MIT Libraries | DSpace@MIT

MIT Open Access Articles

An Optimizing Compiler for a Purely Functional Web-Application Language

The MIT Faculty has made this article openly available. ***Please share*** how this access benefits you. Your story matters.

Citation: Chlipala, Adam. 2015. "An Optimizing Compiler for a Purely Functional Web-Application Language."

As Published: 10.1145/2784731.2784741

Publisher: Association for Computing Machinery (ACM)

Persistent URL: https://hdl.handle.net/1721.1/138082

Version: Author's final manuscript: final author's manuscript post peer review, without publisher's formatting or copy editing

Terms of use: Creative Commons Attribution-Noncommercial-Share Alike

Massachusetts Institute of Technology

An Optimizing Compiler for a Purely Functional Web-Application Language

Adam Chlipala
MIT CSAIL, USA
adamc@csail.mit.edu

Abstract

High-level scripting languages have become tremendously popular for development of dynamic Web applications. Many programmers appreciate the productivity benefits of automatic storage management, freedom from verbose type annotations, and so on. While it is often possible to improve performance substantially by rewriting an application in C or a similar language, very few programmers bother to do so, because of the consequences for human development effort. This paper describes a compiler that makes it possible to have most of the best of both worlds, coding Web applications in a high-level language but compiling to native code with performance comparable to handwritten C code. The source language is Ur/Web, a domain-specific, purely functional, statically typed language for the Web. Through a coordinated suite of relatively straightforward program analyses and algebraic optimizations, we transform Ur/Web programs into almost-idiomatic C code, with no garbage collection, little unnecessary memory allocation for intermediate values, etc. Our compiler is in production use for commercial Web sites supporting thousands of users, and microbenchmarks demonstrate very competitive performance versus mainstream tools.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors - Optimization; D.3.2 [*Programming Languages*]: Language Classifications - Applicative (functional) languages

Keywords Web programming languages; pure functional programming; whole-program optimization

1. Introduction

With the popularity explosion of the Internet within the global economy, tools for building Internet servers are more important than ever. Small improvements in tool practicality can pay off massively. Consider what is probably the most commonly implemented kind of Internet server, dynamic Web applications. A popular application may end up serving thousands of simultaneous user requests, so there is clear economic value to optimizing the application's performance: the better the performance, the fewer physical servers the site owners must pay for, holding user activity constant. One might conclude, then, that most Web applications would be written in C or other low-level languages that tend to enable the highest performance. However, real-world programmers seem overwhelmingly to prefer the greater programming simplicity of high-level languages, be they dynamically typed scripting languages like JavaScript or statically typed old favorites like Java. There is a genuine trade-off to be made between hardware costs for running a service and labor costs for implementing it.

Might there be a way to provide C-like performance for programs coded in high-level languages? While the concept of *domain-specific languages* is growing in visibility, most Web applications are still written in general-purpose languages, coupled with Web-specific framework libraries. Whether we examine JavaScript, Java, or any of the other most popular languages for authoring Web applications, we see more or less the same programming model. Compilers for these languages make heroic efforts to understand program structure well enough to do effective optimization, but theirs is an uphill battle. First, program analysis is inevitably complex within an unstructured imperative model, supporting objects with arbitrary lifetimes, accessed via pointers or references that may be aliased. Second, Web frameworks typically involve integral features like database access and HTML generation, but a general-purpose compiler knows nothing about these aspects and cannot perform specialized optimizations for them.

In this paper, we present a different approach, via an **optimizing compiler for Ur/Web [6], a domain-specific, purely functional, statically typed language for Web applications**. In some sense, Ur/Web starts at a disadvantage, with pure programs written to, e.g., compute an HTML page as a first-class object using pure combinators, rather than constructing the page imperatively. Further, Ur/Web exposes a simple transaction-based concurrency model, guaranteeing simpler kinds of cross-thread interference than in mainstream languages, and there is a corresponding runtime cost. However, in another important sense, Ur/Web has a great advantage over frameworks in general-purpose languages: the compiler is specialized to the context of serving Web requests, generating HTML pages, and accessing SQL databases. As a result, we have been able to build our compiler to generate **the highest-performing applications from the shortest, most declarative programs**, in representative microbenchmarks within a community-run comparison involving over 100 different frameworks/languages.

For a quick taste of Ur/Web, consider the short example program in Figure 1. In general, Ur/Web programs are structured as modules exporting functions that correspond to URL prefixes. A user effectively calls one of these functions from his browser via the function's URL prefix, providing function arguments serialized into the URL, and then seeing the HTML page that the function returns. Figure 1 shows one such remotely callable function, `showCategory`, which takes one parameter `cat`.

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in the following publication:

ICFP'15, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08...
http://dx.doi.org/10.1145/2784731.2784741

```
table message : { Category : string, Id : int,
                  Text : string }

fun showCategory cat =
 messages <-
   queryX1 (SELECT message.Id, message.Text
            FROM message
            WHERE message.Category = {[cat]}
            ORDER BY message.Id)
   (fn r => <xml><li>#{[r.Id]}:
                    {[r.Text]}</li></xml>);
 return <xml><body>
  <h1>Messages for: {[cat]}</h1>

  <ul> {messages} </ul>
 </body></xml>
```

Figure 1. Ur/Web source code for a simple running example

This particular code example is for an imaginary application that maintains textual messages grouped into categories. A user may request, via `showCategory`, to see the list of all messages found in a particular category. A `table` declaration binds a name for a typed SQL database table `message`, where each row of the table includes a textual category name, a numeric unique ID, and the actual text of the message. The Ur/Web compiler type-checks the code to make sure its use of SQL is compatible with the declared schemas of all tables, and a compiled application will check on start-up to be sure that the true SQL schema matches the one declared in the program.

The body of `showCategory` queries the database and generates an HTML page to show to the user. As Ur/Web is a purely functional language, side effects like database access must be encoded somehow. We follow Haskell in adopting monadic IO [20], leading to the `return` function and "bind" operator `<-` in the code, which may generally be read like function-return and immutable variable assignment in more conventional languages, with added type-checking to distinguish pure and impure code.

The database query works via a higher-order library function `queryX1`, for running an SQL query returning columns of just one table, producing a piece of XHTML for each result row and concatenating them all together to produce the final result. Here we see Ur/Web's syntactic sugar for building SQL and XML snippets, which is desugared into calls to combinators typed in terms of the very expressive Ur type system [5]. Each bit of syntax is effectively a quotation, and the syntax `{[e]}` indicates injecting or *antiquoting* the value of expression `e` within a quotation, after converting it into a literal of the embedded language via type classes [27]. For instance, the SQL query is customized to find only messages that belong to the requested category, by antiquoting Ur/Web variable `cat` into the SQL `WHERE` clause; and the piece of HTML generated for each result row uses antiquoting to render the ID and text fields from that row.

The last piece of `showCategory`'s body computes the final HTML page to return to the user. We antiquote both the category name and the list of message items computed above (stored in `messages`). As the latter is already an XML fragment, we inject it with the simpler syntax `{e}`, since we do not want any further interpretation as a literal of the embedded language. Conceptually, the HTML expression here denotes a typed abstract syntax tree, but the Ur/Web implementation takes care of serializing it in standard HTML concrete syntax, as part of the general process of parsing an HTTP request, dispatching to the proper Ur/Web function, and returning an HTTP response.

The example code is written at a markedly higher level of abstraction than in mainstream Web programming, with more expressive static type checking. We might worry that either property

```
void initContext(context ctx) {
  createPreparedStatement(ctx, "stmt1",
    "SELECT Id, Text FROM message "
    "WHERE Category = ? ORDER BY Id");
}

void showCategory(context ctx, char* cat) {
  write(ctx, "<body>\n<h1>Messages for: ");
  escape_w(ctx, cat);
  write(ctx, "</h1>\n<ul> ");

  Cursor c = prepared(ctx, "stmt1", cat, NULL);
  if (has_error(c)) error(ctx, error_msg(c));
  Row r;
  while (r = next_row(c)) {
    write(ctx, "<li>#");
    stringifyInt_w(ctx, atoi(column(r, 0)));
    write(ctx, ": ");
    escape_w(ctx, column(r, 1));
    write(ctx, "</li>");
  }

  write(ctx, "\n</ul>\n</body>");
}
```

Figure 2. C code generated from source code in Figure 1

would add runtime bloat. Instead, the higher-level notation facilitates more effective optimization in our compiler. We compile via C, and Figure 2 shows C code that would be generated for our example. Throughout this paper, we simplify and beautify C output code as compared to the actual behavior of our compiler in pretty-printing C, but the real output leads to essentially the same C abstract syntax trees.

First, the optimizer has noticed that the SQL queries generated by the program follow a particular, simple template. We automatically allocate a *prepared statement* for that template, allowing the SQL engine to optimize the query ahead of time. The definition of a C function `initContext` creates the statement, which is mostly the same as the SQL code from Figure 1, with a question mark substituted for the antiquotation. In Ur/Web, compiled code is always run inside some *context*, which may be thought of as per-thread information. Each thread will generally have its own persistent database connection, for instance. Our code declares the prepared statement with a name, within that connection.

The C function `showCategory` implements the Ur/Web function of the same name. It is passed both the source-level parameter `cat` and the current context. Another important element of context is a mutable buffer for storing the HTML page that will eventually be returned to the user. Our handler function here appends to this buffer in several ways, most straightforwardly with a `write` method that appends a literal string. We also rely on other functions with names suffixed by `_w` for "write," indicating a version of a pure function that appends its output to the page buffer instead of returning it normally. Specifically, `escape_w` escapes a string as an HTML literal, and `stringifyInt_w` renders an integer as a string in decimal. The optimizer figured out that there is no need to allocate storage for the results of these operations, since they would just be written directly to the page buffer and not used again.

The next part of `showCategory` queries the database by calling the prepared statement with the parameter value filled in. A quick error check aborts the handler execution if the SQL engine has signaled failure; another part of a context is a C `setjmp` marker recording an earlier point in program execution, to which we should `longjmp` back upon encountering a fatal error, and the `error` function does exactly that. The common case, though, is to proceed

through reading all of the result rows, via a *cursor* recording our progress. A loop reads each successive row, appending appropriate HTML content to the page buffer.

We finish our tour of Figure 2 by pointing out that the C code writes many fewer whitespace characters than appeared literally in Figure 1. Since the compiler understands the XHTML semantics of collapsing adjacent whitespace characters into single spaces, it is able to cut out extraneous characters, without requiring the programmer to write less readable code without indentation.

A few big ideas underlie the Ur/Web compiler:

- Use **whole-program compilation** to enable the most straightforward analysis of interactions between different modules of a program. Our inspiration is MLton [29], a whole-program optimizing compiler for Standard ML. MLton partially evaluates programs to remove uses of abstraction and modularity mechanisms like parametric polymorphism or functors, in the sense of ML module systems [16], so there need be no runtime representation of such code patterns. Our Ur/Web compiler does the same and goes even further in requiring that *all uses of first-class functions are reduced away during compilation*.
- Where MLton relies on sophisticated control-flow and dataflow analysis [3, 12, 30], we instead rely only on **simple syntactic dependency analysis** and **algebraic rewriting**. We still take advantage of optimizations based on dataflow analysis in the C compiler that processes output of our compiler, but we do not implement any new dataflow analysis.
- A standard C compiler would not be able to optimize our output code effectively if not for another important deviation from MLton and most other functional-language implementations: we generate code that **does not use garbage collection** and that employs **idiomatic C representations for nearly all data types**. In particular, we employ a simple form of *region-based memory management* [25], where memory allocation follows a stack discipline that supports constant-time deallocation of groups of related objects. Where past work has often used nontrivial program analysis to find region structure, we use a simple algorithm based on the normal types of program subexpressions, taking advantage of Ur/Web's pure functional nature. We also employ the safe fallback of running each HTTP request handler inside a new top-level region, since Ur/Web only supports persistent cross-request state via the SQL database.
- We apply a relatively simple and specialized **fusion optimization to combine generating HTML data and imperatively appending it to page buffers**. The result is C code like in Figure 2, without any explicit concatenation of HTML fragments to form intermediate values. While more general fusion optimizations [8] can grow quite involved, our specialized transformation is relatively simple and only requires about 100 lines of code to implement.
- A key enabling optimization for fusion is a **lightweight effect analysis on an imperative intermediate language**. Ur/Web programs like the one in Figure 1 are naturally written in a monadic style, where, e.g., a database query might be performed at a point in the code fairly far from where the results are injected into a result HTML document. Moving the query closer to the injection point enables fusion. We summarize program subexpressions with effects to determine when portions commute with each other, enabling useful code motion.

We now step back and explain the compilation process in more detail. Section 2 introduces the intermediate languages that our compiler uses, and Section 3 presents the key compiler phases. Section 4 evaluates optimization effectiveness with experiments.

Ur/Web is an established open-source project with a growing community of users, and its source code and documentation are available at:

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

We have one unified message in mind, which we hope the rest of the paper supports: **A relatively simple set of algebraic optimizations makes it possible to compile a very high-level language to low-level code with extremely competitive performance**. It should be reasonably straightforward for authors of other domain-specific functional languages to apply our strategy. At the same time, the applicability conditions of these techniques are subtle: for instance, they depend on Ur/Web's flavor of purity. Nonetheless, when the conditions are met, we are able to reduce the runtime costs of high-level abstractions to near zero.

2. Intermediate Languages

First, the bare essentials of the source language: it is **statically typed**, **purely functional**, and **strict**, generally following the syntax of Standard ML by default, but adopting ideas from Haskell and dependently typed languages like Coq.

Ur/Web is a so-called *tierless* language, where programmers write a full application in a single language (potentially even within a single source file), including bits of code that ought to run on the server or on the client (browser). Purity is put to good use in a client-side GUI system based on functional-reactive programming [10], where pages are described as pure functions over data sources that may be mutated out-of-band. In this paper, we will mostly ignore the client-side aspect of Ur/Web, adding a few comments about points in the pipeline where different elements of client-server interaction are processed. The compiler is structured around a number of different representations of full Ur/Web programs:

Sets of textual source files. The starting point of compilation is traditional ASCII source code spread across multiple files, which denote different Ur/Web modules.

Source. The parser translates source code into single abstract syntax trees, standing for programs with multiple top-level module declarations, in general. This tree type matches source code very closely, though some expansions of syntactic sugar are performed.

Expl. Next we have a version of the source language where many types of nodes are annotated **expl**icitly with type information. For instance, all invocations of polymorphic functions include explicit types for the parameters, and all invocations of functions based on type classes [27] include explicit dictionary arguments to witness the results of instance resolution.

Core. Here we remove the module system from the language, so that programs consist only of sequences of declarations of types and values (e.g., functions). Figure 3 summarizes the key parts of the syntax. Briefly, Ur is an extension of System F_ω [21], the higher-order polymorphic λ-calculus. There are functions in the style of λ-calculus at both the type level and the value level, facilitating useful abstraction patterns.

Ur extends F_ω with type-level finite maps (having kinds like $\{\kappa\}$, for finite maps from names to types of kind κ, inspired by row types [28]), which are used to assign types like $\{\mathsf{A} \mapsto \mathsf{int}, \mathsf{B} \mapsto \mathsf{bool}\}$ to value-level records in a flexible way. Names (via name literals N) inhabit a special kind Name, allowing nontrivial compile-time computation with names. Value-level records are built like $\{\mathsf{A} = 1, \mathsf{B} = \mathsf{True}\}$, though the names might also be type variables bound by polymorphic functions. A field is projected from a record like $e.\mathsf{A}$. Tuple types (written like Cartesian products) are

Type vars	α		
Kinds	κ	$::=$	$\mathsf{Type} \mid \kappa \to \kappa \mid \mathsf{Name} \mid \{\kappa\} \mid \dots$
Literal names	N		
Types	c, τ	$::=$	$\alpha \mid \tau \to \tau \mid \forall \alpha :: \kappa .\ \tau \mid \{\overrightarrow{c \mapsto \tau}\}$
			$\mid\ \lambda \alpha :: \kappa .\ c \mid c\ c \mid N \mid \dots$
Value vars	x		
Constructors	X		
Literals	ℓ		
Patterns	p	$::=$	$_ \mid x \mid \ell \mid X \mid X(p) \mid \{\overrightarrow{c = p}\}$
Expressions	e	$::=$	$\ell \mid x \mid X \mid X(e) \mid \lambda x : \tau .\ e$
			$\mid\ \Lambda \alpha :: \kappa .\ e \mid e\ e \mid e\ [c]$
			$\mid\ \{\overrightarrow{c = e}\} \mid e.c \mid \mathsf{let}\ x = e\ \mathsf{in}\ e$
			$\mid\ \mathsf{case}\ e\ \mathsf{of}\ \overrightarrow{p \Rightarrow e} \mid \dots$
Declarations	d	$::=$	$\mathsf{type}\ \alpha :: \kappa = c \mid \mathsf{val}\ x : \tau = e$
			$\mid\ \mathsf{datatype}\ \alpha(\overrightarrow{\alpha}) = \overrightarrow{X\ [\mathsf{of}\ \tau]}$
			$\mid\ \mathsf{val\ rec}\ \overrightarrow{x : \tau = e} \mid \dots$
Programs	P	$::=$	$\overrightarrow{d}$

Figure 3. Syntax of Core language

syntactic sugar for record types with consecutive natural numbers as field names.

Core also supports mutually recursive definitions of algebraic datatypes, where each type takes the form $\alpha(\overrightarrow{\alpha})$, as a type family applied to a (possibly empty) list of formal type parameters. Not shown in the figure, but included in the implementation, are polymorphic variants [13], based on a primitive type of kind $\{\mathsf{Type}\} \to \mathsf{Type}$, which provides a way of constructing a type using a finite map from names to types.

By the end of the compilation process, a program must have been reduced to using only finitely many distinct (possibly anonymous) record and variant types, which are compiled respectively to (named) C struct and union types. All name variables must have been resolved to name literals.

Mono. The next simplification is to remove all support for polymorphism or the more involved type-level features of Ur, basically just dropping the associated productions of Figure 3 to form a more restricted grammar. However, another essential change is that the language becomes **imperative**, adding a primitive function write for appending a string to the thread-local page-content buffer.

Cjr. Our penultimate language is a simplified subset of C, with some of our data-representation and memory-management conventions baked in. All record and variant types must be named by this stage.

C. Finally, we output standard C code as text, feeding it into a normal UNIX compilation toolchain, linking with a runtime system implemented directly in C.

3. Compilation Phases

Figure 4 shows the main phases of our compilation pipeline. Bold rectangles separate the different intermediate languages, named with italic text, and arrows represent transformation phases, labeled with the remaining text. Some arrows connecting to several dots are schematic, representing running several different kinds of phases, some of them multiple times, in orders that we do not bother to specify. As in many compiler projects, our intralanguage phase orderings derive from trial and error, and we do not have any great confidence that the present orderings are especially optimal.

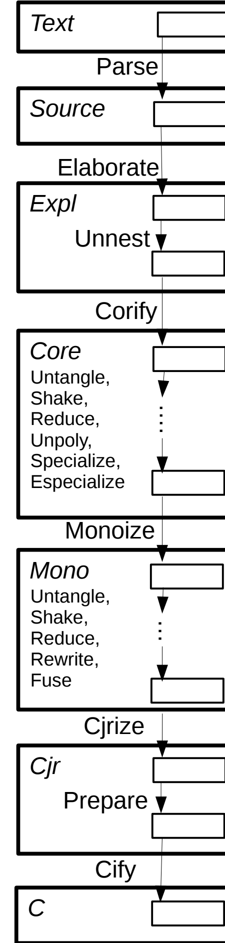

Figure 4. Ur/Web compilation phases

3.1 From Source Code to Core

The first few phases of the compiler are quite standard. The translation from Source to Expl employs the Ur type-inference algorithm [5], which is able to cope with sophisticated uses of type-level computation. One additional intralanguage phase applies to the type-annotated programs: an Unnest transformation does lambda lifting [18] to transform local function definitions into definitions of top-level functions that should be called with extra arguments, specifically those local variables referenced in function bodies. Later phases assume that all expressions of function type either now refer to top-level variables or will be eliminated by algebraic optimizations.

Actually, the positions we call "top-level" above refer to the top levels of module definitions. Expl still supports nested module definitions, and some modules may be *functors* [16], or functions from modules to modules. Following MLton, our compiler eliminates all module-system features at compile time. Basic modules are flattened into sequences of truly top-level declarations, with references to them fixed up. All functor applications are removed by inlining functor definitions. This step breaks all modularity via abstract types from the original program, facilitating optimizations that depend on, e.g., knowing the true representation of an abstract type, at a call site to a function of the module encapsulating that type. The definitions of functions themselves are also all exposed, and they will often be inlined (in later phases) into their use sites.

One unusual aspect of the translation into Core is that we must maintain enough information to build a mapping from URL prefixes to their handler functions. Module paths and function names in the original source program are used to construct a unique name for each handler (which must be top-level after the Unnest transformation), and the Corify translation must tag each new top-level function definition with its module path from the original program.

3.2 Optimizations on Core

A number of transformations are applied repeatedly to Core programs to simplify them in different ways.

Untangle finds mutually recursive function definitions where the dependencies between functions are not actually fully cyclic, expanding those mutual-definition groups into multiple distinct definitions in order. Such a transformation helps later program analyses compute more accurate answers, despite avoiding dataflow analysis and assuming conservatively that all functions in a recursive group may really call each other.

A natural source of spurious apparent recursion is functions that call each other through links in HTML pages. These links are written as Ur/Web function calls, rather than as textual URLs, allowing the compiler to understand the link structure of an application. For instance, we might write this program that mixes "real" runtime recursion with the special sort of recursion through links:

```
fun listElements ls =
  case ls of
    [] => return <xml/>
  | n :: ls' =>
    rest <- listElements ls';
    return <xml>
      <a link={showOne ls n}>{[n]}</a>,
      {rest}</xml>
and showOne ls n =
  return <xml><body>
    Viewing #{[n]};
    <a link={listElements ls}>Back to list</a>
  </body></xml>
```

One Core-level phase, which we do not detail here, is responsible for translating link calls into URLs, by serializing function arguments appropriately, etc. The result for our example is the following, where ^ is the string-concatenation operator:

```
fun listElements ls =
  case ls of
    [] => return <xml/>
  | n :: ls' =>
    rest <- listElements ls';
    return <xml>
      <a href={"/showOne/" ^ serialize(ls)
        ^ "/" ^ serialize(n)}>{[n]}</a>,
      {rest}</xml>
and showOne ls n =
  return <xml><body>
    Viewing #{[n]};
    <a href={"/listElements/"
      ^ serialize(ls)}>Back to list</a>
  </body></xml>
```

When Untangle analyzes this code, it determines that neither function calls the other directly, and only `listElements` calls itself directly. Thus, we may split this mutual definition into two separate definitions, one recursive and one not.

Shake performs tree-shaking, deleting unused definitions from programs. Each Ur/Web program has a designated primary module, and it is the properly typed functions exported by that module that the compiler must guarantee are callable via URLs. Other named functions reachable transitively from the entry points by links, etc., are also included in the final URL mapping. As a result, it is sound to remove definitions that the entry-point functions do not depend on. The next few optimizations build various specializations of type and function definitions, often leaving the originals orphaned, at which point Shake garbage-collects them.

Reduce applies algebraic simplifications in the style of the simplifier of the Glasgow Haskell Compiler [19]. Some of the most important transformations include inlining definitions, doing beta reduction for both type and value lambdas, and simplifying `case` expressions with scrutinees built with known constructors.

In general, definitions are inlined based on a size threshold, but, because later phases of the compiler require monomorphic code, nonrecursive polymorphic definitions are always inlined. For instance, starting from this program using a polymorphic pairing function

```
fun pairUp [a :: Type] (x : a) = (x, x)
val p : int * int = pairUp [int] 7
```

inlining and beta reduction (for both type and value abstraction) replace the second declaration with

```
val p : int * int = (7, 7)
```

Since thanks to the whole-program compilation model `pairUp` may be inlined at all uses, a later Shake phase will remove its definition. A similar pattern applies to functions polymorphic in record-related kinds like `Name`; inlining and partial evaluation replace their uses with operations on fixed record types.

Unpoly is the first of the phases removing polymorphism from Core programs. It replaces polymorphic function applications with calls to specialized versions of those functions. For instance, consider this program working with an ML-style `option` type family and its associated `map` function. In reality, since the function is not recursive, simple inlining by Reduce would accomplish some of the optimization that we attribute to other phases here, but we prefer this example for its simplicity.

```
datatype option a = None | Some of a

fun map [a] [b] (f : a -> b) (x : option a) =
  case x of
    None => None
  | Some x' => Some (f x')

val _ = map [int] [int] (fn n => n + 1) (Some 1)
```

The Unpoly output, after Shaking, would be:

```
datatype option a = None | Some of a

fun map' (f : int -> int) (x : option int) =
  case x of
    None => None
  | Some x' => Some (f x')

val _ = map' (fn n => n + 1) (Some 1)
```

Specialize is the next phase, which creates custom versions of algebraic datatypes, changing our example to:

```
datatype option' = None' | Some' of int

fun map' (f : int -> int) (x : option') =
  case x of
    None' => None'
```

```
  | Some' x' => Some' (f x')

val _ = map' (fn n => n + 1) (Some' 1)
```

Especialize operates in the same spirit, this time removing uses of first-class functions instead of polymorphism. We apply call-pattern specialization [17] to generate versions of functions specialized to patterns of arguments, looking especially for patterns that will remove use of first-class functions. When we identify `map' (fn n => n + 1)` as a specialized calling form for `map'` with no free variables, our running example changes into:

```
datatype option' = None' | Some' of int

fun map'' (x : option') =
  case x of
    None' => None'
  | Some' x' => Some' (x' + 1)

val _ = map'' (Some' 1)
```

More sophisticated examples identify call templates that do contain free variables, generally of function-free types. Those free variables become new parameters to the specialized functions. Such a feature is important for, e.g., calling a list map function with an argument lambda that contains free variables from the local context. Also note that type classes in Ur/Web are represented quite explicitly with first-class values standing for instances, and Especialize will often specialize a function to a particular type-class instance. For example, a generic is-element-in-list function will be specialized to the appropriate instance of the equality-testing type class.

Iterating these transformations in the right order produces a program where all types are broadly compatible with standard C data representations. We formalize that property by translation into Mono.

3.3 The Monoize Phase

Most syntax nodes translate from Core to Mono just by calling the same translation recursively on their child nodes. However, many identifiers from the Ur/Web standard library have their definitions expanded. We do not just link with their runtime implementations, like in usual separate compilation, because we want to expose opportunities for compile-time optimization. For instance, where the programmer writes an SQL query like this one:

```
SELECT t.A, t.B FROM t WHERE t.C = {[n]}
```

the parser produces desugared code like:

```
select
  (selTable [T] (selColumn [A]
    (selColumn [B] selNil)) selDone)
  (from_table t)
  (sql_eq (sql_column [T] [C]) (sql_int n))
```

Each of these standard-library identifiers must be expanded to explain it in more primitive terms. Implicit here is the type family of SQL queries, which is just expanded into the `string` type, as appropriate checking of query validity has been done in earlier stages. A simple operator like `sql_eq`, for building an SQL equality expression given two argument expressions, can be translated as:

```
fn (e1 e2 : string) => e1 ^ " = " ^ e2
```

Similarly for `select` above:

```
fn (sel from where : string) =>
  "SELECT " ^ sel ^ " FROM " ^ from
    ^ " WHERE " ^ where
```

The Mono optimizations, which run soon after Monoize, will beta-reduce applications of such anonymous functions.

Some combinators work in ways that cannot be expressed in Ur. For instance, there is no general way to convert a type-level first-class name to a string, following the System F philosophy of providing *parametricity* guarantees that limit possible behaviors of polymorphic functions [22]. However, this kind of conversion is convenient for implementing some SQL operations, like referencing a column as an expression. Monoize does ad-hoc translation of an expression like `sql_column [T] [C]` into `"T.C"`.

Mono is the first impure intermediate language, meaning it supports side effects. After Monoize, every URL entry-point of the application has an associated function that wraps a call to the primitive function `write` around an invocation of the original monadic function from the source code. Each monadic value is translated into a first-class impure function that performs the appropriate side effects. Much of the later optimization deals with simplifying particular usage patterns of `write`.

3.4 Optimizations on Mono

Mono has its own **Untangle** and **Shake** phases, exactly analogous to those for Core. The other two key optimizations finally bring us to the heart of transforming Figure 1 into Figure 2, moving SQL query code to the places where it is needed and avoiding allocation of intermediate values.

Reduce for Mono acts much like it does for Core, but with the added complexity of analyzing side effects. For example, to move an SQL query to its use point, a crucial transformation is replacing an expression like let $x = e_1$ in e_2 with $e_2[e_1/x]$. The Monoize phase composed with basic reduction will transform a monadic bind operation into exactly this kind of let, when enough is known about the structure of the operands. However, this let inlining is only sound under conditions on what side effects e_1 and e_2 may have, the position(s) of x within e_2, etc.

To allow sound let inlining, Mono Reduce applies a simple one-pass program analysis. We define a fixed, finite set of abstract effects that an expression may have: WritePage, for calling the write operation to append to the page buffer; ReadDb and WriteDb, for SQL database access (i.e., calls to distinguished standard-library functions); and ?, for any case where the analysis is too imprecise to capture an effect (for instance, for any call to a local variable of function type).

A simple recursive traversal of any expression e computes a set $\mathsf{PRE}_x(e)$ of the effects that may occur before the first use of x. Now the condition for inlining in let $x = e_1$ in e_2 is:

1. x occurs exactly once along every control-flow path in e_2, and that occurrence is not within a λ.
2. $? \notin \mathsf{PRE}_x(e_1)$.
3. If $\mathsf{WritePage} \in \mathsf{PRE}_x(e_2)$, then $\mathsf{WritePage} \notin \mathsf{PRE}_x(e_1)$. (Note that x cannot occur in e_1, so $\mathsf{PRE}_x(e_1)$ includes all effects.)
4. If $\mathsf{ReadDb} \in \mathsf{PRE}_x(e_2)$, then $\mathsf{WriteDb} \notin \mathsf{PRE}_x(e_1)$.
5. If $\mathsf{WriteDb} \in \mathsf{PRE}_x(e_2)$, then $\mathsf{WriteDb} \notin \mathsf{PRE}_x(e_1)$ and $\mathsf{ReadDb} \notin \mathsf{PRE}_x(e_1)$.

That is, different database read operations commute with each other, but database writes commute with no other database operations. Page-write operations do not commute with each other. However, either class of operation commutes with the other. This simple, sound rule is sufficient to show that the database query of Figure 1 is safe to inline to its use site, since database reads commute with page writes. (Even this simple level of reasoning is only necessary after the next optimization has run to split the code into a sequence of distinct write operations.)

```
table fortune : {Id : int, Message : string}
val new_fortune = {Id = 0, Message = "XXX"}

fun fortunes () =
  fs <- queryL1 (SELECT fortune.Id, fortune.Message
                 FROM fortune);
  return <xml>
    <head><title>Fortunes</title></head>
    <body><table>
      <tr><th>id</th><th>message</th></tr>
      {List.mapX (fn f => <xml><tr>
        <td>{[f.Id]}</td><td>{[f.Message]}</td>
      </tr></xml>)
      (List.sort
        (fn x y => x.Message > y.Message)
        (new_fortune :: fs))}
    </table></body>
  </xml>
```

Figure 5. Simplified version of Fortunes benchmark

The current Ur/Web optimizer supports only this hardcoded set of effects. It may also be worthwhile to make the effect set extensible, with hooks into the foreign function interface to describe which effects are produced by new impure primitives.

Rewrite applies simple algebraic optimization laws. The primitive combinators for building XML trees are translated by Monoize into operations that do a lot of string concatenation. In general, the page returned by a URL-handler function is a bushy tree of string concatenations $\wedge$. Thus, the most basic rewrite law is $\mathsf{write}(e_1 \wedge e_2) = (\mathsf{write}(e_1); \mathsf{write}(e_2))$. Another essential one is replacing an explicit concatenation of two string literals with a single literal, which is their compile-time concatenation.

Other important rules deal with functions like escape and stringifyInt, inserted by the SQL and XML combinators to convert data types into the proper forms for inclusion in code fragments, which are represented as strings. An application of one of these functions to a literal is evaluated at compile time into a different literal. When one of these functions is applied to a nonconstant value, there may still be optimization opportunities. For instance, we have the rewrite rule $\mathsf{write}(\mathsf{escape}(e)) = \mathsf{escape_w}(e)$, using the version of escape specialized to write its result imperatively to the page-output buffer, avoiding allocation of an intermediate string.

One other essential rule applies to writing results based on SQL queries. By this point, all queries are reduced to calls to a primitive function in the style of list folds, of the form $\mathsf{query}\ Q\ (\lambda r, a.\ e)\ a_0$. We fold over all rows r in the response to query string Q, building up an accumulator value. We may apply the rule

$$\begin{aligned} & \mathsf{write}\ (\mathsf{query}\ Q\ (\lambda r, a.\ a \wedge e)\ \texttt{""}) \\ = \ & \mathsf{query}\ Q\ (\lambda r, _.\ \mathsf{write}(e))\ \{\} \end{aligned}$$

The original expression computes a string of HTML code and then writes it to the page, while the second version writes each page chunk as it becomes ready. Applying this rule tends to create more opportunities for Rewrite transformations within the body of the fold.

Fuse is the crucial final element of Mono optimization. Its job is to push write inside calls to recursive functions. Consider the Ur/Web example of Figure 5, which is a slight simplification of one of the benchmarks we use in Section 4. This page-handler function starts by querying a list of all rows in the `fortune` database table, using the `queryL1` function. Next, it adds a new fortune to the list and sorts the new list by message text. Finally, it generates some HTML displaying the fortunes in a table, using `List.mapX` to apply an HTML-producing function to each element of a list, returning the concatenation of all the resulting fragments. Earlier phases will have produced a specialized version of `List.mapX` like the following.

```
fun mapX' ls =
  case ls of
    [] => ""
  | f :: ls' => "<tr>\n<td>" ^ stringifyInt f.Id
              ^ "</td><td>" ^ escape f.Message
              ^ "</td>\n</tr>" ^ mapX' ls
```

The Fuse optimization activates when a call to a function like this one, returning a string, is passed immediately to the `write` operation. We simply clone a new version of each such function, specialized to its context of producing page output. For our example:

```
fun mapX'_w ls =
  write (case ls of
    [] => ""
  | f :: ls' => "<tr>\n<td>" ^ stringifyInt f.Id
              ^ "</td><td>" ^ escape f.Message
              ^ "</td>\n</tr>" ^ mapX' ls)
```

We then run one pass of Rewrite simplification, resulting in:

```
fun mapX'_w ls =
  case ls of
    [] => ()
  | f :: ls' =>
    (write "<tr>\n<td>"; stringifyInt_w f.Id;
     write "</td><td>"; escape_w f.Message;
     write "</td>\n</tr>"; write (mapX' ls))
```

Notice that Rewrite has replaced calls to `stringifyInt` and `escape`, which allocate intermediate strings, with calls to the write-fused `stringifyInt_w` and `escape_w`, which produce their output directly in the page buffer. The finishing touch is to scan the simplified function body for calls to the original function, used as arguments to `write`, replacing each call with a recursive call to the new function. In our example, `write (mapX' ls)` becomes `mapX'_w ls`.

3.5 The Cjrize Phase

One of the final compilation steps is translating from Mono to Cjr, the intermediate language that is very close to the abstract syntax of C. The gap from Mono is also rather small: the main simplification is introducing a name for each distinct record type in the program, to prepare for referencing record types in C as named `struct` types. The Cjrize translation fails if any lambdas remain that are not at the beginnings of `val` or `val rec` declarations; that is, we begin enforcing that functions are only defined at the top level of a program, after earlier optimizations have removed other explicit uses of first-class functions.

3.6 Optimizations on Cjr

A crucial phase that runs on Cjr code is **Prepare**, which spots bits of SQL syntax that are constructed in regular enough ways that they can be turned into *prepared statements*, which are like statically compiled functions stored in the database engine. Using prepared statements reduces the cost of executing individual queries and database mutation operations, for much the same reasons that static compilation can improve performance of conventional programs. While Ur/Web allows arbitrary programmatic generation of (well-typed) SQL code, most queries wind up looking like simple concatenations of string literals and conversions of primitive-typed values to strings, after all of the Mono optimizations run. The Prepare phase replaces each such string concatenation with a reference to a named prepared statement, maintaining a dictionary that helps

find opportunities to reuse a prepared statement at multiple program points. Figure 2 showed a prepared statement in action for our first example. Though this transformation is conceptually very simple, it delivers one of the biggest pay-offs in practice among our optimizations, without requiring that Ur/Web programmers pay any attention to which queries may be amenable to static compilation. In fact, many queries are built using nontrivial compile-time metaprogramming [5], such that it would be quite burdensome to predict which will wind up in simple enough forms.

The final compiler phase, translation from Cjr to C, mostly operates as simple macro expansion. The interesting parts have to do with memory management. Server-side Ur/Web code runs without garbage collection, instead relying on *region-based memory management* [25], where memory allocation follows a stack discipline. The scheme is not quite as simple as traditional stack allocation with function-call activation records; the stack structure need not mirror the call structure of a program. Instead, at arbitrary points, the program may *push* a new region onto a global region stack, and every allocation operation reserves memory within the top region on the stack. A region may be *popped* from the stack, which deallocates *all* objects allocated within it, in constant time. Of course, it is important to enforce restrictions on pointers across regions, which the original work on regions [25] addressed as a nontrivial type-inference problem.

Ur/Web follows a much simpler strategy that requires no new program analysis. A key language design choice is that **all mutable state on the server is in the SQL database**. The database does its own memory management, and it is impossible for server-side Ur/Web code to involve pointers that persist across page requests. Therefore, as a simple worst-case scenario, it is always sound to place the code for each page request in its own region, freeing all its allocated objects en masse after the page is served. However, another simple strategy helps us identify profitable regions within page handlers: ***any* expression of a pointer-free type may be evaluated in its own region**, since there is no way for pointers to flow indirectly from within an expression to the rest of the code.

We define as pointer-free the usual base types like integers, records of pointer-free values, and any algebraic datatype where no constructor takes an argument (which we represent as an `enum` in C). The empty record type, the result of e.g. `write` operations, is a simple degenerate case of a pointer-free type. Thus, in the common pattern where a page handler is compiled into a series of `write` operations, every `write` can be placed in its own region. Take the example of the `write` for the `List.mapX` call in Figure 5, which would already have been optimized into a call to the `mapX'_w` function developed at the end of Section 3.4. The generated C code would look like:

```
begin_region(ctx);
mapXprime_w(ctx, sort(
  {.head = new_fortune, .tail = fs}));
end_region(ctx);
```

Here `sort` is a version of the list-sorting library function specialized to the comparison function passed to it in Figure 5. It will allocate plenty of extra list cells, not just for the sorted output list, but also for intermediate lists within the merge-sort strategy that it follows. However, the sorted list is only used to generate output via `mapXprime_w`, leading to an empty-record type for the whole expression. Therefore, the C rendering wraps it in begin- and end-region operations, so that all of the new list cells are freed immediately afterward, in constant time. The same optimization can apply inside of a larger loop (perhaps compiled from a recursive function at the source level), so that this inferred region allocation can reduce the high-water mark of the Ur/Web heap by arbitrarily much.

We note that this simple region-based memory management is not suitable for all applications. Rather, it is tuned to provide good performance for typical Web applications that avoid heavy computation, focusing instead on querying data sources and rendering the results. Many classic applications of functional programming will be poor fits for the model. For instance, the complex allocation pattern of a compiler will fall well beyond what Ur/Web's region analysis understands, and an Ur/Web program that needs to include such code is most likely best served by making a foreign function interface call to another language. Still, for the common case of Web applications, there are many benefits to the simple memory regime.

For instance, it allows us to give each server thread a private heap. An important consequence is that the C code we generate involves **no sharing of C-level objects across threads**, avoiding synchronization overhead in highly concurrent server operation. Synchronization only occurs in dividing HTTP requests across threads and in the off-the-shelf database-access libraries that we use. We also avoid the unpredictable latency of garbage collection. Ur/Web server-side code follows a *transactional* model [6], where every page-handler function call appears to execute *atomically*. To support that model, features for aborting and restarting a transaction (e.g., because the database engine reports a deadlock) are integrated throughout the compiler and runtime system. An Ur/Web server thread may run out of space in its private heap, but instead of running garbage collection in that situation, we simply allocate a new heap of twice the original size and abort and restart the transaction. Thread heaps do not shrink, so a given thread can only experience a few such restarts over its lifetime.

Another optimization avoids redundant memory allocation connected to processing results of database queries. Figure 2 showed the basic sort of code that we generate for queries, iterating over results using a cursor into query results. We simplified a bit in that figure, as our actual code generation will define an explicit record for each query result, relying on the C optimizer to inline record-field values as appropriate, leading to code more like the following for the query portion of Figure 2:

```
Cursor c = prepared(ctx, "stmt1", cat, NULL);
if (has_error(c)) error(ctx, error_msg(c));
Row r;
while (r = next_row(c)) {
  stmt1_row sr = {.id = atoi(column(r, 0)),
                  .text = uw_strdup(ctx,
                                    column(r, 1))};
  /* ...code using 'sr'... */
}
```

Notice that we conservatively duplicate the value of one column using `uw_strdup`, a function that creates a copy of a string in the thread's local heap. A string returned directly by `column` may live within a buffer that will be reused when we advance the cursor with `next_row`, so in general we have to copy to avoid unintended aliasing. However, when the query loop maintains an accumulator of pointer-free type, it is sound to skip the `uw_strdup` operations, because there is no place that the body could hide a string. Compared to inferring region boundaries, here we are even more generous, applying the no-duplication optimization whenever the accumulator type of the loop does not mention the `string` type transitively. The empty-record type associated with the loop above is more than simple enough to enable that optimization, as would be e.g. an integer recording the sum of lengths of strings returned by `column`.

3.7 Classifying Page Handlers

The compiler employs one last category of simple program analysis and optimization, providing outsized benefits where applicable.

A single application may contain page-handler functions that use different subsets of the language features, with associated runtime set-up costs for particular mixes of features. We want to avoid those costs on page views that do not use associated features.

Since ours is a whole-program compiler, we approximate the key properties in terms of *graph reachability* in a graph whose nodes are top-level identifiers, e.g. of functions, and where an edge connects identifier A to identifier B when the definition of A mentions B. Mutually recursive definitions may induce self-loops or longer cyclic paths. In this general framework, we answer a number of questions about each page handler.

Client-side scripting? A page with client-side scripting must include a reference to Ur/Web's client-side runtime system, implemented in JavaScript. We also include, with the runtime system, compiled JavaScript versions of all recursive Ur/Web functions used in client code in the particular application. When a page handler will not generate any client-side scripting, we can skip the expense of conveying the runtime system. To sniff out client-side scripting, we first add a label to any graph node whose definition contains string literals with certain red-flag contents, like `"<script"`, the HTML tag for embedded JavaScript. Then we ask, *can the node for this page handler reach any labeled identifier?*

Asynchronous message-passing? Ur/Web also supports asynchronous message-passing from server to clients [6]. This feature creates even more overhead than basic scripting, on both server and client. The server maintains a mailbox for each client that might receive messages, and the client must open a separate, long-lived HTTP connection to the server for message delivery. The question we ask, to see if we can avoid creating a mailbox for a page handler, is, *can the node for this page handler reach the standard-library function that allocates a message-passing channel?*

Read-only transactions? Highly concurrent applications rely on the underlying SQL database engine to find parallelism opportunities in the execution of concurrent transactions. The overhead of locking and so forth may be significant. If a page handler reads the database but does not change it, the SQL engine may employ less expensive concurrency management, but it needs to be warned in advance of the first read operation. Our compiler tags these page handlers so that they initiate transactions marked explicitly as read-only, based on the question, *can the node for this page handler reach the standard-library function for database modifications?*

Single-command transactions? Creating and committing a transaction each incur one extra communication with the database engine. If a page handler can only involve at most one database operation, however, transaction initialization and teardown can be combined with that single operation, if and when it is sent to the database, without breaking the transactional abstraction. To analyze the possibilities, we first label each identifier according to a simple lattice: NoDb, when its definition does not mention database functions; OneQuery, when its definition mentions a database function exactly once, not inside of a query loop (the only form of in-place iteration that Ur/Web supports); and AnyDb otherwise. We mark a page handler to skip separate transaction set-up based on the question, *does the node for this page handler have no path to any node marked* AnyDb *and have no two distinct paths to nodes marked* OneQuery*?*

Dependency on implicit context from browsers? Web browsers often send state implicitly on behalf of clients. For instance, *cookies* are a venerable means of storing key-value pairs on clients, such that owning servers may modify them, and the client always sends the latest values when contacting those servers. This sort of implicit context may enable a kind of security problem called *cross-site request forgery*. For instance, Alice may send Bob a URL labeled as "cute cat photos" but which actually points to a URL at Bob's bank standing for "transfer $1000 to Alice." Alice does not know Bob's bank credentials, so she could not get away with making the request herself. However, it is possible that Bob's browser is storing a credentials cookie that will be enough to satisfy his bank. To rule out such attacks, Ur/Web arranges to send *a cryptographic signature over all implicit context that it might read* along with each legitimate request, where only the server knows the private signing key. (Whole-program analysis makes it easy to compute an upper bound on the set of cookies that might be read.) To avoid the expense of unnecessary cryptography, we ask for each page handler, *can its node both reach some node that reads a cookie (or other source of implicit context) and reach some node that writes to the database or to a cookie?*

Table 1. Performance comparison with other Web frameworks, with Ur/Web's ranking for each statistic

Test	Throughput (kreq/s)	Latency (ms/req)
`fortune`	219 (2/103)	1.09 (2/103)
`db`	154 (7/154)	1.54 (5/154)
`query`	10 (23/151)	23.5 (40/151)
`update`	0.4 (85/102)	2050 (98/102)
`json`	387 (19/147)	0.61 (15/147)
`plaintext`	603 (21/109)	1020 (22/109)

Persistent side effects? The HTTP standard defines several different sorts of client-to-server requests, including `GET`, which should not be allowed to cause persistent side effects on the server; and `POST`, which may cause effects. Browsers will take these semantics for granted, for instance by warning the user when reloading a `POST` page, as irrevocable actions may be repeated unintentionally, but issuing no warning for a `GET` page. In Ur/Web applications, regular links produce `GET` requests, while all other requests use `POST`. To ensure compliance to the standard (with explicit escape-hatch options when the programmer intended to disobey), the compiler will warn when it computes the wrong answer to the question, *can this node, which is the target of a link, reach a node that writes to the database or to a cookie?*

4. Evaluation

As reported previously [6], Ur/Web is in production use for a few different Web applications. One of them, BazQux Reader[1], has thousands of paying customers. It is a reader for syndicated web content (e.g., RSS) with comments. Its usual load is around 10 HTTP requests/second, with busy-period peaks above 150 requests/second. It does not seem that any of the publicly deployed applications are pushing the limits of Ur/Web server performance yet, but they at least provide an existence proof for a reasonably effective compiler for a purely functional Web language based on dependent type theory.

To measure performance more, under more trying conditions, we turn now to some microbenchmarks and one application deployed internally within MIT.

4.1 The TechEmpower Web Framework Benchmarks

The TechEmpower Web Framework Benchmarks[2] compare the popular Web application frameworks for performance. They are not run by the author of this paper, and they are driven by a healthy spirit of competition among framework fans: the supporters of each framework contribute benchmark implementation code on GitHub,

[1] `http://www.bazqux.com/`

[2] `http://www.techempower.com/benchmarks/`

and poorly written implementations for popular frameworks rarely survive for long. The latest results measure 118 different frameworks, including substantially all of the most popular ones written in C, C#, C++, Haskell, Java, JavaScript, PHP, Python, Ruby, and Scala.

The current results involve 6 different benchmarks:

- `fortune`, the closest to a full Web application (and the inspiration for Figure 5): read the contents of a database table, add one new row to an in-memory representation of the contents, sort it by numeric key, and render the results as an HTML table
- `db`: make one simple query for a random row of a database table
- `query`: repeat the `db` query 20 times
- `update`: repeat the `db` query 20 times, also randomly mutating each queried row and writing it back to the database
- `json`: render a "hello world" string in the JSON serialization format
- `plaintext`: return "hello world" as plain text (this test also pushes the concurrency level higher than the others, challenging the ability of most frameworks to stay responsive)

Here we note that 4 out of 6 tests work in the relatively new style of exposing Web services whose outputs are designed to be processed by other programs, rather than seen directly by humans; and Ur/Web provides good support for that style of application. A library uses type-safe compile-time metaprogramming for parsing and generation of JSON-format strings from native values, and the same optimizations outlined above apply just as well to JSON generation as to HTML generation. For instance, while JSON generation is coded as purely functional construction of strings, the optimizations from Section 3.4 will translate to imperative code that appends to a buffer incrementally. Ur/Web also includes native support for presenting such API calls as typed function calls within client-side Ur/Web code, using Ur/Web's own serialization format that the compiler is especially adept at manipulating efficiently, though the TechEmpower Benchmarks do not test that functionality.

TechEmpower runs the tests on 3 servers, one each for Web application, SQL database, and simulated client. Each server is identical, supporting 40 simultaneous hardware threads. More information on the platform is available on the benchmarks Web site.

Here we give results for Ur/Web running with the PostgreSQL database, though a version with MySQL is also entered into the comparison. The Ur/Web implementation is compiled into a standalone HTTP server. Table 1 summarizes Ur/Web's standing on the different tests in Round 10 of the benchmarks, released on April 21, 2015. For each test, we give Ur/Web's throughput (as thousands of requests per second) and latency (as average delay in milliseconds per request). We also give Ur/Web's ranking within the entrants for each statistic.

For the `fortune` test, which is closest to a real Web application, out of 103 entrants, Ur/Web is narrowly outperformed by one other framework, implemented in C++, to score second best for both throughput (about 220,000 requests per second) and latency (about 1 millisecond to serve one request), beating out, e.g., several other implementations using C and C++.

Ur/Web is also near the head of the pack for the `db` test, involving a single SQL query; and the `query` test, involving 20 queries. In the latter case, we pay a performance penalty for Ur/Web's pervasive use of transactions [6], as Ur/Web runs extra database commands to begin and end a transaction on each request. However, the optimization from Section 3.7 allows the compiler to skip those commands for `fortune` and `db`, which run just one database command per request. Almost all other frameworks do not bother to provide transactional semantics. The performance hit for transactions is particularly severe on the `update` test, where Ur/Web is near the bottom of the rankings; though we note that this workload, with tens of simultaneous requests making 20 random database updates each, is rather unrealistic compared to most applications. Ur/Web also places respectably for the `json` and `plaintext` tests, which return constant pages without consulting the database.

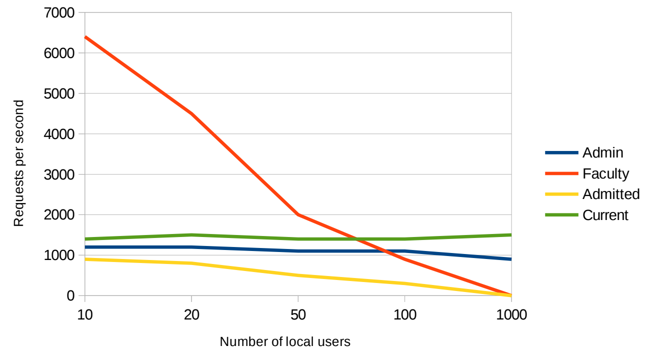

Figure 6. Performance scaling results for Visit application

The TechEmpower results are not currently highlighting any statistics related to programmer productivity, but such information seems important to put performance numbers in context. Ur/Web's main competition in the database tests are applications written in C, C++, and Java. Their corresponding source code is always at least twice as long as Ur/Web's. We also expect that fans of functional programming would feel, perhaps just subjectively, that the functional Ur/Web code is easier to understand. Readers may judge for themselves by examining the project GitHub repository[3], where the more involved tests like `fortune` are the best starting points for comparison, since they require enough code to surface interesting differences between frameworks.

Overall, we conclude from these benchmark results that **purely functional programming languages can offer extremely competitive performance in concert with domain-specific compiler optimizations**.

4.2 The PhD Visit Weekend Application

We built and deployed an Ur/Web application[4] to manage our academic department's yearly visit weekend for students admitted into our PhD program. The application includes complicated interrelated pieces of functionality. There are four classes of users:

1. **Admins**, who oversee the process, with read and write access to all data
2. **Faculty**, who arrange meetings with admitted students, sign up to give short research talks, RSVP for research-area dinners (including giving dietary constraints), and, during the visit weekend itself, see live updates of their personal schedules
3. **Admitted students**, who sign up for hotels and other travel details, indicate preferences for meeting with particular faculty, indicate when they are unavailable for meetings, RSVP for dinners, and, during the weekend itself, see live updates of their personal schedules
4. **Current students**, who RSVP for dinners

[3] E.g., the Ur/Web code in `https://github.com/TechEmpower/FrameworkBenchmarks/blob/master/frameworks/Ur/urweb/bench.ur`

[4] A version of the source code, simplified to work outside of MIT: `https://github.com/achlipala/upo/blob/master/examples/visit.ur`

A variety of bits of nontrivial functionality cut across user classes, including grids for live collaborative editing of the meeting schedule, grouped by faculty or by admitted student (maintaining consistency across the two views); and live RSVP summaries for the different dinners, listing who is going and their dietary restrictions. The application-specific code amounts to about 600 lines of Ur/Web, relying on a library we are developing for custom event-organizer applications. The server program is compiled via about 30,000 lines of generated C code, including about 100 distinct SQL prepared statements.

We benchmarked the performance of the server under increasing numbers of users. In these experiments, the application was compiled to a standalone HTTP server, while our deployment instead connects to Apache via the FastCGI protocol for easier manageability. We generated random data, parameterized on N:

- 8 different time slots for faculty-student meetings
- 10 different research-area dinners
- N different faculty or current-student users, split randomly between the two categories
- N different admitted students
- $8N$ different randomly selected faculty-student meetings

For each tested value of N, we used `wrk` (the same benchmark tool[5] from the TechEmpower Benchmarks) to repeatedly hit the main entry-point URL for each user class: admins, faculty, and admitted and current students. Each benchmark first chooses a random user in the appropriate class to authenticate as via cookies. So, each N value leads to benchmarking of 4 different URLs with appropriate cookies. We hit each URL for 5 seconds with 8 concurrent client connections, each with its own OS thread. The benchmarker, application, and PostgreSQL 9.1 database server all run on the same workstation, which has 8 1.4-GHz AMD FX cores and 32 GB of RAM. The RAM does not turn out to be a limiting factor, as the server only has 2.4 GB of resident memory after the most demanding test.

Figure 6 summarizes our throughput results for different N values from 10 to 1000. The views for admins and current students hold pretty steady near 1000 requests/second. The faculty view begins with the highest throughput of around 6500 requests/second for $N = 10$, falling to only about 10 requests/second with $N = 1000$. The admitted-student view starts near 1000 requests/second and also falls to about 10 requests/second. Parameter $N = 100$ approximates our real deployment, and there all user classes see on the order of 1000 requests/second, which is well above what we need in the field for this application. We expect that the degradation seen for higher N is mostly a function of the inherent inefficiency of constructing an $8 \times N \times N$ matrix to record the meeting schedule.

Overall, the results compare favorably to the baselines established by the TechEmpower Benchmarks, where the best throughput for a 20-query microbenchmark is about 15,000 requests/second. The Visit application makes a comparable number of queries, does much more involved processing on the results, and is being benchmarked using a total of 8 hardware threads, as opposed to 120 hardware threads in the TechEmpower Benchmarks; yet for $N = 50$ we remain within about an order of magnitude of those best-in-class results. This comparison is effectively between Ur/Web and all of the most popular Web frameworks, thanks to broad participation in the TechEmpower Benchmarks.

This application was used for MIT's 2015 visit weekend for admitted computer-science PhD students, serving an audience of about 100 faculty and about 100 admitted students. The application ran on a virtual machine provided by our laboratory, with 2 GB of RAM and 2 virtual CPU cores. Peeking at the server process at an arbitrary point, when it had been running continuously for a few days, we saw that its resident-set memory footprint was only about 10 MB. Unsurprisingly given the small set of users, there were no issues with server-side scaling. Therefore, our more modest conclusion from this experiment is that **with the right compiler support, server-side performance need not be an impediment to deploying a realistic application, with hundreds of users, written in a purely functional language based on dependent type theory**.

[5] `https://github.com/wg/wrk`

5. Related Work

SMLserver [11] supports a similar style of Web programming within the Standard ML language. It uses the ML Kit compiler, which does type inference to infer region structure, allowing some of the same memory-optimization tricks as Ur/Web employs. It also runs many server threads with their own private memory areas, to promote locality for performance and programming simplicity. There are both benefits and costs to SMLserver's embedding within a general-purpose ML implementation: it is easier to reuse existing libraries, but there are no domain-specific optimizations. As SMLserver was first described more than 10 years ago [11], it is hard to do a direct performance comparison with Ur/Web, but a direct unfair comparison of "hello world" programs shows a 2015 Ur/Web server (from the TechEmpower Benchmarks results) supporting approximately 500× better throughput than a 2003 SMLserver application (as reported in a paper [11]).

Hop [24] is another unified Web programming language, dynamically typed and based on Scheme. Hop has no distinguished database integration, instead supporting access to a variety of database systems via libraries. Hop servers support a novel means of configurable concurrency for pipeline execution [23], which has been shown to provide Web-serving performance comparable with well-known daemons like Apache. Hop also does efficient compilation of client-side code to JavaScript [15].

Links [7] is a pioneering language that introduced the tierless style that Ur/Web also adopts. Rather than exposing SQL directly, the Links implementation compiles more Haskell-style idiomatic query-comprehension code into SQL, possibly with multiple queries from a single comprehension. A static type system [4] characterizes which queries are susceptible to normalization via rewrite rules, such that a single SQL query always results.

A variety of other practical Web application frameworks have been built around functional languages, including the PLT Scheme Web Server [14], Seaside [9], and Ocsigen [1, 2]. Several Haskell and Scala frameworks, and one Racket framework, are included in the TechEmpower Benchmarks and thus covered by Table 1. To our knowledge, Ur/Web's implementation is the first application of whole-program optimizing compilation to a domain-specific Web language.

The MLton optimizing compiler [29] has been our inspiration in designing our compiler. Ur/Web's compiler goes further than MLton in generating low-level code that does not use garbage collection, while also simplifying many phases by avoiding dataflow analysis in favor of algebraic rewriting. Orthogonally, we also apply domain-specific optimizations connected to HTML generation and SQL interaction, producing code that outperforms most C and C++ implementations in the TechEmpower Benchmarks.

Our compiler applies many other established ideas from optimization of functional programs. The general technique of deforestation [26] inspires our Fuse optimization, which admits a substantially simpler implementation because it is specialized to a particular common case. The low-level memory management strategy is inspired by region type systems [25] but requires much less sophisticated compile-time analysis, thanks in part to Ur/Web's un-

usual combination of purity with an SQL database. Our compiler applies lambda lifting [18], call-pattern specialization [17], and algebraic simplification [19] in much the way made popular by the Glasgow Haskell Compiler.

6. Conclusion

A fundamental tension in the design of programming languages is between the convenience of high-level abstractions and the performance of low-level code. Optimizing compilers help bring the best of both worlds, and in this paper we have shown how an optimizing compiler can provide very good server-side performance for dynamic Web applications compiled from a very high-level functional language, Ur/Web, based on dependent type theory. Some of our optimization techniques are domain-agnostic, as in compile-time elimination of higher-order features. Other crucial techniques are domain-specific, as in understanding generation of HTML pages, database queries, and their effect interactions. With all these features combined, the Ur/Web compiler produces servers that outperform all (or almost all) of the most popular Web frameworks on key microbenchmarks, and Ur/Web has also been used successfully in deployed applications with up to thousands of users. The techniques we suggest are simple enough to reimplement routinely for a variety of related domain-specific functional languages.

One Achilles heel of whole-program compilers, and certainly of ours, is poor compile-time performance. For instance, our Visit application takes about 30 seconds to compile for production use. In a sense, Ur/Web is shifting performance costs from runtime to compile-time, which is often the right trade-off; but we still hope to improve compilation performance. We plan to study domain-specific languages for compilation that make it easy to develop new implementations like ours, doing whole-program analysis of the suite of algebraic transformations and optimizations to find ways to fuse them together and otherwise avoid overheads.

Acknowledgments

This work has been supported in part by National Science Foundation grant CCF-1217501. The author thanks all the contributors to the Ur/Web implementation and the TechEmpower Benchmarks, whose efforts supported the expansive performance comparison in this paper. Special thanks are due to Brian Hauer, Mike Smith, and Hamilton Turner, for their efforts in management and implementation for the benchmarks project and in helping debug issues with the Ur/Web benchmark implementations; and to Eric Easley for discovering that benchmarks project and providing its first Ur/Web implementation.

References

[1] V. Balat. Ocsigen: typing Web interaction with Objective Caml. In *Proc. ML Workshop*, 2006.

[2] V. Balat, J. Vouillon, and B. Yakobowski. Experience report: Ocsigen, a Web programming framework. In *Proc. ICFP*, pages 311–316. ACM, 2009.

[3] H. Cejtin, S. Jagannathan, and S. Weeks. Flow-directed closure conversion for typed languages. In *Proc. ESOP*, pages 56–71. Springer-Verlag, 2000.

[4] J. Cheney, S. Lindley, and P. Wadler. A practical theory of language-integrated query. In *Proc. ICFP*, pages 403–416. ACM, 2013.

[5] A. Chlipala. Ur: Statically-typed metaprogramming with type-level record computation. In *Proc. PLDI*, pages 122–133. ACM, 2010.

[6] A. Chlipala. Ur/Web: A simple model for programming the Web. In *Proc. POPL*, pages 153–165. ACM, 2015.

[7] E. Cooper, S. Lindley, P. Wadler, and J. Yallop. Links: Web programming without tiers. In *Proc. FMCO*, pages 266–296, 2006.

[8] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: From lists to streams to nothing at all. In *Proc. ICFP*, pages 315–326. ACM, 2007.

[9] S. Ducasse, A. Lienhard, and L. Renggli. Seaside – a multiple control flow Web application framework. In *European Smalltalk User Group – Research Track*, 2004.

[10] C. Elliott and P. Hudak. Functional reactive animation. In *Proc. ICFP*, pages 263–273, 1997.

[11] M. Elsman and N. Hallenberg. Web programming with SMLserver. In *Proc. PADL*, pages 74–91. Springer-Verlag, January 2003.

[12] M. Fluet and S. Weeks. Contification using dominators. In *Proc. ICFP*, pages 2–13. ACM, 2001.

[13] J. Garrigue. Code reuse through polymorphic variants. In *Proc. Workshop on Foundations of Software Engineering*, 2000.

[14] S. Krishnamurthi, P. W. Hopkins, J. McCarthy, P. T. Graunke, G. Pettyjohn, and M. Felleisen. Implementation and use of the PLT Scheme Web Server. *Higher Order Symbol. Comput.*, 20(4):431–460, 2007.

[15] F. Loitsch and M. Serrano. Hop client-side compilation. In *Proc. TFL*, 2007.

[16] D. MacQueen. Modules for Standard ML. In *Proc. LFP*, pages 198–207. ACM, 1984.

[17] S. Peyton Jones. Call-pattern specialisation for Haskell programs. In *Proc. ICFP*, pages 327–337. ACM, 2007.

[18] S. L. Peyton Jones and D. Lester. A modular fully-lazy lambda lifter in Haskell. *Softw. Pract. Exper.*, 21(5):479–506, May 1991.

[19] S. L. Peyton Jones and A. L. M. Santos. A transformation-based optimiser for Haskell. *Sci. Comput. Program.*, 32(1-3):3–47, Sept. 1998.

[20] S. L. Peyton Jones and P. Wadler. Imperative functional programming. In *Proc. POPL*, pages 71–84. ACM, 1993.

[21] B. C. Pierce. Higher-order polymorphism. In *Types and Programming Languages*, chapter 30. MIT Press, 2002.

[22] J. C. Reynolds. Types, abstraction and parametric polymorphism. In *IFIP Congress*, pages 513–523, 1983.

[23] M. Serrano. Hop, a fast server for the diffuse web. In *Proc. COORDINATION*, pages 1–26. Springer-Verlag, 2009.

[24] M. Serrano, E. Gallesio, and F. Loitsch. Hop, a language for programming the Web 2.0. In *Proc. DLS*, 2006.

[25] M. Tofte and J.-P. Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, Feb. 1997.

[26] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *Proc. ESOP*, pages 344–358. Springer-Verlag, 1988.

[27] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc. POPL*, pages 60–76. ACM, 1989.

[28] M. Wand. Type inference for record concatenation and multiple inheritance. *Information and Computation*, 93(1), 1991.

[29] S. Weeks. Whole-program compilation in MLton. In *Proc. ML Workshop*, pages 1–1. ACM, 2006.

[30] L. Ziarek, S. Weeks, and S. Jagannathan. Flattening tuples in an SSA intermediate representation. *Higher Order Symbol. Comput.*, 21(3): 333–358, Sept. 2008.