs in which q does not divide into p.
- Pairs of the form (1,q),(-1,q),(p,1),(p,-1) for various p and q also may be interesting given the properties of rational numbers.

The specification also says that the code will check that q is not zero. We should construct some test cases to ensure this checking is done as advertised. Trying (1,0), (maxint,0),(minint,0),(-1,0), (0,0) would probably be an adequate set of black-box tests.

Of course, we can't tell much by simply creating a rational number; in testing we'll have to use some other routine of rationals, such as `toReal`, to *observe* the rational created and see that it conforms to our expectations. We shouldn't confuse this with testing of `toReal`, however; we'll want to also test that routine, by invoking create with arguments that correspond to ways of cutting up the space of real numbers.

Here is another example: consider the routine **`max`**:

```
(* Return the maximum element in the list. *)
let max: int list -> int
```

What is a good set of black box test cases? Here the input space is the set of all possible lists of ints. We need to try some typical inputs and also consider boundary cases. Based on this spec, boundary cases include the following:

- A list containing one element. In fact, an empty list is probably the first boundary case we think of. Looking at the spec above, we realize that it doesn't specify what happens in the case of an empty list. Thus, thinking about boundary cases is also useful in identifying errors in the specification.
- A list containing two elements.
- A list in which the maximum is the first element. Or the last element. Or somewhere in the middle of the list.
- A list in which every element is equal.
- A list in which the elements are arranged in ascending sorted order, and one in which they are arranged in descending sorted order.
- A list in which the maximum element is **`maxint`**, and a list in which the maximum element is **`minint`**.

Aliasing

When a routine has side effects or manipulates mutable data structures, another important class of boundary conditions to check for is calls that **alias** two different arguments. For example, if we have a routine that copies elements from one set to another, we should try applying this routine to the same set to see whether we get the expected result. This routine might easily be implemented in a way that causes it not to work when the inputs are aliased.

```
(* copy(s1,s2): add all the elements of s2 into the set s1 *)
let copy: set * set -> unit
...
(* test case: *)
let s: set = ...
copy(s,s)
```

Often programmers do not think enough about the possibility of aliasing.

Summary

Black-box testing has some important advantages:

- It doesn't require that we see the code we are testing. Sometimes code will not be available in source code form, yet we can still construct useful test cases without it. The person writing the test cases does not need to understand the implementation.
- The test cases do not depend on the implementation. They can be written in parallel with or before the implementation. Further, good black-box test cases do not need to be changed. even if the implementation is completely rewritten.
- Constructing black-box test cases causes the programmer to think carefully about the specification and its implications. Many specification errors are caught this way.

The disadvantage of black box testing is that its coverage may not be as high as we'd like, because it has to work without the implementation. But it's a good place to start when writing test cases.

Testing against the implementation

Using the representation invariant

Looking at the implementation gives us more information about what is likely to give us good coverage of the space of possible inputs. Testing with knowledge of the implementation is known as **glass-box testing**.

A simple first step is to look at the abstraction function and representation invariant for hints about what boundaries may exist in the space of values manipulated by a data abstraction. The rep invariant is a particularly effective tool for constructing useful test cases.

Looking at the rep invariant of the rational data abstraction above, we see that it requires that q is non-zero and that the gcd of p and q is 1. Therefore we should construct test cases that make q as close to 0 as possible (1 or -1) , test cases in which p is 0, and test cases in which the gcd turns out to be 1 in relatively interesting ways, such as by having p=1 or q=1, or test cases in which a bug in the arithmetic algorithms might reasonably break the gcd property (e.g., 3/5 + -3/5, 7/20+13/20).

Path completeness

Another way we can identify interesting regions of the input space is by trying to find a set of test cases that exercises every **path** through the program text. Test cases that collectively exercise all paths are said to be **path-complete**. At a minimum, path-completeness requires that for every line of code, and even for every expression in the program, there should be a test case that causes it to be executed. Any unexecuted code could contain a bug if has never been tested.

For example, here is our code for taking the union of two sets represented as lists:

```
let union(s1, s2) =
  fold_left (function (x,s) -> if contains(x,s) then s else x::s) s1 s2
```

Path completeness requires at a minimum that we write test cases that exercise both the "`then`" and "`else`" branches of the `if` statement.

For true path-completeness we must consider all possible execution paths from start to finish of each function, and try to exercise every distinct path. In general this is infeasible, because there are too many paths. For example, a program with a loop in it has at least one path for each number of times that the loop can execute. The code for `union` actually has many paths because `foldl` may invoke its function many times, with different branches taken on different executions. If there are several such loops, the number of paths can easily become intractably large. A good approach is to think of the set of paths as the space that we are trying to explore, and to identify boundary cases within this space that are worth testing.

For example, for a program containing a loop, we want test cases that cause the loop to exercise 0, 1, and some*n*>=2 times. If there are several paths within the loop, we will want test cases for each number of loop iterations that exercise all these paths.

Testing strategies

So far we have mostly talked about selecting test cases that ensure that modules meet their specification. How should we go about testing? There are two approaches, which are both useful: **integration testing** and **unit testing**.

In integration testing we try the program as a whole and check that its behavior is as expected. We construct test cases using the techniques described above.

For example, if a program reads an input file we will want to consider boundary cases such as the file not existing or the file being empty or incorrectly formatted, in addition to data-specific boundary cases of the sort discussed above.

The problem with integration testing is that we are unlikely to be able to fully test program modules by running the program with appropriately chosen inputs; programs typically do not exercise the full functionality of the modules that make them up. Integration testing can leave undiscovered bugs that surface later when the program is changed in some way.

Therefore we should also perform unit testing to check the individual modules of the program. To test modules, we have to write code whose purpose is only to test out the module. This code is known as a **test harness**.

A test harness should thoroughly test each of the operations provided by the module so that any possible use of the module by the containing program will also work correctly. Typically the test harness is written in the same programming language as the module itself. It can be hard to motivate oneself to write code that is not going to be part of the final program, but the increased assurance that is obtained by unit testing often makes it worthwhile.

Exploiting the representation invariant

If the module being tested has a **`repOK`** operation or equivalent, it can be very useful in testing. The problem with abstract data types is that they can only be manipulated through their interface; even if a data structure has gone bad, it may be difficult to find a way to observe the problem through the external interface. Because **`repOK`** checks the internal consistency of the representation in a thorough way, the test harness can use it after each module operation to gain additional confidence that the module works correctly.

Exhaustive testing: the small-counterexample hypothesis

True exhaustive testing is infeasible, but a limited form is very often successful in finding bugs. The observation is that bugs in programs can usually be reproduced with a small test case. If this is true, we can find that test case by exhaustively trying small test cases. By limiting ourselves to test cases smaller than some fixed size, we keep the number of test cases manageable.

For example, if we were testing red-black trees, we might try constructing all possible red-black trees of up to 6 elements and performing operations on them. Some thought is usually required to keep the input state space from exploding. For example, we know that only the order of the elements matters to the algorithm, so we don't need to try all possible sets of 6 values; we could just insert the numbers 1 through 6 into the red-black tree in all possible orders. This results in only 720 separate cases that we can write a test harness to generate automatically, or 6^6 = 46656 cases if duplicate elements are allowed. Testing this many cases is feasible and frequently useful.

Regression tests

Once we have constructed a nice set of test cases with good coverage and convinced ourselves that the code works on them, what do we do with the test cases? Perhaps we should throw them away -- after all, the program works; why would we need test cases?

One reason is that software is not static; it evolves over times as bugs are fixed in it and new features added. The test cases developed for one version of the software are likely to be useful in later versions as well. The time invested in developing good test cases can be amortized across the whole lifetime of the software being developed. In fact, in a long-running software project, the test cases form a **test suite** that is part of the project, and is stored along with the code of the project proper. Whenever the software is updated, **regression testing** is performed: the test cases that have been developed are rerun on the program to make sure that the program still works properly.

Often regression testing can be automated; the output of the program when run on the existing test cases is saved in a file. When the new version of the program is run on the regression tests, its output is compared to the previous output to make sure that it has not changed -- that the program has not "regressed".

There are tools available, such as **expect**, that make the writing of test harnesses for even complex regression tests easier. Expect makes it easy to generate program input and to automatically compare the output against the expected output while ignoring unimportant differences.

When an existing program is updated, new test cases typically need to be added to the regression test suite. If the program was updated to fix a bug, a test case should be added that causes the bug to show up. If new features are added, new test cases relating to that feature should be added as well.

Testing and debugging

Testing only tells us whether a program is correct or not; it doesn't usually tell us where the bug is. But careful unit testing of a program can help considerably in debugging. One of the hardest things in debugging a complex program is knowing what to trust, because you don't know where the error is. If the individual modules of the program have been carefully unit tested, this can help you debug because you gain confidence that those modules can be relied upon to perform their function. Good specifications are also useful because they allow you to decide whether or not a given piece of code is doing what it is supposed to in isolation from the rest of the program.

Once you find a bug, it is tempting to slap a quick fix into the code and move on. This is quite dangerous. Industry statistics show that about *one in every three* bug fixes introduces a new bug! If a bug is difficult to find, it is often because the program logic is complex and hard to reason about. You should think carefully about why the bug happened in the first place and what the right solution to the problem is. Too often programmers simply put a "band-aid" on the code and hope that it does the trick. Regression testing is important whenever a bug fix is introduced, but nothing can replace careful thinking about the code.

More reading

- S.L. Pfleeger, *Software Engineering: Theory and Practice*.
- G.J. Myers, *The Art of Software Testing*.