Lesson 14

STRUCTURES AND SIGNATURES

July 6, 2023

Lesson Plan

1. Structures and Namespaces
2. Signatures and Interfaces
3. Abstraction
4. Information Hiding
5. Representation Independence

1 - Structures and Namespaces

Organizing Software

So far we have explored the core SML language features that let us write interesting functions.

We've seen functions and datatypes, exceptions and variables, and many applications thereof. These are essential for the purpose of solving interesting problems, but what about problems that are not strictly computational?

Code is not just meant to be written, it is meant to be used, documented, and organized. We are interested in the problem of organizing software.

Namespaces

One particular idea for organizing software is the idea of **namespaces**.

Def A **namespace** is a particular group of defined elements which live separately from others. In a programming language, there will usually be ways of referring to different namespaces, to separate out used names from each other.

For instance, we exhibit this with the syntax that we have used so far, with names such as `List.length` and `Int.compare`. The `compare` function lives in the namespace associated to `Int`, and the `length` function lives in the namespace associated to `List`.

Note This means there could be definitions of the same name in different namespaces! We could have separate `List.compare` and `Int.compare` functions.

Structures

It turns out that `List` and `Int` are just examples of what SML calls **modules**, or **structures**.

Def A **module** or **structure** is a grouping of declarations underneath a particular name.

For instance, we could write the following syntax:

```
structure Foo =
  struct
    datatype t = Bar of int
    val x = Bar 5
    exception E of t
  end
```

such that the names `t`, `Bar`, `x`, and `E` are entirely local to the structure `Foo`.

Separation of Concerns

This means that outside the structure, we could write:

```
val y = Foo.Bar 1
val z = raise Foo.E y
```

to access the inner contents.

This is helpful for when we're working with many definitions, and we want to group them distinctly! For instance, we might be interested in functions specific to a particular part of our codebase. With modules, we can group domain-specific logic under the module for that specific application.

Note Because modules can contain exceptions and types, and their components can be referred to by name, they are more powerful than just simple tuples, which can only contain values organized by position![1]

[1] Technically, SML has a notion of **records**, which are tuples with fields organized by name. But records also can only contain values, and not things like types.

A Collection of Definitions

How does this differ from just organizing the contents of modules into their own files? For one, files do not promote their own personal namespacing in the same way as modules. If we have two files, with contents:

```
val x = 2
```

```
val y = 3
```

then loading both of these SML files will result in an environment containing both `x` and `y`. We want to be able to separate out unrelated parts by more than just the file they are located in.

But, modules share the same drawbacks as files in other senses. Both are just collections of definitions, at the end of the day. Consider the problem of the `msort` function.

An Example File

```
fun split [] = ([], [])
  | split [x] = ([x], [])
  | split (x::y::xs) =
      let
        val (A, B) = split xs
      in
        (x::A, y::B)
      end

fun merge ([], R) = R
  | merge (L, []) = L
  | merge (x::xs, y::ys) =
      case Int.compare (x, y) of
        LESS => x :: merge (xs, y::ys)
      | _ => y :: merge (x::xs, ys)

fun msort [] = []
  | msort [x] = [x]
  | msort L =
      let
        val (A, B) = split L
      in
        merge (msort A, msort B)
      end
```

Here’s an example of a potential SML file, call it `msort.sml`.

Namespace Clutter

The pitfalls of packaging code as just a collection of definitions is that there's no way to be clear about which parts are important!

The definition of `msort` relies on two helper functions, `merge` and `split`. We cannot avoid writing these functions, but because they are written at the top-level, loading this SML file will also load those functions into the namespace automatically.

This causes **namespace clutter**, because now we have introduced two definitions for `merge` and `split`, when we only wanted them for the purpose of writing `msort`!

A Local Approach

One way to get around this is with something called a **local definition**.

This looks like:

```
local
  fun split L = (* ... *)
  fun merge (L1, L2) = (* ... *)
in
  fun msort [] = []
    | msort [x] = [x]
    | msort L = (* ... *)
end
```

This form makes it more clear where the dependencies are. But it’s also a burden on the writer of the code! We’d like a solution which ideally doesn’t burden the source code.

What’s in a File?

Additionally, we want to make it very clear to a user of a library, what the contents of the library are, and what it approximately does. How can we somehow convey the contents of a collection of code, without necessarily needing to alter the source significantly?

We could just write this at the top of the file, for instance:

```
(* This file contains ‘msort : int list -> int list’ *)

(* IGNORE *)
fun split L = (* ... *)
fun merge (L1, L2) = (* ... *)
fun msort L = (* ... *)
```

But now, what happens if we refactor our code?

What’s in a File?

```
(* This file contains ‘msort : int list -> int list’ *)

(* IGNORE *)
fun split L = (* ... *)
fun merge cmp (L1, L2) = (* ... *)
fun msort cmp L = (* ... *)
```

Suppose that we change our implementation of `msort`, to one which is now `msort : ('a * 'a -> order) -> 'a list -> 'a list`.

Well, now we have to go and change our documentation! Our comment isn’t actually checked for accuracy, so it might be the case that it gets outdated. We can’t necessarily trust documentation, because it’s not verified by anything.
Can we do better?

An Interface

Idea What if we had an interface for our file that was checked by the compiler?

What kind of information are we interested in having present in this interface?

We will check for:

- the presence of certain declarations
- the type of value bindings within the file

This is the idea of a **signature**.

2 - Signatures and Interfaces

Signatures

Def A **signature** is an SML construct, consisting of a collection of **specifications** for things such as types, values, and exceptions.

Here is an example of a signature we could have for a module containing our `msort` code, instead of putting it into a separate file `msort.sml`:

```
signature MSORT =
  sig
    val msort : (’a * ’a -> order) -> ’a list -> ’a list
  end
```

By convention, we usually put the name of a signature in all-caps.

This is the signature of a module which publicly contains just a single value, which is a function `msort` of type `(’a * ’a -> order) -> ’a list -> ’a list`.

A Module for msort

Suppose we wanted to define our new shiny Msort module. We might write:

```
structure Msort =
  struct
    fun split L = (* ... *)
    fun merge cmp (L1, L2) = (* ... *)
    fun msort cmp L = (* ... *)
  end
```

Now, we can access our msort function via Msort.msort.

But, we haven't yet looped in our MSORT signature! This means that as currently written, we can still write Msort.split and Msort.merge. We also don't have any check that msort is truly of type
('a * 'a -> order) -> 'a list -> 'a list.

Ascription

Def The act of specifying that a module should implement a given signature is called **ascription**.

To ensure that our `Msort` module has to be compatible with the `MSORT` signature, we have to perform **ascription**. We write it like this:

```
structure Msort : MSORT =
  struct
    fun split L = (* ... *)
    fun merge cmp (L1, L2) = (* ... *)
    fun msort cmp L = (* ... *)
  end
```

This means that, after this ascription, the *only declarations* that are visible within `Msort` from a user are the declarations contained within the `MSORT` signature!

Ascription and Typechecking

This means that we cannot use `Msort.split` or `Msort.merge`, because we're restricted to knowledge of the interface.

The other great advantage of ascription is that it only succeeds if all of the declarations that are present in the signature are present in the structure, *and* the declarations in the structure must have consistent types with those in the signature.

So if we implemented `msort` with any type other than `('a * 'a -> order) -> 'a list -> 'a list`, the program would fail to compile. This means that our interfaces are not just well-specified, but significantly stronger than comments. They are guaranteed to be safe.

3 - Abstraction

Abstraction Boundaries

One of the greatest wins in computer science is the idea of **abstraction**.

This means, essentially, deliberately forming higher-level models of things which ignore irrelevant details. This helps us a lot in understanding things with our human brains. For instance, we choose to think about evaluation of expressions, rather than flipping of bits in computer hardware.

At the software level, abstraction abounds as well. We want to choose to ignore the parts of implementations that do not matter to us. This is the entire point of specifications.

Signatures and structures offer us a way to enforce this idea of abstraction.

An Abstract Sort

The way that we defined `MSORT` and `Msort` earlier are actually in violation of this idea of abstraction!

Consider the signature `MSORT`:

```
signature MSORT =
  sig
    val msort : ('a * 'a -> order) -> 'a list -> 'a list
  end
```

Why is it important to me, as a user of this sorting library, that it is implemented as a merge sort? The reason why I am using it is that I want a sorted list – but the concrete implementation details **do not matter** to me.[2]

[2] The astute reader might raise complaints about how it may be important to know the run-time complexity of the sorting function. Merge sort isn't the only $O(n \log n)$ sorting algorithm out there, though, and generally when using a library, you should be able to trust that the authors implemented it in a vaguely efficient way.

Abstracting Implementation

Here’s a better signature:

```
signature SORT =
  sig
    val sort : ('a * 'a -> order) -> 'a list -> 'a list
  end
```

We changed very little, but the idea is that we want to remove as many irrelevant details as possible from the users of the library! They want a sort, and they get a sort. The name is suggestive enough.

Alternate Implementation

Another reason for doing this, is that we might define multiple sorting libraries, and we want them to ascribe to a common signature! So now, we can also define a `InsertionSort` module:

```
structure InsertionSort : SORT =
  struct
    fun sort cmp L = (* ... *)
  end
```

4 - Information Hiding

Types in Structures

Recall that structures can also contain types. Consider a signature which describes a library for sets of integers.

```
signature INTSET =
  sig
    type t

    val empty : t
    val insert : int -> t -> t
    val remove : int -> t -> t
    val mem : int -> t -> bool
  end
```

We call the type `t` in this signature **abstract**, because it is left unspecified! Structures which implement this signature (or **ascribe** to it) can choose whatever representation they want.

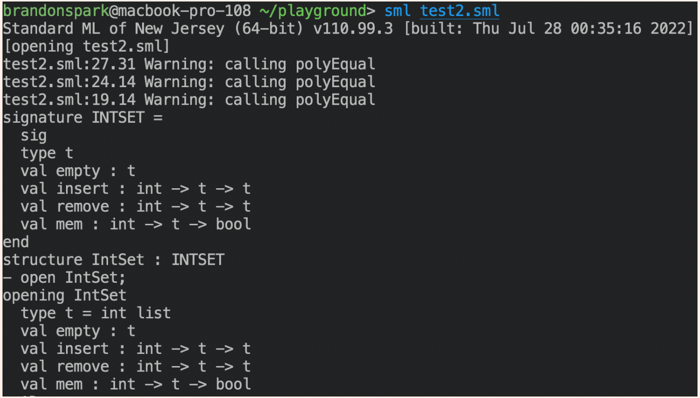
Types in Structures

Consider a structure which implements `INTSET` with lists.

```
structure IntSet : INTSET =
  struct
    type t = int list

    val empty = []
    fun insert v [] = [v]
      | insert v (x::xs) =
          if v = x then x::xs
          else (x :: insert v xs)
    fun remove v [] = []
      | remove v (x::xs) =
          if v = x then xs
          else (x :: remove v xs)
    fun mem v [] = false
      | mem v (x::xs) = v = x orelse mem v xs
  end
```

Opening Up



```
brandonspark@macbook-pro-108 ~/playground> sml test2.sml
Standard ML of New Jersey (64-bit) v110.99.3 [built: Thu Jul 28 00:35:16 2022]
[opening test2.sml]
test2.sml:27.31 Warning: calling polyEqual
test2.sml:24.14 Warning: calling polyEqual
test2.sml:19.14 Warning: calling polyEqual
signature INTSET =
  sig
  type t
  val empty : t
  val insert : int -> t -> t
  val remove : int -> t -> t
  val mem : int -> t -> bool
end
structure IntSet : INTSET
- open IntSet;
opening IntSet
  type t = int list
  val empty : t
  val insert : int -> t -> t
  val remove : int -> t -> t
  val mem : int -> t -> bool
```

Note The `open` keyword allows you to open all the things in a module into the enclosing scope.

The important thing is the line `type t = int list`.

Transparent Ascription

The type of ascription we showed you earlier is called **transparent ascription**.

The key thing that it does is that, even though the type of `t` in the signature `INTSET` is unspecified, transparent ascription makes it so that the type of `IntSet.t` is publicly known to be `int list`.

This is **really bad**.

I’m Gonna Wreck It

Because `IntSet.t` is the same as `int list`, it is OK to write the following:

```
val set : IntSet.t = [1, 1, 1, 1]

val set_without_1_i_promise = IntSet.remove 1 set

val _ =
  if IntSet.mem 1 set_without_1_i_promise then
    destroy_universe ()
  else
    dont ()
```

This code destroys the universe.[3]

[3] Sometimes that happens.

Invariant Violation

What happened here? We violated an invariant.

The `IntSet` library was carefully constructed so that by using `empty`, `insert`, and `remove`, every set would act like a set – in particular, `mem x (remove x S)` $\cong$ `false`. Every set should have precisely at most one entry for each integer.

When outside users of the library know *how it's implemented*, they can violate this invariant! This means if there was code somewhere which relied on receiving an `IntSet.t`, we could mess them up.

So how do we prevent this?

Opaque Ascription

The converse to transparent ascription is called **opaque ascription**.

Def **Opaque ascription** is transparent ascription, but any abstract types in the signature are unknown to users of the structure.

```
structure IntSetOpaque :> INTSET =
  struct
    type t = int list

    val empty = []
    (* ... *)
    fun mem v [] = false
      | mem v (x::xs) = v = x orelse mem v xs
  end
```

Here’s how we write it.

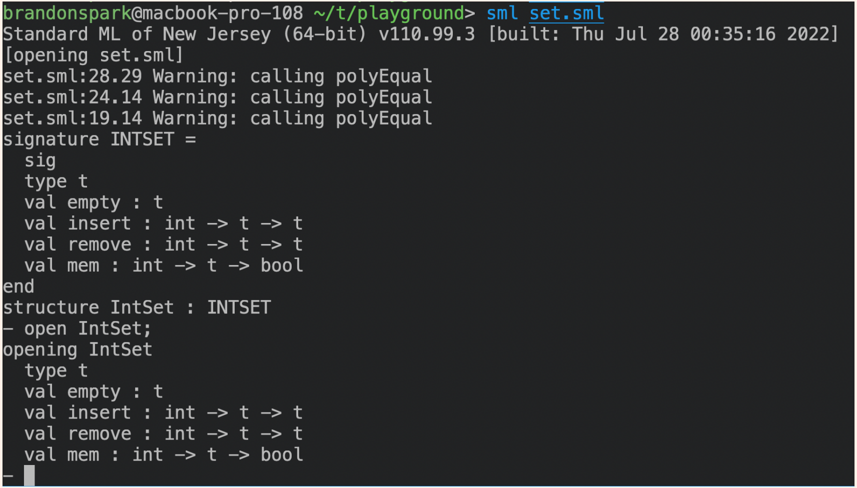
Invariants, Invincible

In the resulting structure `IntSetOpaque`, users of the module have no idea that the type `IntSetOpaque.t` is `int list`, and the compiler will enforce that. The compiler will fail to recognize that `IntSetOpaque.t` is the same as `int list`. We have **hidden** the fact that the `IntSetOpaque` library is implemented with lists.

This means that now it is *impossible* to obtain a value of type `IntSetOpaque.t` without going through `IntSetOpaque.empty`, `IntSetOpaque.remove`, or `IntSetOpaque.insert`. This means that now it is **provably impossible** to ever break our set invariant.

That's pretty neat.

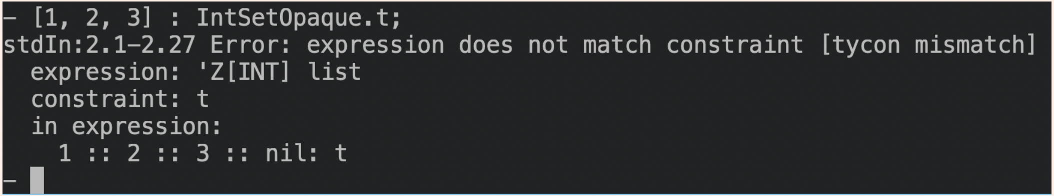
Using Opaque Structures



```
brandonspark@macbook-pro-108 ~/t/playground> sml set.sml
Standard ML of New Jersey (64-bit) v110.99.3 [built: Thu Jul 28 00:35:16 2022]
[opening set.sml]
set.sml:28.29 Warning: calling polyEqual
set.sml:24.14 Warning: calling polyEqual
set.sml:19.14 Warning: calling polyEqual
signature INTSET =
  sig
  type t
  val empty : t
  val insert : int -> t -> t
  val remove : int -> t -> t
  val mem : int -> t -> bool
end
structure IntSet : INTSET
- open IntSet;
opening IntSet
  type t
  val empty : t
  val insert : int -> t -> t
  val remove : int -> t -> t
  val mem : int -> t -> bool
- 
```

Using Opaque Structures

Now, we cannot see what the type `t` is at all, and it will not type-check if we try to declare a list to be of type `IntSetOpaque.t`.



```
- [1, 2, 3] : IntSetOpaque.t;
stdIn:2.1-2.27 Error: expression does not match constraint [tycon mismatch]
  expression: 'Z[INT] list
  constraint: t
  in expression:
    1 :: 2 :: 3 :: nil: t
- 
```

On Users and Maintainers

Again, this might seem strange. We just implemented `IntSetOpaque` as an `int list` like five minutes ago, what do you mean we don't know it's an `int list`?

The idea is that **signatures are for the user**, and **structures are for the maintainer**. The user of a library should only have to know things which are in the interface, and the implementation details are left to the maintainer, in the structure.

Being a programmer is a tenuous dance, because you're both. You implement libraries that you end up using, meaning you are both consumer and producer. Why is this important?

On Users and Maintainers

There are several advantages to being able to close your eyes and think like a user, rather than a maintainer:

- **it lightens your conceptual load**. Instead of thinking about the implementation of `insert`, `remove`, and co, you can think intuitively about what a set is.
- **it prevents you from breaking your own invariants**. If you set your invariants ahead of time, and design your API so you can never break it through the interface, then being unable to access the representation from outside the library prevents you from mangling your invariants later.
- **it helps you maintain your code**. If you later decide to refactor your code to use a different representation, such as a `int tree`, you don't need to touch any of the code outside of the module, so long as you adhere to the original interface.

SML's ability to *enforce* that you don't break this abstraction layer is one of the most powerful benefits that it provides.

5 - Representation Independence

Thinking Conceptually

So far in this course, we've trained our brains to be like mini-SML interpreters, such that we generally understand the stepping that SML programs will do, when we feed them into SML/NJ.

Something that has been mentioned several times in the past, however, is that this is ultimately inefficient! We can't think about stepping arbitrarily complex expressions in our head, because it will become too much mental load.

A more powerful tool we've relied on is using *invariants*, and in particular `ENSURES` postconditions, to reason about why recursive functions should be correct.

With representation independence, we can do even better. We can rely on *pictures* to reason about our code.

Thinking Conceptually

For instance, take the following code:

```
IntSet.insert 3 (IntSet.insert 2 (IntSet.insert 1 IntSet.empty))
```

Which is easier to think about – this:

mulligan trace

or this?

$$\{\} \xrightarrow{\text{insert } 1} \{1\} \xrightarrow{\text{insert } 2} \{1, 2\} \xrightarrow{\text{insert } 3} \{1, 2, 3\}$$

Thinking Conceptually

Due to opacity disallowing visibility into what the code is *really* doing, it ends up not mattering how the structure is actually implemented, so long as it *behaves* the way that it should.

This is the ultimate form of the duck test – if it walks like a duck, swims like a duck, and quacks like a duck, then it's probably a duck. In this case, if it behaves like a set, you can think of it like a set.

Now, instead of thinking of the code stepping in great detail, we just think of the picture in our head, of a set.

Thinking Conceptually

This will lead to a related concept called **representation independence**.

Def **Representation independence** is the phenomena where use of a library is independent of how it is represented. Ideally, any library involving an abstract type should be representationally independent.

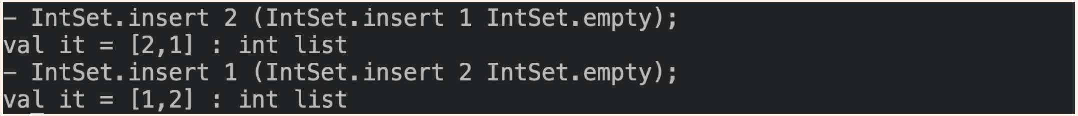
For instance, let's think of the difference between the following two code fragments:

```
IntSet.insert 2 (IntSet.insert 1 IntSet.empty)
```

```
IntSet.insert 1 (IntSet.insert 2 IntSet.empty)
```

Sets, Transparently

In terms of the concrete behavior of the function, we can see the actual value used to represent both, via using `IntSet` transparently:



```
- IntSet.insert 2 (IntSet.insert 1 IntSet.empty);
val it = [2,1] : int list
- IntSet.insert 1 (IntSet.insert 2 IntSet.empty);
val it = [1,2] : int list
```

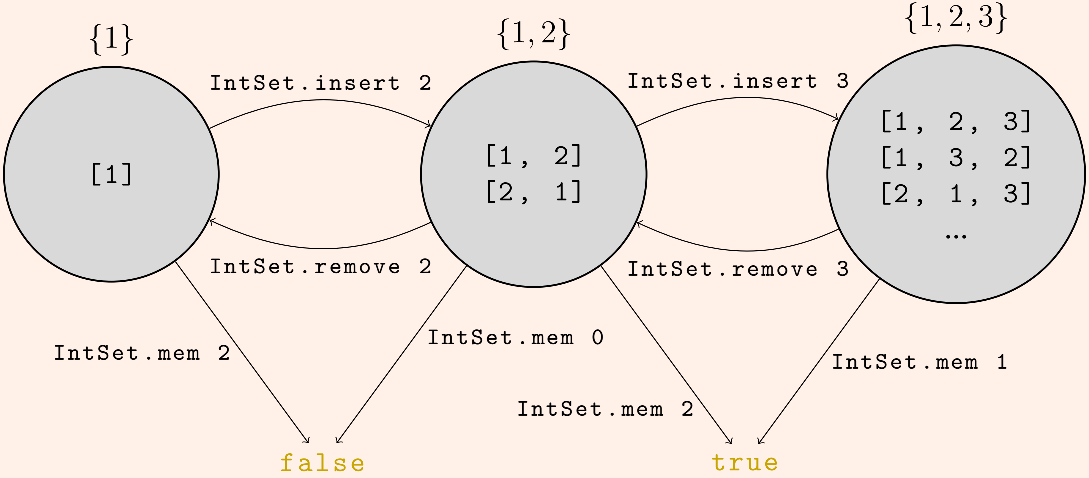
We see that the actual value we obtain in either case, when we expose the inner representation, is `[1, 2]` and `[2, 1]`.

Obviously, at this point in the course we recognize that those are different values. What gives? Isn't that an issue, since they are supposed to represent the same set?

The answer is *no*, and it's precisely because of this representationally independent thinking. Instead of thinking of values, we can group the values of the implementation into **representationally equivalent** classes, such that no two values in the same equivalence class can be distinguished.

Representational Equivalence Classes

The picture might look like this:



Representational Equivalence Classes

The previous image had many[4] nodes and edges missing from it, but the basic idea is that the circles denote the representational equivalence classes for values which *cannot be distinguished*, from outside the structure. They all denote the same basic mathematical set, which labels the circle at the top.

This means that you can transform values within the classes into other classes via edges that correspond to functions like `IntSet.insert` and `IntSet.remove`, and no matter what the actual precise value is, it will still respect the equivalence classes and their arrows.

In other words, you can define a **relation** on values, showing that they must always produce equivalently related values, from the same operations.

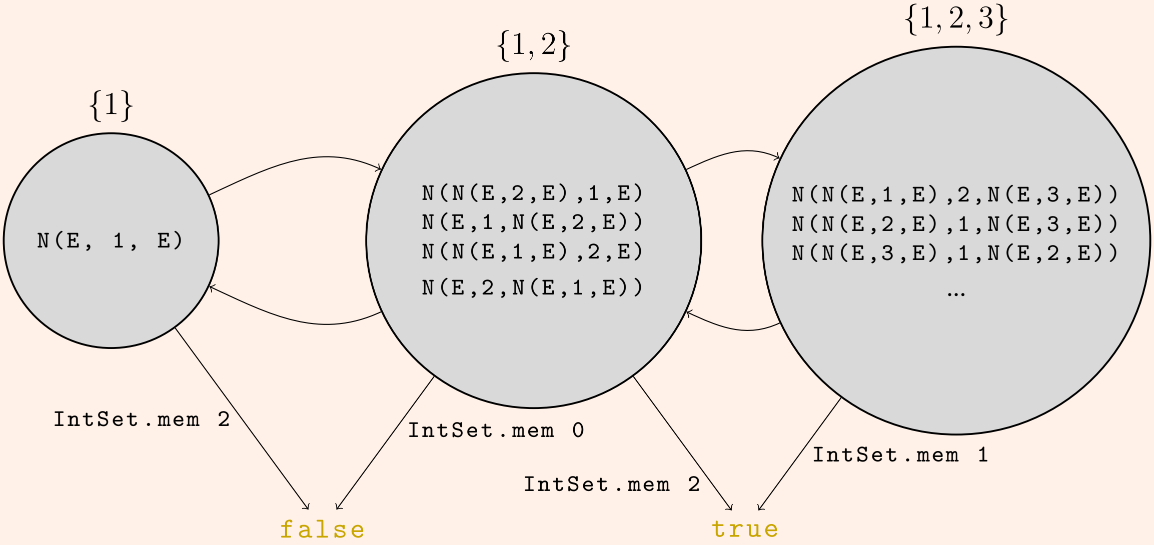
[4] In fact, infinitely many.

Representation Independence for Trees

This is a generalizable idea, however! This kind of diagram isn't specific to lists, it might apply to any structure which chooses to implement `INTSET`, and does it in a faithful way. For instance, suppose we had implemented `IntSet` using trees.

So now we might be able to draw a different picture, of what the same diagram looks like if we had implemented `IntSet` with trees. Note that for brevity, `Node` has been shortened to `N`, and `Empty` to `E`:

Representational Equivalence Classes (Trees)



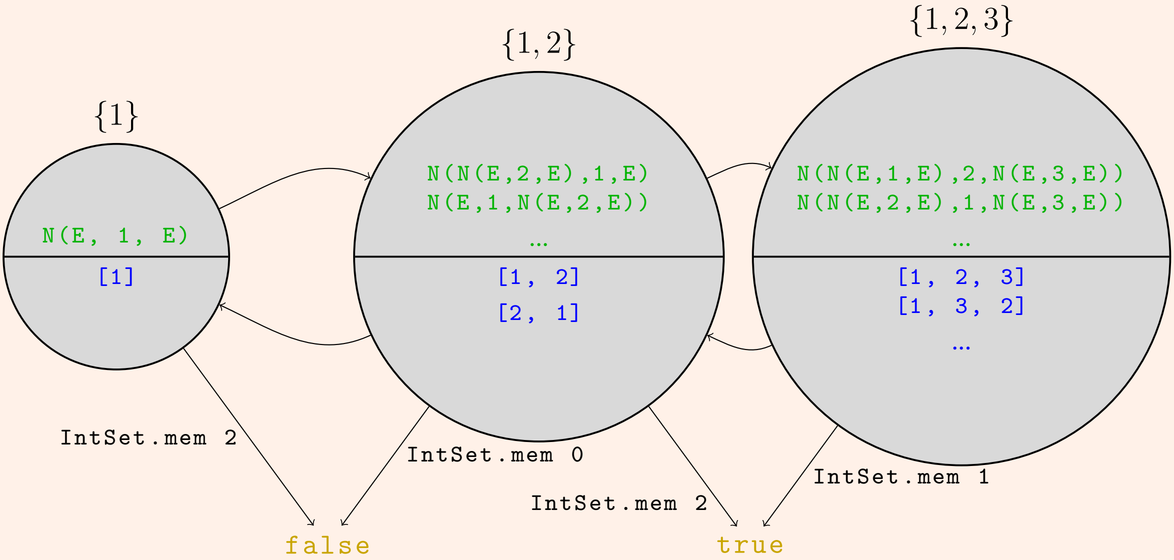
Representation Independence Between Types

In a sense, the fact that these relations can be defined for different implementations of the same signature, even with completely different representations, demonstrates how thinking conceptually about the mathematical object being represented is valid.

Once you are able to think in a representationally independent way, it’s clear that it shouldn’t matter at all that the values even came from the same structure! Even if it’s represented as a tree, rather than a list, the equivalence classes and behavioral equivalences still apply.

The most important part of this is that then, we can use this idea of representationally equivalent classes to *prove* that two implementations, even with differing representations, are equivalent. Visually, we smush the two diagrams together:

Representation Equivalence Classes Between Types



The ultimate point is that we can write a relation $R(\texttt{s1}, \texttt{s2})$ such that `s1 : IntSetList.t` and `s2 : IntSetTree.t`, and prove that the relation is preserved by all of the operations in the signature, starting from `IntSetList.empty` and `IntSetTree.empty`.

If we do this, then we prove that no matter what you do, it is impossible from outside of the structure to distinguish the two implementations. There is no sequence of operations that can ever give non-agreeing answers.

The basis of the proof conceptually consists of relating values from the same equivalence class, so for instance we could say that $R(\texttt{[1, 2]}, \texttt{Node(Node(Empty, 1, Empty), 2, Empty)})$ holds. This entails that `IntSetList.mem` and `IntSetTree.mem` must agree, for any int, on them, and that the removing or adding elements to either preserves the relation.

Conclusion

This may seem like a lecture with few practical applications, but these ideas are extremely profound in terms of how you should think about software development.

The ML module system is one of the most sophisticated out of any programming language that there is, and lots of ideas in it are applicable elsewhere. Being able to cleanly separate your implementation from your interface is essential when doing any kind of programming.

Abstract types are essential for architecting a safe interface that cannot be violated, even by yourself. Although it’s tough to see the benefits without working on the components of a software project yourself, the ability to conceptually and programmatically separate yourself from the implementation is amazingly helpful.

Thank you!