Ur: Statically-Typed Metaprogramming with Type-Level Record Computation

Adam Chlipala

Impredicative LLC, Cambridge, MA, USA

adamc@impredicative.com

Abstract

Dependent types provide a strong foundation for specifying and verifying rich properties of programs through type-checking. The earliest implementations combined dependency, which allows types to mention program variables; with type-level computation, which facilitates expressive specifications that compute with recursive functions over types. While many recent applications of dependent types omit the latter facility, we argue in this paper that it deserves more attention, even when implemented without dependency.

In particular, the ability to use functional programs as specifications enables *statically-typed metaprogramming*: programs write programs, and static type-checking guarantees that the generating process never produces invalid code. Since our focus is on generic validity properties rather than full correctness verification, it is possible to engineer type inference systems that are very effective in narrow domains. As a demonstration, we present Ur, a programming language designed to facilitate metaprogramming with first-class records and names. On top of Ur, we implement Ur/Web, a special standard library that enables the development of modern Web applications. Ad-hoc code generation is already in wide use in the popular Web application frameworks, and we show how that generation may be tamed using types, without forcing metaprogram authors to write proofs or forcing metaprogram users to write any fancy types.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Applicative (Functional) Programming; F.3.3 [*Logics and Meanings of Programs*]: Type Structure

General Terms Languages, Reliability, Security

Keywords dependent types, metaprogramming

1. Introduction

Dependent types are a technique that is picking up momentum in practical language design. A dependent type system allows types to refer to program variables whose values are not determined until runtime. The classical approach, exemplified by Coq [3] and Agda [17], is based on dependent type theory. These languages combine dependent typing with rich facilities for type-level computation. Types may be computed via calls to recursive functions. This paper is about a very practical application of type-level computation, without dependency, in a system that is still very much inspired by Coq and Agda.

Until recently, languages with dependent type systems were invariably designed to be usable for full correctness verification. A type system must be quite complex to support that goal. It will almost certainly have undecidable type inference, as well as intractable type inference in practice. Thus, to use such a system to verify serious applications, a programmer must inevitably spend significant effort writing annotations and proofs that exist only to appease the type checker. At the same time, the same language features can be very useful in checking general program validity properties, in a sense more in line with mainstream usage of type systems. By identifying a narrow domain of properties, we can hope to build an effective type inference procedure, without compromising the programmer's ability to employ clever abstractions.

We suggest *metaprogramming in Web applications* as one such "killer application." The term "metaprogramming" is used to describe programs that perform code generation, code introspection, or both. In this paper, we restrict our attention to code generation. This will include both *heterogeneous metaprogramming*, where programs in our Turing-complete language build SQL queries and HTML pages; and *homogeneous metaprogramming*, where programs in our Turing-complete language generate other such programs. In the latter category, we follow standard ideas from the world of dependent typing, using type-level computation to avoid explicit syntax manipulation. Nonetheless, we achieve the same functionality that is traditionally implemented with code generation.

The most popular approach to Web application programming today involves usage of frameworks implemented in dynamic languages, including Ruby on Rails[1] and Django[2]. These systems include metaprogramming components to help coders get new Web applications up and running quickly. In some cases, this is ad-hoc generation of source code as strings; in other cases, it is reflection-based runtime code generation. No matter the details, there is no static checking of code generators. It is easy to have bugs that go uncaught even by systematic testing.

Lurking bugs in Web code generators are a serious business. One study [7] found that over 30% of Web applications are susceptible to some kind of code injection attack, where code from an untrusted source is relayed to browsers or database servers through a Web application that does insufficient input sanitization. It is hard enough to treat input securely in a standalone application; it is even harder to write a code generator that never outputs a vulnerable pro-

PLDI'10, June 5–10, 2010, Toronto, Ontario, Canada.
Copyright © 2010 ACM 978-1-4503-0019/10/06...$10.00

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

[2] `http://www.djangoproject.com/`

gram. A rich enough static type system can guarantee that metaprograms are valid in this sense.

For instance, we can model HTML pages and SQL queries with rich abstract syntax tree types. When treating these types as simple strings, it is easy to splice in unsanitized user input in a way that has surprising parsing consequences. By working instead with syntax trees, we avoid such complexities. We can go even further and use advanced type system features to guarantee that, for instance, every constructible SQL query makes correct assumptions about a database schema. Finally, with further type system sophistication, we can assign static types to programs that *generate* code; for instance, we can assign a static type to a function that is generic in a database schema, producing different queries for different schemas. Any type-correct metaprogram of this kind is guaranteed to output query code that is immune to code injection attacks.

Statically-typed code generation is attractive for reasons beside security. A serious barrier in the way of wider use of metaprogramming is the difficulty in building abstractions that programmers can figure out how to use. A good static type system can provide a structuring principle and a source of machine-checked documentation about the interfaces of generic components.

How can we arrive at a language environment that can support our vision? One strategy is to start with a less esoteric programming language and gradually add expressivity. In particular, features once associated only with dependently-typed languages have recently been added to Haskell, in the form of extensions like multiparameter type classes with functional dependencies [12], generalized algebraic datatypes [27], and open type functions [26].

An alternative strategy is to design a new programming language, picking and choosing features inspired by traditional dependently-typed languages. Languages like Cayenne [2] and Sage [14] sacrifice decidable type-checking but keep all three of dependent types, type-level computation, and broad applicability. Another popular approach is to introduce some form of dependent types without type-level computation, as in ATS [5], Deputy [6], and liquid types [25]. Finally, a language can support rich type-level computation without dependent typing, as in Ωmega [29].

Considering classical tools like Coq and Agda, Haskell extensions, and the previous paragraph's new languages, there is a serious common weakness. They do not provide very good support for the construction of new abstractions that manipulate richly-typed values. In each case, either the type system is too weak to support interesting metaprogramming applications, or the annotation and/or proof burden needed to support metaprogramming is very high. In the latter category, type checkers are usually very good at checking normal programs, where all relevant pieces of type-level data are fully determined. However, when some of these pieces of data are unknown, the undecidability of type inference becomes a serious problem. Serious metaprograms generally need to include some kind of *explicit proof terms* to convince the type-checker.

We illustrate the problem with some example code in Coq, a system that can be considered as maximally expressive within this design space. Consider code that manipulates *heterogeneous lists*, where the type of a list conveys exactly how many elements it has, as well as what type each element should have. These types may be different for different list positions. It is natural to define a concatenation operator +++ for heterogeneous lists, whose type is defined in terms of normal list concatenation ++. Richer versions of all of these constructions are very useful in our metaprogramming domain, for such tasks as modeling database schemas. This particular contrived example is chosen for its simplicity, but the problem of "satisfying the type-checker" is a pervasive one that underlies the practical differences between Ur and related languages.

```
Definition l1 := [int, string].
Definition l2 := [bool].
Definition l3 := [int].

Definition h1 : hlist l1 = [< 1, "ABC" >].
Definition h2 : hlist l2 = [< true >].
Definition h3 : hlist l3 = [< 4 >].
Definition h123 : hlist (l1 ++ (l2 ++ l3)) =
  (h1 +++ h2) +++ h3.
```

The `l`*i* variables are lists of types, written in usual ML-like notation. In the definitions of the `h`*i* variables, each type list is used to describe the shape of a heterogeneous list. The final definition of `h123` concatenates our three lists into a single list. Notice that, in the type of `h123`, we apply normal list concatenation as if it were right-associative, while, in the body of the definition, we apply heterogeneous list concatenation left-associatively. In Coq (and in the other languages we surveyed above with type-level computation), it is easy to see that this implicit use of associativity is legal. The lists `l`*i* are constants, so we simply evaluate the two type lists `l1 ++ (l2 ++ l3)` and `(l1 ++ l2) ++ l3`, verifying that the resulting constant lists are equal.

Things get more complicated if we want to write a *generic* associative-concatenation function.

```
Definition acat l1 l2 l3
  (h1 : hlist l1) (h2 : hlist l2) (h3 : hlist l3)
  : hlist (l1 ++ (l2 ++ l3)) =
  cast (assoc l1 l2 l3) ((h1 +++ h2) +++ h3).
```

Here, the variables `l`*i* are type parameters, as in parametric polymorphism. Since we do not know their values, we cannot check type equivalences via the simple evaluate-and-compare method. Instead, to get our function to type-check, we had to include an explicit *cast* expression. The sub-expression `assoc l1 l2 l3` is a *proof term*, establishing the fact `(l1 ++ l2) ++ l3 = l1 ++ (l2 ++ l3)`. The theorem `assoc` was proved separately in a library, using techniques more mathematical than programming-oriented.

In all of the systems we mentioned earlier, this kind of explicit casting is the best that can be done to support the combination of rich typing and genericity. The burden may not seem too great from the classical perspective of dependent-types-for-full-verification. However, when writing a code generator for a Web application, such typing details are far from the main point of the code. Facts like concatenation associativity should be applied automatically. Today's type system state of the art fails to provide this facility.

In this paper, we argue for use of a language with type-level computation but no dependency. Moreover, we are not interested in supporting full correctness verification. We only care to handle the sorts of typing issues that come up in metaprogramming. This is an opportunity to avoid any need for explicit proof terms by building a customized type inference engine. Our approach is by no means a deep theoretical advancement. The key structuring ideas are already there in Coq and Agda, and our system could be implemented as a library in one of those languages. However, using such a library would require heroic efforts in type annotation and theorem-proving. Dependent types are popularly viewed as more theory than practice; in this work, we show that, with just a small injection of domain-specific smarts, this old piece of theory leads to a practical tool that is highly competitive in a popular real-world application domain.

We present the **Ur** programming language, whose novel features center on *first-class, type-level names and records*. Ur includes specialized heuristic type inference that makes it possible to write record-manipulating metaprograms that are free of explicit proof terms. On top of Ur, we have built **Ur/Web**, a domain-specific language for constructing modern Web applications. Ur/Web adds a special standard library, some parsing extensions, and a specialized compiler, but it relies only on the generic Ur type inference engine. The signature of the Ur/Web library describes the syntax and typing constraints of HTML documents and SQL queries, and the inference engine is sufficient to type-check metaprograms that build programs that build documents and queries.

Type inference for Ur is undecidable, and we have no theorems that support our choice of inference procedure. Instead, we point to empirical evidence of Ur/Web's effectiveness. We have built the tool set to be a real, practical Web application framework that is highly competitive with mainstream frameworks. We claim that, in the hands of an experienced functional programmer, Ur/Web is far ahead of the competition in each of the critical areas of programmer productivity, security, and performance. In this paper, we focus on productivity, measured in terms of case studies in using Ur/Web to build practically-useful metaprograms.

In particular, we have based our design on two firm principles.

1. **The author of a metaprogram should never need to write a proof term.** He may write more involved types than usual, and he may need to add new type parameters to some functions, but he should never need to do any work to show that his program really has the type he wrote.
2. **The users of a metaprogram should need to write neither proofs nor types more complex than those found in mainstream programming languages.**

In the next section, we introduce the key features of Ur by example. In Section 3, we formalize a core calculus based on these features. The following section discusses effective type inference for the full language. After that, we describe our implementation and provide evidence for its practicality, in the form of case studies building statically-typed versions of common metaprogramming functionality. We conclude by discussing related and future work.

The open source distribution of the Ur/Web compiler, along with the source code for our case studies, is available at

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

2. Ur By Example

Ur is an extension of System F_ω [22], the higher-order polymorphic lambda calculus, presented with ML-style syntax. The foundation of the key extensions to F_ω is support for type-level names and records. As a simple introduction to these features and their usefulness, we will write a generic record field projection function. For example, the function call `proj [#A] {A = 1, B = 2.3}` will evaluate to `1`, while the call `proj [#D] {C = True, D = "xyz", E = 8}` will evaluate to `"xyz"`. This `proj` function will be usable on *any* record, with arbitrary fields of arbitrary types. Further, `proj` will have a static type that expresses its requirements exactly, and the Ur type-checker will verify that `proj` will work correctly on any input compatible with its type.

The definition of `proj` is not very long, but it depends on a few unusual constructs, which we will introduce below.

```
fun proj [nm :: Name] [t :: Type] [r :: {Type}]
  [[nm] ~ r] (x : $([nm = t] ++ r)) = x.nm
```

Type-level formal arguments to functions are declared inside square brackets. Our `proj` function binds three type-level variables `nm`, `t`, and `r`. Unlike in usual ML code, the type variables appear explicitly in the function definition. Each type variable is assigned a *kind*. Kinds are to types as types are to values; kinds classify different varieties of types. The kind annotations above indicate that `nm` is a field name, `t` is a normal type, and `r` is a record of types, otherwise known in the literature as a *row type*.

The next piece of the function definition is `[[nm] ~ r]`, which declares a *disjointness constraint*. This particular constraint asserts that the name `nm` is not used by the type-level record `r`.

The final formal argument is `x`, which is a normal, value-level argument. We write `x`'s type using the `$` operator, which converts a type-level record `r` (of kind `{Type}`) to a record type (of kind `Type`) with field names and types as indicated by `r`. The code `[nm = t]` is an example of a type-level record literal, denoting the singleton record associating the name `nm` with the type `t`. We use record concatenation `++` to add this singleton to the other fields `r`. This concatenation would be invalid if we had not included the disjointness constraint; Ur enforces lack of field name duplication in any concatenation.

The function body just uses the primitive record field projection operator. By encapsulating that operator in this way, we arrive at a function with the following Ur type.

```
nm :: Name -> t :: Type -> r :: {Type}
  -> [[nm] ~ r] => $([nm = t] ++ r) -> t
```

Here code like `x :: K -> T` indicates a polymorphic function, whose argument is of kind `K`. The function's argument is given name `x`, which may appear free in the function result type `T`. For instance, the normal polymorphic identity function has type `a :: Type -> a -> a`. The parsing precedence of the `::` operator is such that it binds more tightly than any other, in any situation where ambiguity would arise otherwise.

Our `proj` function is straightforward to apply. For instance, `proj [#A] [int] [[B = float]] ! {A = 1, B = 2.3}` has type `int` and reduces to `1`. We write type-level arguments to value-level functions inside square brackets, and a first-class name literal is written by prefixing the name with a `#` character. The `!` stands for a disjointness proof to be inferred. The Ur type inference engine contains a special prover for this purpose. There is no syntax for writing manual proofs; disjointness proofs are always inferred.

The Ur implementation contains a facility for marking some type-level function arguments as implicit. We will not go into detail here on that facility, but we can use it so that our function can be called as simply as `proj [#A] {A = 1, B = 2.3}`. If the remaining arguments are marked as implicit at `proj`'s definition site, the Ur compiler knows to expand this call to `proj [#A] [_] [_] ! {A = 1, B = 2.3}`. Each underscore is treated as a distinct unification variable. A specialized unification procedure for type-level records infers values for these variables.

2.1 A Generic Table Formatter

One very common source of repetitive coding in Web applications is formatting application-specific records for display as tables. In this subsection, we show how a particular "copy-and-paste recipe" can be reified as a well-typed Ur function. The example will demonstrate how code generators may work by iteration over all fields of records of metadata.

Our end product is a function `mkTable` that we will be able to call like this, assuming that we have available functions `showInt : int -> string` and `showFloat : float -> string`.

```
val f = mkTable
  {A = {Label = "A", Show = showInt},
   B = {Label = "B", Show = showFloat}}
```

This defines a function `f` of type `{A : int, B : float} -> string`. When called with a record of the right type, `f` will format it as an HTML table, using the labels `"A"` and `"B"` as headings, and using the functions `showInt` and `showFloat` to render columns of the table. For example, the call `f {A = 2, B = 3.4}` would evaluate to the HTML

```
<tr> <th>A</th> <td>2</td> </tr>
<tr> <th>B</th> <td>3.4</td> </tr>
```

To write `mkTable` in a completely generic way, such that it works with arbitrary sets of fields of arbitrary types, we must develop some type system machinery. This machinery is involved, but the final product is a function that may be called as simply as above.

The first ingredient is a way of folding over the fields of a type-level record. We define a type family `folder`, such that `folder r` is the type of permutations of the fields of record `r`. To avoid introducing too much detail at once, we will give some definitions specialized to records of normal types, though our implementation uses kind polymorphism to generalize the definitions to other varieties of type-level data.

This is the definition of `folder` in terms of simpler constructs. We present it in full first and then consider it a piece at a time, introducing syntactic and semantic elements as they appear.

```
type folder (r :: {Type}) = tf :: ({Type} -> Type)
  -> (nm :: Name -> t :: Type -> r :: {Type}
        -> [[nm] ~ r] => tf r -> tf ([nm = t] ++ r))
  -> tf [] -> tf r
```

This is a type-level function definition. In general, code like `type f (x :: K) = T` introduces a function `f` of kind `K -> K'`, when `T` has kind `K'` in an environment where variable `x` has kind `K`.

The body of `folder`'s definition has the form `tf :: ({Type} -> Type) -> STEP -> INIT -> RESULT`. A `folder` is simply a first-class polymorphic function that may be called to iterate over the fields of `r`. The type-level argument `tf` is the counterpart of the accumulator type, in analogy to the standard list fold functions of functional programming. The component `STEP` gives the type of a suitable function for computing one iteration of the fold, `INIT` the type of the initial value for the fold, and `RESULT` the final result type of the fold.

Following this analogy with traditional list fold functions, one notable difference is that `INIT` and `RESULT` are not the same type. This is because we want to allow the accumulator type to *depend on which prefix of the record's fields we have already stepped through*. That is, when folding over record `r`, the initial accumulator should have type `tf []` (where `[]` is the empty record), and the final accumulator should have type `tf r`. We see this progression reflected in the last line of `folder`'s definition, in the concrete choices of `INIT` and `RESULT`.

The most interesting part of the definition comes in the second and third lines, where we have the type `STEP` of the function to fold over `r`:

```
nm :: Name -> t :: Type -> r :: {Type}
  -> [[nm] ~ r] => tf r -> tf ([nm = t] ++ r)
```

This function takes three type-level arguments: `nm`, the name of the field we are processing; `t`, the type associated with `nm`; and `r`, a record of all of the fields that we already processed. A disjointness constraint asserts that `r` does not use the name `nm`. Finally, we have a normal function type, with different versions of the accumulator type `tf` as the domain and range. In the domain, we have an accumulator type appropriate for the point just after processing every field in `r`. In the range, we extend `r` to indicate that we have now also processed `nm`.

Building on the generic concept of a folder, we can implement our table generator. We define a type-level function that expresses which metadata we will need for each record field. In particular, for each field of type `t`, we need a display label and a function for rendering `t` values as strings.

```
type meta (t :: Type) = {Label : string,
                         Show : t -> string}
```

We use `meta` to express the type of our table generator, which we call `mkTable`. To render a record with fields `r`, we require a `folder r`. We also need a metadata record, whose type is `$(map meta r)`. When `r` is `[f1 = t1, ..., fn = tn]`, the type of this metadata record is `{f1 : meta t1, ..., fn : meta tn}`, which is syntactic sugar for `$[f1 = meta t1, ..., fn = meta tn]`. Here is the definition of `mkTable`.

```
fun mkTable [r :: {Type}] (fl : folder r)
            (mr : $(map meta r)) (x : $r) =
  fl [fn r => $(map meta r) -> $r -> string]
    (fn [nm] [t] [r] [[nm] ~ r] acc mr x =>
      "<tr> <th>" ^ mr.nm.Label ^ "</th> <td>"
      ^ mr.nm.Show x.nm ^ "</td> </tr>"
      ^ acc (mr -- nm) (x -- nm))
    (fn _ _ => "") mr x
```

The function body is a call to the input folder `fl`. In passing the first argument of `fl`, we choose our accumulator type so that each accumulated value is a function to a string from a metadata record and a record of field values. These record types have an explicit dependency on the set of record fields considered so far.

In the step function, the value-level variables are `acc`, the `string` rendering of the fields already processed; `mr`, a version of the input metadata record where already-processed fields have been removed; and `x`, a similarly abbreviated version of the original argument `x`. We build a string with the concatenation operator `^`. We use the name variable `nm` to project individual entries out of the local versions of the records `mr` and `x`. Additionally, we use the operator `x -- nm`, which removes the field `nm` from value-level record `x`. This field removal is necessary to produce arguments of the proper types to pass to the accumulator `acc`.

The type of `mkTable` is easily read off from the function definition:

```
val mkTable : r :: {Type} -> folder r
  -> $(map meta r) -> $r -> string
```

Ur's implicit argument facility does more than just infer types; it can also generate folders automatically, using the order of field names in code as a hint to the permutation the programmer wants. Taking advantage of this possibility, it is easy to bind a version of `mkTable` specialized to a particular record type, with the code we used to introduce this example.

```
val f = mkTable
  {A = {Label = "A", Show = showInt},
   B = {Label = "B", Show = showFloat}}
```

The type of `f` is inferred to be `{A : int, B : float} -> string`. Notice that we did not need to write the type-level record `[A = int, B = float]` explicitly. Rather, the compiler infers that type from the type of the record we pass to `mkTable`. The inference engine is able to solve unification problems like this one to find the value of `r`:

```
  $(map meta r)
= {A : {Label : string, Show : int -> string},
   B : {Label : string, Show : float -> string}}
```

We call this kind of inference *reverse-engineering unification*, because a record is inferred by looking at the output of some operation performed on it. This is the key feature behind making Ur/Web metaprograms no harder to use than the ad-hoc code generators that are popular today.

While `mkTable` is easy to use, its definition is somewhat involved. Our vision for the real-world use of these techniques follows the trajectory of today's mainstream metaprogramming. We find relatively expert developers building metaprogramming tools and dealing with the difficulty of debugging them. Many novice programmers rely on the libraries that the experts build. The novices need simple interfaces. Our experience with Ur leads us to believe that a similar decomposition is plausible with statically-typed metaprogramming. The novice's experience is almost unchanged, and the expert trades off between dynamic debugging and static checking with extra typing-induced overhead. The expert must learn type system idioms that are outside even today's functional programming mainstream, but we hope our examples here provide evidence that this extra training can pay off.

When high security is critical, as is the case with many Web applications, we believe that the static typing approach reduces the overall cost of development. We simplified our definition of `mkTable` by outputting HTML in the `string` type. The real Ur/Web implementation uses a special XML tree type whose type indices track which tags are allowed. Thus, a real Ur/Web definition of `mkTable` would look more like this:

```
fun mkTable [r :: {Type}] (fl : folder r)
           (mr : $(map meta r)) (x : $r) =
  fl [fn r => $(map meta r) -> $r -> xml table]
    (fn [nm] [t] [r] [[nm] ~ r] acc mr x =>
      concat (tr (th (cdata mr.nm.Label)
               :: td (cdata (mr.nm.Show x.nm))
               :: nil))
             (acc (mr -- nm) (x -- nm)))
    (fn _ _ => empty) mr x
```

Compared to the string-based version, we get a stronger guarantee: no matter which record we pass to `mkTable`, the resulting program is free of code injection vulnerabilities. Strings can only be included in XML trees via the explicit `cdata` constructor, which forces appropriate quoting. In general, Ur/Web uses similar strongly-typed syntax trees for any kind of code that Web browsers or database servers might run, providing a global guarantee of freedom from code injection attacks, even in the presence of an expressive metaprogramming facility, and without the need to evaluate specific metaprogram applications to determine if they are safe.

2.2 Generic Database Modification

It is common for Web applications to work with "native" representations of data when possible but then convert such representations into alternate formats for database access. In this section, we develop a toy example inspired by that kind of usage. In particular, we want to write a function that adds a row to a database table, where doing so requires first applying a conversion operation to each element of a record.

The larger lesson from this example has to do with how we may write generic functions that output SQL-like queries. We want these queries to be expressed using rich types that guarantee validity, including type compatibility with a fixed database schema. Therefore, few type systems outside of the dependent types world are equipped to capture the essential well-formedness invariants. Ur supports this kind of programming using the same relatively lightweight features that we have been introducing, with effective inference to minimize the cost of applying generic functions.

Assume we have two abstract type families, corresponding to database tables and to expressions that the database engine understands.

```
type table :: {Type} -> Type
type exp :: {Type} -> Type -> Type
```

A table type is parameterized by a record assigning types to the table's columns. An expression type is parameterized first, by a record expressing which free variables may be mentioned and what their types are; and second, by the type of the expression, according to the database's expression typing rules.

These type families are abstract in the sense that the programmer may not rely on any details of their implementation. In our concrete implementation, both `table` and `exp` are aliases for `string`, but programmers should think of them as abstract syntax tree types. While there are varieties of metaprogramming that deal with both code generation and code introspection, we only deal with the former in Ur/Web and in this paper. No method is provided to, for example, pattern-match on the syntax of an `exp`, as that turns out not to be needed in our domain.

In this example, we do not need to work with expressions that mention table columns. We only need to rely on a single function for constructing expressions: `const`, which converts a constant value into an expression. This function's polymorphic type expresses the fact that the output expression mentions no free variables; since we abstract over an arbitrary environment `r` and assert that the output expression is valid in `r`, it must be the case that there is no particular variable that `const` depends on being able to mention.

```
val const : r :: {Type} -> t :: Type -> t -> exp r t
```

Assume that there is a primitive function for adding a row to a table. The column values for a new row are expressed as a record of expressions with no free column variables.

```
val insert : r :: {Type} -> table r
  -> $(map (exp []) r) -> unit
```

Our final function `toDb` will be expressed in terms of a type parameter of kind {`Type * Type`}, the kind of records of pairs of types. The convention is that each pair `(native, db)` describes a column represented natively in type `native` but converted to type `db` for database insertion. Each column must have an associated function for translating from `native` to `db`, and we use a type function `arrow` to express that. The type-level functions `fst` and `snd` from the standard library project the first and second elements of type-level pairs, respectively.

```
type arrow (dom :: Type, ran :: Type) = dom -> ran
val toDb : r :: {Type * Type} -> folder r
  -> $(map arrow r) -> table (map snd r)
  -> $(map fst r) -> unit
```

We can implement `toDb` using mostly the same techniques as from our last example. We fold over the record of type pairs, building a value-level record that is a suitable parameter to `insert`. We use the value-level operator `++`, which implements record concatenation.

```
fun toDb [r :: {Type * Type}] (fl : folder r)
  (mr : $(map arrow r)) (tab : table (map snd r))
  (x : $(map fst r)) =
  insert tab
    (fl [fn r => $(map arrow r) -> $(map fst r)
          -> $(map (fn p => exp [] (snd p)) r)]
      (fn [nm] [p] [r] [[nm] ~ r] acc mr x =>
        {nm = const (mr.nm x.nm)}
        ++ acc (mr -- nm) (x -- nm))
      (fn _ _ => {}) mr x)
```

There is a subtlety in type-checking the definition of `toDb`. The result type of the fold over `r` is not in the right form to be a valid argument to `insert`. The Ur type inference engine applies this type equality implicitly:

```
  $(map (fn p => exp [] (snd p)) r)
= $(map (exp []) (map snd r))
```

This is a corollary of a more general fusion law:

```
map f (map g r) = map (fn x => f (g x)) r
```

In all related systems that we are aware of, the programmer would need to apply an explicit coercion to make use of this law. The coercion would most likely appeal to an explicit inductive proof of the fusion law. For a general-purpose language intended to be used in correctness verification, it is not clear how to do any better. However, since Ur is only intended to handle reasoning about records and names, we can streamline the programming process.

The fusion law and a handful of other algebraic identities are built into our inference engine, and they are applied automatically during unification whenever possible. The formal presentation of Ur in Section 3 gives the complete list of laws that we have added. Our present implementation only includes five laws that would be proved by induction in traditional dependently-typed programming. These laws have been sufficient to avoid any proofs about type equality in all of the Ur/Web case studies we have undertaken.

Our `toDb` function has a type even more involved than that of last subsection's `mkTable` function. It is unlikely that many non-expert programmers are prepared to deal with types of this complexity. Luckily, implicit arguments and reverse-engineering unification do not fail us. The following example code is sufficient to instantiate `toDb` at a specific record type.

```
fun addInts (n : int, m : int) = n + m
val inserter =
  toDb {A = addInts, B = fn x : float => x}
```

Ur infers that the type of `inserter` is

```
table [A = int, B = float]
  -> {A : int * int, B : float} -> unit
```

Reverse-engineering unification found that the proper value for `r` is `[A = (int * int, int), B = (float, float)]`.

2.3 Building Typed Expressions

Many database operations accept predicates over the columns of a table. For instance, the SQL delete command removes all rows of a table that satisfy a particular user-specified predicate. To interface with a database, it can be useful to convert a record of values into a database expression that characterizes those table rows whose columns match the record. A function to do this generically will be our final worked example, and the implementation will demonstrate how more complicated richly-typed abstract syntax trees may be built generically.

We will need a few more of the constructors for the `exp` type. The following functions reference a database column, compare two expressions for equality, and form the conjunction of two boolean expressions, respectively.

```
val column : nm :: Name -> t :: Type -> r :: {Type}
  -> [[nm] ~ r] => exp ([nm = t] ++ r) t
val eq : r :: {Type} -> t :: Type
  -> exp r t -> exp r t -> exp r bool
val and : r :: {Type}
  -> exp r bool -> exp r bool -> exp r bool
```

We are able to give our generic function, `selector`, a type that is simple in comparison to those from the earlier examples. The implementation of the function is interesting because it performs a fold with an accumulator type that involves an explicit record disjointness assertion.

```
fun selector [r :: {Type}] (fl : folder r) (x : $r)
    : exp r bool =
  fl [fn r => $r -> rest :: {Type} -> [rest ~ r] =>
       exp (r ++ rest) bool]
     (fn [nm] [t] [r] [[nm] ~ r]
       acc x [rest] [rest ~ r] =>
       and (eq (column [nm]) (const x.nm))
           (acc (x -- nm) [[nm = t] ++ rest] !))
     (fn _ [rest] [rest ~ []] => const True)
     x [[]] !
```

At each stage of folding through the record `r`, our accumulator is a function. Its first argument is a record containing one field for every field of `r` that *we have already folded over*. The next argument, `rest`, is a type-level argument, which is meant to be instantiated to *those fields that we have not yet folded over*. In the course of the fold, we gradually shift fields "from `rest` to `r`," until at the end `rest` may be the empty record. After binding `rest`, the accumulator type includes an explicit assertion that `rest` and `r` share no field names, which is a prerequisite of being able to concatenate these records. We include just such a concatenation in the result type of accumulator functions, which is the type of boolean expressions that may mention columns included in either of `r` or `rest`.

The step function used in the fold takes many arguments, but it mostly uses features that we have already seen. The interesting part is in the application of the accumulator `acc`. We need to choose the right `rest` record to pass to it. This turns out to be the current `rest` value, extended with the current field mapping from `nm` to `t`. We write `!` to denote a "proof" of the disjointness assertion in `acc`'s type. As always in Ur, there are no proof terms; rather, the `!` is just a signal that the inference engine should prove the assertion automatically. This proof is assembled from the disjointness assertions `[nm] ~ r` and `rest ~ r` that are available in the typing context.

Last subsection's example demonstrated the Ur inference engine's smarts in reasoning about equality of records, through the automatic application of algebraic equivalences. Our new example showcases the other kind of domain-specific reasoning in inference, which is automatic proof of record disjointness facts. The disjointness prover is able to prove a wide range of implications involving record concatenation and mapping, without burdening the programmer with any of the details.

3. Syntax and Semantics of Featherweight Ur

We hope that the previous section's examples have motivated why Ur has the features that it does. In this section, we refine that design down to its core elements, presenting a formal definition of the idealized language Featherweight Ur.

3.1 Syntax

Figure 1 presents the syntax of Featherweight Ur, which is influenced heavily by System F_ω [22], the higher-order polymorphic lambda calculus. This syntax closely follows what we have seen in ASCII format in the examples; we hope that the correspondence is plain. One change that we make is referring to the general class of compile-time values as *constructors* rather than types. Types are the subset of constructors that have kind Type. We try to use metavariables τ for types and c for constructors that may not be types. Guarded types, which we have also referred to as disjoint-

$$
\begin{array}{rcll}
\text{Kinds} & k & ::= & \mathsf{Type} \mid \mathsf{Name} \mid k \to k \mid \{k\} \\
\text{Constructors} & c, \tau & ::= & \tau_1 \to \tau_2 \mid \alpha \mid \forall \alpha :: k.\ \tau \mid c\ c \\
& & & \mid \lambda \alpha :: k.\ c \mid \#n \mid \$c \mid []_k \mid [c = c] \\
& & & \mid c \mathbin{+\!\!+} c \mid \mathsf{map}_{k,k} \mid [c \sim c] \Rightarrow \tau \\
\text{Expressions} & e & ::= & x \mid e\ e \mid \lambda x : \tau.\ e \mid e\ [c] \mid \Lambda \alpha :: k.\ e \\
& & & \mid \{\} \mid \{c = e\} \mid e.c \mid e - c \mid e \mathbin{+\!\!+} e \\
& & & \mid [c \sim c] \Rightarrow e \mid e\ !
\end{array}
$$

Figure 1. Syntax of Featherweight Ur

ness assertions, are written $[c_1 \sim c_2] \Rightarrow \tau$, with analogous notation for guarded expression abstraction.

In a general-purpose dependently-typed language like Agda, we would build a type of records and its associated operations from first principles. In contrast, Ur omits the traditional facilities for inductive and recursive definitions, instead building the key kinds and type-level operators into the language. Type-level concatenation and mapping are the sole type-level computation facilities that would be implemented as recursive definitions in Agda. This approach is entirely compatible with viewing Ur as a convenient surface language for Coq or Agda, such that it is possible to fall back on the more expressive language when Ur's feature set is insufficient. However, it seems desirable to stick to this restricted fragment because we have been able to implement an effective type inference procedure for it, as described in the next section.

Our experience writing programs in Ur/Web suggests that the feature set we have chosen is more than sufficient for our application domain. It is also true that a few apparent omissions have solid theoretical justification. For instance, it may seem more natural to choose "fold" over "map" as a primitive traversal. However, in combination with considering records as *unordered* sets of keys and values, this would lead to an unsound semantics, unless we increased the language complexity to the point where it could be proved that any function used with "fold" is insensitive to the order in which record fields are visited. The choice of record field disjointness as the sole variety of constraint may also seem arbitrary. However, from this base, it is easy to define other constraints, including record equality and inclusion. In fact, the Ur/Web standard library relies critically on such constraints to encode the typing rules of SQL.

3.2 Static Semantics

Figure 2 gives selected rules of the *kinding* judgment, which assigns kinds to constructors. We omit those rules already used by F_ω. To simplify the presentation, we assume side conditions for each rule asserting the well-typedness of all expressions and well-kindedness of all constructors that appear. We define all of our judgments in terms of a single variety of context Γ. Entries in such contexts are kinding assertions $\alpha :: k$, typing assertions $x : \tau$, or row disjointness assertions $c_1 \sim c_2$. The kinding judgment only uses the first and last of these assertion sorts. We hope that the rules are intuitive, following our examples.

The two interesting cases are for row concatenation and guarded constructors. To concatenate rows c_1 and c_2, it must be proved that c_1 and c_2 share no field names. This is captured by the judgment $\Gamma \vdash c_1 \sim c_2$. In the rule for guarded types $[c_1 \sim c_2] \Rightarrow \tau$, within the body τ, the context is extended with the fact that c_1 is disjoint from c_2. We omit here the details of the disjointness judgment; it simply captures decomposition of each side of the constraint into irreducible pieces and checking of disjointness between all pairs of pieces.

This decomposition is phrased via a *definitional equality judgment* $c \equiv c'$, which encodes the computational semantics of constructors. In checking disjointness, we may replace any constructor with another that is computationally equivalent. Figure 3 presents the definitional equality. On the first line of the figure, we include the standard rules from F_ω.

The next set of rules defines the semantics of the basic row operations. We have that the empty record is an identity element for concatenation, and that concatenation is commutative and associative. After this, we have the semantics of map. The rules for map mostly mirror a list map definition in Haskell.

Perhaps the most surprising set of rules comes in the last two lines of the figure. Within the definitional equality, we have row equivalents of standard theorems about higher-order list functions. In order, the last three rules of Figure 3 express the fact that map applied to an identity function is itself an identity function, the distributivity of map over concatenation, and a fusion law for one map over another. One of our (perhaps surprising) empirical results is that these are the only laws we have needed to implement a variety of practical metaprograms.

Finally, we come to the typing rules, with selected rules shown in Figure 4. Most of the novelties of Featherweight Ur have already come up in relation to the previous judgments. One rule makes the typing judgment a congruence over the definitional equality; when e has type τ, it also automatically has type τ', for any $\tau' \equiv \tau$. The typing rule for guarded expressions $[c_1 \sim c_2] \Rightarrow e$ shows that they have types like $[c_1 \sim c_2] \Rightarrow \tau$; we apply the rule for the ! operator to reduce this type to simply τ, when the constraint is provable.

Like other languages with symmetric concatenation of records, Ur lacks subtyping, since this could lead to ambiguous situations where it is not clear which of two records being concatenated to drop a field from. However, many of subtyping's common usage patterns can be encoded with polymorphism over type-level records.

3.3 Dynamic Semantics

Rather than defining an operational semantics for Featherweight Ur, we give an elaborative semantics, translating Featherweight Ur programs into terms of the Calculus of Inductive Constructions, the logic behind the Coq proof assistant [3]. We do not have space to go into the details here. The exact translation is available in the `src/coq` directory of the Ur/Web distribution, as part of a Coq formalization of Featherweight Ur syntax and semantics.

The basic idea is that we translate kinds to CIC types, constructors to terms of those types, and typing derivations to terms of those further types. A derivation $c \equiv c'$ is compiled to a proof that c and c' have equal denotations. Our implementation of records is a standard exercise in programming with heterogeneous list types, a common tool in dependently-typed languages.

It is worth stating explicitly that, since we give our semantics elaboratively, there is no need to prove a separate type soundness theorem. The translation is implemented in Coq and outputs native Coq terms directly, so we get type preservation by construction. Since CIC has been proved type-sound, Featherweight Ur is type-sound, too, almost by definition. This formalization also inherits other properties of CIC, like strong normalization [20], that do not hold of the full Ur language.

This elaboration is meant only to specify the semantics of Ur, rather than an implementation technique. The actual Ur/Web compiler works more traditionally, as sketched in Section 5.

4. Effective Type Inference

Ur, or even plain F_ω, includes type system features that are rarely found outside of programming languages based on dependent type theory. Type inference for CIC and other such systems is unde-

$$\frac{}{\Gamma \vdash \#n :: \mathsf{Name}} \quad \frac{\Gamma \vdash c :: \{\mathsf{Type}\}}{\Gamma \vdash \$c :: \mathsf{Type}} \quad \frac{}{\Gamma \vdash []_k :: \{k\}} \quad \frac{\Gamma \vdash c_1 :: \mathsf{Name} \quad \Gamma \vdash c_2 :: k}{\Gamma \vdash [c_1 = c_2] :: \{k\}} \quad \frac{\Gamma \vdash c_1 :: \{k\} \quad \Gamma \vdash c_2 :: \{k\} \quad \Gamma \vdash c_1 \sim c_2}{\Gamma \vdash c_1 +\!\!+ c_2 :: \{k\}}$$

$$\frac{}{\Gamma \vdash \mathsf{map}_{k_1,k_2} :: (k_1 \to k_2) \to \{k_1\} \to \{k_2\}} \quad \frac{\Gamma \vdash c_1 :: \{k_1\} \quad \Gamma \vdash c_2 :: \{k_2\} \quad \Gamma, c_1 \sim c_2 \vdash \tau :: \mathsf{Type}}{\Gamma \vdash [c_1 \sim c_2] \Rightarrow \tau :: \mathsf{Type}}$$

Figure 2. Selected kinding rules of Featherweight Ur

$$\frac{}{(\lambda\alpha :: k.\ c_1)\ c_2 \equiv c_1[\alpha \mapsto c_2]} \quad \frac{}{c \equiv c} \quad \frac{c_2 \equiv c_1}{c_1 \equiv c_2} \quad \frac{c_1 \equiv c_2 \quad c_2 \equiv c_3}{c_1 \equiv c_3} \quad \frac{c \equiv c'}{C[c] \equiv C[c']} \quad \frac{}{[]_k +\!\!+ c \equiv c} \quad \frac{}{c_1 +\!\!+ c_2 \equiv c_2 +\!\!+ c_1}$$

$$\frac{}{c_1 +\!\!+ (c_2 +\!\!+ c_3) \equiv (c_1 +\!\!+ c_2) +\!\!+ c_3} \quad \frac{}{\mathsf{map}_{k_1,k_2}\ f\ []_{k_1} \equiv []_{k_2}} \quad \frac{}{\mathsf{map}_{k_1,k_2}\ f\ ([c_1 = c_2] +\!\!+ c_3) \equiv [c_1 = f\ c_2] +\!\!+ \mathsf{map}_{k_1,k_2}\ f\ c_3}$$

$$\frac{}{\mathsf{map}_{k,k}\ (\lambda\alpha : k.\ \alpha)\ c \equiv c} \quad \frac{}{\mathsf{map}_{k_1,k_2}\ f\ (c_1 +\!\!+ c_2) \equiv \mathsf{map}_{k_1,k_2}\ f\ c_1 +\!\!+ \mathsf{map}_{k_1,k_2}\ f\ c_2}$$

$$\frac{}{\mathsf{map}_{k_2,k_3}\ f\ (\mathsf{map}_{k_1,k_2}\ f'\ c) \equiv \mathsf{map}_{k_1,k_3}\ (\lambda\alpha :: k_1.\ f\ (f'\ \alpha))\ c}$$

Figure 3. Definitional equality rules of Featherweight Ur

$$\frac{\tau \equiv \tau' \quad \Gamma \vdash e : \tau'}{\Gamma \vdash e : \tau} \quad \frac{}{\Gamma \vdash \{\} : \$[]_{\mathsf{Type}}} \quad \frac{\Gamma \vdash c :: \mathsf{Name} \quad \Gamma \vdash e : \tau}{\Gamma \vdash \{c = e\} : \$[c = \tau]} \quad \frac{\Gamma \vdash e : \$([c = \tau] +\!\!+ c')}{\Gamma \vdash e.c : \tau} \quad \frac{\Gamma \vdash e : \$([c = \tau] +\!\!+ c')}{\Gamma \vdash e - c : \$c'}$$

$$\frac{\Gamma \vdash e_1 : \$c_1 \quad \Gamma \vdash e_2 : \$c_2 \quad \Gamma \vdash c_1 \sim c_2}{\Gamma \vdash e_1 +\!\!+ e_2 : \$(c_1 +\!\!+ c_2)} \quad \frac{\Gamma \vdash c_1 :: \{k_1\} \quad \Gamma \vdash c_2 :: \{k_2\} \quad \Gamma, c_1 \sim c_2 \vdash e : \tau}{\Gamma \vdash [c_1 \sim c_2] \Rightarrow e : [c_1 \sim c_2] \Rightarrow \tau} \quad \frac{\Gamma \vdash e : [c_1 \sim c_2] \Rightarrow \tau \quad \Gamma \vdash c_1 \sim c_2}{\Gamma \vdash e\,! : \tau}$$

Figure 4. Selected typing rules of Featherweight Ur

cidable, seen via fairly straightforward arguments: general mathematical proof search can be reduced to type inference in such rich type systems, with unification variables standing for mathematical proofs encoded syntactically. Even type inference for System F has been proved undecidable [32], and impredicative (or "first-class") polymorphism is crucial to Ur's usefulness, as the example of `folder` functions demonstrates. To this already undecidable base, Ur adds the type-level computation features of F$_\omega$ and type-level `map`. There may very well be worthwhile theoretical completeness results for Ur that fall short of providing full inference, but we leave such results for future work. In this section, we present the type inference heuristics that we have had success with.

Like in some proposed solutions to the type inference problem for System F, we require that all polymorphism be annotated explicitly at the definitions of functions. In practice, most type arguments may be inferred, making *uses* of polymorphic functions look similar to uses in ML, while *definitions* may be considerably more verbose.

We follow the usual approach of type-checking expressions by introducing unification variables, whose values are determined later during unification of constructors. Since our type system is more complex than those handled by classic Hindley-Milner inference, we must do more than just solve type equality constraints as they appear. We follow more recent formulations based on systems of constraints. Type-checking generates a set of constructor equality and record disjointness constraints. As an optimization, we try to solve constraints when they are first generated, but the general case involves building a global set of constraints and then iterating through finding an immediately-solvable constraint, until no constraints remain.

Most of Ur constructor unification could be implemented by normalizing constructors and then comparing normal forms for simple syntactic equality. We can refactor the definitional equality rules of Figure 3 so that, when applied only left-to-right, they form a rewrite system that we conjecture is terminating and confluent. This requires removing at least the concatenation commutativity rule, so we handle row unification in a special way that we will describe shortly. For the other constructor language elements, rather than following the naive normalize-and-compare unification strategy, we apply a standard optimization. We use a loop of reducing constructors to *head normal form*, where we reduce only as much as is needed to expose top-level structure. Head normal forms are compared syntactically, where unification of subterms appeals to the original algorithm, which will head-normalize and compare those subterms, and so on. There is no doubt further opportunity for optimizing type inference performance by applying techniques from the type-preserving compilation literature [28].

Higher-order unification is a well-studied subject with some standard heuristic approaches [21]. There, the key problem is inferring type-level functions. In contrast, the Ur/Web implementation has more in common with the GHC Haskell compiler, in that only first-order unification techniques are used to make a best effort at guessing functions. This is because we have not found the classical higher-order unification techniques to be critical in our setting of metaprogramming. Instead, our inference engine focuses on understanding type-level records and computations over them. Since Ur *does* support general type-level λ, the result is easy-toobserve inference incompleteness that still tends not to cause trouble in practice. For instance, our inference engine is unable to type the following code.

```
fun id [f :: Type -> Type] [t] (x : f t) : f t = x
val x = id 0
```

4.1 Proving Disjointness

All proofs of row disjointness happen automatically. Whenever a new known constraint is introduced via an expression like `fn [r1 ~ r2] => e`, our type-checker calculates all atomic disjointness facts implied by `r1 ~ r2`. Each of `r1` and `r2` is decomposed using a function D defined as follows, where `hnf` is the constructor head normalization function.

$$\begin{array}{rcl} D(c) & = & D'(\mathtt{hnf}(c)) \\ D'([c_1 = c_2]) & = & \{[c_1]\} \\ D'(c_1 \mathbin{+\!\!+} c_2) & = & D(c_1) \cup D(c_2) \\ D'(x) & = & \{x\} \\ D'(\mathtt{map}\ f\ c) & = & D(c) \\ D'(\_) & = & \emptyset \end{array}$$

The calls to D yield two finite sets, and we calculate the symmetric closure of their Cartesian product to add to the typing context. When we encounter a disjointness goal `r1 ~ r2`, we decompose these rows with the same function and again take the Cartesian product of the results. This time, we check that every resulting pair is either in our database of facts or consists of two singleton rows with constant, distinct field names. In checking constraints, the last, "wildcard" case of the definition of D' must be changed to instead signal that the constraint is not provable yet. In such cases, we hope that when we revisit this constraint after solving other constraints first, some unification variables will have been determined, so that the proof can finish successfully.

4.2 Reverse-Engineering Rows

Another key element of our type inference process is the reverse-engineering unification that we have mentioned several times. Sometimes we want to infer implicit arguments to polymorphic functions whose types contain maps. This often leads to inference queries of the form $\mathtt{map}_{k_1,k_2}\ f\ \alpha \equiv c$, for some unification variable α and (usually ground) constructor c. If c is empty, then we can set α to be empty, too. When $c = [c_1 = c_2] \mathbin{+\!\!+} c_3$, to choose a value for α, we can generate fresh unification variable α' and then unify $f\ \alpha'$ with c_2. Afterward, we can replace α by $[c_1 = \alpha'] \mathbin{+\!\!+} \alpha''$ for some new α''. We can repeat this process to reverse-engineer the value of α in a wide variety of cases.

4.3 Unifying Rows

During unification, when the standard algorithm finds a row operator like concatenation at the top level of one of the constructors that it is comparing, it switches to using a special row unification algorithm. First, each constructor is summarized using a summary function S from constructors to triples of sets. S works like the D function in decomposing a record, and the triples it outputs break a record's components into singleton field mappings, unification variables, and other miscellaneous components.

To unify two records, we first consider each component of their summarizing triples separately, looking for unifiable pieces between the ith component of the first summary and the ith component of the second summary. Any such unifications trigger "crossing off" of components on both sides. If both sides are now empty, we are done. If either side is reduced to a single unification variable, then we finish by replacing it everywhere with the other side's contents. If each side is reduced to a single unification variable plus zero or more singleton fields, and if the singleton fields on one side do not overlap with those on the other, then each of the two unification variables may be rewritten in terms of a single new variable. When none of these rules apply, we make a last attempt to apply reverse-engineering unification. If iterating these rules leaves any contents on either side of the equation, we remember the new unification variable values that we learned, but we leave this equality in the set of unsolved constraints.

4.4 Generating Folders

Instances of the `folder` type family from Section 2 may be omitted, in which case the compiler waits to generate concrete instances until after type inference is complete and the type of every subterm of the program is known. Since the type inference process never commutes the order of fields in a record type unnecessarily, the order of record fields in the elaborated program is easy to predict, based on the order in which fields were written in the source code. Because of this, it works well for the compiler to generate any unknown folder using the permutation implied by the order in which fields appear in the folder's inferred type.

5. The Ur/Web Compiler

We have used the generic Ur type inference engine to implement Ur/Web, a domain-specific language for Web application development. To the programmer, Ur/Web appears as a special standard library for Ur, plus helpful parsing extensions supporting features like inline XML and SQL code. Under the hood, the Ur/Web compiler is specialized to deal with this library and with the requirements of real Web browsers and database servers.

Even setting metaprogramming aside, we found many uses for Ur's facility for building type-level records with mapping. To support richly-typed versions of the standard structures that Web applications manipulate, we did not need to write any custom type inference code. Instead, we encoded those structures in the signature of the main module of the standard library. In particular, Ur records show up throughout the encodings of the syntax and typing rules of SQL queries and commands and HTML documents.

Our experience applying Ur/Web suggests that any application using these features, written without polymorphism, can be type-checked with no more type annotation than is needed when representing these structures as strings. Moreover, the inference engine is quite effective at dealing with polymorphic and metaprogramming uses of these library types.

As we use rich tree types to classify any structure that browsers will interpret, Ur/Web applications are automatically immune to cross-site scripting, code injection, and several other of the most common security vulnerabilities. One might worry that we need to trade performance for this benefit, due to the use of an advanced type system. However, we compile Ur/Web programs with a whole-program optimizing compiler. In the tradition of MLton[3], we eliminate all polymorphism at compile time, which, given the complexity of our type system, requires simple partial evaluation by reduction. This produces an intermediate form that is much like ML.

6. Case Studies

In this section, we discuss some case studies centered on Ur/Web versions of common metaprogramming components from the world of Web application frameworks. We highlight interesting uses of type-level `map`, the most distinguishing feature of Ur compared to other systems that require little annotation. The message

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

of this section is that, despite the relatively minimalistic set of type-level features we chose for Ur, each of these case studies meets our two main design criteria of "no proofs for library writers" and "no fancy types for rank-and-file programmers."

Object-Relational Mapping Many applications maintain a dual view of SQL database tables. There is a view where table rows are represented with the programming language's native records or objects; and there is the database view, which can only be manipulated via queries and commands sent to the database server. The popular Web frameworks provide their own, untyped implementations of such *object-relational mapping*, or ORM. We implemented ORM as a richly-typed generic component in Ur/Web. Here are two example invocations of our component. These examples use the Ur module system, which is inspired by the ML module systems [16], with few surprises encountered in adapting that idea to Ur's base language.

```
structure T
  = Table(struct
            val cols = {A = local [int],
                        B = local [string]}
          end)

structure S
  = Table(struct
            val cols = {C = T.id,
                        D = local [float]}
          end)
```

We build ORM modules customized to two specific tables. The identifier `Table` names a functor, i.e., a function from modules to modules. The `Table` functor is a function from a module describing a table to a module implementing the classic ORM operations: listing all rows and adding, deleting, modifying, or looking up a row. All operations work directly on native Ur records. The argument used to build module `T` says that we want a table with a column `A` of type `int` and a column `B` of type `string`. The argument used for `S` dictates that there be a column `D` of type float and a column `C` that is a *foreign key* reference to rows of `T`. Module `S` thus contains a function for retrieving the `T` record associated with an `S` record.

Input modules to `Table` must contain more type-level information than is included explicitly here. Ur includes an extension to the ML module system paradigm, where type-level module components may be omitted to ask that they be inferred. In this way, the client of a metaprogram can be shielded from the complexity of its type. To support foreign keys, we require that a table be described in a terms of a record of kind `{Type * Type}`, where each field is associated both with its own type and with the type of the table it references, if it is a foreign key. The foreign key link-following function is typed in terms of a map over this record. We also use `map` to support an abstract type of columns specific to a table. These columns may be combined to form predicates in a table-specific abstract type of filters, and filters may be passed to lookup and search functions.

Versioned Database Access In some applications, it is important not only to be able to query the current state of a database, but also to "roll back" to a past view of the database at a particular time. We implemented a versioned database access component that provides this functionality generically, for any set of table columns. Normal database access is just as easy as with the ORM component, but a table-specific abstract type of versions may be used to look up old values. Output modules of our versioning functor provide functions for listing all versions that exist and for looking up an old row by version and key.

The versioning functor takes two type-level records as input. The first describes which rows should collectively be considered the *primary key* of the table, such that they are enforced to be unique across rows. The second type-level record describes the remaining columns. Our concrete SQL implementation of the versioning abstraction involves a table whose columns consist of a version ID, the key columns, and a *nullable* version of each non-key column. The idea is that each update to a virtual row adds a new concrete row where every non-key column that has not changed is represented as `NULL`, while those columns that *have* changed have their new values recorded. We represent the type of the concrete database table with a type-level `map` over the non-key record, replacing each type `t` with `option t`. This example also tests the versatility of our domain-specific proving by including types based on the concatenation of the key and non-key records, which are asserted to be disjoint in an explicit constraint.

Database Admin Interface The most popular Ruby on Rails metaprogram builds a standard interface for administering an arbitrary database table, including viewing and modifying its contents in a Web browser via HTML tables and forms. We implemented comparable functionality as an Ur/Web functor. The functor may be used by providing just an SQL table reference, a string to display as the page title, and a record of metadata for each table column. Metadata values for common types can be built with expressions like `int "A"` and `float "B"` (passing a display name for the column), and support is provided for building custom metadata for domain-specific column handling.

Each column is associated with a pair of types, giving its SQL and client-side representations. Maps over this record of pairs are used to calculate types for the database table and for the widget environments that occur in HTML forms.

Web 2.0 Admin Interface We also implemented a *batched* counterpart to the last component. This modernized version takes advantage of the possibility to run some code in Web browsers as JavaScript. In particular, when the user submits a form asking to add a new row, the remote Web server is not contacted. Instead, local code adds the new row data to an HTML table. The user can click a button to submit his batched changes en masse. This functor may be used in the same way as its Web 1.0 ancestor, despite the fact that we are checking a more complicated piece of code. Every implementation that our functor outputs includes new URL-addressed *remote procedure calls* (RPCs) for client-server communication, and types guarantee that any functor output uses RPCs correctly.

This functor is similar to the previous example in involving a record assigning each table column an SQL version and a client-side version. To come up with the proper type for the RPC that adds a list of rows, we need to map over this record, forming a table-specific type of records that must be passed to the RPC in serialized form.

In-Browser Spreadsheet Another Ur/Web component implements common spreadsheet functionality, such that most code is run in browsers, but the remote Web server is contacted to query and modify a database table storing the persistent version of a spreadsheet. The functor supports foreign keys represented as automatically-populated dropdown listboxes, and each spreadsheet application provides input validation, summary rows displaying aggregate information, paging, sorting, per-column filtering, and an access point that other program modules can use to check which rows of the spreadsheet the user has selected. Here is a simple example of constructing a spreadsheet implementation.

```
table t : {Id : int, A : int, B : bool}
sequence s
```

Component	Int.	Imp.	Disj.	Id.	Dist.	Fuse
ORM	40	77	580	-	13	5
Versioned	20	122	616	6	4	2
Table Admin	22	158	1412	-	1	2
Web 2.0 Admin	21	134	1105	-	1	1
Spreadsh. (base)	46	291	1667	6	-	1
Spreadsh. (SQL)	110	391	1257	3	11	-

Figure 5. Code sizes (in lines of code) of case study components' interfaces and implementations, along with invocation counts for critical pieces of type inference

```
open Make(struct
  val tab = t
  con key = [Id = _]

  val raw = {Id = init (nextval s),
             A = init (return 0),
             B = init (return False)}

  val cols = {Id = readOnly [#Id] "Id" int,
              A = editable [#A] "A" int,
              B = editable [#B] "B" bool,
             DA = computed "2A" (fn r => 2 * r.A)}

  val aggregates = {Sum = fold (fn r n => r.A + n) 0,
                    AllTrue =
                      fold (fn r b => r.B && b) True}
end)
```

The `key` component identifies which table columns make up the table key. The `raw` record explains how to generate the initial value of each column of a fresh row. The `cols` record contains metadata values for the four columns to be displayed in our spreadsheet: a read-only rendering of the key `Id`, widgets for editing the values of the `A` and `B` columns, and a computed column that always shows twice the current value of `A`. The final record, `aggregates`, requests to include a summary row at the bottom of the spreadsheet, where we are told the sum of all `A` values and the iterated boolean "and" of all `B` values.

The main type-level record behind this component has kind `{Type * Type * Type}`. Each displayed column is associated with a type of global state, a type returned by its main input widget, and a type returned by its filtering widget. Many applications of `map` are used throughout the functor to express the typing relationship among these different elements. We reduce the complexity of our code by first building a functor for constructing spreadsheets backed by arbitrary data sources, and we then derive a functor for spreadsheets backed by SQL databases.

6.1 Evaluation

As far as we know, ours is the first investigation into statically-typed instances of this variety of practical metaprogramming. It was not obvious at first that it would be possible to write such programs without violating one or both of our two central design principles:

1. Metaprograms should contain no proofs or other type-cast expressions with no computational effects.
2. Client code should contain no proofs or types more complicated than types found in mainstream programming languages.

Nonetheless, all of our case studies fit this description, as do all of the other Ur/Web metaprograms we have written.

Figure 5 summarizes the amount of code needed to implement our components, measured in lines with content besides whitespace and comments. We also gathered some statistics that give a sense of how much annotation effort Ur/Web saves the programmer over similar coding in related statically-typed systems, which all require proof terms to apply algebraic identities that Ur inference applies automatically. For each component in the Figure 5, we note how many times the main type inference procedure invoked the disjointness prover, along with how many times inference applied the map-over-identity-function, map distributivity, and map fusion laws, respectively. These numbers consider only the generic components; client code usually triggers additional uses of the laws.

7. Related Work

The design of Ur was influenced heavily by our experience with programming in Coq [3]. The possibilities for generic programming in dependently-typed languages have been recognized and implemented for several years now, at least [1]. This body of work has tended to focus on more involved examples like generation of parsers and pretty-printers for arbitrary algebraic datatypes. We mean to argue that Ur, by focusing on a specific domain, provides a much more user-friendly experience to programmers, both attracting a broader range of developers and enhancing productivity of those who *are* attracted. Dependent ML [33] follows a similar path, with convenient automated reasoning that is mostly restricted to formulas of linear arithmetic.

There have been many investigations into the inclusion of extensible records in statically-typed languages. Wand's initial work on row types [31] has inspired many follow-ups. The work of Rémy [24] is also well-known, as it has directly influenced the object system and polymorphic variant facilities of Objective Caml. Ohori [18] developed a compiler for a language with extensible records, demonstrating an index-passing encoding that facilitates separate compilation. Harper and Pierce [10] defined a calculus supporting general record concatenation, via row-quantified types that include general disjointness constraints like those in Ur, but without discussion of type inference. Gaster and Jones [9] define a system for extensible records and variants, achieving a complete type inference algorithm by restricting constraints to the form "label l is not present in row r." Pottier [23] demonstrated a general type inference system equipped to deal with general record concatenation and first-class names. Blume et al. implemented the MLPolyR language [4], which, using type-level records, exploits the duality of records and variants to support an extensible case construct. The idea of extensible records is a natural one, and it has appeared in many other cases that we do not have space to cite. As far as we are aware, every construction with records that can be coded in these past languages can also be coded in Ur, with no additional type annotation needed at uses of polymorphic functions, though polymorphism must be annotated explicitly in function definitions. The crucial facility distinguishing Ur from this past work is type-level computation, in the form of F_ω features and type-level map.

Embedding SQL syntax in general-purpose languages has been studied before, with various levels of static assurance. Ohori and Buneman [19] added explicit support for typing associated with database operations to an ML-like language while maintaining principal typing. Leijen and Meijer [15] embedded a subset of SQL in an extension of Haskell, with static validation of a subset of the properties enforced by Ur/Web. Silva and Visser [30] later completed a similar project with broader static validation, using more of the harnessing of type classes with functional dependencies that has become very popular in the Haskell community. The HList library [13] for GHC Haskell is a prominent example of this trend; it provides extensible records, using a notion of type-level computation driven by Haskell's type class resolution mechanisms. We (somewhat subjectively) feel that this style leads to code that is needlessly more complicated and verbose than what is possible in Ur. It is also unclear how to handle in Haskell applications of the kinds of algebraic laws that are at the heart of Ur's type inference; at best, it seems that explicit proof terms must be written.

There is a long history of code generation in the worlds of object-oriented and procedural programming, and the standard techniques in this area suffer from lack of static validation of metaprograms. Recent language extensions like Compile-Time Reflection (CTR) [8] for C# and MorphJ [11] for Java address this shortcoming for programs that inspect and generate classes in stylized ways. Similar issues of name disjointness checking arise in these tools. One significant advantage of Ur's approach is that metaprograms are not only checked statically, but they are also assigned self-contained types, which makes it possible for functions to abstract over metaprograms in a statically-safe way. Another difference is that Ur is built from simple, orthogonal constructs of type theory, which can make it easier to see the essence of metaprogramming with names.

8. Conclusion

While novice programmers can use Ur metaprograms without writing fancy types, erroneous metaprogram applications can trigger hard-to-understand error messages that do use advanced concepts explicitly. Improved heuristics for phrasing these messages would be a useful subject for future work. We are also still investigating the limitations that arise from use of a compiler that must resolve all polymorphism statically. This policy limits opportunities for constructing syntax of dynamically-varying type; for instance, to access different database tables based on values read from a configuration file. Perhaps genuine dependent types will even prove crucial in supporting such use cases.

Ur/Web is already a practical system for implementing modern Web applications with metaprogramming. Programs that write programs are notoriously hard to debug, and Ur helps reduce development cost by using static types to guarantee validity of metaprograms. We built on the rich body of work on dependent type theory and added just a few domain-specific conveniences, in the form of a specialized type inference engine. This "last mile" effort makes a crucial difference in building a tool to be competitive with the tools used in the Web application domain today, where few programmers are willing to write formal proofs just to get programs to type-check.

Acknowledgments

We thank Manu Sridharan, Ryan Wisnesky, and the anonymous referees for helpful feedback on drafts of this paper.

References

[1] Thorsten Altenkirch and Conor McBride. Generic programming within dependently typed programming. In *Proc. IFIP TC2/WG2.1 Working Conference on Generic Programming*, 2003.

[2] Lennart Augustsson. Cayenne - a language with dependent types. In *Proc. ICFP*, 1998.

[3] 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.

[4] Matthias Blume, Umut A. Acar, and Wonseok Chae. Extensible programming with first-class cases. In *Proc. ICFP*, 2006.

[5] Chiyan Chen and Hongwei Xi. Combining programming with theorem proving. In *Proc. ICFP*, 2005.

[6] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George Necula. Dependent types for low-level programming. In *Proc. ESOP*, 2007.

[7] Web Application Security Consortium. 2007 Web application security statistics. `http://projects.webappsec.org/Web-Application-Security-Statistics`.

[8] Manuel Fähndrich, Michael Carbin, and James R. Larus. Reflective program generation with patterns. In *Proc. GPCE*, 2006.

[9] Benedict R. Gaster and Mark P. Jones. A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, University of Nottingham, 1996.

[10] Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. In *Proc. POPL*, 1991.

[11] Shan Shan Huang and Yannis Smaragdakis. Expressive and safe static reflection with MorphJ. In *Proc. PLDI*, 2008.

[12] Mark P. Jones. Type classes with functional dependencies. In *Proc. ESOP*, 2000.

[13] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Proc. Haskell Workshop*, 2004.

[14] Kenneth Knowles, Aaron Tomb, Jessica Gronski, Stephen N. Freund, and Cormac Flanagan. Sage: Unified hybrid checking for first-class types, general refinement types, and `Dynamic`. In *Proc. Scheme Workshop*, 2006.

[15] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *Proc. DSL*, 1999.

[16] David MacQueen. Modules for Standard ML. In *Proc. LFP*, 1984.

[17] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[18] Atsushi Ohori. A polymorphic record calculus and its compilation. *TOPLAS*, 17(6), 1995.

[19] Atsushi Ohori and Peter Buneman. Type inference in a database programming language. In *Proc. LFP*, 1988.

[20] Christine Paulin-Mohring. Inductive definitions in the system Coq - rules and properties. In *Proc. TLCA*, 1993.

[21] Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *Proc. LFP*, 1988.

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

[23] François Pottier. A 3-part type inference engine. In *Proc. ESOP*, 2000.

[24] Didier Rémy. Type inference for records in a natural extension of ML. *Theoretical Aspects of Object-Oriented Programming*, 1994.

[25] Patrick Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In *Proc. PLDI*, 2008.

[26] Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. Type checking with open type functions. In *Proc. ICFP*, 2008.

[27] Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. Complete and decidable type inference for GADTs. In *Proc. ICFP*, 2009.

[28] Zhong Shao, Christopher League, and Stefan Monnier. Implementing typed intermediate languages. In *Proc. ICFP*, 1998.

[29] Tim Sheard. Languages of the future. In *Proc. OOPSLA*, 2004.

[30] Alexandra Silva and Joost Visser. Strong types for relational databases. In *Proc. Haskell Workshop*, 2006.

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

[32] J. B. Wells. Typability and type checking in System F are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98:111–156, 1999.

[33] Hongwei Xi. Dependent ML: an approach to practical programming with dependent types. *J. Functional Programming*, 17(2):215–286, 2007.