Harmony Programmer's Manual

J. Nathan Foster, Benjamin C. Pierce, and Alan Schmitt
with
Aaron Bohannon, Michael B. Greenwald, Owen Gunden,
Malo Denielou, Jonathan T. Moore, Christian Kirkegaard,
Stéphane Lescuyer, and Zhe Yang

January 12, 2008

Mailing List

Active users of Harmony are encouraged to subscribe to the `harmony-hackers` mailing list by visiting the following URL:

```
http://lists.seas.upenn.edu/mailman/listinfo/harmony-hackers
```

Caveats

The Harmony system is a work in progress. We are distributing it in hopes that others may find it useful or interesting, but it has some significant shortcomings that we know about—as well as, surely, some that we don't—plus a multitude of minor ones. In particular, the documentation and user interface are... minimal. Also, the Focal implementation has not been carefully optimized. It's fast enough to run medium-sized (hundreds of lines) programs on small to medium-sized (kilobytes to tens of kilobytes) trees, but it's not up to industrial use.

Copying

Contributing

Contributions to Harmony—especially in the form of interesting or useful lenses—are very welcome. By sending us your code for inclusion in Harmony, you are signalling your agreement with the copying policy described above.

Contents

Chapter 1

Introduction

Optimistic replication strategies are attractive in a growing range of settings where weak consistency guarantees can be accepted in return for higher availability and the ability to update data while disconnected. These uncoordinated updates must later be *synchronized* (or *reconciled*) by automatically combining non-conflicting updates while detecting and reporting conflicting updates.

The Harmony project aims to develop a generic framework that can be used to build high-quality synchronizers for a wide variety of application data formats with minimal effort. The current Harmony system is the realization of our progress toward this goal, focusing on the important special cases of unordered and rigidly ordered data (including sets, relations, tuples, records, feature trees, etc.), with only limited support for list-structured data such as structured documents.

The Harmony system has two main components: (1) a domain-specific programming language, called Focal, for writing *lenses*—bi-directional transformations on trees—which we use to convert low-level (and possibly heterogeneous) concrete data formats into a high-level *synchronization schema*, and (2) a generic synchronization algorithm, whose behavior is controlled by the synchronization schema.

This document describes both components in detail.

Chapter 2

Quick Start

This chapter is intended for people who are installing Harmony on their own machines. If you are only using the web demo, you'll want to skip on to later sections.

Installation

1. Grab and unpack the most recent tarball from here:

   `http://www.cis.upenn.edu/~bcpierce/harmony/download`

2. Follow the instructions in the file `src/INSTALL` to install the OCaml compiler (if necessary) and the libraries on which Harmony depends.

3. Type `make test` in the top-level directory to build the Harmony executable and run all the regression tests.

Using Harmony with Unison

If you just want to run Harmony instances from Unison (rather than experimenting with writing your own), you should now skip to page 38.

Playing with the Demos

The `examples` directory contains a number of different Harmony instances. Most of them are also available for live experimentation via the web. (Go to the main Harmony page and follow the "demo" link.) Play with some of these to familiarize yourself with the basic concepts and capabilities of Harmony.

You'll find the Focal source code for demo `XXX` in a file called `examples/XXX/demos.php`, if you want to see how things work.

First Steps in Lens Programming

Now let's roll up our sleeves and write a few lenses.

1. Make a copy of the directory `examples/template` in a place where you can play with it:

```
cp -r examples/template experimental/play
cd experimental/play
```

2. Type `make test`. You should see this:

   ```
   Test result: {a = {1}, b = {2}}
   ```

3. Open the file `main.fcl` in your favorite text editor. Edit the tree in the last line to add a new child named `c` with a subtree `{3}` (i.e., add `c={3}` just before the closing curly brace) and re-run `make test`.

   The three-line Focal program in `main.fcl` can be read as follows:

   - The first line declares that this file contains a module named `Main`.
   - The second line defines a new lens named `l`, which behaves the same as the predefined identity lens, `id`.
   - The third line asks the Focal system to evaluate the *get* direction of `l` on the concrete argument (an expression of type `tree`) following the `/`. The `?` on the right of the = asks that the result be printed out.

4. Replace the `?` by the same tree as on the left of the = and re-run `make test`. This time, nothing at all should be printed. You've just written your first unit test.

   Edit the tree on the left-hand side of the = and run `make test`. You should now see an error message indicating that the test is failing.

   Change the right-hand side of the = back to `?`.

5. Change the `id` in the second line to `filter {a,c} {}`. Re-run `make test`. You should see this:

   ```
   Test result: {a = {1}, c = {3}}
   ```

   Note that the `b` child has been filtered away.

   Edit the tree in the test to add a `d` child, re-run the test, and note that this child is also filtered away.

6. Now let's see what this lens does in the *putback* direction. In the third line, change the `/` to `\` and add, just after the `\`, the tree `{a={5}}`. Run `make test`. You should see a new concrete tree with an updated value for `a`, with `c` missing, and with the subtrees under `b` and `d` carried over from the second argument tree.

7. Now let's play with some XML. Change the right-hand side of the definition of `l` back to `id` and replace the third line with this:

   ```
   let myxml : name = "<a><b>hello world</b></a>"

   let myxmltree : tree = (load "xml" myxml)

   test l / myxmltree = ?
   ```

   Run `make test`. What's printed is the low-level internal representation of XML in Focal—the way an XML string is parsed as a tree.

8. Change the definition of `l` to `Xml.squash`. Run `make test`. That looks a little better.

9. Change the definition of `l` again to

   ```
   let l : lens = Xml.squash; hoist "a"; hoist "b"; hoist Xml.PCDATA
   ```

   and note that the result of running the test changes to just the string `hello world` (encoded as the unique edge in a tree with an empty subtree).

10. Change the final `test` to

    ```
    test l \ {"goodbye cruel world"} myxmltree = ?
    ```

    and note how the new string has been re-inserted into the original XML structure. (The quote marks are needed here because we are defining a tree with an edge label containing the space character.)

11. Now let's play with transforming an external file. Change the definition of `l` to just `Xml.squash` and type:

    ```
    make get
    ```

    So far, we've just been running Harmony in its testing mode, which causes unit tests embedded in the code to be evaluated and their results printed. Doing `make get` runs Harmony in a different mode, asking it to run the lens `Main.l` over the contents of the file `test.xml` (which is parsed as XML because of its extension) and put the resulting tree (in Focal's tree notation) in the file `temp.meta`.

12. Change the definition of `l` to

    ```
    let l : lens = Xml.squash; focus "a" {}; focus "b" {}
    ```

    and observe the results of `make get`.

13. Now let's synchronize. Make two copies of the file `test.xml`:

    ```
    cp test.xml r1.xml
    cp test.xml r2.xml
    ```

    Now synchronize them by doing

    ```
    make sync
    ```

    Observe that a new file `archive.xml` has been created and that its contents are equal to those of `r1.xml` and `r2.xml`.

14. Change the string `hello` in the file `r1.xml` to `goodbye`. Change `world` to `cruel world` in `r2.xml`. Do `make sync` again and observe the results.

    Hooray.

This has been a lightning tour of some of the main features of the Harmony system. If you want to go further, the first thing to do is to read over the two main technical papers on which the system is based [**?**, **?**], both of which are available from the Harmony home page:

```
http://www.cis.upenn.edu/~bcpierce/harmony
```

After that, you should skim the rest of this document to get a sense of what is there. Then you should have a look at the `examples` directory and play with some of the larger demos that we've implemented.

Good luck and have fun!

Chapter 3

The Focal Language

The Focal language provides convienent concrete syntax for writing lenses programs along with names, trees, schemas, functions, and embedded unit tests. The concrete syntax is based on the fully-annotated, monomorphic core of OCaml.

3.1 Lexing

Whitespace characters are space, newline and tab. Comments are delimited by `(*` and `*)` and may be nested. Comments are equivalent to white space.

String literals can be any sequence of characters and escape sequences enclosed in double-quotes. The escape sequences `\"`, `\\`, `\b`, `\n`, `\r`, and `\t` stand for the characters double-quote, backslash, backspace, newline, vertical tab, and tab. To facilitate lining up columns in indented string literals, within a string, a newline followed by whitespace and then `|` is equivalent to a single newline. For example,

```
"University
 of
 Pennsylvania"
```

is equivalent to both

```
"University
of
Pennsylvania"
```

(in the leftmost column) and

```
"University\nof\nPennsylvania"
```

(anywhere).

3.1.1 Identifiers

Identifiers are non-empty strings drawn from the following set of characters:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9
' _ - @
```

and the following keywords and symbols are reserved and are not valid as identifers:

```
and      assert  at     begin  end      error  fun    in    lens    let
missing  module  name   open   protect  sync   test   view  schema  with
check    fmap    where  pred   fds      TRUE   FALSE
->       (       )      ;      .        *      !      |     =       {
}        [       ]      ,      :        \      /      ,     :       +
?        `       ~      -      &        <<~>   <<=>   <<->
```

3.2 Syntactic Structure

This section gives the formal definition of Focal syntax, using an EBNF grammar. The productions for each syntactic category are followed by a brief, informal discussion. In the rules we adopt the following conventions:

- Literals are written in a typewriter font and enclosed in single quotes, like this: '`module`';
- Non-terminals are enclosed in angle brackets, like this: ⟨*Exp*⟩.
- Optional elements are enclosed in square brackets: ['`:`' ⟨*Sort*⟩];
- Terms may be grouped using parentheses;
- Repeated elements are specified using * (0 or more) and + (1 or more).

3.2.1 Modules and Declarations

⟨*CompilationUnit*⟩ ::= '`module`' ⟨*Ident*⟩ '=' ('`open`' ⟨*Qid*⟩)* ⟨*Decl*⟩*

⟨*Decl*⟩ ::= '`let`' ⟨*BindingList*⟩
  | '`schema`' ⟨*SchemaBindingList*⟩
  | '`module`' ⟨*Ident*⟩ '=' ⟨*Decl*⟩* '`end`'
  | '`test`' ⟨*Exp*⟩ '=' ⟨*Exp*⟩
  | '`test`' ⟨*Exp*⟩ '=' '`error`'
  | '`test`' ⟨*Exp*⟩ '=' '?'
  | '`sync`' '`with`' ⟨*AExp*⟩ '`at`' ⟨*AExp*⟩ ⟨*AExp*⟩ '=' ⟨*AExp*⟩
  | '`sync`' '`with`' ⟨*AExp*⟩ ⟨*AExp*⟩ ⟨*AExp*⟩ '`at`' ⟨*AExp*⟩ ⟨*AExp*⟩ '=' ⟨*AExp*⟩

A Focal compilation unit contains a single module declaration, such as `module Foo`, that must appear in a file named `foo.src` (for "literate" sources) or `foo.fcl` (for plain sources). A module contains a sequence of `open` declarations, which import all the declarations from another module, followed by a sequence of declarations. A declaration is either a `let`, a `schema`, a nested `module`, or a unit test. Focal modules are currently only used to group declarations in a common namespace. In particular, Focal does not yet support module signatures or sealing.

Unit Tests

The syntax includes several forms of unit tests (which are only tested when the system is run in testing mode; see Section 6.1). The first form, `test` t_1 = t_2, checks that t_1 and t_2, which must both be expressions of sort `view`, evaluate to equal values. This kind of test is often used to check the behavior of the *get* and *putback* components of lenses, as in the following snippet: `test id / {} = {}`. The second form, `test` l_1 `/\` t_1 = t_3, is shorthand for two tests of a bijective lens: `test` l_1 `/` t_1 = t_2 and `test` l_1 `\` t_2 `missing` = t_1. The third form, `test` t_1 = `error`, checks that t_1 raises an exception when it evaluates t_1. This kind of test is used to ensure that native lenses, such `hoist`, check side conditions on their inputs correctly; for example, `test hoist "n" {} = error`. The fourth form, `test` t_1 =?, causes Harmony to run the unit test and print out the result. The final two forms are used to test the behavior of the synchronization algorithm:

`sync with` l `at` s `{O=`o`, A=`a`, B=`b`} = {O=`o' `, A=`a' `, B=`b'`}.`

applies $l\nearrow$ to the trees o, a, and b, synchronizes those trees at schema s, and then applies $l\searrow$ to the results, yielding o', a' and b'. The unit test

`sync with` l_o l_a l_b `at` s `{O=`o`, A=`a`, B=`b`} = {O=`o' `, A=`a' `, B=`b'`}.`

is similar, but applies a different lens to o, a, and b.

3.2.2 Bindings

⟨*Binding*⟩ ::= ⟨*Ident*⟩ (⟨*Ident*⟩'`:`'⟨*Sort*⟩)+ '`:`' ⟨*ASort*⟩ '=' ⟨*Exp*⟩
| ⟨*Ident*⟩ (⟨*Ident*⟩'`:`'⟨*Sort*⟩)+ '`:`' ⟨*AExp*⟩ ⟨*LensArrow*⟩ ⟨*AExp*⟩ '=' ⟨*Exp*⟩

⟨*BindingList*⟩ ::= ⟨*Binding*⟩
| ⟨*Binding*⟩ 'and' ⟨*BindingList*⟩

⟨*SchemaBinding*⟩ ::= ⟨*Ident*⟩ '=' ⟨*Exp*⟩

⟨*SchemaBindingList*⟩ ::= ⟨*SchemaBinding*⟩
| ⟨*SchemaBinding*⟩ 'and' ⟨*SchemaBindingList*⟩

Focal `let`-bindings are identical to OCaml `let`-bindings except that the `let`-bound variable and the expression must both be annotated with their sorts.

In a schema declaration, the bound variable may be used recursively, as in the following declaration: `schema X = { "n" = X } | {}`, which denotes the infinite set of trees containing the empty tree and every tree with a single child `n` whose subtree is also in the schema. For more on schemas, see Sections 3.2.4 and 3.3.

3.2.3 Sorts

⟨*Sort*⟩ ::= ⟨*ASort*⟩ '`->`' ⟨*Sort*⟩
| ⟨*ASort*⟩

⟨*ASort*⟩ ::= '`lens`'
| '`name`'
| '`view`'
| '`schema`'
| '`(`' ⟨*Sort*⟩ '`)`'

⟨*LensArrow*⟩ ::= '`<<~>`'
| '`<<=>`'
| '`<<->`'

Focal does not yet have a full-blown lens type system. However, the compiler performs rudimentary sort checking. The sorts are `lens`, `name`, `view`, `schema`, and function sorts, of the form s_1`->`s_2. The sort checker supports exactly one form of subtyping: it automatically promotes an expressions with sort view to the singleton schema containing that tree.

3.2.4 Expressions

⟨*Exp*⟩ ::= '`let`' ⟨*BindingList*⟩ '`in`' ⟨*Exp*⟩
| '`schema`' ⟨*SchemaBindingList*⟩ '`in`' ⟨*Exp*⟩
| '`fun`' (['(']⟨*Ident*⟩'`:`'⟨*Sort*⟩[')'])+ ['`:`'⟨*Sort*⟩] '`->`' ⟨*Exp*⟩
| ⟨*GetPutExp*⟩

⟨*ExpList*⟩ ::= ⟨*Exp*⟩
| ⟨*Exp*⟩ ‘,’ ⟨*ExpList*⟩

⟨*GetPutExp*⟩ ::= ⟨*ComposeExp*⟩ ‘/’ ⟨*PlusExp*⟩
| ⟨*ComposeExp*⟩ ‘\’ ⟨*AExp*⟩ ⟨*AExp*⟩
| ⟨*ComposeExp*⟩ ‘\’ ⟨*AExp*⟩ ‘`missing`’
| ⟨*ComposeExp*⟩

⟨*ComposeExp*⟩ ::= ⟨*ComposeExp*⟩ ‘;’ ⟨*BarExp*⟩
| ⟨*BarExp*⟩

⟨*BarExp*⟩ ::= ⟨*BarExp*⟩ ‘|’ ⟨*AndExp*⟩
| ⟨*AndExp*⟩

⟨*InterExp*⟩ ::= ⟨*InterExp*⟩ ‘&’ ⟨*MinusExp*⟩
| ⟨*MinusExp*⟩ ⟨*MinusExp*⟩ ::= ⟨*MinusExp*⟩ ‘+’ ⟨*PlusExp*⟩
| ⟨*PlusExp*⟩

⟨*PlusExp*⟩ ::= ⟨*PlusExp*⟩ ‘+’ ⟨*ConsExp*⟩
| ⟨*ConsExp*⟩

⟨*ConsExp*⟩ ::= ⟨*AppExp*⟩ ‘::’ ⟨*ConsExp*⟩
| ⟨*AppExp*⟩ ‘:|:’ ⟨*ConsExp*⟩
| ⟨*AppExp*⟩

⟨*AppExp*⟩ ::= ⟨*AppExp*⟩ ⟨*AExp*⟩
| ⟨*AExp*⟩

⟨*AExp*⟩ ::= ⟨*String*⟩
| ⟨*QId*⟩
| ‘~’ ⟨*AExp*⟩
| ‘{’ ⟨*MapList*⟩ ‘}’
| ‘{’ ⟨*TreeEltList*⟩ ‘}’
| ‘[’ ⟨*ExpList*⟩ ‘]’
| ‘[|’ ⟨*ExpList*⟩ ‘|]’
| ‘(’ ⟨*Exp*⟩ ‘)’
| ‘`begin`’ ⟨*Exp*⟩ ‘`end`’
| ‘`protect`’ ⟨*AExp*⟩
| ‘`assert`’ ⟨*AExp*⟩

⟨*Map*⟩ ::= ⟨*QuotedName*⟩ ‘->’ ⟨*Exp*⟩

⟨*MapList*⟩ ::= ⟨*Map*⟩
| ⟨*Map*⟩ ‘,’ ⟨*MapList*⟩

⟨*TreeElt*⟩ ::= ⟨*QuotedName*⟩
| [‘?’] ⟨*QuotedName*⟩ ‘=’ ⟨*Exp*⟩
| ‘*’ [‘\’ [‘(’] ⟨*QuotedNameList*⟩ [‘)’]] ‘=’ ⟨*Exp*⟩
| ‘!’ [‘\’ [‘(’] ⟨*QuotedNameList*⟩ [‘)’]] ‘=’ ⟨*Exp*⟩
| ‘?’ [‘\’ [‘(’] ⟨*QuotedNameList*⟩ [‘)’]] ‘=’ ⟨*Exp*⟩
| ‘+’ [‘\’ [‘(’] ⟨*QuotedNameList*⟩ [‘)’]] ‘=’ ⟨*Exp*⟩

⟨*TreeEltList*⟩ ::= ⟨*TreeElt*⟩
| ⟨*TreeElt*⟩ ',' ⟨*TreeEltList*⟩

Focal expressions are similar to OCaml expressions with only a few special forms for composing and applying lenses, and for writing down trees and schemas. The main forms, in decreasing order of precedence, are:

- Fully annotated `let`-expressions,
- Anonymous λ-expressions, written `fun` x_1`:`$s_1 \ldots x_k$`:`s_k `:`s`->` e, where each x_i is a plain identifier and s_i a sort. The parameters and sorts may optionally be enclosed in parentheses, and the return sort may be omitted.
- get and put expressions, which apply the $\nearrow$ and $\searrow$ components of a lens respectively. A get expression is written e_1 `/` e_2, where e_1 has sort `lens` and e_2 has sort `view`. Put expressions are written $e_1 \setminus e_2\ e_3$ or $e_1 \setminus e_2$ `missing`, if the concrete argument is missing.
- Lens composition, written e_1`;`e_2,
- Schema unions, written e_1 `|` e_2,
- Tree and schema concatenations, written e_1 `+` e_2,
- Tree and schema cons cells, written e_1`::`e_2,
- Application expressions, written $e_1\ e_2$,
- and "atomic" expressions
  - Name literals, written `"n"`,
  - Qualified identifiers, written `M.N.x`, and described below,
  - Finite maps from names to lenses, written `{`n_1`->L` e_1`, ..., `n_k`->L` e_k`}`, where each n_i is a quoted name (see below) and e_i is an expression with sort lens.
  - Trees and schemas, written `{` t_1`, ..., `t_k`}`, where each t_i is a "tree element"; i.e., either a:
    - named atom of the form n=e, where n is a quoted name (see below) and e an expression,
    - or in schemas, a wildcard of the form `?=`e, `!=`e, `+=`e, `*=$e$`, or `*=`e, where `?` matches trees with zero or one children, `!` trees with exactly one child, `+` trees with one or more children, and `*` trees with any number of children. Each of these wildcards may also optionally be annotated with "exception lists" that exclude finite sets of names from the schema. Exception lists are written `\(`n_1,...,n_k`)`; the parentheses enclosing the exception list may optionally be omitted. As an example, the schema `!\(n)=`t denotes the set of trees with exactly one child *not* named `n` that has a subtree in t.
  - Trees representing lists and list schemas, written `[`e_1`, ..., `e_n`]`,
  - Constant relational databases, written `{{ A_1(fields_1) = relation_1,...,A_m(fields_m) = rela` where each `fields_i` has the form `x_1 ... x_k`, each `relation_i` has the form `{tuple_1,...,tuple_` and `tuple_i` has the form `(y_1 ... y_n)`.
  - Arbitrary expressions enclosed in parentheses (or equivalently, in `begin` and `end`).
  - `protect` expressions, which cause the enclosed expression to be evaluated using lazy (call-by-need) evaluation strategy instead of an eager strategy.

– `assert` expressions, which behave like the identity lens but perform some run-time type checking. If the *get* function of `assert T` is invoked with a concrete argument not belonging to the schema `T` or its *putback* direction is invoked with either argument not belonging to `T`, then it prints a message to this effect and halts the interpreter.

  `assert` expressions take an argument with sort `schema` and yield a lens. The lens checks the trees, in both directions, for membership in the schema. If the membership test succeeds, then the entire expression behaves like the identity lens; otherwise the lens prints an error message and halts.

3.2.5 Quoted Names

$\langle$*QuotedName*$\rangle$ ::= $\langle$*Ident*$\rangle$
  | $\langle$*String*$\rangle$
  | '`' $\langle$*AExp*$\rangle$

$\langle$*QuotedNameList*$\rangle$ ::= $\langle$*QuotedName*$\rangle$
  | $\langle$*QuotedName*$\rangle$ ',' $\langle$*QuotedNameList*$\rangle$

Within a map, tree or schema expression, such as `{a={},b=[]}`, the tokens appearing to the left of the "->" and "=" symbols (in maps and in trees and schemas respectively) symbols are treated as names (string literals), not as variables. The anti-quotation token, " `", allows one to refer to the expression `a`, which will typically be a variable, rather than the name.

3.2.6 Qualified Identifiers

$\langle$*QId*$\rangle$ ::= $\langle$*Ident*$\rangle$
  | $\langle$*QId*$\rangle$ '.' $\langle$*Ident*$\rangle$

An identifier may be qualified with an optional module prefix. Thus, `M.N.x` refers to `x` from module `M.N`.

3.3 Schemas

When constructing schemas, the Focal compiler checks three well-formedness constraints:

- All schemas must be contractive. The compiler ensures that the schemas it produces are contractive using a syntactic check: all recursive uses of `schema`-bound variables must appear at least one level deeper in the schema. The compiler rejects non-contractive schemas such as `schema X = X + {}` and `schema Y = Y | { "n" = {} }` but allows `schema X = { n = X } | {}`.
- When concatenating two schemas, the sets of names that may appear as immediate children in the two schemas must be disjoint, except for the infinite sets generated by wildcards. For example, the schemas `{"n" = {}} + {"n" = []}` and `{"n" = {}} + {!=[]}` are both rejected because the name `n` appears on both sides of the concatenation. However, the schema `{! = {}} + {! = {}}` is allowed since the overlap is on every child.
- When forming a schema using union (or concatenation), for every name, the subschema below each name appearing on both sides of the union (or concatenation) must be equivalent. This restriction is analogous to the restriction on tree grammars embodied in W3C schema. The Focal compiler currently only supports a simple syntactic approximation of schema equivalence (e.g., it does not equate a recursive variable with its expansion or distribute unions). For example, the schemas `{"n" = {}} | {"n" = []}` and `{"n" = []} | {! = {}}` are both rejected since the subschemas below `n`, `{}` and `{}` are not equivalent.

Chapter 4

The Focal Libraries

The Harmony programming environment includes an assortment of primitive lenses and many useful derived lenses, along with associated schemas and predefined names. All these are described in this chapter, grouped according to module.

In most cases, the easiest way to understand what a lens does is to see it in action on examples; most lens descriptions therefore include several unit tests, using the notation explained in Section 3.2.1.

More thorough descriptions of most of the primitive lenses can be found in an earlier technical paper [**?**]. The technical report version of this paper includes full proofs that the definitions are "well behaved," but the shorter conference version should be sufficient for getting up to speed with Focal programming.

4.1 The Standard Prelude

The lenses and schemas described in this section are available by default and need not be qualified with a module name. For the sake of coherent grouping, some of the unit tests and derived forms mention lenses that are only defined later in the section.

4.1.1 Predefined Schemas

`Any` The schema `Any` denotes the set of all trees.

`KeyedTree` A "keyed" tree consists of a single name pointing to some subtree.

```
let KeyedTree (X:schema) : schema = {!=X}
```

`Value` The schema `Value` describes trees with just one child pointing to an empty tree:

```
schema Value = KeyedTree {}
```

`NonNullValue` The schema `NonNullValue` describes all values except `{""}`:

```
schema NonNullValue = {!\""={}}
```

4.1.2 Generic Lenses

`id` The `id` lens returns its concrete argument unchanged in the *get* direction and its abstract argument unchanged in the *putback* direction.

```
test id / \ {a={b}} = {a={b}}
```

$\boxed{\texttt{const}}$ The lens `const` t d always return the tree t in the *get* direction.

```
test const {a} {b} / {} = {a}
test const {}  {b} / {a} = {}
```

In the *putback* direction, it is only defined if the abstract tree has not changed (if it is exactly t).

```
test const {a} {b} \ {b} {} = error
```

In this case, it returns the original concrete tree if there is one, and the default tree d otherwise.

```
test const {a} {b} \ {a} {} = {}
test const {a} {b} \ {a} {c} = {c}
test const {a} {b} \ {a} missing = {b}
```

$\boxed{;}$ The lens composition operation is written as an infix semicolon. In the *get* direction, `l;k` simply applies the *get* component of `l` followed by the *get* component of `k`. In the other direction, the two *putback* functions are applied in turn: first, the *putback* function of `k` is used to put the abstract tree a into the concrete tree that the *get* of `k` was applied to, i.e., `l`$\nearrow c$; the result of this *putback* is then put into c using the *putback* function of `l`. (If the concrete tree c is `missing`, then, `l`$\nearrow c$ is also defined to be `missing`, so the effect of `l;k`$\searrow a$ `missing` is to use `k` to put a into `missing` and then `l` to put the result into `missing`.)

```
test (hoist "x"; hoist "y") / \ {x = {y = {z}}} = {z}
```

A similar test case using two instances of `const` illustrates the strong schema constraint imposed by `const` (the second argument of the second `const` *must* be `{a}`).

```
test (const {a} {d}; const {b} {a}) / {} = {b}
test (const {a} {d}; const {b} {a}) \ {b} {foo,bar} = {foo,bar}
test (const {a} {d}; const {b} {a}) \ {b} missing = {d}
test (const {a} {d}; const {b} {foo}) \ {b} missing = error
```

4.1.3 Forking Lenses

$\boxed{\texttt{xfork}}$ The lens combinator `xfork` applies different lenses to different parts of a tree: it splits the tree into two parts according to the names of its immediate children, applies a different lens to each, and concatenates the results. Formally, `xfork` takes as arguments two sets of names and two lenses. For example:

```
test (xfork {a} {x,y,z} (hoist "a") id)  / {a={x,y}, b={z}, c} = {x,y,b={z},c}
```

(Note that sets of names are represented syntactically as trees of values.) The first set of names, `{a}`, specifies how concrete trees should be split: in the *get* direction, the input tree (here `{a={x,y},b={z},c}`) is split into a tree with just the top-level edge labeled `a` (i.e., `{a={x,y}}`), which is passed through `hoist "a"`, and a tree with the edges `b` and `c` (i.e., `{b={z},c}`), which is passed through `id`. The resulting trees (`{x,y}` and `{b={z},c}`) are merged to form the result.

The second set of names, `{x,y,z}`, specifies that only the names `x`, `y`, and `z` may appear at the top level in the tree returned by the first *get*, and also that these three names may *not* appear at the top level in the tree returned by the second *get*; these conditions ensure that the final merge always makes sense.

```
test (xfork {a} {x} (hoist "a") id)  / {a={x,y}, b={z}, c} = error
test (xfork {a} {x,y,z,b,c} (hoist "a") id)  / {a={x,y}, b={z}, c} = error
```

The *putback* direction of `xfork` works similarly: the concrete argument is split according to the first set of names, the abstract argument according to the second set of names, the two lenses are applied to the corresponding pairs of constituent trees, and the results are merged. Again, the *putback* direction of the first lens must always yield trees whose top-level names fall in the first set, and the second lens must yield trees whose names fall in the second set.

```
test (xfork {a} {x,y,z} (hoist "a") id)
       \ {x, z={foo}, b, d}
         {a={x,y}, b, c}
     = {a={x,z={foo}}, b, d}
```

`fork` Often, the two sets of names passed to `xfork` are the same. The derived lens `fork` packages this case for convenience:

```
let fork (p:view) (l1:lens) (l2:lens) : lens = xfork p p l1 l2
```

`bfork` Another way to divide a tree is using a *schema*. The lens `bfork C pa` splits the concrete view into one tree whose children all lead to subtrees belonging to `C`, and another whose children all lead to subtrees belonging to `not C`. The *putback* function, divides the abstract tree using a set of names like `xfork`.

```
test (bfork {!={}} {"tmp"} (const {tmp} {})
     (fork {tmp} (const {} {}) id))
     / {tmp = {1, 2}, a = {1}, b = {2, 3}} = { tmp = {}, b = {2,3}}
```

4.1.4 Lenses for Structural Transformations on Trees

`hoist` The lens `hoist` n is used to shorten a tree by removing an edge at the top. In the *get* direction, it expects a tree that has exactly one child, named n. It returns this child, removing the edge n. In the *putback* direction, the value of the old concrete tree is ignored and a new one is created, with a single edge n pointing to the given abstract tree.

```
test (hoist "n") /\ {n} = {}
test (hoist "n") /\ {n={a}} = {a}
```

It is an error to apply `hoist` n to a concrete tree whose domain is not exactly n.

```
test (hoist "n") / {} = error
test (hoist "n") / {a} = error
test (hoist "n") / {n,a} = error
```

`hoist_nonunique` The derived lens `hoist_nonunique` n is a generalized variant of `hoist` that removes the restriction that its concrete argument have domain exactly n.

```
let hoist_nonunique (n:name) (p:view) : lens = xfork {‘n} p (hoist n) id
```

It takes an extra argument, a set of names represented as a tree, specifying the possible names of top-level edges in the subtree reached under n, which must be disjoint from the names of top-level edges other than n.

```
test (hoist_nonunique "n" {a,b,c}) /\ {n={a},x={b}} = {a, x={b}}
```

`plunge` The lens `plunge n`, defined as

```
let plunge (n:name) : lens = invert (hoist n)
```

is the exact inverse of `hoist n`.

```
test (plunge "n") /\ {a} = {n={a}}

let smash : (name -> lens) = Native.Prelude.smash
```

filter The `filter` lens allows all but a given set of children (specified as a tree) to be projected away in the *get* direction and restored in the *putback* direction.

```
let filter (p:view) (d:view) : lens = fork p id (const {} d)

test (filter {x,y,z} {new}) / {x,y,a,b,c} = {x,y}
test (filter {x,y,z} {new}) \ {x,z} {x,y,a,b,c} = {x,z,a,b,c}
```

The second argument to `filter` is a tree (with top-level names not in the given set) that is used as a default value for the projected away part of the tree in the case where the *putback* part of `filter` is applied to `missing`.

```
test (filter {x,y,z} {new}) \ {x,y} missing = {x,y,new}
```

focus The lens `focus` n d, defined as

```
let focus (n:name) (d:view) : lens = filter {'n} d ; hoist n
```

projects away all the top-level names except n (restoring them in the *putback* direction) and then hoists the children of n up to the top level.

```
test (focus "n" {new}) / {x={a},n={b={c}}} = {b={c}}
test (focus "n" {new}) \ {b={d},e} {x={a},n={b={c}}} = {x={a},n={b={d},e}}
```

As for `filter`, the second argument is used when the *putback* direction of `focus` is applied to `missing`.

```
test (focus "n" {new}) \ {b={d},e} missing = {n={b={d},e}, new}
```

prune The lens `prune` n removes just the name n in the *get* direction (if it is present) and restores it in the *putback* direction.

```
let prune (n:name) (d:view) : lens = fork {'n} (const {} {'n=d}) id

test (prune "n" {new}) / {} = {}
test (prune "n" {new}) / {n={a},m={b}} = {m={b}}
test (prune "n" {new}) \ {m={c},d} {n={a},m={b}} = {n={a}, m={c}, d}
test (prune "n" {new}) \ {m={c},d} missing = {n={new}, m={c}, d}
```

add The lens `add` n v adds a new subtree v named n in the *get* direction and removes it in the *putback* direction.

```
let add (n:name) (v:view) : lens = xfork {} {'n} (const {'n=v} {}) id

test (add "n" {x}) / \ {a={z}} = {a={z},n={x}}
```

rename The lens `rename` m n renames m to n in the *get* direction and n to m in the *putback* direction.

```
let rename (m:name) (n:name) : lens = xfork {'m} {'n} (hoist m; plunge n) id

test (rename "m" "n") /\ {m={a}, x={b}} = {n={a}, x={b}}
```

rename_if_present The lens `rename_if_present` m n is just like `rename` m n except that it does not demand that its concrete argument actually possess a subtree named m or that its abstract argument possess a subtree named n.

```
let rename_if_present (m:name) (n:name) : lens =
  acond {'m=Any, * \ 'm = Any } {'n=Any, * \ 'n = Any }
    (rename m n)
    id

test (rename_if_present "m" "n") /\ {m={a}, x={b}} = {n={a}, x={b}}
test (rename_if_present "m" "n") /\ {x={b}} = {x={b}}
test (rename "m" "n") / {x={b}} = error
```

`pivot` The lens `pivot` k performs a "rotation" on the root of the tree, taking a tree of the form `{k={x}}+w` to one of the form `{x=w}`—i.e., promoting the value under k to the top so that it becomes a key for the remaining children, `w`.

```
test (pivot "k") / \ {k={x},a,b={c}} = {x={a,b={c}}}
```

The most common use of `pivot` is in the idiom `List.map (pivot "k"); List.flatten`.

4.1.5 Mapping Lenses

`map` The lens `map l` applies `l` to each of the *children* of the argument tree. In the *get* direction, it applies $l\nearrow$ to each subtree of the root and combines the results together into a new tree.

```
test (map (const {x} {new})) / {a={b},c} = {a={x},c={x}}
```

In the *putback* direction, there are three cases to consider.

1. Names that appear only in the concrete tree (i.e., they have been deleted in the abstract tree) are deleted in the resulting concrete tree.

   ```
   test (map (const {x} {new})) \ {a={x}} {a={b},c} = {a={b}}
   ```

2. Names that appear only in the abstract tree (i.e., they have been created) are put back into `missing`.

   ```
   test (map (const {x} {new})) \ {a={x}} {} = {a={new}}
   ```

3. For names that appear in both the concrete and abstract arguments, the *putback* direction of `l` is applied to corresponding subtrees.

   ```
   test (map (const {x} {new})) \ {a={x}} {a={b},c} = {a={b}}
   test (map (plunge "n")) \ {a={n={x,foo}}} {a={x}} = {a={x,foo}}
   ```

`mapp` The lens `mapp p l` is just like `map l` except that it ignores subtrees whose names are not in the set `p` (which is specified as a tree).

```
let mapp (p:view) (l:lens) : lens = fork p (map l) id

test (mapp {a,b} (const {x} {new})) / {a={b},c} = {a={x},c}
```

`mapn` The lens `mapn n l` performs `l` just on the child named `n`.

```
let mapn (n:name) (l:lens) : lens = mapp {`n} l

test (mapn "a" (const {x} {new})) / {a={b},c} = {a={x},c}
test (mapn "a" (const {x} {new})) \ {a={x},c} missing = {a={new},c}
```

`wmap` The `wmap` combinator is a generalized variant of `map` that can apply a *different* lens to each subtree. Its argument is a finite map from names to lenses (see Section **??**).

```
test (wmap { a -> const {x} {new}, b -> plunge "n" })
       / {a,b={foo}}
       = {a={x}, b={n={foo}}}
```

4.1.6 Conditional Lenses

`ccond` The "concrete conditional" `ccond C lt lf` tests whether its concrete argument belongs to the schema `C`, acting like `lt` if so and `lf` if not, in both *get* and *putback* directions.

```
test (ccond {a=Any} (hoist "a") id) / {a={x}} = {x}
test (ccond {a=Any} (hoist "a") id) \ {y,z} {a={x}} = {a={y,z}}
test (ccond {a=Any} (hoist "a") id) / {b={x}} = {b={x}}
test (ccond {a=Any} (hoist "a") id) \ {y,z} {b={x}} = {y,z}
```

Note that, to ensure well-behavedness, the lenses `lt` and `lf` must have exactly the same range. That is, if `T` is the entire set of trees that the lens `ccond C lt lf` is ever going to be applied to, then the set of possible results of the *get* direction of `lt` when applied to trees in `C` should coincide with the set of possible results of the *get* direction of `lf` when applied to trees in `T` \ `C`. See [**?**] for more discussion of this point.

`acond` The "abstract conditional" `acond C A lt lf` tests its concrete argument against the schema `C` in the *get* direction and tests its abstract argument against the schema `A` in the *putback* direction, sending its arguments through `lt` or `lf` in each case.

```
let acond_test : lens = acond {a=Any,b=Any} {a=Any} (filter {a} {new}) id

test acond_test / {a={x},b={y}} = {a={x}}
test acond_test / {a={x},c={z}} = {a={x},c={z}}
test acond_test / {} = {}

test acond_test \ {a={foo}} {a={x},b={y}} = {a={foo},b={y}}
test acond_test \ {a={foo},c={z}} {a={x},b={y}} = {a={foo},c={z}}
test acond_test \ {a={foo}} {a={x}} = {a={foo},new}
```

(The final example shows an important detail: if the *putback* of `acond C A lt lf` is applied to a tree `a` that *does* belong to `A` together with a tree `c` that does *not* belong to `C`, then the *putback* of `lt` is used to put `a` into `missing`.)

Note that the lenses `lt` and `lf` should have disjoint ranges. If `T` is the entire set of trees that the lens `acond C A lt lf` is ever going to be applied to, then the set of possible results of the *get* direction of `lt` when applied to trees in `C` should be exactly `A`, and the set of possible results of the *get* direction of `lf` when applied to trees in `T` \ `C` should be disjoint from `A`.

`cond` The lens combinator `cond` is a "generalized conditional." It is seldom used by itself—most often, either `ccond` or `acond` suffices—but occasionally its full power is needed. Please see [**?**] for more details. (The standard prelude also provides four variants, called `cond_ww`, `cond_fw`, `cond_wf`, `cond_ff`, that treat the "fixup functions" in slightly different ways. For details, read the source.)

4.1.7 Merging and Copying Lenses

`merge` It sometimes happens that a concrete representation requires equality between two distinct subtrees within a view. A `merge` lens is one way to preserve this invariant when the abstract view is updated. In the *get* direction, `merge` takes a tree with two (equal) branches and deletes one of them. In the *putback* direction, `merge` copies the updated value of the remaining branch to *both* branches in the concrete view.

```
test (merge "m" "n") / {m={a}, n={a}} = {m={a}}
test (merge "m" "n") / {m={a}, n={b}} = {m={a}}
test (merge "m" "n") / {m={b}, n={b}} = {m={b}}
test (merge "m" "n") \ {m={a}} missing = {m={a}, n={a}}
test (merge "m" "n") \ {m={a}} {m={a}, n={b}} = {m={a}, n={b}}
test (merge "m" "n") \ {} missing = error
```

`copy` In the *get* direction, `copy` m n takes a tree, c, that has no child labeled n. If $c(m)$ exists, then `copy` m n duplicates $c(m)$ by setting both $a(m)$ and $a(n)$ equal to $c(m)$. In the *putback* direction, `copy` simply discards $a(n)$.

```
test (copy "m" "n") / {m={a}} = {m={a}, n={a}}
test (copy "m" "n") \ {m={a}, n={a}} missing = {m={a}}
test (copy "m" "n") \ {m={a}, n={a}} {m={a}} = {m={a}}
test (copy "m" "n") \ {m={a}, n={a}} {m={b}} = {m={a}}
test (copy "m" "n") / {} = error
test (copy "m" "n") \ {m={a}} missing = error
test (copy "m" "n") \ {m={a}, n={b}} missing = error
```

The *putback* of the `copy` lens is only guaranteed to be well behaved if the subtrees under m and n in the abstract view are always identical. This is a very strong constraint, which makes `copy` nearly uselesss in practical programming. See [**?**] for more discussion.

4.1.8 Lenses for "Keyed Relations"

`join` The `join` lens, based on an idea by Daniel Spoonhower [**?**], is related to the *full outer join* operator from databases.

```
test (join "x" "y") / {x, y} = {}
test (join "x" "y") / {x={1={a}, 2={b}}, y={2={c}}}
                    = {1={x={a}}, 2={x={b}, y={c}}}
test (join "x" "y") / {x={1,2}, y={2}}
                    = {1={x}, 2={x,y}}
test (join "x" "y") / {x={1}, y}
                    = {1={x}}

test (join "x" "y") \ {} missing = {x, y}
test (join "x" "y") \ {} {x={1}, y={2}} = {x, y}
test (join "x" "y") \ {a={x={1}, y={3}}, b={x={2}}, c={y={4}}}
                      missing
                    = {x={a={1}, b={2}},  y={a={3}, c={4}}}
```

Section 4.5 describes a more sophisticated set of relational lenses, relying on a heavier-weight

4.1.9 Debugging Lenses

`probe` The lens `probe msg` behaves like `id` except that, whenever it is invoked, it dumps its arguments (along with the identifying string `msg`) to the standard output.

`progress` The lens `progress msg` behaves like `id` except that, whenever it is invoked, it prints `msg` to the standard output. It is used by potentially long-running lenses to give an indication of what part of a large tree they are currently working on.

`tracepoint` The lens `tracepoint msg l` behaves just like `l` except that, when it is invoked, it places a marker on the call stack. If the execution of `l` results in a run-time exception (because of a bad argument to a lens, for example), these stack markers and their associated arguments are printed as part of the termination message.

4.1.10 I/O Functions

`read` The expression `read f` reads the file `f` from the disk and returns its whole contents as a single name.

4.1.11 Viewer Functions

`load` The expression `load ekey blob` parses the name `blob` using the viewer identified by the encoding key `ekey` (see Section 6.1) and yields a tree. Section 4.3 shows some examples of its use.

`save` The expression `save ekey v` converts the view `v` using the viewer identified by the encoding key `ekey` (see Section 6.1) and yields a view.

`load_file` The expression `load_file name` parses a file into a tree, using the encoding key specified as part of the file name or the default encoding key for this file schema if none is specified explicitly.

4.1.12 Miscellaneous

`fconst` The lens `fconst v cmd` behaves exactly like `const`, except that instead of using a default tree `d` in the case of creation, it executes `cmd` and uses the value returned by the command.

```
test (fconst {} "echo youpi") \ {} missing = { "youpi" }
```

`invert` The function `invert` maps a bijective lens `l` to a bijective lens whose *get* is the *putback* of `l` and vice versa.

4.2 Module `List`

4.2.1 The List Encoding

Lists are represented in Focal as trees of "cons cells." For example, the list `[1,2]` is represented by the tree `{HD={1}, TL={HD={2}, TL={NIL}}}`. This encoding is recognized and treated specially by Harmony's parsing and printing functions, so the concrete syntax `[1,2]` can be used freely in both trees and schemas.

4.2.2 Names and Schemas

`HD, TL, NIL` The actual string names of the tags `HD`, `TL`, and `NIL` in the encoding are an implementation detail. To avoid writing them in code, we define three variables of sort `name` that hold their actual values and use these variables instead of the literal tags in Focal programming.

`tags` It is also convenient to have a tree whose domain is exactly these three names (for situations where we want to fork on these names, for example):

```
let tags : view = {'HD, 'TL, 'NIL}
```

`Nil` The schema `Nil` denotes just the empty list.

```
let Nil : schema = {'NIL={}}
```

`Cons` The schema `Cons X Y` denotes the set of cons cells whose head is of type `X` and whose tail is of type `Y`. (Note that `Y` need not necessarily be `List`, though it typically will be.)

```
let Cons (X: schema) (Y: schema) : schema = {'HD=X, 'TL=Y}
```

`T` The schema `T X` (which is written `List.T X` in other modules) denotes all lists whose elements have type `X`.

```
let T (X:schema) : schema =
  schema F = Nil | Cons X F in F
```

`NonEmptyList` It is also convenient to have a schema denoting just non-empty lists:

```
let NonEmptyList (X:schema) : schema = Cons X (T X)
```

4.2.3 Lenses

`hd` In the *get* direction, `hd d` yields the head of its concrete argument (which must be a non-empty list). In the *putback* direction, it yields a list consisting of its abstract argument as the head and the tail of its concrete argument as the tail. When the concrete argument to *putback* is `missing`, it uses its default argument `d` as the tail of the result list.

```
let hd (d:view) : lens = focus HD { 'TL = d }

test (hd [{2}]) / [{1}] = {1}
test (hd [{2}]) / [] = error
test (hd [{2}]) \ {1} [{2},{3}] = [{1},{3}]
test (hd [{2}]) \ {1} missing = [{1},{2}]
```

`tl` Similarly, `tl d` yields the tail of its (non-empty) concrete argument in the *get* direction; in the *putback* direction, it recombines a new abstract tail with the original concrete head (or with `d` when the concrete argument to the *putback* is `missing`).

```
let tl (d:view) : lens = focus TL { 'HD = d }

test (tl {}) / [{1},{2},{3}] = [{2},{3}]
test (tl {}) \ [{2},{3}] missing = [{},{2},{3}]
test (tl {}) \ [{2},{3}] [{1}]  = [{1},{2},{3}]
test (tl {}) / [] = error
```

`map` The lens `map l` applies `l` to each element of its argument, which must be a list. In the *putback* direction, the resulting list will have the same length as the abstract argument; if the abstract argument is longer than the concrete one, then the elements at the end will be *putback* into `missing`.

```
let map (l:lens) : lens =
  wmap { 'HD -> l, 'TL -> (protect (map l)) }

test (map (const {b} {c})) / [] = []
test (map (const {b} {c})) / [{1},{2},{3}] = [{b},{b},{b}]
test (map (const {b} {c})) \ [{b}] [{1},{2}] = [{1}]
test (map (const {b} {c})) \ [{b},{b},{b}] [{1},{2}] = [{1},{2},{c}]
test (map (const {b} {c})) \ [{b}] missing = [{c}]
```

`fold_right` The lens `fold_right init l` is analogous to the familiar `fold_right` function on lists in any functional programming language. In the *get* direction, the lens `l` is recursively applied to the tail of the list, or the `init` tree is used if the list is empty. In the *putback* direction, `l`↘ is recursively applied to the abstract tree and the concrete list, until the abstract tree is exactly `init`. For that reason, note that `l` must not return `init`.

```
let fold_right (init:view) (l:lens) : lens =
  mapn TL (protect (fold_right init l));
  acond [] init
    (const init [])
    l
```

The following lens uses `fold_right` to transform a list of values into a *chain* of values.

```
let fold_right_test : lens = fold_right {} (pivot HD; Prelude.map (hoist TL))

test fold_right_test / [] = {}
test fold_right_test / [{1},{2},{3}] = {1={2={3}}}
test fold_right_test \ {2={4={6}}} [{1},{hello}] = [{2},{4},{6}]

test fold_right {bar} (const {foo} {new}) / [{1},{2}] = {foo}
test fold_right {bar} (const {foo} {new}) / [] = {bar}
test fold_right {bar} (const {foo} {new}) \ {foo} [{1},{2}] = [{1},{2}]
test fold_right {bar} (const {foo} {new}) \ {foo} missing = {new}
test fold_right {bar} (const {foo} {new}) \ {baz} {anyway} = error
```

`reverse` The bijective lens `reverse` simply reverses its argument list in both directions.

```
let reverse : lens =
  let old_HD : name = "old_hd" in
  let rotate : lens =
    acond  ([] | [Any]) ([] | [Any])
        id
        (rename HD old_HD;
         hoist_nonunique TL tags;
         xfork {`old_HD, `TL} {`TL}
           (rename old_HD HD;
            plunge TL;
            mapn TL (protect rotate))
           id)
  in
    mapn TL (protect reverse);
    rotate

test reverse / \ [] = []
test reverse / \ [{1},{2},{3}] = [{3},{2},{1}]
```

`concat2` The bijective lens `concat2 sep` transforms (in the *get* direction) a list of two lists into a single list consisting of the elements of the two given lists appended together and separated by `sep` (which must not occur in either of the given lists). In the *putback* direction, a single abstract list is split (at the single occurrence of `sep`) into a pair of concrete lists.

```
let concat2 (sep:name) : lens =
  acond ([]::Any) ({`sep={}}::Any)
    (mapn HD (const {`sep} []); mapn TL (hd []))
    (fork {`TL} id (hoist HD; rename TL "tmp");
     fork {`HD} id (rename "tmp" HD; protect (concat2 sep); plunge TL))
```

```
test (concat2 " ") / \ [[],[]]
                     = [{" "}]
test (concat2 " ") / \ [[{u}],[{p},{e},{n},{n}]]
                     = [{u},{" "},{p},{e},{n},{n}]
test (concat2 " ") / \ [[{u},{n},{i},{v}],[{p},{e},{n},{n}]]
                     = [{u},{n},{i},{v},{" "},{p},{e},{n},{n}]
```

`groupby2` The bijective lens `groupby2` takes a list (in the *get* direction) and yields a list of lists, all of whose elements are of length exactly two, except the last, which can have length either one or two. I.e., `groupby2` groups a list into pairs, possibly with a leftover singleton at the end. The *putback* direction simply concatenates such a list of lists.

```
let groupby2 : lens =
  acond [] []
    id
    (acond [Any] [[Any]]
       (plunge HD; add TL { `NIL={} })
       (rename HD "tmp";
        hoist_nonunique TL tags;
        fork { `TL }
          (Prelude.map (protect groupby2))
          (xfork { `HD } { `TL }
             (add TL { `NIL={} }; plunge TL)
             (rename "tmp" HD);
           plunge HD)))

test groupby2 / \  [] = []
test groupby2 / \ [{1}] = [[{1}]]
test groupby2 / \ [{1},{2},{3},{4}] = [[{1},{2}], [{3},{4}]]
test groupby2 / \ [{1},{2},{3},{4},{5}] = [[{1},{2}], [{3},{4}], [{5}]]
```

`filter` The lens `filter D E d` maps a concrete list consisting of elements belonging to `D|E` and yields (in the *get* direction) a list containing just those elements belonging to `D`. In the *putback* direction, a new abstract list of `D`s is "woven back" into the concrete list, retaining all the concrete `E`s (and maintaining their positions in the list) and replacing concrete `D`s by elements from the new abstract list. A detailed explanation of `filter`'s implementation may be found in [**?**].

```
let simple_filter (E:schema) : lens =
  mapn TL (protect (simple_filter E));
  ccond  (E::Any)
    (tl {"error"})
    id

let filter (D:schema) (E:schema) (some_D:view) : lens =
  let append (v:view) : lens =
    acond [] [Any]
      (const v {})
      (mapn TL (protect (append v)))
  in
  let filter_aux (D:schema) (E:schema) (some_D:view) : lens =
    cond_ff (T E) []  (D::(T D))
      (protect (filter E D {}))
      (append [some_D])
      (const [] [])
      (inner_filter)
```

```
  and inner_filter : lens =
    ccond (E::Any)
      (mapn TL (protect inner_filter); tl {"error"})
      (mapn TL (protect (filter_aux D E some_D)))
  in
    filter_aux D E some_D

let filter_test : lens = filter {a=Any} {b=Any} {a}

test filter_test / [] = []
test filter_test / [{a={1}}, {a={2}}, {b={3}}, {a={4}}]
                 = [{a={1}}, {a={2}}, {a={4}}]
test filter_test / [{a={1}}, {a={2}}, {a={3}}]
                 = [{a={1}}, {a={2}}, {a={3}}]

test filter_test \ [{a={1}}, {a={2}}, {a={3}}]
                   [{a={1}}, {a={2}}, {b={3}}, {a={4}}]
                 = [{a={1}}, {a={2}}, {b={3}}, {a={3}}]
test filter_test \ [{a={1}}, {a={2}}, {a={3}}]
                   [{b={3}}]
                 = [{b={3}}, {a={1}}, {a={2}}, {a={3}}]
test filter_test \ [{a={1}}, {a={2}}, {a={3}}]
                   [{a}, {b={3}}]
                 = [{a={1}}, {b={3}}, {a={2}}, {a={3}}]
test filter_test \ [{a={1}}]
                   [{a},{a},{b={3}}]
                 = [{a={1}}, {b={3}}]
```

4.2.4 Lenses for "Keyed Lists"

`flatten` The `flatten` lens takes a list of keyed trees and flattens it into a bush whose top-level children are the keys from the original list:

```
test flatten / [{k1={a}}, {k2={b}}] = {k1=[{a}], k2=[{b}]}
```

The "keys" in the concrete list need not be distinct. If a key `k` is repeated, the corresponding subtrees from the original concrete list are collected into a list under `k` in the resulting abstract tree:

```
test flatten / [{a={1}}, {b={2}}, {a={3}}] = {a=[{1},{3}], b=[{2}]}
```

In the *putback* direction, the list of subtrees under each key in the abstract tree is distributed into the corresponding positions in the concrete list.

```
test flatten \ {a=[{4},{6}], b=[{5}]}
               [{a={1}}, {b={2}}, {a={3}}]
             = [{a={4}}, {b={5}}, {a={6}}]
```

Any "left over" subtrees from the abstract list are placed at the end of the new concrete list (in some fixed but unspecified order).

```
test flatten \ {a=[{4},{6}], b=[{5}]}
               [{b={1}}, {a={2}}]
             = [{b={5}}, {a={4}}, {a={6}}]
```

Left over elements of the concrete list are deleted from the result.

```
test flatten \ {a=[{1}], b=[{2}]}
               [{b={1}}, {a={2}}, {a={3}}]
             = [{b={2}}, {a={1}}]
```

Note that `flatten` does not obey PutPut:

```
test flatten \ {a=[{}], b=[{}]} [{a},{b}] = [{a},{b}]
test flatten \ {a=[{}], b=[{}]} ( flatten \ {b=[{}]} [{a},{b}]) = [{b},{a}]
```

The `flatten` lens is in fact built from the `flatten_op` lens operator, as `flatten = flatten_op id`. In the *get* direction, `flatten_op l` applies `l` to each element then proceeds as `flatten`. In the *putback* direction, the list of subtrees under each key in the abstract tree is distributed into the corresponding positions in the concrete list, the position being determined by the key exhibited by an element of the concrete list after apply the *get* of `l`. The new concrete element is created by the *putback* of `l`.

```
let l : lens = flatten_op (Prelude.filter {a,b} {})

test l / [{a={1}, z={junk}}, {b={3}}] = {a=[{1}], b=[{3}]}

test l \ {b=[{3}]} [{a={1}, z={junk}}, {b={3}}] = [{b={3}}]
```

It is interesting to notice that `flatten_op l` is not simply `List.map l; flatten`, as `List.map` does not know which elements in the concrete list where deleted, it simply applies *putback* according to the order:

```
let f : lens = Prelude.filter {a,b} {}
let l' : lens = map f; flatten
test l' / [{a={1}, z={junk}}, {b={3}}] = {a=[{1}], b=[{3}]}

test l' \ {b=[{3}]} [{a={1}, z={junk}}, {b={3}}] = [{b={3},z={junk}}]
```

This surprising result is because of the *putback* of `List.map`:

```
test (map f) \ [{b={3}}] [{a={1}, z={junk}}, {b={3}}] = [{b={3},z={junk}}]
```

Currently, `flatten_op` is defined as a built-in lens. We conjecture that it can be defined in terms of other, more primitive, lenses, but we have not (yet) been able to do this.

4.2.5 Conversions Between Names and Lists

It is sometimes necessary to write lenses that manipulate single names as strings. The following primitives form the basis of such lenses: we first (in the *get* direction) use `explode` or `lines` to convert the name to a list of either single characters or lines, as appropriate, then process this list using the list-processing lenses above, and finally use `implode` or `unlines` to convert the processed list back to a single name. The `structuredtext` example shows this process in detail.

`explode` The lens `explode` converts a `Value` to a list of single characters.

```
test explode / \ {""} = []
test explode / \ {focal} = [{f},{o},{c},{a},{l}]
test explode / {} = error
```

`implode` The lens `implode` reverses the action of `explode`.

```
let implode : lens = invert (explode)

test implode / \ [{f},{o},{c},{a},{l}] = {focal}
```

`split` The lens `split k` converts a `Value` to a list of values by splitting it at `k` characters.

`unsplit` The lens `unsplit` reverses the action of `split`.

```
let unsplit (k:name) : lens = invert (split k)
```

`lines` The lens `lines` converts a `Value` to a list of values by splitting it at newline characters.

```
test lines / {""} = [{""}]
test lines / {focal} = [{focal}]
test lines / {"lenses
               |focal"}
            = [{lenses},{focal}]
test lines / {"focal\n"} = [{focal},{""}]
test lines / {} = error
test lines \ [] missing = error
```

`unlines` The lens `unlines` reverses the action of `lines`.

```
let unlines : lens = invert (lines)
```

4.3 Module `Xml`

4.3.1 Encoding

The encoding of XML documents as trees is a simple extension of the encoding of lists described in Section 4.2.1. A sequence of XML elements is encoded as a list of keyed trees, where the keys are the element names and where each element consists of a subtree labeled with a special tag (we use the variable `CHILDREN` to refer to it) representing the sequence of child elements.

```
test id / (load "xml" "<a></a>") =
            [{a = {`CHILDREN = []}}]
test id / (load "xml" "<a><b/><c/><b/></a>") =
            [{a = {`CHILDREN = [{b = {`CHILDREN = []}},
                                {c = {`CHILDREN = []}},
                                {b = {`CHILDREN = []}}]}}]
```

Attributes are represented by an (unordered) collection of additional children of each element node, at the same level as the required `CHILDREN` subtree.

```
test id / (load "xml" "<a attr='foo'></a>") =
            [{a = {`CHILDREN = [], attr = {foo}}}]
```

Parsed character data (PCDATA) is represented by a special tag (we use the variable `PCDATA` to refer to it):

```
test id / (load "xml" "<a>foo<b>bar</b>baz</a>") =
            [{a = {`CHILDREN = [{`PCDATA = {foo}},
                                {b = {`CHILDREN = [{`PCDATA = {bar}}]}},
                                {`PCDATA = {baz}}]}}]
```

4.3.2 Names and Schemas

`CHILDREN, PCDATA` `CHILDREN` and `PCDATA` are predefined names that denote the special tags used by the XML viewer.

`Pcdata, XmlElt, T` The encoding of XML described above is expressed by the following schemas:

```
schema Pcdata = {'PCDATA = NonNullValue}
schema XmlElt = {! \ 'PCDATA = {'CHILDREN = T,              (* subelts *)
                                *\'CHILDREN = Value }}      (* attrs *)
                | Pcdata
and T = List.T XmlElt
```

(Note that the value under `PCDATA` is never the empty string: our XML parser throws away whitespace-only character sequences, so they never show up as PCDATA.)

4.3.3 Lenses

`hoist_pcdata` The `hoist_pcdata` lens behaves almost like `hoist PCDATA`, but also handles the case where the whole tree is empty (corresponding to a null `PCDATA` at this point).

```
let hoist_pcdata : lens =
  acond Pcdata NonNullValue
    (hoist PCDATA)
    (const {""} {})
```

`flatten` The `flatten` lens for XML is a generalized version of `List.flatten`. It removes all the ordering from an XML structure, leaving a "bush" of schema `FlattenedXML`:

```
let ATTRS : name = "@attrs"
schema FlattenedXML = { ¿PCDATA = (List.T NonNullValue),
                        ¿ATTRS = {!*=Value},
                        * \ ('PCDATA,'ATTRS) = (List.T FlattenedXML) }
```

The recursive definition of `flatten` uses an auxiliary lens

```
let flatten : lens =
  assert T;
  List.map (protect flatten_elt);
  List.flatten;
  assert FlattenedXML

and flatten_elt : lens =
  assert XmlElt;
  acond Pcdata Pcdata
    id
    (map (acond {'CHILDREN=Any} {*\'ATTRS=Any}
            (* no attributes: *)
            (hoist CHILDREN; protect flatten)
            (* one or more attributes: *)
            (fork {'CHILDREN} (map (protect flatten)) (plunge ATTRS);
             fork {'ATTRS} id (hoist CHILDREN))))

test flatten_elt /
      ((List.hd []) / (load "xml" "<a>text</a>"))
    = {a = {'PCDATA = [{text}]}}

test flatten /
       (load "xml" "<a> <c/> <b/> </a>")
     = {a = [{b = [{}],
              c = [{}]}]}
```

```
test flatten /
        (load "xml" "<a>foo</a>")
      = {a = [{`PCDATA = [{foo}]}]}

test flatten /
        (load "xml" "<a> <b/> <c/> <b/> </a>")
      = {a = [{b = [{}, {}],
               c = [{}]}]}

test flatten /
        (load "xml" "<a>foo<b/></a>")
      = {a = [{`PCDATA = [{foo}], b = [{}]}]}

test flatten /
        (load "xml" "<a foo='attrfoo'><b/>text</a>")
      = {a = [{`ATTRS = {foo = {attrfoo}},
               `PCDATA = [{text}],
               b = [{}]}]}
```

`squash_flattened` The `squash` lens defined below goes a step further. It assumes that its concrete argument is "non-repetitive" in the sense that each element contains at most one sub-element with a given tag (and at most one PCDATA sub-element).

```
schema FlattenedPcdata = { `PCDATA = [Value] }
schema SimpleFlattenedXML = { ¿PCDATA = [Value],
                              ¿ATTRS = {!=Value,*=Value},
                              *\`PCDATA,`ATTRS = [SimpleFlattenedXML] }
```

For such XML structures, there is no need for any list structure in their abstract representations—i.e., we can take just the head elements of all the lists, yielding something of this schema:

```
schema SquashedPCDATA = { `PCDATA = Value }
schema SquashedXML = { ¿ATTRS = {!=Value,*=Value},
                       ¿PCDATA = Value,
                       *\`PCDATA,`ATTRS = SquashedXML }
```

The `squash_flattened` lens is defined as follows:

```
let squash_flattened : lens =
  assert SimpleFlattenedXML;
  fork {`ATTRS}
    id
    (fork {`PCDATA}
      (map (List.hd []))
      (map (List.hd []; protect squash_flattened)));
  assert SquashedXML
```

`squash` Often, `flatten` and `squash_flattened` are used together, so we give the combination a name.

```
let squash : lens = flatten; squash_flattened

test squash /
        (load "xml" "<a>foo</a>")
      = {a = {`PCDATA = {foo}}}

test squash /
        (load "xml" "<a>foo<b>bar</b><c>baz</c></a>")
      = {a = {`PCDATA = {foo}, b = {`PCDATA = {bar}}, c = {`PCDATA = {baz}}}}
```

```
test squash /
        (load "xml" "<a attr='foo'>text<b/></a>")
      = {a = {'ATTRS = {attr = {foo}}, 'PCDATA = {text}, b = {}}}
```

It is also useful to squash a single element:

```
let squash_elt : lens = flatten_elt; map squash_flattened

test squash_elt /
          ((List.hd []) / (load "xml" "<a attr='foo'>text<b/></a>"))
        = {a = {'ATTRS = {attr = {foo}}, 'PCDATA = {text}, b = {}}}
```

The above tests actually happen to work in both directions. In general, though, the *putback* direction of squash doesn't have enough information to determine the ordering of newly created elements, so it just inserts them in alphabetical order. This means that completely new structures will be completely sorted:

```
test squash \
        {a = {'PCDATA={foo},
              d={'PCDATA={bar},A={first}},
              c={'PCDATA={baz},z={second}}}}
        missing
      =
        (load "xml" "<a>foo
                    |   <c>baz<z><second/></z></c>
                    |   <d>bar
                    |      <A><first/></A>
                    |   </d>
                    |</a>")
```

(Note that PCDATA sorts before elements.)

More interestingly, if we *putback* an abstract structure into an existing concrete one, the existing structures will retain their old order. New substructures will be placed at the end.

```
test squash \
        {a = {'PCDATA = {foo},
              extra = {},
              b = {'PCDATA = {bar}, extra={}},
              c = {'PCDATA = {baz}}}}
        (load "xml" "<a> foo <b>bar</b> <c>baz</c> </a>")
      =
        (load "xml" "<a> foo <b>bar<extra/></b> <c>baz</c> <extra/> </a>")
```

4.4 Module Plist

The plist format is a generic XML representation for structured data (strings, arrays, and finite maps). It is heavily used in OS X for storage of application preference files, system configuration information, etc.

The abstract form of plists is described by the following schema:

```
schema T =
    {dict = {*=T}}
    {array = List.T T}
    {string = Value}
    {integer = Value}

and DictElt = List.Cons Value (List.Cons T List.Nil)
```

The core of the module is a group of mutually recursive lenses that walk over the plist structure and perform an appropriate transformation at each node.

```
let plist_object_lens : lens =
  mapn "dict" (protect dict_lens);
  mapn "array" (protect array_lens);
  mapp {"string","integer"} (protect leaf_lens)

and array_lens : lens =
  mapn Xml.CHILDREN (List.map (protect plist_object_lens));
  hoist Xml.CHILDREN

and dict_lens : lens =
  mapn Xml.CHILDREN
    (List.groupby2;
     List.map (protect keypair_lens; pivot List.HD; map (focus List.TL {}; List.hd []));
     List.flatten;
     map (List.hd []));
  hoist Xml.CHILDREN

and keypair_lens : lens =
  mapn List.HD (hoist "key"; protect leaf_lens);
  mapn List.TL (mapn List.HD (protect plist_object_lens))

and leaf_lens : lens =
  hoist Xml.CHILDREN;
  acond [] { `(List.TL)=[], `(List.HD)={` Xml.PCDATA = {"BLANK"={}, *\("BLANK")=Any}}}
    (const [{`Xml.PCDATA={"BLANK"}}] [])
    id;
  List.hd [];
  hoist Xml.PCDATA
```

The module's main lens, `l`, deals with a little top-level boilerplate and invokes `plist_object_lens` to process the body of the file.

```
let l : lens =
  List.hd [];
  hoist "plist";
  focus Xml.CHILDREN { "version" = {"1.0"} };
  List.hd [];
  plist_object_lens
```

4.5 Module `Relational`

```
let _ : lens =
  check (rename "R" "S" "A" "D") :
  {{ R(A, B, C) with {(A, C) -> B} where A <> B }}
  <<~>
  {{ S(B, C, D) with {(C, D) -> B} where B = D -> FALSE }}

test rename "R" "S" "A" "C" /
  {{{ R(A, B) = {(1, 2)} }}}
= {{{ S(B, C) = {(2, 1)} }}}
test rename "R" "S" "A" "C" \
  {{{ S(B, C) = {(3, 4),
                 (2, 1)} }}}
```

```
  {{{ R(A, B) = {(1, 2)} }}}
= {{{ R(A, B) = {(1, 2),
                 (4, 3)} }}}

let _ : lens =
  check (select "R" with {A -> B} "S" where C = "1") :
  {{ R(A, B, C) with {A -> B} }}
  <<->
  {{ S(A, B, C) where C = "1" with {A -> B} }}

test select "R" with {} "S" where (A = "1" \/ B = C) /
  {{{ R(A, B, C) = {(1, 1, 1),
                    (2, 2, 2),
                    (1, 2, 3),
                    (3, 2, 1)} }}}
= {{{ S(A, B, C) = {(1, 1, 1),
                    (2, 2, 2),
                    (1, 2, 3)} }}}

test select "R" with {A -> B} "S" where C = "1" /
  {{{ R(A, B, C) = {(1, 2, 1),
                    (1, 2, 3),
                    (3, 2, 1),
                    (3, 2, 3)} }}}
= {{{ S(A, B, C) = {(1, 2, 1),
                    (3, 2, 1)} }}}

test select "R" with {A -> B} "S" where C = "1" \
  {{{ S(A, B, C) = {(1, 3, 1),
                    (4, 4, 1)} }}}
  {{{ R(A, B, C) = {(1, 2, 1),
                    (1, 2, 3),
                    (3, 2, 1),
                    (3, 2, 3)} }}}
= {{{ R(A, B, C) = {(1, 3, 1),
                    (1, 3, 3),
                    (4, 4, 1),
                    (3, 2, 3)} }}}

(* "Surprising" example from PODS paper. *)
test select "R" with {A -> B} "S" where B = "2" \
  {{{ S(A, B, C) = {(1, 2, 2)} }}}
  {{{ R(A, B, C) = {(1, 1, 1)} }}}
= {{{ R(A, B, C) = {(1, 2, 2)} }}}

let _ : lens =
  check (drop "R" "S" "B" {B} "0") :
  {{ R(A, B) }}
  <<->
  {{ S(A) }}

let _ : lens =
  check (drop "R" "S" "B" {B} "0") :
  {{ R(A, B, C) where (A = C /\ B = "0") /\ (C = "3" /\ A <> B) }}
  <<->
  {{ S(A, C) where A = "3" /\ A = C }}
```

```
let _ : lens =
  check (drop "R" "S" "D" {B, C} "0") :
  {{ R(A, B, C, D) with {A -> C, (B, C) -> D} }}
  <<->
  {{ S(A, B, C) with {A -> C} }}

test drop "R" "S" "C" {C} "0" /
  {{{ R(A, B, C) = {(1, 1, 1)} }}}
= {{{ S(A, B)    = {(1, 1)}    }}}

test drop "R" "S" "C" {C} "0" \
  {{{ S(A, B)    = {(1, 1)}    }}}
  {{{ R(A, B, C) = {(1, 1, 1),
                    (1, 1, 2),
                    (1, 2, 3)} }}}
= {{{ R(A, B, C) = {(1, 1, 1),
                    (1, 1, 2)} }}}

test drop "R" "S" "C" {B} "0" \
  {{{ S(A, B)    = {(1, 1),
                    (2, 1),
                    (2, 2)}    }}}
  {{{ R(A, B, C) = {(1, 1, 3)} }}}
= {{{ R(A, B, C) = {(1, 1, 3),
                    (2, 1, 3),
                    (2, 2, 0)} }}}

test drop "R" "S" "C" {B} "0" \
  {{{ S(A, B)    = {(1, 1),
                    (2, 1),
                    (2, 2)}    }}}
  {{{ R(A, B, C) = {(3, 1, 3)} }}}
= {{{ R(A, B, C) = {(1, 1, 3),
                    (2, 1, 3),
                    (2, 2, 0)} }}}
let _ : lens =
  check (join_dl "R" with {A -> (B,C)} "S" with {B -> (C,D)} "R") :
  {{ R(A, B, C) with {A -> (B,C)},
     S(B, C, D) with {B -> (C,D)} }}
  <<->
  {{ R(A, B, C, D) with {A -> (B,C), B -> (C,D)} }}

test join_dl "R" with {} "S" with {B -> C} "T" /
  {{{ R(A, B) = {(1, 1),
                 (2, 1),
                 (3, 3)}
      S(B, C) = {(1, 1),
                 (2, 4),
                 (3, 9),
                 (4, 16)} }}}
= {{{ T(A, B, C) = {(1, 1, 1),
                    (2, 1, 1),
                    (3, 3, 9)} }}}

test join_dl "R" with {} "S" with {B -> C} "T" \
  {{{ T(A, B, C) = {(1, 1, 2),
```

```
                        (3, 4, 9)} }}}
  {{{ R(A, B) = {(1, 1),
                 (2, 1),
                 (3, 3)}
      S(B, C) = {(1, 1),
                 (2, 4),
                 (3, 9),
                 (4, 16)} }}}
= {{{ R(A, B) = {(1, 1),
                 (3, 4)}
      S(B, C) = {(1, 2),
                 (2, 4),
                 (3, 9),
                 (4, 9)} }}}
```

Chapter 5

Synchronization

*Under construction. For now, see [**?**].*

Chapter 6

The Harmony System

Under construction.

6.1 Running Harmony

- command-line arguments
- `FOCALPATH`
- encoding keys

6.2 Running the Web Demos Locally

If you want to build new demos and make them available on the web for others to play with, you'll need to run the demo script on your own web server. The basic steps are as follows:

1. * Start a local webserver and make sure it can serve PHP pages. On OSX, for example, it can be done something like this. First, edit the web server configuration file

   ```
   sudo emacs /private/etc/httpd/httpd.conf
   ```

   and uncomment two lines involving PHP.

   Now put a symlink from the web server's default location to wherever you keep your personal web space.

   ```
   sudo mv /Library/WebServer/Documents /Library/WebServer/Documents.orig
   sudo ln -s ~/pub /Library/WebServer/Documents
   ```

   Next, restart the web server by toggling "Personal web sharing" control in the "Sharing" system preference panel.

2. Make a symlink to your harmony directory from somewhere in your web space:

   ```
   ln -s ~/current/harmony ~/pub
   ```

3. Point your browser to `http://localhost/harmony/html/demobody.php` and see if the usual demo page gets displayed.

4. Put your demos in a new subdirectory (say, `mydemo`) under `harmony/examples`. This directory should definitely contain a file `demos.php` and an executable file `harmonize-mydemos` — see the existing subdirectories of `examples` to see how this is done.

5. Edit `harmony/html/demobody.php`, search for `get_demos_from`, and add a line

```
get_demos_from("mydemo");
```

   to what's already there.

6.3 Navigating the Distribution

If you want to check out the code, here is one reasonable order to look at the files:

`src/v.mli`	basic definitions of trees
`src/lens.mli`	basic definitions of lenses
`src/lib/native/prelude.ml`	the most important primitive lenses
`src/lib/lenses/prelude.fcl`	some important derived lenses
`examples/*`	lots of real-world lenses
`src/sync.ml`	the synchronization algorithm
`src/harmony.ml`	the top-level program

Chapter 7

Using Harmony From Unison

Harmony is designed to work with files on a single machine. To support cross-machine harmonization, we have extended the Unison file synchronizer so that it can call out to Harmony as an external merge program. Here are the steps for setting this up:

1. Build the harmony instances you plan to use and install them someplace on your search path on the machine that is going to act as client when Unison runs (i.e., the machine where Unison is invoked and where the user interface is displayed).

   You can install all of the Harmony instance binaries at once (in your `$HOME/bin` directory by default) by typing `make install` at the top level.

   There is no need to put Harmony's Focal library files anywhere special: the needed libraries are embedded in the binary of each Harmony instance when it is linked.

2. Get yourself a recent version of Unison, either from the Unison home page (if you're willing to recompile from source)

   ```
   http://www.cis.upenn.edu/~bcpierce/unison
   ```

   or from one of the many binary distributions.

   Version 2.17 has the basic functionality needed here, but we continue making improvements and fixing bugs, so later versions may work better.

3. Add `merge` commands to your Unison profile(s) to tell it how to call the appropriate Harmony instances for different sorts of files. For example, if you want to use the structured text instance of Harmony for all `.txt` files and the bookmark instance for the Safari bookmark file, you might add this to your profile:

   ```
   merge = Name *.txt -> harmonize-structuredtext
               -ar 'CURRENTARCH' -r1 'CURRENT1' -r2 'CURRENT2'
               -newar 'NEWARCH' -newr1 'NEW1' -newr2 'NEW2'
   merge = Name Bookmarks.plist -> harmonize-bookmarks
               -ar CURRENTARCH -r1 CURRENT1 -r2 'CURRENT2'
               -newar 'NEWARCH' -newr1 'NEW1' -newr2 'NEW2'
   ```

   (These rules are written out on several lines here to avoid going off the edge of the page; in your profile, there should be just a single long line for each `merge` rule.)

   These rules will apply only when Unison is *not* running in batch mode. Use `mergebatch` instead of `merge` if you want to live dangerously.

4. Add the rules

```
backupcurrent = Name *.txt
backupcurrent = Name Bookmarks.plist
```

to your Unison profile.

The effect of these rules is to make Unison keep backups for these files more aggressively than usual: instead of just making backups when it overwrites some file with a new version from the other replica (or not making any backups at all, depending on how you've set the `backup` preference), it will make sure that there is *always* a backup copy corresponding to the current, synchronized state at the end of every successful run. This means that, if both replicas should be changed between runs of Unison (so that Unison sees a conflict), it will always have an appropriate file to pass to Harmony to use as the last common state for this file.

5. Test your setup. (The test will be illustrated using the structured text instance of Harmony, but you can substitute another if you like.)

   (a) Create a test file (e.g., `foo.txt`) on one replica containing several lines of text

   ```
   A
   b
   c
   D
   e
   ```

   and run Unison to propagate it to the other:

   ```
   unison myprofile -path foo.txt
   ```

   (Assuming that you put the above rules, together with the usual rules for `root` and so on, into myprofile.prf in your Unison directory and that you created `foo.txt` the root directory of your replica.)

   (b) Edit `foo.txt` on one replica and run Unison again (with the same command line). Note that, because only one replica has changed, there is no conflict and Unison just copies the file directly without invoking Harmony.

   (c) Now edit both copies of `foo.txt` in different ways (e.g., change `A` to —AAAAY— in one copy and `e` to `aieee` in the other). Run Unison again. You should see something like this:

   ```
   changed  <-M-> changed    foo.txt
   ```

   The two arrowheads signal a conflict (as usual in Unison), and the `M` signals that, by default, Unison is going to attempt to merge the two versions. (You can override this default as usual, to tell it to skip this file or just copy one version over the other.)

   Tell Unison to proceed. After a pause (while Unison transfers the server's copy of the file onto the client machine), you should see Harmony running.

   (d) Harmony itself has no user interface: it just tells you what it thinks needs to be done to synchronize the changes in the two copies of the file and then does it and outputs the updated files. Unison, however, will ask you for confirmation before actually overwriting the real files on both replicas. It is a good idea to look carefully at what Harmony has done before agreeing to this!

   (e) Now, change the `foo.txt` in conflicting ways — e.g., change `c` to `see` in one replica and to `sea` in the other. At the same time, make a non-conflicting change to each file (e.g., change the second line in one and the last line in the other; make sure to keep at least one lowercase letter in each of these lines, so as not to complicate the example by changing the way the file is parsed into a tree).

   Run Unison and Harmony again. Notice, in Harmony's output, that the non-conflicting changes have been accepted and the conflicting one marked as such.

(f) Take a look at both copies of `foo.txt`. Notice that the non-conflicting changes are now reflected in both, while the conflicting changes are left alone.

Since this run of Harmony ended with at least one unresolved conflict, Unison will *not* update its own archive to reflect any of the changes that have been made. That is, as far as Unison is concerned, this file has not been successfully synchronized, so it does not update its own recollection of the file's last synchronized state. This means that

i. if you change the third line in one of the copies back to `c` and run Unison (and Harmony), the third line from the other replica will be copied into this one; and

ii. if you make another change to one of the lines where you made non-conflicting changes before, this change will be flagged as a conflict rather than being propagated.

The bottom line is that, when Harmony signals a conflict, you should repair it as soon as possible. This can be achieved either by editing the two copies so that they become equal or by editing one of the copies to "back out" its version of the conflicting change.

6. That's it!

7. OK, not quite it. A couple of fine points should be noted:

   - This combination of Unison and Harmony can handle only homogeneous harmonization: it is not currently possible to synchronize a Safari bookmarks file with a Mozilla bookmarks file on another host.
   - If Harmony signals that it has succeeded with no conflicts but the final versions of the files are not byte-for-byte identical (which can happen if the lens used by this Harmony instance omits some of the information in the concrete files for purposes of synchronbization), then Unison will overwrite one of the copies with the other. (This may seem a little surprising, but it is consistent with the fact that Unison deals with updates to just one of the replicas at a time by simply copying the updated one over the unchanged one. This avoids the cost of running Harmony most of the time.)

Chapter 8

Case Studies

Under construction. For now, see the demos in the `examples` *directory.*

Bibliography