Functional Pearl: Implicit Configurations – or, Type Classes Reflect the Values of Types

Oleg Kiselyov
and
Chung-chieh Shan

TR-15-04

Computer Science Group
Harvard University
Cambridge, Massachusetts

Functional Pearl: Implicit Configurations

—or, Type Classes Reflect the Values of Types

Oleg Kiselyov
Fleet Numerical Meteorology
and Oceanography Center
Monterey, CA 93943, USA
oleg@pobox.com

Chung-chieh Shan
Division of Engineering and Applied
Sciences, Harvard University
Cambridge, MA 02138, USA
ccshan@post.harvard.edu

ABSTRACT

The *configurations problem* is to propagate run-time preferences throughout a program, allowing multiple concurrent configuration sets to coexist safely under statically guaranteed separation. This problem is common in all software systems, but particularly acute in Haskell, where currently the most popular solution relies on unsafe operations and compiler pragmas.

We solve the configurations problem in Haskell using only stable and widely implemented language features like the type-class system. In our approach, a term expression can refer to run-time configuration parameters as if they were compile-time constants in global scope. Besides supporting such intuitive term notation and statically guaranteeing separation, our solution also helps improve the program's performance by transparently dispatching to specialized code at run-time. We can propagate any type of configuration data—numbers, strings, *IO* actions, polymorphic functions, closures, and abstract data types. No previous approach to propagating configurations implicitly in any language provides the same static separation guarantees.

The enabling technique behind our solution is to propagate values via types, with the help of polymorphic recursion and higher-rank polymorphism. The technique essentially emulates local type-class instance declarations while preserving coherence. Configuration parameters are propagated throughout the code implicitly as part of type inference rather than explicitly by the programmer. Our technique can be regarded as a portable, coherent, and intuitive alternative to implicit parameters. It motivates adding local instances to Haskell, with a restriction that salvages principal types.

Categories and Subject Descriptors: D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.3.2 [**Language Classifications**]: Haskell; D.3.3 [**Programming Techniques**]: Language Constructs and Features—*abstract data types; polymorphism; recursion*

General Terms: Design, Languages

Keywords: Type classes; implicit parameters; polymorphic recursion; higher-rank polymorphism; existential types

An appendectomized version of this Harvard University technical report is published in the proceedings of the 2004 Haskell Workshop [15].

.

1. INTRODUCTION

Most programs depend on configuration parameters. For example, a pretty-printing function needs to know the width of the page, modular arithmetic depends on the modulus, numerical code heavily depends on tolerances and rounding modes, and most end-user applications depend on user preferences. Sometimes the parameters of the computation are known when the code is written or compiled. Most of the time, however, the parameters are initialized at the beginning of the computation, such as read from a configuration file. Sometimes the parameters stay the same throughout a program execution, but other times they need to be re-initialized. For example, numerical code may need to be re-executed with different rounding modes [12]. Also, a cryptography program may need to perform modular arithmetic with various moduli. Library code especially should support *multiple* sets of parameters that are simultaneously in use, possibly in different threads.

We refer to the problem of setting and propagating preference parameters as the *configurations problem*. We use the term "configurations" in the plural to emphasize that we aim to parameterize code at run time for several concurrent sets of preferences.

A solution to the configurations problem should keep configuration parameters out of the way: code that uses no parameters should not require any change. In particular, the programmer should not be forced to sequence the computation (using a monad, say) when it is not otherwise needed. The parameters should be statically typed and fast to access—ideally, just like regular lexical variables. Configuration data should be allowed to become known only at run time. Moreover, different configurations should be able to coexist. When different configurations do coexist, the user should be statically prevented from inadvertently mixing them up; such subtle errors are easy to make when the first goal above (that configuration parameters be implicit) is achieved. The solution should be available on existing programming language systems.

Given how pervasive the configurations problem is, it is not surprising that the topic provokes repeated discussions in mailing lists [1, 7, 29], conferences [19], and journals [9]. As these discussions conclude, no previous solution is entirely satisfactory.

Historically, the configurations problem is "solved" with mutable global variables or dynamically-scoped variables. Neither solution is satisfactory, because concurrent sets of parameters are hard to support reliably, be the language pure or impure, functional or imperative. Furthermore, in a pure functional language like Haskell, mutable global variables are either unwieldy (all code is written in monadic style) or unsafe (*unsafePerformIO* is used). Another common solution is to store configuration data in a globally accessible *registry*. That approach suffers from run-time overhead and often the loss of static typing. Finally, one type-safe and

pure approach is to place all configuration data into a record and pass it from one function to another. However, it is unappealing to do so explicitly throughout the whole program, not the least because managing concurrent sets of parameters is error-prone.

Implicit parameters [19] are a proposal to extend Haskell with dynamically-scoped variables like LISP's [28]. As a solution to the configurations problem, implicit parameters inherit from dynamically-scoped variables the difficulty of supporting concurrent sets of parameters: they interact unintuitively with other parts of Haskell [26] and easily lead to quite subtle errors [9], whether or not the monomorphism restriction is abolished. As Peyton Jones [26] puts it, "it's really not clear what is the right thing to do."

In this paper, we present a solution to the configurations problem in Haskell that meets all the requirements enumerated above. We rely on type classes, higher-rank polymorphism, and—in advanced cases—the foreign function interface. These are all well-documented and widely-implemented Haskell extensions, for instance in Hugs and in the Glasgow Haskell Compiler. The notation is truly intuitive; for example, the term

$$foo :: (Integral\ a, Modular\ s\ a) \Rightarrow M\ s\ a$$
$$foo = 1000 \times 1000 \times 5 + 2000$$

expresses a modular computation in which *each* addition and multiplication is performed modulo a modulus. The type signature here[1] describes a polymorphic "modulus-bearing number" of type $M\ s\ a$. As we detail in Section 3, the type-class constraints require that the type a be an *Integral* type (such as *Int*), and that the type s carry configuration data for *Modular* arithmetic on a. The modulus is supplied at run time, for example:

$$withIntegralModulus'\ 1280\ foo$$

The same computation can be re-executed with different moduli:

$$[withIntegralModulus'\ m\ foo \mid m \leftarrow [1\,..\,100]]$$

We take advantage of the compiler's existing, automatic type inference to propagate configuration data as type-class constraints. Thus the type annotations that are sometimes needed are infrequent and mostly attached to top-level definitions. Type inference also affords the programmer the flexibility to choose the most convenient way to pass configuration data: take an argument whose type mentions s; return a result whose type mentions s (as *foo* above does), and let unification propagate the type information in the opposite direction of data flow; or even propagate configuration data from one argument of a function to another, by unifying their types. This flexibility reflects the fact that the compile-time flow of configuration types need not follow the run-time flow of configuration values.

Our technique handles not only "conventional" parameters, like numbers and strings, but any Haskell value, including polymorphic functions and abstract data types. We let configuration data include functions tuned to run-time input, such as faster modular-arithmetic functions that exist for composite moduli [16, 25, 30]. For another example, we can treat an array lookup function as a configuration parameter, and selectively disable bounds-checking where we have verified already that array indices are in bounds. In general, we can treat global imports like the Prelude as configuration data, so that different pieces of code can "import their own specialized Prelude".

The basic idea behind our approach is not new. Thurston [31] independently discovered and used our technique for modular arithmetic. Our contribution here is not just to introduce Thurston's technique to a broader audience, but also to extend it to the general configurations problem at any type, beyond integers. We achieve more intuitive notation, as shown above, as well as better performance by specializing code at run-time. Along the way, we survey existing attempts at solving the configurations problem. For multiple configurations, our solution is more portable, coherent, and intuitive than implicit parameters. Finally, our technique effectively declares local type-class instances, which prompts us to sketch an extension to Haskell.

This paper is organized as follows. In Section 2, we discuss the configurations problem and survey previous attempts at solving it. We demonstrate why these attempts are unsatisfactory and illustrate how acute the problem is if otherwise pure functional programmers are willing to resort to operations with no safety guarantee. Section 3 introduces the running example of modular arithmetic. This example calls for the peaceful coexistence and static separation of several concurrent configurations. Section 4 develops our solution in three stages: passing integers; passing serializable data types, including floating-point numbers and strings; and finally, passing any type, including functional and abstract values. In Section 5 we present two real-world examples to demonstrate that our solution scales to multiple parameters and helps the programmer write fast code with intuitive notation. Our solution improves over the OpenSSL cryptography library, where the lack of static separation guarantees seems to have stunted development. In Section 6, we compare our solution to previous work, especially Lewis et al.'s implicit parameters [19]. We argue for adding local type-class instances to Haskell and sketch how. We then conclude in Section 7.

2. THE CONFIGURATIONS PROBLEM

A Haskell program is a collection of definitions, which are rarely closed. For example,

$$result\ approx = last\ \$\ take\ maxIter\ \$\ iterate\ approx\ (pi\ /\ 2)$$

is an open definition: *last*, *take*, *iterate*, *pi*, and *maxIter* are defined elsewhere. The values associated with these symbols are known at compile time. Such a static association is proper for *pi*, which is truly a constant. However, *maxIter* is more of a user preference. A user may reasonably wish to run the program for different values of *maxIter*, without recompiling the code. Unfortunately, if the value of *maxIter* is to be read from a configuration file at the beginning of the computation, or may change from run to run of *result*, it seems that we can no longer refer to *maxIter* as neatly as above.

The *configurations problem* is to make run-time user preferences available throughout a program. As "configurations" in the plural shows, we aim to support concurrent sets of preferences and keep them from being accidentally mixed. The sets of preferences should stay out of the way, yet it should be clear to both the programmer and the compiler which set is used where. (We discuss the latter *coherence* requirement in Section 6.) In this general formulation, the problem is an instance of run-time code parameterization.

The configurations problem is pervasive and acute, as evidenced by recurrent discussions on the Haskell mailing list [1, 7, 29]. It is often pointed out, for example, that numerical code typically depends on a multitude of parameters like *maxIter*: tolerances, initial approximations, and so on. Similarly, most end-user applications support some customization.

The existing approaches to the configurations problem can be summarized as follows [1, 9].

The most obvious solution is to pass all needed parameters explicitly as function arguments. For example:

$$result\ maxIter\ approx =$$
$$\quad last\ \$\ take\ maxIter\ \$\ iterate\ approx\ (pi\ /\ 2)$$

An obvious drawback of this solution is that many computations depend on many parameters, and passing a multitude of positional arguments is error-prone. A more subtle problem is that, if there

[1]This type signature is required. We argue at the end of Section 5.1 that this is an advantage.

are several sets of configuration data (as in Section 3.1), it is easy to make a mistake and pass parameters of the wrong set deep within the code. The mix-up cannot be detected or prevented statically.

The second solution is to group all the parameters in a single Haskell record with many fields, and pass it throughout the code:

```
data ConfigRecord = ConfigRecord
                      {maxIter :: Int, tolerance :: Float ...}
result conf approx =
  last $ take (maxIter conf) $ iterate approx (pi / 2)
```

This approach effectively turns the configuration parameters from positional arguments to keyword arguments. This way, the functions are easier to invoke and have tidier signatures. However, to refer to a configuration parameter, we have to write the more verbose *maxIter conf*. Moreover, we still have to pass the configuration record explicitly from function to function. Therefore, there is still a danger of passing the wrong record in the middle of the code when several configuration records are in scope. The same criticism applies to the analogous approach of passing configuration data in first-class objects or modules in the ML language family.

The third approach, advocated with some reluctance by Hughes [9], is to use implicit parameters [19]. As the name implies, implicit parameters do not need to be passed explicitly among functions that use them. Unfortunately, implicit parameters disturbingly weaken the equational theory of the language: a piece of code may behave differently if we add or remove a type signature, or even just perform a β- or η-expansion or reduction. We compare implicit parameters to our approach in more detail in Section 6.2.

The fourth approach to the configurations problem is to use a reader monad [3]. Its drawback is that any code that uses configuration data (even only indirectly, by calling other functions that do) must be sequenced into monadic style—even if it does not otherwise have to be. Alternatively, we may use mutable reference cells (*IORef*) in conjunction with the *IO* monad. This method obviously emulates mutable global variables, which are often used to store configuration data in impure languages. If our program uses multiple configurations, we may need to mutate the global variables in the middle of the computation, which, as is well-known in imperative languages, is greatly error-prone. Because *IORef* calls for the *IO* monad, using *IORef* for configuration data requires either the tedium of coding in monadic style all the time or the unsoundness of using *unsafePerformIO* [5]. Regrettably, the most popular solution to the configurations problem in Haskell seems to be the latter: issue compiler pragmas to disable inlining and common subexpression elimination, invoke *unsafePerformIO*, and pray [7, 9].

A fifth approach is to generate code at run time, after the necessary configuration data is known [14]. At that time, *maxIter* above can be treated just like *pi*: as a compile-time constant. This approach has the drawback that a compiler and a linker enter the footprint of the run-time system, and can become a performance bottleneck. Moreover, it is harder for program components using different sets of configuration data to communicate.

A final possible solution to the configurations problem is to turn global definitions into local ones:

```
topLevel maxIter tolerance epsilon ... = main where
  main = ···
  ...
  result approx = last $ take maxIter $ iterate approx (pi / 2)
```

Most of the code above is local inside *topLevel*. We pass parameters explicitly to that function only. Within a local definition like *result*, the configuration parameters are in scope, do not have to be passed around, and can be used just by mentioning their names. Furthermore, to use different sets of configuration data, we merely invoke *topLevel* with different arguments. We are statically assured that computations with different configuration data cannot get mixed up. The solution seems ideal—except putting all code within one giant function completely destroys modularity and reuse.

In the following sections, we show how to attain all the benefits of the last approach with modular code arranged in top-level definitions. Our type-class constraints, like *Modular s a* in the introduction, can be thought of as top-level labels for local scopes.

3. MOTIVATING EXAMPLE: MODULAR ARITHMETIC

Modular arithmetic is arithmetic in which numbers that differ by multiples of a given *modulus* are treated as identical: $2 + 3 = 1 \pmod 4$ because $2 + 3$ and 1 differ by a multiple of 4. Many applications, such as modern cryptography, use modular arithmetic with multiple moduli determined at run time. To simplify these computations, we can define functions in Haskell like

```
add :: Integral a ⇒ a → a → a → a
add m a b = mod (a + b) m

mul :: Integral a ⇒ a → a → a → a
mul m a b = mod (a × b) m
```

(where *mod* is a member function of the *Integral* type class in the Prelude) so we can write

```
test1 m a b = add m (mul m a a) (mul m b b)
```

to compute $a \times a + b \times b$ modulo m. The modulus m is the parameter of these computations, which is passed explicitly in the above examples, and which we want to pass implicitly. Like $test_1$ above, many cryptographic routines perform long sequences of arithmetic operations with the same modulus. Since the parameter m is passed explicitly in $test_1$ above, it is easy to make a mistake and write, for example, *add m′ (mul m a a) (mul m b b)*, where m' is some other integral variable in scope. As the first step towards making the modulus parameter implicit, let us make sure that sequences of modular operations like $test_1$ indeed all use the same modulus.

3.1 Phantom Types for Parameter Threads

Our first subgoal, then, is to statically guarantee that a sequence of modular operations is executed with the same modulus. Launchbury and Peyton Jones's [17, 18] *ST* monad for state threads in Haskell uses a type parameter *s* to keep track of the state thread in which each computation takes place. Similarly, we use a type parameter *s* to keep track of the modulus used for each computation. However, because this piece of state is fixed over the course of the computation, we do not force the programmer to sequence the computation by writing in monadic or continuation-passing style.

```
newtype Modulus s a = Modulus a deriving (Eq, Show)
newtype M s a       = M a       deriving (Eq, Show)

add :: Integral a ⇒ Modulus s a → M s a → M s a → M s a
add (Modulus m) (M a) (M b) = M (mod (a + b) m)

mul :: Integral a ⇒ Modulus s a → M s a → M s a → M s a
mul (Modulus m) (M a) (M b) = M (mod (a × b) m)
```

Also, we need the function *unM* to give us the number back from the wrapped data type *M s a*.

```
unM :: M s a → a
unM (M a) = a
```

The type parameter *s* is *phantom*. That is, it has no term representation: the parameter *s* occurs only in type expressions without affecting term expressions. The expression $test_1$ remains the same, but it now has a different type:

$test_1 :: Integral\ a \Rightarrow Modulus\ s\ a \to M\ s\ a \to M\ s\ a \to M\ s\ a$

The argument and result types of *add* and *mul* share the same type variable *s*. During type checking, the compiler automatically propagates this type information to infer the above type for $test_1$. As with the *ST* monad, the type parameter *s* is threaded, but unlike with the *ST* monad, the term-level expression is not sequenced monadically. Hence the compiler knows that the subexpressions *mul m a a* and *mul m b b* of $test_1$ can be computed in any order.

We can now existentially quantify over the type variable *s* to distinguish among different moduli at the type level and make sure that a series of modular operations is performed with the same modulus.

```
data AnyModulus a = ∀s. AnyModulus (Modulus s a)

makeModulus :: a → AnyModulus a
makeModulus a = AnyModulus (Modulus a)
```

This *makeModulus* function is typically used as follows.

```
case makeModulus 4 of
  AnyModulus m →
    let a = M 3; b = M 5 in
      unM $ add m (mul m a a) (mul m b b)
```

In the case alternative **case** *makeModulus* 4 **of** *AnyModulus m* →, the type variable *s* is existentially quantified. The compiler will therefore make sure that *s* does not "leak"—that is, accidentally unify with other quantified type variables or types. Because *s* is threaded through the type of *add* and *mul*, all modular operations in the argument to *unM* are guaranteed to execute with the same *s*, that is, with the same modulus.

There is a redundancy, though: the data constructor *AnyModulus* is applied in *makeModulus*, then peeled off right away in the case alternative. To eliminate this redundant packing and unpacking, we apply a continuation-passing-style transform to turn the existential type in *makeModulus* into a rank-2 polymorphic type:

```
withModulus :: a → (∀s. Modulus s a → w) → w
withModulus m k = k (Modulus m)
```

The *withModulus* function is more usable than *makeModulus*, because it avoids the verbiage of unpacking data constructors.

We can now write

```
test2 = withModulus 4 (λm →
          let a = M 3; b = M 5 in
            unM $ add m (mul m a a) (mul m b b))
```

to get the result 2. If we by mistake try to mix moduli and evaluate

```
withModulus 4 (λm →
  withModulus 7 (λm' →
    let a = M 3; b = M 5 in
      unM $ add m' (mul m a a) (mul m b b)))
```

we get a type error, as desired:

```
Inferred type is less polymorphic than expected
  Quantified type variable s  escapes
  It is mentioned in the environment: m :: Modulus s a
In the second argument of withModulus , namely (λm' → ···)
```

3.2 Type Classes for Modulus Passing

The second step in our development is to avoid explicitly mentioning the modulus *m* in terms. On one hand, in the term $test_1$ above, every occurrence of *add* and *mul* uses the same modulus value *m*. On the other hand, in the type of $test_1$ above, every instantiation of the type-schemes of *add* and *mul* uses the same phantom type *s*. Given that the type checker enforces such similarity between *m* and *s* in appearance and function, one may wonder if we could avoid explicitly mentioning *m* by somehow associating it with *s*.

The idea to *associate a value with a type* is not apocryphal, but quite easy to realize using Haskell's type-class facility. If we constrain our type variable *s* to range over types of a specific type class, then the compiler will associate a class dictionary with *s*. Whenever *s* appears in the type of a term, the corresponding dictionary is available. We just need a slot in that dictionary for our modulus:

```
class Modular s a | s → a where modulus :: s → a

normalize :: (Modular s a, Integral a) ⇒ a → M s a
normalize a :: M s a = M (mod a (modulus (⊥ :: s)))
```

The functional dependency $s \to a$ signifies the fact that the type *s* represents a value of at most one type *a* [11]. As we shall see below, a stronger invariant holds: each value of type *a* is represented by a (different) type *s*.

For conciseness, the code uses lexically-scoped type variables [27] in a non-essential way:[2] in the left-hand side *normalize a* :: *M s a* above, the type *M s a* annotates the result of *normalize* and binds the type variable *s* in $\bot :: s$ to the right of the equal sign. Also, we denote *undefined* with ⊥. One may be aghast at the appearance of ⊥ in terms, but that appearance is only symptomatic of the fact that the polymorphic function *modulus* does not need the value of its argument. The type checker needs the *type* of that argument to choose the correct instance of the class *Modular*. Once the instance is chosen, *modulus* returns the modulus value stored in that class dictionary. Informally speaking, *modulus* retrieves the value associated with the type *s*. If Haskell had a way to pass a type argument, we would have used it.

We can now avoid mentioning *m* in *add* and *mul*. This move makes these functions binary rather than ternary, so we overload the ordinary arithmetic operators + and × for modular arithmetic, simply by defining an instance of the class *Num* for our "modulus-bearing numbers" *M s a*. Modular arithmetic now becomes an instance of general arithmetic, which is mathematically pleasing.

```
instance (Modular s a, Integral a) ⇒ Num (M s a) where
  M a + M b     = normalize (a + b)
  M a − M b     = normalize (a − b)
  M a × M b     = normalize (a × b)
  negate (M a)  = normalize (negate a)
  fromInteger i = normalize (fromInteger i)
  signum        = error "Modular numbers are not signed"
  abs           = error "Modular numbers are not signed"
```

It is thanks to signatures in the *Num* class that this code propagates the modulus so effortlessly. For example, the arguments and result of + share the modulus because *Num* assigns + the type $M\ s\ a \to M\ s\ a \to M\ s\ a$. As we will keep seeing, it is often natural to propagate parameters via types. By contrast, if we think of + as taking two equal modulus *terms* as input, and passing that modulus on to *normalize*, then we might define + much less simply:

```
(M a :: M s1 a) + (M b :: M s2 a) = normalize (a + b) :: M s1 a
  where _ = [⊥ :: s1, ⊥ :: s2]   -- equate the two input moduli
```

Anyway, it seems that we are done. We just need to redefine the function *withModulus* to incorporate our new type class *Modular*.

```
withModulus :: a → (∀s. Modular s a ⇒ s → w) → w
```

But here we encounter a stumbling block: how to actually implement *withModulus*? Given a modulus value *m* of type *a* and a polymorphic continuation *k*, we need to pass to *k* an instance of *Modular s a* defined by *modulus s = m*, for some type *s*. That

[2] This paper is written in literate Haskell and works in the Glasgow Haskell Compiler. (The code is available alongside this technical report.) Not shown here is another version of the code that avoids lexically-scoped type variables and (so) works in Hugs.

is, we need to construct an instance of the class *Modular* such that the function *modulus* in that instance returns the desired value *m*. Constructing such instances is easy when *m* is statically known:

```
m = 5
data Label
instance Modular Label Int where modulus _ = m
```

Hughes [8] shows many practical examples of such instances. But in our case, *m* is not statically known. We want *withModulus* to manufacture a new instance, based on the value of its first argument. One may wonder if this is even possible in Haskell, given that instance declarations cannot appear in the local scope of a definition and cannot be added at run time.

Another way to look at our difficulty is from the point of view of type-class dictionaries. The function *withModulus* must pass to *k* an implicit parameter, namely a dictionary for the type class *Modulus*. This dictionary is not hard to construct—it just contains the term $\lambda s \to m$. However, even though type classes have always been explicated by translating them to dictionary passing [6, 33], Haskell does not expose dictionaries to the programmer. In other words, Haskell does not let us explicitly pass an argument for a double arrow $\Rightarrow$ (as in *Modular s a* $\Rightarrow \cdots$), even though it is internally translated to a single arrow $\to$ (as in *Modulus s a* $\to \cdots$).

In the next section, we explain how to pass dictionary arguments using some widely-implemented extensions to Haskell. We build up this capability in three stages:

1. We describe how to pass an integer as a dictionary argument. This case handles the motivating example above: modular arithmetic over an integral domain.
2. We use Haskell's foreign function interface to pass any type in the *Storable* class as a dictionary argument. This case handles modular arithmetic over a real (fractional) domain.
3. We take advantage of stable pointers in the foreign function interface to pass any type whatsoever—even functions and abstract data types—as a dictionary argument. This technique generalizes our approach to all configuration data.

4. BUILDING DICTIONARIES BY REFLECTING TYPES

Dictionaries at run time reflect context reductions at compile time, in a shining instance of the Curry-Howard correspondence. To pass a dictionary argument explicitly, then, we need to reify it as a type that can in turn reflect back as the intended value.

4.1 Reifying Integers

We start by reifying integers. We build a family of types such that each member of the family corresponds to a unique integer. To encode integers in binary notation, we introduce the type constant *Zero* and three unary type constructors.

```
data Zero; data Twice s; data Succ s; data Pred s
```

For example, the number 5, or 101 in binary, corresponds to the type *Succ* (*Twice* (*Twice* (*Succ Zero*))). This representation is inspired by the way Okasaki [23] encodes the sizes of square matrices. Our types, unlike Okasaki's, have no data constructors, so they are only inhabited by the bottom value $\perp$. We are not interested in values of these types, only in the types themselves.[3]

We need to convert a type in our family to the corresponding integer—and back. The first process—*reflecting* a type into the corresponding integer—is given by the class *ReflectNum*:

```
class ReflectNum s where reflectNum :: Num a ⇒ s → a
instance                    ReflectNum Zero where
  reflectNum _ = 0
instance ReflectNum s ⇒ ReflectNum (Twice s) where
  reflectNum _ = reflectNum (⊥ :: s) × 2
instance ReflectNum s ⇒ ReflectNum (Succ s) where
  reflectNum _ = reflectNum (⊥ :: s) + 1
instance ReflectNum s ⇒ ReflectNum (Pred s) where
  reflectNum _ = reflectNum (⊥ :: s) − 1
```

The instances of the class deconstruct the type and perform corresponding operations (doubling, incrementing, and so on). Again, we should not be afraid of $\perp$ in terms. As the underscores show, the function *reflectNum* never examines the value of its argument. We only need the type of the argument to choose the instance. Informally speaking, *reflectNum* "maps types to values".

The inverse of *reflectNum* is *reifyIntegral*, which turns a signed integer into a type that represents the given number in binary notation. In other words, the type says how to make the given number by applying increment, decrement and double operations to zero.

```
reifyIntegral :: Integral a ⇒
                 a → (∀s. ReflectNum s ⇒ s → w) → w
reifyIntegral i k = case quotRem i 2 of
  (0,  0) → k (⊥ :: Zero)
  (j,  0) → reifyIntegral j (λ(_ :: s) → k (⊥ :: Twice s))
  (j,  1) → reifyIntegral j (λ(_ :: s) → k (⊥ :: Succ (Twice s)))
  (j, −1) → reifyIntegral j (λ(_ :: s) → k (⊥ :: Pred (Twice s)))
```

The second argument to the function *reifyIntegral* is a continuation *k* from the generated type *s* to the answer type *w*. The generated type *s* is in the class *ReflectNum*, so the *reflectNum* function can convert it back to the value it came from. To be more precise, *reifyIntegral* passes to the continuation *k* a value whose type belongs to the class *ReflectNum*. As we are interested only in the type of that value, the value itself is $\perp$. The continuation passed to *reifyIntegral* should be able to process a value of any type belonging to the class *ReflectNum*. Therefore, the continuation is polymorphic and the function *reifyIntegral* has a rank-2 type.

At the end of Section 3.2, we stumbled over creating an instance of the class *Modular* to incorporate a modulus unknown until run time. Haskell does not let us create instances at run time or locally, but we can now get around that. We introduce a polymorphic instance of the class *Modular*, parameterized over types in the class *ReflectNum*. Each instance of *ReflectNum* corresponds to a unique integer. In essence, we introduce instances of the *Modular* class for *every* integer. At run time, we do not create a new instance for the *Modular* class—rather, we use polymorphic recursion to choose from the infinite family of instances already introduced.

```
data ModulusNum s a
instance (ReflectNum s, Num a) ⇒
         Modular (ModulusNum s a) a where
  modulus _ = reflectNum (⊥ :: s)
```

We can now implement the function *withModulus*, which was the stumbling block above. We call this function *withIntegralModulus*, as it will be generalized below.

[3]Like Okasaki, we include *Twice* to perform reification and reflection in time linear (rather than exponential) in the number of bits. We also include *Pred* to encode negative numbers. These two type constructors make our type family larger than necessary: an integer can be encoded in an infinite number of different ways. For example, the number 5 also corresponds to the type *Succ* (*Succ* (*Succ* (*Succ* (*Succ Zero*)))). We can easily use a different set of type constructors to enforce a unique representation of integers (we elide the code for brevity), but there is no need for the representation to be unique in this paper, and the type constructors above are easier to understand.

```
withIntegralModulus :: Integral a ⇒
                       a → (∀s. Modular s a ⇒ s → w) → w
withIntegralModulus i k =
  reifyIntegral i (λ(_ :: s) → k (⊥ :: ModulusNum s a))
```

We can test the function by evaluating *withIntegralModulus* (−42) *modulus*. The result is −42: the round-trip through types even leaves negative numbers unscathed. Our ability to reify any integer, not just positive ones, is useful below beyond modular arithmetic.

One caveat: The correctness of the round-trip is not checked by the type system, unlike what one might expect from type systems that truly offer singleton or dependent types. For example, if we accidentally omitted *Succ* in *reifyIntegral* above, the compiler would not detect the error. The reflection and reification functions therefore belong to a (relatively compact) trusted kernel of our solution, which must be verified manually and can be put into a library.

We can now write our running example as

```
test'3 :: (Modular s a, Integral a) ⇒ s → M s a
test'3 _ = let a = M 3; b = M 5 in a × a + b × b
test3 = withIntegralModulus 4 (unM ∘ test'3)
```

The sequence of modular operations appears in the mathematically pleasing notation $a \times a + b \times b$. The modulus is implicit, just as desired. Because we defined the method *fromInteger* in the class *Num*, this example can be written more succinctly:

```
test'3 :: (Modular s a, Integral a) ⇒ s → M s a
test'3 _ = 3 × 3 + 5 × 5
```

Section 5.1 further simplifies this notation.

A word on efficiency: With an ordinary compiler, every time a modulus needs to be looked up (which is quite often), *reflectNum* performs recursion of time linear in the number of bits in the modulus. That is pretty inefficient. Fortunately, we can adjust the code so that Haskell's lazy evaluation memoizes the result of *reflectNum*, which then only needs to run once per reification, not once per reflection. For clarity, we do not make the adjustment here. However, the code in Section 4.3 is so adjusted to memoize appropriately, out of necessity; the crucial subexpression there is *const a* in *reflect*.

Thurston [31] independently discovered the above techniques for typing modular arithmetic in Haskell. The following extends this basic idea to reifying values of serializable type, then any type.

4.2 Reifying Lists

Our immediate goal of implementing modular arithmetic without explicitly passing moduli around is accomplished. Although the type-class machinery we used to achieve this goal may seem heavy at first, it statically and implicitly distinguishes multiple concurrent moduli, which cannot be said of any previous solution to the configurations problem in any pure or impure language. We also avoid using *unsafePerformIO*. Section 5 below shows more real-world examples to further illustrate the advantages of our approach. Those examples are independent of the rest of Section 4 here.

We now turn to a larger goal—passing configuration data other than integers. For example, many parameters for numerical code are floating point numbers, such as tolerances. Also, user preferences are often strings.

A string can be regarded as a list of integers (character codes). As the next step, we reify lists of integers into types. In principle, this step is redundant: we already know how to reify integers, and a list of integers can always be represented as one (huge) integer. Supporting lists directly, however, is faster and more convenient. We extend our family of types with a type constant *Nil* and a binary type constructor *Cons*, to build singly-linked lists at the type level.

```
data Nil; data Cons s ss
class ReflectNums ss where reflectNums :: Num a ⇒ ss → [a]
instance ReflectNums Nil where
  reflectNums _ = []
instance (ReflectNum s, ReflectNums ss) ⇒
         ReflectNums (Cons s ss) where
  reflectNums _ = reflectNum (⊥ :: s) : reflectNums (⊥ :: ss)
reifyIntegrals :: Integral a ⇒
                  [a] → (∀ss. ReflectNums ss ⇒ ss → w) → w
reifyIntegrals [] k = k (⊥ :: Nil)
reifyIntegrals (i : ii) k = reifyIntegral i (λ(_ :: s) →
                              reifyIntegrals ii (λ(_ :: ss) →
                                k (⊥ :: Cons s ss)))
```

We can check that lists of numbers round-trip unscathed: the expression *reifyIntegrals* [−10 .. 10] *reflectNums* returns the list of integers from −10 to 10.

Being able to reify a list of numbers to a type is more useful than it may appear: we gain the ability to reflect any value whose type belongs to the *Storable* type class in Haskell's foreign function interface, or FFI [4]. A *Storable* value is one that can be serialized as a sequence of bytes, then reconstructed after being transported—over the network; across a foreign function call; or, in our case, to the left of ⇒. (For reference, Appendix B summarizes what we use of FFI.)

```
type Byte = CChar
data Store s a
class ReflectStorable s where
  reflectStorable :: Storable a ⇒ s a → a
instance ReflectNums s ⇒ ReflectStorable (Store s) where
  reflectStorable _ = unsafePerformIO
                      $ alloca
                      $ λp → do pokeArray (castPtr p) bytes
                                peek p
    where bytes = reflectNums (⊥ :: s) :: [Byte]
reifyStorable :: Storable a ⇒
                 a → (∀s. ReflectStorable s ⇒ s a → w) → w
reifyStorable a k =
  reifyIntegrals (bytes :: [Byte]) (λ(_ :: s) → k (⊥ :: Store s a))
    where bytes = unsafePerformIO
                  $ with a (peekArray (sizeOf a) ∘ castPtr)
```

The *reifyStorable* function defined here first serializes the value *a* into an array of (*sizeOf a*) bytes, temporarily allocated by FFI's *with*. It then uses *reifyIntegrals* above to reify the bytes into a type. In the opposite direction, the *reflectStorable* function first uses *reflectNums* to reflect the type into another array of bytes, temporarily allocated by FFI's *alloca* to ensure proper alignment. It then reconstructs the original value using FFI's *peek*.

We must comment on the use of *unsafePerformIO* above, which emphatically neither compromises static typing nor weakens static guarantees. The type signatures of *reifyStorable* and *reflectStorable* make it clear that the values before reification and after reflection have the same type; we do not replace type errors with run-time exceptions. The code above invokes *unsafePerformIO* only because it relies on FFI, in which even mere serialization operates in the *IO* monad. If functions like *pokeArray*, *peek*, and *peekArray* operated in the *ST* monad instead, then we would be able to (happily) replace *unsafePerformIO* with *runST*. We do not see any reason why serialization should require the *IO* monad.

We can now round-trip floating-point numbers through the type system into a dictionary: the expression

```
reifyStorable (2.5 :: Double) reflectStorable
```

returns 2.5. This capability is useful for modular arithmetic over

a real (fractional) domain—that is, over a circle with a specified circumference as a metric space. Although multiplication no longer makes sense in such a domain, addition and subtraction still do.

Admittedly, a floating-point number can be converted into a pair of integers using the *decodeFloat* function, which provides a more portable way to reify a value whose type belongs to the *RealFloat* type class in the Prelude. Furthermore, any type that belongs to both the *Show* class and the *Read* class can be transported without involving FFI, as long as *read* ∘ *show* is equivalent to the identity as usual so that we can serialize the data thus. However, we are about to reify *StablePtr* values from FFI, and the *StablePtr* type belongs to none of these classes, only *Storable*.

4.3 Reifying Arbitrary Values

We now turn to our ultimate goal: to round-trip any Haskell value through the type system, so as to be able to pass any dictionary as an explicit argument, even ones involving polymorphic functions or abstract data types. To achieve this, we use FFI to convert the value to a *StablePtr* ("stable pointer"), which we then reify as a type. From the perspective of an ordinary Haskell value, Haskell's type system and type-class instances are foreign indeed![4]

```
class Reflect s a | s → a where reflect :: s → a

data Stable (s :: ⋆ → ⋆) a

instance ReflectStorable s ⇒ Reflect (Stable s a) a where
  reflect = unsafePerformIO
          $ do a ← deRefStablePtr p
               return (const a)
    where p = reflectStorable (⊥ :: s p)

reify :: a → (∀s. Reflect s a ⇒ s → w) → w
reify (a :: a) k = unsafePerformIO
                 $ do p ← newStablePtr a
                      reifyStorable p (λ(_ :: s p) →
                                         k′ (⊥ :: Stable s a))
    where k′ s = return (k s)
```

We can now define the completely polymorphic *withModulus* function that we set out to implement.

```
data ModulusAny s

instance Reflect s a ⇒ Modular (ModulusAny s) a where
  modulus _ = reflect (⊥ :: s)

withModulus a k = reify a (λ(_ :: s) → k (⊥ :: ModulusAny s))
```

This code passes configuration data "by reference", whereas the code in Sections 4.1–2 passes them "by value". Configuration data of arbitrary type may not be serialized, so they must be passed by reference. We use a stable pointer as that reference, so that the value is not garbage-collected away while the reference is in transit.

The code above has a memory leak: it allocates stable pointers using *newStablePtr* but never deallocates them using *freeStablePtr*. Thus every set of configuration data leaks a stable pointer when reified. Configuration data in programs are typically few and long-lived, so this memory leak is usually not a problem. However, if the program dynamically generates and discards many pieces of configuration data over its lifetime, then leaking one stable pointer per reification is a significant resource drain.

If these memory leaks are significant, then we need to carefully ensure that the *StablePtr* allocated in each reification operation is freed exactly once. Unfortunately, this requires us to worry about how lazy evaluation and *seq* interact with impure uses of *unsafePerformIO*: we need to make sure that each stable pointer is freed exactly once. Below is the modified code.

[4]The type variable *p* in this section is bound but never used.

```
instance ReflectStorable s ⇒ Reflect (Stable s a) a where
  reflect = unsafePerformIO
          $ do a ← deRefStablePtr p
                freeStablePtr p
                return (const a)
    where p = reflectStorable (⊥ :: s p)

reify :: a → (∀s. Reflect s a ⇒ s → w) → w
reify (a :: a) k = unsafePerformIO
                 $ do p ← newStablePtr a
                      reifyStorable p (λ(_ :: s p) →
                                         k′ (⊥ :: Stable s a))
    where k′ (s :: s) = (reflect :: s → a) `seq` return (k s)
```

We emphasize that this impure use of *unsafePerformIO* is only necessary if the program reifies many non-serializable parameters outside the *IO* monad over its lifetime. Such programs are rare in practice; for example, a numerical analysis program or a cryptography server may reify many parameters in a single run, but these parameters are *Storable* values, like numbers.

5. MORE EXAMPLES

In this section we discuss two more examples of our approach to the configurations problem. The first example illustrates how the flexibility of our solution and its integration with type inference helps the programmer write code in the most intuitive notation. The second example demonstrates how our solution helps write fast code by guaranteeing that specialized versions of algorithms are used when appropriate. The second example also shows that our approach is wieldy to apply to more realistic problems. In particular, it shows that it is straightforward to generalize our technique from one parameter (*modulus*) to many. Appendix A contains another real-world example, where we contrast our approach more concretely with implicit parameters.

5.1 Flexible Propagation for Intuitive Notation

Let us revisit the modular arithmetic example from Section 4.1, and trace how the modulus is propagated.

```
withIntegralModulus :: Integral a ⇒
                       a → (∀s. Modular s a ⇒ s → w) → w
withIntegralModulus i k =
  reifyIntegral i (λ(_ :: t) → k (⊥ :: ModulusNum t a))

test′3 :: (Modular s a, Integral a) ⇒ s → M s a
test′3 _ = 3 × 3 + 5 × 5

test3 = withIntegralModulus 4 (unM ∘ test′3)
```

The modulus 4 starts out as the argument to *withIntegralModulus*. Given this modulus, the function *reifyIntegral* finds the corresponding type of the *ReflectNum* family. That type, denoted by the type variable *t*, is then used to build the type *ModulusNum t a*. The latter type is an instance of the *Modular s a* class, with the type variable *s* now instantiated to *ModulusNum t a*. When the function $test'_3$ is applied to the (bottom) value of the latter type, *s* propagates from the argument of $test'_3$ throughout the body of $test'_3$. Because *s* is instantiated to *ModulusNum t a*, and *t* uniquely corresponds to a particular modulus, the modulus is available throughout $test'_3$.

In this example, then, a parameter is propagated to $test'_3$ when the argument type *s* of $test'_3$ is unified with *ModulusNum t a*. Because type unification works the same way for a function's argument type and return type, the type checker can propagate type information not only via arguments of the function but also via its result. In the case of modular arithmetic, propagating configuration information via the return type rather than argument type of $test'_3$ leads to a

particularly concise and intuitive notation. As the first step, we move the function *unM* inside *withIntegralModulus*:

```
withIntegralModulus :: Integral a ⇒
                       a → (∀s. Modular s a ⇒ s → M s w) → w
withIntegralModulus i k =
  reifyIntegral i (λ(_ :: t) → unM $ k (⊥ :: ModulusNum t a))
```

The type variable *s* now appears in the result type of *k*. The modulus is now propagated to *k*—in other words, the type variable *s* is now instantiated in the type of *k*—in two ways: through its argument type as well as its return type. If only for brevity, we can now eliminate the first way by getting rid of the argument to *k*:

```
withIntegralModulus' :: Integral a ⇒
                        a → (∀s. Modular s a ⇒ M s w) → w
withIntegralModulus' (i :: a) k :: w =
  reifyIntegral i (λ(_ :: t) →
                       unM (k :: M (ModulusNum t a) w))
test4' :: (Modular s a, Integral a) ⇒ M s a
test4' = 3 × 3 + 5 × 5
test4  = withIntegralModulus' 4 test4'
```

In the terminology of logic programming, we have switched from one mode of invoking *k*, where the argument type is bound and the result type is free, to another mode, where the result type is bound. The resulting definition $test4' = 3 \times 3 + 5 \times 5$ cannot be more intuitive. The body of *test4′* performs a sequence of arithmetic computations using the same modulus, which however appears nowhere in the term, only in the type. The modulus parameter is implicit; it explicitly appears only in the function *normalize* used in the implementation of modular operations. The configuration data are indeed pervasive and do stay out of the way. Furthermore, *test4′* is a top-level binding, which can be exported from its home module and imported into other modules. We have achieved implicit configuration while preserving modularity and reuse.

The definition $test4' = 3 \times 3 + 5 \times 5$ looks so intuitive that one may even doubt whether every arithmetic operation in the term is indeed performed modulo the invisible modulus. One might even think that we first compute $3 \times 3 + 5 \times 5$ and later on divide 34 by the modulus. However, what *term4′* actually computes is

```
mod (mod (mod 3 m × mod 3 m) m
         + mod (mod 5 m × mod 5 m) m) m
```

Each operation is performed modulo the modulus *m* corresponding to the type *s* in the signature of *test4′*. That top-level type signature is the only indication that implicit configuration is at work, as desired. To check that each operation in *term4′* is performed modulo *m*, we can trace the code using a debugger. We can also try to omit the type signature of *test4′*. If we do that, we get a type error:

```
Inferred type is less polymorphic than expected
  Quantified type variable s  escapes
  It is mentioned in the environment: test4' :: M s w
In the second argument of withIntegralModulus' , namely test4'
In the definition of test4 : test4 = withIntegralModulus' 4 test4'
```

The fact that we get an error contrasts with the implicit parameter approach [19]. In the latter, omitting the signature may silently change the behavior of the code. Our approach thus is both free from unpleasant surprises and notationally intuitive.

5.2 Run-Time Dispatch for Fast Performance

We now turn from optimizing the visual appearance of the code to optimizing its run-time performance. A general optimization strategy is to identify "fast paths"—that is, particular circumstances that permit specialized, faster algorithms. We can then structure our code to first check for auspicious circumstances. If they are present, we branch to the specialized code; otherwise, generic code is run.

Modular arithmetic is a good example of such a specialization. Modern cryptography uses lots of modular arithmetic, so it is important to exploit fast execution paths. OpenSSL [24], a well-known open-source cryptography library, uses specialized code on many levels. At initialization time, it detects any cryptographic acceleration hardware and sets up method handlers accordingly. Cryptographic operations include sequences of modular addition and multiplication over the same modulus. Moduli of certain forms permit faster computations. OpenSSL maintains a context *CTX* with pointers to addition and multiplication functions for the modulus in effect. When initializing *CTX*, OpenSSL checks the modulus to see if a faster version of modular operations can be used.

To use these optimized functions, one can pass them as explicit function arguments, as OpenSSL does. This impairs the appearance and maintainability of the code. If several moduli are in use, each with its own *CTX* structure, it is easy to pass the wrong one by mistake. Our technique can improve this situation. Because we can pass functions implicitly, we can pass the addition and multiplication functions themselves as configuration data.

In simple cases, specialized functions use the same data representation but a more efficient implementation. For example, the Haskell *mod* function can be specialized to use bitwise operators when the modulus is a power of 2. More often, however, specialized functions operate on custom representations of input data. For example, Montgomery's technique for modular multiplication [22] is much faster than the standard algorithm when the modulus is odd, but it requires input numbers to be represented by their so-called *N*-residues. Furthermore, the algorithm needs several parameters that are pre-computed from the modulus. Therefore, at the beginning of a sequence of operations, we have to convert the inputs into their *N*-residues, and compute and cache required parameters. At the end, we have to convert the result from its *N*-residue back to the regular representation. For a long sequence of operations, switching representations induces a net performance gain.

OpenSSL uses Montgomery multiplication for modular exponentiation when the modulus is odd. Modular exponentiation is a long sequence of modular multiplications. As exponentiation begins, OpenSSL converts the radix into its *N*-residue, computes the parameters, and caches them. At the end, the library converts the result back from its *N*-residue and disposes of the cache. Diffie-Hellman key exchanges, for example, invoke modular exponentiation several times. To avoid converting representations and computing parameters redundantly, OpenSSL can save the Montgomery context as the part of the overall *CTX*. This option raises correctness concerns that are more severe than the mere inconvenience of explicitly passing *CTX* around: While the Montgomery context is in effect, what appears to be modular numbers to the client are actually their *N*-residues. The client must take care not to pass them to functions unaware of the Montgomery context. The programmer must keep track of which context—generic or Montgomery—is in effect and thus which representation is in use. In sum, although the Montgomery specialization is faster, its implementation in OpenSSL invites user errors that jeopardize data integrity.

In this section, we show how to use a specialized representation for modular numbers that is even more different from the standard representation than Montgomery multiplication calls for. We represent a modular number as not one *N*-residue but a pair of residues. The type system statically guarantees the safety of the specialization; different representations are statically separated. Yet actual code specifying what to compute is not duplicated.

In our code so far, only the modulus itself is propagated through

the type environment. Our instance of the *Num* class for the modulus-bearing numbers *M s a* implements general, unspecialized algorithms for modular addition and multiplication. If the modulus m is even, say of the form $2^p q$ where p is positive and q is odd, we can perform modular operations more efficiently: taking advantage of the Chinese Remainder Theorem, we can represent each modular number not as one residue modulo $2^p q$ but as two residues, modulo 2^p and q. When we need to perform a long sequence of modular operations, such as multiplications to compute $a^n \bmod m$ for large n, we first determine the residues of $a \bmod 2^p$ and q. We perform the multiplications on each of the two residues, then recombine them into one result. We use the fact that the two factor moduli are smaller, and operations modulo 2^p are very fast. This technique is known as *residue number system arithmetic* [16, 25, 30].

Four numbers need to be precomputed that depend on the modulus: p, q, u, and v, such that the modulus is $2^p q$ and

$$u \equiv 1 \pmod{2^p},\ u \equiv 0 \pmod{q},\ v \equiv 0 \pmod{2^p},\ v \equiv 1 \pmod{q}.$$

In order to propagate these four numbers as configuration data for even-modulus-bearing numbers, we define a new data type *Even*. The type arguments to *Even* specifies the configuration data to propagate; the data constructor *E* of *Even* specifies the run-time representation of even-modulus-bearing numbers, as a pair of residues.

```
data Even p q u v a = E a a deriving (Eq, Show)
```

We then define a *Num* instance for *Even*.

```
normalizeEven :: (ReflectNum p, ReflectNum q, Integral a,
                  Bits a) ⇒ a → a → Even p q u v a
normalizeEven a b :: Even p q u v a =
  E (a .&. (shiftL 1 (reflectNum (⊥ :: p)) − 1))     -- a mod 2^p
    (mod b (reflectNum (⊥ :: q)))                    -- b mod q

instance (ReflectNum p, ReflectNum q,
          ReflectNum u, ReflectNum v,
          Integral a, Bits a) ⇒ Num (Even p q u v a) where
  E a1 b1 + E a2 b2 = normalizeEven (a1 + a2) (b1 + b2)
                    ⋮
```

Following this pattern, we can introduce several varieties of modulus-bearing numbers, optimized for particular kinds of moduli.

Each time the *withIntegralModulus′* function is called with a modulus, it should select the best instance of the *Num* class for that modulus. The implementation of modular operations in that instance will then be used throughout the entire sequence of modular operations. This pattern of run-time dispatch and compile-time propagation is illustrated below with two *Num* instances: the general instance for *M*, and the specialized instance for *Even*.

```
withIntegralModulus″ :: (Integral a, Bits a) ⇒
                        a → (∀s. Num (s a) ⇒ s a) → a
withIntegralModulus″ (i :: a) k = case factor 0 i of
  (0, i) → withIntegralModulus′ i k   -- odd case
  (p, q) → let (u, v) = ··· in        -- even case: i = 2^p q
           reifyIntegral p (λ(_ :: p) →
           reifyIntegral q (λ(_ :: q) →
           reifyIntegral u (λ(_ :: u) →
           reifyIntegral v (λ(_ :: v) →
           unEven (k :: Even p q u v a)))))

factor :: (Num p, Integral q) ⇒ p → q → (p, q)
factor p i = case quotRem i 2 of
  (0, 0) → (0, 0)              -- just zero
  (j, 0) → factor (p + 1) j    -- accumulate powers of two
  _      → (p, i)              -- not even

unEven :: (ReflectNum p, ReflectNum q, ReflectNum u,
  ReflectNum v, Integral a, Bits a) ⇒ Even p q u v a → a
unEven (E a b :: Even p q u v a) =
  mod (a × (reflectNum (⊥ :: u)) + b × (reflectNum (⊥ :: v)))
      (shiftL (reflectNum (⊥ :: q)) (reflectNum (⊥ :: p)))
```

The function *withIntegralModulus″* checks at run time whether the received modulus is even. This check is done only once per sequence of modular operations denoted by the continuation *k*. If the modulus is even, the function chooses the instance *Even* and computes the necessary parameters for that instance: p, q, u, and v. The continuation *k* then uses the faster versions of modular operations, without any further checks or conversions between representations.

In Section 4, we introduced our technique with a type class with a single member (*modulus*), parameterized by a single integer. The code above propagates multiple pieces of configuration information (namely the members of the *Num* class: +, −, ×, etc.), parameterized by four integers. The generalization is straightforward: *withIntegralModulus″* calls *reifyIntegral* four times, and the instance *Num* (*Even p q u v a*) defines multiple members at once.

OpenSSL's source code for modular exponentiation (`bn_exp.c`) mentions, in comments, this specialized multiplication algorithm for even moduli. However, it does not implement the specialization, perhaps because it is too much trouble for the programmer to explicitly deal with the significantly different representation of numbers (as residue pairs) and ensure the correctness of the C code.

The example below tests both the general and specialized cases:

```
test5 :: Num (s a) ⇒ s a
test5 = 1000 × 1000 + 513 × 513

test5′  = withIntegralModulus″ 1279 test5 :: Integer
test5″ = withIntegralModulus″ 1280 test5 :: Integer
```

The body of *test5* contains two multiplications and one addition. Whereas *test5′* uses the generic implementation of these operations, *test5″* invokes the specialized versions as the modulus 1280 is even. We can see that by tracing both versions of functions.

This example shows that types can propagate not just integers but also functions parameterized by them—in other words, closures. Crucially, exactly the same sequence of operations *test5* uses either generic or specialized modular operations, depending on the modulus value at run time. The specialized modular operations use a different representation of numbers, as residue pairs. The type system encapsulates the specialized representation of numbers. We thus attain a static correctness guarantee that OpenSSL cannot provide. This comparison underscores the fact that our approach to the configurations problem benefits pure and impure languages alike.

6. DISCUSSION AND RELATED WORK

Our solution to the configurations problem can be understood from several different perspectives.

1. It emulates local type-class instance declarations while preserving principal types.
2. It ensures the coherence of implicit parameters by associating them with phantom types.
3. It fakes dependent types: types can depend on not values but types that faithfully represent each value.

We now detail these perspectives in turn. Overall, we recommend that local type-class instances be added to Haskell as a built-in feature to replace implicit parameters and fake dependent types.

6.1 Local Type-Class Instances

The purpose of the type-system hackery in Section 4, first stated in Section 3.2, is not to market headache medicine but to explicitly pass a dictionary to a function with a qualified type. For example, we want to apply a function of type $\forall s.\ \mathit{Modular}\ s\ a \Rightarrow s \to w$ to

a dictionary witnessing the type-class constraint *Modular s a*. In general, we want to manufacture and use type-class instances at run time. In other words, we want to declare type-class instances not just at the top level but also *locally*, under the scope of variables.

Sections 3 and 5 of this paper show that local type-class instances are very useful. Although we can emulate local instances using the hackery in Section 4, it would be more convenient if a future version of Haskell could support them directly as a language feature. At first try, the syntax for this feature might look like the following.

```
data Label

withModulus :: a → (∀s. Modular s a ⇒ s → w) → w
withModulus (m :: a) k =
  let instance Modular Label a where modulus _ = m
  in k (⊥ :: Label)
```

The new syntax added is the **instance** declaration under **let**, against which the continuation *k* resolves its overloading.

A problem with this first attempt, pointed out early on by Wadler and Blott [33, Section A.7], is that principle types are lost in the presence of unrestricted local instances. For example, the term

```
data Label1; data Label2

let instance Modular Label1 Int where modulus _ = 4
    instance Modular Label2 Int where modulus _ = 4
in modulus
```

has no principle type, only the types $Label_1 \to Int$ and $Label_2 \to Int$, neither of which subsumes the other. (It may seem that this term should have the (principal) type *Modular s Int* ⇒ $s \to Int$, but that would result in unresolved overloading and defeat the purpose of the local instances.) This problem is one reason why Haskell today allows only global instances, as Wadler and Blott suggested.

Wadler and Blott close their paper by asking the open question "whether there is some less drastic restriction that still ensures the existence of principal types." We conjecture that one such restriction is to require that the type-class parameters of each local instance mention some *opaque* type at the very same **let**-binding scope. We define an opaque type at a given scope to be a type variable whose existential quantification is eliminated ("opened"), or universal quantification is introduced ("generalized"), at that scope. For example, *withModulus* would be implemented as follows.

```
data Any = ∀s. Any s

withModulus (m :: a) k =
  let Any (_ :: s) = Any ()
      instance Modular s a where modulus _ = m
  in k (⊥ :: s)
```

The above code satisfies our proposed restriction because the local instance *Modular s a* mentions the type variable *s*, which results from existential elimination (**let** *Any* (_ :: *s*) = ⋯) at the very same scope. This restriction is directly suggested by our technique in Section 4. There, we build a different type for each modulus value to be represented, so a function that can take any modulus value as input is one that can take any modulus-representing opaque type as input. Just as Launchbury and Peyton Jones [17, 18] use an opaque type to represent an unknown state thread, we use an opaque type to represent an unknown modulus.

The term below is analogous to the problematic term above without a principal type, but adheres to our proposed restriction.

```
let Any (_ :: s1) = Any ()
    instance Modular s1 Int where modulus _ = 4
    Any (_ :: s2) = Any ()
    instance Modular s2 Int where modulus _ = 4
in modulus
```

This term satisfies the principal type property—vacuously, because it simply does not type! Although *modulus* has both the type $s_1 \to Int$ and the type $s_2 \to Int$ within the scope of the **let**, neither type survives outside, because the type variables s_1 and s_2 cannot escape.

Our proposed restriction not only rescues the principal type property in Wadler and Blott's example above, but also preserves the *coherence* of type classes. Coherence means that two typing derivations for the same term at the same type in the same environment must be observationally equivalent. Coherence is important in our solution to the configurations problem, because we need each type to represent at most one value in order to statically separate multiple configuration sets—be they multiple moduli as in the examples above, or multiple threads of the Java virtual machine as in Appendix A. Standard Haskell ensures coherence by prohibiting overlapping instances. By requiring that every local instance mention an opaque type, we ensure that two local instances from different scopes cannot overlap—at least, not if their parameters are fully instantiated. We leave local instances with uninstantiated type variables in the head for future research.

To sum up, when examined from the perspective of local type-class instances, our type-system hackery suggests a restriction on local instances that (we conjecture) salvages principal types. In other words, we suggest adding local instances to Haskell as syntactic sugar for our reification technique. As an aside, local instances as a built-in language feature would allow constraints in their contexts. To support such constraints under our current technique would call for Trifonov's simulation [32].

6.2 Implicit Parameters

Our approach to the configurations problem is in the same implicit spirit as Lewis et al.'s implicit parameters [19]. Emulating LISP's dynamically-scoped variables (as explained by Queinnec [28] among others), Lewis et al. extend Haskell's type-class constraints like *Modular s a* with implicit-parameter constraints like ?*modulus* :: *a*. Under this proposal, modular arithmetic would be implemented by code such as

```
add :: (Integral a, ?modulus :: a) ⇒ a → a → a
add a b = mod (a + b) ?modulus

mul :: (Integral a, ?modulus :: a) ⇒ a → a → a
mul a b = mod (a × b) ?modulus
```

The type checker can infer the signatures above. The implicit parameter ?*modulus* can be assigned a value within a dynamic scope using a new **with** construct; for example:[5]

```
add (mul 3 3) (mul 5 5) with ?modulus = 4    -- evaluates to 2
```

Lewis et al., like us, intend to solve the configurations problem, so the programming examples they give to justify their work apply equally to ours. Both approaches rely on dictionaries, which are arguments implicitly available to any polymorphic function with a quantified type. Dictionary arguments are passed like any other argument at run-time, but they are hidden from the term representation and managed by the compiler, so the program is less cluttered.

Whereas we take advantage of the type-class system, implicit parameters augment it. Lewis et al. frame their work as "the first half of a larger research programme to de-construct the complex type class system of Haskell into simpler, orthogonal language features". Unfortunately, because implicit parameters are a form of dynamic scoping, they interact with the type system in several undesirable ways [26]:

[5] In the Glasgow Haskell Compiler, implicit parameters are bound not using a separate **with** construct but using a special **let** or **where** binding form, as in **let** ?*modulus* = 4 **in** *add* (*mul* 3 3) (*mul* 5 5). We stick with Lewis et al.'s notation here.

1. It is not sound to inline code (in other words, to β-reduce) in the presence of implicit parameters.
2. A term's behavior can change if its signature is added, removed, or changed.
3. Generalizing over implicit parameters is desirable, but may contradict the monomorphism restriction.
4. Implicit parameter constraints cannot appear in the context of a class or instance declaration.

One may claim that the many troubles of implicit parameters come from the monomorphism restriction, which ought to be abandoned. Without defending the monomorphism restriction in any way, we emphasize that trouble (such as unexpected loss of sharing and undesired generalization) would still remain without the monomorphism restriction. Hughes [9, Section 6] shows a problem that arises exactly when the monomorphism restriction does not apply.

The trouble with implicit parameters begins when multiple configurations come into play in the same program, as Lewis et al. allow. We blame the trouble on the fact that implicit parameters express configuration dependencies in dynamic scopes, whereas we express those dependencies in static types. Dynamic scopes change as the program executes, whereas static types do not. Because dependencies should not change once established by the programmer, static types are more appropriate than dynamic scopes for carrying multiple configurations. Expressing value dependencies in static types is the essence of type classes, which our solution relies on. Because Haskell programmers are already familiar with type classes, they can bring all their intuitions to bear on the propagation of configuration data, along with guarantees of coherence. In particular, a type annotation can always be added without ill effects.

We ask the programmer to specify which configurations to pass where by giving type annotations. Taking advantage of type flow as distinct from data flow in this way enables notation that can be more flexible than extending the term language as Lewis et al. propose, yet more concise than passing function arguments explicitly. Appendix A shows a real-world example, where we contrast our type-based approach more concretely with the scope-based approach of implicit parameters.

Because we tie configuration dependencies to type variables, we can easily juggle multiple sets of configurations active in the same scope, such as multiple modular numbers with different moduli. More precisely, we use phantom types to distinguish between multiple instances of the same configuration class. For example, if two moduli are active in the same scope, two instances $Modular\ s_1\ a$ and $Modular\ s_2\ a$ are available and do not overlap with each other. Another way to make multiple instances available while avoiding the incoherence problem caused by overlapping instances is to introduce *named instances* into the language, as proposed by Kahl and Scheffczyk [13]. By contrast, when multiple implicit parameters with the same name and type are active in the same scope, Hughes [9] cautions that "programmers must just be careful!"

One way to understand our work is that we use the coherence of type classes to temper ambiguous overloading among multiple implicit parameters. There is a drawback to using types to propagate configurations, though: any dependency must be expressed in types, or the overloading will be rejected as unresolved or ambiguous. For example, whereas *sort* can have the type

$$sort :: (?compare :: a \to a \to Ordering) \Rightarrow [a] \to [a]$$

with implicit parameters, the analogous type on our approach

$$sort :: Compare\ s\ a \Rightarrow [a] \to [a] \quad \text{-- illegal}$$
$$\textbf{class}\ Compare\ s\ a\ \textbf{where}\ compare :: s \to a \to a \to Ordering$$

is illegal because the phantom type s does not appear in the type $[a] \to [a]$. Instead, we may write one of the following signatures.

$$sort_1 :: Compare\ s\ a \Rightarrow s \to [a] \to [a] \quad \text{-- ok}$$
$$sort_2 :: Compare\ s\ a \Rightarrow [M\ s\ a] \to [M\ s\ a] \quad \text{-- ok}$$

Using $sort_1$ is just like passing the comparison function as an explicit argument. Using $sort_2$ is just like defining a type class to compare values. Standard Haskell already provides for both of these possibilities, in the form of the *sortBy* function and the *Ord* class. We have nothing better to offer than using them directly, except we effectively allow an instance of the *Ord* class to be defined locally, in case a comparison function becomes known only at run time. By contrast, a program that uses only one comparison function (so that coherence is not at stake) can be written more succinctly and intuitively using implicit parameters, or even *unsafePerformIO*.

This problem is essentially the ambiguity of $show \circ read$. Such overloading issues have proven reasonably intuitive for Haskell programmers to grasp and fix, if only disappointedly. The success of type classes in Haskell suggests that the natural type structure of programs often makes expressing dependencies easy. Our examples, including the additional example in Appendix A, illustrate this point. Nevertheless, our use of types to enforce coherence incurs some complexity that is worthwhile only in more advanced cases of the configurations problem, when multiple configurations are present.

6.3 Other Related Work

Our use of FFI treats the type (class) system as foreign to values, and uses phantom types to bridge the gap. Blume's foreign function interface for SML/NJ [2] also uses phantom types extensively—for array dimensions, *const*-ness of objects, and even names of C structures. For names of C structures, he introduces type constructors for each letter that can appear in an identifier. The present paper shows how to reflect strings into types more frugally.

We showed how to specialize code at run time with different sets of primitive operations (such as for modular arithmetic). Our approach in this regard is related to overloading but specifically not partial evaluation, nor run-time code generation. It can however be fruitfully complemented by partial evaluation [10], for example when an integral modulus is fixed at compile time. In our approach, specialized code can use custom data representations.

The example in Section 5.2 shows that we effectively select a particular class instance based on run-time values. We are therefore "faking it" [21]—faking a dependent type system—more than before. McBride's paper [21] provides an excellent overview of various approaches to dependent types in Haskell. In approaches based on type classes, Haskell's coherence property guarantees that each type represents at most one value (of a given type), so compile-time type equality entails (that is, soundly approximates) run-time value equality. Appendix A demonstrates the utility of this entailment.

McBride mentions that, with all the tricks, the programmer still must decide if data belong in compile-time types or run-time terms. "The barrier represented by :: has not been broken, nor is it likely to be in the near future." If our *reflect* and especially *reify* functions have not broken the barrier, they at least dug a tunnel underneath.

7. CONCLUSIONS

We have presented a solution to the configurations problem that satisfies our desiderata. Although its start-up cost in complexity is higher than previous approaches, it is more flexible and robust, especially in the presence of multiple configurations. We have shifted the burden of propagating user preferences from the programmer to the type checker. Hence, the configuration data are statically typed, and differently parameterized pieces of code are statically separated. Type annotations are required, but they are infrequent and

mostly attached to top-level terms. The compiler will point out if a type annotation is missing, as a special case of the monomorphism restriction. By contrast, implicit parameters interact badly with the type system, with or without the monomorphism restriction.

Our solution leads to intuitive term notation: run-time configuration parameters can be referred to just like compile-time constants in global scope. We can propagate any type of configuration data—numbers, strings, polymorphic functions, closures, and abstract data like *IO* actions. Our code only uses *unsafePerformIO* as part of FFI, so no dynamic typing is involved. Furthermore, *unsafePerformIO* is unnecessary for the most frequent parameter types—numbers, lists, and strings. At run-time, our solution introduces negligible time and space overhead: linear in the size of the parameter data or pointers to them, amortized over their lifetimes. Our solution is available in Haskell today; this paper shows all needed code.

Our solution to the configurations problem lends itself to performance optimizations by dynamically dispatching to specialized, optimized versions of code based on run-time input values. The optimized versions of code may use specialized data representations, whose separation is statically guaranteed. Refactoring existing code to support such run-time parameterization requires minimum or no changes, and no code duplication.

Our approach relies on phantom types, polymorphic recursion, and higher-rank polymorphism. To propagate values via types, we build a family of types, each corresponding to a unique value. In one direction, a value is reified into its corresponding type by a polymorphic recursive function with a higher-rank continuation argument. In the other direction, a type is reflected back into its corresponding value by a type class whose polymorphic instances encompass the type family. In effect, we emulate local type-class instance declarations by choosing, at run time, the appropriate instance indexed by the member of the type family that reifies the desired dictionary. This emulation suggests adding local instances to Haskell, with a restriction that we conjecture preserves principal types and coherence. This technique allows Haskell's existing type system to emulate dependent types even more closely.

8. ACKNOWLEDGEMENTS

Thanks to Jan-Willem Maessen, Simon Peyton Jones, Andrew Pimlott, Gregory Price, Stuart Shieber, Dylan Thurston, and the anonymous reviewers for the 2004 ICFP and Haskell Workshop. The second author is supported by the United States National Science Foundation Grant BCS-0236592.

9. REFERENCES

[1] J. Adriano. Re: I need some help. Message to the Haskell mailing list; `http://www.mail-archive.com/haskell@haskell.org/msg10565.html`, 26 Mar. 2002.

[2] M. Blume. No-longer-foreign: Teaching an ML compiler to speak C "natively". In P. N. Benton and A. Kennedy, editors, *BABEL'01: 1st International Workshop on Multi-Language Infrastructure and Interoperability*, number 59(1) in Electronic Notes in Theoretical Computer Science, Amsterdam, Nov. 2001. Elsevier Science.

[3] A. Bromage. Dealing with configuration data. Message to the Haskell mailing list; `http://www.haskell.org/pipermail/haskell-cafe/2002-September/003411.html`, Sept. 2002.

[4] M. Chakravarty, S. Finne, F. Henderson, M. Kowalczyk, D. Leijen, S. Marlow, E. Meijer, S. Panne, S. L. Peyton Jones, A. Reid, M. Wallace, and M. Weber. The Haskell 98 foreign function interface 1.0: An addendum to the Haskell 98 report. `http://www.cse.unsw.edu.au/~chak/haskell/ffi/`, 2003.

[5] K. Claessen. Dealing with configuration data. Message to the Haskell mailing list; `http://www.haskell.org/pipermail/haskell-cafe/2002-September/003419.html`, Sept. 2002.

[6] C. V. Hall, K. Hammond, S. L. Peyton Jones, and P. L. Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2): 109–138, Mar. 1996.

[7] L. Hu et al. Dealing with configuration data. Messages to the Haskell mailing list; `http://www.haskell.org/pipermail/haskell-cafe/2002-September/thread.html`, Sept. 2002.

[8] J. Hughes. Restricted datatypes in Haskell. In E. Meijer, editor, *Proceedings of the 1999 Haskell Workshop*, number UU-CS-1999-28 in Tech. Rep. Department of Computer Science, Utrecht University, 1999.

[9] J. Hughes. Global variables in Haskell. *Journal of Functional Programming*, 2001. To appear. `http://www.cs.chalmers.se/~rjmh/Globals.ps`.

[10] M. P. Jones. Dictionary-free overloading by partial evaluation. In *Proceedings of the 1994 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, New York, 1994. ACM Press.

[11] M. P. Jones. Type classes with functional dependencies. In G. Smolka, editor, *Programming Languages and Systems: Proceedings of ESOP 2000, 9th European Symposium on Programming*, number 1782 in Lecture Notes in Computer Science, pages 230–244, Berlin, 2000. Springer-Verlag.

[12] W. Kahan. How Java's floating-point hurts everyone everywhere. Invited talk, ACM 1998 Workshop on Java for High-Performance Network Computing; `http://www.cs.ucsb.edu/conferences/java98/papers/javahurt.pdf`, 1 Mar. 1998.

[13] W. Kahl and J. Scheffczyk. Named instances for Haskell type classes. In R. Hinze, editor, *Proceedings of the 2001 Haskell Workshop*, number UU-CS-2001-23 in Tech. Rep., pages 71–99. Department of Computer Science, Utrecht University, 2 Sept. 2001.

[14] O. Kiselyov. Pure file reading (was: Dealing with configuration data). Message to the Haskell mailing list; `http://www.haskell.org/pipermail/haskell-cafe/2002-September/003423.html`, Sept. 2002.

[15] O. Kiselyov and C.-c. Shan. Functional pearl: Implicit configurations—or, type classes reflect the values of types. In *Proceedings of the 2004 Haskell Workshop*, New York, 2004. ACM Press.

[16] I. Koren. *Computer Arithmetic Algorithms*. A K Peters, Natick, MA, 2002.

[17] J. Launchbury and S. L. Peyton Jones. Lazy functional state threads. In *PLDI '94: Proceedings of the ACM Conference on Programming Language Design and Implementation*, volume 29(6) of *ACM SIGPLAN Notices*, pages 24–35, New York, 1994. ACM Press.

[18] J. Launchbury and S. L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, Dec. 1995.

[19] J. R. Lewis, M. B. Shields, E. Meijer, and J. Launchbury. Implicit parameters: Dynamic scoping with static types. In *POPL '00: Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, pages 108–118, New York, 2000. ACM Press.

[20] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *POPL '95: Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, pages 333–343, New York, 1995. ACM Press.

[21] C. McBride. Faking it: Simulating dependent types in Haskell. *Journal of Functional Programming*, 12(4–5):375–392, 2002.

[22] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, Apr. 1985.

[23] C. Okasaki. From fast exponentiation to square matrices: An adventure in types. In *ICFP '99: Proceedings of the ACM International Conference on Functional Programming*, volume 34(9) of *ACM SIGPLAN Notices*, pages 28–35, New York, 1999. ACM Press.

[24] OpenSSL. The open source toolkit for SSL/TLS. Version 0.9.7d; `http://www.openssl.org/`, 17 Mar. 2004.

[25] B. Parhami. *Computer Arithmetic: Algorithms and Hardware Designs*. Oxford University Press, New York, 2000.

[26] S. L. Peyton Jones. Solution to the monomorphism restriction/implicit parameter problem. Message to the Haskell mailing list; `http://www.haskell.org/pipermail/haskell/2003-August/012412.html`, 5 Aug. 2003.

[27] S. L. Peyton Jones and M. B. Shields. Lexically-scoped type variables, Mar. 2002. To be submitted to Journal of Functional Programming.

[28] C. Queinnec. *Lisp in Small Pieces*. Cambridge University Press, Cambridge, 1996.

[29] G. Russell. Initialisation without unsafePerformIO. Message to the Haskell mailing list; `http://www.haskell.org/pipermail/haskell/2004-June/014104.html`, June 2004.

[30] M. A. Soderstrand, W. K. Jenkins, G. A. Jullien, and F. J. Taylor, editors. *Residue Number System Arithmetic: Modern Applications in Digital Signal Processing*. IEEE Computer Society Press, Washington, DC, 1986.

[31] D. Thurston. Modular arithmetic. Messages to the Haskell mailing list; `http://www.haskell.org/pipermail/haskell-cafe/2001-August/002132.html`; `http://www.haskell.org/pipermail/haskell-cafe/2001-August/002133.html`, 21 Aug. 2001.

[32] V. Trifonov. Simulating quantified class constraints. In *Proceedings of the 2003 Haskell Workshop*, pages 98–102, New York, 2003. ACM Press.

[33] P. L. Wadler and S. Blott. How to make *ad-hoc* polymorphism less *ad hoc*. In *POPL '89: Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, pages 60–76, New York, 1989. ACM Press.

APPENDIX

A. ANOTHER EXAMPLE: JAVA NATIVE INTERFACE

This appendix shows a real-world example in which the natural type structure of the program lends itself to phantom-type annotations on not just one type (like *M* in modular arithmetic) but many types. We contrast our solution with the implicit parameters solution originally given by Lewis et al. [19, Section 4.4].

The problem arises in a Haskell binding to the Java Native Interface (JNI). JNI is to Java as FFI is to Haskell: the purpose of JNI is for Java code to call and be called by code written in C and (from there) Haskell (say). These calls require passing an abstract *JNIEnv* value back and forth, which points to a virtual method table (approximated with *Int* below) and roughly corresponds to a thread in the Java virtual machine. Lewis et al. propose that this *JNIEnv* value be passed as an implicit parameter.

```
type JNIEnv = Ptr Int
```

To take Lewis et al.'s example, suppose we want to implement the following Java class with a method written in Haskell.

```
class HaskellPrompt {
  String prompt;
  native String ask();
}
```

The `ask` method is supposed to display the `prompt` string, then read and return a line of input. Although the `ask` method appears to take no arguments on the Java side, the corresponding Haskell-side function *ask* takes two arguments: a *JNIEnv* pointer representing the current thread, and a *Jobject* pointer representing the `HaskellPrompt` object whose `ask` method is being invoked.

```
ask :: JNIEnv → Jobject → IO JString
```

To do its job, *ask* needs to access the `prompt` field of the *Jobject* it received, as well as create a new Java `String` containing the user's response. JNI provides myriad utility functions for such operations, each of which takes *JNIEnv* as the first argument.[6]

```
getObjectClass      :: JNIEnv → Jobject → IO Jclass
getFieldID          :: JNIEnv → Jclass →
                           String → String → IO FieldID
getObjectField      :: JNIEnv →
                           Jobject → FieldID → IO JString
getStringUTFChars   :: JNIEnv → JString → IO String
newStringUTF        :: JNIEnv → String → IO JString
```

Passing parameters explicitly, then, the *ask* function can be implemented as follows.

```
ask jnienv this =
  do cls      ← getObjectClass jnienv this
     field    ← getFieldID jnienv cls "prompt"
                    "Ljava/lang/String;"
     jprompt  ← getObjectField jnienv this field
     prompt   ← getStringUTFChars jnienv jprompt
     putStr prompt
     answer   ← getLine
     newStringUTF jnienv answer
```

It is tedious to pass the same *JNIEnv* argument all over the place, as we have to above.

Suppose we move the *JNIEnv* argument into an implicit parameter. That is, suppose we change the signatures of the JNI utility functions to the following.

```
getObjectClass      :: (?jnienv :: JNIEnv) ⇒
                           Jobject → IO Jclass
getFieldID          :: (?jnienv :: JNIEnv) ⇒ Jclass →
                           String → String → IO FieldID
getObjectField      :: (?jnienv :: JNIEnv) ⇒
                           Jobject → FieldID → IO JString
getStringUTFChars   :: (?jnienv :: JNIEnv) ⇒
                           JString → IO String
newStringUTF        :: (?jnienv :: JNIEnv) ⇒
                           String → IO JString
```

We can then write cleaner code for *ask*:

```
ask this =
  do cls      ← getObjectClass this
     field    ← getFieldID cls "prompt"
                    "Ljava/lang/String;"
     jprompt  ← getObjectField this field
     prompt   ← getStringUTFChars jprompt
     putStr prompt
     answer   ← getLine
     newStringUTF answer
```

Gone is the tedious sprinkle of *jnienv* throughout our code. Moreover, the compiler automatically infers the correct type for *ask*:

```
ask :: (?jnienv :: JNIEnv) ⇒ Jobject → IO JString
```

However, nothing prevents the programmer from mixing up one *JNIEnv* with another. For example, the following code overrides the value of ?*jnienv* during the call to *getFieldID*.

```
ask this = do cls   ← getObjectClass this
              field ← getFieldID cls "prompt"
                          "Ljava/lang/String;" with ?jnienv = ···
              ...
```

This code passes the type-checker, but is disallowed by JNI.

Using the technique in this paper, we can pass *JNIEnv* values implicitly while statically preventing this illicit mixing. To apply our approach, we need to involve a phantom type in the signatures of functions like *getObjectClass* and *ask*. As in Section 5.2, one phantom type suffices for any number of parameters, now or potentially added later. Fortunately, as is often the case, our code already uses many custom types, namely JNI's *Jobject*, *Jclass*, and *JString*. These types are the ideal host for a parasitic phantom type.

We change the abstract types *Jobject*, *Jclass*, and *JString* to take a phantom-type argument. They become *Jobject s*, *Jclass s*, and *JString s*. The *JNIEnv* can then piggyback on any of these types, without affecting the run-time representation of any data or otherwise inflicting too much pain.

```
data Jobject s   -- abstract
data Jclass  s   -- abstract
data JString s   -- abstract
```

We use the fact that *JNIEnv* is a *Storable* type to reify it. Because all JNI calls take place in the *IO* monad, we no longer need to resort to *unsafePerformIO* for reification and reflection, however safe it was to do so in Section 4.2.

```
data JNIENV s

class JNI s where jnienv :: s → IO JNIEnv
instance ReflectNums s ⇒ JNI (JNIENV s) where
  jnienv _ = alloca $ λp → do pokeArray (castPtr p) bytes
                              peek p
    where bytes = reflectNums (⊥ :: s) :: [Byte]

reifyJNIEnv :: JNIEnv → (∀s. JNI s ⇒ s → IO w) → IO w
reifyJNIEnv jnienv k =
```

[6]For simplicity, we assume that the *getObjectField* function returns a *JString*. To be more precise, it returns a *Jobject* that in our case can be coerced to a *JString*.

```
do bytes ← with jnienv (peekArray (sizeOf jnienv) ∘ castPtr)
     reifyIntegrals (bytes :: [Byte])
                    (λ(_ :: s) → k (⊥ :: JNIENV s))
```

We then assign new type signatures to the JNI utility functions.

```
getObjectClass      :: JNI s ⇒ Jobject s → IO (Jclass s)
getFieldID          :: JNI s ⇒ Jclass s →
                           String → String → IO FieldID
getObjectField      :: JNI s ⇒ Jobject s →
                           FieldID → IO (JString s)
getStringUTFChars :: JNI s ⇒ JString s → IO String
newStringUTF        :: JNI s ⇒ String → IO (JString s)
```

The cleaner version of *ask* above, written to use implicit parameters, works exactly as is! Of course, our approach assigns it a different type signature:

```
ask :: JNI s ⇒ Jobject s → IO (JString s)
```

Moreover, it no longer type-checks to call *getObjectClass* with one *JNIEnv* and *getFieldID* subsequently with another *JNIEnv*, as allowed under the implicit parameters approach. The mismatch is caught under our approach, because the return type *Jclass s* from *getObjectClass* is unified against the argument type *Jclass s* to *getFieldID*. Incidentally, it is semantically significant for us to associate the implicit *JNIEnv* value with other *J*-types: for example, a *Jobject* pointer makes sense only in the context of the *JNIEnv* where it was obtained. Thus our approach statically ensures an invariant of JNI. This kind of invariant is present in many application frameworks—graphical toolkits, database libraries, and so on.

Unlike with implicit parameters, we can ensure that *JNIEnv* values are equal at run time by unifying phantom types at compile time. We can do so because, as mentioned in Section 3.2, each type *s* corresponds to at most one *JNIEnv* value. This uniqueness guarantee in turn obtains because Haskell prohibits overlapping instances to ensure the coherence of overloading.

One advantage shared by implicit parameters and our approach is the ability to interact transparently with higher-order combinators. For example, consider the *handle* function that is part of Haskell's exception facility.

```
handle :: (Exception → IO a) → IO a → IO a
```

The *handle* function takes two arguments, an exception handler and an *IO* action. The exception handler is invoked in case the *IO* action throws an exception. In our *ask* function, to prepare for times when the Java virtual machine is not feeling well, we can wrap *handle* around some monadic code, as follows.

```
ask′ this             = handle handler (ask this)
handler exception = newStringUTF "error"
```

Even though the exception *handler* is outside the lexical scope of *ask* and *ask′*, the necessary configuration information is still propagated. Haskell knows that an exception handler must use the same *JNIEnv* value to invoke *newStringUTF* as the main *ask* function does, because both arguments to *handle* return the same type *IO a*, or *IO* (*JString s*) with the same phantom type *s*. Here, as in Section 5.1, the configuration information flows from the return type of a function (*handle*) to its arguments, and from one argument to another—not necessarily in the same direction as data flow. Here, just as in Section 3.2, it is more natural to propagate configurations via types than via terms: The two arguments to *handle*, like multiple branches of a **case** expression, can receive the same configuration via a single type signature.

By contrast, if we chose to solve the configurations problem by putting *JNIEnv* information in a reader monad transformer applied to the *IO* monad, then the *handle* combinator would need to undergo custom lifting in order to have the right type in order to operate on the lifted *IO* monad. That is, if we lift *IO* to *IO′*, then we also need to lift *handle* to the type (*Exception* → *IO′ a*) → *IO′ a* → *IO′ a*. This is a special case of the long-standing issue of lifting monadic operations through a monad transformer [20].

B. FOREIGN FUNCTION INTERFACE

The following type signatures summarize the part of Haskell's foreign function interface that Section 4 uses.

```
unsafePerformIO :: IO a → a
castPtr             :: Ptr a → Ptr b
```

A value of type *Ptr a* points to a storage area to or from which Haskell values of type *a*, where *a* belongs to the *Storable* type class, may be marshalled.

```
alloca      :: Storable a ⇒ (Ptr a → IO b) → IO b
peek        :: Storable a ⇒ Ptr a → IO a
peekArray :: Storable a ⇒ Int → Ptr a → IO [a]
pokeArray :: Storable a ⇒ Ptr a → [a] → IO ()
sizeOf      :: Storable a ⇒ a  {-unused-}   → Int
with        :: Storable a ⇒ a → (Ptr a → IO b) → IO b
```

A value of type *StablePtr a* refers to a Haskell value of type *a* that will not be garbage-collected until *freeStablePtr* is called.

```
newStablePtr    :: a → IO (StablePtr a)
deRefStablePtr :: StablePtr a → IO a
freeStablePtr    :: StablePtr a → IO ()
```