Scrap Your Web Application Boilerplate

or Metaprogramming with Row Types

Adam Chlipala
University of California, Berkeley
adamc@cs.berkeley.edu

Abstract

I introduce a new functional programming language, called Laconic/Web, for rapid development of web applications. Its strong static type system guarantees that entire sequences of interaction with these applications "can't go wrong." Moreover, a higher-order dependent type system is used to enable statically-checked metaprogramming. In contrast to most dependently-typed programming languages, Laconic/Web can be used by programmers with no knowledge of proof theory. Instead, more expert developers develop libraries that extend the Laconic/Web type checker with type rewrite rules that have proofs of soundness. I compare Laconic/Web against Ruby on Rails, the most well-known representative of a popular class of web application frameworks based around dynamic languages and runtime reflection, and show that my approach leads both to more concise programs and to better runtime efficiency.

1. Introduction

A pervasive flaw in computer programs, considered across most all domains, is the repetition of the same "boilerplate" code in many places. This boilerplate expresses "implementation details" that are usually irrelevant to what the programmer thinks of as his problem statement and its solution. When someone new comes along and tries to understand a boilerplate-heavy program, he needs to do some mental decompilation to extract his predecessor's original intent. In a way, the history of programming language design is the history of finding better ways to let programmers write more exactly what it is that they really mean, through the use of well-chosen abstraction mechanisms.

The domain of web application development is particularly interesting in this light. Web applications ("web apps" for short) are programs that live on a World-Wide-Web server, processing requests for URLs by returning HTML pages and other documents. Historically, they were often uniformly called "CGI scripts," despite the fact that CGI is just one possible protocol for interfacing web applications with web servers, and the fact that these page generation programs can as easily be compiled as interpreted. The social practice of web app development has a number of characteristics that make it a good proving ground for efforts to raise the level of abstraction in programming:

- There is an amazing amount of similarity among web apps being developed today. As researchers in programming languages, we're used to writing fundamentally new programs with novel algorithms on a regular basis, yet most of traditional web app development involves doing the same thing over and over again. A good portion of all of these applications serve mostly as graphical interfaces to relational databases, interacting with relatively small amounts of truly novel code. The common patterns are mostly informal, but we should seek to design programming languages that make it possible to reify them.
- Web app development is perhaps the most common kind of programming done today by people who think of themselves as programmers. These applications are big business, from high-volume electronic commerce sites, to intranet sites used by handfuls of people, to personal information management systems with single users. An improvement to web programmer productivity can have a massive combined effect.
- Last but not least, a web app is executed on a small set of Internet servers but accessible to users around the world through a standardized interface. The server administrators have the freedom to choose programming languages and tools without worrying about compatibility with client environments. This is quite attractive for language research, because we have a much better chance than usual at getting our ideas adopted "in the real world."

In this paper, I will suggest a general approach to ameliorating dependence on boilerplate code, along with an instantiation of that approach to web development and a prototype implementation for it. The idea is based around a family of dependently-typed functional programming languages called Laconic, instantiated to web apps as Laconic/Web. "Design patterns" are reified as parameterized "metaprograms" that produce code. In contrast to macro-style approaches that involve the programmatic construction of abstract syntax trees, I use a higher-order language allowing computation over types. Compilation involves partially evaluating applications of these metaprograms.

Laconic has a strong static type system that guarantees that any legal application of a metaprogram partial-evaluates to a type-correct program. For Laconic/Web, beyond the usual properties, type safety guarantees that all SQL queries and generated HTML will be legal and free of code injection vulnerabilities, all intra-application links will be valid, and the generation of HTML form fields is properly matched up with the code to use the field values. An alternate statement is that Laconic/Web provides the traditional "well-typed programs can't go wrong" guarantee, but in a setting where we consider an execution of the program to consist of all of its interactions with an unlimited set of users over time, even if those interactions span many executions of the program proper.

Providing these guarantees is quite non-trivial. In my setting, it involves dependent type checking and type inference that share the main challenges of mechanized theorem checking and proving in general. The Laconic type checker must find proofs that metaprograms can only generate programs of the proper types. At the same time, we want to avoid asking programmers to construct explicit proof objects, especially if the language is to be widely adopted. I have taken a middle ground where Laconic programmers effectively extend the type checker with type rewrite rules and proofs of their correctness. Authors of metaprogram libraries write a handful of these rules and proofs and include them in their libraries. This makes it possible for client developers to use these libraries without ever seeing a formal proof. Experiments with my implementation show that this approach really is sufficient to support the development of real applications.

A recurring idea in the design of Laconic/Web is a reliance on a higher-order analogue of “row types” or “extensible records.” I use these at the type level to represent and compute over such objects as records of types, which are used to describe the schemas of relational databases, the reified field environments introduced by HTML forms, and more. The type language is a lambda calculus of its own where the evaluation of every term terminates. Thanks to the central use of extensible records, it is sufficient to allow recursive definitions in types only through a record fold operation.

I’ll begin by introducing the core of Laconic/Web in the next section, and I introduce the Laconic approach to metaprogramming in Section 3. I summarize the issues involved in typechecking Laconic/Web in Section 4 and formalize the core of the language and its type system in Section 5. In Section 6, I describe my prototype implementation and the results of a case study comparing against the Ruby on Rails web application framework. Section 7 surveys related work.

2. The Basics of Laconic/Web

The foundation of Laconic/Web is a sub-language with a first-order type system, based on ideas pioneered in other web programming language projects (which I survey in Section 7), though I often make different syntactic choices. The interesting new elements are related to support for “metaprogramming,” which inspires some modifications to this first-order type system so that it is more amenable to automated reasoning. I start by setting the stage in this section with a description of this first-order foundation, leading into an introduction to metaprogramming in the next section.

In the general spirit of statically-typed functional programming, the design at this level follows two main principles.

First, avoid implicit constraints between different pieces of an application. This idea most commonly takes the form of assigning types to functions and checking that these functions are passed properly typed arguments. The world of web programming provides us with a wealth of additional dependencies to worry about, including the connection between a relational database schema and an application’s view of it, the connections between sections of an application (represented by hyperlinks), and the connection between an HTML form and the server-side function that is meant to process its results. In the spirit of strong static type systems, I’ve chosen to design Laconic/Web to validate all of these connections statically.

The second principle is to reify high-level ideas into single programming language objects whenever possible. By doing this, we win the ability to use libraries of higher-order functions and combinators to express complicated ideas very succinctly. A common paradigm in mainstream web app development is the Model-View-Controller design pattern, which uses an object-oriented separation of data handling, presentation, and core processing code into separate objects, often in separate files or otherwise separated textually. This has some modularity advantages, but I believe that its benefits are outweighed by the potential of metaprogramming. I chose to reify entire applications and all of their major pieces as single values whose types express their roles. This makes it possible to describe metaprogramming with functions of higher-order type, in stark contrast to the ad-hoc code generation popular in object-oriented web frameworks today.

```
val rec loop = fn count : integer =>
   fn inputs : {Name : text, Color : text} =>
   <html><body>
      This is iteration number
         <integer value={count}/>.<br/>
      The current name is
         <text value={#Name inputs}/>.<br/>
      The current color is
         <text value={#Color inputs}/>.<br/>

      <form>
         Next name: <input{#Name}/><br/>
         Next color: <input{#Color}/><br/>
         <submit handler={loop (count+1)}/>
      </form>
   </body></html>

val main = fn () => loop 1
   {Name = "Anonymous", Color = "blue"}
```

Figure 1. A basic Laconic/Web program demonstrating use of closures to track state

Without further ado, I’ll introduce the core language with two short examples. The first example program, in Figure 1, implements a contrived interaction where a web site visitor is prompted repeatedly to enter a name and a color in form fields. Each time he submits this form, the application echoes his entries back to him. Independently, it tracks how many times the form has been submitted, displaying this count on each new page.

The first thing to notice about the syntax is that, so far, it follows Standard ML, with an extension for building XML documents (HTML in this example). The `main` value is a function from the unit type to HTML documents; this is the standard type required of the value that represents a web app. `main` proceeds by calling the `loop` function, which implements the main interaction loop, seeding it with a starting count of 1 and a starting name and color. The latter two are passed separately in a record, for reasons that should become clear soon.

The `loop` function uses the special embedded XML syntax to describe an HTML document. It displays the required information using special `integer` and `text` tags. For these uses, we give values of XML attributes between curly braces instead of double quotes to “escape out of” the XML view and embed evaluations of expressions from the core language. It’s worth noting that these tags do the work of avoiding code injection attacks. Laconic/Web doesn’t provide a way to coerce arbitrary strings into values of XML types.

Next we have the most interesting part, the form. Each `input` tag defines a textbox for user input. In Laconic/Web, tags can take type arguments as well as expression arguments. Here the arguments `#Name` and `#Color` give the names for these textboxes to use in forming a reified *environment record*. That is, `input` tags are binding constructs, and the current binding environment is represented explicitly with a typed Laconic value, which will turn out to be important for enabling statically-checked metaprogramming.

The `submit` tag closes out the form by associating a server-side action with a button presented to the user. Pressing the button triggers a call to `loop`, with the first argument given explicitly as `count+1`. Notice that this means that we are using closures, stored on the client side, to track state. The second argument to `loop` is passed implicitly; it is the reified environment generated based on the `input` tags.

```
dbtable Person = {Id : integer, Name : text,
                  Height : integer}

val main = fn () => <html><body>
   <table>
      <tr><th>Name</th> <th>Height</th></tr>

      {(SELECT Person.Name, Person.Height
           FROM Person)
       (fn row : {Person.Name : text,
                  Person.Height : integer} =>
          fn document => <table>
             <tr>
                <td><text
                   value={#Person.Name row}/></td>
                <td><integer
                   value={#Person.Height row}/></td>
             </tr>
             {document}
          </table>)
       <table></table>}
   </table>
</body></html>
```

Figure 2. A Laconic/Web program demonstrating an SQL query

As a final note for this example, I'll point out that it's easy for the Laconic/Web compiler to prove that page generation always terminates for this application. In fact, besides folding over SQL query results, which I introduce in the next example, the only runtime recursion or iteration allowed by Laconic/Web is through `val rec` mutually recursive definitions where every cycle in the call graph is broken through a case where a recursive reference is only found in a `handler` attribute. Handlers are only called after user input, so this guarantees termination of individual page request responses.

The next example, in Figure 2, shows how to interact with a relational database using statically-checked SQL. This application produces a table of information on a set of people. Each person is represented by a row of a relational database table, `Person`. The columns of `Person` are unique integer identifiers `Id` and people's textual names and integer heights. Notice that the program text includes the schema of the database that it will interact with. When the compiled code starts up, it performs an analogue of dynamic linking where it connects to the database and verifies that the tables it expects exist with the proper types. Tables that don't exist are created automatically, while those that exist but have the wrong types trigger "link-time errors." All of the program's possible SQL queries (actually, query templates or "prepared statements" that retain "holes" to be filled with values of primitive data types) are also compiled and optimized once and for all during this linking phase.

The `main` function generates an HTML document containing the table we want. An arbitrary expression with the right XML tags type is spliced into the body by enclosing it in curly braces. In this case, we splice in the result of folding over an SQL query. The `SELECT` expression is an embedded SQL query that produces a relation by projecting out the `Name` and `Height` fields from the `Person` relation. The type of an SQL query expression is like the type of a standard list fold function, though here we avoid actually materializing a list. The elements of the (conceptual) list are records of the tuples of the relation returned by the query.

Here, we use the fold to build an HTML fragment. The first argument to the query is the recursive case of the fold. It takes as input the data for the next tuple returned by the query and the HTML fragment that we've built up so far. The fragment is updated by adding another HTML table row before it, containing the name and height of a person. The old fragment is spliced in with nested curly braces. The base case of the fold is just an empty fragment.

```
val main = interaction [[Name : (), Color : ()]]
   {Name = "name", Color = "color"}
   {Name = "Anonymous", Color = "blue"}
```

Figure 3. Metaprogramming version of Figure 1

```
con allText =
   fold (fn row :: {Unit} => {Type})
   (fn name :: FieldName =>
      fn u :: Unit =>
      fn tail :: {Unit} =>
      fn acc :: {Type} =>
      [name : text] ++ acc)
   []

val interaction : fields :: {Unit}
   -> $(allText fields) (* Display names for fields *)
   -> $(allText fields) (* Initial field values *)
   -> unit -> htmlTags
```

Figure 4. Signature of library code for first example

3. Introduction to Metaprogramming

The examples from the last section were small and manageable, but most web app development involves a great many bits of code like these. There tend not to be too many fundamentally different varieties of code in an application or the range of applications in a particular domain, so we'd like to do better and avoid rewriting the same boilerplate each time. As I've alluded to, Laconic language features with the flavor of metaprogramming make this possible. In this section, I'll revisit the previous examples and show how they can be implemented more effectively with metaprogramming. I defer a discussion of static type checking of metaprograms to the next section.

3.1 Computing with Row Types

We can consider a whole class of web applications like the one in Figure 1 that continuously prompt the user for a set of textual input fields and echo the values back to him. Rather than keeping our conception of this class informal, we can reify it with a Laconic/Web metaprogram. Figure 3 shows a new implementation that relies on just such a library metaprogram, called `interaction`.

The new, short program consists of a single call to `interaction`. The arguments are a list of fields to query the user about, a record of display names for those fields, and a record of initial field values.

The first argument is a type, as shown syntactically by its inclusion between an extra pair of brackets. Actually, it's not a type, but a *constructor*. I follow a standard convention for higher-order type systems, where *expressions* are classified statically by *types*, *types* are a subset of the *constructors*, and *kinds* classify constructors statically. The kind of `interaction`'s first argument is `{Unit}`, indicating a record of named constructors of kind `Unit`, which is the trivial kind whose only constructor is `()`. Since constructors of `Unit` kind carry no information, `interaction`'s first argument is effectively a list of field names.

Figure 4 shows the type signature of the library. Throughout Laconic, I follow the convention that a single colon (after a function formal parameter and elsewhere) gives the type of an expression, while a double colon gives the kind of a constructor. The first element of this signature is a constructor `allText` of kind `{Unit} -> {Type}`; that is, a constructor-level function from lists of field names to records of types. `allText` produces a record that keeps the field names from its input and gives each field type `text`. It's implemented using the constructor-level `fold` operation, which is a dependently-typed iterator over record constructors.

Its first argument is a type schema written with function notation, giving the relationship between the constructor to fold over and the kind of the result constructor. Next come the recursive and base cases for the fold. The recursive case is a constructor function having as its respective arguments the name of the current field, the value of the current field, the suffix (tail) of the record that has already been processed, and the current accumulator. In the implementation of `allText`, the recursive case adds the current field name with type `text` to the start of the accumulator, and the base case is simply the empty record.

`allText` is purely a definitional abbreviation, but it is convenient for expressing the type of `interaction`. We can see that its first argument is a list of field names, based on the `x :: K -> T` notation. This indicates a function taking as input a constructor of kind `K` and returning an expression of type `T`, where the variable `x` is bound in `T` and refers to the argument. The next two arguments of `interaction` have type `$(allText fields)`. `$` is an operation of kind `{Type} -> Type`; that is, it builds a type from a record of types. The values of type `$(allText fields)` are records having the field names from `fields` and where each field has type `text`.

Figure 5 shows the implementation of the library. It starts with the same definitional abbreviation `allText`. Next we have the definition of `interaction`, which follows the same basic structure as the original code in Figure 1, but parameterized over the list `fields`.

The generalizations are localized at the two places in the recursive definition of `loop` that depend on the fields. We splice in the results of folds for the echoing of the previous field values and the construction of the input textboxes to query for the next values.

The first fold builds the listing of the previous values. This time we are folding to produce a result at the expression level, but the fold syntax is almost identical to the constructor case. Instantiating the type schema, we see that this fold produces a function of type `{Names : $(allText fields), Inputs : $(allText fields)} -> bodyTags`; that is, given names and prior inputs for all of the fields, we get back a chunk of HTML that is suitable for use in a document's body. The fold is applied to `fields` and an appropriate record of these values to produce the actual HTML. Notice that some non-trivial record subtyping is needed to typecheck the fold.

The next fold builds the inputs of the form. Giving a sense of the meaning of its result type requires a brief summary of how XML typing works in Laconic/Web. Each XML tag that allows different child tags than its parent is assigned a constructor of kind `{Type} -> {Type} -> Type`. For some tag `foo`, a value of type `foo in out` represents a legal child of `foo` that has inputs `in` and outputs `out`. More precisely, `in` is a record of the environment fields consumed by `handler` attributes of `submit`s and some other tags, and `out` is a record of the environment produced by `input`s in this tag and its children. We generalize from single XML tags to blocks of tags using the constructor `tags` of kind `({Type} -> {Type} -> Type) -> {Type} -> {Type} -> Type`.

No doubt the library implementation stands out as involved and complicated compared to the original application code from Figure 1. Nonetheless, we see a key benefit of the Laconic approach: the client code in Figure 3 can be written without any deep understanding of dependent types. Furthermore, even the implementation avoids the manipulation of explicit proof objects, which often show up in expressive dependently-typed languages.

```
dbtable Person = {Id : idField,
                  Name : textField["Name"],
                  Height : intField["Height"]}

val main = tabulate[@Person]
```

Figure 6. Metaprogramming version of Figure 2

```
con tabulateField = fn t :: Type =>
   {Name : text,
    Display : t -> htmlTags}

val intFieldBody : text -> tabulateField integer
val textFieldBody : text -> tabulateField text

con intField = fn name : text =>
   pack integer
      with intFieldBody name
      as tabulateField
   end

con idField = intField["ID"]

con textField = fn name : text =>
   pack text
      with textFieldBody name
      as tabulateField
   end

val tabulate : fields ::: {Exists tabulateField}
   -> tab :: Table([Id : idField] ++ fields)
   -> unit -> htmlTags
```

Figure 7. Signature of library code for second example

3.2 Existential Kinds

Now let's reconsider the example in Figure 2 and look at a new implementation based on a library for tabular HTML display of SQL table contents. Figure 6 gives the new implementation.

First, let's take an informal look at what this code says. The SQL table definition from Figure 2 is modified slightly: besides a type, columns are assigned constructors that also specify their human-readable names and how to render their values as HTML. I'll get to exactly what is going on there shortly, but the constructors `idField`, `textField`, and `intField` do indeed carry this additional information. Since the table itself now carries in its type all of the information needed for rendering, we build `main` with a single easy application of a metaprogram `tabulate` to the table name. We can consider many metrics for programming language expressivity, but I believe that this example program demonstrates a case where Laconic/Web makes it possible to write a program by writing no more than exactly what it is you want.

The basic feature behind making this work smoothly is *existential kinds*. These are the natural generalization of existential types [Pie02]; an existential kind describes a constructor-level package of a constructor and an expression. In the setting of Laconic, this means that the contents of all existential constructors

```
con allText =
   fold (fn row :: {Unit} => {Type})
   (fn name :: FieldName =>
      fn u :: Unit =>
      fn tail :: {Unit} =>
      fn acc :: {Type} =>
      [name : text] ++ acc)
   []

val interaction = fn fields :: {Unit} =>
   fn names : $(allText fields) =>
   fn defaults : $(allText fields) =>
let
   val rec loop = fn count : integer =>
      fn inputs : $(allText fields) =>
      <html><body>
         This is iteration number <integer value={count}/>.<br/>
         {fold (fn r :: {Unit} => {Names : $(allText r), Inputs : $(allText r)} -> bodyTags)
            (fn name :: FieldName =>
               fn u :: Unit =>
               fn tail :: {Unit} =>
               fn acc : ({Names : $(allText tail), Inputs : $(allText tail)} -> bodyTags) =>
                  fn fieldInfo : {Names : $([name : text] ++ allText tail),
                                  Inputs : $([name : text] ++ allText tail)} => <body>
                     The current <text value={#name (#Names fieldInfo)}/> is
                        <text value={#name (#Inputs fieldInfo)}/>.<br/>
                     {acc fieldInfo}
                  </body>)
            (fn fieldInfo : {Names : unit, Inputs : unit} => <body></body>)
            [fields] {Names = names, Inputs = inputs}}

         <form>
            {fold (fn r :: {Unit} => tags form [] (allText r))
               (fn name :: FieldName =>
                  fn u :: Unit =>
                  fn tail :: {Unit} =>
                  fn acc : tags form [] (allText tail) => <form>
                     Next name: <input{name}/><br/>
                     {acc}
                  </form>)
               <form></form>
               [fields]}

            <submit handler={loop (count+1)}/>
         </form>
      </body></html>
in
   fn () => loop 1 defaults
end
```

Figure 5. Implementation of library code for first example

(constructors that have existential kinds) will be *resolved at compile time*, during partial evaluation. Each field of the table in our example is described with an existential package of the type for representing values in the field, a human-readable name for that field, and a function for formatting values of the field for HTML display. The function's type is dependent on the type that is the first component of the package.

Figure 7 illustrates more detail. It gives the signature of the library used in Figure 6. The shape of our existential packages is described with the higher-kind constructor `tabulateField`. The library implementation provides two functions `intFieldBody` and `textFieldBody`, which map human-readable field names to realizations of `tabulateField` for `integer` and `text`, respectively. `intField`, `idField`, and `textField` are particular packagings of this functionality. For instance, the `pack` syntax for `intField` says that we are building a package with `integer` for its constructor component, `intFieldBody name` for its expression component, and `tabulateField` describing the particular kind of "information hiding" to use. Any application of `intField` to a name has kind `Exists tabulateField`.

Next, we have the type of the `tabulate` function, which returns an HTML page generator when passed two arguments. The first is a record of existential packages, and the second is a table whose fields match up with that record, plus an additional unique identifier field `Id`. The first argument is declared with a triple colon, which indicates that that argument is *implicit*; like type arguments in ML, it is inferred and needn't be passed explicitly. In the example application of `tabulate`, `fields` is inferred easily by looking up the type of the `Person` table, "crossing off" the `Id` field, and keeping the remaining fields as the value of `fields`.

Figure 8 gives the library implementation. It demonstrates three new elements that are worth explaining.

First, we have `unpack` expressions, which are the counterparts to the `pack` constructors seen in Figure 7. There are two forms: `unpack con`, which unpacks a constructor; and plain `unpack`, which unpacks a constructor and a value of its type simultaneously, preserving the typing relationship between them. "`unpack con t as t2 with r in ... end`" binds `t2` as a name for the constructor in package `t` and `r` as a name for the expression in the package. Plain `unpack` takes an expression of an existential type as its first argument instead and also has an extra position for binding a new version of that expression known to have the packaged type.

The SQL query uses the notation `T.*fields` to denote selecting every field named in the record `fields` from the table `T`. This is not standard SQL syntax, and indeed SQL has no concept of first-class field lists. This syntax is necessary to support metaprogramming. In fact, the SQL syntax is just syntactic sugar for a form where the list of fields to project out of a relation is a `{Type}`, a record of types, denoted in the usual way.

The SQL query also binds a local name `T` for the unknown table `tab`. I elected to require known names for all relations involved in a query, since this makes it easier to do static checking for naming conflicts. The constructor function `enter` implements some of the associated functionality: modifying every field name of a record to reflect that it belongs to a particular table. In general, reflecting table information in record field names is important because a query may even join two copies of the same relation, which would lead to duplicate fields.

4. Static Type Checking

All of the features whose usage I've sketched in the last section must be validated with static type checking. The algorithmic challenges here go beyond even those associated with most research type systems. The type checking problem starts to look like general theorem proving, especially the kind based on constructive type theory as found in Coq [BC04] and similar systems. Simple syntax-directed or unification-based algorithms are insufficient to discharge obligations which may require inductive reasoning about operations on arbitrary records.

In the rest of this section, I'll discuss the major techniques of Laconic type-checking, to be followed in the next section with a formalization for the core of the language.

4.1 Basic Reduction Rules

The basic judgment involved in Laconic type-checking is subtyping. Simple syntactic techniques are used to determine a type for an expression. To compare this type against, for instance, the domain of a function, it's necessary to normalize both types. This can involve an arbitrary amount of lambda calculus-style computation via reduction rules.

Laconic is modeled closely after Coq, and the same reduction rules determine their definitional equality judgments. While Coq requires explicit coercions to take advantage of known equality facts in type-checking, I've designed Laconic to be more convenient to use. Since Laconic is meant to deal with much narrower domains than the formalization of mathematics, it turns out to be possible to automate the important equality reasoning in a way that works in practice.

The basic foundation shared with the Coq type checker is the use of the definitional equality reduction rules: β-reduction, which is the usual rule for simplifying applications of function abstractions; δ-reduction, which expands named definitions; and ι-reduction, which simplifies uses of the recursion principles of inductive types. For Laconic/Web, the only relevant inductive types are records and existential packages, and the respective "recursion principles" are the `fold` and unpacking constructors.

As an example, let's say that we are checking an application of a function `f` with type `fs :: {Unit} -> $(allText fs) -> htmlTags`, where `allText` is the constructor defined in Figure 4. The type directly apparent for the application of `f` to the record `[A : (), B : ()]` is `$(allText [A : (), B : ()]) -> htmlTags`. Say that the next argument is `{A = "1", B = "2"}`, whose type is `$[A : text, B : text]`. This is equivalent to the expected argument type, but it's not immediately apparent syntactically. We normalize the formal argument type into this form by: first, using a δ-reduction to expand the definition of `allText`; and then applying a sequence of ι- and β-reductions to simplify `fold` constructors.

4.2 Record Subtyping

For programming convenience, Laconic/Web supports traditional width and depth subtyping of records [Pie02]. However, type inference is more interesting than usual because it must deal with non-constant records. We may need to compare constructors of record kind that involve multiple constructor variables standing for unknown records, combined with the concatenation operator `++`, which is commutative and associative.

The client code in Figure 6 provides a concrete example. The function `tabulate` takes an implicit `{Exists tabulateField}` parameter, and its value must be determined through unification of record constructors, as sketched in Section 3.2.

4.3 Inductive Reasoning

Even if we forget subtyping for now and concentrate on type equality, there are some thorny issues left to consider. The definitional equality relation based on reductions from Section 4.1 is precisely what we mean by equality, and, happily enough, it is decidable. The catch is that it is decidable for fully determined terms only; when constructors have variables in them, equality becomes undecidable.

```
con tabulateField = fn t :: Type => {Name : text,
                                     Display : t -> htmlTags}

val intFieldBody = fn name : text => {Name = name,
                                      Display = fn n => <html><integer value={n}/></html>}

con intField = fn name : text => pack integer
                                    with intFieldBody name
                                    as tabulateField
                                 end

con idField = intField["ID"]

val textFieldBody = fn name : text => {Name = name,
                                       Display = fn t => <html><text value={t}/></html>}

con textField = fn name : text => pack text
                                     with textFieldBody name
                                     as tabulateField
                                  end

val tabulate = fn fields ::: {Exists tabulateField} =>
   fn tab :: Table([Id : idField] ++ fields) =>
   fn () => <html><body>
      <table>
         <tr>
            {fold con (fn r :: {Exists tabulateField} => trTags)
                (fn name :: FieldName =>
                   fn t :: Exists tabulateField =>
                   fn tail :: {Exists tabulateField} =>
                   fn acc : trTags =>
                      unpack con t as t2 with record in
                         <tr>
                            <th><text value={#Name record}/></th>
                            {acc}
                         </tr>
                      end)
                <tr></tr>
                [fields]}
         </tr>

         {(SELECT T.*fields FROM tab AS T)
         (fn row : $(enter T fields) => fn doc =>
         <table><tr>
            {fold (fn t :: {Exists tabulateField} => trTags)
               (fn name :: FieldName =>
                  fn t :: Exists tabulateField =>
                  fn tail :: {Exists tabulateField} =>
                  fn v : t => fn doc : trTags =>
                  unpack v as v2 : t2 with record in
                     <tr>
                        <td>{#Display record v2}</td>
                        {doc}
                     </tr>
                  end)
            <tr></tr>
            [enter T fields] row}
            </tr>
            {doc}
          </table>)
          <table></table>}
      </table>
   </body></html>
```

Figure 8. Implementation of library code for second example

In the more expressive setting of Coq, it's easy to see why: Consider implementing a constructor-level Turing machine simulator parameterized on the number of simulation steps to run. The halting problem is equivalent to checking whether a constant "it didn't halt yet" answer is equivalent to an application of the simulator to a variable number of steps. Laconic may not be as expressive as Coq, but fully automatic type equality reasoning is still likely to be intractable.

Purely theoretical considerations aside, here's an example of a tricky type equality constraint that comes up in practice. I'll simplify the situation slightly, since I don't have enough space to go into the details that aren't fundamental to metaprogramming. In Laconic/Web, legal expressions to use in SQL queries have their own type `exp`. There is a built-in operator `typeof : exp -> Type` that maps SQL expressions to their Laconic/Web types.

Figure 8 demonstrated the `T.*fs` syntax, for projecting a variable set of fields from a table. This syntax is compiled into:

```
fold (fn ([name : t] ++ tail) => fn acc =>
   [name : T.name] ++ acc) [] fs
```

I use an abbreviated syntax for the function argument of `fold`. This fold takes a record of types and replaces the value of each field with a projection of that field from the table `T`.

At a later point in typechecking, we need to figure out the result type of the entire query. Part of that type is determined by the record produced by the above fold. We need to use this equality, which involves the unknown record `fs` that is compatible with the fields of table `T`:

```
  fold (fn ([name : e] ++ tail) => fn acc =>
     [name : typeof(e)] ++ acc) []
     (fold (fn ([name : t] ++ tail) => fn acc =>
        [name : T.name] ++ acc) [] fs)
= fs
```

This theorem is easy to prove inductively, using the definitional equality reductions to do the meat of the work in the base and inductive cases, plus a suitable lemma to the effect that `typeof` and field projection are inverses. Yet it seems intractable to perform type-checking in a setting where inductive proofs must be considered at most points. At the same time, following the lead of other dependently-typed programming languages and asking programmers to construct proof objects themselves seems impractical.

Instead, I chose a stratified approach, where library authors package useful identities along with the rest of their code. Each identity has a proof of correctness, constructed in a proof assistant like Coq. The individual proofs aren't hard for someone with training in such tools, since facts like these about functional programs are just the kinds of theorems that proof tools based on constructive logic were built to handle. When a client of the library uses its packaged functions, the Laconic type checker applies the identities automatically. The end-programmer doesn't even need to understand what a formal proof is. At the same time, he can rest assured that none of the identities introduce type-checking unsoundness.

5. A Formalization of Core Laconic

In this section, I'll present a formal account of a simplified version of Laconic/Web, without any of the web app-specific features. This small calculus mostly illustrates the features of Laconic in general, with the specific examples of records and existential constructors included. My formalization is directly inspired by the design and metatheory of the Calculus of Constructions [CH88] and related dependently-typed languages.

Figure 9 gives the grammar for Core Laconic. As in the full language, the main syntactic categories are kinds, constructors, and expressions. A program is a sequence of named definitions of constructors and expressions as well as *proofs*. These are proofs in a proof system that I leave unspecified here, though Coq's Calculus of Inductive Constructions is the canonical model. It need only support proofs of universally quantified equalities, to be used in automatic rewriting during constructor normalization; and, of course, it must be sound with respect to the language semantics that I introduce here. A facility for giving inductive proofs is essential to completeness, but not soundness. Note that, unlike in most other dependently typed programming languages, proofs are segregated from the other syntactic term categories and may only occur *at the top level* of a program. This means that we don't need to deal with issues that arise in assigning "computational" semantics to proofs or programs that manipulate proofs. A need for proof irrelevance is one common wrinkle that would show up in such a setting.

$$
\begin{array}{rcrl}
\text{Variables} & x & & \\
\text{Names} & X & & \\
\text{Proofs} & \varphi & & \\
& & & \\
\text{Kinds} & \kappa & ::= & \mathsf{Unit} \mid \mathsf{Name} \mid \mathsf{Type} \\
& & & \mid\ \Pi x :: \kappa.\ \kappa \mid \{\kappa\} \mid \exists c \\
\text{Constructors} & c, \tau & ::= & () \mid X \\
& & & \mid\ \tau \rightarrow \tau \mid \Pi x :: \kappa.\ \tau \mid \$c \\
& & & \mid\ x \mid \lambda x :: \kappa.\ c \mid c\ c \\
& & & \mid\ [] \mid [c : c] \mid c \oplus c \\
& & & \mid\ \mathsf{fold}(\lambda x :: \{\kappa\}.\ \kappa) \\
& & & \mid \langle c, e \rangle_c \mid \pi_1 c \\
\text{Expressions} & e & ::= & x \mid \lambda x : \tau.\ e \mid e\ e \\
& & & \mid \lambda x :: \kappa.\ e \mid e @ c \\
& & & \mid\ \{X_i = e_i\}_{i=1}^n \mid \pi_c \\
& & & \mid \mathsf{fold}_\kappa \mid \pi_2 c \\
\text{Programs} & p & ::= & \cdot \mid p, x = c \mid p, x = e \mid p, \varphi
\end{array}
$$

Figure 9. The syntax of Core Laconic

In the class of kinds, we have base kinds for the trivial constructor, record field names, and types; dependent function kinds Π; record kinds; and existential kinds, parameterized by constructors of function kind. I'll sometimes abbreviate the kind $\Pi x :: \kappa_1.\ \kappa_2$ as $\kappa_1 \rightarrow \kappa_2$ when x is not free in κ_2.

Constructors consist of the trivial (unit) constructor; constant field names; non-dependent function types over expressions; dependent function types over constructors; record types; constructor variables, function abstractions, and applications; the empty record, singleton records, and record concatenation; the record fold operation, parameterized over a kind schema relating the constructor to fold over to the kind of the result; existential constructor packages $\langle c_1, e \rangle_{c_2}$, packaging constructor c_1 and expression e according to the package signature c_2; and projection π_1 of the constructor in an existential package.

Expressions are variables; function abstraction and application for expression and constructor arguments; record construction and projection; expression-level record folding; and projection π_2 of the expression in an existential package. Fold expressions can be denoted more simply than fold constructors because Laconic doesn't support kind polymorphism, necessitating the use of special kind schemas in the second case.

Figures 10 and 11 give the basic static validity judgments for Core Laconic. $\Gamma \vdash c :: \kappa$ indicates that constructor c has kind κ in context Γ, where contexts are an extension of the class of programs to allow abstract bindings of constructor $(x :: \kappa)$ and expression $(x : \tau)$ variables. Similarly, $\Gamma \vdash e : \tau$ says that expression e has type τ in Γ. I use the notational shorthand $\Gamma \vdash \kappa$ ok to mean that

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

$$\frac{\Gamma \vdash \tau_1 :: \mathsf{Type} \quad \Gamma \vdash \tau_2 :: \mathsf{Type}}{\Gamma \vdash \tau_1 \to \tau_2 :: \mathsf{Type}} \qquad \frac{\Gamma \vdash \kappa \ \mathsf{ok} \quad \Gamma, x :: \kappa \vdash \tau :: \mathsf{Type}}{\Gamma \vdash \Pi x :: \kappa.\ \tau :: \mathsf{Type}}$$

$$\frac{}{\Gamma, x :: \kappa, \Gamma' \vdash x :: \kappa} \qquad \frac{\Gamma \vdash c :: \{\mathsf{Type}\}}{\Gamma \vdash \$c :: \mathsf{Type}}$$

$$\frac{\Gamma \vdash \kappa \ \mathsf{ok} \quad \Gamma, x :: \kappa \vdash c :: \kappa'}{\Gamma \vdash \lambda x :: \kappa.\ c :: \Pi x :: \kappa.\ \kappa'} \qquad \frac{\Gamma \vdash c_1 :: \Pi x :: \kappa.\ \kappa' \quad \Gamma \vdash c_2 :: \kappa}{\Gamma \vdash c_1\ c_2 :: [x \mapsto c_2]\kappa'}$$

$$\frac{\Gamma \vdash \kappa \ \mathsf{ok}}{\Gamma \vdash [] :: \{\kappa\}} \qquad \frac{\Gamma \vdash c_1 :: \mathsf{Name} \quad \Gamma \vdash c_2 :: \kappa}{\Gamma \vdash [c_1 : c_2] :: \{\kappa\}}$$

$$\frac{\Gamma \vdash c_1 :: \{\kappa\} \quad \Gamma \vdash c_2 :: \{\kappa\}}{\Gamma \vdash c_1 \oplus c_2 :: \{\kappa\}} \qquad \frac{\Gamma \vdash c :: \exists c' \quad \Gamma \vdash c' :: \Pi x :: \kappa.\ \kappa'}{\Gamma \vdash \pi_1 c :: \kappa}$$

$$\frac{\Gamma \vdash c_2 :: \Pi x :: \kappa.\ \kappa' \quad \Gamma \vdash c_1 :: \kappa \quad \Gamma \vdash e : c_2\ c_1}{\Gamma \vdash \langle c_1, e \rangle_{c_2} :: \exists c_2}$$

$$\frac{\Gamma \vdash \kappa \ \mathsf{ok} \quad \Gamma, x :: \{\kappa\} \vdash \kappa' \ \mathsf{ok}}{\begin{array}{c}\Gamma \vdash \mathsf{fold}(\lambda x :: \{\kappa\}.\ \kappa') \\ :: (\Pi n :: \mathsf{Name}.\Pi v :: \kappa.\Pi tl :: \{\kappa\}.\Pi acc :: [x \mapsto tl]\kappa'. \\ [x \mapsto [n : v] \oplus tl]\kappa') \to [x \mapsto []]\kappa' \to \Pi x :: \{\kappa\}.\ \kappa'\end{array}}$$

Figure 10. Kinding judgment for Core Laconic

$$\frac{}{\Gamma, x : \tau, \Gamma' \vdash x : \tau}$$

$$\frac{\Gamma \vdash \tau :: \mathsf{Type} \quad \Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau.\ e : \tau \to \tau'}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1' \quad \Gamma \vdash \tau_1' \leq \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\frac{\Gamma \vdash \kappa \ \mathsf{ok} \quad \Gamma, x :: \kappa \vdash e : \tau}{\Gamma \vdash \lambda x :: \kappa.\ e : \Pi x :: \kappa.\ \tau}$$

$$\frac{\Gamma \vdash e : \Pi x :: \kappa.\ \tau \quad \Gamma \vdash c :: \kappa}{\Gamma \vdash e@c : [x \mapsto c]\tau}$$

$$\frac{\text{For each } i \in 1..n\text{: } \Gamma \vdash e_i : \tau_i}{\Gamma \vdash \{X_i = e_i\}_{i=1}^n : \$[X_i : \tau_i]_{i=1}^n}$$

$$\frac{}{\Gamma \vdash \pi_c : \Pi \tau :: \mathsf{Type}.\Pi r :: \{\mathsf{Type}\}.\$([c : \tau] \oplus r) \to \tau}$$

$$\frac{\Gamma \vdash \kappa \ \mathsf{ok}}{\begin{array}{c}\Gamma \vdash \mathsf{fold}_\kappa :: \Pi s :: (\{\kappa\} \to \mathsf{Type}). \\ (\Pi n :: \mathsf{Name}.\Pi v :: \kappa.\Pi tl :: \{\kappa\}.\Pi acc :: s\ tl. \\ s\ ([n : v] \oplus tl)) \to s\ [] \to \Pi x :: \{\kappa\}.\ s\ x\end{array}}$$

$$\frac{\Gamma \vdash c :: \exists c'}{\Gamma \vdash \pi_2 c : c'\ (\pi_1 c)}$$

Figure 11. Typing judgment for Core Laconic

$$\frac{}{\Gamma \vdash c \sqsubseteq c}$$

$$\frac{\Gamma \vdash \tau_1' \sqsubseteq \tau_1 \quad \Gamma \vdash \tau_2 \sqsubseteq \tau_2'}{\Gamma \vdash \tau_1 \to \tau_2 \sqsubseteq \tau_1' \to \tau_2'}$$

$$\frac{\Gamma, x :: \kappa \vdash \tau \sqsubseteq \tau'}{\Gamma \vdash \Pi x :: \kappa.\ \tau \sqsubseteq \Pi x :: \kappa.\ \tau'}$$

$$\frac{}{\Gamma \vdash c \sqsubseteq []}$$

$$\frac{\Gamma \vdash c_1 \sqsubseteq c_2}{\Gamma \vdash [n : c_1] \sqsubseteq [n : c_2]}$$

$$\frac{\Gamma \vdash r_1 \in c \rightsquigarrow c' \quad \Gamma \vdash c' \leq r_2}{\Gamma \vdash c \sqsubseteq r_1 \oplus r_2}$$

$$\frac{\Gamma \vdash c \in r_1 \rightsquigarrow r_1'}{\Gamma \vdash c \in r_1 \oplus r_2 \rightsquigarrow r_1' \oplus r_2}$$

$$\frac{\Gamma \vdash c \in r_2 \rightsquigarrow r_2'}{\Gamma \vdash c \in r_1 \oplus r_2 \rightsquigarrow r_1 \oplus r_2'}$$

$$\frac{\Gamma \vdash c \sqsubseteq c'}{\Gamma \vdash c' \in c \rightsquigarrow []}$$

$$\frac{\Gamma \vdash c_1 \Downarrow c_1' \quad \Gamma \vdash c_2 \Downarrow c_2' \quad \Gamma \vdash c_1' \sqsubseteq c_2'}{\Gamma \vdash c_1 \leq c_2}$$

Figure 12. Selected constructor subsumption rules for Core Laconic

there exists a c such that $\Gamma \vdash c :: \kappa$; κ is well-formed because it describes some constructor.

The rules I present in this section are algorithmic. That is, their interpretation as a logic program defines an algorithm. Running the logic program on a judgment about fully-determined terms checks that the judgment really holds. The logic program can also be used to do type inference by running it on a goal with some unification variables.

I make the standard assumptions about freshness of variables. I overload the notation $[x \mapsto t_1]t_2$ to denote the capture-avoiding substitution of t_1 for x in t_2 for different classes of terms. For simplicity here, I ignore issues of uniqueness of field names within a record.

A crucial element of the typing judgment is the use of a subtyping relation $\Gamma \vdash c_1 \leq c_2$ in the expression function application rule of Figure 11. All of the "theorem proving" style reasoning is done to support this judgment, which is defined partly in Figure 12. The omitted rules are simple congruences in the standard style.

Subtyping is split into three judgments: the basic syntax-directed subtyping judgment $\Gamma \vdash c_1 \sqsubseteq c_2$; the syntactic record matching judgment $\Gamma \vdash c_1 \in c_2 \rightsquigarrow c_3$, which says that a component matching c_1 was found inside record constructor c_2, and c_3 is the result of removing that component; and the main judgment $\Gamma \vdash c_1 \leq c_2$, which first normalizes the two constructors and then verifies that they belong to $\sqsubseteq$. The main points of interest are in record subtyping, where a componentwise matching is performed between trees of $\oplus$ operations in a way that takes commutativity

$$\frac{\Gamma \vdash \tau_1 \Downarrow \tau_1' \quad \Gamma \vdash \tau_2 \Downarrow \tau_2'}{\Gamma \vdash \tau_1 \to \tau_2 \downarrow \tau_1' \to \tau_2'}$$

$$\frac{\Gamma \vdash c_1 \Downarrow (\lambda x :: \kappa . c_1') \quad \Gamma \vdash c_2 \Downarrow c_2' \quad \Gamma \vdash [x \mapsto c_2']c_1' \Downarrow c'}{\Gamma \vdash c_1\ c_2 \downarrow c'}$$

$$\frac{x = c \in \Gamma \quad \Gamma \vdash c \Downarrow c'}{\Gamma \vdash x \downarrow c'}$$

$$\frac{\Gamma \vdash c_1 \Downarrow [] \quad \Gamma \vdash c_2 \Downarrow c_2'}{\Gamma \vdash c_1 \oplus c_2 \downarrow c_2'}$$

$$\frac{\Gamma \vdash c_1 \Downarrow [n : c] \oplus c_1' \quad \Gamma \vdash c_2 \Downarrow c_2' \quad \Gamma \vdash (c_1' \oplus c_2') \Downarrow c'}{\Gamma \vdash c_1 \oplus c_2 \downarrow [n : c] \oplus c'}$$

$$\frac{\Gamma \vdash r \Downarrow [] \quad \Gamma \vdash \mathit{base} \Downarrow c'}{\Gamma \vdash \mathsf{fold}(\lambda x :: \{\kappa\}.\ \kappa')\ \mathit{ind}\ \mathit{base}\ r \downarrow c'}$$

$$\frac{\begin{array}{c}\Gamma \vdash r \Downarrow [n : c] \oplus r' \\ \Gamma \vdash \mathit{base}\ n\ c\ r'\ (\mathsf{fold}(\lambda x :: \{\kappa\}.\ \kappa')\ \mathit{ind}\ \mathit{base}\ r') \Downarrow c'\end{array}}{\Gamma \vdash \mathsf{fold}(\lambda x :: \{\kappa\}.\ \kappa')\ \mathit{ind}\ \mathit{base}\ r \downarrow c'}$$

$$\frac{\Gamma \vdash c \Downarrow \langle c_1, e \rangle_{c_2}}{\Gamma \vdash \pi_1 c \downarrow c_1}$$

$$\frac{\Gamma \vdash c \downarrow c' \quad \Gamma \sharp c'}{\Gamma \vdash c \Downarrow c'}$$

$$\frac{\Gamma \vdash c \downarrow c' \quad \Gamma \vdash c' = c'' \quad \Gamma \vdash c'' \Downarrow c'''}{\Gamma \vdash c \Downarrow c'''}$$

Figure 13. Selected constructor reduction rules for Core Laconic

into account. This matching approach works with record variables as well as constant records.

Now we come to the most interesting and novel part, normalization of constructors. This is handled via a pair of judgments partially defined in Figure 13. The basic idea is that we normalize constructors by visiting their abstract syntax trees starting at the leaves and moving up. At each node that is visited, we first perform standard reductions based on definitional equality; e.g., β-reductions. Once the node is reduced as far as possible by these rules, we check for user-provided rewrite rules whose lefthand sides unify directly against this node. If a rule matches, we apply it and restart the normalization process for the node's new form. When no rule matches, we are done with this node, and we can move on to its parent if all of its siblings have also been handled already.

The two judgments capture the different stages of this process. $\Gamma \vdash c \downarrow c'$ means that c is normalized to c' using full normalization on all but the root of c's abstract syntax tree, and using definitional reductions at the root, if applicable. $\Gamma \vdash c \Downarrow c'$ is the full reduction relation that adds usage of user rewrite rules. $\Gamma \sharp c$ denotes that no rewrite rule corresponding to a proof in Γ has a lefthand side that unifies with c. $\Gamma \vdash c = c'$ denotes that there is a proof in Γ whose universally quantified conclusion can be instantiated to yield $c = c'$.

Figure 14 completes the picture by giving the overall validity judgment for Core Laconic programs. It simply steps through a program, checking each definition in the context of the previous definitions. The case for proofs uses an auxiliary judgment $\Gamma \vdash \varphi$ ok, which can be used to enforce a number of useful properties of proofs. Its basic role is to check that a proof is logically valid. If the proof language is expressive enough to prove more than universally quantified equalities between constructors, then this judgment should rule out the alternatives, and it should disallow proofs of equalities where some quantified variable appears on the righthand side of the equation but not the left. To make it easier for Laconic programmers to use compositional reasoning, the judgment should probably disallow the binding of a proof that could apply in the same case as an already-bound proof; the programmer doesn't want to have to remember complicated precedence rules to understand typechecking. Finally, we would like typechecking to terminate in any environment, so we should enforce that every rewrite rule decreases the size of a constructor according to an appropriate global well-founded relation.

$$\frac{}{\cdot\ \mathsf{ok}} \qquad \frac{\Gamma\ \mathsf{ok} \quad \Gamma \vdash c :: \kappa}{\Gamma, x = c\ \mathsf{ok}} \qquad \frac{\Gamma\ \mathsf{ok} \quad \Gamma \vdash e : \tau}{\Gamma, x = e\ \mathsf{ok}} \qquad \frac{\Gamma\ \mathsf{ok} \quad \Gamma \vdash \varphi\ \mathsf{ok}}{\Gamma, \varphi\ \mathsf{ok}}$$

Figure 14. Program type correctness judgment for Core Laconic

I won't give a dynamic semantics for Core Laconic here; there aren't any surprises in how expressions should be executed. We obtain a standard language safety theorem, where every well-typed expression evaluates to a value whose form is determined by that type. Since Laconic programs are usually compiled by using specialization and partial evaluation to remove higher-order types, another important property is that normalization of constructors (and expressions, via a similar judgment that I haven't given here) preserves kinding and typing judgments.

6. Implementation and Experiments

I've implemented a prototype Laconic/Web compiler in Standard ML. Compilation works through specialization that converts programs into a form without higher-order types. These programs are easy to translate to Standard ML, in which form they are compiled by version 20060213 of the MLton whole-program optimizing compiler[1]. SQL interaction uses version 8.1 of the PostgreSQL relational database management system[2], and Laconic/Web programs interface with web servers through the FastCGI protocol[3].

The only omission in my prototype implementation compared to the Core Laconic language from the last section is the facility for extending the type-checker with proven rewrite rules. Instead, I've added each required rule manually into the compiler. It was through experimentation with my prototype that I came to the conclusion that the rewrite rule mechanism that I've described is sufficient to support practical web app development. I've only needed to add on the order of ten rewrite rules to allow type-checking of all of the Laconic/Web programs I've written so far, including the case study that I am about to present. There are practical issues left to figure out about how to integrate rewrite rule correctness proofs with the rest of the development process, through use of a proof assistant or otherwise, but my experience to date makes me confident that proof construction can be kept to a minimum in practice.

[1] `http://mlton.org/`

[2] `http://www.postgresql.org/`

[3] `http://www.fastcgi.com/`

	Rails	Laconic
Lines of code	91	65
Directory size	51K	2K
Throughput (pages/minute)	5k	42k
Virtual memory usage	25MB	5MB
Resident set size	22MB	2MB

Table 1. Comparison of the Rails and Laconic solutions

6.1 Case Study: Comparison with Ruby on Rails

Ruby on Rails[4] is a very popular web application development framework based around the Ruby programming language. Its designers say that their goal is to make developers more productive by letting them write more nearly what they really mean to say, avoiding lots of plumbing and complicated XML configuration files. Key elements include a Model-View-Controller paradigm with conventions determining which classes' methods are called for which URL requests, an object-relational database interface providing an OO view of SQL database interaction, and common use of code generation scripts whose outputs are edited by humans.

My comparison is based on a Laconic/Web reimplementation of the application developed in the course of a popular Rails tutorial[5]. The result is a simple management system for the financial accounts of a fictitious company. The standard database operations are provided on accounts: creation of new accounts, listing existing accounts, and display, editing, and deletion of individual accounts, including validation of user inputs. In addition, each account has a list of expenses associated with it, which are listed and added on that account's display page. My Laconic/Web reimplementation duplicates all of the functionality of the Rails original, with the exception of treating one field as arbitrary text instead of a specialized type of dates.

Appendix A gives the code for this application, called "Expenses." It relies on a library of generally useful functions. The actual application-specific code fits on a single page. This is in contrast to the Rails implementation, which exists as a complicated directory structure generated by a script, where on the order of ten of the files have been modified from their original versions.

The Rails tutorial can be read as a sort of informal program, with instructions meant to be executed by the reader. The resulting program isn't represented in a single file, but, for comparison purposes, we can follow this analogy of the tutorial to a program and see how long and complex a program it is. It asks the reader to invoke code generation scripts 5 times and make 12 different modifications to 7 different files, demonstrated with 74 lines of code. This last number is surprisingly close to 65, the number of lines of code in the Laconic/Web implementation in Appendix A plus the associated configuration file (which I haven't included in this paper).

Besides conciseness and maintainability of code, runtime performance is another serious consideration in many cases, so I also benchmarked the performance of the Rails and Laconic implementations. Both applications were run through the LightTPD web server[6] via FastCGI, with the Rails application running in "production" mode.

I used Apache JMeter[7] to drive a stress test and record throughput figures. Somewhat arbitrarily, I chose to allocate 20 parallel request handler processes in each test. Each test involved 100 local threads acting as clients, repeatedly requesting pages from the web server. The requests alternated at random between requesting six representative pages of the application. I ran all experiments on a 3.2 GHz Pentium 4 with 2 GB of RAM.

Table 1 summarizes the overall comparison results. The "Lines of code" row follows the methodology I described for treating the text of the tutorial as a program. "Directory size" gives the size of a gzipped tarball of all of the application-specific files in an implementation. "Throughput" displays how many pages per minute each implementation was serving when it reached a quiescent state. The last two rows give the average total virtual memory usage and actual usage of physical memory per request handling process, after the stress test had completed.

In summary, this case study demonstrates preliminary support for the assertion that Laconic compares favorably to Rails for productivity. While I don't have any comparison on how hard it was to *write* the two applications, the Laconic application specific program code is shorter than its Rails counterpart, and the total source tree is much smaller, which makes it easier to *read* and understand a new application's code. Moreover, through the use of a modern optimizing compiler, the Laconic version is able to achieve a roughly 10x throughput improvement over Rails, and its memory footprint is several times smaller. This translates into needing to purchase less hardware to run a web application with a fixed amount of traffic.

7. Related Work

There has been a flurry of interest lately in programming language solutions to the "impedance mismatch" problem (in web development and elsewhere), which roughly refers to issues of implicit dependencies among pieces of software arising from the mixing of multiple languages, multiple executions of a program in what is conceptually a single interaction, etc.. New languages like Links [CLWY06] and Hop [SGL] address these concerns, with a special focus on supporting the "AJAX" style of web application development, which uses JavaScript to drive asynchronous communication between web browser and web server. The WASH/CGI framework [Thi02] achieves many of the same benefits for traditional CGI programming using a Haskell library. The LINQ language [MBB06] from Microsoft focuses on integration of features related to relational databases and XML. Queinnec's influential paper [Que00] first proposed using the language mechanism of continuations to encapsulate web application state in a way that supports easy backtracking. These ideas have provided inspiration for the base fragment of Laconic/Web, but none address issues of metaprogramming, which I believe has a critical role to play in enabling productive web development.

Practical dependently-typed programming languages are another popular subject of study today. Recent proposals include ATS [CX05], Cayenne [Aug98], Epigram [MM04], Ωmega [SP04], and RSP [WSW05]. Each language can be viewed as an attempt to make Coq-style programming more convenient, along with the inclusion of some traditional programming language features usually considered too unruly for a theorem proving system's logic. To my knowledge, all significant applications of these languages to date have dealt with theoretical computer science. In contrast, I designed Laconic/Web to be a viable competitor in the popular and mainstream domain of web application development. The mechanisms that I use to allow "naive" programmers to write Laconic/Web programs without any knowledge of proof theory are novel. The extension of the type checker with user-provided rewrite rules and their inductive proofs seems to be the biggest departure from past work. There is discussion in the literature about the importance of equality reasoning, but I've found no mention of prag-

[4] `http://www.rubyonrails.org/`

[5] `http://developer.apple.com/tools/rubyonrails.html`

[6] `http://www.lighttpd.net/`

[7] `http://jakarta.apache.org/jmeter/`

matic methods for automatic equality reasoning that is able to use arbitrary inductively-proved facts. My results here are interesting as evidence that the simple technique that I suggest is effective in practice.

The Lisp and Scheme communities have long been using macros to automate the construction of code. Of course, they operate in a dynamically typed setting, where there are no guarantees that individual macros will always behave properly and no compiler-checked documentation (e.g., type signatures) on how to use particular macros. Languages like MetaML [TS00] provide a statically-typed answer to the same problems in terms of multi-stage programming. In contrast, Laconic has no notion of multiple stages. Similar effects come from applying partial evaluation to a language with a higher-order type system, where a compile-time error is signaled when higher-order constructs survive partial evaluation. On the other hand, Laconic/Web does have an element of explicit construction of code, but only when the final web applications is meant to interact with the outside world through programmatic interfaces. Concrete examples include SQL queries and HTML fragments with forms. In both cases, these "embedded languages" are encoded with dependent type theory instead of through specialized meta-language mechanisms, similar to, e.g., implementation of metaprogramming in Coq and some studies carried out with Ωmega [PL04].

8. Conclusion

I have presented the programming language Laconic/Web and shown that it is useful in practice for developing web applications with static guarantees of good behavior. Laconic is notable for its combination of an expressive dependent type system with good usability compared to other proposed dependently-typed languages. In particular, most Laconic/Web programmers shouldn't need to do any explicit manipulation of proof objects, thanks to a language design that allows the type checker to be extended with proven rewrite rules packaged in libraries, as well as some simplifications made possible by Laconic/Web's design as a domain-specific language rather than a general theorem proving tool. Through a case study comparing Laconic/Web against one of today's most popular frameworks for rapid coding of web applications, I've demonstrated that Laconic/Web can enable more concise programs, as well as considerably better runtime performance. These results provide a helpful counterpoint to a common "folk belief" in the mainstream development world that dynamic typing and runtime reflection are critical for effective "agile" programming.

As mentioned earlier, the mechanism for integrating formal proofs of rewriting rule correctness remains to be designed. There are many newer web application features that Laconic/Web doesn't handle yet, including the AJAX style of asynchronous client/server communication and other uses of XML-based "web services." I hope to include support for these in a future version.

The mechanism I suggest for usage of rewrite rules in type checking can be unsatisfying for its lack of simple completeness guarantees. Type inference in dependently typed languages is usually undecidable, but that doesn't mean that there aren't restricted languages that prove useful in practice. Whether or not such languages exist for the domain of web application programming that I've treated here is an open problem. On the other hand, it could be fruitful to formalize the essences of problems that occur in this and similar domains and attempt to prove undecidability or intractability results, justifying the open approach to using programmer-provided proofs.

I plan to build a toolkit for the effective construction of practical compilers for languages in the Laconic family. Each new domain targeted by a Laconic language will share many features with the Laconic family or with subsets of it, so enabling re-use is critical for effective "language-oriented programming." I believe that a good common basis for this framework would be Coq and its Calculus of Inductive Constructions (CIC). Language features can be given semantics in terms of a translation into CIC, and proofs of rewrite rule correctness can be expressed in the same language.

References

[Aug98] Lennart Augustsson. Cayenne – a language with dependent types. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 239–250, 1998.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer Verlag, 2004.

[CH88] Thierry Coquand and Gerard Huet. The calculus of constructions. *Inf. Comput.*, 76(2-3):95–120, 1988.

[CLWY06] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web programming without tiers. Unpublished manuscript, 2006.

[CX05] Chiyan Chen and Hongwei Xi. Combining programming with theorem proving. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 66–77, 2005.

[MBB06] Erik Meijer, Brian Beckman, and Gavin Bierman. LINQ: Reconciling objects, relations, and XML in the .NET framework. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, 2006.

[MM04] Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[PL04] Emir Pasalic and R. Nathan Linger. Meta-programming with typed object-language representations. In Gabor Karsai and Eelco Visser, editors, *Generative Programming and Component Engineering: Third International Conference*. Lecture Notes in Computer Science, Springer-Verlag, October 2004.

[Que00] Christian Queinnec. The influence of browsers on evaluators or, continuations to program web servers. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 23–33, 2000.

[SGL] Manuel Serrano, Erick Gallesio, and Florian Loitsch. Hop, a language for programming the Web 2.0. `http://hop.inria.fr/usr/local/share/hop/weblets/home/articles/hop-lang/article.html`.

[SP04] Tim Sheard and Emir Pasalic. Meta-programming with built-in type equality. In *Proceedings of the Logical Frameworks and Meta-Languages workshop*, 2004.

[Thi02] Peter Thiemann. WASH/CGI: Server-side web scripting with sessions and typed, compositional forms. In *PADL '02: Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages*, pages 192–208, 2002.

[TS00] Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1–2):211–242, 2000.

[WSW05] Edwin Westbrook, Aaron Stump, and Ian Wehrman. A language-based approach to functionally correct imperative programming. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 268–279, 2005.

```
(* Existential type body for the fields of a table *)
con crudField = fn t :: Type =>
   { (* Record of field-specific code *) }

(* Default integer field handling *)
val intFieldBody : text -> crudField integer
con intField = (* ... *)

(* Dollar-amount field handling *)
val dollarFieldBody : text -> crudField real
con dollarField = (* ... *)

(* ... other pre-defined field constructors ... *)

(* Parameters used in generating CRUD pages *)
con crudParams = {
   Title : text,    (* Title for this section of the application *)
   Singular : text, (* Singular version of database table name *)
   Plural : text,   (* Plural version of database table name *)
   ShowExtra : integer -> (unit -> htmlTags) -> bodyTags
                    (* Extra HTML code to include at the bottom
                     * of the display page for a row. *)
}

(* Generate the CRUD web site *)
val crud : fs ::: {Exists crudField}
   -> tab :: Table([Id : idField] ++ fs)
   -> seq :: Sequence
   -> crudParams
   -> unit -> htmlTags

(* Build table rows for all SQL rows of a table *)
val listAllRows : fname :: FieldName (* Sort by this field *)
   -> ftype :: Exists crudField      (* Type of fname *)
   -> fsShow :: {Exists crudField}   (* Remaining fields to display *)
   -> fsSkip ::: {Exists crudField}  (* All other fields *)
   -> tab :: Table([fname : ftype] ++ fsShow ++ fsSkip)
   -> tableTags

(* Sum up the real-valued fields of a table in rows with a particular key value *)
val sumByKey : keyName :: FieldName (* Name of key field *)
   -> keyType ::: Type              (* Type of key field *)
   -> amount :: FieldName           (* Name of real-valued amount field *)
   -> rest ::: {Type}               (* The rest of the fields in the table *)
   -> tab :: Table([keyName : keyType] ++ [amount : real] ++ rest)
   -> keyType -> real

(* Count the number of rows with a particular key value *)
val countByKey : keyName :: FieldName (* Name of key field *)
   -> keyType ::: Type                (* Type of key field *)
   -> rest ::: {Type}                 (* The rest of the fields in the table *)
   -> tab :: Table([keyName : keyType] ++ rest)
   -> keyType -> integer

(* Look up a single table field by primary key *)
val getField : keyName :: FieldName
   -> keyType ::: Type
   -> getMe :: FieldName
   -> getMeType ::: Type
   -> rest ::: {Type}
   -> tab :: Table([keyName : keyType] ++ [getMe : getMeType] ++ rest)
   -> keyType -> getMeType
```

Figure 15. Signature of the library used for the Expenses example

```
dbtable Account = {
   Id : idField,
   Name : textField["Name"],
   Budget : realField["Budget"]
}
sequence AccountSeq

dbtable Expense = {
   Id : idField,
   PaidOn : textField["Paid on"],
   PayableTo : textField["Payable to"],
   Amount : dollarField["Amount"],
   AccountId : intField["Account#"]
}
sequence ExpenseSeq

val showExtra = fn id : integer => fn genPage : (unit -> htmlTags) =>
   let
      val addExpense = fn input : {PaidOn : text, PayableTo : text, Amount : text} =>
         ((INSERT INTO Expense (Id, PaidOn, PayableTo, Amount, AccountId)
           VALUES ({nextval(@(ExpenseSeq))},
                   {#PaidOn input},
                   {#PayableTo input},
                   {realFromString (#Amount input)},
                   {id})) ();
          genPage ())
   in
      if countByKey[#AccountId][@Expense] id = 0 then
         <body></body>
      else <body>
         <h3>Itemized Expenses</h3>
         <table>
            {listAllRows[#PaidOn]
                        [(textField["Paid on"])]
                        [[PayableTo : textField["Payable to"],
                          Amount : dollarField["Amount"]]]
                        [@Expense]}
            <tr><td align="right" colspan="3">
               {let
                   val total = sumByKey[#AccountId][#Amount][@Expense] id
                   val style = if total > getField[#Id][#Budget][@Account] id then
                                  "color: red"
                               else
                                  "color: black"
                in
                   <tr><b>Total</b>: <span style={style}>$<text value={dollarToString total}/></span></tr>
                end}
            </td></tr>
         </table>

         <form><p>
            On <input{#PaidOn} size="10" />
            to <input{#PayableTo} size="25" />
            in the amount of $<input{#Amount} size="9" />
            <submit value="Record!" handler={addExpense}/>
         </p></form>
      </body>
   end

val main = crud[@Account][@(AccountSeq)]
   {Title = "Expenses",
    Singular = "account",
    Plural = "accounts",
    ShowExtra = showExtra}
```

Figure 16. Implementation of the Expenses application

A. Code for the Case Study

Figures 15 and 16 give the signature of the library and the actual application code used for the "Expenses" case study, respectively.