Discrete Mathematics

Supervision 7 Optional Exercise – Solutions

Marcelo Fiore Ohad Kammar Dima Szamozvancev

Set isomorphisms establish a type of equivalence between two sets: we can always translate between elements of isomorphic sets with the bijection exhibiting the isomorphism, so "for all intents and purposes" two isomorphic sets are interchangeable. Some isomorphisms come from deep, nontrivial results (such as Cantor's proof that $\mathbb{N} \cong \mathbb{Q}$, discussed later), others come from properties of set operations (such as $A \times B \cong B \times A$, the commutativity of Cartesian product). This question will ask you to define the set isomorphisms from the Calculus of Bijections I, but instead of boring old pen-and-paper maths, we will use fun hip trendy OCaml.

As it happens, sets and algebraic data types have quite a lot in common: elements of a set (e.g. $5 \in \mathbb{Z}$) can be treated as value of some type (e.g. `5 : int`). Certain set operations also have a direct analogue in types: the Cartesian product of sets corresponds to the product type `*`, while the disjoint union is the sum type (the `|` of `type` declarations). Algebraically, this gives us a *semiring* of types, just like how $(\mathcal{P}(A), \uplus, \times, \emptyset, [1])$ is a semiring in the universe of sets.[1] Functions between sets naturally correspond to functions between types; while in the discussion of this course functions were just special kinds of relations, they take a leading role in functional programming and other constructs are defined in terms of them.

Consider the following OCaml definitions.

Sum type

```
type ('a, 'b) sum = L of 'a | R of 'b
```

The type `('a, 'b) sum` is the sum type, a concrete type constructor representing the `|` of type declarations. Sum types correspond to the disjoint union of sets: a value of type `('a, 'b) sum` must either be a left-tagged value `L x` of type `'a`, or a right-tagged value `R y` of type `'b`.

Empty type

```
type empty = |
```

The `empty` type corresponds to the empty set. The syntax is a bit unusual, but it is OCaml's way of defining a type without any constructors. Since there are no constructors, there are no values of type `empty`. Empty types also come with an associated pattern-matching principle called a *refutation pattern*: if we have a variable of type `empty` (which is entirely possible, e.g. for the argument of a function of type `empty -> 'a`), we can use `match` to pattern-match on it. But since it is of empty type, there are no associated constructors and nothing to pattern-match on. This is expressed in OCaml by the pattern `_ -> .` (the wildcard pattern with a dot for the body), which is statically checked to verify that the pattern really is not possible. For instance, we define a function of type `empty -> 'a` as

[1] More details on the algebraic treatment of data types are given in the solution notes for FoCS SV2 Exercise 5.

```
fun (x : empty) -> match x with _ -> .
```

Set-theoretically this corresponds to the (unique) empty function from the empty set to any other set, simply defined as $\{\} : \emptyset \to A$. If we tried using the refutation pattern with a non-empty type, OCaml would complain, saying that the case we marked impossible with the dot is not unreachable after all: `fun (x : unit) -> match x with _ -> .` will give a counterexample of such a case, specifically the pattern `()`.

Powerset type

```
type 'a pows = Pows of ('a -> bool)
```

A very natural way to represent a subset S of elements of a set A is via its *characteristic function* $\chi_S : A \to [2]$: this is simply a predicate (a Boolean-valued function) on the elements of A answering the question "am I a member of the subset"? Thus, the set of all subsets of a set, called its powerset, is isomorphic to the set of predicates on the set: $\mathcal{P}(A) \cong (A \Rightarrow [2])$. Since OCaml has no notion of a "subtype", we make use of this isomorphism to represent the powerset of a type `'a` as the type of functions from `'a` to `bool`. To "hide" the implementation from view, we make this into a new data type, rather than a type synonym.

Partial function type

```
type ('a, 'b) pfun = PFun of ('a -> 'b option)
```

In OCaml we can represent partial functions by functions that raise an exception for an unsuitable input, or by returning an option type that gives a `None` for an unsuitable input. For the sake of purity, we use the second option to implement the set of partial functions $\mathrm{PFun(A, B)}$.

Relation type

```
type ('a, 'b) rel = Rel of ('a * 'b) pows
```

The set/type of relations between two types is just the powerset of their Cartesian product.

Isomorphism type

```
type ('a, 'b) iso = Iso of ('a -> 'b) * ('b -> 'a)
```

The type `('a, 'b) iso` represents an isomorphism between the types `'a` and `'b`. To create such an isomorphism, we need to provide two functions converting between `'a` and `'b`. Formally this does not define a bijection, as we don't supply any proofs for the inverse properties (which we cannot express in OCaml) – however, as we will be dealing with polymorphic types, we often expect the correct functions and inverses to be the only definitions that typecheck.

Your task is to give definitions for the following OCaml values corresponding to some of the isomorphisms in the Calculus of Bijections I. Some notes:

- Remember that OCaml's `function` syntax is actually a combination of `fun` and `match`, a pattern-matching anonymous function, so it can be written inline as one part of the isomorphism (but it has to be in parentheses otherwise OCaml gets confused).
- The isomorphisms must have the fully polymorphic type as given in the question. If OCaml infers, for example, `val prod_comm : ('a * 'a, 'a * 'a) iso` for the commutativity of the product, you made a mistake (e.g. returning `(b,b)` in one of the functions).
- You are encouraged to use various higher-order functions introduced in the OCaml course. One useful definition is that of function composition, written in infix form as `g << f`:

```
let (<<) g f x = g (f x)
> val (<<) : ('b -> 'c) -> ('a -> 'b) -> 'a -> 'c = <fun>
```

- You will often be in the middle of a definition (or later, an equational proof) and want to know what is the type of the expression or goal that you need to provide. A simple trick to see what OCaml expects is simply to trigger a type error! If you leave in a value of some type that should not appear in the expression (such as an integer), OCaml will gladly complain that it found a value of type `int` but expected an expression of type `<type>`. For instance, if you typecheck

```
let prod_comm : ('a * 'b, 'b * 'a) iso = Iso
   ( (fun (a, b) -> 1) , 2) )
```

  OCaml will say that the type expected in the place of `1` should be `'b * 'a`.

Properties of set operations

1. Cartesian product and disjoint union are commutative (example)

```
let prod_comm : ('a * 'b, 'b * 'a) iso = Iso
   ( (fun (a, b) -> (b, a))
   , (fun (b, a) -> (a, b)) )
let sum_comm : (('a, 'b) sum, ('b, 'a) sum) iso = Iso
   ( (function L a -> R a | R b -> L b)
   , (function L b -> R b | R a -> L a) )
```

2. Cartesian product and disjoint union are associative

```
let prod_assoc : ( ('a * 'b) * 'c
                 , 'a * ('b * 'c) )          iso = Iso ...
let sum_assoc  : ( (('a, 'b) sum, 'c) sum
                 , ('a, ('b, 'c) sum) sum ) iso = Iso ...
```

Straightforward reassociation of the nested tuples, and nested case analysis.

```
let prod_assoc : ( ('a * 'b) * 'c
                 , 'a * ('b * 'c) ) iso = Iso
  ( (fun ((a, b), c) -> (a, (b, c)))
  , (fun (a, (b, c)) -> ((a, b), c)) )

let sum_assoc : ( (('a, 'b) sum, 'c) sum
                , ('a, ('b, 'c) sum) sum ) iso = Iso
  ( (function L (L a) -> L a | L (R b) -> R (L b)
            | R c -> R (R c))
  , (function L a -> L (L a) | R (L b) -> L (R b)
                             | R (R c) -> R c) )
```

3. The units of products and disjoint unions are singleton sets and the empty set, respectively

```
let prod_unit : ('a * unit, 'a)       iso = Iso ...
let sum_unit  : (('a, empty) sum, 'a) iso = Iso ...
```

Note the use of the refutation pattern in the `sum_unit` case: if the input is a right injection, it would need to be of type `empty`, but we know that is impossible so the case is vacuously satisfied. The typechecker will be happy giving the impossible case any type, including `'a`.

```
let prod_unit : ('a * unit, 'a) iso = Iso
  ( (fun (a, ()) -> a)
  , (fun a -> (a, ())) )

let sum_unit : (('a, empty) sum, 'a) iso = Iso
  ( (function L a -> a | R _ -> .)
  , (fun  a -> L a) )
```

4. Products distribute over sums

```
let prod_sum_distr : ( (('a, 'b) sum) * 'c
                     , ('a * 'c, 'b * 'c) sum ) iso = Iso ...
```

```
let prod_sum_distr : ( (('a, 'b) sum) * 'c
                     , ('a * 'c, 'b * 'c) sum ) iso = Iso
  ( (function (L a, c) -> L (a, c) | (R b, c) -> R (b, c))
  , (function L (a, c) -> (L a, c) | R (b, c) -> (R b, c)) )
```

5. Functions and products

```
let fun_to_prod   : ( 'a -> ('b * 'c)
                    , ('a -> 'b) * ('a -> 'c) ) iso = Iso ...
let fun_from_prod : ( ('a * 'b) -> 'c
                    , 'a -> ('b -> 'c) )        iso = Iso ...
```

> Of course, `fun_from_prod` corresponds to currying and uncurrying!
>
> ```
> let fun_to_prod : ( 'a -> ('b * 'c)
>                   , ('a -> 'b) * ('a -> 'c) ) iso = Iso
>   ( (fun f -> (fun a -> fst (f a)) , (fun b -> snd (f b)))
>   , (fun (f, g) -> fun a -> (f a , g a)) )
>
> let fun_from_prod : ( ('a * 'b) -> 'c
>                     , 'a -> ('b -> 'c) ) iso = Iso
>   ( (fun f -> fun a -> fun b -> f (a, b))
>   , (fun f -> fun (a, b) -> f a b) )
> ```

6. Functions and sums

```
let fun_from_sum : ( ('a, 'b) sum -> 'c
                   , ('a -> 'c) * ('b -> 'c) ) iso = Iso ...
```

What about `fun_to_sum : ('a -> ('b, 'c) sum, ('a -> 'b, 'a -> 'c) sum) iso`?

> We can define one direction of `fun_to_sum`, but not the other: the arguments of type `'a` are independent in the two components of the pair, so we would need to "commit" to a side of the pair before we can call the function and pattern-match on its result. This is not really an OCaml quirk, it does not work in set theory either!
>
> On the other hand, `fun_from_sub` works as expected. Compare this transformation to the universal property of disjunction, unions, least common divisors, etc.: to show that the sum is "below" a type, one needs to show that *both* terms of the sum are "below" the type. If the RHS of the isomorphism was `('a -> 'c, 'b -> 'c) sum` (which is what you may have expected to be the dual of `fun_to_prod`), you will not be able to handle either case of the input since only one of the "handlers" is available at one time.
>
> ```
> let fun_to_sum   : ( 'a -> ('b, 'c) sum
>                    , ('a -> 'b, 'a -> 'c) sum ) iso = Iso
>   ( (fun f -> ?????)
>   , (function L f -> (fun a -> L (f a))
>             | R g -> (fun b -> R (g b))) )
> ```

```
let fun_from_sum : ( ('a, 'b) sum -> 'c
                   , ('a -> 'c) * ('b -> 'c) ) iso = Iso
  ( (fun f -> ((fun a -> f (L a)), (fun b -> f (R b))))
  , (fun (f, g) -> (function L a -> f a | R b -> g b)) )
```

7. Functions and neutral elements

```
let fun_from_unit  : (unit -> 'a, 'a)    iso = Iso ...
let fun_to_unit    : ('a -> unit, unit)  iso = Iso ...
let fun_from_empty : (empty -> 'a, unit) iso = Iso ...
```

What would be the difficulty in defining `fun_to_empty : ('a -> empty, empty) iso` for an arbitrary type `'a`? How about `('a option -> empty, empty) iso`?

```
let fun_from_unit : (unit -> 'a, 'a) iso = Iso
  ( (function f -> f ())
  , (function a -> fun () -> a) )

let fun_to_unit : ('a -> unit, unit) iso = Iso
  ( (function f -> ())
  , (function () -> fun _ -> ()) )

let fun_from_empty : (empty -> 'a, unit) iso = Iso
  ( (function f -> ())
  , (function () -> function _ -> .) )

let fun_to_empty : ('a -> empty, empty) iso = Iso
  ( (function f -> ?????)
  , (function e -> fun _ -> e) )
```

The problem with defining a function of type `('a -> empty) -> empty` is that we need to produce a return value of type `empty` – since it's the empty type, there is no distinguished value we could use (like `()` for `unit`), but we can still construct a term of type `empty` using variables in the scope (like returning the constant function of `e : empty` in the inverse case). However, given `f : 'a -> empty`, the only way we could create a term of type `empty` would be by applying `f` to an argument of type `'a`. As we know, we cannot produce polymorphic values, so this will be impossible – we cannot call `f` and can't return anything from the function.

However, there is a caveat: the isomorphism $(A \Rightarrow [0]) \cong [0]$ only holds if A is nonempty. More precisely, it says that there are no functions from a nonempty set to the empty

> set. This side-condition is needed because there *is* a function from the empty set to the empty set: the identity $\mathrm{id}_\emptyset = \{\}$, which the same as the unique empty function. We cannot implement the OCaml function above because `'a` can always be instantiated to be `empty`, in which case the isomorphism `(empty -> empty, empty) iso` doesn't hold (for the same reasons as above), but `(empty -> empty, unit) iso` does, as an instance of `fun_from_empty`. OCaml doesn't have a way of enforcing that `'a` should *not* be the empty type, but we can always ensure that it's nonempty by adding a distinguished element, which is exactly what the `option` constructor does. Now we can define
>
> ```
> let fun_to_empty : ('a option -> empty, empty) iso = Iso
>   ( (function f -> f None)
>   , (function e -> fun _ -> e) )
> ```

8. Set-theoretic representations of built-in types

```
let bool_to_sum   : (bool, (unit, unit) sum)    iso = Iso ...
let option_to_sum : ('a option, ('a, unit) sum) iso = Iso ...
```

> The set of Boolean values only has two elements, and a simple way of generating it is as the disjoint union of two singleton sets. The left injection of the singleton value (whatever it is) corresponds to true, the right injection to false (or vice versa). Option types are similar, except we replace one of the singleton sets with an arbitrary one, keeping the other injection a distinguished value.
>
> ```
> let bool_to_sum : (bool, (unit, unit) sum) iso = Iso
>   ( (fun b -> if b then L () else R ())
>   , (function L () -> true | R () -> false) )
>
> let option_to_sum : ('a option, ('a, unit) sum) iso = Iso
>   ( (function Some a -> L a | None -> R ())
>   , (function L a -> Some a | R () -> None) )
> ```

9. Powersets and characteristic functions

```
let pows_to_chfun : ('a pows, 'a -> bool) iso = Iso ...
```

> Remember that we define the `pows` type using this isomorphism, so this definition is nothing but extracting the constructor argument.
>
> ```
> let pows_to_chfun : ('a pows, 'a -> bool) iso = Iso
>   ( (fun (Pows f) -> f)
> ```

```
  , (fun f -> Pows f) )
```

10. Partial functions and relations

```
let pfun_to_optfun : (('a, 'b) pfun, 'a -> 'b option) iso = Iso ...
let rel_to_pows_prod : (('a, 'b) rel, ('a * 'b) pows) iso = Iso ...
```

> Again, these are *trivial*. We are defining them merely to have a consistent equational calculation system of such isomorphisms – a calculus, if you will.

```
let pfun_to_optfun : (('a, 'b) pfun, 'a -> 'b option) iso = Iso
  ( (fun (PFun f) -> f)
  , (fun f -> PFun f) )

let rel_to_pows_prod : (('a, 'b) rel, ('a * 'b) pows) iso = Iso
  ( (fun (Rel f) -> f)
  , (fun f -> Rel f) )
```

Properties of isomorphisms

So far we have defined some fundamental isomorphisms between types and type operations, but these are of not much use to us yet (other than as an educational exercise!). However, we can extend our system and develop a full library for programming with conversions and type isomorphisms – or alternatively, solve set theory exercises in OCaml!

The "calculus" part of the *Calculus of Bijections* refers not to differential or integral calculus, but a formal system for calculation: a collection of terms and equalities between them that lets us establish relationships between expressions via a sequence of syntactic manipulations. You're very familiar with this meaning of the word "calculus" from algebra: establishing algebraic equalities such as $a^{2x} \times (a^x)^y = a^{x\cdot(2+y)}$ happens via repeated applications of standard algebraic identities on the whole, or one part of the expression. We can provide a detailed equational proof of this equality:

$$
\begin{aligned}
a^{2x} \cdot (a^x)^y &= a^{2x} \cdot a^{xy} && \text{(power of power law in } \left(a^2\right)^x \cdot [-]\text{)} \\
&= a^{2x} \cdot a^{yx} && \text{(commutativity of multiplication in } \left(a^2\right)^x \cdot a^{[-]}\text{)} \\
&= \left(a^2\right)^x \cdot (a^y)^x && \text{(power of power law in both } [-] \cdot [-]\text{)} \\
&= \left(a^2 \cdot a^y\right)^x && \text{(power of product law)} \\
&= \left(a^{2+y}\right)^x && \text{(sum in power law in } [-]^x\text{)} \\
&= a^{x\cdot(2+y)} && \text{(power of power law)}
\end{aligned}
$$

This is of course one possible proof, but not the shortest one. However, it demonstrates the level of rigour we can achieve (if we wish), for example, by being explicit about the location in the expression

where the algebraic law is applied. Above, we mark this with a $[-]$, which can be read as "the power of product law $a^2 \cdot a^y = a^{2+y}$ is substituted for the $[-]$ in the expression $[-]^x$ to get the derived identity $(a^2 \cdot a^y)^x = (a^{2+y})^x$". This is of course needlessly verbose and seemingly superfluous – however, our ability to perform this substitution relies on a simple, but important property of equality, namely that subexpressions of an expression can be replaced with equal subexpressions. If we think of expressions as trees (recall the `expr` data type in FoCS), this states that any subtree of an expression can be replaced with another, as long as they evaluate to the same thing (i.e. they are *semantically* equivalent), even if syntactically they are completely different. Stated like this, we can see that such a strong property (confusingly called *congruence*) may not come for free, especially if we work with a custom equivalence relation, rather than equality. Recall the proofs in §2.1.2 that modular congruence can be applied on either side of addition or multiplication, or under an exponent:

$$i \equiv j \pmod{m} \implies i^n \equiv j^n \pmod{m}$$

This is essentially an analogue of saying that $i = j$ implies $i^n = j^n$, except for the equivalence relation of modular congruence – the proof is simple, but not trivial.

The equivalence and congruence properties of isomorphisms let us combine base isomorphisms into increasingly complex equalities between sets – they are the basis for *equational reasoning* within a formal system. Whenever you're asked to show that two sets are isomorphic, you can define the bijection and its inverse from scratch, prove the inverse laws or injectivity and surjectivity, etc. Alternatively, you can reason by transforming one set to the other using the calculus of bijections, and the isomorphism laws will be proved automatically, by construction. The latter approach of course depends on some prior work of developing the calculus, but once that is done, a surprising number of complex-looking isomorphisms can be established just by "algebraic" manipulation. This idea appears in many places in mathematics and computer science, and it's a good example of our philosophy of compositionality, abstraction, reuse of prior work – all principles you should be familiar with from programming.

In the rest of this exercise we raise the level of abstraction and deal with the isomorphism type itself, writing combinators corresponding to the equivalence and congruence properties of `('a, 'b) iso`. Then, you will be asked to write some equational "proofs" of more complex type isomorphisms, where, as much as possible, you should strive to construct the isomorphism by combining existing ones into more complex bijections, without defining the explicit conversion functions. Don't forget that we're not actually proving anything: all we're doing is defining the components of the bijection, but proving that the two functions are actually two-sided inverses of each other is not part of the OCaml code.

1. **Bijection** An isomorphism lets us convert between values of different, but isomorphic types.

```
let to_  : ('a, 'b) iso -> ('a -> 'b) = function ...
let fro_ : ('a, 'b) iso -> ('b -> 'a) = function ...
```

> We simply project out the relevant components of the pair of functions. This essentially means that we will never have to pattern-match on isomorphisms – you can even imagine

> hiding the implementation behind an abstract data type.
>
> ```
> let to_  : ('a, 'b) iso -> ('a -> 'b) = fun (Iso (t, _)) -> t
> let fro_ : ('a, 'b) iso -> ('b -> 'a) = fun (Iso (_, f)) -> f
> ```

2. **Equivalence relation** Any type is isomorphic to itself (reflexivity), isomorphisms can be flipped (symmetry), and they can be chained together (transitivity).

```
let iso_refl                      : ('a, 'a) iso = Iso ...
let iso_sym   (i  : ('a, 'b) iso) : ('b, 'a) iso = Iso ...
let iso_trans (i1 : ('a, 'b) iso)
              (i2 : ('b, 'c) iso) : ('a, 'c) iso = Iso ...
```

Once defined, we can get convenient shorthands for these combinators, as they will be frequently used in the isomorphism constructions later.

```
let same  = iso_refl
let inv   = iso_sym
let (>>>) = iso_trans
```

> We make use of the `to_` and `fro_` operators right away, decomposing and recombining the inputs to the desired form. Note the use of the composition operator `<<` to chain together the component OCaml functions.
>
> ```
> let iso_refl : ('a, 'a) iso = Iso
>   (id, id)
>
> let iso_sym (i : ('a, 'b) iso) : ('b, 'a) iso = Iso
>   (fro_ i, to_ i)
>
> let iso_trans (i1 : ('a, 'b) iso)
>               (i2 : ('b, 'c) iso) : ('a, 'c) iso = Iso
>   (to_ i2 << to_ i1, fro_ i1 << fro_ i2)
> ```

3. **Congruent relation** Isomorphisms can be applied on operands of type constructors.

```
let prod_cong (i1 : ('a, 'x) iso)
              (i2 : ('b, 'y) iso)
                  : ('a * 'b, 'x * 'y) iso = Iso ...
let sum_cong  (i1 : ('a, 'x) iso)
              (i2 : ('b, 'y) iso)
                  : (('a,'b) sum, ('x, 'y) sum) iso = Iso ...
```

```
let fun_cong  (i1 : ('a, 'x) iso)
              (i2 : ('b, 'y) iso)
                  : ('a -> 'b, 'x -> 'y) iso = Iso ...
```

Again, we can define shorthands for the congruence combinators, including ones specialised to altering only one of the two operands. While these can be written in pointfree style, we eta-expand everything to avoid OCaml's annoying weak polymorphic type inference.

```
let (=*=) = prod_cong and (=+=) = sum_cong and (=->=) = fun_cong
let in_fst  i = prod_cong i same and in_snd   i = prod_cong same i
let in_left i = sum_cong  i same and in_right i = sum_cong  same i
let in_dom  i = fun_cong  i same and in_cod   i = fun_cong  same i
```

> You've seen many similar properties before, such as with products, disjoint unions, and the subset relation in §5.2.4(a) and §5.2.5(a). The relation in this case is different (namely, an isomorphism), but the properties are similar: unwrap/case analyse, and apply the respective isomorphisms to the components before wrapping them back. The story is slightly more interesting with functions, as the isomorphism has to be applied "in reverse" to the input: that is because given a function `'a -> 'b`, the output can be transformed with a `'b -> 'y` (by post-composition), but the input requires the opposite direction `'x -> 'a` (by pre-composition). An isomorphism of course contains both directions, but we have to be explicit about which one is applied first.
>
> ```
> let prod_cong (i1 : ('a, 'x) iso)
>               (i2 : ('b, 'y) iso)
>                   : ('a * 'b, 'x * 'y) iso = Iso
>   ( (fun (a,b) -> (to_ i1 a, to_ i2 b))
>   , (fun (x,y) -> (fro_ i1 x, fro_ i2 y)) )
>
> let sum_cong (i1 : ('a, 'x) iso)
>              (i2 : ('b, 'y) iso)
>                  : (('a,'b) sum, ('x, 'y) sum) iso = Iso
>   ( (function L a -> L (to_ i1 a) | R b -> R (to_ i2 b))
>   , (function L x -> L (fro_ i1 x) | R y -> R (fro_ i2 y)) )
>
> let fun_cong (i1 : ('a, 'x) iso)
>              (i2 : ('b, 'y) iso)
>                  : ('a -> 'b, 'x -> 'y) iso = Iso
>   ( (fun f -> to_ i2 << f << fro_ i1)
>   , (fun g -> fro_ i2 << g << to_ i1) )
> ```

Compound isomorphisms

The point of all this setup is that we can define isomorphisms between complex types with the help of a few base isomorphisms and combinators. As an example, let's look at the proof of the isomorphism $[2] \Rightarrow A \cong A \times A$, that is, a function from the Booleans to a set A is isomorphic to a pair of values from A. Intuitively, we do an if-expression branching on the argument and the expressions of the `then` and `else` branches can be combined in a pair. It wouldn't be too bad writing this out explicitly:

```
let fun_from_bool_to_pair : (bool -> 'a, 'a * 'a) iso = Iso
  ( (fun f -> (f true, f false))
  , (fun (t,e) -> fun b -> if b then t else e) )
```

However, let's see if we can avoid the "low-level" details of the definition, and define the isomorphism abstractly, using the calculus of bijections.

$$
\begin{aligned}
([2] \Rightarrow A) &\cong ([1] \uplus [1]) \Rightarrow A && \text{(Booleans are the sum of two units)} \\
&\cong ([1] \Rightarrow A) \times ([1] \Rightarrow A) && \text{(function from a sum is product of functions)} \\
&\cong A \times A && \text{(function from unit isomorphic to an element)}
\end{aligned}
$$

We can represent this proof almost exactly in our OCaml library: all of the proof steps correspond to existing base combinators defined previously. There is some extra baggage coming from the need to be explicit about symmetry, transitivity, congruence, etc. (which are clear from the context in the written proof, but required in the code), but our combinators make this relatively unobtrusive.

```
let fun_from_bool_to_pair : (bool -> 'a, 'a * 'a) iso =
  in_dom bool_to_sum >>> fun_from_sum
                     >>> (fun_from_unit =*= fun_from_unit)
```

You might well argue that this is not a lot better than the explicit definition, but the trade-off becomes greater once you have more complicated expressions. In addition, if we prove that all of our base definitions are indeed isomorphisms, the inverse laws for compound isomorphisms all come for free – we would need to redo them every time for explicit definitions.

A nice thing about implementing all this in a programming language is that we immediately get the appropriate computational behaviour: for example,

```
to_ fun_from_bool_to_pair (fun b -> if b then 3 else 5)
```

evaluates to the pair `(3,5)`, as expected. And of course the value `fun_from_bool_to_pair` is a perfectly appropriate "proof" of `(bool -> 'a, 'a * 'a) iso` and can be used in other compound isomorphisms where such a bijection is needed.

1. Some simple helper lemmas: commuted unit laws, swapping of function arguments, transitivity which inverts its left or right argument (to avoid writing `inv` in chains of isomorphisms).

```
let prod_unit_l : (unit * 'a, 'a)       iso = ...
let sum_unit_l  : ((empty, 'a) sum, 'a) iso = ...
let (>><) (i1 : ('a, 'b) iso)(i2 : ('c, 'b) iso) : ('a, 'c) iso = ...
let (<>>) (i1 : ('b, 'a) iso)(i2 : ('b, 'c) iso) : ('a, 'c) iso = ...
let swap_iso : 'a -> 'b -> 'c, 'b -> 'a -> 'c) iso = ...
```

> Do we need all these? No, but they're easy enough to define and make proofs less verbose – so why not?
>
> ```
> let prod_unit_l : (unit * 'a, 'a) iso =
>       prod_comm >>> prod_unit
> let sum_unit_l : ((empty, 'a) sum, 'a) iso =
>       sum_comm >>> sum_unit
>
> let (>><) (i1 : ('a, 'b) iso)(i2 : ('c, 'b) iso) : ('a, 'c) iso =
>   i1 >>> inv i2
> let (<>>) (i1 : ('b, 'a) iso)(i2 : ('b, 'c) iso) : ('a, 'c) iso =
>   inv i1 >>> i2
>
> let swap_iso : ('a -> 'b -> 'c, 'b -> 'a -> 'c) iso =
>   fun_from_prod <>> in_dom prod_comm >>> fun_from_prod
> ```

2. Lifting isomorphisms to isomorphisms between powersets, and dualising relations:

```
let pows_cong  (i : ('a, 'b) iso) : ('a pows, 'b pows) iso = ...
let rel_op : (('a, 'b) rel, ('b, 'a) rel) iso = ...
```

> Powersets are isomorphic to characteristic functions, and domains of functions can be transformed. This, then, can be used to swap the order of sets in a relation.
>
> ```
> let pows_cong  (i : ('a, 'b) iso) : ('a pows, 'b pows) iso =
>   pows_to_chfun >>> in_dom i >>< pows_to_chfun
> let rel_op : (('a, 'b) rel, ('b, 'a) rel) iso =
>   rel_to_pows_prod >>> pows_cong (prod_comm) >>< rel_to_pows_prod
> ```

3. Exam question 2015 Paper 2 Question 9(b):

$$\mathcal{P}(A \uplus B) \cong \mathcal{P}(A) \times \mathcal{P}(B)$$

```
let pows_sum_to_prod_pows
      : (('a, 'b) sum pows, 'a pows * 'b pows) iso = ...
```

> The crucial step is transforming the characteristic function from the disjoint union to a pair of characteristic functions – the rest is interpreting powersets as characteristic functions.
>
> ```
> let pows_sum_to_prod_pows
>       : (('a, 'b) sum pows, 'a pows * 'b pows) iso =
>   pows_to_chfun >>> fun_from_sum
>                 >>< (pows_to_chfun =*= pows_to_chfun)
> ```

4. Relations can be transformed to functions with the powerset codomain:

$$\mathrm{Rel}(A, B) \cong A \Rightarrow \mathcal{P}(B)$$

```
let rel_to_pows_fun : (('a, 'b) rel, 'a -> 'b pows) iso = ...
```

> An elegant chain of isomorphisms that transforms $\mathcal{P}(A \times B)$ to $A \times B \Rightarrow [2]$, then curries this to a map from A to the characteristic function of B. Intuitively, this gives a functional way of interpreting relations: given one half of the relation in A, we get the *set* of elements that it is related to in B. In a total relation each such set has at least one element, while in a partial function all of them have at most one element; thus, for a function, the subset in B corresponding to an element in A has exactly one element – the value of the function. We can also turn relations $A \nrightarrow B$ to functions $B \to \mathcal{P}(A)$ by swapping the inputs.
>
> ```
> let rel_to_pows_fun : (('a, 'b) rel, 'a -> 'b pows) iso =
>   rel_to_pows_prod >>> pows_to_chfun
>                    >>> fun_from_prod
>                    >>> in_cod (inv pows_to_chfun)
> ```

5. Subsets are isomorphic to unit-valued partial functions

$$\mathcal{P}(A) \cong \mathrm{PFun}(A, [1])$$

```
let pows_to_pfun : ('a pows, ('a, unit) pfun) iso = ...
```

> Who said these necessarily have to be useful or enlightening? Note the chain of isomorphisms performed inside the `in_cod` congruence – this nesting avoids having to apply `in_cod` twice in a linear chain.
>
> ```
> let pows_to_pfun : ('a pows, ('a, unit) pfun) iso =
>   pows_to_chfun >>> in_cod (bool_to_sum >>< option_to_sum)
>                 >>< pfun_to_optfun
> ```

6. A bit of arithmetic: show the set-theoretic analogue of

$$2 \times 2 = 2 + 2 = 2^2$$

Why do functions represent exponentiation?

```
let times_to_plus : (bool * bool, (bool, bool) sum) iso = ...
let plus_to_exp   : ((bool, bool) sum, bool -> bool) iso = ...
```

> An intuitive argument is by looking at how the sizes of products, sums and functions relate to the sizes of the individual sets. We have $\#(A \times B) = \#A \cdot \#B$ and $\#(A \uplus B) = \#A + \#B$, which corresponds to the idea of "multiplication" and "addition" of sets (which, as you proved above, satisfy all the expected properties). Similarly, $\#(A \Rightarrow B) = \#B^{\#A}$, since for every input in A we have $\#B$ possible elements to map it to. This interpretation is also supported by the laws proved above, and it's worth seeing how they relate to standard algebraic identities you learnt in school: for example, `fun_from_sum` is the sum-exponent law $a^{b+c} = a^b \cdot a^c$, and `fun_from_prod` (a.k.a. currying) is $a^{b \cdot c} = \left(a^b\right)^c$!
>
> The proofs themselves are slightly clunky, but generally straightforward: for `times_to_plus` we reiterpret Booleans as a disjoint union, apply the distributivity law, and multiplicative unit laws. The translation from the product to the function space has already been shown as an instance of `fun_from_bool_to_pair` above! Then we chain these two together to turn the sum into a function. This is perhaps not how you would do the proof with numbers, but it works nevertheless!
>
> ```
> let times_to_plus : (bool * bool, (bool, bool) sum) iso =
>   (bool_to_sum =*= bool_to_sum) >>> prod_sum_distr >>>
>       ((prod_comm >>> prod_unit >>< bool_to_sum)
>     =+= (prod_comm >>> prod_unit >>< bool_to_sum))
>
> let plus_to_exp : ((bool, bool) sum, bool -> bool) iso =
>   inv (fun_from_bool_to_pair >>> times_to_plus)
> ```

7. Prove the set-theoretic version of the algebraic identity above:

$$a^{2x} \times (a^x)^y = a^{x \cdot (2+y)}$$

```
let example : ( ((bool * 'x) -> 'a) * ('y -> ('x -> 'a))
              , ('x * (bool, 'y) sum) -> 'a) iso = ...
```

> Translating from the example proof above – other proofs are available!
>
> ```
> let example : ( ((bool * 'x) -> 'a) * ('y -> ('x -> 'a))
> ```

```
                , ('x * (bool, 'y) sum) -> 'a) iso =
  fun_from_prod =*= swap_iso >>< fun_to_prod
                             >>> in_cod (inv fun_from_sum)
                             >>< fun_from_prod
```

8. Come up with some examples of your own!