Submitted to the *Ninth International Conference on Foundations of Object-Oriented Languages*

First-Class Modules for Haskell

Mark Shields Simon Peyton Jones
Microsoft Research Cambridge
{markshie, simonpj}@microsoft.com

Abstract

Though Haskell's module language is quite weak, its core language is highly expressive. Indeed, it is tantalisingly close to being able to express much of the structure traditionally delegated to a seperate module language. However, the encodings are awkward, and some situations can't be encoded at all.

In this paper we refine Haskell's core language to support *first-class modules* with many of the features of ML-style modules. Our proposal cleanly encodes signatures, structures and functors with the appropriate type abstraction and type sharing, and supports recursive modules. All of these features work across compilation units, and interact harmoniously with Haskell's class system. Coupled with support for staged computation, we believe our proposal would be an elegant approach to run-time dynamic linking of structured code.

Our work builds directly upon Jones' work on parameterised signatures, Odersky and Läufer's system of higher-ranked type annotations, Russo's semantics of ML modules using ordinary existential and universal quantification, and Odersky and Zenger's work on nested types. We motivate the system by examples, and include a more formal presentation in the appendix.

1 Introduction

There are two competing techniques for expressing the large-scale structure of programs. The "brand leader" is the two-level approach, in which the language has two layers: a *core language*, and a *module language*. The most sophisticated example of this structure is ML and its variants, but many other languages, such as Haskell or Modula, take the same form, only with weaker module languages.

In the last few years, however, the core language of (extended versions of) Haskell has become very rich, to the point where it is tantalisingly close to being able to compete in the large-scale-structure league. If that were possible, it would of course be highly desirable: it would remove the need for a second language; and it would automatically mean that modules were first-class citizens, so that functors become ordinary functions.

The purpose of this paper is to show that, by bringing together several separate pieces of existing work, we can indeed bridge this final gap. More specifically, we propose several more-or-less orthogonal extensions to Haskell that work together towards this goal.

- Record types, with fields of polymorphic type, dot notation, and the ability to use a single field name in distinct record types (Section 2.1).
- Nested type declarations inside such records (Section 2.3). These nested declarations are purely syntactic sugar; there is nothing complicated.
- First-class universal and existential quantification (Section 4.1). Together with record types, this allows us conveniently to express the types of (generative) functors.
- A declaration-oriented construct for opening an existentially-quantified value (Section 4.2), together with a notation to allow opened types to appear in type annotations (Section 4.3). The standard approach is expression-oriented, which is unbearably clumsy in practice, whereas our construct works fine at the top level (Section 4.4).

Taken individually, all of these ideas have been proposed before. Our contribution is to put them all together in a coherent design for a core language that can reasonably claim to compete with, and in some ways improve on, the ML brand leader. In particular, our system treats module structures as first-class values, supports type inference, and interacts harmoniously with Haskell's constrained polymorphism. From the module point of view, it separates signatures from structures, and offers type abstraction, generative functors, type sharing, separate compilation, and recursive and nested signatures and structures. Our proposal is, at the top-level, fairly compatible with Haskell's existing module system (though for clarity we shall bend the syntax somewhat in this paper).

We present our system by a series of worked examples. A more formal presentation may be found in the appendix. At the time of writing we have only just begun to establish the formal properties of our system. We have, however, implemented a prototype interpreter which is available for download, and hope to merge these extensions into GHC, a production Haskell compiler.

2 Concrete Modules as Records

Following Jones [6], we encode *interfaces* as parameterised record types, and *implementations* as records. Haskell already has some record-like syntax for data constructors with named arguments, and many Haskell implementations allow these fields to be assigned a polymorphic type. However, our requirements are more demanding, as we wish to share field names between records, and allow nested type declarations. So we begin by introducing a new form of type declaration.

2.1 Parameterised Records

Record types are introduced (only) by explicit declaration, and may be parameterised:

```
record Set a f = {
  empty  :: f a
  add    :: a -> f a -> f a
  union  :: f a -> f a -> f a
  asList :: f a -> [a]
}
```

(Note that `f` has kind `Type -> Type`.) Equality between record types is nominal rather than structural. Unlike Haskell, a single field name may be re-used in different record types.

Record terms are constructed by applying a *record constructor* to a set of (possibly mutually recursive[1]) bindings:

```
intListSet :: Set Int []  {- inferred -}
intListSet = Set {
  empty  = []
  add    = \(x :: Int) xs -> x : filter (/= x) xs
  union  = foldr add
  asList = id
}
```

Record terms may be used within patterns, but we also support the usual "dot notation" for field projection:

```
one :: [Int]  {- inferred -}
one = intListSet.asList
        (intListSet.add 1 intListSet.empty)
```

As in Haskell, the type signature on a binding — such as `one :: [Int]` — is optional; the system will infer a type for `one`, but the programmer may constrain the type with a type signature.

Regarding a module as a record allows an ML functor to be replaced by an ordinary function. For example:

```
record EqR a = { eq :: a -> a -> Bool }

mkListSet :: forall a . EqR a -> Set a []
mkListSet eq = Set {
  empty = []
  add = \x xs ->
    x : filter (\y -> not (eq.eq x y)) xs
  asList = id
}
```

Since "functors" are ordinary functions, they integrate smoothly with Haskell's type class mechanism:

[1]This is another (cosmetic, but important) difference from Haskell 98 records.

```
mkListSet' :: forall a . Eq a => Set a []
mkListSet' = Set {
  empty = []
  add = \x xs -> x : filter ((/=) x) xs
  asList = id
}
```

By using the overloaded operator `(/=)` we have replaced the *explicit* parameterisation over the record `EqR a` with *implicit* parameterisation over the class `Eq a`.

Record fields may have polymorphic types:

```
record Monad f = {
  fmap :: forall a b . (a -> b) -> f a -> f b
  unit :: forall a . a -> f a
  bind :: forall a b . f a -> (a -> f b) -> f b
}
```

Such records may be constructed and taken apart in the same way as before:

```
listMonad :: Monad []   {- inferred -}
listMonad = Monad {
  fmap = map
  unit = \a -> [a]
  bind = \ma f -> concat (map f ma)
}

singleton :: a -> [a]
singleton x = listMonad.unit x
```

We do not permit subtyping or extensibility for records, deferring such extensions to future work.

2.2 Type inference

Type inference in this system is problematic. For example, consider:

```
g = \m f x -> m.fmap f (m.unit x)
```

Since `fmap` may be a field name in many records, the type of `m.fmap` depends on the type of `m` — which we do not know. We avoid this, and other difficulties relating to higher-ranked polymorphism, by placing imposing the *binder rule*: *the programmer must supply a type annotation for every lambda-bound, or letrec-bound, variable whose type mentions a record type constructor.* With the binder rule in place, it becomes easy to share field names between distinct record types. The binder rule is somewhat conservative — a clever inference engine could sometimes do without such an annotation — but it ensures that the typability of a program does not depend on the inference algorithm. We discuss alternative approaches in Section 6.

In practice, it may be tricky to give such a type annotation. In our example, the type of `m` is `Monad` α, where α is the type in which `g` is polymorphic. We provide two ways to solve this, both of which have been validated by practical experience in GHC. First, we can suppply a type signature for g rather than m:

```
g :: forall m a b .
       Monad m -> (a -> b) -> a -> m b
g = \m f x -> m.fmap f (m.unit x)
```

Here, we rely on the type checker to propagate the type annotation for g to an annotation for m, in the "obvious" way — this statement can be made precise, but we do not do so here. Alternatively, g's argument m may be annotated directly:

```
g = free t in \(m :: Monad t) f x ->
      m.fmap f (m.unit x)
```

Here the term `free t in ...` introduces a fresh *type* variable t standing for any type within the scope of a *term*. Thus g's first argument may be assigned any type of the form Monad τ for some type τ.

2.3 Nested Type Declarations

Modules typically contain a mix of term-level and type-level declarations. Following Odersky and Zenger [13], we allow record declarations to contain nested type declarations:

```
record BTSet a = {
  data BinTree = Leaf | Node BinTree a BinTree

  empty :: BinTree
  add :: a -> BinTree -> BinTree
}
```

A nested type may be projected from a type in much the same way as a field may be projected from a term. For various syntactical reasons, we write ^ instead of the usual . to denote type projection. For example, we may write:

```
unitSet :: BTSet a -> a -> Set a^BinTree
unitSet set a = set.add a set.empty
```

(^ binds weaker than type application.) Notice that there is another way of writing the signature for add in the above record declaration:

```
add :: a -> Set a^BinTree -> Set a^BinTree
```

Indeed, all four ways of writing add's type signature are equivalent: referring to a type relatively (by relying on the type declarations currently in scope) is equivalent to referring to it absolutely (by following a path from some top-level record type).

Since records are just another type declaration, they may also be nested within other records:

```
record Graph ver = {
  record Edge = { from :: ver; to :: ver }
  data   Rep  = Rep [ver] [Edge]

  mkGraph      :: [ver] -> [edge] -> Rep
  transClosure :: Rep -> Rep
}
```

We may reference nested data and record constructors within *terms* by a similar projection syntax:

```
leaf :: forall a . Set a^BinTree {- inferred -}
leaf = Set^Leaf
edge :: Graph Int^Edge            {- inferred -}
edge = Graph^Edge { from = 1; to = 2 }
```

Notice that type inference supplies the necessary type arguments for Set and Graph.

Record terms whose types contain nested types are constructed in the usual way:

```
trivGraph :: Graph () {- inferred -}
trivGraph = Graph {
    mkGraph = \vs es ->
      Rep [()] [Edge { from = (); to = () }]
    transClosure = \r -> r
  }
```

As for types, data and record constructors may be referred to relatively or absolutely.

Our approach to nested types diverges from the usual treatment of ML-style nested modules in two critical ways. Firstly, we never allow record *terms* to contain type declarations. (Later we will allow type declarations within top-level implementations, but this is merely a syntactical convenience.) As a consequence, our system avoids entirely the need for any dependent types, and manifestly respects the phase distinction [3] between type checking and evaluation. The work of Odersky and Zenger [13] takes a similar approach. Secondly, we never allow record types to contain *abstract* types, *i.e.*, types which are named but whose definition is hidden. (Again, we will later allow abstract type declarations within top-level interfaces, but again this is a syntactical convenience.)

Together, these restrictions mean that nested type declarations may always be flattened into a non-nested declarations. For example, our BTSet declaration may be rewritten:

```
data BTSet_BinTree a
  = BTSet_Leaf
  | BTSet_Node (BTSet_BinTree a) a
               (BTSet_BinTree a)

record BTSet a = {
  empty :: BTSet_BinTree a
  cmp :: a -> a -> BTSet_Cmp
  add :: a -> BTSet_BinTree a -> BTSet_BinTree a
}
```

Jones [6] advocates not supporting nested types on the grounds that they may always be translated away in this manner. We support them in our system because they are convenient, they subsume the usual namespace mechanism, and they turn out to be easily implemented.

3 Abstract Modules and Existentials

We now turn our attention to one of the essential properties of a module language: the ability to hide implementation types. As we mentioned in the Introduction there are two main approaches to implementation hiding, which we briefly review in this Section. The classic approach is to use existential types (Section 3.1), but the approach that has so far been more successful in practice, exemplified by ML, uses dependent sums (Section 3.2).

3.1 Type Abstraction in Haskell

In the `intListSet` example of Section 2.1 the representation type of sets as lists was exposed. This is bad, because a client of the module could pass *any* list to the `add` operation, whereas the implementation of `add` will expect the set it is passed to obey invariants maintained by the module (e.g. the list has no duplicates).

It has long been recognised that existential quantification provides an appropriate mechanism for hiding such a representation type [11]. Many Haskell implementations already support existential types, allowing us to write:[2]

```
data AbsSet a = exists f . MkAbsSet (Set a f)

intSet :: AbsSet Int  {- inferred -}
intSet = MkAbsSet intListSet
```

Consider typing the binding of `intSet`. Within the body of the `MkAbsSet` data constructor, `f` is bound to `[]`, and so the application `MkAbsSet intListSet` is well-typed. Outside of the `AbsSet` constructor, the existential quantifier over `f` hides this binding[3].

Programs wishing to use the operations of `intSet` must first "open" the existential quantification using a `case` expression:

```
one :: [Int]  {- inferred -}
one = case intSet of
        MkAbsSet s -> s.asList (s.add 1 s.empty)
```

Typing the arm of the case involves checking that the term `s.asList (s.add 1 s.empty)` is well-typed under the assumption $\mathtt{s} :: \mathsf{Set\ Int}\ F$ for *any* type constructor F. Equivalently, we must check the term `\s -> s.asList (s.add 1 s.empty)` has the polymorphic type `forall f . Set Int f ->` υ, for υ a type not containing `f`.

Often we wish to manipulate implementations containing abstract, but *equal* types, known as the "diamond import problem" [9] in the literature. For example, assume we have a function which, given any implementation of sets, generates some additional "helper" functions:

```
record SetHelp a f = {
  unionAll :: [f a] -> f a
}

mkSetHelp :: forall a f . Set a f -> SetHelp a f
mkSetHelp set = SetHelp {
  unionAll = foldr set.union set.empty
}
```

Now consider constructing some set helpers for our abstract `intSet`. Clearly we cannot simply write:

```
intSetHelp = mkSetHelp intSet
error: Type "AbsSet Int" is incompatible with
       type "Set Int f"
```

One way to avoid this mismatch between `AbsSet` and `Set` is to write a version of `mkSetHelp` which works on abstract sets directly:

```
data AbsSetHelp a
   = exists f . MkAbsSetHelp (SetHelp a f)

mkAbsSetHelp :: forall a .
                  AbsSet a -> AbsSetHelp a
mkAbsSetHelp absset
  = case absset of
      MkAbsSet set ->
        MkAbsSetHelp (mkSetHelp set)
```

Notice that we had to introduce (another) datatype, `AbsSetHelp`, to hide the representation of sets in `SetHelp`. Using this function, we may now write:

```
intSetHelp :: AbsSetHelp Int {- inferred -}
intSetHelp = mkAbsSetHelp intSet
```

However, `intSet` and `intSetHelp` may never be mixed, defeating the whole purpose of `mkAbsSetHelp`:

```
main = case (intSet, intSetHelp) of
         (MkAbsSet s, MkAbsSetHelp h) ->
           s.asList (h.unionAll
             [s.add 1 s.empty, s.add 2 s.empty])
error: arm of case is insufficiently polymorphic
```

Somehow we must convey the information that a particular set and its helpers share the same representation, without exposing the representation itself.

The only solution to this problem within Haskell is to carefully structure our program so that `intSet` is opened in a scope containing both the definition of `intSetHelp`, and all uses of these two terms which need to share their representation types:

[2]Somewhat confusingly, these implementations require the keyword `forall` to be used in this situation rather than `exists`.

[3]The alert reader may be alarmed by our use of existential quantification over higher-kinded type variables. Haskell uses a simple but incomplete unification algorithm for higher-kinded types which turns out to work very well in practice [4].

```
intSet = MkAbsSet intListSet

two :: forall f .
         Set Int f -> SetHelp Int f -> [Int]
two s h = s.asList (h.unionAll [s.add 1 s.empty,
                                s.add 2 s.empty])

main = case intSet of
         MkAbsSet s -> let h = mkSetHelp s
                       in two s h
```

These examples illustrate two serious drawbacks to the existential-type approach to type abstraction within Haskell:

(i) We are forced to introduce an entirely spurious datatype (*e.g.*, `AbsSet`) for every instance of type abstraction. This datatype is simply there to tell the type inference system where to expect existential types.

(ii) More seriously, this spurious datatype must be stripped away within a scope which covers all of the terms which need to share a particular implementation type. This is awkward in large programs, and impossible if uses of an abstract type must be split between compilation units.

3.2 Type Abstraction in ML/OCaml

These drawbacks have led most module language designers to abandon the simple-minded approach to type abstraction through existential quantification in preference for strong or translucent (dependent) sums [2] (the later are also known as manifest types [7]). For example, in OCaml our abstract set would be described by the signature:

```
module type SET =
  sig
    type a
    type 'a f
    val empty  : a f
    val add    : a -> a f -> a f
    val union  : a f -> a f -> a f
    val asList : a f -> a list
  end
```

Here the type constructor `f` is a nested type of `SET` which is left abstract. In ML-based module languages, signatures are not parameterised, and nested types are abstract by default. A binding for `f` must be supplied in any structure implementing signature `SET`:

```
module IntListSet : SET =
  struct
    type a = int
    type 'a f = 'a list
    let empty = []
    let add = fun x xs ->
                x :: filter (fun y -> y <> x) xs
    let union = fun xs ys -> fold_right add xs ys
    let asList = fun xs -> xs
  end
```

The binding of `f` to `list` in `IntListSet` is hidden by the explicit signature coercion. Of course, the binding of `a` to `int` is also hidden, even though this is probably not intended.

Sharing of abstract types is expressed using manifest types:

```
module type SETHELP =
  sig
    type a
    type 'a f
    unionAll : (a f) list -> a f
  end

module type MKSETHELP =
  functor (S : SET) ->
    (SETHELP with type a = S.a
                  type 'a f = 'a S.f)

module MkSetHelp : MKSETHELP =
  functor (S : SET) ->
    struct
      type a     = S.a
      type 'a f  = 'a S.f
      let unionAll = fold_right S.union
    end
```

Here the sharing of types `a` and `f` in the argument and result of the `MkSetHelp` functor is made explicit by the `with` clause in the functor's type.

To sum up: In OCaml, all nested types are abstract unless explicitly made manifest, while in Haskell all type parameters are concrete unless explicitly hidden by existential quantification.

4 Putting Existentials to Work

The dependent-sum approach to modular structure has proved to be very fruitful in practice. Nevertheless, there are strong reasons for continuing to search for alternatives. ML-style module systems can be extended to support both first-class and recursive modules but, although the details for these extensions have been worked out [16, 18], the resulting system is dauntingly complicated. Furthermore, it would be difficult to adopt such a system for Haskell, because the interaction with Haskell's system of *type classes* is entirely unclear. Indeed, no one has even attempted to work out the details for an ML-style module system supporting type classes. Lastly, there is an uncomfortable duplication of functionality between a rich core language and a rich module language; other things being equal, it would clearly be better to have a single layer.

So, instead of abandoning existentials for dependent sums, we shall tackle head-on the two problems we identified with existentials: the need for spurious datatypes (Section 4.1), and the need to open existentials in a common scope (Section 4.4).

4.1 Type Inference for Higher-ranked Polymorphism

We would like to get rid of the spurious data type `AbsSet` that we were forced to introduce in Section 3.1. The data type served to tell the type inference engine where to *introduce* existential quantification (at occurrences of the `MkAbsSet` constructor) and where to *eliminate* it (at `case` expressions that match `MkAbsSet`).

Instead, we would like to be able to use existential quantification freely within type schemes, without a mediating data type. For example, we'd like to write the `intSet` example directly, thus:

```
intSet :: exists f . Set Int f
intSet = Set {
  empty  = []
  add    = \x xs -> x : filter ((/=) x) xs
  asList = id
}
```

Existential quantifiers must now be able to occur in the result of a function type. For example, here are the types we would like for `mkListSet` and `mkListSet'`, which we saw in Section 2.1:

```
mkListSet :: forall a .
               EqR a -> exists f . Set a f

mkListSet' :: forall a . Eq a =>
                exists f . Set a f
```

The type signature for `mkListSet` expresses both that we can construct a set implementation from any equality on type `a`, and that for each such equality the representation type of the result is abstract. That is to say, this type signature mimics the *generative functor application* of ML. (We shall see in Section 4.2 that our system cannot mimic OCaml's *applicative* functors [8], and instead requires all type sharing to be made manifest.)

Our system supports higher-ranked signatures such as these by adopting the system of type annotations of Odersky and Läufer [12]. We extend the *binder rule* of Section 2.2 by requiring a type annotation on every lambda-bound, or letrec-bound, variable whose type uses existential or universal quantification. (Exception: in the case of letrec, when the universal quantification is at the top level, the annotation may be omitted, using the standard Hindley-Milner trick for recursive definitions.)

The Odersky/Läufer system strictly generalises the type inference algorithm used by those Haskell implementations already supporting rank-two polymorphism. Type inference reduces to solving a set of subsumption constraints over types with mixed prefix. For example, consider inferring the type of:

```
(\(f :: forall a . a -> Int -> a) -> f 1 2)
  (\x y -> x)
```

The system discovers that `\x y -> x` has most general type `forall b c . b -> c -> b`. Type checking the outer application reduces to checking

`forall b c . b -> c -> b` $\leq$ `forall a . a -> Int -> a`

where we write $\leq$ to denote "subsumes." The check proceeds by skolemizing the right-hand side quantified variables:

`forall b c . b -> c -> b` $\leq$ `a' -> Int -> a'`
where `a'` skolem constant

then freshening the left-hand-side quantifier variables:

`b' -> c' -> b'` $\leq$ `a' -> Int -> a'`
where `a'` skolem constant

and, finally, unifying the result. Since [`b'` $\mapsto$ `a'`, `c'` $\mapsto$ `Int`] is a most general unifier, the subsumption check succeeds.

We must extend the system of Odersky and Läufer in two ways. Firstly, we allow type schemes to arbitrarily mix universal and existential quantifiers. Though this adds no expressive power[4], it is a great aid when reporting type errors! The subsumption of existentials is exactly dual to that of universals.

Secondly, we must account for Haskell's class constraints. In particular, any quantifier may introduce a constraint, and we may need to decide constraint entailment during subsumption checking. Consider our previous example amended to include class constraints:

```
(\(f :: forall a . Num a => a -> Int -> a) ->
    f 1 2)
  (\x y -> if x == x then x else undefined)
```

To type check the outer application, the system must decide the subsumption:

`forall b c . Eq b => b -> c -> b` $\leq$
`forall a . Num a => a -> Int -> a`

The `Eq b` constraint arises from the use of `==`, and `Num a` from the type annotation on `f`. The check proceeds as before, skolemizing the right-hand side quantified variables, and freshening the left, to yield:

`Eq b' => b' -> c' -> b'` $\leq$ `Num a' => a' -> Int -> a'`
where `a'` skolem constant

Then the constraint `Num a'` is added to the set of "known" constraints:

`Eq b' => b' -> c' -> b'` $\leq$ `a' -> Int -> a'`
assuming `Num a'`, and `a'` skolem constant

For the moment, the `Eq b'` constraint is ignored, and the left- and right-hand side types are unified to yield the most general unifier [`b'` $\mapsto$ `a'`, `c'` $\mapsto$ `Int`]. Finally, the system must check that

`Num a'` $\vdash^{e}$ `Eq a'`

[4] E.g. the rank-one existential `exists a .` τ`(a)` may be replaced by the rank-two universal `forall b . (forall a .` τ`(a) -> b) -> b`.

where $\vdash^e$ denotes the constraint entailment relation. This is true, since `Eq` is a superclass of `Num`. Hence the term is well typed.

We have given only illustrative examples here, but the Appendix gives the technical details. This type inference algorithm is potentially more expensive than that used by existing Haskell implementations. In particular, the expensive operations of constraint simplification and generalisation occur, by default, for every step of type inference rather than just once per `let`-bound term. We plan to investigate refining the inference algorithm to avoid these operations as much as possible.

4.2 Opening Existentials

Now that we allow existentials to appear without a mediating data constructor, we must find a replacement for the rôle previously played by `case`. For example, recall from the previous section that:

```
intSet :: exists f . Set Int f
```

Attempting to project from `intSet` directly would lead to a type error:

```
one = intSet.asList (intSet.add 1 intSet.empty)
error: cannot project "empty" from term of
       non-record type "exists f . Set Int f"
```

Motivated by Russo's semantics for ML modules [17], we introduce a variation of `let` which explicitly "opens" any existential quantified type variables of the let-bound term:

```
one :: [Int]  {- inferred -}
one = let open s = intSet
      in s.asList (s.add 1 s.empty)
```

The keyword `open` indicates that the let-body `s.asList (s.add 1 s.empty)` should be type checked assuming `s :: Set Int f'`, where `f'` is a fresh (skolem) type constant replacing the existentially quantified `f` in the type of `intSet`[5]. By opening `intSet` explicitly we eliminate the existential quantifier on its type without compromising its type abstraction:

```
bad = let open s = intSet
      in s.add 2 [1]
error: Incompatible types "f'" and "[]", where
       type variable "f'" arises from open of
       "absIntSet"
```

Writing Γ to range over type and kind contexts, and Δ to range over kind contexts, we may write the typing rule for `let open` as follows[6]:

$$\frac{\begin{array}{c}\Gamma \vdash u : \texttt{exists}\ \Delta\ \texttt{.}\ \sigma \\ dom(\Delta) \cap dom(\Gamma) = \emptyset \\ \Gamma +\!\!+ \Delta, x : \sigma \vdash t : \sigma' \\ \Gamma \vdash \sigma' : \mathsf{scheme}\end{array}}{\Gamma \vdash \texttt{let open}\ x\ \texttt{=}\ u\ \texttt{in}\ t : \sigma'}\ \text{OPENLET}$$

Notice how the existentially quantified type variables Δ arising from u are "lifted over" the binding for x, and become free (skolemized) type variables when checking the type of t. The side condition on Δ ensures each existentially quantified type variable is indeed free—alpha-conversion may always be used to satisfy this condition. The check that σ' is a well-formed type scheme prevents any type variable in Δ from escaping the scope of x. For example, this term is ill-typed:

```
let open s = intSet
in s.empty
error: Skolemized type "f'" introduced in open of
       "s" escapes scope of binding in type
       "f' Int"
```

Without this restriction our system would be unsound:

```
let f = \x -> let open y =
  ((x, (== x)) :: exists a . (a, a -> Bool)) in y
in (snd (f 1)) (fst (f True)) -- Crash!
```

An alternative design would be to modify the typing rule for projection instead of that for `let`; in other words, make existential quantifiers transparent to projection. We prefer the present design because it makes explicit the *generative* nature of existential types. For example, the following term is (rightly) ill-typed, because it attempts to mix sets created from different equalities on `Int`:

```
inEq :: EqR Int  -- Normal equality on integers
z2Eq :: EqR Int  -- Equality mod 2

let open s1 = mkListSet intEq
in let open s2 = mkListSet z2Eq
in s1.asList s2.empty
error: Incompatible types "Set Int f1" and
       "Set Int f2", where type variable "f1"
       arises from open of "s1", and type
       variable "f2" arises fromm open of "s2"
```

A limitation of our approach is that we cannot mimic the *applicative* functors of OCaml:

```
let open s1 = mkListSet intEq
in let open s2 = mkListSet intEq
in s1.asList s2.empty
error: ...
```

Even though (thanks to the absence of side effects) `s1` and `s2` are observationally equivalent, the type system considers

[5] Haskell's existing monadic `do` notation also uses a binding construct whose left-hand and right-hand side types differ.

[6] See Figure 14 for the actual rule, which this only approximates.

their implementation types to be distinct. *All* type sharing in our system must be manifest; even this extreme case is rejected:

```
let open s1 = intSet
in let open s2 = intSet
in s1.asList s2.empty
error: ...
```

4.3 Type Annotations for Opened Bindings

Recall again our running example:

```
intSet :: exists f . Set Int f

one :: [Int]  {- inferred -}
one = let open s = intSet
      in s.asList (s.add 1 s.empty)
```

Is it possible for the programmer to give a type signature for `s`? The trouble is that, in the body of the `let`, `s` has type `Set Int f'`, where `f'` is a fresh (skolem) type constant, and the programmer has no way to write such a thing. Yet such annotations are desirable for documentation reasons, and will be absolutely essential when we come to top-level bindings (see Section 4.5).

Our solution is to add a new `open` form of type signature, dual to the `open` form of term binding:

```
one = let open s :: exists f. Set Int f
          open s = intSet
      in s.asList (s.add 1 s.empty)
```

The `open` type signature simply declares that `s` has the type obtained by opening (skolemizing) the type `exists f. Set Int f`. The type signature for `s` behaves exactly like any other type signature: it is optional, and may constrain the type to be less polymorphic than the inferred type.

However, we are not done yet. Suppose we write (rather artificially):

```
let open s :: exists f. Set Int f
    open s = intSet
in let t = s
in s.asList (t.add 1 t.empty)
```

How can we give a type signature to `t`? We cannot say:

```
open t :: exists f. Int -> f Int
t = s
```

because that would introduce a *fresh* skolemized type `f`, distinct from the one introduced by the type signature for `s`. Instead, we want to say "`t` has the same type as `s`". Following some preliminary work of Odersky and Zenger [13], we allow the programmer to say precisely that:

```
t :: s!
t = s
```

The *type* "x`!`" where x is an in-scope *term* variable, denotes the type of x[7]. This new type form can occur in any type. For example,

```
\x (y :: x!) . (x, y)
```

has type `forall a . a -> a -> (a, a)`, since the annotation on `y` forces it to have the same type as `x`. It is illegal to take the type of a variable of scheme type:

```
id :: forall a . a -> a   {- inferred -}
id = \x -> x

\(f :: id!) . f 1
error: "id" has a type scheme as its type, and
       cannot be dereferenced
```

Even this is not quite enough, however. Consider yet another version of our example:

```
let open s :: exists f. Set Int f
    open s = intSet
in let unit = \x -> s.add x s.empty
in s.asList (unit 1)
```

How can we write the type of `unit`? If `s` has type `Set Int` f, `unit` has type `Int ->` f `Int`. So we need to be able to refer to a *component* of `s`'s type. Again following Odersky and Zengler, we add another new type form, thus:

```
unit :: Int -> s!^f Int
unit = \x -> s.add x s.empty
```

The "`^f`" projects the `f`-component out of the type application `s!`. We use the type-variable names from the original definition of `Set` (back in Section 2.1) as the "field names" for these type projections.

These two new type forms give rise to a small algebra over types. For example, the following three types are all equal to `Int`:

$$\texttt{Set Int []\^{}a} \tag{1}$$
$$\texttt{(Int, Set Int [])\^{}t2\^{}a} \tag{2}$$
$$\texttt{(Set Int [] -> Int)\^{}arg\^{}a} \tag{3}$$

In (1) we know record `Set` has a type parameter named `a`, and this parameter is bound to `Int` in the application `Set Int []` (recall type application binds tighter than `^`). In (2), we rely on the built-in type parameters `t1`, `t2` *etc.* to refer to the successive type arguments of the tuple type constructor. Similarly, in (3) we rely on the built-in type parameters `arg` and `res` to refer to the argument and result types respectively of the function type constructor.

We also allow a record field to be dereferenced. For example:

[7] Though this notation involves term variables in type expressions, the type does not depend on the *value* of the term variable, only on its type.

```
unit :: Int -> s!^empty! Int
unit = \x -> s.add x s.empty
```

Here we say the result of `unit` has the same type as the `empty` field in the record type denoted by `s!`.

4.4 Opening Top-Level Bindings

So far we have not tackled the second of the two problems we identified in Section 3.1, namely that an existential must be opened over a scope that contains all terms that must share an implementation type. Indeed, we identified it as the more serious of the two problems.

The design we have presented so far was carefully chosen to solve this problem as well. *All that is needed is to allow `open` to be used for top-level bindings.*

Consider again the `intSet` and `intSetHelp` example of Section 3.1. Our improved support for existential quantification eliminates the need for any spurious `AbsSet` and `AbsSetHelp` datatypes. By using `open`, we may also both open and bind `intSet` in a single top-level declaration:

```
open intSet :: exists f . Set Int f
open intSet = intListSet
```

In the rest of the program, `intSet` has type `Set Int f'`, where `f'` is a fresh skolem type constant.

The `mkSetHelp` and `two` functions remain unchanged:

```
mkSetHelp :: forall a f . Set a f -> SetHelp a f
mkSetHelp set = SetHelp { ... }

two :: forall f . Set Int f ->
         SetHelp Int f -> [Int]
two s h = ...
```

With these definitions, we may now create `setHelp` directly:

```
setHelp = mkSetHelp intSet
main    = two intSet setHelp
```

Looking at the type of `mkHelpSet` we see `setHelp` has type `Set Int f'`, and thus the application of `two` is well-typed.

In Section 4.2 we mentioned that, to preserve soundness, skolemized type variables cannot escape the scope of the term variable which introduced them. Since the scope of a top-level binding is the entire program, this check is unnecessary for opened top-level signatures. This is indeed fortunate, since separate compilation means that we may not be able to "see" the entire scope of the binding.

4.5 Top-level Interfaces and Implementations

Haskell's existing module system combines the implementation of a module and its interface specification into a single compilation unit. In our system we split these notions. Roughly speaking, we take a top-level interface to be the body of a parameterless record type declaration, and a top-level implementation to be the body of a record, both appearing in a notional "cosmic" global scope.

Top-level interfaces appear in "hsi" files. For example, file `Lists.hsi` could look something like:

```
module Lists where

data List a = Nil | Cons a (List a)
map :: forall a b . (a -> b) -> List a -> List b
... etc ...
```

Such a file induces the type declaration in the "cosmic" scope:

```
record Lists = {
  data List a = Nil | Cons a (List a)
  map :: forall a b . (a -> b) -> List a-> List b
  ... etc ...
}
```

Top-level implementations appear in "hs" files. Continuing the above example, file `Lists.hs` could resemble:

```
module Lists where

map = ...
... etc ...
```

This induces the "cosmic" term declaration:

```
Lists :: Lists
Lists = Lists {
  map = ...
  ... etc ...
}
```

In effect, we simply introduce a term variable, `Lists`, into the initial type context with type `Lists`.

Both interfaces and implementations may `import` other interfaces. Interfaces supply enough type information to be able to type check implementations independently and in any order. Interfaces may be mutually recursive in their type declarations, subject to the usual rule that all recursion passes through a data or record constructor. Using type dereferencing, it is also possible to write mutually recursive type signatures (see Section 5.1).

We must be a little more generous with "cosmic" record type declarations and record terms in order to support `open` in signatures and bindings, and `instance` declarations.[8]

For example, we may have within `Lists.hsi` the declarations:

[8] Neither of these constructs make sense within arbitrary records. Allowing `open` anywhere leads to unsoundness for the same reason given in Section 4.2. Allowing `instance` declarations anywhere leads to local instance declarations [19] and would be a profound change to Haskell's type class system.

```
open llen :: exists a . forall b . List b -> a
isGT :: Int -> llen!^res -> Bool

instance eqList :: Eq a => Eq (List a)
```

Here we declare a function `llen` to calculate an abstract representation of a list's length which function `isGT` may test. We also have an instance declaration which is named `eqList` so that it may later be reconciled against its definition.

These declarations must have matching bindings within `Lists.hs`:

```
open llen = \xs -> xs
isGT n xs = case xs of
              Nil -> n > 0
              Cons _ xs' ->
                if n > 0 then isGT (n - 1) xs'
                         else False

instance eqList where ...
```

Haskell's existing module system allows top-level term and type bindings to be hidden. Our system supports a similar mechanism, though for brevity we do not consider it here.

5 Working Out The Details

In this section we complete our exposition by describing how existentials interact with recursion and type classes.

5.1 Recursive Abstract Types

Being a lazy language, Haskell allows top-level definitions to be arbitrarily mutually recursive. In this section we consider how mutual recursion interacts with our type abstraction mechanism.

Consider the recursive top-level definitions:

```
record Pair a b = { fst :: a; snd :: b }

x :: Pair Int Bool
x = Pair { fst = 1; snd = y.snd }

y :: Pair Int Bool
y = Pair { fst = x.fst; snd = True }
```

We now wish to hide the implementation types `Int` and `Bool`. Of course, for this example its easy to collapse the recursion into a single term:

```
open xy :: exists a b . (Pair a b, Pair a b)
open xy = let x = Pair { fst = 1; snd = y.snd }
              y = Pair { fst = x.fst; snd= True }
          in  (x, y)

x :: xy!^t1
x = fst xy
y :: xy!^t2
y = snd xy
```

However, this may be awkward in practice, and impossible if `x` and `y` must be defined in separate compilation units.

A better solution is to allow type dereferences to be mutually recursive:

```
open x :: exists a . Pair a (y!^b)
x = Pair { fst = 1; snd = y.snd }

open y :: exists b . Pair (x!^a) b
y = Pair { fst = x.fst; snd = True }
```

Notice the type-level recursion of `x!` and `y!` exactly mirrors the term-level recursion of `x` and `y`. Furthermore, even if `x` and `y` were defined in separate implementation files, both their signatures would be visible to the compiler within their respective interface files. Thus these mutually recursive bindings may be type checked in isolation.

We must be a little more restrictive on type-level recursion than term-level recursion. For example, all of the following bindings are rejected:

```
undefined :: undefined!
undefined = undefined
error: "undefined" has a cyclicly defined type

pair :: Pair (pair!^b) (pair!^a)
pair = Pair { fst = pair.snd; snd = pair.fst }
error: "pair" has a cyclicly defined type
```

They must instead be annotated in the usual way:

```
undefined  :: forall a . a
undefined = undefined

pair :: forall a . Pair a a
pair = Pair { fst = pair.snd; snd = pair.fst }
```

Why are the bindings for `x` and `y` accepted, while those for `undefined` and `pair` rejected? Roughly speaking, though `x` and `y` are mutually recursive, their resulting values are fully defined, and similarly their types. However, `undefined` and `pair` contain undefined elements, and hence their types in those positions remain undetermined.

To deal with this, we typecheck a recursive binding group in five phases; we illustrate using the `x`, `y` example of above.

1 The first phase skolemizes the existentially quantified type variables of all opened definitions, producing an environment that gives the types of `x` and `y`:

```
x :: Pair a' (y!^b)
y :: Pair (x!^a) b'
```

Here, `a'` and `b'` are the skolem types introduced to instantiate `a` and `b` respectively.

2 In phase 2, all types are rewritten to avoid any use of type dereference, type variable projection, and field projection.

Furthermore, relative types are rewritten to a canonical absolute form. We use a normal-order (call-by-name) evaluation strategy so as to accept as many recursively defined types as possible. A type of the form $x!$ is rewritten to the type of x already in the environment (though care must be taken to detect cycles.)

In our example, we rewrite the type of x as follows:

```
     Pair a' (y!^b)
---> Pair a' ((Pair (x!^a) b')^b)
---> Pair a' b'
```

After rewriting our environment, we have:

```
x :: Pair a' b'
y :: Pair a' b'
```

3 Next, we perform standard kind inference for the types in the new environment, which for Haskell reduces to type inference for a simply-typed lambda-calculus.

4 Next, we carry out standard type inference for the right-hand side of each binding, in the type environment computed by the earlier phases. In Haskell, type inference involves a weak form of higher-kinded kind-preserving unification. Since all relative types have been normalized to an absolute form, the equality theory on types is free.

```
Pair { fst = 1; snd = y.snd }   :: Pair Int b'
Pair { fst = x.fst; snd = True} :: Pair a' Bool
```

5 Lastly, we check that each right-hand side does indeed have the claimed existentially quantified type:

```
Pair { fst = 1; snd = y.snd }
  :: exists a. Pair a b'
Pair { fst = x.fst; snd = True}
  :: exists b. Pair a' b
```

(Notice that we must, of course, rewrite the original existential type signatures, just as in phase 2, to obtain the claimed types.)

A consequence of rewriting types (phase 2) before kind inference (phase 3) is that our system admits some very dubious looking type annotations. For example:

```
record Pair a b = { fst :: a; snd :: b }
strange :: (Pair, Int)^t1 Int Pair^a
strange = 1
```

In phase 2, the type annotation for strange is rewritten:

```
     (Pair, Int)^t1 Int Pair^a
---> Pair Int Pair^a
---> Int
```

Hence, kind inference finds nothing amiss here! We could perform kind inference before rewriting by augmenting the kind system with record kinds, but the additional complexity does not seem justified. Though confusing, these types are harmless.

The above exposition also applies to recursive let bindings. The only difference is that we must ensure no skolemized type variables escape the scope of the term as a whole. To ensure type inference remains complete in the presence of recursive bindings, we require that all letrec-bound variables be type annotated if any single letrec-bound variable is opened.

5.2 Type Classes and Existentials

So far we have used existential quantification to hide everything about a type parameter:

```
open intSet :: exists f . Set Int f
```

However, by exploiting Haskell's type class system we can selectively expose information about abstract types. For example, we can expose that f is a functor:

```
open intSet :: exists f . Functor f => Set Int f
intSet = intListSet
```

With this signature for intSet we have two interfaces for sets of integers. We have already been using the first *explicit* interface, which is simply the fields of Set reached via projection from intSet. The second *implicit* interface is provided by the overloaded operators of class Functor. These operations may be used directly. For example, the following term has type [Int]:

```
intSet.asList (fmap (+ 2)
  (intSet.add 1 intSet.empty))
```

Notice the use of the overloaded operator from the Functor class:

```
fmap :: forall f a b . Functor f =>
          (a -> b) -> f a -> f b
```

When type checking the binding of intSet, the system checks that Functor [] is satisfiable, then extends the known constraint context with Functor f', where f' is the skolemized type corresponding to f in the signature for intSet. Hence the function fmap may be instantiated at type (Int -> Int) -> f' Int -> f' Int.

In Haskell, an instance declaration allows the programmer to make a new data type into an instance of a given class. For example:

```
data Age = MkAge Int

instance Eq Age where
  (==) (MkAge i) (MkAge j) = i == j
```

Our `open` mechanism also introduces a new data type, the skolemized type constant, so it makes sense to allow it, too, to be an instance of a class. For example:

```
open absEq :: exists a . (a -> a -> Bool)
absEq = ((==) :: Int -> Int -> Bool)

instance Eq (absEq!^arg) where
  (==) = absEq
```

6 Related work

Our system draws together the work of four separate systems. Firstly, from Jones [6] work on Parameterised Signatures we took the idea that, at heart, a module implementation is just a record, and a module interface is just a record type with polymorphic fields parameterised over all its abstract types. The problems of type abstraction and type sharing then became almost trivial: we used ordinary existential quantification to hide types, and ordinary parametric polymorphism to capture type sharing. This approach avoided the need for dependent types, and thus automatically respected a phase distinction between types and terms. To further simplify matters, we disallowed anonymous records, and thus type equality for record types in our system is nominal.

Secondly, we adopted the annotations-based type system of Odersky and Läufer [12] to allow higher-ranked polymorphic types to be used in conjunction with type inference of rank-one types. In particular, this system allowed us to write existential quantifiers within the *result* type of functions, and thus write Haskell functions which mimic ML functors. This system also allowed us to share field names of polymorphic type between records without further complicating type inference. A little care had be taken to extend this system with support for Haskell's constrained polymorphism. Some existing Haskell implementations support rank-two polymorphism. Our extension of Odersky and Läufer's system can be seen as a natural generalisation of the existing type inference algorithm to arbitrary-ranked polymorphism. Another possibility would have been to abandon Hindley/Damas/Milner-style type inference in preference for local type inference [15, 14]. However, we felt that would have been too great a change for Haskell.

Thirdly, we examined Russo's semantics for ML signatures and structures [17] in order to understand how the *dot notation* of ML modules interacts with ordinary existential and universal polymorphism. As a result, we refined Haskell's `let` construct so as to be able to optionally *open* an existentially-quantified type within the scope of the let-binding. This new construct made it possible to access records of existential type using the dot notation, which in turn allowed records of abstract type to be used across compilation units. With this refinement in place, we may view our research agenda as one of refining Haskell to be as expressive as ML's language of *sematic objects* [10], and argue this is almost as convenient as programming in ML's module language directly.

Finally, we borrowed some notation (but, as it turns out, not the underlying type-theoretic machinery) from the work of Odersky and Zenger on Nested Types [13]. This notation allowed type annotations to capture type sharing of abstract types by refering to the type of other *term* variables in scope. This aspect of our system is probably the most unusual.

7 Conclusions

We tried to make each of our refinements as orthogonal as possible. That is to say, our proposal is not to add a monolithic module language to Haskell, but rather to refine Haskell's core language with a number of features which, taken together, capture the desired expressiveness.

The biggest deficiency of our system is that programs are subject to non-local changes when making a previously concrete types abstract. Not only must record types be changed to parameterise over such types, but all uses of those record types must be similarly changed to encode the appropriate propagation of type information. This has long been used as a justification for the move to dependent sum-based module systems [3, 1]. Hovever, we feel this will not be a problem in practice. After all, Haskell programmers have been getting by for years with a comparatively simple module system in which only top-level type abstraction was possible.

Our aim in this paper was to motivate each of our design decisions. Most of our effort to date has been invested in experimenting with a prototype compiler, which we have found to be an invaluable design tool. At the time of writing we have only just begun to establish the usual soundness and completeness properties.

We hope to transfer these ideas into GHC, an industrial-strength Haskell compiler. We have also begun to explore an extension of our system with *method constraints* [19], and believe this provides an expressive framework for *interface-oriented* programming.

Acknowledgements

We thank Claudio Russo for useful discussions.

References

[1] K. Crary, R. Harper, and D. Dreyer. A type system for higher-order modules. (To appear in POPL'02), Sept. 2001.

[2] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages, Portland, Oregon*, pages 123–137, Portland, Oregon, Jan. 1994.

[3] R. Harper, J. C. Mitchell, and E. Moggi. Higher-order modules and the phase distinction. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 341–354, 1990.

[4] M. P. Jones. A system of constructor classes: Overloading and implicit higher-order polymorphism. In *Pro-*

ceedings of the ACM SIGPLAN Conference on Functional Programming Languages and Computer Architecture (FPCA'93), Copenhagen, Denmark*, 1993.

[5] M. P. Jones. *Qualified Types: Theory and Practice.* Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.

[6] M. P. Jones. Using parameterized signatures to express modular structure. In *Proceedings of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida*, pages 68–78. ACM Press, Jan. 1996.

[7] X. Leroy. Manifest types, modules, and separate compilation. In *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages, Portland, Oregon*, pages 109–122. ACM Press, Jan. 1994.

[8] X. Leroy. Applicative functors and fully transparent higher-order modules. In *Proceedings of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95), San Francisco, California*, pages 142–153. ACM Press, Jan. 1995.

[9] D. B. MacQueen. Using dependent types to express modular structure. *Proceedings of the 13th ACM Symposium on Principles of Programming Languages, St. Petersburg, USSR*, pages 277–286, Jan. 1986.

[10] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised).* The MIT Press, Oct. 1997.

[11] J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

[12] M. Odersky and K. Läufer. Putting type annotations to work. In *Proceedings of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida*, pages 54–67. ACM Press, Jan. 1996.

[13] M. Odersky and C. Zenger. Nested types. In *Proceeding of the Workshop on Foundations of Object-Oriented Languages (FOOL8), London, UK*, Jan. 2001.

[14] M. Odersky, C. Zenger, and M. Zenger. Colored local type inference. In *Proceedings of the 28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01), London, England*, pages 41–53. ACM Press, Jan. 2001.

[15] B. C. Pierce and D. N. Turner. Local type inference. In *Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 252–265. ACM Press, Jan. 1998.

[16] C. V. Russo. First-class structures for Standard ML. *Nordic Journal of Computing*, 7(4):348–374.

[17] C. V. Russo. *Types for Modules.* PhD thesis, Department of Computer Science, The University of Edinburgh, June 1998.

[18] C. V. Russo. Recursive structures for Standard ML. In *Proceedings of the sixth ACM SIGPLAN International Conference on Functional Programming (ICFP'01), Firenze, Italy*, pages 50–61, Sept. 2001.

[19] M. Shields and S. Peyton Jones. Object-oriented style overloading for Haskell. In *First Workshop on Multi-language Inferastructure and Interoperability (BABEL'01), Firenze, Italy*, Sept. 2001.

A Type Checking

This section gives a brief overview of type checking for an idealized Haskell with our extensions. We offer this without any proofs of correctness: A proper account is in preparation.

Figure 1 presents the syntax. In common with many other module systems, we represent names by a pair, written X^i, of a base name X and a *stamp* i. This way it is always possible to alpha-convert a binder without changing its base name, which is significant when projecting types and fields. We assume stamps in source-program names begin as 0. (Actually, we could avoid stamps on class, term variable and instance names since type substitution can never capture such names. However, we retain them here for uniformity.)

Constructs marked "top-level only" may only appear within record declaration bodies or record term bodies at the outermost level of a program. We assume a program is a set of unparameterised record type declarations (corresponding to imported interfaces) and a single record term (corresponding to the implementation being compiled). In practice, each of these elements would reside in a separate file. To preserve soundness our system must reject any record declaration which shadows a top-level interface name. However, we allow shadowing in all other contexts.

Type contexts (ranged over by Γ) map type variables, record names, class names and term variables to their kinds, record bodies, class bodies and type schemes respectively. All four of these constructs have a separate name space. Kind contexts (ranged over by Δ) map only type variables to kinds. We let $dom(X)$ denote the domain of X, whether it be a type context, kind context, or record type declaration body.

Figure 2 presents rules which simultaneously kind-check and rewrite a type. Types of the form τ`^`a and τ`^`x`!` are expanded to the types they denote. Furthermore, all relative types are rewritten as absolute types. This is achieved by recording, with each record and class declaration in scope, an optional type *prefix* denoting the scope at which the declaration was introduced. When a relative record or class type is looked-up in the environment, it is replaced by the coresponding absolute type.

For example, in the declaration:

```
record A a = {
  record B b = { ... }
```

Type var names	a, b	
Type names	A, B	
Instance names	w	
Class names	C, D	
Term var names	x, y	
Integers/Stamps	i, j	
Kinds	κ	$::= \texttt{Type} \mid \kappa \texttt{ -> } \kappa'$
Types	τ, υ	$::= \texttt{Int} \mid \upsilon \texttt{ -> } \tau \mid \texttt{(} \overline{\tau} \texttt{)}$
		$\mid a^i\ \overline{\upsilon} \mid A^i\ \overline{\upsilon} \mid x^i \texttt{!}$
		$\mid \tau\texttt{\^{}}a\ \overline{\upsilon} \mid \tau\texttt{\^{}}A\ \overline{\upsilon} \mid \tau\texttt{\^{}}x\texttt{!}$
Type schemes	σ	$::= \texttt{forall } \Delta \texttt{ . } \overline{\phi} \texttt{ => } \sigma$
		$\mid \texttt{exists } \Delta \texttt{ . } \overline{\phi} \texttt{ => } \sigma$
		$\mid \sigma \texttt{ -> } \sigma'$
		$\mid \tau$
Primitive constraints	ϕ	$::= C^i\ \overline{\upsilon} \mid \tau\texttt{\^{}}C\ \overline{\upsilon}$
Constraint scheme	ξ	$::= \texttt{forall } \Delta \texttt{ . } \overline{\phi'} \texttt{ => } \phi$
Record decl bodies	S	$::= \texttt{record } A^i\ \Delta \texttt{ = \{ } S' \texttt{ \}} \mathrel{;} S$
		$\mid x^i \texttt{ :: } \sigma \mathrel{;} S$
(top-level only)		$\mid \texttt{open } x^i \texttt{ :: } \sigma \mathrel{;} S$
(top-level only)		$\mid \texttt{class } \overline{\phi} \texttt{ => } C^i\ \Delta \texttt{ where } \tau \mathrel{;} S$
(top-level only)		$\mid \texttt{instance } w \texttt{ :: } \xi \mathrel{;} S$
		$\mid \epsilon$
Record constructor	P	$::= A^i \mid P\texttt{\^{}}A$
Class constructor	Q	$::= C^i \mid P\texttt{\^{}}C$
Terms	t, u	$::= i \mid x^i \mid \backslash x^i\ \{\texttt{::}\ \sigma\}^{opt} \texttt{ . } t \mid t\ u$
		$\mid \texttt{(} \overline{t} \texttt{)} \mid \texttt{let (} \overline{x^i} \texttt{) = } u \texttt{ in } t$
		$\mid \texttt{let } \{\texttt{open}\}^{opt}\ x^i\ \{\texttt{::}\ \sigma\}^{opt}$
		$\quad \texttt{= } u \texttt{ in } t$
		$\mid \texttt{(} t \texttt{ :: } \sigma \texttt{)} \mid \texttt{free } \Delta \texttt{ in } t$
		$\mid P \texttt{ \{ } s \texttt{ \}} \mid t\texttt{.}x$
		$\mid \texttt{?}Q$
		$\mid \texttt{record } A^i\ \Delta \texttt{ = \{ } S \texttt{ \} in } t$
Record bodies	s	$::= \{\texttt{open}\}^{opt}\ x^i \texttt{ = } t \mathrel{;} s$
(top-level only)		$\mid \texttt{instance } w^i \texttt{ = } t \mathrel{;} s$
		$\mid \epsilon$
Type var contexts	Δ	$::= \Delta, a^i : \kappa \mid \cdot$
Type contexts	Γ	$::= \texttt{record } \{\tau\texttt{\^{}}\}^{opt}\ A^i\ \Delta \texttt{ = \{ } S \texttt{ \}}, \Gamma$
		$\mid \texttt{class } \overline{\phi} \texttt{ => } \{\tau\texttt{\^{}}\}^{opt} C^i\ \Delta$
		$\quad \texttt{where } \tau, \Gamma$
		$\mid a^i : \kappa, \Gamma$
		$\mid x^i : \sigma, \Gamma$
		$\mid \epsilon$
Constraint contexts	Ξ	$::= w^i : \xi, \Xi \mid \epsilon$
Programs	prog	$::= \overline{\texttt{record } A^i \texttt{ = \{ } S \texttt{ \}}} \mathrel{;} B^j \texttt{ \{ } s \texttt{ \}}$

Figure 1: Syntax

```
    x :: B Int
  }
```

the relative type `B Int` of `x` is rewritten to the absolute type `A a^B Int`, since the prefix for the declaration of `B` is the type

$$\boxed{\Gamma \vdash \tau : \kappa \hookrightarrow \tau'}$$

$$\frac{}{\Gamma \vdash \texttt{Int} : \texttt{Type} \hookrightarrow \texttt{Int}}$$

$$\frac{\Gamma \vdash \upsilon : \texttt{Type} \hookrightarrow \upsilon' \qquad \Gamma \vdash \tau : \texttt{Type} \hookrightarrow \tau'}{\Gamma \vdash (\upsilon \texttt{ -> } \tau) : \texttt{Type} \hookrightarrow (\upsilon' \texttt{ -> } \tau')}$$

$$\frac{\Gamma \vdash \overline{\tau : \texttt{Type} \hookrightarrow \tau'}}{\Gamma \vdash \texttt{(} \overline{\tau} \texttt{)} : \texttt{Type} \hookrightarrow \texttt{(} \overline{\tau'} \texttt{)}}$$

$$\frac{(a^i : \overline{\kappa \texttt{ ->}}\ \kappa') \in \Gamma \qquad \Gamma \vdash \overline{\upsilon : \kappa \hookrightarrow \upsilon'}}{\Gamma \vdash (a^i\ \overline{\upsilon}) : \kappa' \hookrightarrow (a^i\ \overline{\upsilon'})}$$

$$\frac{(\texttt{record } \{\tau\texttt{\^{}}\}^{opt}\ A^i\ (\overline{a^i : \kappa} +\!\!\!+ \overline{b^i : \kappa'}) \texttt{ = \{ } S \texttt{ \}}) \in \Gamma \qquad \Gamma \vdash \overline{\upsilon : \kappa \hookrightarrow \upsilon'}}{\Gamma \vdash (A^i\ \overline{\upsilon}) : \overline{\kappa' \texttt{ ->}}\ \texttt{Type} \hookrightarrow (\{\tau\texttt{\^{}}\}^{opt} A^i\ \overline{\upsilon'})}$$

$$\frac{(x^i : \tau) \in \Gamma \qquad \Gamma \vdash \tau : \texttt{Type} \hookrightarrow \tau'}{\Gamma \vdash x^i \texttt{!} : \texttt{Type} \hookrightarrow \tau'}$$

$$\frac{\Gamma \vdash \tau : \langle \theta \mid \Delta \mid S \rangle \qquad (a^i : \kappa') \in \Delta \qquad \Gamma \vdash (\theta\ a^i)\ \overline{\upsilon} : \kappa \hookrightarrow \upsilon'}{\Gamma \vdash (\tau\texttt{\^{}}a\ \overline{\upsilon}) : \kappa' \hookrightarrow \upsilon'}$$

$$\frac{\Gamma \vdash \tau : \langle \theta \mid \Delta \mid S \rangle \qquad (\texttt{record } A^i\ (\overline{a^i : \kappa} +\!\!\!+ \overline{b^i : \kappa'}) \texttt{ = \{ } S' \texttt{ \}}) \in S \qquad \Gamma \vdash \tau : \texttt{Type} \hookrightarrow \tau' \qquad \Gamma \vdash \overline{\upsilon : \kappa \hookrightarrow \upsilon'}}{\Gamma \vdash (\tau\texttt{\^{}}A\ \overline{\upsilon}) : \overline{\kappa' \texttt{ ->}}\ \texttt{Type} \hookrightarrow (\tau'\texttt{\^{}}A\ \overline{\upsilon'})}$$

$$\frac{\Gamma \vdash \tau : \langle \theta \mid \Delta \mid S \rangle \qquad (x^i : \upsilon) \in S \qquad \Gamma \vdash \upsilon : \texttt{Type} \hookrightarrow \upsilon'}{\Gamma \vdash (\tau\texttt{\^{}}x\texttt{!}) : \texttt{Type} \hookrightarrow \upsilon'}$$

Figure 2: Well-kinded types

`A a`.

Note that the record declarations within the body of a program *prog* have no prefix, since they are not contained within any other record declaration. Hence we write prefix types as $\{\tau\}^{opt}$, and write $\cdot$ to denote the absent prefix.

Figure 3 presents supporting rules for resolving a type into a record. Three components are derived: a substitution from the record's type arguments to their actual values in the type; the kind context of the record's arguments, and the record type declaration body, suitably instantiated. If the given type is not a record, a built-in record is returned where possible. For example, given:

$$\texttt{record A a\^{}C b c = \{ x :: (a, b, c) \}} \in \Gamma$$

$$\boxed{\Gamma \vdash \tau : \langle \theta \mid \Delta \mid S \rangle}$$

$$\frac{}{\Gamma \vdash \texttt{Int} : \langle \mathbf{Id} \mid \cdot \mid \epsilon \rangle}$$

$$\frac{}{\begin{array}{l}\Gamma \vdash (\upsilon \ \texttt{->}\ \tau) : \langle\ [\texttt{arg} \mapsto \upsilon, \texttt{res} \mapsto \tau] \mid \\ \qquad\qquad \texttt{arg} : \texttt{Type}, \texttt{res} : \texttt{Type} \mid \epsilon\ \rangle\end{array}}$$

$$\frac{}{\begin{array}{l}\Gamma \vdash (\ \overline{\tau}\ ) : \langle\ [\texttt{t1} \mapsto \tau_1, \ldots, \texttt{t}n \mapsto \tau_n] \mid \\ \qquad\qquad \texttt{t1} : \texttt{Type}, \ldots, \texttt{t}n : \texttt{Type} \mid \epsilon\ \rangle\end{array}}$$

$$\frac{\begin{array}{c}(\texttt{record}\ \{\tau\texttt{\^{}}\}^{opt}\ A^i\ \overline{a^i : \kappa}\ \texttt{=}\ \texttt{\{}\ S\ \texttt{\}}) \in \Gamma \\ \overline{a^i} \cap dom(\Gamma) = \emptyset \\ dom(S) \cap dom(\Gamma) = \emptyset \\ |\overline{\upsilon}| = |\kappa|\end{array}}{\Gamma \vdash (A^i\ \overline{\upsilon}) : \langle [\overline{a^i \mapsto \upsilon}] \mid \overline{a^i : \kappa} \mid [\overline{a^i \mapsto \upsilon}]\ S \rangle}$$

$$\frac{\begin{array}{c}(x^i : \tau) \in \Gamma \\ \Gamma \vdash \tau : \langle \theta \mid \Delta \mid S \rangle\end{array}}{\Gamma \vdash x^i\texttt{!} : \langle \theta \mid \Delta \mid S \rangle}$$

$$\frac{\begin{array}{c}\Gamma \vdash \tau : \langle \theta \mid \Delta \mid S \rangle \\ (a^i : \kappa') \in \Delta \\ \Gamma \vdash (\theta\ a^i)\ \overline{\upsilon} : \langle \theta' \mid \Delta' \mid S' \rangle\end{array}}{\Gamma \vdash (\tau\texttt{\^{}}a\ \overline{\upsilon}) : \langle \theta' \mid \Delta' \mid S' \rangle}$$

$$\frac{\begin{array}{c}\Gamma \vdash \tau : \langle \theta \mid \Delta \mid S \rangle \\ (\texttt{record}\ A^i\ \overline{a^i : \kappa}\ \texttt{=}\ \texttt{\{}\ S'\ \texttt{\}}) \in S \\ \overline{a^i} \cap dom(\Gamma) = \emptyset \\ dom(S') \cap dom(\Gamma) = \emptyset \\ |\overline{\upsilon}| = |\kappa|\end{array}}{\Gamma \vdash (\tau\texttt{\^{}}A\ \overline{\upsilon}) : \langle [\overline{a^i \mapsto \upsilon}] \mid \overline{a^i : \kappa} \mid [\overline{a^i \mapsto \upsilon}]\ S' \rangle}$$

$$\frac{\begin{array}{c}\Gamma \vdash \tau : \langle \theta \mid \Delta \mid S \rangle \\ (x^i :: \upsilon) \in S \\ \Gamma \vdash \upsilon : \langle \theta' \mid \Delta' \mid S' \rangle\end{array}}{\Gamma \vdash (\tau\texttt{\^{}}x\texttt{!}) : \langle \theta' \mid \Delta' \mid S' \rangle}$$

Figure 3: Record corresponding to a type

we have:

$$\begin{array}{l}\Gamma \vdash \texttt{C Int Bool} : \langle\ [\texttt{b} \mapsto \texttt{Int}, \texttt{c} \mapsto \texttt{Bool}] \\ \qquad\qquad\qquad \mid \texttt{b} : \texttt{Type}, \texttt{c} : \texttt{Type} \\ \qquad\qquad\qquad \mid \texttt{x} :: \texttt{(a, Int, Bool)}\ \rangle\end{array}$$

Figure 4 is the analogue of Figure 3 for primitive constraints, and allows the given primitive constraint to be resolved to a class. Three components are derived: the superclass context of the class declaration, the kind context of the class's arguments, and the class's body type, all suitably instantiated.

Figures 5 and 6 extends the kind-checking rules to primitive constraints and type schemes in the obvious way.

The next three figures formalize kind-checking record type

$$\boxed{\Gamma \vdash \phi : \langle \overline{\phi'} \mid \Delta \mid \tau \rangle}$$

$$\frac{\begin{array}{c}\overline{a^j} \cap dom(\Gamma) = \emptyset \\ (\texttt{class}\ \overline{\phi}\ \texttt{=>}\ \{\tau\texttt{\^{}}\}^{opt}\ C^i\ \overline{a^j : \kappa}\ \texttt{where}\ \tau') \in \Gamma \\ |\kappa| = |\upsilon|\end{array}}{\Gamma \vdash C^i\ \overline{\upsilon} : \langle [\overline{a^j \mapsto \upsilon}]\ \overline{\phi} \mid \overline{a^j : \kappa} \mid [\overline{a^j \mapsto \upsilon}]\ \tau' \rangle}$$

$$\frac{\begin{array}{c}\Gamma \vdash \tau : \langle \theta' \mid \Delta' \mid S' \rangle \\ \overline{a^j} \cap dom(\Gamma) = \emptyset \\ (\texttt{class}\ \overline{\phi}\ \texttt{=>}\ C^i\ \overline{a^j : \kappa}\ \texttt{where}\ \tau') \in S' \\ |\kappa| = |\upsilon|\end{array}}{\Gamma \vdash \tau\texttt{\^{}}C\ \overline{\upsilon} : \langle [\overline{a^j \mapsto \upsilon}]\ \overline{\phi} \mid \overline{a^j : \kappa} \mid [\overline{a^j \mapsto \upsilon}]\ \tau' \rangle}$$

Figure 4: Class corresponding to a primitive constraint

$$\boxed{\Gamma \vdash \phi \hookrightarrow \phi\ \textsf{constraint}}$$

$$\frac{\begin{array}{c}(\texttt{class}\ \overline{\phi}\ \texttt{=>}\ \{\tau\texttt{\^{}}\}^{opt}\ C^i\ \overline{a^j : \kappa}\ \texttt{where}\ \tau') \in \Gamma \\ \Gamma \vdash \upsilon : \kappa \hookrightarrow \upsilon'\end{array}}{\Gamma \vdash C^i\ \overline{\upsilon} \hookrightarrow \{\tau\texttt{\^{}}\}^{opt} C^i\ \overline{\upsilon'}\ \textsf{constraint}}$$

$$\frac{\begin{array}{c}\Gamma \vdash \tau : \langle \theta' \mid \Delta' \mid S' \rangle \\ (\texttt{class}\ \overline{\phi}\ \texttt{=>}\ C^i\ \overline{a^j : \kappa}\ \texttt{where}\ \tau') \in S' \\ \Gamma \vdash \tau : \texttt{Type} \hookrightarrow \tau' \\ \Gamma \vdash \overline{\upsilon : \kappa \hookrightarrow \upsilon'}\end{array}}{\Gamma \vdash \tau\texttt{\^{}}C\ \overline{\upsilon} \hookrightarrow \tau'\texttt{\^{}}C\ \overline{\upsilon'}\ \textsf{constraint}}$$

Figure 5: Well-kinded primitive constraints

declaration bodies. This is little involved because of the combination of recursive record bodies with type skolemization on "opened" signatures.

Figure 7 gives rules for collecting the skolemized type variables of a record body. The result is the set of these type variables, and a rewritten body in which each variable signature is augmented by its external type signature (which we write within braces). For ordinary (unopened) variables, the external type signature is identical to the supplied (internal) type signature. For opened variables, the external type signature is a copy of the supplied signature with any outermost existential quantifiers removed. For example, the signature:

```
open x :: exists a b . C a b =>
          forall c . a -> b -> c
```

is rewritten to the signature:

```
open x :: exists a b . C a b =>
          forall c . a -> b -> c
          { C a b => forall c . a -> b -> c }
```

assuming the type variables `a` and `b` are fresh.

Figure 8 shows how to extract both the type context and constraint context corresponding to a record body. This judgement is parameterised by a type prefix representing

$$\boxed{\Gamma \vdash \sigma \hookrightarrow \sigma' \ \mathsf{scheme}}$$

$$\frac{\begin{array}{c} dom(\Delta) \cap dom(\Gamma) = \emptyset \\ \forall i \ . \ fv(\phi) \cap dom(\Delta) \neq \emptyset \\ \Gamma \mathbin{+\!\!+} \Delta \vdash \overline{\phi \hookrightarrow \phi' \ \mathsf{constraint}} \\ \Gamma \mathbin{+\!\!+} \Delta \vdash \sigma \hookrightarrow \sigma' \ \mathsf{scheme} \end{array}}{\begin{array}{l} \Gamma \vdash \texttt{forall/exists} \ \Delta \ \texttt{.} \ \overline{\phi} \ \texttt{=>} \ \sigma \hookrightarrow \\ \quad \texttt{forall/exists} \ \Delta \ \texttt{.} \ \overline{\phi'} \ \texttt{=>} \ \sigma' \ \mathsf{scheme} \end{array}}$$

$$\frac{\begin{array}{c} \Gamma \vdash \sigma \hookrightarrow \sigma'' \ \mathsf{scheme} \\ \Gamma \vdash \sigma' \hookrightarrow \sigma''' \ \mathsf{scheme} \end{array}}{\Gamma \vdash (\sigma \ \texttt{->} \ \sigma') \hookrightarrow (\sigma'' \ \texttt{->} \ \sigma''') \ \mathsf{scheme}}$$

$$\frac{\Gamma \vdash \tau \hookrightarrow \tau' : \texttt{Type}}{\Gamma \vdash \tau \hookrightarrow \tau' \ \mathsf{scheme}}$$

Figure 6: Well-kinded type schemes

$$\boxed{\Gamma \vdash S \hookrightarrow \langle \Delta' \mid S' \rangle \ \mathsf{skolem}}$$

$$\frac{\begin{array}{c} dom(\Delta_1) \cap dom(\Gamma) = \emptyset \\ \Gamma \mathbin{+\!\!+} \Delta_1 \vdash S_1 \hookrightarrow \langle \Delta_2 \mid S_3 \rangle \ \mathsf{skolem} \\ \Gamma \mathbin{+\!\!+} \Delta_2 \vdash S_2 \hookrightarrow \langle \Delta_3 \mid S_4 \rangle \ \mathsf{skolem} \end{array}}{\begin{array}{l} \Gamma \vdash \texttt{record} \ A^i \ \Delta_1 \ \texttt{= \{} \ S_1 \ \texttt{\}} \ ; S_2 \hookrightarrow \\ \quad \langle \Delta_2 \mathbin{+\!\!+} \Delta_3 \mid \texttt{record} \ A^i \ \Delta_1 \ \texttt{= \{} \ S_3 \ \texttt{\}} \ ; S_4 \rangle \ \mathsf{skolem} \end{array}}$$

$$\frac{\begin{array}{c} dom(\Delta) \cap dom(\Gamma) = \emptyset \\ \Gamma \mathbin{+\!\!+} \Delta \vdash S \hookrightarrow \langle \Delta' \mid S' \rangle \ \mathsf{skolem} \end{array}}{\begin{array}{l} \Gamma \vdash \texttt{open} \ x^i \ \texttt{::} \ \texttt{exists} \ \Delta \ \texttt{.} \ \overline{\phi} \ \texttt{=>} \ \sigma \ ; S \hookrightarrow \\ \quad \langle \Delta \mathbin{+\!\!+} \Delta' \mid \texttt{open} \ x^i \ \texttt{::} \ \texttt{exists} \ \Delta \ \texttt{.} \ \overline{\phi} \ \texttt{=>} \ \sigma \\ \qquad\qquad \texttt{\{} \ \overline{\phi} \ \texttt{=>} \ \sigma \ \texttt{\}} \ ; S' \rangle \ \mathsf{skolem} \end{array}}$$

$$\frac{\Gamma \vdash S \hookrightarrow \langle \Delta \mid S' \rangle \ \mathsf{skolem}}{\begin{array}{l} \Gamma \vdash \{\texttt{open}\}^{opt} \ x^i \ \texttt{::} \ \sigma \ ; S \hookrightarrow \\ \quad \langle \Delta \mid \{\texttt{open}\}^{opt} \ x^i \ \texttt{::} \ \sigma \ \texttt{\{} \ \sigma \ \texttt{\}} \ ; S' \rangle \ \mathsf{skolem} \end{array}}$$

$$\frac{\Gamma \vdash S \hookrightarrow \langle \Delta' \mid S' \rangle \ \mathsf{skolem}}{\begin{array}{l} \Gamma \vdash \texttt{class} \ \overline{\phi} \ \texttt{=>} \ C^i \ \Delta \ \texttt{where} \ \tau \ ; S \hookrightarrow \\ \quad \langle \Delta' \mid \texttt{class} \ \overline{\phi} \ \texttt{=>} \ C^i \ \Delta \ \texttt{where} \ \tau \ ; S' \rangle \ \mathsf{skolem} \end{array}}$$

$$\frac{\Gamma \vdash S \hookrightarrow \langle \Delta \mid S' \rangle \ \mathsf{skolem}}{\begin{array}{l} \Gamma \vdash \texttt{instance} \ w^i \ \texttt{::} \ \xi \ ; S \hookrightarrow \\ \quad \langle \Delta \mid \texttt{instance} \ w^i \ \texttt{::} \ \xi \ ; S' \rangle \ \mathsf{skolem} \end{array}}$$

$$\frac{}{\Gamma \vdash \epsilon \hookrightarrow \langle \epsilon \mid \epsilon \rangle \ \mathsf{skolem}}$$

Figure 7: Skolemizing opened type schemes

the record body's scope of definition. The type context collects just the outmost declarations of the record body S. Notice that only the external signature of variable declarations is significant when forming type contexts. We record the type prefix of the record body with each nested record

$$\boxed{\{\tau\}^{opt} \vdash S : \langle \Gamma \mid \Xi \rangle}$$

$$\frac{\begin{array}{c} \overline{a^j} \cap fv(\{\tau\}^{opt}) = \emptyset \\ \{\tau\texttt{\^{}}\}^{opt} A^i \ \overline{a^j} \vdash S' : \langle \_ \mid \Xi' \rangle \\ \{\tau\}^{opt} \vdash S : \langle \Gamma \mid \Xi \rangle \end{array}}{\begin{array}{l} \{\tau\}^{opt} \vdash \texttt{record} \ A^i \ \overline{a^j : \kappa} \ \texttt{= \{} \ S' \ \texttt{\}} \ ; S : \\ \quad \langle \texttt{record} \ \{\tau\texttt{\^{}}\}^{opt} \ A^i \ \overline{a^j : \kappa} \ \texttt{= \{} \ S' \ \texttt{\}}, \Gamma \mid \Xi' \mathbin{+\!\!+} \Xi \rangle \end{array}}$$

$$\frac{\{\tau\}^{opt} \vdash S : \langle \Gamma \mid \Xi \rangle}{\begin{array}{l} \{\tau\}^{opt} \vdash \{\texttt{open}\}^{opt} \ x^i \ \texttt{::} \ \sigma \ \texttt{\{}\overline{\phi} \ \texttt{=>} \ \sigma'\texttt{\}} \ ; S : \\ \quad \langle x^i : \sigma', \Gamma \mid named(\overline{\phi}) \mathbin{+\!\!+} \Xi \rangle \end{array}}$$

$$\frac{\{\tau\}^{opt} \vdash S : \langle \Gamma \mid \Xi \rangle}{\begin{array}{l} \{\tau\}^{opt} \vdash \texttt{class} \ \overline{\phi} \ \texttt{=>} \ C^i \ \Delta \ \texttt{where} \ \tau \ ; S : \\ \quad \langle \texttt{class} \ \overline{\phi} \ \texttt{=>} \ \{\tau\texttt{\^{}}\}^{opt} \ C^i \ \Delta \ \texttt{where} \ \tau, \Gamma \mid \Xi \rangle \end{array}}$$

$$\frac{\{\tau\}^{opt} \vdash S : \langle \Gamma \mid \Xi \rangle}{\begin{array}{l} \{\tau\}^{opt} \vdash \texttt{instance} \ w^i \ \texttt{::} \ \xi \ ; S : \\ \quad \langle \Gamma \mid compname(\{\tau\}^{opt}, w^i) : \xi, \Xi \rangle \end{array}}$$

$$\frac{}{\{\tau\}^{opt} \vdash \epsilon : \langle \epsilon \mid \epsilon \rangle}$$

Figure 8: Extracting a type and constraint context from a record body

and class declaration so that relative types may be rewritten to absolute types (see Figure 2). The constraint context collects all instance declarations, and all constraints appearing underneath existentially quantified type variables in opened signatures. The latter are assigned arbitrary fresh names using the function *named*.

For reasons which will soon become clear, instance declarations must be added to the constraint context with a name which combines both the instance declaration name and the prefix of its scope of definition. We assume *compname* is some injective function from a type prefix and an instance name to a composite instance name. For example:

$$compname(\cdot, \texttt{w\^{}i}) = \texttt{w\^{}i}$$
$$compname(\texttt{A}, \texttt{w\^{}i}) = \texttt{A\_w\^{}i}$$

Figure 9 extends the kind checking judgments to record bodies which have already been rewritten by the rules of Figure 7. Since record bodies are mutually recursive, and since types may de-reference any term variable in scope, when kind-checking a nested record body we must first extend the type context with the definitions of the record body itself (suitably alpha-converted).

We may now turn our attention to type checking. Figure 10 presents constraint entailment for class constraints, which is entirely standard [5]. Figure 11 present rules for deciding type scheme subsumption. They extend the system presented in [12] to handle existential types and constraints.

Figure 12 presents rules to expand record constructors and

$$\boxed{\{\tau\}^{opt} \mid \Gamma \vdash S \hookrightarrow S' \text{ record}}$$

$$\frac{\begin{array}{c}\overline{a^j} \cap dom(\Gamma) = \emptyset \\ \{\tau\hat{}\}^{opt}\ A^i\ \overline{a^j} \vdash S_1 : \langle \Gamma' \mid \_ \rangle \\ dom(\Gamma') \cap dom(\Gamma) = \emptyset \\ \{\tau\hat{}\}^{opt}\ A^i\ \overline{a^j} \mid \Gamma \mathbin{+\!\!+} \overline{a^j : \kappa} \mathbin{+\!\!+} \Gamma' \vdash S_1 \hookrightarrow S_3 \text{ record}\end{array}}{\begin{array}{c}\{\tau\}^{opt} \mid \Gamma \vdash \texttt{record}\ A^i\ \overline{a^j : \kappa}\ \texttt{= \{}\ S_1\ \texttt{\}}\ ;\ S_2 \hookrightarrow \\ \texttt{record}\ A^i\ \overline{a^j : \kappa}\ \texttt{= \{}\ S_3\ \texttt{\}}\ ;\ S_4 \text{ record}\end{array}}$$

$$\frac{\begin{array}{c}\Gamma \vdash \sigma_1 \hookrightarrow \sigma_3 \text{ scheme} \\ \Gamma \vdash \overline{\phi \hookrightarrow \phi' \text{ constraint}} \\ \Gamma \vdash \sigma_2 \hookrightarrow \sigma_4 \text{ scheme} \\ \{\tau\}^{opt} \mid \Gamma \vdash S \hookrightarrow S' \text{ record}\end{array}}{\begin{array}{c}\{\tau\}^{opt} \mid \Gamma \vdash \{\texttt{open}\}^{opt}\ x^i\ \texttt{::}\ \sigma_1\ \{\overline{\phi}\ \texttt{=>}\ \sigma_2\}\ ;\ S \hookrightarrow \\ \{\texttt{open}\}^{opt}\ x^i\ \texttt{::}\ \sigma_3\ \{\overline{\phi'}\ \texttt{=>}\ \sigma_4\}\ ;\ S' \text{ record}\end{array}}$$

$$\frac{\begin{array}{c}\overline{a^j} \cap dom(\Gamma) = \emptyset \\ \Gamma \mathbin{+\!\!+} \overline{a^j : \kappa} \vdash \overline{\phi \hookrightarrow \phi' \text{ constraint}} \\ \Gamma \mathbin{+\!\!+} \overline{a^j : \kappa} \vdash \tau : \texttt{Type} \hookrightarrow \tau' \\ \{\tau\}^{opt} \mid \Gamma \vdash S \hookrightarrow S' \text{ record}\end{array}}{\begin{array}{c}\{\tau\}^{opt} \mid \Gamma \vdash \texttt{class}\ \overline{\phi}\ \texttt{=>}\ C^i\ \overline{a^j : \kappa}\ \texttt{where}\ \tau\ ;\ S \hookrightarrow \\ \texttt{class}\ \overline{\phi'}\ \texttt{=>}\ C^i\ \overline{a^j : \kappa}\ \texttt{where}\ \tau'\ ;\ S' \text{ record}\end{array}}$$

$$\frac{\begin{array}{c}dom(\Delta) \cap dom(\Gamma) = \emptyset \\ \Gamma \mathbin{+\!\!+} \Delta \vdash \phi \hookrightarrow \phi' \text{ constraint} \\ \Gamma \mathbin{+\!\!+} \Delta \vdash \overline{\phi'' \hookrightarrow \phi''' \text{ constraint}} \\ \{\tau\}^{opt} \mid \Gamma \vdash S \hookrightarrow S' \text{ record}\end{array}}{\begin{array}{c}\{\tau\}^{opt} \mid \Gamma \vdash \texttt{instance}\ w^i\ \texttt{::}\ \texttt{forall}\ \Delta\ \texttt{.}\ \overline{\phi''}\ \texttt{=>}\ \phi\ ;\ S \hookrightarrow \\ \texttt{instance}\ w^i\ \texttt{::}\ \texttt{forall}\ \Delta\ \texttt{.}\ \overline{\phi'''}\ \texttt{=>}\ \phi'\ ;\ S' \text{ record}\end{array}}$$

$$\frac{}{\Gamma \vdash \epsilon \hookrightarrow \epsilon \text{ record}}$$

Figure 9: Well-kinded record declaration bodies

$$\boxed{\Gamma \mid \Xi \vdash^p \phi}$$

$$\frac{(w^i : \phi) \in \Xi}{\Gamma \mid \Xi \vdash^p \phi} \qquad \frac{\begin{array}{c}\Gamma \mid \Xi \vdash^p \phi \\ \Gamma \vdash \phi : \langle \overline{\phi'} \mid \Delta \mid \tau \rangle\end{array}}{\Gamma \mid \Xi \vdash^p \phi'_i}$$

$$\boxed{\Gamma \mid \Xi \vdash^e \phi}$$

$$\frac{\Gamma \mid \Xi \vdash^p \phi}{\Gamma \mid \Xi \vdash^e \phi} \qquad \frac{\begin{array}{c}(w^i : \texttt{forall}\ \overline{a^j : \kappa}\ \texttt{.}\ \overline{\phi'}\ \texttt{=>}\ \phi) \in \Xi \\ \overline{a^j} \cap dom(\Gamma) = \emptyset \\ \Gamma \vdash \overline{\tau : \kappa} \\ \Gamma \mid \Xi \vdash^e \overline{[\overline{a^j \mapsto \tau}]\ \phi'}\end{array}}{\Gamma \mid \Xi \vdash^e [\overline{a^j \mapsto \tau}]\ \phi}$$

Figure 10: Constraint entailment

$$\boxed{\Gamma \mid \Xi \vdash \sigma \leq \sigma'}$$

$$\frac{}{\Gamma \mid \Xi \vdash \tau \leq \tau}$$

$$\frac{\begin{array}{c}\Gamma \mid \Xi \vdash \sigma'_1 \leq \sigma_1 \\ \Gamma \mid \Xi \vdash \sigma_2 \leq \sigma'_2\end{array}}{\Gamma \mid \Xi \vdash (\sigma_1\ \texttt{->}\ \sigma_2) \leq (\sigma'_1\ \texttt{->}\ \sigma'_2)}$$

$$\frac{\begin{array}{c}\overline{a^i} \cap dom(\Gamma) = \emptyset \\ \Gamma \vdash \overline{\tau : \kappa} \\ \Gamma \mid \Xi \vdash [\overline{a^i \mapsto \tau}]\ \sigma \leq \sigma' \\ \Xi \vdash^e \overline{[\overline{a^i \mapsto \tau}]\ \phi}\end{array}}{\Gamma \mid \Xi \vdash \texttt{forall}\ \overline{a^i : \kappa}\ \texttt{.}\ \overline{\phi}\ \texttt{=>}\ \sigma \leq \sigma'}$$

$$\frac{\begin{array}{c}\overline{a^i} \cap dom(\Gamma) = \emptyset \\ \Xi' = named(\overline{\phi}) \\ \Gamma \mathbin{+\!\!+} \overline{a^i : \kappa} \mid \Xi \mathbin{+\!\!+} \Xi' \vdash \sigma \leq \sigma'\end{array}}{\Gamma \mid \Xi \vdash \texttt{exists}\ \overline{a^i : \kappa}\ \texttt{.}\ \overline{\phi}\ \texttt{=>}\ \sigma \leq \sigma'}$$

$$\frac{\begin{array}{c}\overline{a^i} \cap dom(\Gamma) = \emptyset \\ \Xi' = named(\overline{\phi}) \\ \Gamma \mathbin{+\!\!+} \overline{a^i : \kappa} \mid \Xi \mathbin{+\!\!+} \Xi' \vdash \sigma \leq \sigma'\end{array}}{\Gamma \mid \Xi \vdash \sigma \leq \texttt{forall}\ \overline{a^i : \kappa}\ \texttt{.}\ \overline{\phi}\ \texttt{=>}\ \sigma'}$$

$$\frac{\begin{array}{c}\overline{a^i} \cap dom(\Gamma) = \emptyset \\ \Gamma \vdash \overline{\tau : \kappa} \\ \Gamma \mid \Xi \vdash \sigma \leq [\overline{a^i \mapsto \tau}]\ \sigma' \\ \Xi \vdash^e \overline{[\overline{a^i \mapsto \tau}]\ \phi}\end{array}}{\Gamma \mid \Xi \vdash \sigma \leq \texttt{exists}\ \overline{a^i : \kappa}\ \texttt{.}\ \overline{\phi}\ \texttt{=>}\ \sigma'}$$

Figure 11: Type subsumption

class constructors to types denoting a particular record or class. This is necessary because record and class constructors within terms elide all type arguments. (Of course our type inference system infers these types.)

Figures 13 and 14 present the type checking rules. In common with [12] we must generalise the type of nearly every sub-term, in contrast to Hindley/Damas/Milner-based systems in which only let-bound terms need be generalised. This is accomplished using the $\vdash_G$ judgement form. Each binding construct shadows any variables in scope with the same name. In rules RECORD and NEWREC, we use the function *unstamp* to reset to 0 the stamps of the context corresponding to the record declaration body before using it to extend the current context. This is because the alpha-conversion of record bodies during kind checking is not reflected within terms, which continue to use names with a default stamp of 0. Since all types within the type context are absolute, and we prevent the shadowing of top-level module names, it is sound to shadow type declarations in this simple-minded way. In rule PROJECT, we use the external signature of the projected variable to determine the expression's type.

Record bodies are type checked by the rules of Figure 15. We assume record bodies are permuted to match their record

$$\boxed{\Gamma \vdash P \hookrightarrow \tau}$$

$$\frac{\begin{array}{c}\texttt{record}\ \{\tau\texttt{\^{}}\}^{opt}\ A^i\ \overline{a^j : \kappa}\ \texttt{= \{}\ S\ \texttt{\}} \in \Gamma \\ \Gamma \vdash \overline{v : \kappa}\end{array}}{\Gamma \vdash A^i \hookrightarrow \{\tau\texttt{\^{}}\}^{opt}\ A^i\ \overline{v}}$$

$$\frac{\begin{array}{c}\Gamma \vdash P \hookrightarrow \tau \\ \Gamma \vdash \tau : \langle \theta \mid \Delta \mid S \rangle \\ \texttt{record}\ A^i\ \overline{a^j : \kappa}\ \texttt{= \{}\ S'\ \texttt{\}} \in S \\ \Gamma \vdash \overline{v : \kappa}\end{array}}{\Gamma \vdash P\texttt{\^{}}A \hookrightarrow \tau\texttt{\^{}}A\ \overline{v}}$$

$$\boxed{\Gamma \vdash Q \hookrightarrow \sigma}$$

$$\frac{\texttt{class}\ \overline{\phi}\ \texttt{=>}\ \{\tau\texttt{\^{}}\}^{opt}\ C^i\ \overline{a^j : \kappa}\ \texttt{where}\ \tau \in \Gamma}{\Gamma \vdash C^i \hookrightarrow \texttt{forall}\ \overline{a^j : \kappa}\ \texttt{.}\ \{\tau\texttt{\^{}}\}^{opt}\ C^i\ \overline{a}\ \texttt{=>}\ \tau}$$

$$\frac{\begin{array}{c}\overline{b^j} \cap dom(\Gamma) = \emptyset \\ \Gamma +\!\!+ \overline{b^j : \kappa'} \vdash P \hookrightarrow \tau \\ \Gamma \vdash \tau : \langle \theta \mid \Delta \mid S \rangle \\ \texttt{class}\ \overline{\phi}\ \texttt{=>}\ C^i\ \overline{a^j : \kappa}\ \texttt{where}\ \tau \in S \\ \overline{a^j} \cap \overline{b^j} = \emptyset\end{array}}{\Gamma \vdash P\texttt{\^{}}C \hookrightarrow \texttt{forall}\ \overline{a^j : \kappa} +\!\!+ \overline{b^j : \kappa'}\ \texttt{.}\ \tau\texttt{\^{}}C\ \overline{a^j}\ \texttt{=>}\ \tau}$$

Figure 12: Expanding record and class constructors

declaration bodies. Checking instance bindings against instance declarations is somewhat involved, as we must ensure not only that the supplied term is of the appropriate type, but also that the superclass context of the instance declaration's class may also be satisfied. The later check requires the instance declaration under construction to be temporarily removed from the known constraint context. This is the reason we used *compname* to form the composite name of the instance declaration in Figure 8.

For example, assume we have the class declaration:

```
class B a => C a where a -> Int
```

the instance declaration:

```
instance w :: forall a . A a => C a
```

and the instance binding:

```
instance w = t
```

Furthermore, assume t has type:

```
forall a b . A a => a -> b
```

We must now check that (i) t matches its instance declaration, that (ii) its type is compatible with that given by its corresponding class declaration, and (iii) that the superclass constraints of the class are satisfiable. Let Ξ be the known constraint context. With a little cunning, its possible to

$$\boxed{\Gamma \mid \Xi \vdash_G t : \sigma}$$

$$\frac{\begin{array}{c}dom(\Delta) \cap dom(\Gamma) = \emptyset \\ \Gamma +\!\!+ \Delta \mid \overline{\phi\ \mathsf{constraint}} \\ \Xi' = named(\overline{\phi}) \\ \Gamma +\!\!+ \Delta \mid \Xi +\!\!+ \Xi' \vdash t : \sigma\end{array}}{\Gamma \mid \Xi \vdash_G t : \texttt{forall}\ \Delta\ \texttt{.}\ \overline{\phi}\ \texttt{=>}\ \sigma}\ \text{GEN}$$

$$\boxed{\Gamma \mid \Xi \vdash t : \sigma \hookrightarrow T}$$

$$\frac{}{\Gamma \mid \Xi \vdash i : \texttt{Int}}\ \text{INT}$$

$$\frac{(x^i : \sigma) \in \Gamma}{\Gamma \mid \Xi \vdash x^i : \sigma}\ \text{VAR}$$

$$\frac{\begin{array}{c}\Gamma \vdash \tau : \texttt{Type} \\ \Gamma_{\backslash x^i}, x^i : \tau \mid \Xi \vdash_G t : \sigma\end{array}}{\Gamma \mid \Xi \vdash \texttt{\textbackslash}x^i\ \texttt{.}\ t : (\tau\ \texttt{->}\ \sigma)}\ \text{ABS}$$

$$\frac{\begin{array}{c}\Gamma \vdash \sigma'\ \mathsf{scheme} \\ \Gamma_{\backslash x^i}, x^i : \sigma' \mid \Xi \vdash_G t : \sigma\end{array}}{\Gamma \mid \Xi \vdash \texttt{\textbackslash}x^i \texttt{::} \sigma'\ \texttt{.}\ t : (\sigma'\ \texttt{->}\ \sigma)}\ \text{AABS}$$

$$\frac{\begin{array}{c}\Gamma \mid \Xi \vdash_G u : \sigma_1 \\ \Gamma \mid \Xi \vdash_G t : \sigma_2 \\ \Gamma \mid \Xi \vdash \sigma_2 \leq (\sigma_3\ \texttt{->}\ \sigma_4) \\ \Gamma \mid \Xi \vdash \sigma_1 \leq \sigma_3\end{array}}{\Gamma \mid \Xi \vdash t\ u : \sigma_4}\ \text{APP}$$

$$\frac{\begin{array}{c}\Gamma \mid \Xi \vdash_G \overline{t : \sigma} \\ \Gamma \mid \Xi \vdash \overline{\sigma \leq \tau}\end{array}}{\Gamma \mid \Xi \vdash \texttt{(}\ \overline{t}\ \texttt{)} : \texttt{(}\ \overline{\tau}\ \texttt{)}}\ \text{TUP}$$

$$\frac{\begin{array}{c}\Gamma \mid \Xi \vdash_G u : \sigma' \\ \Gamma \mid \Xi \vdash \sigma' \leq \texttt{(}\ \overline{v}\ \texttt{)} \\ \Gamma_{\backslash \overline{x^i}} +\!\!+ \overline{x^i : v} \mid \Xi \vdash t : \sigma\end{array}}{\Gamma \mid \Xi \vdash \texttt{let (}\ \overline{x^i}\ \texttt{) =}\ u\ \texttt{in}\ t : \sigma}\ \text{LETTUP}$$

$$\frac{\begin{array}{c}\Gamma \mid \Xi \vdash_G u : \sigma' \\ \Gamma_{\backslash x^i}, x^i : \sigma' \mid \Xi \vdash t : \sigma\end{array}}{\Gamma \mid \Xi \vdash \texttt{let}\ x^i\ \texttt{=}\ u\ \texttt{in}\ t : \sigma}\ \text{LET}$$

$$\frac{\begin{array}{c}\Gamma \vdash \sigma_1 \hookrightarrow \sigma_2\ \mathsf{scheme} \\ \Gamma \mid \Xi \vdash_G u : \sigma_3 \\ \Gamma \mid \Xi \vdash \sigma_3 \leq \sigma_2 \\ \Gamma_{\backslash x^i}, x^i : \sigma_2 \mid \Xi \vdash t : \sigma_4\end{array}}{\Gamma \mid \Xi \vdash \texttt{let}\ x^i\ \texttt{::}\ \sigma_1\ \texttt{=}\ u\ \texttt{in}\ t : \sigma_4}\ \text{ALET}$$

Figure 13: Well-typed terms (part 1 of 2)

accomplish all three of these tests by checking:

$$\begin{array}{ll}\Xi_{\backslash \texttt{w}} \vdash & \texttt{forall a b . (A a, B a) => a -> b} \leq \\ & \texttt{forall a . A a => a -> Int}\end{array}$$

Notice that the constraint `B a` was added to the type scheme for t so as to check (iii). This subsumption test succeeds if Ξ contains a constraint of the form:

```
w'' :: forall a . B a
```

If we didn't remove `w` from Ξ, the above subsumption test would also succeed if Ξ contains a constraint of the form:

```
w' :: forall a . A a
```

This is because:

$$\mathtt{w' :: forall\ a\ .\ A\ a}, \mathtt{w :: forall\ a\ .\ A\ a => C\ a} \vdash^{e} \mathtt{B\ a}$$

To avoid this circularity problem, we must temporarily remove the instance declaration we are trying to type check from the known constraint context.

Finally, Figure 16 shows how to type check an entire program. We first split the program into a record type declaration body and a single term. The record declaration body is then skolemized, kind checked, and the result used to construct the initial type context. The term is then type checked assuming that, for each interface A, a variable A of type A is in scope. Since the term must correspond to an implementation, we require it has the appropriate record type.

For brevity we have not formalized three aspects of the system. Firstly, we must detect cyclic types such as:

```
x :: x!
```

This is easily achieved by keeping track of how many times each variable has been dereferenced while rewriting a given type. Secondly, we must prevent top-level interface names from being shadowed, and must disallow the explicit construction of a record using a top-level interface constructor:

```
module A where
record A = { ... }     -- illegal
a = A { ... }          -- illegal
```

Finally, we have not formalized abstract top-level types. These are handled by augmenting the interface with the local type declarations of the implementation. Abstract top-level types named in the interface are simply overwritten by their concrete declaration given in the implementation. The only subtlety is checking the two type declarations have equal kinds.

Type inference for our system has been formalized as a Haskell program, available from `http://www.cse.ogi.edu/~mbs`. This is easily transliterated into a more familiar presentation as a set of inductive relations.

$$\frac{\begin{array}{c} \Gamma \mid \Xi \vdash_G u : \mathtt{exists}\ \Delta\ .\ \overline{\phi}\ \mathtt{=>}\ \sigma' \\ dom(\Delta) \cap dom(\Gamma) = \emptyset \\ \Xi' = named(\overline{\phi}) \\ \Gamma_{\backslash x^i} + \!\!+\ \Delta, x^i : \sigma' \mid \Xi + \!\!+\ \Xi' \vdash t : \sigma \\ \Gamma \vdash \sigma\ \mathsf{scheme} \end{array}}{\Gamma \mid \Xi \vdash \mathtt{let\ open}\ x^i\ \mathtt{=}\ u\ \mathtt{in}\ t : \sigma}\ \textsc{OpenLet}$$

$$\frac{\begin{array}{c} \sigma_1 = \mathtt{exists}\ \Delta\ .\ \overline{\phi}\ \mathtt{=>}\ \sigma_2 \\ dom(\Delta) \cap dom(\Gamma) = \emptyset \\ \Gamma + \!\!+\ \Delta \vdash \overline{\phi \hookrightarrow \phi'\ \mathsf{constraint}} \\ \Gamma + \!\!+\ \Delta \vdash \sigma_2 \hookrightarrow \sigma_3\ \mathsf{scheme} \\ \Gamma \mid \Xi \vdash_G u : \sigma_4 \\ \Gamma \mid \Xi \vdash \sigma_4 \leq \mathtt{exists}\ \Delta\ .\ \overline{\phi'}\ \mathtt{=>}\ \sigma_3 \\ \Xi' = named(\overline{\phi'}) \\ \Gamma_{\backslash x^i} + \!\!+\ \Delta, x^i : \sigma_3 \mid \Xi + \!\!+\ \Xi' \vdash t : \sigma_5 \\ \Gamma \vdash \sigma_5\ \mathsf{scheme} \end{array}}{\Gamma \mid \Xi \vdash \mathtt{let\ open}\ x^i\ \mathtt{::}\ \sigma_1\ \mathtt{=}\ u\ \mathtt{in}\ t : \sigma_5}\ \textsc{AOpenLet}$$

$$\frac{\begin{array}{c} \Gamma \vdash \sigma_1 \hookrightarrow \sigma_2\ \mathsf{scheme} \\ \Gamma \mid \Xi \vdash_G t : \sigma_3 \\ \Gamma \mid \Xi \vdash \sigma_3 \leq \sigma_2 \end{array}}{\Gamma \mid \Xi \vdash \mathtt{(}t\ \mathtt{::}\ \sigma_1\mathtt{)} : \sigma_2}\ \textsc{Annot}$$

$$\frac{\begin{array}{c} \overline{a} \cap dom(\Gamma) = \emptyset \\ \Gamma \vdash \overline{v : \kappa} \\ \Gamma \mid \Xi \vdash [\overline{a^i \mapsto v}]\ t : \sigma \end{array}}{\Gamma \mid \Xi \vdash \mathtt{free}\ \overline{a^i : \kappa}\ \mathtt{in}\ t : \sigma}\ \textsc{Free}$$

$$\frac{\begin{array}{c} \Gamma \vdash P \hookrightarrow \tau \\ \Gamma \vdash \tau : \langle \theta \mid \Delta \mid S \rangle \\ \tau \vdash S : \langle \Gamma' \mid \epsilon \rangle \\ \Gamma'' = unstamp(\Gamma') \\ \tau \mid \Gamma_{\backslash dom(\Gamma'')} + \!\!+\ \Gamma'' \mid \Xi \vdash s : S \end{array}}{\Gamma \mid \Xi \vdash P\ \mathtt{\{}\ s\ \mathtt{\}} : \tau}\ \textsc{Record}$$

$$\frac{\begin{array}{c} \Gamma \mid \Xi \vdash_G t : \sigma_1 \\ \Gamma \mid \Xi \vdash \sigma_1 \leq \tau \\ \Gamma \vdash \tau : \langle \theta \mid \Delta \mid S \rangle \\ (\{\mathtt{open}\}^{opt}\ x^i\ \mathtt{::}\ \sigma_2\ \{\overline{\phi}\ \mathtt{=>}\ \sigma_3\}) \in S \end{array}}{\Gamma \mid \Xi \vdash t\mathtt{.}x : \sigma_3}\ \textsc{Project}$$

$$\frac{\Gamma \vdash Q \hookrightarrow \sigma}{\Gamma \mid \Xi \vdash \mathtt{?}Q : \sigma}\ \textsc{Class}$$

$$\frac{\begin{array}{c} A^i \notin dom(\Gamma) \\ dom(\Delta) \cap dom(\Gamma) = \emptyset \\ \Gamma + \!\!+\ \Delta \vdash S \hookrightarrow \langle \epsilon \mid S' \rangle\ \mathsf{skolem} \\ \cdot \vdash S' : \langle \Gamma' \mid \epsilon \rangle \\ \Gamma'' = unstamp(\Gamma') \\ \cdot \mid \Gamma_{\backslash dom(\Gamma'')} + \!\!+\ \Delta + \!\!+\ \Gamma'' \vdash S' \hookrightarrow S''\ \mathsf{record} \\ \Gamma, \mathtt{record}\ A^i\ \Delta\ \mathtt{=\ \{}\ S''\ \mathtt{\}} \mid \Xi \vdash t : \sigma \\ \Gamma \vdash \sigma\ \mathsf{scheme} \end{array}}{\Gamma \mid \Xi \vdash \mathtt{record}\ A^i\ \Delta\ \mathtt{=\ \{}\ S\ \mathtt{\}\ in\ t} : \sigma}\ \textsc{NewRec}$$

Figure 14: Well-typed terms (part 2 of 2)

$$\boxed{\{\tau\}^{opt} \mid \Gamma \mid \Xi \vdash s : S}$$

$$\frac{\{\tau\}^{opt} \mid \Gamma \mid \Xi \vdash s : S}{\{\tau\}^{opt} \mid \Gamma \mid \Xi \vdash s : (\texttt{record}\ A^i\ \Delta\ \texttt{= \{}\ S'\ \texttt{\}}\ ; S)}$$

$$\frac{\{\tau\}^{opt} \mid \Gamma \mid \Xi \vdash s : S}{\{\tau\}^{opt} \mid \Gamma \mid \Xi \vdash s : (\texttt{class}\ \overline{\phi}\ \texttt{=>}\ C^i\ \Delta\ \texttt{where}\ \tau\ ; S)}$$

$$\frac{\begin{array}{c} \Gamma \mid \Xi \vdash_G t : \sigma_3 \\ \Gamma \mid \Xi \vdash \sigma_3 \leq \sigma_1 \\ \{\tau\}^{opt} \mid \Gamma \mid \Xi \vdash s : S \end{array}}{\begin{array}{c} \{\tau\}^{opt} \mid \Gamma \mid \Xi \vdash (\{\texttt{open}\}^{opt}\ x^i\ \texttt{=}\ t\ ; s) : \\ (\{\texttt{open}\}^{opt}\ x^i\ \texttt{::}\ \sigma_1\ \texttt{\{}\overline{\phi}\ \texttt{=>}\ \sigma_2\texttt{\}}\ ; S) \end{array}}$$

$$\frac{\begin{array}{c} \overline{a^j} \cap dom(\Gamma) = \emptyset \\ \Gamma \mathbin{+\!\!+} \overline{a^j : \kappa} \vdash \phi : \langle \overline{\phi''} \mid \Delta \mid \tau' \rangle \\ \sigma_1 = \texttt{forall}\ \overline{a^j : \kappa}\ \texttt{.}\ \overline{\phi'}\ \texttt{=>}\ \tau' \\ \Gamma \mid \Xi \vdash_G t : \sigma_2 \\ \sigma_2 = \texttt{forall}\ \overline{b^j : \kappa'}\ \texttt{.}\ \overline{\phi'''}\ \texttt{=>}\ \tau'' \\ \overline{b^j} \cap dom(\Gamma) = \emptyset \\ \Gamma \mathbin{+\!\!+} \overline{b^j : \kappa'} \vdash \overline{\upsilon : \kappa} \\ \Gamma \mathbin{+\!\!+} \overline{b^j : \kappa'} \vdash \overline{\upsilon' : \kappa} \\ [\overline{a^j \mapsto \upsilon}]\ \tau' = [\overline{b^j \mapsto \upsilon'}]\ \tau'' \\ \sigma_4 = \texttt{forall}\ \overline{b^j : \kappa'}\ \texttt{.}\ \overline{\phi'''} \mathbin{+\!\!+} [\overline{a^j \mapsto \upsilon}]\ \overline{\phi''}\ \texttt{=>}\ \tau'' \\ \Gamma \mid \Xi_{\backslash compname(\{\tau\}^{opt}, w^i)} \vdash \sigma_4 \leq \sigma_1 \\ \{\tau\}^{opt} \mid \Gamma \mid \Xi \vdash s : S \end{array}}{\begin{array}{c} \{\tau\}^{opt} \mid \Gamma \mid \Xi \vdash (\texttt{instance}\ w^i\ \texttt{=}\ t\ ; s) : \\ (\texttt{instance}\ w^i\ \texttt{::}\ \texttt{forall}\ \overline{a^j : \kappa}\ \texttt{.}\ \overline{\phi'}\ \texttt{=>}\ \phi\ ; S) \end{array}}$$

$$\frac{}{\{\tau\}^{opt} \mid \Gamma \mid \Xi \vdash \epsilon : \epsilon}$$

Figure 15: Well-typed record bodies

$$\boxed{\vdash prog\ \textsf{toplevel}}$$

$$\frac{\begin{array}{c} \epsilon \vdash \overline{\texttt{record}\ A^i\ \texttt{= \{}\ S\ \texttt{\}}} \hookrightarrow \langle \Delta \mid S' \rangle\ \textsf{skolem} \\ \cdot \vdash S' : \langle \Gamma \mid \_ \rangle \\ \cdot \mid \Delta \mathbin{+\!\!+} \Gamma \vdash S' \hookrightarrow S''\ \textsf{record} \\ \cdot \vdash S'' : \langle \Gamma' \mid \Xi \rangle \\ \Delta \mathbin{+\!\!+} \Gamma' \mathbin{+\!\!+} \overline{A^i : A^i} \mid \Xi \vdash_G B^j\ \texttt{\{}\ s\ \texttt{\}} : B^j \end{array}}{\vdash \overline{\texttt{record}\ A^i\ \texttt{= \{}\ S\ \texttt{\}}}\ ; B^j\ \texttt{\{}\ s\ \texttt{\}}\ \textsf{toplevel}}$$

Figure 16: Well-formed programs