The Ur/Web Manual

Adam Chlipala

April 21, 2013

Contents

1 Introduction

Ur is a programming language designed to introduce richer type system features into functional programming in the tradition of ML and Haskell. Ur is functional, pure, statically typed, and strict. Ur supports a powerful kind of *metaprogramming* based on *type-level computation with type-level records*.

Ur/Web is Ur plus a special standard library and associated rules for parsing and optimization. Ur/Web supports construction of dynamic web applications backed by SQL databases. The signature of the standard library is such that well-typed Ur/Web programs "don't go wrong" in a very broad sense. Not only do they not crash during particular page generations, but they also may not:

- Suffer from any kinds of code-injection attacks
- Return invalid HTML
- Contain dead intra-application links
- Have mismatches between HTML forms and the fields expected by their handlers
- Include client-side code that makes incorrect assumptions about the "AJAX"-style services that the remote web server provides
- Attempt invalid SQL queries
- Use improper marshaling or unmarshaling in communication with SQL databases or between browsers and web servers

This type safety is just the foundation of the Ur/Web methodology. It is also possible to use metaprogramming to build significant application pieces by analysis of type structure. For instance, the demo includes an ML-style functor for building an admin interface for an arbitrary SQL table. The type system guarantees that the admin interface sub-application that comes out will always be free of the above-listed bugs, no matter which well-typed table description is given as input.

The Ur/Web compiler also produces very efficient object code that does not use garbage collection. These compiled programs will often be even more efficient than what most programmers would bother to write in C. The compiler also generates JavaScript versions of client-side code, with no need to write those parts of applications in a different language.

The official web site for Ur is:

`http://www.impredicative.com/ur/`

2 Installation

If you are lucky, then the following standard command sequence will suffice for installation, in a directory to which you have unpacked the latest distribution tarball.

```
./configure
make
sudo make install
```

Some other packages must be installed for the above to work. At a minimum, you need a standard UNIX shell, with standard UNIX tools like sed and GCC (or an alternate C compiler) in your execution path; MLton, the whole-program optimizing compiler for Standard ML; and the development files for the OpenSSL C library. As of this writing, in the "testing" version of Debian Linux, this command will install the more uncommon of these dependencies:

```
apt-get install mlton libssl-dev
```

To build programs that access SQL databases, you also need one of these client libraries for supported backends.

```
apt-get install libpq-dev libmysqlclient15-dev libsqlite3-dev
```

It is also possible to access the modules of the Ur/Web compiler interactively, within Standard ML of New Jersey. To install the prerequisites in Debian testing:

```
apt-get install smlnj libsmlnj-smlnj ml-yacc ml-lpt
```

To begin an interactive session with the Ur compiler modules, run `make smlnj`, and then, from within an `sml` session, run `CM.make "src/urweb.cm";`. The `Compiler` module is the main entry point.

To run an SQL-backed application with a backend besides SQLite, you will probably want to install one of these servers.

```
apt-get install postgresql-8.4 mysql-server-5.1
```

To use the Emacs mode, you must have a modern Emacs installed. We assume that you already know how to do this, if you're in the business of looking for an Emacs mode. The demo generation facility of the compiler will also call out to Emacs to syntax-highlight code, and that process depends on the `htmlize` module, which can be installed in Debian testing via:

```
apt-get install emacs-goodies-el
```

If you don't want to install the Emacs mode, run `./configure` with the argument `-without-emacs`.

Even with the right packages installed, configuration and building might fail to work. After you run `./configure`, you will see the values of some named environment variables printed. You may need to adjust these values to get proper installation for your system. To change a value, store your preferred alternative in the corresponding UNIX environment variable, before running `./configure`. For instance, here is how to change the list of extra arguments that the Ur/Web compiler will pass to the C compiler and linker on every invocation. Some older GCC versions need this setting to mask a bug in function inlining.

```
CCARGS=-fno-inline ./configure
```

Since the author is still getting a handle on the GNU Autotools that provide the build system, you may need to do some further work to get started, especially in environments with significant differences from Linux (where most testing is done). The variables `PGHEADER`, `MSHEADER`, and `SQHEADER` may be used to set the proper C header files to include for the development libraries of PostgreSQL, MySQL, and SQLite, respectively. To get libpq to link, one OS X user reported setting `CCARGS="-I/opt/local/include -L/opt/local/lib/postgresql84"`, after creating a symbolic link with `ln -s /opt/local/include/postgresql84 /opt/local/include/postgresql`.

The Emacs mode can be set to autoload by adding the following to your `.emacs` file.

```
(add-to-list 'load-path "/usr/local/share/emacs/site-lisp/urweb-mode")
(load "urweb-mode-startup")
```

Change the path in the first line if you chose a different Emacs installation path during configuration.

3 Command-Line Compiler

3.1 Project Files

The basic inputs to the `urweb` compiler are project files, which have the extension `.urp`. Here is a sample `.urp` file.

```
database dbname=test
sql crud1.sql

crud
crud1
```

The `database` line gives the database information string to pass to libpq. In this case, the string only says to connect to a local database named `test`.

The `sql` line asks for an SQL source file to be generated, giving the commands to run to create the tables and sequences that this application expects to find. After building this `.urp` file, the following commands could be used to initialize the database, assuming that the current UNIX user exists as a Postgres user with database creation privileges:

```
createdb test
psql -f crud1.sql test
```

A blank line separates the named directives from a list of modules to include in the project. Any line may contain a shell-script-style comment, where any suffix of a line starting at a hash character `#` is ignored.

For each entry `M` in the module list, the file `M.urs` is included in the project if it exists, and the file `M.ur` must exist and is always included.

Here is the complete list of directive forms. "FFI" stands for "foreign function interface," Ur's facility for interaction between Ur programs and C and JavaScript libraries.

- `[allow|deny] [url|mime|requestHeader|responseHeader|env] PATTERN` registers a rule governing which URLs, MIME types, HTTP request headers, HTTP response headers, or environment variable names are allowed to appear explicitly in this application. The first such rule to match a name determines the verdict. If `PATTERN` ends in `*`, it is interpreted as a prefix rule. Otherwise, a string must match it exactly.
- `alwaysInline PATH` requests that every call to the referenced function be inlined. Section 10 explains how functions are assigned path strings.
- `benignEffectful Module.ident` registers an FFI function or transaction as having side effects. The optimizer avoids removing, moving, or duplicating calls to such functions. Every effectful FFI function must be registered, or the optimizer may make invalid transformations. This version of the `effectful` directive registers that this function only has side effects that remain local to a single page generation.
- `clientOnly Module.ident` registers an FFI function or transaction that may only be run in client browsers.
- `clientToServer Module.ident` adds FFI type `Module.ident` to the list of types that are OK to marshal from clients to servers. Values like XML trees and SQL queries are hard to marshal without introducing expensive validity checks, so it's easier to ensure that the server never trusts clients to send such values. The file `include/urweb.h` shows examples of the C support functions that are required of any type that may be marshalled. These include `attrify`, `urlify`, and `unurlify` functions.
- `coreInline TREESIZE` sets how many nodes the AST of a function definition may have before the optimizer stops trying hard to inline calls to that function. (This is one of two options for one of two intermediate languages within the compiler.)

- `database DBSTRING` sets the string to pass to libpq to open a database connection.
- `debug` saves some intermediate C files, which is mostly useful to help in debugging the compiler itself.
- `effectful Module.ident` registers an FFI function or transaction as having side effects. The optimizer avoids removing, moving, or duplicating calls to such functions. Every effectful FFI function must be registered, or the optimizer may make invalid transformations. (Note that merely assigning a function a `transaction`-based type does not mark it as effectful in this way!)
- `exe FILENAME` sets the filename to which to write the output executable. The default for file `P.urp` is `P.exe`.
- `ffi FILENAME` reads the file `FILENAME.urs` to determine the interface to a new FFI module. The name of the module is calculated from `FILENAME` in the same way as for normal source files. See the files `include/urweb.h` and `src/c/urweb.c` for examples of C headers and implementations for FFI modules. In general, every type or value `Module.ident` becomes `uw_Module_ident` in C.
- `include FILENAME` adds `FILENAME` to the list of files to be `#include`d in C sources. This is most useful for interfacing with new FFI modules.
- `jsFunc Module.ident=name` gives the JavaScript name of an FFI value.
- `library FILENAME` parses `FILENAME.urp` and merges its contents with the rest of the current file's contents. If `FILENAME.urp` doesn't exist, the compiler also tries `FILENAME/lib.urp`.
- `limit class num` sets a resource usage limit for generated applications. The limit `class` will be set to the non-negative integer `num`. The classes are:
  - `cleanup`: maximum number of cleanup operations (e.g., entries recording the need to deallocate certain temporary objects) that may be active at once per request
  - `clients`: maximum number of simultaneous connections to one application by web clients waiting for new asynchronous messages sent with `Basis.send`
  - `database`: maximum size of a database file (currently only used by SQLite, which interprets the parameter as a number of pages, where page size is itself a quantity configurable in SQLite)
  - `deltas`: maximum number of messages sendable in a single request handler with `Basis.send`
  - `globals`: maximum number of global variables that FFI libraries may set in a single request context
  - `headers`: maximum size (in bytes) of per-request buffer used to hold HTTP headers for generated pages
  - `heap`: maximum size (in bytes) of per-request heap for dynamically allocated data
  - `inputs`: maximum number of top-level form fields per request
  - `messages`: maximum size (in bytes) of per-request buffer used to hold a single outgoing message sent with `Basis.send`
  - `page`: maximum size (in bytes) of per-request buffer used to hold HTML content of generated pages
  - `script`: maximum size (in bytes) of per-request buffer used to hold JavaScript content of generated pages
  - `subinputs`: maximum number of form fields per request, excluding top-level fields
  - `time`: maximum running time of a single page request, in units of approximately 0.1 seconds
  - `transactionals`: maximum number of custom transactional actions (e.g., sending an e-mail) that may be run in a single page generation

- `link FILENAME` adds `FILENAME` to the list of files to be passed to the linker at the end of compilation. This is most useful for importing extra libraries needed by new FFI modules.
- `linker CMD` sets `CMD` as the command line prefix to use for linking C object files. The command line will be completed with a space-separated list of `.o` and `.a` files, `-L` and `-l` flags, and finally with a `-o` flag to set the location where the executable should be written.
- `minHeap NUMBYTES` sets the initial size for thread-local heaps used in handling requests. These heaps grow automatically as needed (up to any maximum set with `limit`), but each regrow requires restarting the request handling process.
- `monoInline TREESIZE` sets how many nodes the AST of a function definition may have before the optimizer stops trying hard to inline calls to that function. (This is one of two options for one of two intermediate languages within the compiler.)
- `noXsrfProtection URIPREFIX` turns off automatic cross-site request forgery protection for the page handler identified by the given URI prefix. This will avoid checking cryptographic signatures on cookies, which is generally a reasonable idea for some pages, such as login pages that are going to discard all old cookie values, anyway.
- `onError Module.var` changes the handling of fatal application errors. Instead of displaying a default, ugly error 500 page, the error page will be generated by calling function `Module.var` on a piece of XML representing the error message. The error handler should have type $\mathsf{xbody} \to \mathsf{transaction\ page}$. Note that the error handler *cannot* be in the application's main module, since that would register it as explicitly callable via URLs.
- `path NAME=VALUE` creates a mapping from `NAME` to `VALUE`. This mapping may be used at the beginnings of filesystem paths given to various other configuration directives. A path like `$NAME/rest` is expanded to `VALUE/rest`. There is an initial mapping from the empty name (for paths like `$/list`) to the directory where the Ur/Web standard library is installed. If you accept the default `configure` options, this directory is `/usr/local/lib/urweb/ur`.
- `prefix PREFIX` sets the prefix included before every URI within the generated application. The default is `/`.
- `profile` generates an executable that may be used with gprof.
- `rewrite KIND FROM TO` gives a rule for rewriting canonical module paths. For instance, the canonical path of a page may be `Mod1.Mod2.mypage`, while you would rather the page were accessed via a URL containing only `page`. The directive `rewrite url Mod1/Mod2/mypage page` would accomplish that. The possible values of `KIND` determine which kinds of objects are affected. The kind `all` matches any object, and `url` matches page URLs. The kinds `table`, `sequence`, and `view` match those sorts of SQL entities, and `relation` matches any of those three. `cookie` matches HTTP cookies, and `style` matches CSS class names. If `FROM` ends in `/*`, it is interpreted as a prefix matching rule, and rewriting occurs by replacing only the appropriate prefix of a path with `TO`. The `TO` field may be left empty to express the idea of deleting a prefix. For instance, `rewrite url Main/*` will strip all `Main/` prefixes from URLs. While the actual external names of relations and styles have parts separated by underscores instead of slashes, all rewrite rules must be written in terms of slashes. An optional suffix of `[-]` for a `rewrite` directive asks to additionally replace all `_` characters with `-` characters, which can be handy for, e.g., interfacing with an off-the-shelf CSS library that prefers hyphens over underscores.
- `safeGet URI` asks to allow the page handler assigned this canonical URI prefix to cause persistent side effects, even if accessed via an HTTP `GET` request.

- `script URL` adds `URL` to the list of extra JavaScript files to be included at the beginning of any page that uses JavaScript. This is most useful for importing JavaScript versions of functions found in new FFI modules.
- `serverOnly Module.ident` registers an FFI function or transaction that may only be run on the server.
- `sigfile PATH` sets a path where your application should look for a key to use in cryptographic signing. This is used to prevent cross-site request forgery attacks for any form handler that both reads a cookie and creates side effects. If the referenced file doesn't exist, an application will create it and read its saved data on future invocations. You can also initialize the file manually with any contents at least 16 bytes long; the first 16 bytes will be treated as the key.
- `sql FILENAME` sets where to write an SQL file with the commands to create the expected database schema. The default is not to create such a file.
- `timeFormat FMT` accepts a time format string, as processed by the POSIX C function `strftime()`. This controls the default rendering of time values, via the show instance for time.
- `timeout N` sets to `N` seconds the amount of time that the generated server will wait after the last contact from a client before determining that that client has exited the application. Clients that remain active will take the timeout setting into account in determining how often to ping the server, so it only makes sense to set a high timeout to cope with browser and network delays and failures. Higher timeouts can lead to more unnecessary client information taking up memory on the server. The timeout goes unused by any page that doesn't involve the `recv` function, since the server only needs to store per-client information for clients that receive asynchronous messages.

3.2 Building an Application

To compile project `P.urp`, simply run

```
urweb P
```

The output executable is a standalone web server. Run it with the command-line argument `-h` to see which options it takes. If the project file lists a database, the web server will attempt to connect to that database on startup. See Section 10 for an explanation of the URI mapping convention, which determines how each page of your application may be accessed via URLs.

To time how long the different compiler phases run, without generating an executable, run

```
urweb -timing P
```

To stop the compilation process after type-checking, run

```
urweb -tc P
```

It is often worthwhile to run `urweb` in this mode, because later phases of compilation can take significantly longer than type-checking alone, and the type checker catches many errors that would traditionally be found through debugging a running application.

A related option is `-dumpTypes`, which, as long as parsing succeeds, outputs to stdout a summary of the kinds of all identifiers declared with `con` and the types of all identifiers declared with `val` or `val rec`. This information is dumped even if there are errors during type inference. Compiler error messages go to stderr, not stdout, so it is easy to distinguish the two kinds of output programmatically. A refined version of this option is `-dumpTypesOnError`, which only has an effect when there are compilation errors.

It may be useful to combine another option `-unifyMore` with `-dumpTypes`. Ur/Web type inference proceeds in a series of stages, where the first is standard Hindley-Milner type inference as in ML, and the later phases add more complex aspects. By default, an error detected in one phase cuts off the execution of later phases. However, the later phases might still determine more values of unification variables. These

value choices might be "misguided," since earlier phases have not come up with reasonable types at a coarser detail level; but the unification decisions may still be useful for debugging and program understanding. So, if a run with `-dumpTypes` leaves unification variables undetermined in positions where you would like to see best-effort guesses instead, consider `-unifyMore`. Note that `-unifyMore` has no effect when type inference succeeds fully, but it may lead to many more error messages when inference fails.

To output information relevant to CSS stylesheets (and not finish regular compilation), run

```
urweb -css P
```

The first output line is a list of categories of CSS properties that would be worth setting on the document body. The remaining lines are space-separated pairs of CSS class names and categories of properties that would be worth setting for that class. The category codes are divided into two varieties. Codes that reveal properties of a tag or its (recursive) children are `B` for block-level elements, `C` for table captions, `D` for table cells, `L` for lists, and `T` for tables. Codes that reveal properties of the precise tag that uses a class are `b` for block-level elements, `t` for tables, `d` for table cells, `-` for table rows, `H` for the possibility to set a height, `N` for non-replaced inline-level elements, `R` for replaced inline elements, and `W` for the possibility to set a width.

Ur/Web type inference can take a significant amount of time, so it can be helpful to cache type-inferred versions of source files. This mode can be activated by running

```
urweb daemon start
```

Further `urweb` invocations in the same working directory will send requests to a background daemon process that reuses type inference results whenever possible, tracking source file dependencies and modification times. To stop the background daemon, run

```
urweb daemon stop
```

Communication happens via a UNIX domain socket in file `.urweb_daemon` in the working directory.

Some other command-line parameters are accepted:

- `-boot`: Run Ur/Web from a build tree (and not from a system install). This is useful if you're testing the compiler and don't want to install it. It forces generation of statically linked executables.
- `-db <DBSTRING>`: Set database connection information, using the format expected by Postgres's `PQconnectdb()`, which is `name1=value1 ... nameN=valueN`. The same format is also parsed and used to discover connection parameters for MySQL and SQLite. The only significant settings for MySQL are `host`, `hostaddr`, `port`, `dbname`, `user`, and `password`. The only significant setting for SQLite is `dbname`, which is interpreted as the filesystem path to the database. Additionally, when using SQLite, a database string may be just a file path.
- `-dbms [postgres|mysql|sqlite]`: Sets the database backend to use.
  - `postgres`: This is PostgreSQL, the default. Among the supported engines, Postgres best matches the design philosophy behind Ur, with a focus on consistent views of data, even in the face of much concurrency. Different database engines have different quirks of SQL syntax. Ur/Web tends to use Postgres idioms where there are choices to be made, though the compiler translates SQL as needed to support other backends.

    A command sequence like this can initialize a Postgres database, using a file `app.sql` generated by the compiler:

    ```
    createdb app
    psql -f app.sql app
    ```

  - `mysql`: This is MySQL, another popular relational database engine that uses persistent server processes. Ur/Web needs transactions to function properly. Many installations of MySQL use non-transactional storage engines by default. Ur/Web generates table definitions that try to use MySQL's InnoDB engine, which supports transactions. You can edit the first line of a generated `.sql` file to change this behavior, but it really is true that Ur/Web applications will exhibit bizarre behavior if you choose an engine that ignores transaction commands.

    A command sequence like this can initialize a MySQL database:

    ```
    echo "CREATE DATABASE app" | mysql
    mysql -D app <app.sql
    ```

  - `sqlite`: This is SQLite, a simple filesystem-based transactional database engine. With this backend, Ur/Web applications can run without any additional server processes. The other engines are generally preferred for large-workload performance and full admin feature sets, while SQLite is popular for its low resource footprint and ease of set-up.

    A command like this can initialize an SQLite database:

    ```
    sqlite3 path/to/database/file <app.sql
    ```

- `-dumpSource`: When compilation fails, output to stderr the complete source code of the last intermediate program before the compilation phase that signaled the error. (Warning: these outputs can be very long and aren't especially optimized for readability!)
- `-limit class num`: Equivalent to the `limit` directive from `.urp` files
- `-moduleOf FILENAME`: Prints the Ur/Web module name corresponding to source file `FILENAME`, exiting immediately afterward.
- `-output FILENAME`: Set where the application executable is written.
- `-path NAME VALUE`: Set the value of path variable `$NAME` to `VALUE`, for use in `.urp` files.
- `-prefix PREFIX`: Equivalent to the `prefix` directive from `.urp` files
- `-protocol [http|cgi|fastcgi|static]`: Set the protocol that the generated application speaks.
  - `http`: This is the default. It is for building standalone web servers that can be accessed by web browsers directly.
  - `cgi`: This is the classic protocol that web servers use to generate dynamic content by spawning new processes. While Ur/Web programs may in general use message-passing with the `send` and `recv` functions, that functionality is not yet supported in CGI, since CGI needs a fresh process for each request, and message-passing needs to use persistent sockets to deliver messages.

    Since Ur/Web treats paths in an unusual way, a configuration line like this one can be used to configure an application that was built with URL prefix `/Hello`:

    ```
    ScriptAlias /Hello /path/to/hello.exe
    ```

    A different method can be used for, e.g., a shared host, where you can only configure Apache via `.htaccess` files. Drop the generated executable into your web space and mark it as CGI somehow. For instance, if the script ends in `.exe`, you might put this in `.htaccess` in the directory containing the script:

    ```
    Options +ExecCGI
    AddHandler cgi-script .exe
    ```

Additionally, make sure that Ur/Web knows the proper URI prefix for your script. For instance, if the script is accessed via `http://somewhere/dir/script.exe`, then include this line in your `.urp` file:

```
prefix /dir/script.exe/
```

To access the `foo` function in the `Bar` module, you would then hit `http://somewhere/dir/script.exe/Bar/foo`.

If your application contains form handlers that read cookies before causing side effects, then you will need to use the `sigfile .urp` directive, too.

- `fastcgi`: This is a newer protocol inspired by CGI, wherein web servers can start and reuse persistent external processes to generate dynamic content. Ur/Web doesn't implement the whole protocol, but Ur/Web's support has been tested to work with the `mod_fastcgi`s of Apache and lighttpd.

  To configure a FastCGI program with Apache, one could combine the above `ScriptAlias` line with a line like this:

  ```
  FastCgiServer /path/to/hello.exe -idle-timeout 99999
  ```

  The idle timeout is only important for applications that use message-passing. Client connections may go long periods without receiving messages, and Apache tries to be helpful and garbage collect them in such cases. To prevent that behavior, we specify how long a connection must be idle to be collected.

  Also see the discussion of the `prefix` directive for CGI above; similar configuration is likely to be necessary for FastCGI. An Ur/Web application won't generally run correctly if it doesn't have a unique URI prefix assigned to it and configured with `prefix`.

  Here is some lighttpd configuration for the same application.

  ```
  fastcgi.server = (
    "/Hello/" =>
    (( "bin-path" => "/path/to/hello.exe",
    "socket" => "/tmp/hello",
    "check-local" => "disable",
    "docroot" => "/",
    "max-procs" => "1"
    ))
  )
  ```

  The least obvious requirement is setting `max-procs` to 1, so that lighttpd doesn't try to multiplex requests across multiple external processes. This is required for message-passing applications, where a single database of client connections is maintained within a multi-threaded server process. Multiple processes may, however, be used safely with applications that don't use message-passing.

  A FastCGI process reads the environment variable `URWEB_NUM_THREADS` to determine how many threads to spawn for handling client requests. The default is 1.

- `static`: This protocol may be used to generate static web pages from Ur/Web code. The output executable expects a single command-line argument, giving the URI of a page to generate. For instance, this argument might be `/main`, in which case a static HTTP response for that page will be written to stdout.

- `-root Name PATH`: Trigger an alternate module convention for all source files found in directory `PATH` or any of its subdirectories. Any file `PATH/foo.ur` defines a module `Name.Foo` instead of the usual `Foo`. Any file `PATH/subdir/foo.ur` defines a module `Name.Subdir.Foo`, and so on for arbitrary nesting of subdirectories.

- `-sigfile PATH`: Same as the `sigfile` directive in `.urp` files
- `-sql FILENAME`: Set where a database set-up SQL script is written.
- `-static`: Link the runtime system statically. The default is to link against dynamic libraries.

There is an additional convenience method for invoking `urweb`. If the main argument is `FOO`, and `FOO.ur` exists but `FOO.urp` doesn't, then the invocation is interpreted as if called on a `.urp` file containing `FOO` as its only main entry, with an additional `rewrite all FOO/*` directive.

3.3 Tutorial Formatting

The Ur/Web compiler also supports rendering of nice HTML tutorials from Ur source files, when invoked like `urweb -tutorial DIR`. The directory `DIR` is examined for files whose names end in `.ur`. Every such file is translated into a `.html` version.

These input files follow normal Ur syntax, with a few exceptions:

- The first line must be a comment like `(* TITLE *)`, where `TITLE` is a string of your choice that will be used as the title of the output page.
- While most code in the output HTML will be formatted as a monospaced code listing, text in regular Ur comments is formatted as normal English text.
- A comment like `(* * HEADING *)` introduces a section heading, with text `HEADING` of your choice.
- To include both a rendering of an Ur expression and a pretty-printed version of its value, bracket the expression with `(* begin eval *)` and `(* end *)`. The result of expression evaluation is pretty-printed with `show`, so the expression type must belong to that type class.
- To include code that should not be shown in the tutorial (e.g., to add a `show` instance to use with `eval`), bracket the code with `(* begin hide *)` and `(* end *)`.

A word of warning: as for demo generation, tutorial generation calls Emacs to syntax-highlight Ur code.

3.4 Run-Time Options

Compiled applications consult a few environment variables to modify their behavior:

- `URWEB_NUM_THREADS`: alternative to the `-t` command-line argument (currently used only by FastCGI)
- `URWEB_STACK_SIZE`: size of per-thread stacks, in bytes
- `URWEB_PQ_CON`: when using PostgreSQL, overrides the compiled-in connection string

4 Ur Syntax

In this section, we describe the syntax of Ur, deferring to a later section discussion of most of the syntax specific to SQL and XML. The sole exceptions are the declaration forms for relations, cookies, and styles.

4.1 Lexical Conventions

We give the Ur language definition in LaTeX math mode, since that is prettier than monospaced ASCII. The corresponding ASCII syntax can be read off directly. Here is the key for mapping math symbols to ASCII character sequences.

LaTeX	ASCII
$\rightarrow$	`->`
$\longrightarrow$	`-->`
$\times$	`*`
λ	`fn`
$\Rightarrow$	`=>`
$\Longrightarrow$	`==>`
$\neq$	`<>`
$\leq$	`<=`
$\geq$	`>=`

x	Normal textual identifier, not beginning with an uppercase letter
X	Normal textual identifier, beginning with an uppercase letter

We often write syntax like e^* to indicate zero or more copies of e, e^+ to indicate one or more copies, and $e,^*$ and $e,^+$ to indicate multiple copies separated by commas. Another separator may be used in place of a comma. The e term may be surrounded by parentheses to indicate grouping; those parentheses should not be included in the actual ASCII.

We write ℓ for literals of the primitive types, for the most part following C conventions. There are int, float, char, and string literals. Character literals follow the SML convention instead of the C convention, written like `#"a"` instead of `'a'`.

This version of the manual doesn't include operator precedences; see `src/urweb.grm` for that.

As in the ML language family, the syntax `(* ... *)` is used for (nestable) comments. Within XML literals, Ur/Web also supports the usual `<!- ... ->` XML comments.

4.2 Core Syntax

Kinds classify types and other compile-time-only entities. Each kind in the grammar is listed with a description of the sort of data it classifies.

Kinds	κ	::=	Type	proper types
			Unit	the trivial constructor
			Name	field names
			$\kappa \rightarrow \kappa$	type-level functions
			$\{\kappa\}$	type-level records
			$(\kappa\times^+)$	type-level tuples
			X	variable
			$X \longrightarrow \kappa$	kind-polymorphic type-level function
			_	wildcard
			(κ)	explicit precedence

Ur supports several different notions of functions that take types as arguments. These arguments can be either implicit, causing them to be inferred at use sites; or explicit, forcing them to be specified manually at use sites. There is a common explicitness annotation convention applied at the definitions of and in the types of such functions.

Explicitness	?	::=	::	explicit
			:::	implicit

Constructors are the main class of compile-time-only data. They include proper types and are classified by kinds.

Constructors	c, τ	::=	$(c) :: \kappa$	kind annotation
			$\hat{x}$	constructor variable
			$\tau \to \tau$	function type
			$x \; ? \; \kappa \to \tau$	polymorphic function type
			$X \longrightarrow \tau$	kind-polymorphic function type
			$\$c$	record type
			$c \; c$	type-level function application
			$\lambda x \; :: \; \kappa \Rightarrow c$	type-level function abstraction
			$X \Longrightarrow c$	type-level kind-polymorphic function abstraction
			$c[\kappa]$	type-level kind-polymorphic function application
			$()$	type-level unit
			$\#X$	field name
			$[(c = c)^*]$	known-length type-level record
			$c +\!\!+ c$	type-level record concatenation
			map	type-level record map
			$(c,^+)$	type-level tuple
			$c.n$	type-level tuple projection ($n \in \mathbb{N}^+$)
			$[c \sim c] \Rightarrow \tau$	guarded type
			$\_ :: \kappa$	wildcard
			(c)	explicit precedence
Qualified uncapitalized variables	$\hat{x}$	::=	x	not from a module
			$M.x$	projection from a module

We include both abstraction and application for kind polymorphism, but applications are only inferred internally; they may not be written explicitly in source programs. Also, in the "known-length type-level record" form, in $c_1 = c_2$ terms, the parser currently only allows c_1 to be of the forms X (as a shorthand for $\#X$) or x, or a natural number to stand for the corresponding field name (e.g., for tuples).

Modules of the module system are described by *signatures*.

Signatures	S	::=	$\mathsf{sig}\ s^*\ \mathsf{end}$	constant
			X	variable
			$\mathsf{functor}(X : S) : S$	functor
			$S\ \mathsf{where\ con}\ x = c$	concretizing an abstract constructor
			$M.X$	projection from a module
Signature items	s	::=	$\mathsf{con}\ x :: \kappa$	abstract constructor
			$\mathsf{con}\ x :: \kappa = c$	concrete constructor
			$\mathsf{datatype}\ x\ x^* = dc\ \vert^+$	algebraic datatype definition
			$\mathsf{datatype}\ x = \mathsf{datatype}\ M.x$	algebraic datatype import
			$\mathsf{val}\ x : \tau$	value
			$\mathsf{structure}\ X : S$	sub-module
			$\mathsf{signature}\ X = S$	sub-signature
			$\mathsf{include}\ S$	signature inclusion
			$\mathsf{constraint}\ c \sim c$	record disjointness constraint
			$\mathsf{class}\ x :: \kappa$	abstract constructor class
			$\mathsf{class}\ x :: \kappa = c$	concrete constructor class
Datatype constructors	dc	::=	X	nullary constructor
			$X\ \mathsf{of}\ \tau$	unary constructor

Patterns are used to describe structural conditions on expressions, such that expressions may be tested against patterns, generating assignments to pattern variables if successful.

Patterns	p	::=	$\_$	wildcard
			x	variable
			ℓ	constant
			$\hat{X}$	nullary constructor
			$\hat{X}\ p$	unary constructor
			$\{(x = p,)^*\}$	rigid record pattern
			$\{(x = p,)^+, \ldots\}$	flexible record pattern
			$p : \tau$	type annotation
			(p)	explicit precedence
Qualified capitalized variables	$\hat{X}$	::=	X	not from a module
			$M.X$	projection from a module

Expressions are the main run-time entities, corresponding to both "expressions" and "statements" in

mainstream imperative languages.

Expressions	e	$::=$	$e : \tau$	type annotation
			$\hat{x}$	variable
			$\hat{X}$	datatype constructor
			ℓ	constant
			$e\ e$	function application
			$\lambda x : \tau \Rightarrow e$	function abstraction
			$e[c]$	polymorphic function application
			$\lambda[x\ ?\ \kappa] \Rightarrow e$	polymorphic function abstraction
			$e[\kappa]$	kind-polymorphic function application
			$X \Longrightarrow e$	kind-polymorphic function abstraction
			$\{(c = e,)^*\}$	known-length record
			$e.c$	record field projection
			$e +\!\!+ e$	record concatenation
			e - c	removal of a single record field
			e -- c	removal of multiple record fields
			$\mathsf{let}\ ed^*\ \mathsf{in}\ e\ \mathsf{end}$	local definitions
			$\mathsf{case}\ e\ \mathsf{of}\ (p \Rightarrow e\vert)^+$	pattern matching
			$\lambda[c \sim c] \Rightarrow e$	guarded expression abstraction
			$e\ !$	guarded expression application
			_	wildcard
			(e)	explicit precedence
Local declarations	ed	$::=$	$\texttt{val}\ x : \tau = e$	non-recursive value
			$\texttt{val rec}\ (x : \tau = e\ \texttt{and})^+$	mutually recursive values

As with constructors, we include both abstraction and application for kind polymorphism, but applications are only inferred internally.

Declarations primarily bring new symbols into context.

Declarations	d	::=	con $x :: \kappa = c$	constructor synonym
			datatype $x\ x^* = dc\ \vert^+$	algebraic datatype definition
			datatype $x =$ datatype $M.x$	algebraic datatype import
			val $x : \tau = e$	value
			val rec $(x : \tau = e$ and$)^+$	mutually recursive values
			structure $X : S = M$	module definition
			signature $X = S$	signature definition
			open M	module inclusion
			constraint $c \sim c$	record disjointness constraint
			open constraints M	inclusion of just the constraints from a module
			table $x : c$	SQL table
			view $x = e$	SQL view
			sequence x	SQL sequence
			cookie $x : \tau$	HTTP cookie
			style $x : \tau$	CSS class
			task $e = e$	recurring task
Modules	M	::=	struct d^* end	constant
			X	variable
			$M.X$	projection
			$M(M)$	functor application
			functor$(X : S) : S = M$	functor abstraction

There are two kinds of Ur files. A file named M`.ur` is an *implementation file*, and it should contain a sequence of declarations d^*. A file named M`.urs` is an *interface file*; it must always have a matching M`.ur` and should contain a sequence of signature items s^*. When both files are present, the overall effect is the same as a monolithic declaration structure M : sig s^* end = struct d^* end. When no interface file is included, the overall effect is similar, with a signature for module M being inferred rather than just checked against an interface.

We omit some extra possibilities in table syntax, deferring them to Section 9.1.1. The concrete syntax of view declarations is also more complex than shown in the table above, with details deferred to Section 9.1.1.

4.3 Shorthands

There are a variety of derived syntactic forms that elaborate into the core syntax from the last subsection. We will present the additional forms roughly following the order in which we presented the constructs that they elaborate into.

In many contexts where record fields are expected, like in a projection $e.c$, a constant field may be written as simply X, rather than $\#X$.

A record type may be written $\{(c = c,)^*\}$, which elaborates to $[(c = c,)^*]$.

The notation $[c_1, \ldots, c_n]$ is shorthand for $[c_1 = (), \ldots, c_n = ()]$.

A tuple type $\tau_1 \times \ldots \times \tau_n$ expands to a record type $\{1 : \tau_1, \ldots, n : \tau_n\}$, with natural numbers as field names. A tuple expression $(e_1, \ldots, e_n)$ expands to a record expression $\{1 = e_1, \ldots, n = e_n\}$. A tuple pattern $(p_1, \ldots, p_n)$ expands to a rigid record pattern $\{1 = p_1, \ldots, n = p_n\}$. Positive natural numbers may be used in most places where field names would be allowed.

The syntax $()$ expands to $\{\}$ as a pattern or expression.

In general, several adjacent λ forms may be combined into one, and kind and type annotations may be omitted, in which case they are implicitly included as wildcards. More formally, for constructor-level abstractions, we can define a new non-terminal $b ::= x \mid (x :: \kappa) \mid X$ and allow composite abstractions of the form $\lambda b^+ \Rightarrow c$, elaborating into the obvious sequence of one core λ per element of b^+.

Further, the signature item or declaration syntax con x $b^+ = c$ is shorthand for wrapping of the appropriate λs around the righthand side c. The b elements may not include X, and there may also be an optional $:: \kappa$ before the $=$.

In some contexts, the parser isn't happy with token sequences like $x :: \_$, to indicate a constructor variable of wildcard kind. In such cases, write the second two tokens as $::\_$, with no intervening spaces. Analogous syntax $:::\_$ is available for implicit constructor arguments.

For any signature item or declaration that defines some entity to be equal to A with classification annotation B (e.g., val $x : B = A$), B and the preceding colon (or similar punctuation) may be omitted, in which case it is filled in as a wildcard.

A signature item or declaration type x or type $x = \tau$ is elaborated into con $x ::$ Type or con $x ::$ Type $= \tau$, respectively.

A signature item or declaration class $x = \lambda y \Rightarrow c$ may be abbreviated class x $y = c$.

Handling of implicit and explicit constructor arguments may be tweaked with some prefixes to variable references. An expression $@x$ is a version of x where all type class instance and disjointness arguments have been made explicit. (For the purposes of this paragraph, the type family Top.folder is a type class, though it isn't marked as one by the usual means; and any record type is considered to be a type class instance type when every field's type is a type class instance type.) An expression $@@x$ achieves the same effect, additionally making explicit all implicit constructor arguments. The default is that implicit arguments are inserted automatically after any reference to a variable, or after any application of a variable to one or more arguments. For such an expression, implicit wildcard arguments are added for the longest prefix of the expression's type consisting only of implicit polymorphism, type class instances, and disjointness obligations. The same syntax works for variables projected out of modules and for capitalized variables (datatype constructors).

At the expression level, an analogue is available of the composite λ form for constructors. We define the language of binders as $b ::= p \mid [x] \mid [x \ ? \ \kappa] \mid X \mid [c \sim c]$. A lone variable $[x]$ stands for an implicit constructor variable of unspecified kind. The standard value-level function binder is recovered as the type-annotated pattern form $x : \tau$. It is a compile-time error to include a pattern p that does not match every value of the appropriate type.

A local val declaration may bind a pattern instead of just a plain variable. As for function arguments, only irrefutable patterns are legal.

The keyword fun is a shorthand for val rec that allows arguments to be specified before the equal sign in the definition of each mutually recursive function, as in SML. Each curried argument must follow the grammar of the b non-terminal introduced two paragraphs ago. A fun declaration is elaborated into a version that adds additional λs to the fronts of the righthand sides, as appropriate.

A signature item functor X_1 $(X_2 : S_1) : S_2$ is elaborated into structure $X_1 :$ functor$(X_2 : S_1) : S_2$. A declaration functor X_1 $(X_2 : S_1) : S_2 = M$ is elaborated into structure $X_1 :$ functor$(X_2 : S_1) : S_2 =$ functor$(X_2 : S_1) : S_2 = M$.

An open constraints declaration is implicitly inserted for the argument of every functor at the beginning of the functor body. For every declaration of the form structure $X : S =$ struct ... end, an open constraints X declaration is implicitly inserted immediately afterward.

A declaration table $x : \{(c = c,)^*\}$ is elaborated into table $x : [(c = c,)^*]$.

The syntax where type is an alternate form of where con.

The syntax if e then e_1 else e_2 expands to case e of Basis.True $\Rightarrow e_1 \mid$ Basis.False $\Rightarrow e_2$.

There are infix operator syntaxes for a number of functions defined in the Basis module. There is $=$ for eq, $\neq$ for neq, $-$ for neg (as a prefix operator) and minus, $+$ for plus, $\times$ for times, / for div, % for mod, $<$ for lt, $\leq$ for le, $>$ for gt, and $\geq$ for ge.

A signature item table $x : c$ is shorthand for val $x :$ Basis.sql_table c []. view $x : c$ is shorthand for val $x :$ Basis.sql_view c, sequence x is short for val $x :$ Basis.sql_sequence. cookie $x : \tau$ is shorthand for val $x :$ Basis.http_cookie τ, and style x is shorthand for val $x :$ Basis.css_class.

5 Static Semantics

In this section, we give a declarative presentation of Ur's typing rules and related judgments. Inference is the subject of the next section; here, we assume that an oracle has filled in all wildcards with concrete values.

Since there is significant mutual recursion among the judgments, we introduce them all before beginning to give rules. We use the same variety of contexts throughout this section, implicitly introducing new sorts of context entries as needed.

- $\Gamma \vdash \kappa$ expresses kind well-formedness.
- $\Gamma \vdash c :: \kappa$ assigns a kind to a constructor in a context.
- $\Gamma \vdash c \sim c$ proves the disjointness of two record constructors; that is, that they share no field names. We overload the judgment to apply to pairs of field names as well.
- $\Gamma \vdash c \hookrightarrow C$ proves that record constructor c decomposes into set C of field names and record constructors.
- $\Gamma \vdash c \equiv c$ proves the computational equivalence of two constructors. This is often called a *definitional equality* in the world of type theory.
- $\Gamma \vdash e : \tau$ is a standard typing judgment.
- $\Gamma \vdash p \rightsquigarrow \Gamma; \tau$ combines typing of patterns with calculation of which new variables they bind.
- $\Gamma \vdash d \rightsquigarrow \Gamma$ expresses how a declaration modifies a context. We overload this judgment to apply to sequences of declarations, as well as to signature items and sequences of signature items.
- $\Gamma \vdash S \equiv S$ is the signature equivalence judgment.
- $\Gamma \vdash S \leq S$ is the signature compatibility judgment. We write $\Gamma \vdash S$ as shorthand for $\Gamma \vdash S \leq S$.
- $\Gamma \vdash M : S$ is the module signature checking judgment.
- $\mathsf{proj}(M, \overline{s}, V)$ is a partial function for projecting a signature item from $\overline{s}$, given the module M that we project from. V may be $\mathsf{con}\ x$, $\mathsf{datatype}\ x$, $\mathsf{val}\ x$, $\mathsf{signature}\ X$, or $\mathsf{structure}\ X$. The parameter M is needed because the projected signature item may refer to other items from $\overline{s}$.
- $\mathsf{selfify}(M, \overline{s})$ adds information to signature items $\overline{s}$ to reflect the fact that we are concerned with the particular module M. This function is overloaded to work over individual signature items as well.

5.1 Kind Well-Formedness

$$\frac{}{\Gamma \vdash \mathsf{Type}} \quad \frac{}{\Gamma \vdash \mathsf{Unit}} \quad \frac{}{\Gamma \vdash \mathsf{Name}} \quad \frac{\Gamma \vdash \kappa_1 \quad \Gamma \vdash \kappa_2}{\Gamma \vdash \kappa_1 \to \kappa_2} \quad \frac{\Gamma \vdash \kappa}{\Gamma \vdash \{\kappa\}} \quad \frac{\forall i : \Gamma \vdash \kappa_i}{\Gamma \vdash (\kappa_1 \times \ldots \times \kappa_n)}$$

$$\frac{X \in \Gamma}{\Gamma \vdash X} \quad \frac{\Gamma, X \vdash \kappa}{\Gamma \vdash X \longrightarrow \kappa}$$

5.2 Kinding

We write $[X \mapsto \kappa_1]\kappa_2$ for capture-avoiding substitution of κ_1 for X in κ_2.

$$\frac{\Gamma \vdash c :: \kappa}{\Gamma \vdash (c) :: \kappa :: \kappa} \quad \frac{x :: \kappa \in \Gamma}{\Gamma \vdash x :: \kappa} \quad \frac{x :: \kappa = c \in \Gamma}{\Gamma \vdash x :: \kappa}$$

$$\frac{\Gamma \vdash M : \mathsf{sig}\ \overline{s}\ \mathsf{end} \quad \mathsf{proj}(M, \overline{s}, \mathsf{con}\ x) = \kappa}{\Gamma \vdash M.x :: \kappa} \quad \frac{\Gamma \vdash M : \mathsf{sig}\ \overline{s}\ \mathsf{end} \quad \mathsf{proj}(M, \overline{s}, \mathsf{con}\ x) = (\kappa, c)}{\Gamma \vdash M.x :: \kappa}$$

$$\frac{\Gamma \vdash \tau_1 :: \mathsf{Type} \quad \Gamma \vdash \tau_2 :: \mathsf{Type}}{\Gamma \vdash \tau_1 \to \tau_2 :: \mathsf{Type}} \quad \frac{\Gamma, x :: \kappa \vdash \tau :: \mathsf{Type}}{\Gamma \vdash x\ ?\ \kappa \to \tau :: \mathsf{Type}} \quad \frac{\Gamma, X \vdash \tau :: \mathsf{Type}}{\Gamma \vdash X \longrightarrow \tau :: \mathsf{Type}} \quad \frac{\Gamma \vdash c :: \{\mathsf{Type}\}}{\Gamma \vdash \$c :: \mathsf{Type}}$$

$$\frac{\Gamma \vdash c_1 :: \kappa_1 \to \kappa_2 \quad \Gamma \vdash c_2 :: \kappa_1}{\Gamma \vdash c_1\ c_2 :: \kappa_2} \quad \frac{\Gamma, x :: \kappa_1 \vdash c :: \kappa_2}{\Gamma \vdash \lambda x\ ::\ \kappa_1 \Rightarrow c :: \kappa_1 \to \kappa_2}$$

$$\frac{\Gamma \vdash c :: X \to \kappa \quad \Gamma \vdash \kappa'}{\Gamma \vdash c[\kappa'] :: [X \mapsto \kappa']\kappa} \quad \frac{\Gamma, X \vdash c :: \kappa}{\Gamma \vdash X \Longrightarrow c :: X \to \kappa}$$

$$\frac{}{\Gamma \vdash () :: \mathsf{Unit}} \quad \frac{}{\Gamma \vdash \#X :: \mathsf{Name}}$$

$$\frac{\forall i : \Gamma \vdash c_i : \mathsf{Name} \quad \Gamma \vdash c'_i :: \kappa \quad \forall i \neq j : \Gamma \vdash c_i \sim c_j}{\Gamma \vdash [\overline{c_i = c'_i}] :: \{\kappa\}} \quad \frac{\Gamma \vdash c_1 :: \{\kappa\} \quad \Gamma \vdash c_2 :: \{\kappa\} \quad \Gamma \vdash c_1 \sim c_2}{\Gamma \vdash c_1 +\!\!+ c_2 :: \{\kappa\}}$$

$$\frac{}{\Gamma \vdash \mathsf{map} :: (\kappa_1 \to \kappa_2) \to \{\kappa_1\} \to \{\kappa_2\}}$$

$$\frac{\forall i : \Gamma \vdash c_i :: \kappa_i}{\Gamma \vdash (\overline{c}) :: (\kappa_1 \times \ldots \times \kappa_n)} \quad \frac{\Gamma \vdash c :: (\kappa_1 \times \ldots \times \kappa_n)}{\Gamma \vdash c.i :: \kappa_i}$$

$$\frac{\Gamma \vdash c_1 :: \{\kappa\} \quad \Gamma \vdash c_2 :: \{\kappa'\} \quad \Gamma, c_1 \sim c_2 \vdash \tau :: \mathsf{Type}}{\Gamma \vdash \lambda[c_1 \sim c_2] \Rightarrow \tau :: \mathsf{Type}}$$

5.3 Record Disjointness

$$\frac{\Gamma \vdash c_1 \hookrightarrow C_1 \quad \Gamma \vdash c_2 \hookrightarrow C_2 \quad \forall c'_1 \in C_1, c'_2 \in C_2 : \Gamma \vdash c'_1 \sim c'_2}{\Gamma \vdash c_1 \sim c_2} \quad \frac{X \neq X'}{\Gamma \vdash X \sim X'}$$

$$\frac{c'_1 \sim c'_2 \in \Gamma \quad \Gamma \vdash c'_1 \hookrightarrow C_1 \quad \Gamma \vdash c'_2 \hookrightarrow C_2 \quad c_1 \in C_1 \quad c_2 \in C_2}{\Gamma \vdash c_1 \sim c_2}$$

$$\frac{}{\Gamma \vdash c \hookrightarrow \{c\}} \quad \frac{}{\Gamma \vdash [\overline{c = c'}] \hookrightarrow \{\overline{c}\}} \quad \frac{\Gamma \vdash c_1 \hookrightarrow C_1 \quad \Gamma \vdash c_2 \hookrightarrow C_2}{\Gamma \vdash c_1 +\!\!+ c_2 \hookrightarrow C_1 \cup C_2} \quad \frac{\Gamma \vdash c \equiv c' \quad \Gamma \vdash c' \hookrightarrow C}{\Gamma \vdash c \hookrightarrow C} \quad \frac{\Gamma \vdash c \hookrightarrow C}{\Gamma \vdash \mathsf{map}\ f\ c \hookrightarrow C}$$

5.4 Definitional Equality

We use $\mathcal{C}$ to stand for a one-hole context that, when filled, yields a constructor. The notation $\mathcal{C}[c]$ plugs c into $\mathcal{C}$. We omit the standard definition of one-hole contexts. We write $[x \mapsto c_1]c_2$ for capture-avoiding substitution of c_1 for x in c_2, with analogous notation for substituting a kind in a constructor.

$$\frac{}{\Gamma \vdash c \equiv c} \quad \frac{\Gamma \vdash c_2 \equiv c_1}{\Gamma \vdash c_1 \equiv c_2} \quad \frac{\Gamma \vdash c_1 \equiv c_2 \quad \Gamma \vdash c_2 \equiv c_3}{\Gamma \vdash c_1 \equiv c_3} \quad \frac{\Gamma \vdash c_1 \equiv c_2}{\Gamma \vdash \mathcal{C}[c_1] \equiv \mathcal{C}[c_2]}$$

$$\frac{x :: \kappa = c \in \Gamma}{\Gamma \vdash x \equiv c} \quad \frac{\Gamma \vdash M : \mathsf{sig}\ \overline{s}\ \mathsf{end} \quad \mathsf{proj}(M, \overline{s}, \mathsf{con}\ x) = (\kappa, c)}{\Gamma \vdash M.x \equiv c} \quad \frac{}{\Gamma \vdash (\overline{c}).i \equiv c_i}$$

$$\frac{}{\Gamma \vdash (\lambda x :: \kappa \Rightarrow c)\ c' \equiv [x \mapsto c']c} \quad \frac{}{\Gamma \vdash (X \Longrightarrow c)[\kappa] \equiv [X \mapsto \kappa]c}$$

$$\frac{}{\Gamma \vdash c_1 +\!\!+ c_2 \equiv c_2 +\!\!+ c_1} \quad \frac{}{\Gamma \vdash c_1 +\!\!+ (c_2 +\!\!+ c_3) \equiv (c_1 +\!\!+ c_2) +\!\!+ c_3}$$

$$\frac{}{\Gamma \vdash [] +\!\!+ c \equiv c} \quad \frac{}{\Gamma \vdash [\overline{c_1 = c_1'}] +\!\!+ [\overline{c_2 = c_2'}] \equiv [\overline{c_1 = c_1'}, \overline{c_2 = c_2'}]}$$

$$\frac{}{\Gamma \vdash \mathsf{map}\ f\ [] \equiv []} \quad \frac{}{\Gamma \vdash \mathsf{map}\ f\ ([c_1 = c_2] +\!\!+ c) \equiv [c_1 = f\ c_2] +\!\!+ \mathsf{map}\ f\ c}$$

$$\frac{}{\Gamma \vdash \mathsf{map}\ (\lambda x \Rightarrow x)\ c \equiv c} \quad \frac{}{\Gamma \vdash \mathsf{map}\ f\ (\mathsf{map}\ f'\ c) \equiv \mathsf{map}\ (\lambda x \Rightarrow f\ (f'\ x))\ c}$$

$$\frac{}{\Gamma \vdash \mathsf{map}\ f\ (c_1 +\!\!+ c_2) \equiv \mathsf{map}\ f\ c_1 +\!\!+ \mathsf{map}\ f\ c_2}$$

5.5 Expression Typing

We assume the existence of a function T assigning types to literal constants. It maps integer constants to Basis.int, float constants to Basis.float, character constants to Basis.char, and string constants to Basis.string.

We also refer to a function $\mathcal{I}$, such that $\mathcal{I}(\tau)$ "uses an oracle" to instantiate all constructor function arguments at the beginning of τ that are marked implicit; i.e., replace $x_1 ::: \kappa_1 \to \ldots \to x_n ::: \kappa_n \to \tau$ with $[x_1 \mapsto c_1] \ldots [x_n \mapsto c_n]\tau$, where the c_is are inferred and τ does not start like $x ::: \kappa \to \tau'$.

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e : \tau : \tau} \quad \frac{\Gamma \vdash e : \tau' \quad \Gamma \vdash \tau' \equiv \tau}{\Gamma \vdash e : \tau} \quad \frac{}{\Gamma \vdash \ell : T(\ell)}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \mathcal{I}(\tau)} \quad \frac{\Gamma \vdash M : \mathsf{sig}\ \overline{s}\ \mathsf{end} \quad \mathsf{proj}(M, \overline{s}, \mathsf{val}\ x) = \tau}{\Gamma \vdash M.x : \mathcal{I}(\tau)} \quad \frac{X : \tau \in \Gamma}{\Gamma \vdash X : \mathcal{I}(\tau)} \quad \frac{\Gamma \vdash M : \mathsf{sig}\ \overline{s}\ \mathsf{end} \quad \mathsf{proj}(M, \overline{s}, \mathsf{val}\ X) = \tau}{\Gamma \vdash M.X : \mathcal{I}(\tau)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \quad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1 \Rightarrow e : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash e : x :: \kappa \to \tau \quad \Gamma \vdash c :: \kappa}{\Gamma \vdash e[c] : [x \mapsto c]\tau} \quad \frac{\Gamma, x :: \kappa \vdash e : \tau}{\Gamma \vdash \lambda[x\ ?\ \kappa] \Rightarrow e : x\ ?\ \kappa \to \tau}$$

$$\frac{\Gamma \vdash e : X \longrightarrow \tau \quad \Gamma \vdash \kappa}{\Gamma \vdash e[\kappa] : [X \mapsto \kappa]\tau} \quad \frac{\Gamma, X \vdash e : \tau}{\Gamma \vdash X \Longrightarrow e : X \longrightarrow \tau}$$

$$\frac{\forall i : \Gamma \vdash c_i :: \mathsf{Name} \quad \Gamma \vdash e_i : \tau_i \quad \forall i \neq j : \Gamma \vdash c_i \sim c_j}{\Gamma \vdash \{\overline{c = e}\} : \{\overline{c : \tau}\}} \quad \frac{\Gamma \vdash e : \$([c = \tau] +\!\!+ c')}{\Gamma \vdash e.c : \tau} \quad \frac{\Gamma \vdash e_1 : \$c_1 \quad \Gamma \vdash e_2 : \$c_2 \quad \Gamma \vdash c_1 \sim c_2}{\Gamma \vdash e_1 +\!\!+ e_2 : \$(c_1 +\!\!+ c_2)}$$

$$\frac{\Gamma \vdash e : \$([c = \tau] +\!\!+ c')}{\Gamma \vdash e - c : \$c'} \quad \frac{\Gamma \vdash e : \$(c +\!\!+ c')}{\Gamma \vdash e \mathbin{--} c : \$c'}$$

$$\frac{\Gamma \vdash \overline{ed} \rightsquigarrow \Gamma' \quad \Gamma' \vdash e : \tau}{\Gamma \vdash \mathsf{let}\ \overline{ed}\ \mathsf{in}\ e\ \mathsf{end} : \tau} \quad \frac{\forall i : \Gamma \vdash p_i \rightsquigarrow \Gamma_i, \tau' \quad \Gamma_i \vdash e_i : \tau}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \overline{p \Rightarrow e} : \tau}$$

$$\frac{\Gamma \vdash c_1 :: \{\kappa\} \quad \Gamma \vdash c_2 :: \{\kappa'\} \quad \Gamma, c_1 \sim c_2 \vdash e : \tau}{\Gamma \vdash \lambda[c_1 \sim c_2] \Rightarrow e : \lambda[c_1 \sim c_2] \Rightarrow \tau} \quad \frac{\Gamma \vdash e : [c_1 \sim c_2] \Rightarrow \tau \quad \Gamma \vdash c_1 \sim c_2}{\Gamma \vdash e\,! : \tau}$$

5.6 Pattern Typing

$$\overline{\Gamma \vdash \_ \rightsquigarrow \Gamma; \tau} \quad \overline{\Gamma \vdash x \rightsquigarrow \Gamma, x : \tau; \tau} \quad \overline{\Gamma \vdash \ell \rightsquigarrow \Gamma; T(\ell)}$$

$$\frac{X : \overline{x ::: \mathsf{Type}} \to \tau \in \Gamma \quad \tau \text{ not a function type}}{\Gamma \vdash X \rightsquigarrow \Gamma; \overline{[x_i \mapsto \tau_i']}\tau} \quad \frac{X : \overline{x ::: \mathsf{Type}} \to \tau'' \to \tau \in \Gamma \quad \Gamma \vdash p \rightsquigarrow \Gamma'; \overline{[x_i \mapsto \tau_i']}\tau''}{\Gamma \vdash X\ p \rightsquigarrow \Gamma'; \overline{[x_i \mapsto \tau_i']}\tau}$$

$$\frac{\Gamma \vdash M : \mathsf{sig}\ \overline{s}\ \mathsf{end} \quad \mathsf{proj}(M, \overline{s}, \mathsf{val}\ X) = \overline{x ::: \mathsf{Type}} \to \tau \quad \tau \text{ not a function type}}{\Gamma \vdash M.X \rightsquigarrow \Gamma; \overline{[x_i \mapsto \tau_i']}\tau}$$

$$\frac{\Gamma \vdash M : \mathsf{sig}\ \overline{s}\ \mathsf{end} \quad \mathsf{proj}(M, \overline{s}, \mathsf{val}\ X) = \overline{x ::: \mathsf{Type}} \to \tau'' \to \tau \quad \Gamma \vdash p \rightsquigarrow \Gamma'; \overline{[x_i \mapsto \tau_i']}\tau''}{\Gamma \vdash M.X\ p \rightsquigarrow \Gamma'; \overline{[x_i \mapsto \tau_i']}\tau}$$

$$\frac{\Gamma_0 = \Gamma \quad \forall i : \Gamma_i \vdash p_i \rightsquigarrow \Gamma_{i+1}; \tau_i}{\Gamma \vdash \{\overline{x = p}\} \rightsquigarrow \Gamma_n; \{\overline{x = \tau}\}} \quad \frac{\Gamma_0 = \Gamma \quad \forall i : \Gamma_i \vdash p_i \rightsquigarrow \Gamma_{i+1}; \tau_i}{\Gamma \vdash \{\overline{x = p}, \ldots\} \rightsquigarrow \Gamma_n; \$([\overline{x = \tau}] +\!\!+ c)}$$

$$\frac{\Gamma \vdash p \rightsquigarrow \Gamma'; \tau' \quad \Gamma \vdash \tau' \equiv \tau}{\Gamma \vdash p : \tau \rightsquigarrow \Gamma'; \tau}$$

5.7 Declaration Typing

We use an auxiliary judgment $\overline{y}; x; \Gamma \vdash \overline{dc} \rightsquigarrow \Gamma'$, expressing the enrichment of Γ with the types of the datatype constructors $\overline{dc}$, when they are known to belong to datatype x with type parameters $\overline{y}$.

We presuppose the existence of a function $\mathcal{O}$, where $\mathcal{O}(M, \overline{s})$ implements the `open` declaration by producing a context with the appropriate entry for each available component of module M with signature items $\overline{s}$. Where possible, $\mathcal{O}$ uses "transparent" entries (e.g., an abstract type $M.x$ is mapped to $x :: \mathsf{Type} = M.x$), so that the relationship with M is maintained. A related function $\mathcal{O}_c$ builds a context containing the disjointness constraints found in $\overline{s}$. We write $\kappa_1^n \to \kappa$ as a shorthand, where $\kappa_1^0 \to \kappa = \kappa$ and $\kappa_1^{n+1} \to \kappa_2 = \kappa_1 \to (\kappa_1^n \to \kappa_2)$. We write $\mathsf{len}(\overline{y})$ for the length of vector $\overline{y}$ of variables.

$$\frac{}{\Gamma \vdash \cdot \rightsquigarrow \Gamma} \quad \frac{\Gamma \vdash d \rightsquigarrow \Gamma' \quad \Gamma' \vdash \overline{d} \rightsquigarrow \Gamma''}{\Gamma \vdash d, \overline{d} \rightsquigarrow \Gamma''}$$

$$\frac{\Gamma \vdash c :: \kappa}{\Gamma \vdash \mathsf{con}\ x :: \kappa = c \rightsquigarrow \Gamma, x :: \kappa = c} \qquad \frac{\overline{y}; x; \Gamma, x :: \mathsf{Type}^{\mathsf{len}(\overline{y})} \to \mathsf{Type} \vdash \overline{dc} \rightsquigarrow \Gamma'}{\Gamma \vdash \mathsf{datatype}\ x\ \overline{y} = \overline{dc} \rightsquigarrow \Gamma'}$$

$$\frac{\Gamma \vdash M : \mathsf{sig}\ \overline{s}\ \mathsf{end} \quad \mathsf{proj}(M, \overline{s}, \mathsf{datatype}\ z) = (\overline{y}, \overline{dc}) \quad \overline{y}; x; \Gamma, x :: \mathsf{Type}^{\mathsf{len}(\overline{y})} \to \mathsf{Type} = M.z \vdash \overline{dc} \rightsquigarrow \Gamma'}{\Gamma \vdash \mathsf{datatype}\ x = \mathsf{datatype}\ M.z \rightsquigarrow \Gamma'}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{val}\ x : \tau = e \rightsquigarrow \Gamma, x : \tau}$$

$$\frac{\forall i : \Gamma, \overline{x : \tau} \vdash e_i : \tau_i \quad e_i \text{ starts with an expression } \lambda\text{, optionally preceded by constructor and disjointness } \lambda\text{s}}{\Gamma \vdash \mathsf{val\ rec}\ \overline{x : \tau = e} \rightsquigarrow \Gamma, \overline{x : \tau}}$$

$$\frac{\Gamma \vdash M : S \quad M \text{ not a constant or application}}{\Gamma \vdash \mathsf{structure}\ X : S = M \rightsquigarrow \Gamma, X : S} \qquad \frac{\Gamma \vdash M : \mathsf{sig}\ \overline{s}\ \mathsf{end}}{\Gamma \vdash \mathsf{structure}\ X : S = M \rightsquigarrow \Gamma, X : \mathsf{selfify}(X, \overline{s})}$$

$$\frac{\Gamma \vdash S}{\Gamma \vdash \mathsf{signature}\ X = S \rightsquigarrow \Gamma, X = S}$$

$$\frac{\Gamma \vdash M : \mathsf{sig}\ \overline{s}\ \mathsf{end}}{\Gamma \vdash \mathsf{open}\ M \rightsquigarrow \Gamma, \mathcal{O}(M, \overline{s})}$$

$$\frac{\Gamma \vdash c_1 :: \{\kappa\} \quad \Gamma \vdash c_2 :: \{\kappa\} \quad \Gamma \vdash c_1 \sim c_2}{\Gamma \vdash \mathsf{constraint}\ c_1 \sim c_2 \rightsquigarrow \Gamma} \qquad \frac{\Gamma \vdash M : \mathsf{sig}\ \overline{s}\ \mathsf{end}}{\Gamma \vdash \mathsf{open\ constraints}\ M \rightsquigarrow \Gamma, \mathcal{O}_c(M, \overline{s})}$$

$$\frac{\Gamma \vdash c :: \{\mathsf{Type}\}}{\Gamma \vdash \mathsf{table}\ x : c \rightsquigarrow \Gamma, x : \mathsf{Basis.sql\_table}\ c\ []} \qquad \frac{\Gamma \vdash e :: \mathsf{Basis.sql\_query}\ []\ []\ (\mathsf{map}\ (\lambda\_ \Rightarrow [])\ c')\ c}{\Gamma \vdash \mathsf{view}\ x = e \rightsquigarrow \Gamma, x : \mathsf{Basis.sql\_view}\ c}$$

$$\frac{}{\Gamma \vdash \mathsf{sequence}\ x \rightsquigarrow \Gamma, x : \mathsf{Basis.sql\_sequence}}$$

$$\frac{\Gamma \vdash \tau :: \mathsf{Type}}{\Gamma \vdash \mathsf{cookie}\ x : \tau \rightsquigarrow \Gamma, x : \mathsf{Basis.http\_cookie}\ \tau} \qquad \frac{}{\Gamma \vdash \mathsf{style}\ x \rightsquigarrow \Gamma, x : \mathsf{Basis.css\_class}}$$

$$\frac{\Gamma \vdash e_1 :: \mathsf{Basis.task\_kind}\ \tau \quad \Gamma \vdash e_2 :: \tau \to \mathsf{Basis.transaction}\ \{\}}{\Gamma \vdash \mathsf{task}\ e_1 = e_2 \rightsquigarrow \Gamma}$$

$$\frac{}{\overline{y}; x; \Gamma \vdash \cdot \rightsquigarrow \Gamma} \qquad \frac{\overline{y}; x; \Gamma \vdash \overline{dc} \rightsquigarrow \Gamma'}{\overline{y}; x; \Gamma \vdash X \mid \overline{dc} \rightsquigarrow \Gamma', X : \overline{y ::: \mathsf{Type}} \to x\ \overline{y}} \qquad \frac{\overline{y}; x; \Gamma \vdash \overline{dc} \rightsquigarrow \Gamma'}{\overline{y}; x; \Gamma \vdash X\ \mathsf{of}\ \tau \mid \overline{dc} \rightsquigarrow \Gamma', X : \overline{y ::: \mathsf{Type}} \to \tau \to x\ \overline{y}}$$

5.8 Signature Item Typing

We appeal to a signature item analogue of the $\mathcal{O}$ function from the last subsection.

This is the first judgment where we deal with constructor classes, for the class forms. We will omit their special handling in this formal specification. Section 6.3 gives an informal description of how constructor classes influence type inference.

$$\frac{}{\Gamma \vdash \cdot \rightsquigarrow \Gamma} \quad \frac{\Gamma \vdash s \rightsquigarrow \Gamma' \quad \Gamma' \vdash \overline{s} \rightsquigarrow \Gamma''}{\Gamma \vdash s, \overline{s} \rightsquigarrow \Gamma''}$$

$$\frac{}{\Gamma \vdash \mathsf{con}\ x :: \kappa \rightsquigarrow \Gamma, x :: \kappa} \quad \frac{\Gamma \vdash c :: \kappa}{\Gamma \vdash \mathsf{con}\ x :: \kappa = c \rightsquigarrow \Gamma, x :: \kappa = c} \quad \frac{\overline{y}; x; \Gamma, x :: \mathsf{Type}^{\mathsf{len}(\overline{y})} \to \mathsf{Type} \vdash \overline{dc} \rightsquigarrow \Gamma'}{\Gamma \vdash \mathsf{datatype}\ x\ \overline{y} = \overline{dc} \rightsquigarrow \Gamma'}$$

$$\frac{\Gamma \vdash M : \mathsf{sig}\ \overline{s}\ \mathsf{end} \quad \mathsf{proj}(M, \overline{s}, \mathsf{datatype}\ z) = (\overline{y}, \overline{dc}) \quad \overline{y}; x; \Gamma, x :: \mathsf{Type}^{\mathsf{len}(\overline{y})} \to \mathsf{Type} = M.z \vdash \overline{dc} \rightsquigarrow \Gamma'}{\Gamma \vdash \mathsf{datatype}\ x = \mathsf{datatype}\ M.z \rightsquigarrow \Gamma'}$$

$$\frac{\Gamma \vdash \tau :: \mathsf{Type}}{\Gamma \vdash \mathsf{val}\ x : \tau \rightsquigarrow \Gamma, x : \tau}$$

$$\frac{\Gamma \vdash S}{\Gamma \vdash \mathsf{structure}\ X : S \rightsquigarrow \Gamma, X : S} \quad \frac{\Gamma \vdash S}{\Gamma \vdash \mathsf{signature}\ X = S \rightsquigarrow \Gamma, X = S}$$

$$\frac{\Gamma \vdash S \quad \Gamma \vdash S \equiv \mathsf{sig}\ \overline{s}\ \mathsf{end}}{\Gamma \vdash \mathsf{include}\ S \rightsquigarrow \Gamma, \mathcal{O}(\overline{s})}$$

$$\frac{\Gamma \vdash c_1 :: \{\kappa\} \quad \Gamma \vdash c_2 :: \{\kappa\}}{\Gamma \vdash \mathsf{constraint}\ c_1 \sim c_2 \rightsquigarrow \Gamma, c_1 \sim c_2}$$

$$\frac{\Gamma \vdash c :: \kappa}{\Gamma \vdash \mathsf{class}\ x :: \kappa = c \rightsquigarrow \Gamma, x :: \kappa = c} \quad \frac{}{\Gamma \vdash \mathsf{class}\ x :: \kappa \rightsquigarrow \Gamma, x :: \kappa}$$

5.9 Signature Compatibility

To simplify the judgments in this section, we assume that all signatures are alpha-varied as necessary to avoid including multiple bindings for the same identifier. This is in addition to the usual alpha-variation of locally bound variables.

We rely on a judgment $\Gamma \vdash \overline{s} \leq s'$, which expresses the occurrence in signature items $\overline{s}$ of an item compatible with s'. We also use a judgment $\Gamma \vdash \overline{dc} \leq \overline{dc}$, which expresses compatibility of datatype definitions.

$$\frac{}{\Gamma \vdash S \equiv S} \quad \frac{\Gamma \vdash S_2 \equiv S_1}{\Gamma \vdash S_1 \equiv S_2} \quad \frac{X = S \in \Gamma}{\Gamma \vdash X \equiv S} \quad \frac{\Gamma \vdash M : \mathsf{sig}\ \overline{s}\ \mathsf{end} \quad \mathsf{proj}(M, \overline{s}, \mathsf{signature}\ X) = S}{\Gamma \vdash M.X \equiv S}$$

$$\frac{\Gamma \vdash S \equiv \mathsf{sig}\ \overline{s^1}\ \mathsf{con}\ x :: \kappa\ \overline{s_2}\ \mathsf{end} \quad \Gamma \vdash c :: \kappa}{\Gamma \vdash S\ \mathsf{where}\ \mathsf{con}\ x = c \equiv \mathsf{sig}\ \overline{s^1}\ \mathsf{con}\ x :: \kappa = c\ \overline{s_2}\ \mathsf{end}} \quad \frac{\Gamma \vdash S \equiv \mathsf{sig}\ \overline{s}\ \mathsf{end}}{\Gamma \vdash \mathsf{sig}\ \overline{s^1}\ \mathsf{include}\ S\ \overline{s^2}\ \mathsf{end} \equiv \mathsf{sig}\ \overline{s^1}\ \overline{s}\ \overline{s^2}\ \mathsf{end}}$$

$$\frac{\Gamma \vdash S_1 \equiv S_2}{\Gamma \vdash S_1 \leq S_2} \quad \frac{}{\Gamma \vdash \mathsf{sig}\ \overline{s}\ \mathsf{end} \leq \mathsf{sig\ end}} \quad \frac{\Gamma \vdash \overline{s} \leq s' \quad \Gamma \vdash s' \rightsquigarrow \Gamma' \quad \Gamma' \vdash \mathsf{sig}\ \overline{s}\ \mathsf{end} \leq \mathsf{sig}\ \overline{s'}\ \mathsf{end}}{\Gamma \vdash \mathsf{sig}\ \overline{s}\ \mathsf{end} \leq \mathsf{sig}\ s'\ \overline{s'}\ \mathsf{end}}$$

$$\frac{\Gamma \vdash s \leq s'}{\Gamma \vdash s\ \overline{s} \leq s'} \quad \frac{\Gamma \vdash s \rightsquigarrow \Gamma' \quad \Gamma' \vdash \overline{s} \leq s'}{\Gamma \vdash s\ \overline{s} \leq s'}$$

$$\frac{\Gamma \vdash S_1' \leq S_1 \quad \Gamma, X : S_1' \vdash S_2 \leq S_2'}{\Gamma \vdash \mathsf{functor}(X : S_1) : S_2 \leq \mathsf{functor}(X : S_1') : S_2'}$$

$$\frac{}{\Gamma \vdash \mathsf{con}\ x :: \kappa \leq \mathsf{con}\ x :: \kappa} \quad \frac{}{\Gamma \vdash \mathsf{con}\ x :: \kappa = c \leq \mathsf{con}\ x :: \kappa} \quad \frac{}{\Gamma \vdash \mathsf{datatype}\ x\ \overline{y} = \overline{dc} \leq \mathsf{con}\ x :: \mathsf{Type}^{\mathsf{len}(\overline{y})} \to \mathsf{Type}}$$

$$\frac{\Gamma \vdash M : \mathsf{sig}\ \overline{s}\ \mathsf{end} \quad \mathsf{proj}(M, \overline{s}, \mathsf{datatype}\ z) = (\overline{y}, \overline{dc})}{\Gamma \vdash \mathsf{datatype}\ x = \mathsf{datatype}\ M.z \leq \mathsf{con}\ x :: \mathsf{Type}^{\mathsf{len}(y)} \to \mathsf{Type}}$$

$$\frac{}{\Gamma \vdash \mathsf{class}\ x :: \kappa \leq \mathsf{con}\ x :: \kappa} \quad \frac{}{\Gamma \vdash \mathsf{class}\ x :: \kappa = c \leq \mathsf{con}\ x :: \kappa}$$

$$\frac{\Gamma \vdash c_1 \equiv c_2}{\Gamma \vdash \mathsf{con}\ x :: \kappa = c_1 \leq \mathsf{con}\ x :: \kappa = c_2} \quad \frac{\Gamma \vdash c_1 \equiv c_2}{\Gamma \vdash \mathsf{class}\ x :: \kappa = c_1 \leq \mathsf{con}\ x :: \kappa = c_2}$$

$$\frac{\Gamma, \overline{y :: \mathsf{Type}} \vdash \overline{dc} \leq \overline{dc'}}{\Gamma \vdash \mathsf{datatype}\ x\ \overline{y} = \overline{dc} \leq \mathsf{datatype}\ x\ \overline{y} = \overline{dc'}}$$

$$\frac{\Gamma \vdash M : \mathsf{sig}\ \overline{s}\ \mathsf{end} \quad \mathsf{proj}(M, \overline{s}, \mathsf{datatype}\ z) = (\overline{y}, \overline{dc}) \quad \Gamma, \overline{y :: \mathsf{Type}} \vdash \overline{dc} \leq \overline{dc'}}{\Gamma \vdash \mathsf{datatype}\ x = \mathsf{datatype}\ M.z \leq \mathsf{datatype}\ x\ \overline{y} = \overline{dc'}}$$

$$\frac{}{\Gamma \vdash \cdot \leq \cdot} \quad \frac{\Gamma \vdash \overline{dc} \leq \overline{dc'}}{\Gamma \vdash X; \overline{dc} \leq X; \overline{dc'}} \quad \frac{\Gamma \vdash \tau_1 \equiv \tau_2 \quad \Gamma \vdash \overline{dc} \leq \overline{dc'}}{\Gamma \vdash X\ \mathsf{of}\ \tau_1; \overline{dc} \leq X\ \mathsf{of}\ \tau_2; \overline{dc'}}$$

$$\frac{\Gamma \vdash M.z \equiv M'.z'}{\Gamma \vdash \mathsf{datatype}\ x = \mathsf{datatype}\ M.z \leq \mathsf{datatype}\ x = \mathsf{datatype}\ M'.z'}$$

$$\frac{\Gamma \vdash \tau_1 \equiv \tau_2}{\Gamma \vdash \mathsf{val}\ x : \tau_1 \leq \mathsf{val}\ x : \tau_2} \quad \frac{\Gamma \vdash S_1 \leq S_2}{\Gamma \vdash \mathsf{structure}\ X : S_1 \leq \mathsf{structure}\ X : S_2} \quad \frac{\Gamma \vdash S_1 \leq S_2 \quad \Gamma \vdash S_2 \leq S_1}{\Gamma \vdash \mathsf{signature}\ X = S_1 \leq \mathsf{signature}\ X = S_2}$$

$$\frac{\Gamma \vdash c_1 \equiv c_1' \quad \Gamma \vdash c_2 \equiv c_2'}{\Gamma \vdash \mathsf{constraint}\ c_1 \sim c_2 \leq \mathsf{constraint}\ c_1' \sim c_2'}$$

$$\frac{}{\Gamma \vdash \mathsf{class}\ x :: \kappa \leq \mathsf{class}\ x :: \kappa} \quad \frac{}{\Gamma \vdash \mathsf{class}\ x :: \kappa = c \leq \mathsf{class}\ x :: \kappa} \quad \frac{\Gamma \vdash c_1 \equiv c_2}{\Gamma \vdash \mathsf{class}\ x :: \kappa = c_1 \leq \mathsf{class}\ x :: \kappa = c_2}$$

$$\frac{}{\Gamma \vdash \mathsf{con}\ x :: \kappa \leq \mathsf{class}\ x :: \kappa} \quad \frac{}{\Gamma \vdash \mathsf{con}\ x :: \kappa = c \leq \mathsf{class}\ x :: \kappa} \quad \frac{\Gamma \vdash c_1 \equiv c_2}{\Gamma \vdash \mathsf{con}\ x :: \kappa = c_1 \leq \mathsf{class}\ x :: \kappa = c_2}$$

5.10 Module Typing

We use a helper function sigOf, which converts declarations and sequences of declarations into their principal signature items and sequences of signature items, respectively.

$$\frac{\Gamma \vdash M : S' \quad \Gamma \vdash S' \leq S}{\Gamma \vdash M : S} \qquad \frac{\Gamma \vdash \overline{d} \rightsquigarrow \Gamma'}{\Gamma \vdash \mathsf{struct}\ \overline{d}\ \mathsf{end} : \mathsf{sig}\ \mathsf{sigOf}(\overline{d})\ \mathsf{end}} \qquad \frac{X : S \in \Gamma}{\Gamma \vdash X : S}$$

$$\frac{\Gamma \vdash M : \mathsf{sig}\ \overline{s}\ \mathsf{end} \quad \mathsf{proj}(M, \overline{s}, \mathsf{structure}\ X) = S}{\Gamma \vdash M.X : S}$$

$$\frac{\Gamma \vdash M_1 : \mathsf{functor}(X : S_1) : S_2 \quad \Gamma \vdash M_2 : S_1}{\Gamma \vdash M_1(M_2) : [X \mapsto M_2]S_2} \qquad \frac{\Gamma \vdash S_1 \quad \Gamma, X : S_1 \vdash S_2 \quad \Gamma, X : S_1 \vdash M : S_2}{\Gamma \vdash \mathsf{functor}(X : S_1) : S_2 = M : \mathsf{functor}(X : S_1) : S_2}$$

$$\begin{aligned}
\mathsf{sigOf}(\cdot) &= \cdot \\
\mathsf{sigOf}(s\ \overline{s'}) &= \mathsf{sigOf}(s)\ \mathsf{sigOf}(\overline{s'})
\end{aligned}$$

$$\begin{aligned}
\mathsf{sigOf}(\mathsf{con}\ x :: \kappa = c) &= \mathsf{con}\ x :: \kappa = c \\
\mathsf{sigOf}(\mathsf{datatype}\ x\ \overline{y} = \overline{dc}) &= \mathsf{datatype}\ x\ \overline{y} = \overline{dc} \\
\mathsf{sigOf}(\mathsf{datatype}\ x = \mathsf{datatype}\ M.z) &= \mathsf{datatype}\ x = \mathsf{datatype}\ M.z \\
\mathsf{sigOf}(\mathsf{val}\ x : \tau = e) &= \mathsf{val}\ x : \tau \\
\mathsf{sigOf}(\mathsf{val\ rec}\ \overline{x : \tau = e}) &= \overline{\mathsf{val}\ x : \tau} \\
\mathsf{sigOf}(\mathsf{structure}\ X : S = M) &= \mathsf{structure}\ X : S \\
\mathsf{sigOf}(\mathsf{signature}\ X = S) &= \mathsf{signature}\ X = S \\
\mathsf{sigOf}(\mathsf{open}\ M) &= \mathsf{include}\ S\ (\text{where}\ \Gamma \vdash M : S) \\
\mathsf{sigOf}(\mathsf{constraint}\ c_1 \sim c_2) &= \mathsf{constraint}\ c_1 \sim c_2 \\
\mathsf{sigOf}(\mathsf{open\ constraints}\ M) &= \cdot \\
\mathsf{sigOf}(\mathsf{table}\ x : c) &= \mathsf{table}\ x : c \\
\mathsf{sigOf}(\mathsf{view}\ x = e) &= \mathsf{view}\ x : c\ (\text{where}\ \Gamma \vdash e : \mathsf{Basis.sql_query}\ []\ []\ (\mathsf{map}\ (\lambda_ \Rightarrow [])\ c')\ c) \\
\mathsf{sigOf}(\mathsf{sequence}\ x) &= \mathsf{sequence}\ x \\
\mathsf{sigOf}(\mathsf{cookie}\ x : \tau) &= \mathsf{cookie}\ x : \tau \\
\mathsf{sigOf}(\mathsf{style}\ x) &= \mathsf{style}\ x
\end{aligned}$$

$$
\begin{array}{rcl}
\mathsf{selfify}(M, \cdot) & = & \cdot \\
\mathsf{selfify}(M, s\ \overline{s'}) & = & \mathsf{selfify}(M, s)\ \mathsf{selfify}(M, \overline{s'}) \\
 & & \\
\mathsf{selfify}(M, \mathsf{con}\ x :: \kappa) & = & \mathsf{con}\ x :: \kappa = M.x \\
\mathsf{selfify}(M, \mathsf{con}\ x :: \kappa = c) & = & \mathsf{con}\ x :: \kappa = c \\
\mathsf{selfify}(M, \mathsf{datatype}\ x\ \overline{y} = \overline{dc}) & = & \mathsf{datatype}\ x\ \overline{y} = \mathsf{datatype}\ M.x \\
\mathsf{selfify}(M, \mathsf{datatype}\ x = \mathsf{datatype}\ M'.z) & = & \mathsf{datatype}\ x = \mathsf{datatype}\ M'.z \\
\mathsf{selfify}(M, \mathsf{val}\ x : \tau) & = & \mathsf{val}\ x : \tau \\
\mathsf{selfify}(M, \mathsf{structure}\ X : S) & = & \mathsf{structure}\ X : \mathsf{selfify}(M.X, \overline{s})\ (\text{where } \Gamma \vdash S \equiv \mathsf{sig}\ \overline{s}\ \mathsf{end}) \\
\mathsf{selfify}(M, \mathsf{signature}\ X = S) & = & \mathsf{signature}\ X = S \\
\mathsf{selfify}(M, \mathsf{include}\ S) & = & \mathsf{include}\ S \\
\mathsf{selfify}(M, \mathsf{constraint}\ c_1 \sim c_2) & = & \mathsf{constraint}\ c_1 \sim c_2 \\
\mathsf{selfify}(M, \mathsf{class}\ x :: \kappa) & = & \mathsf{class}\ x :: \kappa = M.x \\
\mathsf{selfify}(M, \mathsf{class}\ x :: \kappa = c) & = & \mathsf{class}\ x :: \kappa = c
\end{array}
$$

5.11 Module Projection

$$
\begin{array}{rcl}
\mathsf{proj}(M, \mathsf{con}\ x :: \kappa\ \overline{s}, \mathsf{con}\ x) & = & \kappa \\
\mathsf{proj}(M, \mathsf{con}\ x :: \kappa = c\ \overline{s}, \mathsf{con}\ x) & = & (\kappa, c) \\
\mathsf{proj}(M, \mathsf{datatype}\ x\ \overline{y} = \overline{dc}\ \overline{s}, \mathsf{con}\ x) & = & \mathsf{Type}^{\mathsf{len}(\overline{y})} \to \mathsf{Type} \\
\mathsf{proj}(M, \mathsf{datatype}\ x = \mathsf{datatype}\ M'.z\ \overline{s}, \mathsf{con}\ x) & = & (\mathsf{Type}^{\mathsf{len}(\overline{y})} \to \mathsf{Type}, M'.z)\ (\text{where } \Gamma \vdash M' : \mathsf{sig}\ \overline{s'}\ \mathsf{end} \\
 & & \text{and } \mathsf{proj}(M', \overline{s'}, \mathsf{datatype}\ z) = (\overline{y}, \overline{dc})) \\
\mathsf{proj}(M, \mathsf{class}\ x :: \kappa\ \overline{s}, \mathsf{con}\ x) & = & \kappa \to \mathsf{Type} \\
\mathsf{proj}(M, \mathsf{class}\ x :: \kappa = c\ \overline{s}, \mathsf{con}\ x) & = & (\kappa \to \mathsf{Type}, c) \\
 & & \\
\mathsf{proj}(M, \mathsf{datatype}\ x\ \overline{y} = \overline{dc}\ \overline{s}, \mathsf{datatype}\ x) & = & (\overline{y}, \overline{dc}) \\
\mathsf{proj}(M, \mathsf{datatype}\ x = \mathsf{datatype}\ M'.z\ \overline{s}, \mathsf{con}\ x) & = & \mathsf{proj}(M', \overline{s'}, \mathsf{datatype}\ z)\ (\text{where } \Gamma \vdash M' : \mathsf{sig}\ \overline{s'}\ \mathsf{end}) \\
 & & \\
\mathsf{proj}(M, \mathsf{val}\ x : \tau\ \overline{s}, \mathsf{val}\ x) & = & \tau \\
\mathsf{proj}(M, \mathsf{datatype}\ x\ \overline{y} = \overline{dc}\ \overline{s}, \mathsf{val}\ X) & = & \overline{y ::: \mathsf{Type}} \to M.x\ \overline{y}\ (\text{where } X \in \overline{dc}) \\
\mathsf{proj}(M, \mathsf{datatype}\ x\ \overline{y} = \overline{dc}\ \overline{s}, \mathsf{val}\ X) & = & \overline{y ::: \mathsf{Type}} \to \tau \to M.x\ \overline{y}\ (\text{where } X \text{ of } \tau \in \overline{dc}) \\
\mathsf{proj}(M, \mathsf{datatype}\ x = \mathsf{datatype}\ M'.z, \mathsf{val}\ X) & = & \overline{y ::: \mathsf{Type}} \to M.x\ \overline{y}\ (\text{where } \Gamma \vdash M' : \mathsf{sig}\ \overline{s'}\ \mathsf{end} \\
 & & \text{and } \mathsf{proj}(M', \overline{s'}, \mathsf{datatype}\ z = (\overline{y}, \overline{dc}) \text{ and } X \in \overline{dc}) \\
\mathsf{proj}(M, \mathsf{datatype}\ x = \mathsf{datatype}\ M'.z, \mathsf{val}\ X) & = & \overline{y ::: \mathsf{Type}} \to \tau \to M.x\ \overline{y}\ (\text{where } \Gamma \vdash M' : \mathsf{sig}\ \overline{s'}\ \mathsf{end} \\
 & & \text{and } \mathsf{proj}(M', \overline{s'}, \mathsf{datatype}\ z = (\overline{y}, \overline{dc}) \text{ and } X \text{ of } \tau \in \overline{dc}) \\
 & & \\
\mathsf{proj}(M, \mathsf{structure}\ X : S\ \overline{s}, \mathsf{structure}\ X) & = & S \\
 & & \\
\mathsf{proj}(M, \mathsf{signature}\ X = S\ \overline{s}, \mathsf{signature}\ X) & = & S \\
 & & \\
\mathsf{proj}(M, \mathsf{con}\ x :: \kappa\ \overline{s}, V) & = & [x \mapsto M.x]\mathsf{proj}(M, \overline{s}, V) \\
\mathsf{proj}(M, \mathsf{con}\ x :: \kappa = c\ \overline{s}, V) & = & [x \mapsto M.x]\mathsf{proj}(M, \overline{s}, V) \\
\mathsf{proj}(M, \mathsf{datatype}\ x\ \overline{y} = \overline{dc}\ \overline{s}, V) & = & [x \mapsto M.x]\mathsf{proj}(M, \overline{s}, V) \\
\mathsf{proj}(M, \mathsf{datatype}\ x = \mathsf{datatype}\ M'.z\ \overline{s}, V) & = & [x \mapsto M.x]\mathsf{proj}(M, \overline{s}, V) \\
\mathsf{proj}(M, \mathsf{val}\ x : \tau\ \overline{s}, V) & = & \mathsf{proj}(M, \overline{s}, V) \\
\mathsf{proj}(M, \mathsf{structure}\ X : S\ \overline{s}, V) & = & [X \mapsto M.X]\mathsf{proj}(M, \overline{s}, V) \\
\mathsf{proj}(M, \mathsf{signature}\ X = S\ \overline{s}, V) & = & [X \mapsto M.X]\mathsf{proj}(M, \overline{s}, V) \\
\mathsf{proj}(M, \mathsf{include}\ S\ \overline{s}, V) & = & \mathsf{proj}(M, \overline{s'}\ \overline{s}, V)\ (\text{where } \Gamma \vdash S \equiv \mathsf{sig}\ \overline{s'}\ \mathsf{end}) \\
\mathsf{proj}(M, \mathsf{constraint}\ c_1 \sim c_2\ \overline{s}, V) & = & \mathsf{proj}(M, \overline{s}, V) \\
\mathsf{proj}(M, \mathsf{class}\ x :: \kappa\ \overline{s}, V) & = & [x \mapsto M.x]\mathsf{proj}(M, \overline{s}, V) \\
\mathsf{proj}(M, \mathsf{class}\ x :: \kappa = c\ \overline{s}, V) & = & [x \mapsto M.x]\mathsf{proj}(M, \overline{s}, V)
\end{array}
$$

6 Type Inference

The Ur/Web compiler uses *heuristic type inference*, with no claims of completeness with respect to the declarative specification of the last section. The rules in use seem to work well in practice. This section

summarizes those rules, to help Ur programmers predict what will work and what won't.

6.1 Basic Unification

Type-checkers for languages based on the Hindley-Milner type discipline, like ML and Haskell, take advantage of *principal typing* properties, making complete type inference relatively straightforward. Inference algorithms are traditionally implemented using type unification variables, at various points asserting equalities between types, in the process discovering the values of type variables. The Ur/Web compiler uses the same basic strategy, but the complexity of the type system rules out easy completeness.

Type-checking can require evaluating recursive functional programs, thanks to the type-level map operator. When a unification variable appears in such a type, the next step of computation can be undetermined. The value of that variable might be determined later, but this would be "too late" for the unification problems generated at the first occurrence. This is the essential source of incompleteness.

Nonetheless, the unification engine tends to do reasonably well. Unlike in ML, polymorphism is never inferred in definitions; it must be indicated explicitly by writing out constructor-level parameters. By writing these and other annotations, the programmer can generally get the type inference engine to do most of the type reconstruction work.

6.2 Unifying Record Types

The type inference engine tries to take advantage of the algebraic rules governing type-level records, as shown in Section 5.4. When two constructors of record kind are unified, they are reduced to normal forms, with like terms crossed off from each normal form until, hopefully, nothing remains. This cannot be complete, with the inclusion of unification variables. The type-checker can help you understand what goes wrong when the process fails, as it outputs the unmatched remainders of the two normal forms.

6.3 Constructor Classes

Ur includes a constructor class facility inspired by Haskell's. The current version is experimental, with very general Prolog-like facilities that can lead to compile-time non-termination.

Constructor classes are integrated with the module system. A constructor class of kind κ is just a constructor of kind κ. By marking such a constructor c as a constructor class, the programmer instructs the type inference engine to, in each scope, record all values of types $c\ c_1\ \ldots\ c_n$ as *instances*. Any function argument whose type is of such a form is treated as implicit, to be determined by examining the current instance database. Any suitably kinded constructor within a module may be exposed as a constructor class from outside the module, simply by using a class signature item instead of a con signature item in the module's signature.

The "dictionary encoding" often used in Haskell implementations is made explicit in Ur. Constructor class instances are just properly typed values, and they can also be considered as "proofs" of membership in the class. In some cases, it is useful to pass these proofs around explicitly. An underscore written where a proof is expected will also be inferred, if possible, from the current instance database.

Just as for constructors, constructors classes may be exported from modules, and they may be exported as concrete or abstract. Concrete constructor classes have their "real" definitions exposed, so that client code may add new instances freely. Automatic inference of concrete class instances will not generally work, so abstract classes are almost always the right choice. They are useful as "predicates" that can be used to enforce invariants, as we will see in some definitions of SQL syntax in the Ur/Web standard library. Free extension of a concrete class is easily supported by exporting a constructor function from a module, since the class implementation will be concrete within the module.

6.4 Reverse-Engineering Record Types

It's useful to write Ur functions and functors that take record constructors as inputs, but these constructors can grow quite long, even though their values are often implied by other arguments. The compiler uses a simple heuristic to infer the values of unification variables that are mapped over, yielding known results. If the result is empty, we're done; if it's not empty, we replace a single unification variable with a new constructor formed from three new unification variables, as in $[\alpha = \beta] ++ \gamma$. This process can often be repeated to determine a unification variable fully.

6.5 Implicit Arguments in Functor Applications

Constructor, constraint, and constructor class witness members of structures may be omitted, when those structures are used in contexts where their assigned signatures imply how to fill in those missing members. This feature combines well with reverse-engineering to allow for uses of complicated meta-programming functors with little more code than would be necessary to invoke an untyped, ad-hoc code generator.

7 The Ur Standard Library

The built-in parts of the Ur/Web standard library are described by the signature in `lib/basis.urs` in the distribution. A module Basis ascribing to that signature is available in the initial environment, and every program is implicitly prefixed by open Basis.

Additionally, other common functions that are definable within Ur are included in `lib/top.urs` and `lib/top.ur`. This Top module is also opened implicitly.

The idea behind Ur is to serve as the ideal host for embedded domain-specific languages. For now, however, the "generic" functionality is intermixed with Ur/Web-specific functionality, including in these two library modules. We hope that these generic library components have types that speak for themselves. The next section introduces the Ur/Web-specific elements. Here, we only give the type declarations from the beginning of Basis.

```
type int
type float
type char
type string
type time
type blob

type unit = {}

datatype bool = False | True

datatype option t = None | Some of t

datatype list t = Nil | Cons of t × list t
```

The only unusual element of this list is the blob type, which stands for binary sequences. Simple blobs can be created from strings via Basis.textBlob. Blobs will also be generated from HTTP file uploads.

Ur also supports *polymorphic variants*, a dual to extensible records that has been popularized by OCaml. A type variant r represents an n-ary sum type, with one constructor for each field of record r. Each constructor c takes an argument of type $r.c$; the type {} can be used to "simulate" a nullary constructor. The `make` function builds a variant value, while `match` implements pattern-matching, with match cases represented

as records of functions.

```
con variant :: {Type} → Type
val make : nm :: Name → t ::: Type → ts ::: {Type} → [[nm] ~ ts] ⇒ t → variant ([nm = t] ++ ts)
val match : ts ::: {Type} → t ::: Type → variant ts → $(map (λt' ⇒ t' → t) ts) → t
```

Another important generic Ur element comes at the beginning of `top.urs`.

```
con folder :: K ⟶ {K} → Type

val fold : K ⟶ tf :: ({K} → Type)
  → (nm :: Name → v :: K → r :: {K} → [[nm] ~ r] ⇒
    tf r → tf ([nm = v] ++ r))
  → tf []
  → r :: {K} → folder r → tf r
```

For a type-level record r, a folder r encodes a permutation of r's elements. The fold function can be called on a folder to iterate over the elements of r in that order. fold is parameterized on a type-level function to be used to calculate the type of each intermediate result of folding. After processing a subset r' of r's entries, the type of the accumulator should be tf r'. The next two expression arguments to fold are the usual step function and initial accumulator, familiar from fold functions over lists. The final two arguments are the record to fold over and a folder for it.

The Ur compiler treats folder like a constructor class, using built-in rules to infer folders for records with known structure. The order in which field names are mentioned in source code is used as a hint about the permutation that the programmer would like.

8 The Ur/Web Standard Library

Some operations are only allowed in server-side code or only in client-side code. The type system does not enforce such restrictions, but the compiler enforces them in the process of whole-program compilation. In the discussion below, we note when a set of operations has a location restriction.

8.1 Monads

The Ur Basis defines the monad constructor class from Haskell.

```
class monad :: Type → Type
val return : m ::: (Type → Type) → t ::: Type
  → monad m
  → t → m t
val bind : m ::: (Type → Type) → t1 ::: Type → t2 ::: Type
  → monad m
  → m t1 → (t1 → m t2)
  → m t2
val mkMonad : m ::: (Type → Type)
  → {Return : t ::: Type → t → m t,
     Bind : t1 ::: Type → t2 ::: Type → m t1 → (t1 → m t2) → m t2}
  → monad m
```

The Ur/Web compiler provides syntactic sugar for monads, similar to Haskell's `do` notation. An expression $x \leftarrow e_1; e_2$ is desugared to bind e_1 $(\lambda x \Rightarrow e_2)$, and an expression $e_1; e_2$ is desugared to bind e_1 $(\lambda() \Rightarrow e_2)$. Note a difference from Haskell: as the $e_1; e_2$ case desugaring involves a function with () as its formal argument, the type of e_1 must be of the form m {}, rather than some arbitrary m t.

8.2 Transactions

Ur is a pure language; we use Haskell's trick to support controlled side effects. The standard library defines a monad transaction, meant to stand for actions that may be undone cleanly. By design, no other kinds of actions are supported.

```
con transaction :: Type → Type
val transaction_monad : monad transaction
```

For debugging purposes, a transactional function is provided for outputting a string on the server process' `stderr`.

```
val debug : string → transaction unit
```

8.3 HTTP

There are transactions for reading an HTTP header by name and for getting and setting strongly typed cookies. Cookies may only be created by the cookie declaration form, ensuring that they be named consistently based on module structure. For now, cookie operations are server-side only.

```
con http_cookie :: Type → Type
val getCookie : t ::: Type → http_cookie t → transaction (option t)
val setCookie : t ::: Type → http_cookie t → {Value : t, Expires : option time, Secure : bool} → transaction unit
val clearCookie : t ::: Type → http_cookie t → transaction unit
```

There are also an abstract url type and functions for converting to it, based on the policy defined by `[allow|deny] url` directives in the project file.

```
type url
val bless : string → url
val checkUrl : string → option url
```

bless raises a runtime error if the string passed to it fails the URL policy.

It is possible to grab the current page's URL or to build a URL for an arbitrary transaction that would also be an acceptable value of a `link` attribute of the `a` tag. These are server-side operations.

```
val currentUrl : transaction url
val url : transaction page → url
```

Page generation may be interrupted at any time with a request to redirect to a particular URL instead.

```
val redirect : t ::: Type → url → transaction t
```

It's possible for pages to return files of arbitrary MIME types. A file can be input from the user using this data type, along with the upload form tag. These functions and those described in the following paragraph are server-side.

```
type file
val fileName : file → option string
val fileMimeType : file → string
val fileData : file → blob
```

It is also possible to get HTTP request headers and environment variables, and set HTTP response headers, using abstract types similar to the one for URLs.

```
type requestHeader
val blessRequestHeader : string → requestHeader
val checkRequestHeader : string → option requestHeader
val getHeader : requestHeader → transaction (option string)

type envVar
val blessEnvVar : string → envVar
val checkEnvVar : string → option envVar
val getenv : envVar → transaction (option string)

type responseHeader
val blessResponseHeader : string → responseHeader
val checkResponseHeader : string → option responseHeader
val setHeader : responseHeader → string → transaction unit
```

A blob can be extracted from a file and returned as the page result. There are bless and check functions for MIME types analogous to those for URLs.

```
type mimeType
val blessMime : string → mimeType
val checkMime : string → option mimeType
val returnBlob : t ::: Type → blob → mimeType → transaction t
```

8.4 SQL

Everything about SQL database access is restricted to server-side code.

The fundamental unit of interest in the embedding of SQL is tables, described by a type family and creatable only via the table declaration form.

```
con sql_table :: {Type} → {{Unit}} → Type
```

The first argument to this constructor gives the names and types of a table's columns, and the second argument gives the set of valid keys. Keys are the only subsets of the columns that may be referenced as foreign keys. Each key has a name.

We also have the simpler type family of SQL views, which have no keys.

```
con sql_view :: {Type} → Type
```

A multi-parameter type class is used to allow tables and views to be used interchangeably, with a way of extracting the set of columns from each.

```
class fieldsOf :: Type → {Type} → Type
val fieldsOf_table : fs ::: {Type} → keys ::: {{Unit}} → fieldsOf (sql_table fs keys) fs
val fieldsOf_view : fs ::: {Type} → fieldsOf (sql_view fs) fs
```

8.4.1 Table Constraints

Tables may be declared with constraints, such that database modifications that violate the constraints are blocked. A table may have at most one `PRIMARY KEY` constraint, which gives the subset of columns that will most often be used to look up individual rows in the table.

```
con primary_key :: {Type} → {{Unit}} → Type
val no_primary_key : fs ::: {Type} → primary_key fs []
val primary_key : rest ::: {Type} → t ::: Type → key1 :: Name → keys :: {Type}
  → [[key1] ~ keys] ⇒ [[key1 = t] ++ keys ~ rest]
  ⇒ $([key1 = sql_injectable_prim t] ++ map sql_injectable_prim keys)
  → primary_key ([key1 = t] ++ keys ++ rest) [Pkey = [key1] ++ map (λ_ ⇒ ()) keys]
```

The type class sql_injectable_prim characterizes which types are allowed in SQL and are not option types. In SQL, a `PRIMARY KEY` constraint enforces after-the-fact that a column may not contain `NULL`s, but Ur/Web forces that information to be included in table types from the beginning. Thus, the only effect of this kind of constraint in Ur/Web is to enforce uniqueness of the given key within the table.

A type family stands for sets of named constraints of the remaining varieties.

```
con sql_constraints :: {Type} → {{Unit}} → Type
```

The first argument gives the column types of the table being constrained, and the second argument maps constraint names to the keys that they define. Constraints that don't define keys are mapped to "empty keys."

There is a type family of individual, unnamed constraints.

```
con sql_constraint :: {Type} → {Unit} → Type
```

The first argument is the same as above, and the second argument gives the key columns for just this constraint.

We have operations for assembling constraints into constraint sets.

```
val no_constraint : fs ::: {Type} → sql_constraints fs []
val one_constraint : fs ::: {Type} → unique ::: {Unit} → name :: Name
  → sql_constraint fs unique → sql_constraints fs [name = unique]
val join_constraints : fs ::: {Type} → uniques1 ::: {{Unit}} → uniques2 ::: {{Unit}} → [uniques1 ~ uniques2]
  ⇒ sql_constraints fs uniques1 → sql_constraints fs uniques2 → sql_constraints fs (uniques1 ++ uniques2)
```

A `UNIQUE` constraint forces a set of columns to be a key, which means that no combination of column values may occur more than once in the table. The unique1 and unique arguments are separated out only to ensure that empty `UNIQUE` constraints are rejected.

```
val unique : rest ::: {Type} → t ::: Type → unique1 :: Name → unique :: {Type}
  → [[unique1] ~ unique] ⇒ [[unique1 = t] ++ unique ~ rest]
  ⇒ sql_constraint ([unique1 = t] ++ unique ++ rest) ([unique1] ++ map (λ_ ⇒ ()) unique)
```

A `FOREIGN KEY` constraint connects a set of local columns to a local or remote key, enforcing that the local columns always reference an existent row of the foreign key's table. A local column of type t may be linked to a foreign column of type option t, and vice versa. We formalize that notion with a type class.

```
class linkable :: Type → Type → Type
val linkable_same : t ::: Type → linkable t t
val linkable_from_nullable : t ::: Type → linkable (option t) t
val linkable_to_nullable : t ::: Type → linkable t (option t)
```

The matching type family uses linkable to define when two keys match up type-wise.

```
con matching :: {Type} → {Type} → Type
val mat_nil : matching [] []
val mat_cons : t1 ::: Type → rest1 ::: {Type} → t2 ::: Type → rest2 ::: {Type} → nm1 :: Name → nm2 :: Name
  → [[nm1] ~ rest1] ⇒ [[nm2] ~ rest2] ⇒ linkable t1 t2 → matching rest1 rest2
  → matching ([nm1 = t1] ++ rest1) ([nm2 = t2] ++ rest2)
```

SQL provides a number of different propagation modes for `FOREIGN KEY` constraints, governing what happens when a row containing a still-referenced foreign key value is deleted or modified to have a different key value. The argument of a propagation mode's type gives the local key type.

```
con propagation_mode :: {Type} → Type
val restrict : fs ::: {Type} → propagation_mode fs
val cascade : fs ::: {Type} → propagation_mode fs
val no_action : fs ::: {Type} → propagation_mode fs
val set_null : fs ::: {Type} → propagation_mode (map option fs)
```

Finally, we put these ingredient together to define the `FOREIGN KEY` constraint function.

```
val foreign_key : mine1 ::: Name → t ::: Type → mine ::: {Type} → munused ::: {Type} → foreign ::: {Type}
  → funused ::: {Type} → nm ::: Name → uniques ::: {{Unit}}
  → [[mine1] ~ mine] ⇒ [[mine1 = t] ++ mine ~ munused] ⇒ [foreign ~ funused] ⇒ [[nm] ~ uniques]
  ⇒ matching ([mine1 = t] ++ mine) foreign
  → sql_table (foreign ++ funused) ([nm = map (λ_ ⇒ ()) foreign] ++ uniques)
  → {OnDelete : propagation_mode ([mine1 = t] ++ mine),
     OnUpdate : propagation_mode ([mine1 = t] ++ mine)}
  → sql_constraint ([mine1 = t] ++ mine ++ munused) []
```

The last kind of constraint is a `CHECK` constraint, which attaches a boolean invariant over a row's contents. It is defined using the sql_exp type family, which we discuss in more detail below.

```
val check : fs ::: {Type} → sql_exp [] [] fs bool → sql_constraint fs []
```

Section 9.1.1 shows the expanded syntax of the table declaration and signature item that includes constraints. There is no other way to use constraints with SQL in Ur/Web.

8.4.2 Queries

A final query is constructed via the sql_query function. Constructor arguments respectively specify the unrestricted free table variables (which will only be available in subqueries), the free table variables that may only be mentioned within arguments to aggregate functions, table fields we select (as records mapping tables to the subsets of their fields that we choose), and the (always named) extra expressions that we select.

```
con sql_query :: {{Type}} → {{Type}} → {{Type}} → {Type} → Type
val sql_query : free ::: {{Type}}
  → afree ::: {{Type}}
  → tables ::: {{Type}}
  → selectedFields ::: {{Type}}
  → selectedExps ::: {Type}
  → [free ~ tables]
  ⇒ {Rows : sql_query1 free afree tables selectedFields selectedExps,
     OrderBy : sql_order_by (free ++ tables) selectedExps,
     Limit : sql_limit,
     Offset : sql_offset}
  → sql_query free afree selectedFields selectedExps
```

Queries are used by folding over their results inside transactions.

```
val query : tables ::: {{Type}} → exps ::: {Type} → [tables ~ exps] ⇒ state ::: Type → sql_query [] [] tables exps
  → ($(exps ++ map (λfields :: {Type} ⇒ $fields) tables)
    → state → transaction state)
  → state → transaction state
```

Most of the complexity of the query encoding is in the type sql_query1, which includes simple queries and derived queries based on relational operators. Constructor arguments respectively specify the unrestricted free table veriables, the aggregate-only free table variables, the tables we select from, the subset of fields that we keep from each table for the result rows, and the extra expressions that we select.

```
con sql_query1 :: {{Type}} → {{Type}} → {{Type}} → {{Type}} → {Type} → Type

type sql_relop
val sql_union : sql_relop
val sql_intersect : sql_relop
val sql_except : sql_relop
val sql_relop : free ::: {{Type}}
  → afree ::: {{Type}}
  → tables1 ::: {{Type}}
  → tables2 ::: {{Type}}
  → selectedFields ::: {{Type}}
  → selectedExps ::: {Type}
  → sql_relop
  → bool (* ALL *)
  → sql_query1 free afree tables1 selectedFields selectedExps
  → sql_query1 free afree tables2 selectedFields selectedExps
  → sql_query1 free afree selectedFields selectedFields selectedExps

val sql_query1 : free ::: {{Type}}
  → afree ::: {{Type}}
  → tables ::: {{Type}}
  → grouped ::: {{Type}}
  → selectedFields ::: {{Type}}
  → selectedExps ::: {Type}
  → empties :: {Unit}
  → [free ~ tables]
  ⇒ [free ~ grouped]
  ⇒ [afree ~ tables]
  ⇒ [empties ~ selectedFields]
  ⇒ {Distinct : bool,
    From : sql_from_items free tables,
    Where : sql_exp (free ++ tables) afree [] bool,
    GroupBy : sql_subset tables grouped,
    Having : sql_exp (free ++ grouped) (afree ++ tables) [] bool,
    SelectFields : sql_subset grouped (map (λ_ ⇒ []) empties ++ selectedFields),
    SelectExps : $(map (sql_expw (free ++ grouped) (afree ++ tables) []) selectedExps)}
  → sql_query1 free afree tables selectedFields selectedExps
```

To encode projection of subsets of fields in SELECT clauses, and to encode GROUP BY clauses, we rely on a type family sql_subset, capturing what it means for one record of table fields to be a subset of another. The main constructor sql_subset "proves subset facts" by requiring a split of a record into kept and dropped parts. The extra constructor sql_subset_all is a convenience for keeping all fields of a record.

```
con sql_subset :: {{Type}} → {{Type}} → Type
val sql_subset : keep_drop :: {({Type} × {Type})}
  → sql_subset
    (map (λfields :: ({Type} × {Type}) ⇒ fields.1 ++ fields.2) keep_drop)
    (map (λfields :: ({Type} × {Type}) ⇒ fields.1) keep_drop)
val sql_subset_all : tables :: {{Type}} → sql_subset tables tables
```

SQL expressions are used in several places, including SELECT, WHERE, HAVING, and ORDER BY clauses. They reify a fragment of the standard SQL expression language, while making it possible to inject "native" Ur values in some places. The arguments to the sql_exp type family respectively give the unrestricted-availability table fields, the table fields that may only be used in arguments to aggregate functions, the available selected expressions, and the type of the expression.

```
con sql_exp :: {{Type}} → {{Type}} → {Type} → Type → Type
```

Any field in scope may be converted to an expression.

```
val sql_field : otherTabs ::: {{Type}} → otherFields ::: {Type}
  → fieldType ::: Type → agg ::: {{Type}}
  → exps ::: {Type}
  → tab :: Name → field :: Name
  → sql_exp ([tab = [field = fieldType] ++ otherFields] ++ otherTabs) agg exps fieldType
```

There is an analogous function for referencing named expressions.

```
val sql_exp : tabs ::: {{Type}} → agg ::: {{Type}} → t ::: Type → rest ::: {Type} → nm :: Name
  → sql_exp tabs agg ([nm = t] ++ rest) t
```

Ur values of appropriate types may be injected into SQL expressions.

```
class sql_injectable_prim
val sql_bool : sql_injectable_prim bool
val sql_int : sql_injectable_prim int
val sql_float : sql_injectable_prim float
val sql_string : sql_injectable_prim string
val sql_time : sql_injectable_prim time
val sql_blob : sql_injectable_prim blob
val sql_channel : t ::: Type → sql_injectable_prim (channel t)
val sql_client : sql_injectable_prim client

class sql_injectable
val sql_prim : t ::: Type → sql_injectable_prim t → sql_injectable t
val sql_option_prim : t ::: Type → sql_injectable_prim t → sql_injectable (option t)

val sql_inject : tables ::: {{Type}} → agg ::: {{Type}} → exps ::: {Type} → t ::: Type → sql_injectable t
  → t → sql_exp tables agg exps t
```

Additionally, most function-free types may be injected safely, via the serialized type family.

```
con serialized :: Type → Type
val serialize : t ::: Type → t → serialized t
val deserialize : t ::: Type → serialized t → t
val sql_serialized : t ::: Type → sql_injectable_prim (serialized t)
```

We have the SQL nullness test, which is necessary because of the strange SQL semantics of equality in the presence of null values.

```
val sql_is_null : tables ::: {{Type}} → agg ::: {{Type}} → exps ::: {Type} → t ::: Type
  → sql_exp tables agg exps (option t) → sql_exp tables agg exps bool
```

As another way of dealing with null values, there is also a restricted form of the standard `COALESCE` function.

```
val sql_coalesce : tables ::: {{Type}} → agg ::: {{Type}} → exps ::: {Type}
  → t ::: Type
  → sql_exp tables agg exps (option t)
  → sql_exp tables agg exps t
  → sql_exp tables agg exps t
```

We have generic nullary, unary, and binary operators.

```
con sql_nfunc :: Type → Type
val sql_current_timestamp : sql_nfunc time
val sql_nfunc : tables ::: {{Type}} → agg ::: {{Type}} → exps ::: {Type} → t ::: Type
  → sql_nfunc t → sql_exp tables agg exps t
```

```
con sql_unary :: Type → Type → Type
val sql_not : sql_unary bool bool
val sql_unary : tables ::: {{Type}} → agg ::: {{Type}} → exps ::: {Type} → arg ::: Type → res ::: Type
  → sql_unary arg res → sql_exp tables agg exps arg → sql_exp tables agg exps res
```

```
con sql_binary :: Type → Type → Type → Type
val sql_and : sql_binary bool bool bool
val sql_or : sql_binary bool bool bool
val sql_binary : tables ::: {{Type}} → agg ::: {{Type}} → exps ::: {Type} → arg1 ::: Type → arg2 ::: Type → res ::: Type
  → sql_binary arg1 arg2 res → sql_exp tables agg exps arg1 → sql_exp tables agg exps arg2 → sql_exp tables agg exps res
```

```
class sql_arith
val sql_int_arith : sql_arith int
val sql_float_arith : sql_arith float
val sql_neg : t ::: Type → sql_arith t → sql_unary t t
val sql_plus : t ::: Type → sql_arith t → sql_binary t t t
val sql_minus : t ::: Type → sql_arith t → sql_binary t t t
val sql_times : t ::: Type → sql_arith t → sql_binary t t t
val sql_div : t ::: Type → sql_arith t → sql_binary t t t
val sql_mod : sql_binary int int int
```

Finally, we have aggregate functions. The COUNT(∗) syntax is handled specially, since it takes no real argument. The other aggregate functions are placed into a general type family, using constructor classes to restrict usage to properly typed arguments. The key aspect of the sql_aggregate function's type is the shift of aggregate-function-only fields into unrestricted fields.

```
val sql_count : tables ::: {{Type}} → agg ::: {{Type}} → exps ::: {Type} → sql_exp tables agg exps int
```

```
con sql_aggregate :: Type → Type → Type
val sql_aggregate : tables ::: {{Type}} → agg ::: {{Type}} → exps ::: {Type} → dom ::: Type → ran ::: Type
  → sql_aggregate dom ran → sql_exp agg agg exps dom → sql_exp tables agg exps ran
```

```
val sql_count_col : t ::: Type → sql_aggregate (option t) int
```

Most aggregate functions are typed using a two-parameter constructor class nullify which maps option types to themselves and adds option to others. That is, this constructor class represents the process of making an SQL type "nullable."

```
class sql_summable
val sql_summable_int : sql_summable int
val sql_summable_float : sql_summable float
val sql_avg : t ::: Type → sql_summable t → sql_aggregate t (option float)
val sql_sum : t ::: Type → nt ::: Type → sql_summable t → nullify t nt → sql_aggregate t nt
```

```
class sql_maxable
val sql_maxable_int : sql_maxable int
val sql_maxable_float : sql_maxable float
val sql_maxable_string : sql_maxable string
val sql_maxable_time : sql_maxable time
val sql_max : t ::: Type → nt ::: Type → sql_maxable t → nullify t nt → sql_aggregate t nt
val sql_min : t ::: Type → nt ::: Type → sql_maxable t → nullify t nt → sql_aggregate t nt
```

Any SQL query that returns single columns may be turned into a subquery expression.

```
val sql_subquery : tables ::: {{Type}} → agg ::: {{Type}} → exps ::: {Type} → nm ::: Name → t ::: Type → nt ::: Type
  → nullify t nt → sql_query tables agg [nm = t] → sql_exp tables agg exps nt
```

There is also an `IF..THEN..ELSE..` construct that is compiled into standard SQL `CASE` expressions.

```
val sql_if_then_else : tables ::: {{Type}} → agg ::: {{Type}} → exps ::: {Type} → t ::: Type
  → sql_exp tables agg exps bool
  → sql_exp tables agg exps t
  → sql_exp tables agg exps t
  → sql_exp tables agg exps t
```

`FROM` clauses are specified using a type family, whose arguments are the free table variables and the table variables bound by this clause.

```
con sql_from_items :: {{Type}} → {{Type}} → Type
val sql_from_table : free ::: {{Type}}
  → t ::: Type → fs ::: {Type} → fieldsOf t fs → name :: Name → t → sql_from_items free [name = fs]
val sql_from_query : free ::: {{Type}} → fs ::: {Type} → name :: Name → sql_query free [] fs → sql_from_items free [name = fs]
val sql_from_comma : free ::: tabs1 ::: {{Type}} → tabs2 ::: {{Type}} → [tabs1 ~ tabs2]
  ⇒ sql_from_items free tabs1 → sql_from_items free tabs2
  → sql_from_items free (tabs1 ++ tabs2)
val sql_inner_join : free ::: {{Type}} → tabs1 ::: {{Type}} → tabs2 ::: {{Type}}
  → [free ~ tabs1] ⇒ [free ~ tabs2] ⇒ [tabs1 ~ tabs2]
  ⇒ sql_from_items free tabs1 → sql_from_items free tabs2
  → sql_exp (free ++ tabs1 ++ tabs2) [] [] bool
  → sql_from_items free (tabs1 ++ tabs2)
```

Besides these basic cases, outer joins are supported, which requires a type class for turning non-option columns into option columns.

```
class nullify :: Type → Type → Type
val nullify_option : t ::: Type → nullify (option t) (option t)
val nullify_prim : t ::: Type → sql_injectable_prim t → nullify t (option t)
```

Left, right, and full outer joins can now be expressed using functions that accept records of nullify instances. Here, we give only the type for a left join as an example.

```
val sql_left_join : free ::: {{Type}} → tabs1 ::: {{Type}} → tabs2 ::: {{(Type × Type)}}
  → [free ~ tabs1] ⇒ [free ~ tabs2] ⇒ [tabs1 ~ tabs2]
  ⇒ $(map (λr ⇒ $(map (λp :: (Type × Type) ⇒ nullify p.1 p.2) r)) tabs2)
  → sql_from_items free tabs1 → sql_from_items free (map (map (λp :: (Type × Type) ⇒ p.1)) tabs2)
  → sql_exp (free ++ tabs1 ++ map (map (λp :: (Type × Type) ⇒ p.1)) tabs2) [] [] bool
  → sql_from_items free (tabs1 ++ map (map (λp :: (Type × Type) ⇒ p.2)) tabs2)
```

We wrap up the definition of query syntax with the types used in representing ORDER BY, LIMIT, and OFFSET clauses.

```
type sql_direction
val sql_asc : sql_direction
val sql_desc : sql_direction

con sql_order_by :: {{Type}} → {Type} → Type
val sql_order_by_Nil : tables ::: {{Type}} → exps :: {Type} → sql_order_by tables exps
val sql_order_by_Cons : tf ::: ({{Type}} → {{Type}} → {Type} → Type → Type)
  → tables ::: {{Type}} → exps ::: {Type} → t ::: Type
  → sql_window tf → tf tables [] exps t → sql_direction → sql_order_by tables exps → sql_order_by tables exps
val sql_order_by_random : tables ::: {{Type}} → exps ::: {Type} → sql_order_by tables exps

type sql_limit
val sql_no_limit : sql_limit
val sql_limit : int → sql_limit

type sql_offset
val sql_no_offset : sql_offset
val sql_offset : int → sql_offset
```

When using Postgres, `SELECT` and `ORDER BY` are allowed to contain top-level uses of *window functions*. A separate type family `sql_expw` is provided for such cases, with some type class convenience for overloading

between normal and window expressions.

```
con sql_expw :: {{Type}} → {{Type}} → {Type} → Type → Type

class sql_window :: ({{Type}} → {{Type}} → {Type} → Type → Type) → Type
val sql_window_normal : sql_window sql_exp
val sql_window_fancy : sql_window sql_expw
val sql_window : tf ::: ({{Type}} → {{Type}} → {Type} → Type → Type)
  → tables ::: {{Type}} → agg ::: {{Type}} → exps ::: {Type} → t ::: Type
  → sql_window tf
  → tf tables agg exps t
  → sql_expw tables agg exps t

con sql_partition :: {{Type}} → {{Type}} → {Type} → Type
val sql_no_partition : tables ::: {{Type}} → agg ::: {{Type}} → exps ::: {Type}
  → sql_partition tables agg exps
val sql_partition : tables ::: {{Type}} → agg ::: {{Type}} → exps ::: {Type} → t ::: Type
  → sql_exp tables agg exps t
  → sql_partition tables agg exps

con sql_window_function :: {{Type}} → {{Type}} → {Type} → Type → Type
val sql_window_function : tables ::: {{Type}} → agg ::: {{Type}} → exps ::: {Type}
  → t ::: Type
  → sql_window_function tables agg exps t
  → sql_partition tables agg exps
  → sql_order_by tables exps
  → sql_expw tables agg exps t

val sql_window_aggregate : tables ::: {{Type}} → agg ::: {{Type}} → exps ::: {Type}
  → t ::: Type → nt ::: Type
  → sql_aggregate t nt
  → sql_exp tables agg exps t
  → sql_window_function tables agg exps nt
val sql_window_count : tables ::: {{Type}} → agg ::: {{Type}} → exps ::: {Type}
  → sql_window_function tables agg exps int
val sql_rank : tables ::: {{Type}} → agg ::: {{Type}} → exps ::: {Type}
  → sql_window_function tables agg exps int
```

8.4.3 DML

The Ur/Web library also includes an embedding of a fragment of SQL's DML, the Data Manipulation Language, for modifying database tables. Any piece of DML may be executed in a transaction.

```
type dml
val dml : dml → transaction unit
```

The function Basis.dml will trigger a fatal application error if the command fails, for instance, because a data integrity constraint is violated. An alternate function returns an error message as a string instead.

```
val tryDml : dml → transaction (option string)
```

Properly typed records may be used to form INSERT commands.

```
val insert : fields ::: {Type} → sql_table fields
  → $(map (sql_exp [] [] []) fields) → dml
```

An UPDATE command is formed from a choice of which table fields to leave alone and which to change, along with an expression to use to compute the new value of each changed field and a WHERE clause. Note that, in the table environment applied to expressions, the table being updated is hardcoded at the name T. The parsing extension for UPDATE will elaborate all table-free field references to use table variable T.

```
val update : unchanged ::: {Type} → changed :: {Type} → [changed ~ unchanged]
  ⇒ $(map (sql_exp [T = changed ++ unchanged] [] []) changed)
  → sql_table (changed ++ unchanged) → sql_exp [T = changed ++ unchanged] [] [] bool → dml
```

A DELETE command is formed from a table and a WHERE clause. The above use of T is repeated.

```
val delete : fields ::: {Type} → sql_table fields → sql_exp [T = fields] [] [] bool → dml
```

8.4.4 Sequences

SQL sequences are counters with concurrency control, often used to assign unique IDs. Ur/Web supports them via a simple interface. The only way to create a sequence is with the sequence declaration form.

```
type sql_sequence
val nextval : sql_sequence → transaction int
val setval : sql_sequence → int → transaction unit
```

8.5 XML

Ur/Web's library contains an encoding of XML syntax and semantic constraints. We make no effort to follow the standards governing XML schemas. Rather, XML fragments are viewed more as values of ML datatypes, and we only track which tags are allowed inside which other tags. The Ur/Web standard library encodes a very loose version of XHTML, where it is very easy to produce documents which are invalid XHTML, but which still display properly in all major browsers. The main purposes of the invariants that are enforced are first, to provide some documentation about the places where it would make sense to insert XML fragments; and second, to rule out code injection attacks and other abstraction violations related to HTML syntax.

The basic XML type family has arguments respectively indicating the *context* of a fragment, the fields that the fragment expects to be bound on entry (and their types), and the fields that the fragment will bind (and their types). Contexts are a record-based "poor man's subtyping" encoding, with each possible set of valid tags corresponding to a different context record. For instance, the context for the `<td>` tag is [Dyn, MakeForm, Tr], to indicate nesting inside a `<tr>` tag with the ability to nest `<form>` and `<dyn>` tags (see below). Contexts are maintained in a somewhat ad-hoc way; the only definitive reference for their meanings is the types of the tag values in `basis.urs`. The arguments dealing with field binding are only relevant to HTML forms.

```
con xml :: {Unit} → {Type} → {Type} → Type
```

We also have a type family of XML tags, indexed respectively by the record of optional attributes accepted by the tag, the context in which the tag may be placed, the context required of children of the tag, which form fields the tag uses, and which fields the tag defines.

```
con tag :: {Type} → {Unit} → {Unit} → {Type} → {Type} → Type
```

Literal text may be injected into XML as "CDATA."

```
val cdata : ctx ::: {Unit} → use ::: {Type} → string → xml ctx use []
```

There is also a function to insert the literal value of a character. Since Ur/Web uses the UTF-8 text encoding, the cdata function is only sufficient to encode characters with ASCII codes below 128. Higher

codes have alternate meanings in UTF-8 than in usual ASCII, so this alternate function should be used with them.

```
val cdataChar : ctx ::: {Unit} → use ::: {Type} → char → xml ctx use []
```

There is a function for producing an XML tree with a particular tag at its root.

```
val tag : attrsGiven ::: {Type} → attrsAbsent ::: {Type} → ctxOuter ::: {Unit} → ctxInner ::: {Unit}
  → useOuter ::: {Type} → useInner ::: {Type} → bindOuter ::: {Type} → bindInner ::: {Type}
  → [attrsGiven ~ attrsAbsent] ⇒ [useOuter ~ useInner] ⇒ [bindOuter ~ bindInner]
  ⇒ css_class
  → option (signal css_class)
  → css_style
  → option (signal css_style)
  → $attrsGiven
  → tag (attrsGiven ++ attrsAbsent) ctxOuter ctxInner useOuter bindOuter
  → xml ctxInner useInner bindInner → xml ctxOuter (useOuter ++ useInner) (bindOuter ++ bindInner)
```

Note that any tag may be assigned a CSS class, or left without a class by passing Basis.null as the first value-level argument. This is the sole way of making use of the values produced by style declarations. The function Basis.classes can be used to specify a list of CSS classes for a single tag. Stylesheets to assign properties to the classes can be linked via URL's with `link` tags. Ur/Web makes it easy to calculate upper bounds on usage of CSS classes through program analysis, with the `-css` command-line flag.

Also note that two different arguments are available for setting CSS classes: the first, associated with the `class` pseudo-attribute syntactic sugar, fixes the class of a tag for the duration of the tag's life; while the second, associated with the `dynClass` pseudo-attribute, allows the class to vary over the tag's life. See Section 8.6.3 for an introduction to the signal type family.

The third and fourth value-level arguments makes it possible to generate HTML `style` attributes, either with fixed content (`style` attribute) or dynamic content (`dynStyle` pseudo-attribute).

Two XML fragments may be concatenated.

```
val join : ctx ::: {Unit} → use1 ::: {Type} → bind1 ::: {Type} → bind2 ::: {Type}
  → [use1 ~ bind1] ⇒ [bind1 ~ bind2]
  ⇒ xml ctx use1 bind1 → xml ctx (use1 ++ bind1) bind2 → xml ctx use1 (bind1 ++ bind2)
```

Finally, any XML fragment may be updated to "claim" to use more form fields than it does.

```
val useMore : ctx ::: {Unit} → use1 ::: {Type} → use2 ::: {Type} → bind ::: {Type} → [use1 ~ use2]
  ⇒ xml ctx use1 bind → xml ctx (use1 ++ use2) bind
```

We will not list here the different HTML tags and related functions from the standard library. They should be easy enough to understand from the code in `basis.urs`. The set of tags in the library is not yet claimed to be complete for HTML standards. Also note that there is currently no way for the programmer to add his own tags. It *is* possible to add new tags directly to `basis.urs`, but this should only be done as a prelude to suggesting a patch to the main distribution.

One last useful function is for aborting any page generation, returning some XML as an error message. This function takes the place of some uses of a general exception mechanism.

```
val error : t ::: Type → xbody → t
```

8.6 Client-Side Programming

Ur/Web supports running code on web browsers, via automatic compilation to JavaScript.

8.6.1 The Basics

All of the functions in this subsection are client-side only.

Clients can open alert and confirm dialog boxes, in the usual annoying JavaScript way.

```
val alert : string → transaction unit
val confirm : string → transaction bool
```

Any transaction may be run in a new thread with the spawn function.

```
val spawn : transaction unit → transaction unit
```

The current thread can be paused for at least a specified number of milliseconds.

```
val sleep : int → transaction unit
```

A few functions are available to registers callbacks for particular error events. Respectively, they are triggered on calls to error, uncaught JavaScript exceptions, failure of remote procedure calls, the severance of the connection serving asynchronous messages, or the occurrence of some other error with that connection. If no handlers are registered for a kind of error, then a JavaScript `alert()` is used to announce its occurrence. When one of these functions is called multiple times within a single page, all registered handlers are run when appropriate events occur, with handlers run in the reverse of their registration order.

```
val onError : (xbody → transaction unit) → transaction unit
val onFail : (string → transaction unit) → transaction unit
val onConnectFail : transaction unit → transaction unit
val onDisconnect : transaction unit → transaction unit
val onServerError : (string → transaction unit) → transaction unit
```

There are also functions to register standard document-level event handlers.

```
val onClick : (mouseEvent → transaction unit) → transaction unit
val onDblclick : (mouseEvent → transaction unit) → transaction unit
val onKeydown : (keyEvent → transaction unit) → transaction unit
val onKeypress : (keyEvent → transaction unit) → transaction unit
val onKeyup : (keyEvent → transaction unit) → transaction unit
val onMousedown : (mouseEvent → transaction unit) → transaction unit
val onMouseup : (mouseEvent → transaction unit) → transaction unit
```

Versions of standard JavaScript functions are provided that event handlers may call to mask default handling or prevent bubbling of events up to parent DOM nodes, respectively.

```
val preventDefault : transaction unit
val stopPropagation : transaction unit
```

8.6.2 Node IDs

There is an abstract type of node IDs that may be assigned to `id` attributes of most HTML tags.

```
type id
val fresh : transaction id
```

The `fresh` function is allowed on both server and client, but there is no other way to create IDs, which includes lack of a way to force an ID to match a particular string. The main semantic importance of IDs within Ur/Web is in uses of the HTML `<label>` tag. IDs play a much more central role in mainstream JavaScript programming, but Ur/Web uses a very different model to enable changes to particular nodes of

a page tree, as the next manual subsection explains. IDs may still be useful in interfacing with JavaScript code (for instance, through Ur/Web's FFI).

One further use of IDs is as handles for requesting that *focus* be given to specific tags.

```
val giveFocus : id → transaction unit
```

8.6.3 Functional-Reactive Page Generation

Most approaches to "AJAX"-style coding involve imperative manipulation of the DOM tree representing an HTML document's structure. Ur/Web follows the *functional-reactive* approach instead. Programs may allocate mutable *sources* of arbitrary types, and an HTML page is effectively a pure function over the latest values of the sources. The page is not mutated directly, but rather it changes automatically as the sources are mutated.

More operationally, you can think of a source as a mutable cell with facilities for subscription to change notifications. That level of detail is hidden behind a monadic facility to be described below. First, there are three primitive operations for working with sources just as if they were ML `ref` cells, corresponding to ML's `ref`, `:=`, and `!` operations.

```
con source :: Type → Type
val source : t ::: Type → t → transaction (source t)
val set : t ::: Type → source t → t → transaction unit
val get : t ::: Type → source t → transaction t
```

Only source creation and setting are supported server-side, as a convenience to help in setting up a page, where you may wish to allocate many sources that will be referenced through the page. All server-side storage of values inside sources uses string serializations of values, while client-side storage uses normal JavaScript values.

Pure functions over arbitrary numbers of sources are represented in a monad of *signals*, which may only be used in client-side code. This is presented to the programmer in the form of a monad signal, each of whose values represents (conceptually) some pure function over all sources that may be allocated in the course of program execution. A monad operation signal denotes the identity function over a particular source. By using signal on a source, you implicitly subscribe to change notifications for that source. That is, your signal will automatically be recomputed as that source changes. The usual monad operators make it possible to build up complex signals that depend on multiple sources; automatic updating upon source-value changes still happens automatically. There is also an operator for computing a signal's current value within a transaction.

```
con signal :: Type → Type
val signal_monad : monad signal
val signal : t ::: Type → source t → signal t
val current : t ::: Type → signal t → transaction t
```

A reactive portion of an HTML page is injected with a dyn tag, which has a signal-valued attribute Signal.

```
val dyn : ctx ::: {Unit} → use ::: {Type} → bind ::: {Type} → [ctx ~ [Dyn]] ⇒ unit
  → tag [Signal = signal (xml ([Dyn] ++ ctx) use bind)] ([Dyn] ++ ctx) [] use bind
```

The semantics of `<dyn>` tags is somewhat subtle. When the signal associated with such a tag changes value, the associated subtree of the HTML page is recreated. Some properties of the subtree, such as attributes and client-side widget values, are specified explicitly in the signal value, so these may be counted on to remain the same after recreation. Other properties, like focus and cursor position within textboxes, are *not* specified by signal values, and these properties will be *reset* upon subtree regeneration. Furthermore, user interaction with widgets may not work properly during regeneration. For instance, clicking a button while it is being regenerated may not trigger its `onclick` event code.

Currently, the only way to avoid undesired resets is to avoid regeneration of containing subtrees. There are two main strategies for achieving that goal. First, when changes to a subtree can be confined to CSS classes of tags, the `dynClass` pseudo-attribute may be used instead (see Section 8.5), as it does not regenerate subtrees. Second, a single `<dyn>` tag may be broken into multiple tags, in a way that makes finer-grained dependency structure explicit. This latter strategy can avoid "spurious" regenerations that are not actually required to achieve the intended semantics.

Transactions can be run on the client by including them in attributes like the Onclick attribute of button, and GUI widgets like ctextbox have Source attributes that can be used to connect them to sources, so that their values can be read by code running because of, e.g., an Onclick event. It is also possible to create an "active" HTML fragment that runs a transaction to determine its content, possibly allocating some sources in the process:

```
val active : unit → tag [Code = transaction xbody] body [] [] []
```

8.6.4 Remote Procedure Calls

Any function call may be made a client-to-server "remote procedure call" if the function being called needs no features that are only available to client code. To make a function call an RPC, pass that function call as the argument to Basis.rpc:

```
val rpc : t ::: Type → transaction t → transaction t
```

There is an alternate form that uses None to indicate that an error occurred during RPC processing, rather than raising an exception to abort this branch of control flow.

```
val tryRpc : t ::: Type → transaction t → transaction (option t)
```

8.6.5 Asynchronous Message-Passing

To support asynchronous, "server push" delivery of messages to clients, any client that might need to receive an asynchronous message is assigned a unique ID. These IDs may be retrieved both on the client and on the server, during execution of code related to a client.

```
type client
val self : transaction client
```

Channels are the means of message-passing. Each channel is created in the context of a client and belongs to that client; no other client may receive the channel's messages. Each channel type includes the type of values that may be sent over the channel. Sending and receiving are asynchronous, in the sense that a client need not be ready to receive a message right away. Rather, sent messages may queue up, waiting to be processed.

```
con channel :: Type → Type
val channel : t ::: Type → transaction (channel t)
val send : t ::: Type → channel t → t → transaction unit
val recv : t ::: Type → channel t → transaction t
```

The channel and send operations may only be executed on the server, and recv may only be executed on a client. Neither clients nor channels may be passed as arguments from clients to server-side functions, so persistent channels can only be maintained by storing them in the database and looking them up using the current client ID or some application-specific value as a key.

Clients and channels live only as long as the web browser page views that they are associated with. When a user surfs away, his client and its channels will be garbage-collected, after that user is not heard from for the timeout period. Garbage collection deletes any database row that contains a client or channel directly.

Any reference to one of these types inside an option is set to None instead. Both kinds of handling have the flavor of weak pointers, and that is a useful way to think about clients and channels in the database.

Note: Currently, there are known concurrency issues with multi-threaded applications that employ message-passing on top of database engines that don't support true serializable transactions. Postgres 9.1 is the only supported engine that does this properly.

9 Ur/Web Syntax Extensions

Ur/Web features some syntactic shorthands for building values using the functions from the last section. This section sketches the grammar of those extensions. We write spans of syntax inside brackets to indicate that they are optional.

9.1 SQL

9.1.1 Table Declarations

table declarations may include constraints, via these grammar rules.

Declarations	d	::=	table $x : c$ [pk[,]] cts | view $x = V$
Primary key constraints	pk	::=	PRIMARY KEY K
Keys	K	::=	f | $(f, (f,)^+)$ | $\{\{e\}\}$
Constraint sets	cts	::=	CONSTRAINT f ct | cts, cts | $\{\{e\}\}$
Constraints	ct	::=	UNIQUE K | CHECK E
			| FOREIGN KEY K REFERENCES F (K) [ON DELETE pr] [ON UPDATE pr]
Foreign tables	F	::=	x | $\{\{e\}\}$
Propagation modes	pr	::=	NO ACTION | RESTRICT | CASCADE | SET NULL
View expressions	V	::=	Q | $\{e\}$

A signature item table x : c is actually elaborated into two signature items: con x_hidden_constraints :: {{Unit}} and val x : sql_table c x_hidden_constraints. This is appropriate for common cases where client code doesn't care which keys a table has. It's also possible to include constraints after a table signature item, with the same syntax as for table declarations. This may look like dependent typing, but it's just a convenience. The constraints are type-checked to determine a constructor u to include in val x : sql_table c (u ++ x_hidden_constraints), and then the expressions are thrown away. Nonetheless, it can be useful for documentation purposes to include table constraint details in signatures. Note that the automatic generation of x_hidden_constraints leads to a kind of free subtyping with respect to which constraints are defined.

9.1.2 Queries

Queries Q are added to the rules for expressions e.

Queries	Q	::=	(q [ORDER BY O] [LIMIT N] [OFFSET N])
Pre-queries	q	::=	SELECT [DISTINCT] P FROM $F,^+$ [WHERE E] [GROUP BY $p,^+$] [HAVING E]
			| q R q | $\{\{\{e\}\}\}$
Relational operators	R	::=	UNION | INTERSECT | EXCEPT
ORDER BY items	O	::=	RANDOM[()] | $\hat{E}$ $[o]$ | $\hat{E}$ $[o], O$

Projections	P	::=	$*$	all columns
			$p,^+$	particular columns
Pre-projections	p	::=	$t.f$	one column from a table
			$t.\{\{c\}\}$	a record of columns from a table (of kind {Type})
			$t.*$	all columns from a table
			$\hat{E}$ [AS f]	expression column
Table names	t	::=	x	constant table name (automatically capitalized)
			X	constant table name
			$\{\{c\}\}$	computed table name (of kind Name)
Column names	f	::=	X	constant column name
			$\{c\}$	computed column name (of kind Name)
Tables	T	::=	x	table variable, named locally by its own capitalizat
			x AS X	table variable, with local name
			x AS $\{c\}$	table variable, with computed local name
			$\{\{e\}\}$ AS t	computed table expression, with local name
			$\{\{e\}\}$ AS $\{c\}$	computed table expression, with computed local na
FROM items	F	::=	$T \mid \{\{e\}\} \mid F\ J$ JOIN F ON E	
			$\mid F$ CROSS JOIN F	
			$\mid (Q)$ AS t	
Joins	J	::=	[INNER]	
			| [LEFT | RIGHT | FULL] [OUTER]	
SQL expressions	E	::=	$t.f$	column references
			X	named expression references
			$\{[e]\}$	injected native Ur expressions
			$\{e\}$	computed expressions, probably using sql_exp direc
			TRUE | FALSE	boolean constants
			ℓ	primitive type literals
			NULL	null value (injection of None)
			E IS NULL	nullness test
			COALESCE(E, E)	take first non-null value
			n	nullary operators
			$u\ E$	unary operators
			$E\ b\ E$	binary operators
			COUNT$(*)$	count number of rows
			$a(E)$	other aggregate function
			IF E THEN E ELSE E	conditional
			(Q)	subquery (must return a single expression column)
			(E)	explicit precedence
Nullary operators	n	::=	CURRENT_TIMESTAMP	
Unary operators	u	::=	NOT	
Binary operators	b	::=	AND | OR $\mid =\mid \neq \mid < \mid \leq \mid > \mid \geq$	
Aggregate functions	a	::=	COUNT | AVG | SUM | MIN | MAX	
Directions	o	::=	ASC | DESC | $\{e\}$	
SQL integer	N	::=	$n \mid \{e\}$	
Windowable expressions	$\hat{E}$	::=	E	
			w [OVER (	(Postgres only)
			[PARTITION BY E]	
			[ORDER BY O])]	
Window function	w	::=	RANK()	
			COUNT$(*)$	
			$a(E)$	

Additionally, an SQL expression may be inserted into normal Ur code with the syntax (SQL E) or (WHERE E). Similar shorthands exist for other nonterminals, with the prefix FROM for FROM items and SELECT1 for pre-queries.

Unnamed expression columns in SELECT clauses are assigned consecutive natural numbers, starting with 1. Any expression in a p position that is enclosed in parentheses is treated as an expression column, rather than a column pulled directly out of a table, even if it is only a field projection. (This distinction affects the record type used to describe query results.)

9.1.3 DML

DML commands D are added to the rules for expressions e.

$$\begin{array}{rrcl} \text{Commands} & D & ::= & (\mathsf{INSERT\ INTO}\ T^E\ (f,^+)\ \mathsf{VALUES}\ (E,^+)) \\ & & & (\mathsf{UPDATE}\ T^E\ \mathsf{SET}\ (f = E,)^+\ \mathsf{WHERE}\ E) \\ & & & (\mathsf{DELETE\ FROM}\ T^E\ \mathsf{WHERE}\ E) \\ \text{Table expressions} & T^E & ::= & x \mid \{\{e\}\} \end{array}$$

Inside UPDATE and DELETE commands, lone variables X are interpreted as references to columns of the implicit table T, rather than to named expressions.

9.2 XML

XML fragments L are added to the rules for expressions e.

XML fragments	L	::=	`<xml/>` | `<xml>`l^*`</xml>`	
XML pieces	l	::=	text	cdata
			`<`g`/>`	tag with no children
			`<`g`>`l^*`</`x`>`	tag with children
			$\{e\}$	computed XML fragment
			$\{[e]\}$	injection of an Ur expression, via the Top.txt function
Tag	g	::=	$h\ (x = v)^*$	
Tag head	h	::=	x	tag name
			$h\{c\}$	constructor parameter
Attribute value	v	::=	ℓ	literal value
			$\{e\}$	computed value

Further, there is a special convenience and compatibility form for setting CSS classes of tags. If a `class` attribute has a value that is a string literal, the literal is parsed in the usual HTML way and replaced with calls to appropriate Ur/Web combinators. Any dashes in the text are replaced with underscores to determine Ur identifiers. The same desugaring can be accessed in a normal expression context by calling the pseudo-function `CLASS` on a string literal.

Similar support is provided for `style` attributes. Normal CSS syntax may be used in string literals that are `style` attribute values, and the desugaring may be accessed elsewhere with the pseudo-function `STYLE`.

10 The Structure of Web Applications

A web application is built from a series of modules, with one module, the last one appearing in the `.urp` file, designated as the main module. The signature of the main module determines the URL entry points to the application. Such an entry point should have type $\mathsf{t1} \to \ldots \to \mathsf{tn} \to \mathsf{transaction\ page}$, for any integer $n \geq 0$, where page is a type synonym for top-level HTML pages, defined in Basis. If such a function is at the top level of main module M, with $n = 0$, it will be accessible at URI `/M/f`, and so on for more deeply nested functions, as described in Section 12.10 below. See Section 3.1 for information on the `prefix` and `rewrite`

`url` directives, which can be used to rewrite the default URIs of different entry point functions. The final URL of a function is its default module-based URI, with `rewrite url` rules applied, and with the `prefix` prepended. Arguments to an entry-point function are deserialized from the part of the URI following `f`.

Elements of modules beside the main module, including page handlers, will only be included in the final application if they are transitive dependencies of the handlers in the main module.

Normal links are accessible via HTTP `GET`, which the relevant standard says should never cause side effects. To export a page which may cause side effects, accessible only via HTTP `POST`, include one argument of the page handler of type Basis.postBody. When the handler is called, this argument will receive a value that can be deconstructed into a MIME type (with Basis.postType) and payload (with Basis.postData). This kind of handler should not be used with forms that exist solely within Ur/Web apps; for these, use Ur/Web's built-in support, as described below. It may still be useful to use Basis.postBody with form requests submitted by code outside an Ur/Web app. For such cases, the function Top.postFields : postBody → list (string × string) may be useful, breaking a `POST` body of type `application/x-www-form-urlencoded` into its name-value pairs.

Any normal page handler may also include arguments of type option Basis.queryString, which will be handled specially. Rather than being deserialized from the current URI, such an argument is passed the whole query string that the handler received. The string may be analyzed by calling Basis.show on it. A handler of this kind may be passed as an argument to Basis.effectfulUrl to generate a URL to a page that may be used as a "callback" by an external service, such that the handler is allowed to cause side effects.

When the standalone web server receives a request for a known page, it calls the function for that page, "running" the resulting transaction to produce the page to return to the client. Pages link to other pages with the `link` attribute of the `a` HTML tag. A link has type transaction page, and the semantics of a link are that this transaction should be run to compute the result page, when the link is followed. Link targets are assigned URL names in the same way as top-level entry points.

HTML forms are handled in a similar way. The action attribute of a submit form tag takes a value of type $use → transaction page, where use is a kind-{Type} record of the form fields used by this action handler. Action handlers are assigned URL patterns in the same way as above.

For both links and actions, direct arguments and local variables mentioned implicitly via closures are automatically included in serialized form in URLs, in the order in which they appear in the source code. Such serialized values may only be drawn from a limited set of types, and programs will fail to compile when the (implicit or explicit) arguments of page handler functions involve disallowed types. (Keep in mind that every free variable of a function is an implicit argument if it was not defined at the top level of a module.) For instance:

- Functions are disallowed, since there is no obvious way to serialize them safely.
- XML fragments are disallowed, since it is unclear how to check client-provided XML to be sure it doesn't break the HTML invariants of the application (for instance, by mutating the DOM in the conventional way, interfering with Ur/Web's functional-reactive regime).
- Blobs ("files") are disallowed, since they can easily have very large serializations that could not fit within most web servers' URL size limits. (And you probably don't want to be serializing, e.g., image files in URLs, anyway.)

Ur/Web programs generally mix server- and client-side code in a fairly transparent way. The one important restriction is that mixed client-server code must encapsulate all server-side pieces within named functions. This is because execution of such pieces will be implemented by explicit calls to the remote web server, and it is useful to get the programmer's help in designing the interface to be used. For example, this makes it easier to allow a client running an old version of an application to continue interacting with a server that has been upgraded to a new version, if the programmer took care to keep the interfaces of all of the old remote calls the same. The functions implementing these services are assigned names in the same way as normal web entry points, by using module structure.

The HTTP standard suggests that GET requests only be used in ways that generate no side effects. Side effecting operations should use POST requests instead. The Ur/Web compiler enforces this rule strictly, via a simple conservative program analysis. Any page that may have a side effect must be accessed through a form, all of which use POST requests, or via a direct call to a page handler with some argument of type Basis.postBody. A page is judged to have a side effect if its code depends syntactically on any of the side-effecting, server-side FFI functions. Links, forms, and most client-side event handlers are not followed during this syntactic traversal, but `<body onload={...}>` handlers *are* examined, since they run right away and could just as well be considered parts of main page handlers.

Ur/Web includes a kind of automatic protection against cross site request forgery attacks. Whenever any page execution can have side effects and can also read at least one cookie value, all cookie values must be signed cryptographically, to ensure that the user has come to the current page by submitting a form on a real page generated by the proper server. Signing and signature checking are inserted automatically by the compiler. This prevents attacks like phishing schemes where users are directed to counterfeit pages with forms that submit to your application, where a user's cookies might be submitted without his knowledge, causing some undesired side effect.

10.1 Tasks

In many web applications, it's useful to run code at points other than requests from browsers. Ur/Web's *task* mechanism facilitates this. A type family of *task kinds* is in the standard library:

```
con task_kind :: Type → Type
val initialize : task_kind unit
val clientLeaves : task_kind client
val periodic : int → task_kind unit
```

A task kind names a particular extension point of generated applications, where the type parameter of a task kind describes which extra input data is available at that extension point. Add task code with the special declaration form task $e_1 = e_2$, where e_1 is a task kind with data τ, and e_2 is a function from τ to transaction unit.

The currently supported task kinds are:

- initialize: Code that is run when the application starts up.
- clientLeaves: Code that is run for each client that the runtime system decides has surfed away. When a request that generates a new client handle is aborted, that handle will still eventually be passed to clientLeaves task code, even though the corresponding browser was never informed of the client handle's existence. In other words, in general, clientLeaves handlers will be called more times than there are actual clients.
- periodic n: Code that is run when the application starts up and then every n seconds thereafter.

11 The Foreign Function Interface

It is possible to call your own C and JavaScript code from Ur/Web applications, via the foreign function interface (FFI). The starting point for a new binding is a `.urs` signature file that presents your external library as a single Ur/Web module (with no nested modules). Compilation conventions map the types and values that you use into C and/or JavaScript types and values.

It is most convenient to encapsulate an FFI binding with a new `.urp` file, which applications can include with the `library` directive in their own `.urp` files. A number of directives are likely to show up in the library's project file.

- `clientOnly Module.ident` registers a value as being allowed only in client-side code.

- `clientToServer Module.ident` declares a type as OK to marshal between clients and servers. By default, abstract FFI types are not allowed to be marshalled, since your library might be maintaining invariants that the simple serialization code doesn't check.
- `effectful Module.ident` registers a function that can have side effects. It is important to remember to use this directive for each such function, or else the optimizer might change program semantics. (Note that merely assigning a function a `transaction`-based type does not mark it as effectful in this way!)
- `ffi FILE.urs` names the file giving your library's signature. You can include multiple such files in a single `.urp` file, and each file `mod.urp` defines an FFI module `Mod`.
- `include FILE` requests inclusion of a C header file.
- `jsFunc Module.ident=name` gives a mapping from an Ur name for a value to a JavaScript name.
- `link FILE` requests that `FILE` be linked into applications. It should be a C object or library archive file, and you are responsible for generating it with your own build process.
- `script URL` requests inclusion of a JavaScript source file within application HTML.
- `serverOnly Module.ident` registers a value as being allowed only in server-side code.

11.1 Writing C FFI Code

A server-side FFI type or value `Module.ident` must have a corresponding type or value definition `uw_Module_ident` in C code. With the current Ur/Web version, it's not generally possible to work with Ur records or complex datatypes in C code, but most other kinds of types are fair game.

- Primitive types defined in `Basis` are themselves using the standard FFI interface, so you may refer to them like `uw_Basis_t`. See `include/types.h` for their definitions.
- Enumeration datatypes, which have only constructors that take no arguments, should be defined using C `enum`s. The type is named as for any other type identifier, and each constructor `c` gets an enumeration constant named `uw_Module_c`.
- A datatype `dt` (such as `Basis.option`) that has one non-value-carrying constructor `NC` and one value-carrying constructor `C` gets special treatment. Where `T` is the type of `C`'s argument, and where we represent `T` as `t` in C, we represent `NC` with `NULL`. The representation of `C` depends on whether we're sure that we don't need to use `NULL` to represent `t` values; this condition holds only for strings and complex datatypes. For such types, `C v` is represented with the C encoding of `v`, such that the translation of `dt` is `t`. For other types, `C v` is represented with a pointer to the C encoding of v, such that the translation of `dt` is `t*`.
- Ur/Web involves many types of program syntax, such as for HTML and SQL code. All of these types are implemented with normal C strings, and you may take advantage of that encoding to manipulate code as strings in C FFI code. Be mindful that, in writing such code, it is your responsibility to maintain the appropriate code invariants, or you may reintroduce the code injection vulnerabilities that Ur/Web rules out. The most convenient way to extend Ur/Web with functions that, e.g., use natively unsupported HTML tags is to generate the HTML code with the FFI.

The C FFI version of a Ur function with type `T1 -> ... -> TN -> R` or `T1 -> ... -> TN -> transaction R` has a C prototype like `R uw_Module_ident(uw_context, T1, ..., TN)`. Only functions with types of the second form may have side effects. `uw_context` is the type of state that persists across handling a client request. Many functions that operate on contexts are prototyped in `include/urweb.h`. Most should only be used internally by the compiler. A few are useful in general FFI implementation:

- `void uw_error(uw_context, failure_kind, const char *fmt, ...);`

  Abort the current request processing, giving a `printf`-style format string and arguments for generating an error message. The `failure_kind` argument can be `FATAL`, to abort the whole execution; `BOUNDED_RETRY`, to try processing the request again from the beginning, but failing if this happens too many times; or `UNLIMITED_RETRY`, to repeat processing, with no cap on how many times this can recur.

  All pointers to the context-local heap (see description below of `uw_malloc()`) become invalid at the start and end of any execution of a main entry point function of an application. For example, if the request handler is restarted because of a `uw_error()` call with `BOUNDED_RETRY` or for any other reason, it is unsafe to access any local heap pointers that may have been stashed somewhere beforehand.

- `void uw_set_error_message(uw_context, const char *fmt, ...);`

  This simpler form of `uw_error()` saves an error message without immediately aborting execution.

- ```
  void uw_push_cleanup(uw_context, void (*func)(void *), void *arg);
  void uw_pop_cleanup(uw_context);
  ```

  Manipulate a stack of actions that should be taken if any kind of error condition arises. Calling the "pop" function both removes an action from the stack and executes it. It is a bug to let a page request handler finish successfully with unpopped cleanup actions.

  Pending cleanup actions aren't intended to have any complex relationship amongst themselves, so, upon request handler abort, pending actions are executed in first-in-first-out order.

- `void *uw_malloc(uw_context, size_t);`

  A version of `malloc()` that allocates memory inside a context's heap, which is managed with region allocation. Thus, there is no `uw_free()`, but you need to be careful not to keep ad-hoc C pointers to this area of memory. In general, `uw_malloc()`ed memory should only be used in ways compatible with the computation model of pure Ur. This means it is fine to allocate and return a value that could just as well have been built with core Ur code. In contrast, it is almost never safe to store `uw_malloc()`ed pointers in global variables, including when the storage happens implicitly by registering a callback that would take the pointer as an argument.

  For performance and correctness reasons, it is usually preferable to use `uw_malloc()` instead of `malloc()`. The former manipulates a local heap that can be kept allocated across page requests, while the latter uses global data structures that may face contention during concurrent execution. However, we emphasize again that `uw_malloc()` should never be used to implement some logic that couldn't be implemented trivially by a constant-valued expression in Ur.

- ```
  typedef void (*uw_callback)(void *);
  typedef void (*uw_callback_with_retry)(void *, int will_retry);
  void uw_register_transactional(uw_context, void *data, uw_callback commit,
                                 uw_callback rollback, uw_callback_with_retry free);
  ```

  All side effects in Ur/Web programs need to be compatible with transactions, such that any set of actions can be undone at any time. Thus, you should not perform actions with non-local side effects directly; instead, register handlers to be called when the current transaction is committed or rolled back. The arguments here give an arbitary piece of data to be passed to callbacks, a function to call on commit, a function to call on rollback, and a function to call afterward in either case to clean up any allocated resources. A rollback handler may be called after the associated commit handler has already

been called, if some later part of the commit process fails. A free handler is told whether the runtime system expects to retry the current page request after rollback finishes.

Any of the callbacks may be `NULL`. To accommodate some stubbornly non-transactional real-world actions like sending an e-mail message, Ur/Web treats `NULL rollback` callbacks specially. When a transaction commits, all `commit` actions that have non-`NULL` rollback actions are tried before any `commit` actions that have `NULL` rollback actions. Thus, if a single execution uses only one non-transactional action, and if that action never fails partway through its execution while still causing an observable side effect, then Ur/Web can maintain the transactional abstraction.

When a request handler ends with multiple pending transactional actions, their handlers are run in a first-in-last-out stack-like order, wherever the order would otherwise be ambiguous.

It is not safe for any of these handlers to access a context-local heap through a pointer returned previously by `uw_malloc()`, nor should any new calls to that function be made. Think of the context-local heap as meant for use by the Ur/Web code itself, while transactional handlers execute after the Ur/Web code has finished.

A handler may signal an error by calling `uw_set_error_message()`, but it is not safe to call `uw_error()` from a handler. Signaling an error in a commit handler will cause the runtime system to switch to aborting the transaction, immediately after the current commit handler returns.

- ```
  void *uw_get_global(uw_context, char *name);
  void uw_set_global(uw_context, char *name, void *data, uw_callback free);
  ```

  Different FFI-based extensions may want to associate their own pieces of data with contexts. The global interface provides a way of doing that, where each extension must come up with its own unique key. The `free` argument to `uw_set_global()` explains how to deallocate the saved data. It is never safe to store `uw_malloc()`ed pointers in global variable slots.

11.2 Writing JavaScript FFI Code

JavaScript is dynamically typed, so Ur/Web type definitions imply no JavaScript code. The JavaScript identifier for each FFI function is set with the `jsFunc` directive. Each identifier can be defined in any JavaScript file that you ask to include with the `script` directive.

In contrast to C FFI code, JavaScript FFI functions take no extra context argument. Their argument lists are as you would expect from their Ur types. Only functions whose ranges take the form `transaction T` should have side effects; the JavaScript "return type" of such a function is `T`. Here are the conventions for representing Ur values in JavaScript.

- Integers, floats, strings, characters, and booleans are represented in the usual JavaScript way.
- Ur functions are represented in an unspecified way. This means that you should not rely on any details of function representation. Named FFI functions are represented as JavaScript functions with as many arguments as their Ur types specify. To call a non-FFI function `f` on argument `x`, run `execF(f, x)`. To lift a normal JavaScript function `f` into an Ur/Web JavaScript function, run `flift(f)`.
- An Ur record is represented with a JavaScript record, where Ur field name `N` translates to JavaScript field name `_N`. An exception to this rule is that the empty record is encoded as `null`.
- `option`-like types receive special handling similar to their handling in C. The "`None`" constructor is `null`, and a use of the "`Some`" constructor on a value `v` is either `v`, if the underlying type doesn't need to use `null`; or `{v:v}` otherwise.
- Any other datatypes represent a non-value-carrying constructor `C` as `"C"` and an application of a constructor `C` to value `v` as `{n:"C", v:v}`. This rule only applies to datatypes defined in FFI module
  ```

signatures; the compiler is free to optimize the representations of other, non-`option`-like datatypes in arbitrary ways.

- As in the C FFI, all abstract types of program syntax are implemented with strings in JavaScript.
- A value of Ur type `transaction t` is represented in the same way as for `unit -> t`.

It is possible to write JavaScript FFI code that interacts with the functional-reactive structure of a document. Here is a quick summary of some of the simpler functions to use; descriptions of fancier stuff may be added later on request (and such stuff should be considered "undocumented features" until then).

- Sources should be treated as an abstract type, manipulated via:
  - `sc(v)`, to create a source initialized to `v`
  - `sg(s)`, to retrieve the current value of source `s`
  - `sv(s, v)`, to set source `s` to value `v`
- Signals should be treated as an abstract type, manipulated via:
  - `sr(v)` and `sb(s, f)`, the "return" and "bind" monad operators, respectively
  - `ss(s)`, to produce the signal corresponding to source `s`
  - `scur(s)`, to get the current value of signal `s`
- The behavior of the `<dyn>` pseudo-tag may be mimicked by following the right convention in a piece of HTML source code with a type like `xbody`. Such a piece of source code may be encoded with a JavaScript string. To insert a dynamic section, include a `<script>` tag whose content is just a call `dyn(pnode, s)`. The argument `pnode` specifies what the relevant enclosing parent tag is. Use value `"tr"` when the immediate parent is `<tr>`, use `"table"` when the immediate parent is `<table>`, and use `"span"` otherwise. The argument `s` is a string-valued signal giving the HTML code to be inserted at this point. As with the usual `<dyn>` tag, that HTML subtree is automatically updated as the value of `s` changes.
- There is only one supported method of taking HTML values generated in Ur/Web code and adding them to the DOM in FFI JavaScript code: call `setInnerHTML(node, html)` to add HTML content `html` within DOM node `node`. Merely running `node.innerHTML = html` is not guaranteed to get the job done, though programmers familiar with JavaScript will probably find it useful to think of `setInnerHTML` as having this effect. The unusual idiom is required because Ur/Web uses a nonstandard representation of HTML, to support infinite nesting of code that may generate code that may generate code that.... The `node` value must already be in the DOM tree at the point when `setInnerHTML` is called, because some plumbing must be set up to interact sensibly with `<dyn>` tags.
- It is possible to use the more standard "IDs and mutation" style of JavaScript coding, though that style is unidiomatic for Ur/Web and should be avoided wherever possible. Recall the abstract type id and its constructor fresh, which can be used to generate new unique IDs in Ur/Web code. Values of this type are represented as strings in JavaScript, and a function `fresh()` is available to generate new unique IDs. Application-specific ID generation schemes may cause bad interactions with Ur/Web code that also generates IDs, so the recommended approach is to produce IDs only via calls to `fresh()`. FFI code shouldn't depend on the ID generation scheme (on either server side or client side), but it is safe to include these IDs in tag attributes (in either server-side or client-side code) and manipulate the associated DOM nodes in the standard way (in client-side code). Be forewarned that this kind of imperative DOM manipulation may confuse the Ur/Web runtime system and interfere with proper behavior of tags like `<dyn>`!

11.3 Introducing New HTML Tags

FFI modules may introduce new tags as values with Basis.tag types. See `basis.urs` for examples of how tags are declared. The identifier of a tag value is used as its rendering in HTML. The Ur/Web syntax sugar for XML literals desugars each use of a tag into a reference to an identifier with the same name. There is no need to provide implementations (i.e., in C or JavaScript code) for such identifiers.

The onus is on the coder of a new tag's interface to think about consequences for code injection attacks, messing with the DOM in ways that may break Ur/Web reactive programming, etc.

12 Compiler Phases

The Ur/Web compiler is unconventional in that it relies on a kind of *heuristic compilation*. Not all valid programs will compile successfully. Informally, programs fail to compile when they are "too higher order." Compiler phases do their best to eliminate different kinds of higher order-ness, but some programs just won't compile. This is a trade-off for producing very efficient executables. Compiled Ur/Web programs use native C representations and require no garbage collection.

In this section, we step through the main phases of compilation, noting what consequences each phase has for effective programming.

12.1 Parse

The compiler reads a `.urp` file, figures out which `.urs` and `.ur` files it references, and combines them all into what is conceptually a single sequence of declarations in the core language of Section 4.2.

12.2 Elaborate

This is where type inference takes place, translating programs into an explicit form with no more wildcards. This phase is the most likely source of compiler error messages.

Those crawling through the compiler source will also want to be aware of another compiler phase, Explify, that occurs immediately afterward. This phase just translates from an AST language that includes unification variables to a very similar language that doesn't; all variables should have been determined by the end of Elaborate, anyway. The new AST language also drops some features that are used only for static checking and that have no influence on runtime behavior, like disjointness constraints.

12.3 Unnest

Named local function definitions are moved to the top level, to avoid the need to generate closures.

12.4 Corify

Module system features are compiled away, through inlining of functor definitions at application sites. Afterward, most abstraction boundaries are broken, facilitating optimization.

12.5 Especialize

Functions are specialized to particular argument patterns. This is an important trick for avoiding the need to maintain any closures at runtime. Currently, specialization only happens for prefixes of a function's full list of parameters, so you may need to take care to put arguments of function types before other arguments. The optimizer will not be effective enough if you use arguments that mix functions and values that must be calculated at run-time. For instance, a tuple of a function and an integer counter would not lead to successful code generation; these should be split into separate arguments via currying.

12.6 Untangle

Remove unnecessary mutual recursion, splitting recursive groups into strongly connected components.

12.7 Shake

Remove all definitions not needed to run the page handlers that are visible in the signature of the last module listed in the `.urp` file.

12.8 Rpcify

Pieces of code are determined to be client-side, server-side, neither, or both, by figuring out which standard library functions might be needed to execute them. Calls to server-side functions (e.g., `query`) within mixed client-server code are identified and replaced with explicit remote calls. Some mixed functions may be converted to continuation-passing style to facilitate this transformation.

12.9 Untangle, Shake

Repeat these simplifications.

12.10 Tag

Assign a URL name to each link and form action. It is important that these links and actions are written as applications of named functions, because such names are used to generate URL patterns. A URL pattern has a name built from the full module path of the named function, followed by the function name, with all pieces separated by slashes. The path of a functor application is based on the name given to the result, rather than the path of the functor itself.

12.11 Reduce

Apply definitional equality rules to simplify the program as much as possible. This effectively includes inlining of every non-recursive definition.

12.12 Unpoly

This phase specializes polymorphic functions to the specific arguments passed to them in the program. If the program contains real polymorphic recursion, Unpoly will be insufficient to avoid later error messages about too much polymorphism.

12.13 Specialize

Replace uses of parameterized datatypes with versions specialized to specific parameters. As for Unpoly, this phase will not be effective enough in the presence of polymorphic recursion or other fancy uses of impredicative polymorphism.

12.14 Shake

Here the compiler repeats the earlier Shake phase.

12.15 Monoize

Programs are translated to a new intermediate language without polymorphism or non-Type constructors. Error messages may pop up here if earlier phases failed to remove such features.

This is the stage at which concrete names are generated for cookies, tables, and sequences. They are named following the same convention as for links and actions, based on module path information saved from earlier stages. Table and sequence names separate path elements with underscores instead of slashes, and they are prefixed by `uw_`.

12.16 MonoOpt

Simple algebraic laws are applied to simplify the program, focusing especially on efficient imperative generation of HTML pages.

12.17 MonoUntangle

Unnecessary mutual recursion is broken up again.

12.18 MonoReduce

Equivalents of the definitional equality rules are applied to simplify programs, with inlining again playing a major role.

12.19 MonoShake, MonoOpt

Unneeded declarations are removed, and basic optimizations are repeated.

12.20 Fuse

The compiler tries to simplify calls to recursive functions whose results are immediately written as page output. The write action is pushed inside the function definitions to avoid allocation of intermediate results.

12.21 MonoUntangle, MonoShake

Fuse often creates more opportunities to remove spurious mutual recursion.

12.22 Pathcheck

The compiler checks that no link or action name has been used more than once.

12.23 Cjrize

The program is translated to what is more or less a subset of C. If any use of functions as data remains at this point, the compiler will complain.

12.24 C Compilation and Linking

The output of the last phase is pretty-printed as C source code and passed to the C compiler.