An Existential Crisis Resolved

Type inference for first-class existential types

RICHARD A. EISENBERG, Tweag, France
GUILLAUME DUBOC, ENS Lyon, France and Tweag, France
STEPHANIE WEIRICH, University of Pennsylvania, USA
DANIEL LEE, University of Pennsylvania, USA

Despite the great success of inferring and programming with universal types, their dual—existential types—are much harder to work with. Existential types are useful in building abstract types, working with indexed types, and providing first-class support for refinement types. This paper, set in the context of Haskell, presents a bidirectional type-inference algorithm that infers where to introduce and eliminate existentials without any annotations in terms, along with an explicitly typed, type-safe core language usable as a compilation target. This approach is backward compatible. The key ingredient is to use *strong* existentials, which support (lazily) projecting out the encapsulated data, not weak existentials accessible only by pattern-matching.

Additional Key Words and Phrases: existential types, type inference, Haskell

1 INTRODUCTION

Parametric polymorphism through the use of universally quantified type variables is pervasive in functional programming. Given its overloaded numbers, a beginning Haskell programmer literally cannot ask for the type of $1 + 1$ without seeing a universally quantified type variable.

However, universal quantification has a dual: existentials. While universals claim the spotlight, with support for automatic elimination (that is, instantiation) in all non-toy typed functional languages we know and automatic introduction (frequently, **let**-generalization) in some, existentials are underserved and impoverished. In every functional language we know, both elimination and introduction must be done explicitly every time, and languages otherwise renowned for their type inference—such as Haskell—require that users define a new top-level datatype for every existential.

While not as widely useful as universals, existential quantification comes up frequently in richly typed programming. Further examples are in Section 2, but consider writing a *dropWhile* function on everyone's favorite example datatype, the length-indexed vector:

```
-- dropWhile predicate vec drops the longest prefix of vec such that all elements in the prefix
-- satisfy predicate. In this type, n is the vector's length, while a is the type of elements.
dropWhile :: (a → Bool) → Vec n a → Vec ??? a
```

How can we fill in the question marks? Without knowing the contents of the vector and the predicate we are passing, we cannot know the length of the output. Furthermore, returning an ordinary, unindexed list would requiring copying a suffix of the input vector, an unacceptable performance degradation.

Existentials come to our rescue: $\mathit{dropWhile} :: (a \to \mathit{Bool}) \to \mathit{Vec}\ n\ a \to \exists m.\ \mathit{Vec}\ m\ a$. Though this example can be written today in a number of languages, all require annotations in terms both to pack (introduce) the existential and unpack (eliminate) it through the application or pattern-matching of a data constructor.

This paper describes a type-inference algorithm that supports implicit introduction and elimination of existentials, with a concrete setting in Haskell. We offer the following contributions:

- Section 4 presents our type-inference algorithm, the primary contribution of this paper. The algorithm is a small extension to an algorithm that accepts a Hindley-Milner language; our

2021. 2475-1421/2021/8-ART64
https://doi.org/10.1145/3473569

language, $\mathbb{X}$, is thus a superset of Hindley-Milner (Theorem 7.3). In addition, it supports several *stability properties* [Bottu and Eisenberg 2021]; a language is *stable* if small, seemingly innocuous changes to the input program (such as **let**-inlining) do not cause a change in the type or acceptability of a program (Theorems 7.4–7.6). Our algorithm is easily integrable with the latest inference algorithm [Serrano et al. 2020] in the Glasgow Haskell Compiler (GHC) (Section 8).

- Section 5 presents a core language based on System F, $\mathbb{FX}$, that is a suitable target of compilation (Section 6) for $\mathbb{X}$. We prove $\mathbb{FX}$ is type-safe (Theorems 5.1 and 5.2) and supports type erasure (Theorem 5.3). It is designed in a way that is compatible with the existing System FC [Sulzmann et al. 2007] language used internally within GHC. All programs accepted by our algorithm elaborate to well-typed programs in $\mathbb{FX}$ (Theorem 7.1). In addition, elaboration preserves the semantics of the source program, as we can observe by examining the result of type erasure (Theorem 7.2).

We normally desire type-inference algorithms to come with a declarative specification, where automatic introduction and elimination of quantifiers can happen anywhere, in the style of the Hindley-Milner type system [Hindley 1969; Milner 1978]. These specifications come alongside syntax-directed algorithms that are sound and complete with respect to the specification [Clément et al. 1986; Damas and Milner 1982]. However, we do not believe such a system is possible with existentials; while negative results are hard to prove conclusively, we lay out our arguments against this approach in Section 9.1. Instead, we present just our algorithm, though we avoid the complication and distraction of unification variables by allowing our algorithm to non-deterministically guess monotypes τ in the style of a declarative specification.

There is a good deal of literature in this area; much of it is focused on module systems, which often wish to hide the nature of a type using an existential package. We review some important prior work in Section 10.

The concrete examples in this paper are set in Haskell, but the fundamental ideas in our inference algorithm are fully portable to other settings, including in languages without **let**-generalization.

2 MOTIVATION AND EXAMPLES

Though not as prevalent as examples showing the benefits of universal polymorphism, easy existential polymorphism smooths out some of the wrinkles currently inherent in programming with indexed types such as GADTs [Xi et al. 2003].

2.1 Unknown Output Indices

We first return to the example from the introduction, writing an operation that drops an indeterminate number of elements from a length-indexed vector:

```
data Nat = Zero | Succ Nat
type Vec :: Nat → Type → Type   -- -XStandaloneKindSignatures, new in GHC 8.10
data Vec n a where
  Nil  :: Vec Zero a
  (:>) :: a → Vec n a → Vec (Succ n) a
infixr 5 :>
```

In today's Haskell, the way to write *dropWhile* over vectors is like this:

```
type ExVec :: Type → Type
data ExVec a where
  MkEV :: ∀(n :: Nat) (a :: Type). Vec n a → ExVec a
```

```
filter :: (a → Bool) → Vec n a → ExVec a
filter _ Nil                    = MkEV Nil
filter p (x :> xs) | p x
                   , MkEV v ← filter p xs
                                = MkEV (x :> v)
                   | otherwise = filter p xs
```

(a)

```
filter :: (a → Bool) → Vec n a → ∃m. Vec m a
filter _ Nil                    = Nil
filter p (x :> xs) | p x        = x :> filter p xs
                   | otherwise  = filter p xs
```

(b)

Fig. 1. Implementations of *filter* over vectors (a) in today's Haskell, and (b) with our extensions

```
dropWhile :: (a → Bool) → Vec n a → ExVec a
dropWhile _ Nil                    = MkEV Nil
dropWhile p (x :> xs) | p x        = dropWhile p xs
                      | otherwise  = MkEV (x :> xs)
```

However, with our inference of existential introduction and elimination, we can simplify to this:

```
dropWhile :: (a → Bool) → Vec n a → ∃m. Vec m a
dropWhile _ Nil                    = Nil
dropWhile p (x :> xs) | p x        = dropWhile p xs
                      | otherwise  = x :> xs
```

There are two key differences: we no longer need to define the *ExVec* type, instead using $\exists m.\ \mathit{Vec}\ m\ a$; and we can omit any notion of packing in the body of *dropWhile*. Similarly, clients of *dropWhile* would not need to unpack the result, allowing the result of *dropWhile* to be immediately consumed by a *map*, for example.

2.2 Increased Laziness

Another function that produces an output of indeterminate length is *filter*. It is enlightening to compare the implementation of *filter* using today's existentials and the version possible with our new ideas; see Figure 1.

Beyond just the change to the types and the disappearance of terms to pack and unpack existentials, we can observe that the *laziness* of the function has changed. (See Aside 1 for why we cannot easily make **unpack** bind lazily.) In Figure 1(a), we see that the recursive call to *filter* must be made before the use of the cons operator :>. This means that, say, computing *take* 2 (*filter p vec*) (assuming *take* is clever enough to expect an *ExVec*) requires computing the result of the entire *filter*, even though the analogous expression on lists would only requiring filtering enough of *vec* to get the first two elements that satisfy *p*. The implementation of *filter* also requires enough stack space to store all the recursive calls, requiring an amount of space linear in the length of the input vector.

By contrast, the implementation in Figure 1(b) is lazy in the tail of the vector. Computing *take* 2 (*filter p vec*) really would only process enough elements of *vec* to find the first 2 that satisfy *p*. In addition, the computation requires only constant stack space, because *filter* will immediately return a cons cell storing a thunk for filtering the tail. If a bounded number of elements satisfy *p*, this is an asymptotic improvement in space requirements.

We can support the behavior evident in Figure 1(b) only because we use *strong* existential packages, where the existentially packed type can be projected out from the existential package, What if **unpack** were simply lazy? The problem is that this is not simple! A straightforward typed operational semantics would not suffice, because there is no way to, say, reduce an **unpack** into a substitution (the way we would handle a lazy **let**). We could imagine an untyped operational semantics that did not require **unpack** to evaluate the existential package, binding its variable with a lazy binding. Without types, though, we would be unable to prove safety. In order to keep a typed operational semantics with a lazy **unpack**, we must model a set of heap bindings and an evaluation stack in our semantics. While this is possible, such an operational semantics is unsuitable for a (dependently typed) language where we also might wish to evaluate in types, which is our eventual goal for Haskell. The claim here is not that a lazy **unpack** is impossible, but that it is not obviously superior to the approach we advocate for here.

Relatedly, one could wonder whether we should just use a lazy Haskell pattern in *filter*. Alas, Haskell does not allow a lazy pattern to bind existential variables: writing $\sim\!(MkEV\ v) \leftarrow filter\ p\ xs$ in Figure 1(a) would cause a compile-time error. This restriction in today's Haskell is not incidental, because the internal language would require exactly the power of the **open** approach we propose here in order to support such a lazy pattern.

Aside 1. Why lazy **unpack** is no easy answer

instead of relying on the use of a pattern-match. Furthermore, projection of the packed type is requires no evaluation of any expression. We return to explain more about this key innovation in Section 3.

2.3 Object Encoding

Suppose we have a pretty-printer feature in our application, making use of the following class:

```
class Pretty a where
  pretty :: a → Doc
```

There are *Pretty* instances defined for all relevant types. Now, suppose we have $order :: Order$, $client :: Client$, and $status :: OrderStatus$; we wish to create a message concatenating these three details. Today, we might say $vcat\ [pretty\ order, pretty\ client, pretty\ status]$, where $vcat :: [Doc] \to Doc$. However, equipped with lightweight existentials, we could instead write $vcat\ [order, client, status]$, where $vcat :: [\exists a.\ Pretty\ a \wedge a] \to Doc$. Here, the $\wedge$ type constructor allows us to pack a witness for a constraint (such as a type class dictionary [Hall et al. 1996]) inside an existential package. Each element of the list is checked against the type $\exists a.\ Pretty\ a \wedge a$. Choosing one, checking *order* against $\exists a.\ Pretty\ a \wedge a$ uses unification to determine that the choice of a should be *Order*, and we will then need to satisfy a *Pretty Order* constraint. In the implementation of *vcat*, elements of type $\exists a.\ Pretty\ a \wedge a$ will be available as arguments to *pretty*:

```
vcat :: [∃a. Pretty a ∧ a] → Doc
vcat []       = empty
vcat (x : xs) = pretty x $$ vcat xs
```

While the code simplification at call sites is modest, the ability to abstract over a constraint in forming a list makes it easier to avoid the types from preventing users from expressing their thoughts more directly.

Our main formal presentation in this paper does not include the packed constraints required here, but Section 9.2 considers an extension to our work that would support this example.

2.4 Richly Typed Data Structures

Suppose we wish to design a datatype whose inhabitants meet certain invariants by construction. If the invariants are complex enough, this can be done only by designing the datatype as a generalized algebraic datatype (GADT) [Xi et al. 2003]. Though other examples in this space abound (for example, encoding binary trees [McBride 2014] and regular expressions [Weirich 2018]), we will use the idea of a well-typed expression language, perhaps familiar to our readers.[1]

The idea is encapsulated in these definitions:

```
data Ty = Ty :→ Ty | ...    -- base types elided
type Exp :: [ Ty ]   -- types of in-scope variables
          → Ty       -- type of expression
          → Type
data Exp ctx ty where
  App :: Exp ctx (arg :→ result) → Exp ctx arg → Exp ctx result
  ...
```

An expression of type *Exp ctx ty* is guaranteed to be well-typed in our object language: note that a function application requires the function to have a function type *arg* :⟶ *result* and the argument to have type *arg*. (The *ctx* is a list of the types of in-scope variables; using de Bruijn indices means we do not need to map names.) We are thus unable to represent the syntax tree applying, say, the number 5 to an argument *True*.

However, if we are to use *Exp* in a running interpreter, we have a problem: users might not type well-typed expressions. How can we take a user-written program and represent it in *Exp*? We must type-check it.

Assuming a type *UExp* ("unchecked expression") that is like *Exp* but without its indices, we would write the following:[2]

```
typecheck :: (ctx :: [ Ty ]) → UExp → Maybe (∃ty. Exp ctx ty)
typecheck ctx (UApp fun arg) = do     -- using the Maybe monad
  fun' ← typecheck ctx fun
  arg' ← typecheck ctx arg
      -- decompose the type of fun' into expectedArgTy :→ _resultTy:
  (expectedArgTy, _resultTy) ← checkFunctionTy (typeOf fun')
      -- Check whether expectedArgTy and the type of arg' are the same (failing if not)
      -- Refl is a proof the types coincide; matching on it reveals this fact to the type-checker:
  Refl ← checkEqual expectedArgTy (typeOf arg')
  return (App fun' arg')
```

The use of an existential type is critical here. There is no way to know what the type of an expression is before checking it, and yet we need this type available for compile-time reasoning to be able

[1] This well-worn idea perhaps originates in a paper by Pfenning and Lee [1989], though that paper does not use an indexed datatype. Augustsson and Carlsson [1999] extend the idea to use a datatype, much as we have done here. A more in-depth treatment of this example is the subject of a functional pearl by Eisenberg [2020].

[2] This rendering of the example assumes the ability to write using dependent types, to avoid clutter. However, do not be distracted: the dependent types could easily be encoded using singletons [Eisenberg and Weirich 2012; Monnier and Haguenauer 2010], while we focus here on the use of existential types.

to accept the final use of *App*. An example such as this one can be written today, but with extra awkward packing and unpacking of existentials, or through the use of a continuation-passing encoding. With the use of lightweight existentials, an example like this is easier to write, lowering the barrier to writing richly typed, finely specified programs.

3 KEY IDEA: EXISTENTIAL PROJECTIONS

In our envisioned source language, introduction and elimination of existential types are implicit. Precise locations are determined by type inference (as pinned down in Section 4)—accordingly, these locations may be hard to predict. Once these locations have been identified, the compiler must produce a fully annotated, typed core language that makes these introductions and eliminations explicit. We provide a precise account of this core language in Section 5. But before we do that, we use this section to informally justify why we need new forms in the first place. Why can we no longer use the existing encoding of existential types (based on Mitchell and Plotkin [1988] and Läufer [1996]) internally?

The key observation is that, since the locations of introductions and eliminations are hard to predict, they must not affect evaluation. Any other design would mean that programmers lose the ability to reason about when their expressions are reduced.

The existing datatype-based approach requires an existential-typed expression to be evaluated to head normal form to access the *type* packed in the existential. This is silly, however: types are completely erased, and yet this rule means that we must perform runtime evaluation simply to access an erased component of a some data.

To illustrate the problem, consider this Haskell datatype:

```
data Exists (f :: Type → Type) = ∀(a :: Type). Ex !(f a)
```

With this construct, we can introduce existential types using the data constructor *Ex* and eliminate them by pattern matching on *Ex*. Note the presence of the strictness annotation, written with !. A use of the *Ex* data constructor, if it is automatically inserted by the type inferencer, must not block reduction.[3]

The difficult issue, however, is elimination. To access the value carried by *Exists*, we must use pattern matching. We cannot use a straightforward projection function *unExists* :: *Exists* *f* → *f* ???: it would allow the abstracted type variable to escape its scope—exactly why we cannot write a well-scoped type signature for *unExists*. As a result, we cannot use this value without weak-head evaluation of the term. As Section 3.2 shows, this forcing can decrease the laziness of our program.

While perhaps not as fundamental as our desire for introduction and elimination to be transparent to evaluation, another design goal is to allow arbitrary **let**-inlining. In other words, if **let** *x* = *e1* **in** *e2* type-checks, then *e2* [*e1* / *x*] should also type-check. This property gives flexibility to users: they (and their IDEs) can confidently refactor their program without fear of type errors.

Taken together, these design requirements—transparency to evaluation and support for **let**-inlining—drive us to enhance our core language with *strong* existentials [Howard 1969]: existentials that allow projection of both the type witness and the packed value, without pattern-matching.[4]

---

[3]Similarly, our choice of explicit introduction form for the core language must be strict in its argument if it is to be unobservable.

[4]Strong existentials stand in contrast to *weak* existentials. A strong existential package supports operators that access the encapsulated type and datum, while a weak existential requires pattern-matching in order to extract the datum and bring its type into scope. In a lazy language, strong existentials thus have greater expressive power, as we can use a lazy projection, as we do here.

3.1 Strong Existentials via pack and open

Our core language $\mathbb{FX}$ adopts the following constructs for introducing and eliminating existential types:[5]

$$\textsc{Pack}\quad \frac{\Gamma \vdash e : \tau_2[\tau_1/a]}{\Gamma \vdash \textbf{pack}\ \tau_1, e\ \textbf{as}\ \exists\, a.\tau_2 : \exists\, a.\tau_2} \qquad \textsc{Open}\quad \frac{\Gamma \vdash e : \exists\, a.\tau}{\Gamma \vdash \textbf{open}\ e : \tau[\lfloor e : \exists\, a.\tau \rfloor\ /\ a]}$$

The **pack** typing rule is fairly standard [Pierce 2002, Chapter 24]. This term creates an existential package, hiding a type τ_1 in the package with an expression e. Our operational semantics (Figure 7) includes a rule that makes this construct strict.

To eliminate existential types, we use the **open** construct (from Cardelli and Leroy [1990]) instead of pattern matching. The **open** construct eliminates an existential without forcing it, as **open**s are simply erased during compilation. The type of **open** e is interesting: we substitute away the bound variable a, replacing it with $\lfloor e : \exists\, a.\tau \rfloor$. This type is an *existential projection*. The idea is that we can think of an existential package $\exists\, a.\tau$ as a (dependent) pair, combining the choice for a (say, τ_0) with an expression of type $\tau[\tau_0\ /\ a]$. The type $\lfloor e : \exists\, a.\tau \rfloor$ projects out the type τ_0 from the pair.

A key aspect of **open** is that the type form $\lfloor e : \exists\, a.\tau \rfloor$ is a completely opaque type. In our surface language, $\lfloor e : \exists\, a.\tau \rfloor$ is equal to itself and no other type. Computation is not necessary in types. One way to think of this is to imagine that $\lfloor e : \exists\, a.\tau \rfloor$ is like a fresh type variable whose name is long—not as a construct that, say, accesses a type within e.

The simple idea of **open** is very powerful. It means that we can talk about the type in an existential package without unpacking the package. It would even be valid to project out the type of an existential package that will never be computed. Because types can be erased in our semantics, even projecting out the type from a bottoming expression (of existential type) is harmless.[6]

Note that the type of the existential package expression is included in the syntax for projections $\lfloor e : \exists\, a.\tau \rfloor$: this annotation is necessary because expressions in our surface language $\mathbb{X}$ might have multiple, different types. (For example, $\lambda x \rightarrow x$ has both type $Int \rightarrow Int$ and type $Bool \rightarrow Bool$.) Including the type annotation fixes our interpretation of e, but see Section 6 for more on this point.

3.2 The unpack Trap

Adding the **open** term to the language comes at a cost to complexity. Let us take a moment to reflect on why a more traditional elimination form (called **unpack**) is insufficient.

A frequent presentation of existentials in a language based on System F uses the **unpack** primitive. Pierce [2002, Chapter 24] presents the idea with this typing rule:

$$\textsc{Unpack}\quad \frac{\Gamma \vdash e_1 : \exists\, a.\tau_2 \qquad \Gamma, a, x{:}\tau_2 \vdash e_2 : \tau \qquad a \notin fv(\tau)}{\Gamma \vdash \textbf{unpack}\ e_1\ \textbf{as}\ a, x\ \textbf{in}\ e_2 : \tau}$$

The idea is that **unpack** extracts out the packed expression in a variable x, also binding a type variable a to represent the hidden type. The typing rule corresponds to the pattern-match in $\textbf{case}\ e_1\ \textbf{of}\ Ex\ (x :: \_\ a) \rightarrow e_2$, where x and a are brought into scope in e_2.[7]

---

[5] These rules are slightly simplified. The full rules appear in Section 5.

[6] Readers may be alarmed at that sentence: how could $\lfloor \bot : \exists\, a.a \rfloor$ be a valid type? Perhaps a more elaborate system might want to reject such a type, but there is no need to. As all types are erased and have no impact on evaluation, an exotic type like this is no threat to type safety.

[7] See Eisenberg et al. [2018] for more details on how Haskell treats that type annotation.

This approach is attractive because it is simple to add to a language like System F. It does not require the presence of terms in types and the necessary machinery that we describe in Section 5. However, it is also not powerful enough to accommodate some of the examples we would like to support.

The **unpack** *term impacts evaluation.* Because it is based on pattern matching, the **unpack** term must reduce its argument to a weak-head normal form before providing access to the hidden type. The standard reduction rule looks like this:

$$\textbf{unpack}\ (\textbf{pack}\ \tau_1, e_1\ \textbf{as}\ \exists\, a.\tau_2)\ \textbf{as}\ a, x\ \textbf{in}\ e_2 \longrightarrow e_1[e_1/x][\tau_1/a]$$

What this rule means is that the only parts of the term that have access to the abstract type are the ones that are evaluated after the existential has been weak-head normalized. Without weak-head normalizing the argument to a **pack**, we have nothing to substitute for x and a.

Let us rewrite the *filter* example from Section 2.2, making more details explicit so that we can see why this is an issue.

```
filter :: ∀n a. (a → Bool) → Vec n a → ∃m. Vec m a
filter = Λn a → λ(p :: a → Bool) (vec :: Vec n a) →
        case vec of
          (:>) n1 (x :: a) (xs :: Vec n1 a)          -- vec is x :> xs
            | p x          → ...
            | otherwise → filter n1 a p xs
          Nil → pack Zero, Nil as ∃m. Vec m a    -- vec is Nil
```

The treatment above makes all type abstraction and application explicit. Note that the pattern-match for the cons operator :> includes a compile-time (or type-level) binding for the length of the tail, *n1*.

The question here is: what do we put in the ... in the case where *p x* holds? One possibility is to apply the (:>) operator to build the result. However, right away, we are stymied: what do we pass to that operator as the length of the resulting vector? It depends on the length of the result of the recursive call. A use of **unpack** cannot help us here, as **unpack** is used in a term, not in a type index; even if we could use it, we would have to return the packed type, not something we can ordinarily do.

Instead, we must use **unpack** (and **pack**) *before* calling the (:>) operator. Specifically, we can write

```
unpack filter n1 a p xs as n2, ys in pack n2, (:>) n2 x ys as ∃m. Vec m a
```

This use of **unpack** is type-correct, but we have lost the laziness of *filter* we so prized in Section 2.2.

On the other hand, **open** allows us to fill in the ... with the following code, using the the existential projection to access the new (type-level) length for the arguments to **pack** and to :>.

```
let ys :: ∃m. Vec m a    -- usual lazy let
    ys = filter n1 a p xs
in pack ⌊ys :: ∃m. Vec m a⌋, (:>) ⌊ys :: ∃m. Vec m a⌋ x (open ys) as ∃m. Vec m a
```

As we expand on in the next subsection, we do not have to **let**-bind *ys*; instead, we could just repeat the sub-expression *filter n1 a p xs*.

3.3 The Importance of Strength

Beyond the peculiarities of the *filter* example, having a lazy construct that accesses the abstracted type in an existential package is essential to supporting inferrable existential types.

Here is a somewhat contrived example to illustrate this point:

```
data Counter a = Counter {zero :: a, succ :: a → a, toInt :: a → Int}

mkCounter :: String → ∃a. Counter a    -- a counter with a hidden representation
mkCounter = ...

initial1 :: Int
initial1 = let c = mkCounter "hello" in (toInt c) (zero c)

initial2 :: Int
initial2 = (toInt (mkCounter "hello")) (zero (mkCounter "hello"))
```

We would like our language to accept both *initial1* and *initial2*. After all, one of the benefits of working in a pure, lazy language is referential transparency: programmers (and tools, such as IDEs) should be able to perform expression inlining with no change in behavior. In both *initial1* and *initial2*, the compiler must automatically eliminate the existential that results from each use of *mkCounter*. In the definition *initial1*, elaboration is not difficult, even if we only have the weak **unpack** elimination form to work with.

However, supporting *initial2* is more problematic. Maintaining the order of evaluation of the source language requires two separate uses of the elimination form.

To type-check the application of *toInt* (*mkCounter* `"hello"`) to *zero* (*mkCounter* `"hello"`), we must first know the type packed into the package returned from *mkCounter* `"hello"`. Accessing this type should not evaluate *mkCounter* `"hello"`, however: a programmer rightly expects that *toInt* is evaluated before any call to *mkCounter* is, which may have performance or termination implications. More generally, we can imagine the need for a hidden type arbitrarily far away from the call site of a function (such as *mkCounter*) that returns an existential; eager evaluation of the function would be most unexpected for programmers.

Note that, critically, both calls to *mkCounter* in *initial2* contain the *same* argument. Since we are working in a pure context, we know that the result of the two calls to *mkCounter* `"hello"` in *initial2* must be the same, and thus that the program is well-typed.

In sum, if the compiler is to produce the elimination form for existentials, that elimination form must be *nonstrict*, allowing the packed witness type to be accessed without evaluation. Any other choice means that programmers must expect hard-to-predict changes to the evaluation order of their program. In addition, if we wish to allow users to inline their **let**-bound identifiers, this projection form must also be *strong*, and remember the existentially typed expression in its type.

Note that we are taking advantage of Haskell's purity in this part of the design. We can soundly support a strong elimination form like **open** only because we know that the expressions which appear in types are pure. All projections of the type witness from the same expression will be equal. In a language without this property, such as ML, we would need to enforce a value restriction on the type projections. Such a value restriction would prevent us from injecting, say, a non-deterministic expression into types; as there is no notion of evaluating a type, it would be unclear when and how often to evaluate the expression which could yield different results at each evaluation.

4 INFERRING EXISTENTIALS

In this section we present the surface language, $\mathbb{X}$, that we use to manipulate existentials, and the bidirectional type system that infers them. As our concrete setting is in Haskell, our starting point is the surface language described by Serrano et al. [2020], modified to add support for existentials. We add a syntax for existential quantifiers $\exists\, a.\epsilon$ and existential projections $\lfloor e : \epsilon \rfloor$. An important part of our type system is the type instantiation mechanism, which implicitly handles the opening of existentials (Section 4.3).

4.1 Language Syntax

The syntax of our types is given in Figure 2.

σ	$::=$	$\epsilon \mid \forall\, a.\sigma$	universally quantified type
ϵ	$::=$	$\rho \mid \exists\, b.\epsilon$	existentially quantified type
ρ	$::=$	$\tau \mid \sigma_1 \to \sigma_2$	top-level monomorphic type
τ	$::=$	$a \mid \mathsf{Int} \mid \tau_1 \to \tau_2 \mid \lfloor e : \epsilon \rfloor$	monomorphic type
a, b	$::=$	$\ldots$	type variable
Γ	$::=$	$\emptyset \mid \Gamma, a \mid \Gamma, x{:}\sigma$	typing context

Fig. 2. Type stratification

Polytypes σ can quantify an arbitrary number (including 0) universal variables and, within the universal quantification, an arbitrary number (including 0) existential variables. This stratification is enforced through the distinction between σ-types and ϵ-types. Note that the type $\exists\, a.\forall\, b.\tau$ is ruled out.[8] Top-level monotypes ρ have no top-level quantification. Monotypes τ include a projection form $\lfloor e : \epsilon \rfloor$ that occurs every time an existential is opened, as described in Section 3.1. Universal and existential variables draw from the same set of variable names, denoted with a or b.

The expressions of $\mathbb{X}$ are defined as follows:

x	$::=$	$\ldots$	term variable
n	$::=$	$\ldots$	integer literal
e	$::=$	$h\,\overline{\pi} \mid \lambda x.e \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \mid n$	expression
h	$::=$	$x \mid e \mid e :: \sigma$	expression head
π	$::=$	$e \mid \sigma$	argument

Fig. 3. Our surface language, $\mathbb{X}$

This language is a fairly small λ-calculus, with type annotations and n-ary application (including type application). The expression $h\,\pi_1 \ldots \pi_n$ applies a head to a sequence of arguments π_i that can be expressions or types. The head is either a variable x, an annotated expression $e :: \sigma$, or an expression e that is not an application.[9]

An important complication of our type system is that expressions may appear in types: this happens in the projection form $\lfloor e : \epsilon \rfloor$. We thus must address how to treat type equality. For example, suppose term variable x (of type Int) is free in a type τ; is $\tau[(\lambda y.y)\ 1 \,/\, x]$ equal to $\tau[1 \,/\, x]$?

[8] As usual, stratifying the grammar of types simplifies type inference. In our case, this choice drastically simplifies the challenge of comparing types with mixed quantifiers. Dunfield and Krishnaswami [2019, Section 2] have an in-depth discussion of this challenge.

[9] Our grammar does not force a head expression h to be something other than an application, but we will consistently assume this restriction is in force. It would add clutter and obscure our point to bake this restriction in the grammar.

$\boxed{\Gamma \vdash^{\forall} e \Leftarrow \sigma}$ *(Universal type checking)*

$$\textsc{Gen}\ \frac{\Gamma, \overline{a} \vdash e \Leftarrow \rho[\overline{\tau} / \overline{b}] \quad fv(\overline{\tau}) \subseteq dom(\Gamma, \overline{a})}{\Gamma \vdash^{\forall} e \Leftarrow \forall \overline{a}.\exists \overline{b}.\rho}$$

$\boxed{\Gamma \vdash e \Rightarrow \rho \qquad \Gamma \vdash e \Leftarrow \rho}$ *(Type synthesis and type checking)*

$$\textsc{App}\ \frac{\Gamma \vdash_h h \Rightarrow \sigma \quad \Gamma \vdash^{\mathsf{inst}} h : \sigma ; \overline{\pi} \leadsto \overline{\sigma} ; \rho_r \quad \overline{e} = exprargs(\overline{\pi}) \quad \overline{\Gamma \vdash^{\forall} e_i \Leftarrow \sigma_i}}{\Gamma \vdash h\,\overline{\pi} \Leftrightarrow \rho_r}$$

$$\textsc{iAbs}\ \frac{\Gamma, x{:}\tau \vdash e \Rightarrow \rho \quad fv(\tau) \subseteq dom(\Gamma) \quad \overline{a}\ \textbf{fresh} \quad \rho' = \rho[\overline{a} / \lfloor \rho \rfloor_x] \quad (\textit{see §4.2.3})}{\Gamma \vdash \lambda x.e \Rightarrow \tau \to \exists \overline{a}.\rho'}$$

$$\textsc{cAbs}\ \frac{\Gamma, x{:}\sigma_1 \vdash^{\forall} e \Leftarrow \sigma_2 \quad fv(\sigma_1) \subseteq dom(\Gamma)}{\Gamma \vdash \lambda x.e \Leftarrow \sigma_1 \to \sigma_2}$$

$$\textsc{Int}\ \frac{}{\Gamma \vdash n \Leftrightarrow \mathsf{Int}}$$

$$\textsc{Let}\ \frac{\Gamma \vdash e_1 \Rightarrow \rho_1 \quad \overline{a} = fv(\rho_1) \backslash dom(\Gamma) \quad \Gamma, x{:}\forall \overline{a}.\rho_1 \vdash e_2 \Leftrightarrow \rho_2}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \Leftrightarrow \rho_2[e_1 / x]}$$

$\boxed{\Gamma \vdash_h h \Rightarrow \sigma}$ *(Head synthesis)*

$$\textsc{H-Var}\ \frac{x{:}\sigma \in \Gamma}{\Gamma \vdash_h x \Rightarrow \sigma}$$

$$\textsc{H-Ann}\ \frac{\Gamma \vdash^{\forall} e \Leftarrow \sigma \quad fv(\sigma) \subseteq dom(\Gamma)}{\Gamma \vdash_h (e :: \sigma) \Rightarrow \sigma}$$

$$\textsc{H-Infer}\ \frac{\Gamma \vdash e \Rightarrow \rho}{\Gamma \vdash_h e \Rightarrow \rho}$$

Fig. 4. Type inference for $\mathbb{X}$

That is, does type equality respect β-reduction? Our answer is "no": we restrict type equality in our language to be syntactic equality (modulo α-equivalence, as usual). We can imagine a richer type equality relation—which would accept more programs—but this simplest, least expressive version satisfies our needs. (However, see Aside 2 in Section 7.3 for a wrinkle here.) Adding such an equality relation is largely orthogonal to the concerns around existential types that draw our focus.[10]

4.2 Type System

The typing rules of our language appear in Figure 4. This bidirectional type system uses two forms for typing judgments: $\Gamma \vdash e \Rightarrow \rho$ means that, in the type environment Γ, the program e has the inferred type ρ, while $\Gamma \vdash e \Leftarrow \rho$ means that, in the type environment Γ, e is checked to have type ρ. We also use a third form to simplify the presentation of the rules: $\Gamma \vdash e \Leftrightarrow \rho$, which means that the rule can be read by replacing $\Leftrightarrow$ with either $\Rightarrow$ or $\Leftarrow$ in both the conclusion and premises. Although the rules are fairly close to the standard rules of a typed λ-calculus, handling existentials through **pack**ing and **open**ing has an impact on the rules Let and Gen.

[10] Our core language $\mathbb{FX}$ *does* need to think harder about this question, in order to prove type safety. See Section 5.1.

We review the rules in Figure 4 here, deferring the most involved rule, App, until after we discuss the instantiation judgment $\vdash^{\mathsf{inst}}$, in Section 4.3.

4.2.1 Simple Subsumption. Bidirectional type systems typically rely on a reflexive, transitive *subsumption relation* $\leqslant$, where we expect that if $e : \sigma_1$ and $\sigma_1 \leqslant \sigma_2$, then $e : \sigma_2$ is also derivable. For example, we would expect that $\forall\, a.a \rightarrow a \leqslant \mathsf{Int} \rightarrow \mathsf{Int}$. This subsumption relation is then used when "switching modes"; that is, if we are checking an expression e against a type σ_2 where e has a form resistant to type propagation (the case when e is a function call), we infer a type σ_1 for e and then check that $\sigma_1 \leqslant \sigma_2$.

However, our type system refers to no such $\leqslant$ relation: we essentially use equality as our subsumption relation, invoking it implicitly in our rules through the use of a repeated metavariable. (Though hard to see, the repeated metavariable is the ρ_r in rule App, when replacing the $\Leftrightarrow$ in the conclusion with a $\Leftarrow$.) We get away with this because our bidirectional type-checking algorithm works over top-level monotypes ρ, not the more general polytype σ. A type ρ has no top-level quantification at all. Furthermore, our type system treats all types as invariant—including $\rightarrow$. This treatment follows on from the ideas in Serrano et al. [2020, Section 5.8], which describes how Haskell recently made its arrow type similarly invariant.

We adopt this simpler approach toward subsumption both to connect our presentation with the state-of-the-art for type inference in Haskell [Serrano et al. 2020] and also because this approach simplifies our typing rules. We see no obstacle to incorporating our ideas with a more powerful subsumption judgment, such as the deep-skolemization judgment of Peyton Jones et al. [2007, Section 4.6.2] or the slightly simpler co- and contravariant judgment of Odersky and Läufer [1996, Figure 2].

4.2.2 Checking against a Polytype. Rule Gen, the sole rule for the $\Gamma \vdash^{\forall} e \Leftarrow \sigma$ judgment, deals with the case when we are checking against a polytype σ. If we want to ensure that e has type σ, then we must *skolemize* any universal variables bound in σ: these variables behave essentially as fresh constants while type-checking e. Rule Gen thus just brings them into scope.

On the other hand, if there are existential variables bound in σ, then we must *instantiate* these. If we are checking that e has some type $\exists\, a.\tau_0$, that means we must find some type τ such that e has type $\tau_0[\tau \,/\, a]$. This is very different than the skolemization of a universal variable, where we must keep the variable abstract. Instead, when checking against $\exists\, a.\epsilon$, we guess a monotype τ and check e against the type $\epsilon[\tau \,/\, a]$. Rule Gen simply does this for nested existential quantification over variables $\overline{b}$. A real implementation might use unification variables, but we here rely on the rich body of literature [e.g., Dunfield and Krishnaswami 2013] that allows us to guess monotypes during type inference, knowing how to translate this convention into an implementation using unification variables.

4.2.3 Abstractions. Rule iAbs synthesizes the type of a λ-abstraction, by guessing the (mono)type τ of the bound variable and then inferring the type of the body e to be ρ. However, rule iAbs *also* can pack existentials. This is necessary to avoid skolem escape: it is possible that the type ρ contains x free. However, it would be disastrous if $\lambda x.e$ was assigned a type mentioning x, as x is no longer in scope.

For example, suppose we have $\Gamma = f{:}\mathsf{Int} \rightarrow \exists\, a.a \rightarrow \mathsf{Bool}$. Now, consider inferring the type ρ in $\Gamma \vdash \lambda x.f\ x \Rightarrow \rho$. Guessing $x : \mathsf{Int}$, we will infer $\Gamma, x{:}\mathsf{Int} \vdash f\ x \Rightarrow \lfloor f\ x : \exists\, a.a \rightarrow \mathsf{Bool} \rfloor \rightarrow \mathsf{Bool}$. It is tempting, then, to say $\Gamma \vdash \lambda x.f\ x \Rightarrow \mathsf{Int} \rightarrow \lfloor f\ x : \exists\, a.a \rightarrow \mathsf{Bool} \rfloor \rightarrow \mathsf{Bool}$, but this is wrong: the type mentions x free, but Γ does not bind x. Instead, rule iAbs infers $\Gamma \vdash \lambda x.f\ x \Rightarrow \exists\, a.\mathsf{Int} \rightarrow a \rightarrow \mathsf{Bool}$, by using a instead of the ill-scoped $\lfloor f\ x : \exists\, a.a \rightarrow \mathsf{Bool} \rfloor$.

More generally, we must identify all existential projections within ρ that have x free. These are replaced with fresh variables $\overline{a}$. We use the notation $\lfloor \rho \rfloor_x$ to denote the list of projections in ρ; multiple projections of the same expression (that is, multiple occurrences of $\lfloor e_0 : \epsilon_0 \rfloor$ for some e_0 and ϵ_0) are commoned up in this list. Formally,

$$\lfloor \rho \rfloor_x = \{ \lfloor e : \epsilon \rfloor \ \mid \ (\lfloor e : \epsilon \rfloor \text{ is a sub-expression of } \rho) \wedge (x \text{ is a free variable in } e) \} \,.$$

The notation $\rho[\overline{a} \,/\, \lfloor \rho \rfloor_x]$ denotes the type ρ where the $\overline{a}$ are written in place of these projections. Note that this notation is set up *backward* from the way it usually works, where we substitute some type for a variable. Here, instead, we are replacing the type with a fresh variable.

In the conclusion of the rule, we existentially quantify the $\overline{a}$, to finally obtain a function type of the form $\tau \to \exists\,\overline{a}.\rho'$.[11]

The checking rule cAbs is much simpler. We know the type of the bound variable by decomposing the known expected type $\sigma_1 \to \sigma_2$. We also need not worry about skolem escape because we have been provided with a well-scoped σ_2 result type for our function. The only small wrinkle is the need to use $\vdash^{\forall}$ in order to invoke rule Gen to remove any quantifiers on the type σ_2.

4.2.4 Let Skolem-escape. Rule Let deals with **let**-expressions, both in synthesis and in checking modes. It performs standard **let**-generalization, computing generalized variables $\overline{a}$ by finding the free variables in ρ_1 and removing any variables additionally free in Γ. Indeed, all that is unexpected in this rule is the type substitution in the conclusion.

The problem, like with rule iAbs is the potential for skolem-escape. The variable x might appear in the type ρ_2. However, x is out of scope in the conclusion, and thus it cannot appear in the overall type of the **let**-expression. One solution to this problem would be to **pack** all the existentials that fall out of scope, much like we do in rule iAbs. However, doing so would mean that our bidirectional type system now infers existential types ϵ instead of top-level monomorphic types ρ; keeping with the simpler ρ is important to avoid the complications of a non-trivial subsumption judgment. Hence we choose to replace all occurrences of x inside of projections by the expression e_1. This does not pose a problem since e_1 is well-typed according to the premises of the Let rule.

4.2.5 Inferring the Types of Heads. Following Serrano et al. [2020], our system treats n-ary applications directly, instead of recurring down a chain of binary applications $e_1\ e_2$. The head of an n-ary application is denoted with h; heads' types are inferred with the $\Gamma \vdash_h h \Rightarrow \sigma$ judgment. Variables simply perform a context lookup, annotated expressions check the contained expression against the provided type, and other expressions infer a ρ-type. It is understood here that we use rule H-Infer only when the other rules do not apply, for example, for λ-abstractions.

4.3 Instantiation Semantics

The instantiation rules of Figure 5 present an auxiliary judgment used in type-checking applications. The judgment $\Gamma \vdash^{\mathsf{inst}} e : \sigma \,;\, \overline{\pi} \leadsto \overline{\sigma} \,;\, \rho_r$ means: with in-scope variables Γ, apply function e of type σ to arguments $\overline{\pi}$ requires $exprargs(\overline{\pi})$ (the value arguments) to have types $\overline{\sigma}$, resulting in an expression $e\,\overline{\pi}$ of type ρ_r. This judgment is directly inspired by Serrano et al. [2020, Figure 4].

The idea is that we use $\vdash^{\mathsf{inst}}$ to figure out the types of term-level arguments to a function in a pre-pass that examines only type arguments. Having determined the expected types of the term-level arguments $\overline{\sigma}$, rule App (in Figure 4) actually checks that the arguments have the correct types. This pre-pass is not necessary in order to infer the types for existentials, but it sets the stage for Section 8, where we integrate our design with the current implementation in GHC.

[11] Our language works well without this special substitution. Instead, we could have a check that the final inferred type in rule iAbs is well scoped. However, this extra existential packing is easy enough to add, and so we have.

$\boxed{\Gamma \vdash^{\mathsf{inst}} e : \sigma \,;\, \overline{\pi} \leadsto \overline{\sigma} \,;\, \rho_r}$ *(Instantiation judgment)*

$$\textsc{ITyArg}\quad \frac{\Gamma \vdash^{\mathsf{inst}} e\,\sigma' : \sigma[\sigma' / a] \,;\, \overline{\pi} \leadsto \overline{\sigma} \,;\, \rho_r \qquad \mathit{fv}(\sigma') \subseteq \mathit{dom}(\Gamma)}{\Gamma \vdash^{\mathsf{inst}} e : \forall\, a.\sigma \,;\, \sigma', \overline{\pi} \leadsto \overline{\sigma} \,;\, \rho_r}$$

$$\textsc{IArg}\quad \frac{\Gamma \vdash^{\mathsf{inst}} e\,e' : \sigma_2 \,;\, \overline{\pi} \leadsto \overline{\sigma} \,;\, \rho_r}{\Gamma \vdash^{\mathsf{inst}} e : (\sigma_1 \to \sigma_2) \,;\, e', \overline{\pi} \leadsto \sigma_1, \overline{\sigma} \,;\, \rho_r}$$

$$\textsc{IAll}\quad \frac{\overline{\pi} \neq \sigma', \overline{\pi}' \qquad \Gamma \vdash^{\mathsf{inst}} e : \sigma[\tau / a] \,;\, \overline{\pi} \leadsto \overline{\sigma} \,;\, \rho_r \qquad \mathit{fv}(\tau) \subseteq \mathit{dom}(\Gamma)}{\Gamma \vdash^{\mathsf{inst}} e : \forall\, a.\sigma \,;\, \overline{\pi} \leadsto \overline{\sigma} \,;\, \rho_r}$$

$$\textsc{IExist}\quad \frac{\Gamma \vdash^{\mathsf{inst}} e : \epsilon[\lfloor e : \exists\, a.\epsilon \rfloor / a] \,;\, \overline{\pi} \leadsto \overline{\sigma} \,;\, \rho_r}{\Gamma \vdash^{\mathsf{inst}} e : \exists\, a.\epsilon \,;\, \overline{\pi} \leadsto \overline{\sigma} \,;\, \rho_r}$$

$$\textsc{IResult}\quad \frac{}{\Gamma \vdash^{\mathsf{inst}} e : \rho_r \,;\, [\,] \leadsto [\,] \,;\, \rho_r}$$

Fig. 5. Instantiation

Application. Rule ITyArg handles type application by instantiating the bound variable a with the supplied type argument σ'. Rule IArg handles routine expression application simply by remembering that the argument should have type σ_1. Note that we do *not* check that the argument e' has type σ_1 here.

Quantifiers. Rule IAll deals with universal quantifiers in the function's type by instantiating with a guessed monotype τ. The first premise is to avoid ambiguity with rule ITyArg; we do not wish to guess an instantiation when the user provides it explicitly with a type argument.

Rule IExist eagerly opens existentials by substituting a projection in place of the bound variable a. This is the only place in the judgment where we need the function expression e: whenever we open an existential type, we must remember what expression has that type, so that we do not confuse two different existentially packed types.

For example, if f has type $\mathsf{Bool} \to \exists\, b.(b, b \to \mathsf{Int})$, then the function application f **True** will be given the opened pair type:

$$(\lfloor f\ \mathbf{True} : \exists\, b.(b, b \to \mathsf{Int})\rfloor, \lfloor f\ \mathbf{True} : \exists\, b.(b, b \to \mathsf{Int})\rfloor \to \mathsf{Int})$$

Rule IResult concludes computing the instantiation in a function application by copying the function type to be the result type.

The App *rule.* Having now understood the instantiation judgment, we turn our attention to rule App. After inferring the type σ for an application head h, σ gets instantiated, revealing argument types $\overline{\sigma}$. Each argument e_i is checked against its corresponding type σ_i, where the entire function application expression has type ρ_r. Rule App operates in both synthesis and checking modes. When synthesizing, it simply returns ρ_r from the instantiation judgment; when checking, it ensures that the instantiated type ρ_r matches what was expected. We need do no further instantiation or skolemization because we have a simple subsumption relation.

5 CORE LANGUAGE

Perhaps we can infer existential types using existential projections $\lfloor e : \epsilon \rfloor$, but how do we know such an approach is sound? We show that it is by elaborating our surface expressions into a core language $\mathbb{FX}$, inspired by a similar language described by Cardelli and Leroy [1990, Section 4], and we prove the standard progress and preservation theorems of this language. This section presents $\mathbb{FX}$ and states key metatheory results; the following section connects $\mathbb{X}$ to $\mathbb{FX}$ by presenting our elaboration algorithm.

The syntax of $\mathbb{FX}$ is in Figure 6 and selected typing rules are in Figure 7; full typing rules appear in the appendix.[12] Note that we use upright Latin letters to denote $\mathbb{FX}$ expressions and types; when we mix $\mathbb{X}$ and $\mathbb{FX}$ in close proximity, we additionally use colors.

B	$::=$	$\rightarrow \mid \mathsf{Int} \mid \ldots$	base type
$\mathrm{t, r, s}$	$::=$	$a \mid B\,\overline{\mathrm{t}} \mid \forall\, a.\mathrm{t} \mid \exists\, a.\mathrm{t} \mid \lfloor \mathrm{e} \rfloor$	type
$\mathrm{e, h}$	$::=$	$x \mid n \mid \lambda x{:}\mathrm{t.e} \mid \mathrm{e}_1\,\mathrm{e}_2 \mid \Lambda a.\mathrm{e} \mid \mathrm{e\,t} \mid \mathbf{pack}\,\mathrm{t, e}\,\mathbf{as}\,\mathrm{t}_2$	
	$\mid$	$\mathbf{open}\,\mathrm{e} \mid \mathbf{let}\,x = \mathrm{e}_1\,\mathbf{in}\,\mathrm{e}_2 \mid \mathrm{e} \rhd \gamma$	expression
v	$::=$	$n \mid \lambda x{:}\mathrm{t.e} \mid \Lambda a.\mathrm{v} \mid \mathbf{pack}\,\mathrm{t, v}\,\mathbf{as}\,\mathrm{t}_2$	value
γ	$::=$	$\langle \mathrm{t} \rangle \mid \mathbf{sym}\,\gamma \mid \gamma_1 \,;\!;\, \gamma_2 \mid \lfloor \eta \rfloor \mid \gamma_1 \,@\, \gamma_2 \mid \mathbf{projpack}\,\mathrm{t, e}\,\mathbf{as}\,\mathrm{t}_2 \mid \ldots$	type coercion
η	$::=$	$\mathrm{e} \rhd \gamma \mid \mathbf{step}\,\mathrm{e}$	expression coercion
G	$::=$	$\emptyset \mid \mathrm{G}, x : \mathrm{t} \mid \mathrm{G}, a$	typing context

Fig. 6. Syntax of the core language, $\mathbb{FX}$

The nub of $\mathbb{FX}$ is System F, with fully applied base types B (because they are fully applied, we do not need to have a kind system) and ordinary universal quantification. We thus omit typing rules from this presentation that are standard. The inclusion of existential types, **pack** and **open** is fitting for a core language supporting existentials. This language necessarily has mutually recursive grammars for types and expressions, but the typing rules are not mutually recursive: rule CT-Proj shows that a projection in a type is well-formed when the expression is well-scoped. (The $\vdash \mathrm{G}\ \mathbf{ok}$ premise refers to a routine context-well-formedness judgment, omitted.) We do not require the existential package to be well-typed (though it would be, in practice).

5.1 Coercions

The biggest surprise in $\mathbb{FX}$ is its need for type and expression *coercions*. The motivation for these can be seen in rule CS-OpenPack. If we are stepping an expression $\mathbf{open}\,(\mathbf{pack}\,\mathrm{t, v}\,\mathbf{as}\,\exists\, a.\mathrm{t}_2)$, we want to extract the value v from the existential package. The problem is that v has the wrong type. Suppose that v has type t_0. Then, we have $\mathbf{pack}\,\mathrm{t, v}\,\mathbf{as}\,\exists\, a.\mathrm{t}_2 : \exists\, a.\mathrm{t}_2$ and $\mathbf{open}\,(\mathbf{pack}\,\mathrm{t, v}\,\mathbf{as}\,\exists\, a.\mathrm{t}_2) : \mathrm{t}_2[\lfloor \mathbf{pack}\,\mathrm{t, v}\,\mathbf{as}\,\exists\, a.\mathrm{t}_2 \rfloor \,/\, a]$, according to rule CE-Open. This last type is not syntactically the same as t_0, although it must be that $\mathrm{t}_0 = \mathrm{t}_2[\mathrm{t} \,/\, a]$ to satisfy the premises of rule CE-Pack. Because the type of the opened existential does not match the type of the packed value, a naïve reduction rule like $\mathrm{G} \vdash \mathbf{open}\,(\mathbf{pack}\,\mathrm{t, v}\,\mathbf{as}\,\mathrm{t}_2) \longrightarrow \mathrm{v}$ would not preserve types.

There are, in general, two ways to build a type system when encountering such a problem. We could have a non-trivial type equality relation, where we say that $\lfloor \mathbf{pack}\,\mathrm{t, e}\,\mathbf{as}\,\mathrm{t}_2 \rfloor \equiv \mathrm{t}$. Doing so would simplify the reduction rules, but this simplification comes at a cost: our language would now have a conversion rule that allows an expression of one type t_1 to have another type t_2 as long as $\mathrm{t}_1 \equiv \mathrm{t}_2$. This rule is not syntax-directed; accordingly, it is hard to determine whether type-checking remains decidable. Furthermore, a non-trivial type equality relation makes proofs considerably more involved. In effect, we are just moving the complexity we see in the right-hand side of a rule like rule CS-OpenPack into the proofs.

[12] https://richarde.dev/papers/2021/exists/exists-extended.pdf

$\boxed{\mathsf{G} \vdash \mathsf{e} : \mathsf{t}}$ *(Expression typing)*

$$\textsc{CE-Abs}\quad \frac{\begin{array}{c}\mathsf{G}, x : \mathsf{t}_1 \vdash \mathsf{e} : \mathsf{t}_2 \\ x \notin \mathit{fv}(\mathsf{t}_2)\end{array}}{\mathsf{G} \vdash \lambda x{:}\mathsf{t}_1.\mathsf{e} : \mathsf{t}_1 \longrightarrow \mathsf{t}_2}
\qquad
\textsc{CE-Let}\quad \frac{\begin{array}{c}\mathsf{G} \vdash \mathsf{e}_1 : \mathsf{t}_1 \\ \mathsf{G}, x : \mathsf{t}_1 \vdash \mathsf{e}_2 : \mathsf{t}_2\end{array}}{\mathsf{G} \vdash \mathbf{let}\, x = \mathsf{e}_1\, \mathbf{in}\, \mathsf{e}_2 : \mathsf{t}_2[\mathsf{e}_1 / x]}$$

$$\textsc{CE-Pack}\quad \frac{\begin{array}{c}\mathsf{G} \vdash \mathsf{t} : \mathbf{type} \\ \mathsf{G} \vdash \exists\, a.\mathsf{t}_2 : \mathbf{type} \\ \mathsf{G} \vdash \mathsf{e} : \mathsf{t}_2[\mathsf{t} / a]\end{array}}{\mathsf{G} \vdash \mathbf{pack}\, \mathsf{t}, \mathsf{e}\, \mathbf{as}\, \exists\, a.\mathsf{t}_2 : \exists\, a.\mathsf{t}_2}$$

$$\textsc{CE-Open}\quad \frac{\mathsf{G} \vdash \mathsf{e} : \exists\, a.\mathsf{t}}{\mathsf{G} \vdash \mathbf{open}\, \mathsf{e} : \mathsf{t}[\lfloor \mathsf{e} \rfloor / a]}
\qquad
\textsc{CE-Cast}\quad \frac{\mathsf{G} \vdash \mathsf{e} : \mathsf{t}_1 \qquad \mathsf{G} \vdash \gamma : \mathsf{t}_1 \sim \mathsf{t}_2}{\mathsf{G} \vdash \mathsf{e} \rhd \gamma : \mathsf{t}_2}$$

$\boxed{\mathsf{G} \vdash \mathsf{t} : \mathbf{type}}$ *(Type well-formedness)*

$$\textsc{CT-Proj}\quad \frac{\vdash \mathsf{G}\, \mathbf{ok} \qquad \mathit{fv}(\mathsf{e}) \subseteq \mathit{dom}(\mathsf{G})}{\mathsf{G} \vdash \lfloor \mathsf{e} \rfloor : \mathbf{type}}$$

$\boxed{\mathsf{G} \vdash \gamma : \mathsf{t}_1 \sim \mathsf{t}_2}$ *(Type coercion typing)*

$$\textsc{CG-Refl}\quad \frac{\mathsf{G} \vdash \mathsf{t} : \mathbf{type}}{\mathsf{G} \vdash \langle \mathsf{t} \rangle : \mathsf{t} \sim \mathsf{t}}
\qquad
\textsc{CG-Sym}\quad \frac{\mathsf{G} \vdash \gamma : \mathsf{t}_1 \sim \mathsf{t}_2}{\mathsf{G} \vdash \mathbf{sym}\, \gamma : \mathsf{t}_2 \sim \mathsf{t}_1}$$

$$\textsc{CG-Trans}\quad \frac{\begin{array}{c}\mathsf{G} \vdash \gamma_1 : \mathsf{t}_1 \sim \mathsf{t}_2 \\ \mathsf{G} \vdash \gamma_2 : \mathsf{t}_2 \sim \mathsf{t}_3\end{array}}{\mathsf{G} \vdash \gamma_1 \,;;\, \gamma_2 : \mathsf{t}_1 \sim \mathsf{t}_3}
\qquad
\textsc{CG-Proj}\quad \frac{\mathsf{G} \vdash \eta : \mathsf{e}_1 \sim \mathsf{e}_2}{\mathsf{G} \vdash \lfloor \eta \rfloor : \lfloor \mathsf{e}_1 \rfloor \sim \lfloor \mathsf{e}_2 \rfloor}$$

$$\textsc{CG-InstExists}\quad \frac{\begin{array}{c}\mathsf{G} \vdash \gamma_1 : (\exists\, a.\mathsf{t}_1) \sim (\exists\, a.\mathsf{t}_2) \\ \mathsf{G} \vdash \gamma_2 : \mathsf{t}_3 \sim \mathsf{t}_4\end{array}}{\mathsf{G} \vdash \gamma_1 \,@\, \gamma_2 : \mathsf{t}_1[\mathsf{t}_3 / a] \sim \mathsf{t}_2[\mathsf{t}_4 / a]}
\qquad
\textsc{CG-ProjPack}\quad \frac{\mathsf{G} \vdash \mathbf{pack}\, \mathsf{t}, \mathsf{e}\, \mathbf{as}\, \mathsf{t}_2 : \mathsf{t}_2}{\mathsf{G} \vdash \mathbf{projpack}\, \mathsf{t}, \mathsf{e}\, \mathbf{as}\, \mathsf{t}_2 : \lfloor \mathbf{pack}\, \mathsf{t}, \mathsf{e}\, \mathbf{as}\, \mathsf{t}_2 \rfloor \sim \mathsf{t}}$$

$\boxed{\mathsf{G} \vdash \eta : \mathsf{e}_1 \sim \mathsf{e}_2}$ *(Expression coercion typing)*

$$\textsc{CH-Coherence}\quad \frac{\mathsf{G} \vdash \mathsf{e} : \mathsf{t}_1 \qquad \mathsf{G} \vdash \gamma : \mathsf{t}_1 \sim \mathsf{t}_2}{\mathsf{G} \vdash \mathsf{e} \rhd \gamma : \mathsf{e} \sim (\mathsf{e} \rhd \gamma)}
\qquad
\textsc{CH-Step}\quad \frac{\begin{array}{c}\mathsf{G} \vdash \mathsf{e} : \mathsf{t} \\ \mathsf{G} \vdash \mathsf{e}' : \mathsf{t} \qquad \mathsf{G} \vdash \mathsf{e} \longrightarrow \mathsf{e}'\end{array}}{\mathsf{G} \vdash \mathbf{step}\, \mathsf{e} : \mathsf{e} \sim \mathsf{e}'}$$

$\boxed{\mathsf{G} \vdash \mathsf{e} \longrightarrow \mathsf{e}'}$ *(Small-step operational semantics)*

$$\textsc{CS-PackCong}\quad \frac{\mathsf{G} \vdash \mathsf{e} \longrightarrow \mathsf{e}'}{\mathsf{G} \vdash \mathbf{pack}\, \mathsf{t}, \mathsf{e}\, \mathbf{as}\, \mathsf{t}_2 \longrightarrow \mathbf{pack}\, \mathsf{t}, \mathsf{e}'\, \mathbf{as}\, \mathsf{t}_2}$$

$$\textsc{CS-OpenPack}\quad \frac{}{\mathsf{G} \vdash \mathbf{open}\, (\mathbf{pack}\, \mathsf{t}, \mathsf{v}\, \mathbf{as}\, \mathsf{t}_2) \longrightarrow \mathsf{v} \rhd \langle \mathsf{t}_2 \rangle \,@(\mathbf{sym}\, (\mathbf{projpack}\, \mathsf{t}, \mathsf{v}\, \mathbf{as}\, \mathsf{t}_2))}$$

$$\textsc{CS-OpenCong}\quad \frac{\mathsf{G} \vdash \mathsf{e} : \mathsf{t} \qquad \mathsf{G} \vdash \mathsf{e} \longrightarrow \mathsf{e}'}{\mathsf{G} \vdash \mathbf{open}\, \mathsf{e} \longrightarrow \mathbf{open}\, \mathsf{e}' \rhd \langle \mathsf{t} \rangle \,@(\mathbf{sym}\, \lfloor \mathbf{step}\, \mathsf{e} \rfloor)}$$

Fig. 7. Selected typing and reduction rules of the core language, $\mathbb{FX}$

The alternative approach to a non-trivial equality relation is to use explicit coercions, as we have here. The cost is clutter. Casts sully our reduction steps, and we need to explicitly shunt coercions in several (omitted, unenlightening) reduction rules—for example, when reducing $((\lambda x{:}\mathrm{t}.\mathrm{e}_1) \triangleright \gamma)\ \mathrm{e}_2$ where the cast intervenes between a λ-abstraction and its argument. Despite the presence of these rules in our operational semantics, coercions can be fully erased: we can write an alternative, untyped operational semantics that omits coercions entirely. Theorem 7.2 shows that erasure preserves program behavior.

Both approaches—an enriched definitional equality vs. explicit coercions—are essentially equivalent: we can view explicit coercions simply as an encoding of the derivation of an equality judgment.[13] We choose explicit coercions both because $\mathbb{FX}$ is a purely internal language (and thus clutter is less noisome) and because it allows for an easy connection to the implementation of the core language in GHC, based on System FC [Sulzmann et al. 2007], with similar explicit coercions.

The coercion language for $\mathbb{FX}$ includes constructors witnessing that they encode an equivalence relation (rules CG-Refl, CG-Sym, and CG-Trans), along with several omitted forms showing that the equivalence is also a congruence over types. Coercions also include several decomposition operations; rule CG-InstExists shows one, used in our reduction rules. The two forms of interest to use are $\lfloor \eta \rfloor$ (rule CG-Proj) and **projpack** (rule CG-ProjPack). The former injects the equivalence relation on expressions (witnessed by expression coercions η) into the type equivalence relation, and the latter witnesses the equivalence between $\lfloor \mathbf{pack}\ \mathrm{t}, \mathrm{e}\ \mathbf{as}\ \mathrm{t} \rfloor$ and its packed type t.

The equivalence relation on expressions is surprisingly simple: we need only the two rules in Figure 7. These rules allow us to drop casts (supporting a coherence property which states that the presence of casts is essentially unimportant) and to reduce expressions.

5.2 Metatheory

We prove (almost) standard progress and preservation theorems for this language:

Theorem 5.1 (Progress). *If* $\mathrm{G} \vdash \mathrm{e} : \mathrm{t}$, *where* G *contains only type variable bindings, then one of the following is true:*

(1) *there exists* e' *such that* $\mathrm{G} \vdash \mathrm{e} \longrightarrow \mathrm{e}'$;
(2) e *is a value* v; *or*
(3) e *is a casted value* $\mathrm{v} \triangleright \gamma$.

Theorem 5.2 (Preservation). *If* $\mathrm{G} \vdash \mathrm{e} : \mathrm{t}$ *and* $\mathrm{G} \vdash \mathrm{e} \longrightarrow \mathrm{e}'$, *then* $\mathrm{G} \vdash \mathrm{e}' : \mathrm{t}$.

In addition, we prove that types can still be erased in this language. Let $|\mathrm{e}|$ denote the expression e with all type abstractions, type applications, **pack**s, **open**s and casts dropped. Furthermore, overload $\longrightarrow$ to mean the reduction relation over the erased language.

Theorem 5.3 (Erasure). *If* $\mathrm{G} \vdash \mathrm{e} \longrightarrow^* \mathrm{e}'$, *then* $|\mathrm{e}| \longrightarrow^* |\mathrm{e}'|$.

The proofs largely follow the pattern set by previous papers on languages with explicit coercions and are unenlightening. They appear, in full, in the appendix.

6 ELABORATION

We now augment our inference rules from Section 4 to describe the elaboration from the surface language $\mathbb{X}$ into our core $\mathbb{FX}$. The notation $\rightrightarrows$ denotes elaboration of a surface term, type or context into its core equivalent. Some of our rules appear in Figure 8. The rest appear in the appendix. In order to aid understanding, we use blue for $\mathbb{X}$ terms and red for $\mathbb{FX}$ terms.

[13] Weirich et al. [2017] makes this equivalence even clearer by presenting two proved-equivalent versions of a language, one with a non-trivial, undecidable type equality relation and another with explicit coercions.

$\Gamma \vdash^{\forall} e \Leftarrow \sigma \rightrightarrows \mathrm{e}$	elaboration of polymorphic expressions
$\Gamma \vdash e \Leftrightarrow \rho \rightrightarrows \mathrm{e}$	elaboration of expressions
$\Gamma \vdash_h h \Rightarrow \sigma \rightrightarrows \mathrm{h}$	elaboration of application heads
$\Gamma \vdash^{\mathsf{inst}} e : \sigma \rightrightarrows \mathrm{e} \,;\, \overline{\pi} \rightsquigarrow \overline{\sigma} \,;\, \rho_r \rightrightarrows \mathrm{e}_r$	elaboration of application spines
$\sigma \rightrightarrows \mathrm{s}$	elaboration of types
$\Gamma \rightrightarrows \mathrm{G}$	elaboration of typing contexts

ELAB-GEN

$$\frac{\begin{array}{c}\Gamma, \overline{a} \vdash e \Leftarrow \rho[\overline{\tau} \,/\, \overline{b}] \rightrightarrows \mathrm{e} \\ \overline{\tau \rightrightarrows \mathrm{t}} \qquad \rho \rightrightarrows \mathrm{r} \\ \mathit{fv}(\overline{\tau}) \subseteq \mathit{dom}(\Gamma, \overline{a})\end{array}}{\Gamma \vdash^{\forall} e \Leftarrow \forall \overline{a}.\exists \overline{b}.\rho \rightrightarrows \Lambda\overline{a}.\mathbf{pack}\,\overline{\mathrm{t}}, \mathrm{e}\ \mathbf{as}\ \exists\overline{b}.\mathrm{r}}$$

ELAB-IABS

$$\frac{\begin{array}{c}\overline{a}\ \mathbf{fresh} \\ \Gamma, x{:}\tau \vdash e \Rightarrow \rho \rightrightarrows \mathrm{e} \\ \mathit{fv}(\tau) \subseteq \mathit{dom}(\Gamma) \\ \rho' = \rho[\overline{a} \,/\, \lfloor \rho \rfloor_x] \qquad \tau \rightrightarrows \mathrm{t} \\ \rho \rightrightarrows \mathrm{r} \qquad \rho' \rightrightarrows \mathrm{r}'\end{array}}{\Gamma \vdash \lambda x.e \Rightarrow \tau \to \exists \overline{a}.\rho' \rightrightarrows \lambda x{:}\mathrm{t}.\mathbf{pack}\,\lfloor \mathrm{r} \rfloor_x, \mathrm{e}\ \mathbf{as}\ \exists\overline{a}.\mathrm{r}'}$$

ELAB-APP

$$\frac{\begin{array}{c}\Gamma \vdash_h h \Rightarrow \sigma \rightrightarrows \mathrm{h} \\ \Gamma \vdash^{\mathsf{inst}} \mathrm{h} : \sigma \rightrightarrows \mathrm{h} \,;\, \overline{\pi} \rightsquigarrow \overline{\sigma} \,;\, \rho_r \rightrightarrows \mathrm{e}_r\end{array}}{\Gamma \vdash h\,\overline{\pi} \Leftrightarrow \rho_r \rightrightarrows \mathrm{e}_r}$$

ELAB-IARG

$$\frac{\begin{array}{c}\Gamma \vdash^{\forall} e' \Leftarrow \sigma_1 \rightrightarrows \mathrm{e}' \\ \Gamma \vdash^{\mathsf{inst}} e\,e' : \sigma_2 \rightrightarrows \mathrm{e}\,\mathrm{e}' \,;\, \overline{\pi} \rightsquigarrow \overline{\sigma} \,;\, \rho_r \rightrightarrows \mathrm{e}_r\end{array}}{\Gamma \vdash^{\mathsf{inst}} e : (\sigma_1 \to \sigma_2) \rightrightarrows \mathrm{e} \,;\, e', \overline{\pi} \rightsquigarrow \sigma_1, \overline{\sigma} \,;\, \rho_r \rightrightarrows \mathrm{e}_r}$$

ELAB-IEXIST

$$\frac{\Gamma \vdash^{\mathsf{inst}} e : \epsilon[\lfloor e : \exists a.\epsilon \rfloor \,/\, a] \rightrightarrows \mathbf{open}\,\mathrm{e} \,;\, \overline{\pi} \rightsquigarrow \overline{\sigma} \,;\, \rho_r \rightrightarrows \mathrm{e}_r}{\Gamma \vdash^{\mathsf{inst}} e : \exists a.\epsilon \rightrightarrows \mathrm{e} \,;\, \overline{\pi} \rightsquigarrow \overline{\sigma} \,;\, \rho_r \rightrightarrows \mathrm{e}_r}$$

ELAB-IRESULT

$$\frac{}{\Gamma \vdash^{\mathsf{inst}} e : \rho_r \rightrightarrows \mathrm{e}_r \,;\, [\,] \rightsquigarrow [\,] \,;\, \rho_r \rightrightarrows \mathrm{e}_r}$$

Fig. 8. Judgments and selected rules for elaborating from $\mathbb{X}$ into $\mathbb{FX}$.

The rules in Figure 8 allow packing multiple existentials at once, when given a list of types as the first argument to **pack**; see rules ELAB-GEN and ELAB-IABS. Rule ELAB-GEN checks a surface expression e against an expected type $\forall \overline{a}.\exists \overline{b}.\rho$. We see that the result of elaboration uses nested Λ-abstractions and our nested **pack** notation to produce an $\mathbb{FX}$ expression that has the desired type. Rule ELAB-IABS echoes rule IABS, producing an $\mathbb{FX}$ expression with **pack**s necessary to accommodate any projections that mention the bound variable x; recall the special treatment of such projections from Section 4.2.3.

Rule ELAB-APP elaborates the head h to h, and then calls the $\vdash^{\mathsf{inst}}$ judgment. This judgment takes the elaborated h as an *input* (despite its appearance on the right of a $\rightrightarrows$). This input of an elaborated expression is built up as the application spine is checked, to be returned in rule ELAB-IRESULT. In order to build this elaborated expression as we go, rule ELAB-IARG elaborates arguments, in contrast to our original rule IArg; rule Elab-App then no longer needs to check these arguments in a second pass.[14] Rule Elab-IExist is the place where **open** is introduced, as it open an expression with an existential type.

The omitted rules all appear in the appendixand broadly follow the pattern set here.

6.1 Tweaking the IExist Rule

In the instantiation judgment for the surface language (Figure 5), rule IExist opens existentials. That is, given an expression e with an existential type $\exists\, a.\epsilon$, it infers for e the type resulting from replacing the type variable with the projection $\lfloor e : \exists\, a.\epsilon \rfloor$. However, these projections pose a problem during the elaboration process. Specifically, if we have an application $e_1\ e_2$ such that e_1 expects an argument whose type mentions $\lfloor e_0 : \epsilon \rfloor$—and e_2 indeed has a type mentioning $\lfloor e_0 : \epsilon \rfloor$—we cannot be sure that the application remains well-typed after elaboration. After all, type-checking in $\mathbb{X}$ is non-deterministic, given the way it guesses instantiations and the types of λ-bound variables. Another wrinkle is that $\lfloor e_0 : \epsilon \rfloor$ might appear under binders, making it even easier for type inference to come to two different conclusions when computing $\Gamma \vdash^{\forall} e_0 \Leftarrow \epsilon$.

There are two approaches to fix this problem: we can require our elaboration process to be deterministic, or we can modify rule IExist to make sure that projections in the surface language actually use pre-elaborated core expressions. We take the latter approach, as it is simpler and more direct. However, we discuss later in this section the possible disadvantages of this choice, and a route to consider the first one.

Accordingly, we now introduce the following new IExistCore and rule LetCore rules, replacing rules IExist and rule Let:

$$\textsc{IExistCore}\quad \frac{\Gamma \vdash^{\forall} e \Leftarrow \exists\, a.\epsilon \rightrightarrows \mathsf{e} \qquad \Gamma \vdash^{\mathsf{inst}} e : \epsilon[\lfloor \mathsf{e} \rfloor / a]\ ;\ \overline{\pi} \rightsquigarrow \overline{\sigma}\ ;\ \rho_r}{\Gamma \vdash^{\mathsf{inst}} e : \exists\, a.\epsilon\ ;\ \overline{\pi} \rightsquigarrow \overline{\sigma}\ ;\ \rho_r}$$

$$\textsc{LetCore}\quad \frac{\Gamma \vdash e_1 \Rightarrow \rho_1 \rightrightarrows \mathsf{e}_1 \qquad \overline{a} = fv(\rho_1) \backslash dom(\Gamma) \qquad \Gamma, x{:}\forall\, \overline{a}.\rho_1 \vdash e_2 \Leftrightarrow \rho_2}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \Leftrightarrow \rho_2[\Lambda \overline{a}.\mathsf{e}_1 / x]}$$

Fig. 9. Updated rules to support $\mathbb{FX}$ expressions in $\mathbb{X}$ types

Now, the elaboration process $\tau \rightrightarrows \mathsf{t}$ is indeed deterministic, making $\rightrightarrows$ a function on types τ and contexts Γ. Having surmounted this hurdle, elaboration largely very straightforward.

6.2 A Different Approach

We may want to refrain from using core expressions inside of projections, because doing so introduces complexity for the programmer who is not otherwise exposed to the core language. To wit, $\mathbb{X}$ would keep using projections of the form $\lfloor e : \epsilon \rfloor$, where we understand that $\Gamma \vdash^{\forall} e \Leftarrow \epsilon$ in the ambient context Γ, while $\mathbb{FX}$ uses the form $\lfloor \mathsf{e} \rfloor$.

It is vitally important that, if our surface-language typing rules accept a program, the elaborated version of that program is type-correct. (We call this property *soundness*; it is Theorem 7.1.) Yet, if elaboration of types is non-deterministic, we will lose this property, as explained above.

[14] Knowledgeable readers will wonder how this new treatment interacts with the Quick-Look algorithm, which critically depends on waiting to type-check arguments *after* a quick look at the entire argument spine. The solution is to be lazy: the elaborated is not needed until after all arguments have been checked. Accordingly, we could, for example, use a mutable cell to hold the elaborated expression, and then fill in this cell only during the second pass. Our formal presentation here need not worry about this technicality, however.

This alternative approach is simply to *assume* that elaboration is deterministic. Doing so is warranted because, in practice, a type-checker implementation will proceed deterministically—it seems far-fetched to think that a real type-checker would choose different types for the same expression and expected type, if any. In essence, a deterministic elaborator means that we can consider $\lfloor e : \epsilon \rfloor$ as a proxy for $\lfloor \mathrm{e} \rfloor$. The first is preferable to programmers because it is written in the language they program in. However, a type-checker implementation may choose to use the latter, and thus avoid the possibility of unsoundness from arising out of a non-deterministic elaborator.

7 ANALYSIS

The surface language $\mathbb{X}$ allows us to easily manipulate existentials in a λ-calculus while delegating type consistency to an explicit core language $\mathbb{FX}$. The following theorems establish the soundness of this approach, via the elaboration transformation $\rightrightarrows$, as well as the general expressivity and consistency of our bidirectional type system.

7.1 Soundness

If our surface language is to be type safe, we must know that any term accepted in the surface language corresponds to a well-typed term in the core language:

Theorem 7.1 (Soundness).

(1) *If* $\Gamma \vdash^{\forall} e \Leftarrow \sigma \rightrightarrows \mathrm{e}$, *then* $\mathrm{G} \vdash \mathrm{e} : \mathrm{s}$, *where* $\Gamma \rightrightarrows \mathrm{G}$ *and* $\sigma \rightrightarrows \mathrm{s}$.
(2) *If* $\Gamma \vdash e \Rightarrow \rho \rightrightarrows \mathrm{e}$, *then* $\mathrm{G} \vdash \mathrm{e} : \mathrm{r}$, *where* $\Gamma \rightrightarrows \mathrm{G}$ *and* $\rho \rightrightarrows \mathrm{r}$.
(3) *If* $\Gamma \vdash e \Leftarrow \rho \rightrightarrows \mathrm{e}$, *then* $\mathrm{G} \vdash \mathrm{e} : \mathrm{r}$, *where* $\Gamma \rightrightarrows \mathrm{G}$ *and* $\rho \rightrightarrows \mathrm{r}$.

Furthermore, in order to eliminate the possibility of a trivial elaboration scheme, we would want the elaborated term to behave like the surface-language one. We capture this property in this theorem:

Theorem 7.2 (Elaboration erasure).

(1) *If* $\Gamma \vdash^{\forall} e \Leftarrow \sigma \rightrightarrows \mathrm{e}$, *then* $|e| = |\mathrm{e}|$.
(2) *If* $\Gamma \vdash e \Rightarrow \rho \rightrightarrows \mathrm{e}$, *then* $|e| = |\mathrm{e}|$.
(3) *If* $\Gamma \vdash e \Leftarrow \rho \rightrightarrows \mathrm{e}$, *then* $|e| = |\mathrm{e}|$.

This theorem asserts that, if we remove all type annotations and applications, the $\mathbb{X}$ expression is the same as the $\mathbb{FX}$ one.

7.2 Conservativity

Not only do we want our $\mathbb{X}$ programs to be sound, but we also want $\mathbb{X}$ to be a comfortable language to program in. As our language is an extension of Hindley-Milner, we know that all the conveniences programmers are used to in that setting carry over here.

Theorem 7.3 (Conservative extension of Hindley-Milner). *If e has no type arguments or type annotations, and Γ, e, τ, σ contain no existentials, then:*

(1) $(\Gamma \vdash_{HM} e : \tau)$ *implies* $(\Gamma \vdash e \Rightarrow \tau)$
(2) $(\Gamma \vdash_{HM} e : \sigma)$ *implies* $(\Gamma \vdash^{\forall} e \Leftarrow \sigma)$

where $\vdash_{HM}$ denotes typing in the Hindley-Milner type system, as described by Clément et al. [1986, Figure 3].

7.3 Stability

The following theorems denote stability properties [Bottu and Eisenberg 2021]. In other words, they ensure that small user-written transformations do not change drastically the static semantics

Theorem 7.4 tells us that expanding out a well-typed **let** remains well typed. However, if we selectively expand a repeated **let**, a larger expression may become ill typed. Suppose we have $f :: Int \to \exists a.\ (a, a \to Int)$ and write $(snd\ (f\ (\mathbf{let}\ x = 5\ \mathbf{in}\ x{+}x)))\ (fst\ (f\ (\mathbf{let}\ x = 5\ \mathbf{in}\ x{+}x)))$. That expression is a well-typed Int. However, if we inline only one of the **let**s, to $(snd\ (f\ (5+5)))\ (fst\ (f\ (\mathbf{let}\ x = 5\ \mathbf{in}\ x+x)))$, we now have an ill-typed expression. The problem is that our language uses a very fine-grained expression equality relation: just α-equivalence. Accordingly, $\mathbf{let}\ x = 5\ \mathbf{in}\ x + x$ and $5 + 5$ are considered distinct, and when these expressions appear in types (via existential projections), the types are different.

The solution is straightforward, if not entirely lightweight: extend the expression equality relation. Doing so would require a more explicit treatment of equality in our type inference algorithm (in particular, rule App of Figure 4 would need to invoke the equality relation), as well as additions to $\mathbb{FX}$ to accommodate this new development. It is not clear whether the added expressiveness are worth the complexity cost, and so we kept our equivalence relationship simple for ease of presentation.

Aside 2. Selective **let**-inlining sometimes causes trouble

of our programs. The **let**-inlining property is specifically permitted by our approach to existentials, and it is a major feature of our type system.

Theorem 7.4 (**let**-inlining). *If x is free in e_2 then:*

$$\begin{array}{lcl}(\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \Rightarrow \rho) & \textit{implies} & (\Gamma \vdash e_2[e_1 / x] \Rightarrow \rho) \\ (\Gamma \vdash^{\forall} \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \Leftarrow \sigma) & \textit{implies} & (\Gamma \vdash^{\forall} e_2[e_1 / x] \Leftarrow \sigma)\end{array}$$

Interestingly, the system we present here does not support a small generalization of the **let**-inlining property, as we explore in Aside 2.

This next theorem tells us that the order variables appear in an existential quantification does not affect usage sites:

Theorem 7.5 (Order of Quantification does not matter). *Let ρ' (resp. σ') be two types that differ from ρ (resp. σ) only by the ordering of quantified type variables in their (eventual) existential types. Then:*

(1) $(\Gamma \vdash e \Rightarrow \rho)$ *if and only if* $(\Gamma \vdash e \Rightarrow \rho')$
(2) $(\Gamma \vdash^{\forall} e \Leftarrow \sigma)$ *if and only if* $(\Gamma \vdash^{\forall} e \Leftarrow \sigma')$

Lastly, this theorem tells us that extra, redundant type annotations do not disrupt typability:

Theorem 7.6 (Synthesis implies checking). *If $\Gamma \vdash e \Rightarrow \rho$ then $\Gamma \vdash e \Leftarrow \rho$.*

8 INTEGRATING WITH TODAY'S GHC AND QUICK LOOK

We envision integrating our design into GHC, allowing Haskell programmers to use existential types in their programs. Accordingly, we must consider how our work fits with GHC's latest type-inference algorithm, dubbed Quick Look [Serrano et al. 2020]. The structure behind our inference algorithm—with heads applied to lists of arguments instead of nested applications—is based directly on Quick Look, and it is straightforward to extend our work to be fully backward-compatible with that design. Indeed, our extension is essentially orthogonal to the innovations of impredicative type inference in the Quick Look algorithm.

$\boxed{\Gamma \vdash^{\forall} e \Leftarrow \sigma}$ *(Universal type checking)*

$$\textsc{GenImpredicative}\quad \frac{\overline{\kappa}\ \textbf{fresh} \qquad \rho' = \rho[\overline{\kappa}\,/\,\overline{b}] \qquad \Gamma, \overline{a} \vdash_{\lightning} e : \rho' \rightsquigarrow \Theta \qquad \rho'' = \Theta\,\rho' \qquad dom\,(\theta) = fiv\,(\rho'') \qquad \Gamma, \overline{a} \vdash e \Leftarrow \theta\,\rho''}{\Gamma \vdash^{\forall} e \Leftarrow \forall\,\overline{a}.\exists\,\overline{b}.\rho}$$

Fig. 10. Allowing impredicative instantiation in the $\vdash^{\forall}$ judgment

It would take us too far afield from our primary goal—describing type inference for existential types—to explain the details of Quick Look here. We thus build on the text already written by Serrano et al. [2020]; readers uninterested in the details may safely skip the rest of this section.

Serrano et al. [2020] explains their algorithm progressively, by stating in their Figures 3 and 4 a baseline system. That baseline also effectively serves as our baseline here. Then, in their Figure 5, the authors add a few new premises to specific rules, along with judgments those premises refer to. Given this modular presentation, we can adopt the same changes: their rule IARG is our rule IArg, and their rule APP-$\Downarrow$ is our rule App. The only wrinkle in merging these systems is that their presentation uses a notion of *instantiation variable*, which Serrano et al. write as κ. An instantiation variable is allowed to unify with a polytype, in contrast to an ordinary unification variable, which must unify with a monotype. Given that impredicative instantiation is not a primary goal of our work, we choose not to use this approach in our main formal presentation, instead preferring the more conventional idiom of using guessed τ-types. However, in order to integrate inferred existentials with Quick Look impredicativity, we must explicitly use instantiation variables in the rule below.

Since we have a more elaborate notion of polytype, one rule needs adjustment in our system: the rule implementing the $\Gamma \vdash^{\forall} e \Leftarrow \sigma$ judgment, rule Gen. That rule skolemizes (makes fresh constants out of) the variables universally quantified in σ and guesses $\overline{\tau}$ to instantiate the existentially quantified variables. In order to allow these instantiations to be impredicative, we must modify the rule, as in Figure 10.

This rule follows broadly the pattern from rule Gen, but using instantiation variables $\overline{\kappa}$ instead of guessing $\overline{\tau}$. The third premise invokes the Quick Look judgment $\vdash_{\lightning}$ [Serrano et al. 2020, Figure 5] to generate a substitution Θ. Such a substitution Θ maps instantiation variables κ to polytypes σ; by contrast, a substitution θ includes only monotypes τ in its codomain. The next two premises of rule GenImpredicative apply the Θ substitution, and then use θ to eliminate any remaining instantiation variables κ: the $fiv(\rho'')$ extracts all the *f*ree *i*nstantiation *v*ariables in ρ''. Note that the range of θ appears unconstrained here; the types in its range are guessed, just like the $\overline{\tau}$ in rule Gen.

With this one new rule—along with the changes evident in Figure 5 of Serrano et al.—our system supports impredicative type inference, and is a conservative extension of their algorithm.

9 DISCUSSION

We have described how our inference algorithm allows users to program with existentials while avoiding the need to thinking about packing and unpacking. Here, we review some subtleties that arise as our approach encounters more practical settings.

9.1 No Declarative (Non-syntax-directed) System with Existentials

When we first set out to understand type inference with existentials better, our goal was to develop a type system with existential types, unguided type inference (no additional annotation obligations for the programmer), and principal types. Our assumption was that if this is possible with universal quantification [Hindley 1969; Milner 1978], it should also be possible for existential quantification. Unfortunately, it seems such a design is out of reach.

To see why, consider $f\ b = \textbf{if}\ b\ \textbf{then}\ (1, \lambda y \rightarrow y + 1)\ \textbf{else}\ (\mathit{True}, \lambda z \rightarrow 1)$. We can see that f can be assigned one of two different types:

(1) $\mathit{Bool} \rightarrow \exists a.\, (a, \mathit{Int} \rightarrow \mathit{Int})$
(2) $\mathit{Bool} \rightarrow \exists a.\, (a, a \rightarrow \mathit{Int})$

Neither of these types is more general than the other, and neither seems likely to be ruled out by straightforward syntactic restrictions (such as the Hindley-Milner type system's requirement that all universal quantification be in prenex form).

One possible approach to inference for a definition like f is to use an *anti-unification* [Pfenning 1991] algorithm to relate the types of $(1, \lambda y \rightarrow y + 1)$ and $(\mathit{True}, \lambda z \rightarrow 1)$: infer the former to have type $(\mathit{Int}, \mathit{Int} \rightarrow \mathit{Int})$ and the latter to have type $(\mathit{Bool}, \alpha \rightarrow \mathit{Int})$ for some unknown type α. The goal then is to find some type τ such that τ can instantiate to either of these two types: this is anti-unification. The problem is, in this case, α: we get different results depending on whether α becomes Int or Bool.

We might imagine a way of choosing between the two hypothetical types for f, above, but any such restriction would break the desired symmetry and elegance of a declarative system that allows arbitrary generalization and specialization. Instead, we settle for the practical, predictable bidirectional algorithm presented in this paper, leaving the search for a more declarative approach as an open problem—one we think unlikely to have a satisfying solution.

9.2 Class Constraints on Existentials

The algorithm we present in this paper works with a typing context storing the types of bound variables. In full Haskell, however, we also have a set of constraint assumptions, and accepting some expressions requires proving certain constraints. A type system with these assumptions and obligations is often called a *qualified type system* [Jones 1992]. Our extension to support both universal and existential qualified types is in Figure 11.

This extension introduces type classes C and constraints Q. Constraints are applied type classes (like $\mathit{Show}\ \mathit{Int}$), and perhaps others; the details are immaterial. Instead, we refer to an abstract logical entailment relation $\Vdash$, which relates assumptions and the constraints they entail. Universally quantified types σ can now require proving a constraint: to use $e : Q \Rightarrow \sigma$, the constraint Q must hold. Existentially quantified types ϵ can now provide the proof of a constraint: the expression $e : Q \wedge \epsilon$ contains evidence that Q holds. Assumed constraints appear in contexts Γ.[15]

The surprising feature here is that we have a new form of assumption, $\lfloor e : \epsilon \rfloor$. This assumption is allowed only when ϵ has the form $Q \wedge \epsilon'$; the assumed constraint is Q. However, by including the expression e that proves Q in the context, we remember how to compute Q when it is required.

9.2.1 *Static Semantics.*

Examining the typing rules, we see rule GenQualified assumes Q_1 as a given (following the usual treatment of givens in qualified type systems) and also assumes an arbitrary list of projections $\overline{\lfloor e : \epsilon \rfloor}$. This arbitrary assumption is quite like how rule Gen assumes

[15] Other presentations of qualified type systems frequently have a judgment that looks like $P \mid \Gamma \vdash e : \rho$, or similar, with a separate set of logical assumptions P. Because our assumptions may include expressions, we must mix the logical assumptions with variable assumptions right in the same context Γ.

$$
\begin{array}{rcll}
C & ::= & \ldots & \text{type class} \\
Q & ::= & C\,\overline{\tau} \mid \ldots & \text{constraint} \\
\sigma & ::= & \epsilon \mid \forall\, a.\sigma \mid Q \Rightarrow \sigma & \text{universally quantified type} \\
\epsilon & ::= & \rho \mid \exists\, b.\epsilon \mid Q \wedge \epsilon & \text{existentially quantified type} \\
\Gamma & ::= & \emptyset \mid \Gamma, a \mid \Gamma, x{:}\sigma \mid \Gamma, Q \mid \Gamma, \lfloor e : \epsilon \rfloor & \text{typing context} \\
 & & \Gamma \Vdash Q & \text{logical entailment}
\end{array}
$$

$$
\textsc{GenQualified}\quad
\frac{\begin{array}{c}\Gamma' = \Gamma, \overline{a}, Q_1, \overline{\lfloor e : Q \wedge \epsilon \rfloor} \\ \overline{\Gamma' \vdash^{\forall} e \Leftarrow Q \wedge \epsilon} \qquad \overline{e \in e_0} \\ \Gamma' \vdash e_0 \Leftarrow \rho[\overline{\tau} \,/\, \overline{b}] \\ \Gamma' \Vdash Q_2[\overline{\tau} \,/\, \overline{b}]\end{array}}{\Gamma \vdash^{\forall} e_0 \Leftarrow (\forall\,\overline{a}.Q_1 \Rightarrow \exists\,\overline{b}.Q_2 \wedge \rho)}
$$

$$
\textsc{IGiven}\quad
\frac{\begin{array}{c}\Gamma \vdash^{\text{inst}} e : \epsilon \,;\, \overline{\pi} \rightsquigarrow \overline{\sigma} \,;\, \rho_r \\ \lfloor e : Q \wedge \epsilon \rfloor \in \Gamma\end{array}}{\Gamma \vdash^{\text{inst}} e : Q \wedge \epsilon \,;\, \overline{\pi} \rightsquigarrow \overline{\sigma} \,;\, \rho_r}
\qquad
\textsc{IWanted}\quad
\frac{\begin{array}{c}\Gamma \vdash^{\text{inst}} e : \sigma \,;\, \overline{\pi} \rightsquigarrow \overline{\sigma} \,;\, \rho_r \\ \Gamma \Vdash Q\end{array}}{\Gamma \vdash^{\text{inst}} e : Q \Rightarrow \sigma \,;\, \overline{\pi} \rightsquigarrow \overline{\sigma} \,;\, \rho_r}
$$

Fig. 11. Type system extension to support existentially packed class constraints

types $\overline{\tau}$ to replace the existential variables $\overline{b}$. To prevent the type system from working in an unbounded search space for assumptions to make, the expressions e must be sub-expressions of our checked expression e_0.

The instantiation judgment $\vdash^{\text{inst}}$ must also accommodate constraints. When, in rule IGiven, it comes across an expression whose type includes a packed assumption $Q \wedge \epsilon$, it checks to make sure that assumption was included in Γ. The design here requiring an arbitrary guess of assumptions, only to validate the guess later, is merely because our presentation is somewhat declarative. By contrast, an implementation would work by emitting constraints and solving them (that is, computing $\Vdash$) later [Pottier and Rémy 2005]; when the constraint-generation pass encounters an expression of type $Q \wedge \epsilon$, it simply emits the constraint as a given. Rule IWanted is a straightforward encoding of the usual behavior of qualified types, where the usage of an expression of type $Q \Rightarrow \sigma$ requires proving Q.

9.2.2 Dynamic Semantics. An interesting new challenge with packed class constraints is that class constraints are not erasable. In practice, a function *pretty* of type *Pretty* *a* $\Rightarrow$ *a* $\rightarrow$ *String* (§2.3) takes *two* runtime arguments: a *dictionary* [Hall et al. 1996] containing implementations of the methods in *Pretty*, as well as the actual, visible argument of type *a*. When this dictionary comes from an existential projection, the expression producing the existential will have to be evaluated.

For example, suppose we have *mk* :: *Bool* $\rightarrow$ $\exists$*a*. *Pretty* *a* $\wedge$ *a* and call *pretty* (*mk* *True*). Calling *pretty* requires passing the dictionary giving the the implementation of the function at the specific type *pretty* is instantiated at ($\lfloor$*mk* *True* :: $\exists$*a*. *Pretty* *a* $\wedge$ *a*$\rfloor$, in this case). Getting this dictionary requires evaluating *mk* *True*. Naïvely, this means *mk* *True* would be evaluated *twice*. This makes some sense if we think of $Q \wedge \epsilon$ as the type of pairs of a dictionary for Q and the inhabitant of ϵ: the naïve interpretation of *pretty* (*mk* *True*) thus is like calling *pretty* (*fst* (*mk* *True*)) (*snd* (*mk* *True*)). We do not address how to do better here, as standard optimization techniques can apply to improve the potential repeated work. Once again, purity works to our advantage here, in that we can be assured that commoning up the calls to *mk* *True* does not introduce (or eliminate) effects.

9.3 Relevance and Existentials

One of the primary motivations for this work is to set the stage for an eventual connection between Liquid Haskell [Vazou et al. 2014] and the rest of Haskell's type system. A Liquid Haskell refinement type is exemplified by $\{v :: Int \mid v \geqslant 0\}$; any element of such a type is guaranteed to be non-negative. Yet what would it mean to have a function *return* such a type? To be concrete, let us imagine $mk :: Bool \to \{v :: Int \mid v \geqslant 0\}$. This function would return a value v of type Int, along with a proof that $v \geqslant 0$: this is a dependent pair, or an existential package. Thus, we can rephrase the type of mk to be $Bool \to \exists(v :: Int).\ Proof\ (v \geqslant 0)$, where $Proof\ q$ encodes a proof of the logical property q.

However, our new form of existential is different than the others considered in this paper. Here, the relevant part is the *first* component, not the second. That is, we want to be able to project out $v :: Int$ at runtime, discarding the compile-time proof that $v \geqslant 0$.

The core language presented in this paper cannot, without embellishment, support relevant first components of existentials. In other words, $\lfloor e : \epsilon \rfloor$ is always a compile-time type, never a runtime term. Nevertheless, existing approaches to deal with relevance will work in this new setting. Haskell's $\forall$ construct universally quantifies over an irrelevant type. Yet, work on dependent Haskell [Eisenberg 2016; Gundry 2013; Weirich et al. 2017] shows how we can make a similar, relevant construct. Similar approaches could work in a core language modeled on $\mathbb{FX}$. Indeed, other dependently typed languages, such as Coq, Agda, and Idris support existential packages with relevant dependent components.

The big step our current work brings to this story is type inference. Whether relevant or not, we would still want existential packages to be packed and unpacked without explicit user direction, and we would still want type inference to have the properties of the algorithm presented in this paper. In effect, the choice of relevance of the dependent component is orthogonal to the concerns in this paper. We are thus confident that our approach would work in a setting with relevant types.

10 RELATED WORK

There is a long and rich body of literature informing our knowledge of existential types. We review some of the more prominent work here.

History. Existential types were present from the beginning in the design of polymorphic programming languages, present in Girard's System F [Girard 1972] and independently discovered by Reynolds [1974], though in a less expressive form. Mitchell and Plotkin [1988] recognized the ability of existential types to model abstract datatypes and remarked on their connection with the Σ-types of Martin-Löf type theory [Martin-Löf 1975]. They proposed an elimination form, called *abstype*, that is equivalent to the now standard **unpack**.

Cardelli and Leroy [1990] compared Mitchell and Plotkin's **unpack** based approach to various calculi with projection-based existentials. Their "calculus with a dot notation" includes the ability for the type language to project the type component from term variables of an existential type. At the end of the report (Section 4), they generalize to allow arbitrary expressions in projections. It is this language that is most similar to our core language. They also note a number of examples that are expressible only in this language.

Integration with type inference. Full type checking and type inference for domain-free System F with existential types is known to be undecidable [Nakazawa and Tatsuta 2009; Nakazawa et al. 2008]. As a result, several language designers have used explicit forms such as datatype declarations or type annotations to extend their languages with existential types.

The datatype-based version of existentials found in GHC was first suggested by Perry [1991] and implemented in Hope+. It was formalized by Läufer and Odersky [1994] and implemented in the Caml Light compiler for ML, along with the Haskell B compiler [Augustsson 1994].

The Utrecht Haskell Compiler (UHC) also supports a version of existential type [Dijkstra 2005], in a form that does not require the explicit connection to datatypes found in GHC. As in this work, values of existential types can be opened in place, without the use of an **unpack** term. However, unlike here, UHC generates a fresh type variable for the abstracted type with each use of **open**. As a result, UHC does not need the form of dependent types that we propose, but also cannot express some of the examples allowed by our system (§3.3).

Leijen [2006] describes an extension of MLF [Le Botlan and Rémy 2003] with first-class existential types. Like this work, programmers never needed to add explicit **pack** or **unpack** expressions. However, because the type system was based on MLF, polymorphic types include instantiation constraints and the type-inference algorithm is very different from that used by GHC. In contrast, our work requires only a small extension of GHC's most recent implementation of first-class polymorphism. Furthermore, Leijen does not describe a translation from his source language to an explicitly typed core language; a necessary implementation step for GHC.

Dunfield and Krishnaswami [2019] extend a bidirectional type system with indexes in existential types in order to support GADTs. As in this work, the introduction and elimination of existentials is implicit and determined by type annotations. Existentials are introduced via subsumption and eliminated via pattern matching. As a result, this type system has the same scoping limitations as one based on **unpack**.

In other contexts, if the domain of types that existentials are allowed to quantify over is restricted, more aggressive type inference is possible. For example, Tate et al. [2008] restrict existentials to hide only class types and develop a type-inference framework for a small object-oriented typed assembly language.

Module systems. This paper also relates to work on ML-style module systems. We do not summarize that field here but mention some papers that are particularly inspirational or relevant.

MacQueen [1986] noted the deficiencies of Mitchell and Plotkin [1988] with respect to expressing modular structure. This work proposed the original form of the ML module system as a dependent type system based on strong Σ-types. As in our system, modules support projections of the abstracted type and values. However, unlike this work, the ML module language supports additional type system features: a phase separation between the compile-time and runtime parts of the language, a treatment of generativity which determines when module expressions should and should not define new types, etc, as described in Harper and Pierce [2005]. We do not intend to use this type system to express modular structure.

F-ing modules [Rossberg et al. 2014] present a formalization of ML modules using existential types and a translation of a module language into System F_ω augmented with **pack** and **unpack**. Our approach is similar to theirs, in that we also use a translation of a surface language into our $\mathbb{FX}$. However, because the ML module system includes a phase separation, our concerns about strictness do not apply in that setting. As a result they can target the non-dependent language F_ω and use **unpack** as their elimination form. Rossberg [2015] extends the source language to a more uniform design while still retaining the translation to a non-dependent core calculus.

Montagu and Rémy [2009] present an extension of System F to compute *open* existential types. They introduce the idea of decomposing the usual explicit **pack** and **unpack** constructs of System F, and we were inspired by those ideas to design the type system of our implicit surface language with opened existentials. Interestingly, for a long time, it was unknown whether full abstraction could be achieved with strong existentials. Crary [2017] plugged this hole, proving Reynold's abstraction theorem for a module calculus based on strong Σ-types.

11 CONCLUSION

By leveraging strong existential types, we have presented a type-inference algorithm that can infer introduction and elimination sites for existential packages. Users can freely create and consume existentials with no term-level annotations. The type annotation burden is small, and it dovetails with programmers' current expectations around bidirectional type inference. The algorithm we present is designed to integrate well with GHC/Haskell's state-of-the-art approach to type inference, the Quick Look algorithm [Serrano et al. 2020].

In order to prove our approach sound, we include an elaboration into a type-safe core language, inspired by Cardelli and Leroy [1990] and supporting the usual progress and preservation proofs. This core language is a small extension on System FC, the current core language implemented within GHC, and thus is suitable for implementation.

Beyond just soundness, we prove that inlining a **let**-binding preserves types, a non-trivial property in a type system with inferred existential types. We also prove that our type-inference algorithm is a conservative extension of a basic Hindley-Milner type system.

We believe and hope that our forthcoming implementation within GHC—in active development at the time of writing—will enable programmers to verify more aspects of their programs, even when that verification requires the use of existential types. We also hope that this new feature will provide a way forward to integrate the user-facing success of Liquid Haskell with GHC's internal language and optimizer.

ACKNOWLEDGMENTS

The authors thank Neel Krishnaswami and Simon Peyton Jones for their collaboration and review, along with our anonymous reviewers. This material is based upon work supported by the National Science Foundation under Grant No. 1703835 and Grant No. 1704041. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Lennart Augustsson. 1994. *Haskell B. user's manual.* https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.50.5800&rep=rep1&type=pdf

Lennart Augustsson and Magnus Carlsson. 1999. An exercise in dependent types: A well-typed interpreter. (1999). http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.39.2895&rep=rep1&type=pdf Unpublished manuscript.

Franz Baader and Tobias Nipkow. 1998. *Term Rewriting and All That.* Cambridge University Press.

Gert-Jan Bottu and Richard A. Eisenberg. 2021. Seeking Stability by being Lazy and Shallow: Lazy and shallow instantiation is user friendly. In *ACM SIGPLAN Haskell Symposium.*

Luca Cardelli and Xavier Leroy. 1990. Abstract types and the dot notation. In *IFIP TC2 working conference on programming concepts and methods.* North-Holland, 479–504.

Dominique Clément, Thierry Despeyroux, Gilles Kahn, and Joëlle Despeyroux. 1986. A Simple Applicative Language: Mini-ML. In *Conference on LISP and Functional Programming* (Cambridge, Massachusetts, USA) *(LFP '86).* ACM.

Karl Crary. 2017. Modules, Abstraction, and Parametric Polymorphism. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL 2017).* Association for Computing Machinery, New York, NY, USA, 100–113. https://doi.org/10.1145/3009837.3009892

Luis Damas and Robin Milner. 1982. Principal Type-schemes for Functional Programs. In *Symposium on Principles of Programming Languages* (Albuquerque, New Mexico) *(POPL '82).* ACM.

Atze Dijkstra. 2005. *Stepping through Haskell.* Ph.D. Dissertation. Universiteit Utrecht.

Jana Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and Easy Bidirectional Typechecking for Higher-rank Polymorphism. In *International Conference on Functional Programming (ICFP '13).* ACM.

Jana Dunfield and Neelakantan R. Krishnaswami. 2019. Sound and Complete Bidirectional Typechecking for Higher-Rank Polymorphism with Existentials and Indexed Types. *Proc. ACM Program. Lang.* 3, POPL, Article 9 (Jan. 2019), 28 pages. https://doi.org/10.1145/3290322

Richard A. Eisenberg. 2016. *Dependent Types in Haskell: Theory and Practice.* Ph.D. Dissertation. University of Pennsylvania.

Richard A. Eisenberg. 2020. Stitch: The Sound Type-Indexed Type Checker (Functional Pearl). In *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell* (Virtual Event, USA) *(Haskell 2020).* Association for Computing Machinery, New York, NY, USA, 39–53. https://doi.org/10.1145/3406088.3409015

Richard A. Eisenberg, Joachim Breitner, and Simon Peyton Jones. 2018. Type Variables in Patterns. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell* (St. Louis, MO, USA) *(Haskell 2018)*. Association for Computing Machinery, New York, NY, USA, 94–105. https://doi.org/10.1145/3242744.3242753

Richard A. Eisenberg and Stephanie Weirich. 2012. Dependently Typed Programming with Singletons. In *ACM SIGPLAN Haskell Symposium*.

Jean-Yves Girard. 1972. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* Ph.D. Dissertation. Université Paris 7.

Adam Gundry. 2013. *Type Inference, Haskell and Dependent Types.* Ph.D. Dissertation. University of Strathclyde.

Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. 1996. Type Classes in Haskell. *ACM Trans. Program. Lang. Syst.* 18, 2 (March 1996).

Robert Harper and Benjamin C. Pierce. 2005. Design Considerations for ML-Style Module Systems. In *Advanced Topics in Types and Programming Languages*, Benjamin C. Pierce (Ed.). The MIT Press, 293–346.

J. Roger Hindley. 1969. The Principal Type-Scheme of an Object in Combinatory Logic. *Trans. Amer. Math. Soc.* 146 (1969).

William Alvin Howard. 1969. The Formulae-as-types Notion of Construction. (1969). https://www.dcc.fc.up.pt/~acm/howard2.pdf Dedicated to H. B. Curry on the occasion of his 80th birthday.

Mark P. Jones. 1992. A Theory of Qualified Types. In *Proceedings of the 4th European Symposium on Programming (ESOP '92)*. Springer-Verlag, Berlin, Heidelberg, 287–306.

Konstantin Läufer. 1996. Type classes with existential types. *Journal of Functional Programming* 6, 3 (1996), 485–518. https://doi.org/10.1017/S0956796800001817

Konstantin Läufer and Martin Odersky. 1994. Polymorphic type inference and abstract data types. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16, 5 (1994), 1411–1430.

Didier Le Botlan and Didier Rémy. 2003. ML^{F}: Raising ML to the power of System F. In *International Conference on Functional Programming*. ACM.

Daan Leijen. 2006. First-class polymorphism with existential types. (2006). Unpublished.

David B MacQueen. 1986. Using dependent types to express modular structure. In *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. 277–286.

Per Martin-Löf. 1975. An intuitionistic theory of types: Predicative part. In *Studies in Logic and the Foundations of Mathematics*. Vol. 80. Elsevier, 73–118.

Conor Thomas McBride. 2014. How to Keep Your Neighbours in Order. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP '14)*. ACM.

Robin Milner. 1978. A Theory of Type Polymorphism in Programming. *J. Comput. System Sci.* 17 (1978).

John C Mitchell and Gordon D Plotkin. 1988. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 10, 3 (1988), 470–502.

Stefan Monnier and David Haguenauer. 2010. Singleton types here, singleton types there, singleton types everywhere. In *Programming languages meets program verification (PLPV '10)*. ACM.

Benoît Montagu and Didier Rémy. 2009. Modeling Abstract Types in Modules with Open Existential Types. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. Association for Computing Machinery, New York, NY, USA, 354–365. https://doi.org/10.1145/1480881.1480926

Koji Nakazawa and Makoto Tatsuta. 2009. Type Checking and Inference for Polymorphic and Existential Types. In *Proceedings of the Fifteenth Australasian Symposium on Computing: The Australasian Theory - Volume 94* (Wellington, New Zealand) *(CATS '09)*. Australian Computer Society, Inc., AUS, 63–72.

Koji Nakazawa, Makoto Tatsuta, Yukiyoshi Kameyama, and Hiroshi Nakano. 2008. Undecidability of Type-Checking in Domain-Free Typed Lambda-Calculi with Existence. In *Computer Science Logic*, Michael Kaminski and Simone Martini (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 478–492.

Martin Odersky and Konstantin Läufer. 1996. Putting Type Annotations to Work. In *Symposium on Principles of Programming Languages (POPL '96)*. ACM.

Nigel Perry. 1991. *The implementation of practical functional programming languages.* Ph.D. Dissertation. Imperial College of Science, Technology and Medicine, University of London.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical type inference for arbitrary-rank types. *Journal of Functional Programming* 17, 1 (Jan. 2007).

F. Pfenning. 1991. Unification and anti-unification in the calculus of constructions. In *Proceedings 1991 Sixth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, Los Alamitos, CA, USA, 74,75,76,77,78,79,80,81,82,83,84,85. https://doi.org/10.1109/LICS.1991.151632

Frank Pfenning and Peter Lee. 1989. LEAP: A language with eval and polymorphism. In *TAPSOFT '89*, J. Díaz and F. Orejas (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 345–359.

Benjamin C. Pierce. 2002. *Types and Programming Languages.* MIT Press, Cambridge, MA.

François Pottier and Didier Rémy. 2005. The Essence of ML Type Inference. In *Advanced Topics in Types and Programming Languages*, Benjamin C. Pierce (Ed.). The MIT Press, 387–489.

John C. Reynolds. 1974. Towards a theory of type structure. In *Programming Symposium*, B. Robinet (Ed.). Lecture Notes in Computer Science, Vol. 19. Springer Berlin Heidelberg, 408–425.

Andreas Rossberg. 2015. 1ML – Core and Modules United (F-Ing First-Class Modules). In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada) *(ICFP 2015)*. Association for Computing Machinery, New York, NY, USA, 35–47. https://doi.org/10.1145/2784731.2784738

Andreas Rossberg, Claudio Russo, and Derek Dreyer. 2014. F-ing modules. *Journal of functional programming* 24, 5 (2014), 529–607.

Alejandro Serrano, Jurriaan Hage, Simon Peyton Jones, and Dimitrios Vytiniotis. 2020. A Quick Look at Impredicativity. *Proc. ACM Program. Lang.* 4, ICFP, Article 89 (Aug. 2020), 29 pages. https://doi.org/10.1145/3408971

Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. 2007. System F with type equality coercions. In *Types in languages design and implementation* (Nice, Nice, France) *(TLDI '07)*. ACM.

Ross Tate, Juan Chen, and Chris Hawblitzel. 2008. *A Flexible Framework for Type Inference with Existential Quantification.* Technical Report MSR-TR-2008-184. https://www.microsoft.com/en-us/research/publication/a-flexible-framework-for-type-inference-with-existential-quantification/

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton Jones. 2014. Refinement Types for Haskell. In *International Conference on Functional Programming* (Gothenburg, Sweden) *(ICFP '14)*. ACM.

Stephanie Weirich. 2018. Dependent Types in Haskell. Haskell eXchange keynote.

Stephanie Weirich, Antoine Voizard, Pedro Henrique Azevedo de Amorim, and Richard A. Eisenberg. 2017. A Specification for Dependent Types in Haskell. *Proc. ACM Program. Lang.* 1, ICFP, Article 31 (Aug. 2017), 29 pages. https://doi.org/10.1145/3110275

Hongwei Xi, Chiyan Chen, and Gang Chen. 2003. Guarded recursive datatype constructors. In *Principles of Programming Languages* (New Orleans, Louisiana, USA) *(POPL '03)*. ACM.

A ELABORATION RULES

We first extend the $\mathbb{FX}$ grammar to include arguments:

$$\mathrm{p} \quad ::= \quad \mathrm{e} \mid \mathrm{t} \quad \text{argument}$$

$\boxed{\Gamma \vdash^{\forall} e \Leftarrow \sigma \rightrightarrows \mathrm{e}}$ *(Elaboration for polymorphic expressions)*

Elab-Gen

$$\frac{\begin{array}{c}\Gamma, \overline{a} \vdash e \Leftarrow \rho[\overline{\tau} \,/\, \overline{b}] \rightrightarrows \mathrm{e} \\ \overline{\tau \rightrightarrows \mathrm{t}} \qquad \rho \rightrightarrows \mathrm{r} \\ fv(\overline{\tau}) \subseteq dom(\Gamma, \overline{a})\end{array}}{\Gamma \vdash^{\forall} e \Leftarrow \forall \overline{a}.\exists \overline{b}.\rho \rightrightarrows \Lambda \overline{a}.\mathbf{pack}\, \overline{\mathrm{t}}, \mathrm{e}\ \mathbf{as}\ \exists \overline{b}.\mathrm{r}}$$

$\boxed{\Gamma \vdash e \Rightarrow \rho \rightrightarrows \mathrm{e} \qquad \Gamma \vdash e \Leftarrow \rho \rightrightarrows \mathrm{e}}$ *(Elaboration for expressions)*

Elab-App

$$\frac{\begin{array}{c}\Gamma \vdash_h h \Rightarrow \sigma \rightrightarrows \mathrm{h} \\ \Gamma \vdash^{\mathsf{inst}} h : \sigma \rightrightarrows \mathrm{h} \,;\, \overline{\pi} \rightsquigarrow \overline{\sigma} \,;\, \rho_r \rightrightarrows \mathrm{e}_r\end{array}}{\Gamma \vdash h\,\overline{\pi} \Leftrightarrow \rho_r \rightrightarrows \mathrm{e}_r}$$

Elab-iAbs

$$\frac{\begin{array}{c}\overline{a}\ \mathbf{fresh} \\ \Gamma, x{:}\tau \vdash e \Rightarrow \rho \rightrightarrows \mathrm{e} \\ fv(\tau) \subseteq dom(\Gamma) \\ \rho' = \rho[\overline{a} \,/\, \lfloor \rho \rfloor_x] \qquad \tau \rightrightarrows \mathrm{t} \\ \rho \rightrightarrows \mathrm{r} \qquad \rho' \rightrightarrows \mathrm{r}'\end{array}}{\Gamma \vdash \lambda x.e \Rightarrow \tau \to \exists \overline{a}.\rho' \rightrightarrows \lambda x{:}\mathrm{t}.\mathbf{pack}\, \lfloor \mathrm{r} \rfloor_x, \mathrm{e}\ \mathbf{as}\ \exists \overline{a}.\mathrm{r}'}$$

Elab-cAbs

$$\frac{\begin{array}{c}\Gamma, x{:}\sigma_1 \vdash^{\forall} e \Leftarrow \sigma_2 \rightrightarrows \mathrm{e} \\ fv(\sigma_1) \subseteq dom(\Gamma) \\ \sigma_1 \rightrightarrows \mathrm{s}_1\end{array}}{\Gamma \vdash \lambda x.e \Leftarrow \sigma_1 \to \sigma_2 \rightrightarrows \lambda x{:}\mathrm{s}_1.\mathrm{e}}$$

Elab-LetCore

$$\frac{\begin{array}{c}\Gamma \vdash e_1 \Rightarrow \rho_1 \rightrightarrows \mathrm{e}_1 \\ \overline{a} = fv(\rho_1) \backslash dom(\Gamma) \\ \Gamma, x{:}\forall \overline{a}.\rho_1 \vdash e_2 \Leftrightarrow \rho_2 \rightrightarrows \mathrm{e}_2\end{array}}{\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \Leftrightarrow \rho_2[\Lambda \overline{a}.\mathrm{e}_1 \,/\, x] \rightrightarrows \mathbf{let}\ x = \Lambda \overline{a}.\mathrm{e}_1\ \mathbf{in}\ \mathrm{e}_2}$$

$\boxed{\Gamma \vdash_h h \Rightarrow \sigma \rightrightarrows \mathrm{h}}$ *(Elaboration for heads)*

Elab-Var

$$\frac{x{:}\sigma \in \Gamma}{\Gamma \vdash_h x \Rightarrow \sigma \rightrightarrows x}$$

Elab-Ann

$$\frac{\begin{array}{c}\Gamma \vdash^{\forall} e \Leftarrow \sigma \rightrightarrows \mathrm{e} \\ fv(\sigma) \subseteq dom(\Gamma)\end{array}}{\Gamma \vdash_h (e :: \sigma) \Rightarrow \sigma \rightrightarrows \mathrm{e}}$$

Elab-Infer

$$\frac{\Gamma \vdash e \Rightarrow \rho \rightrightarrows \mathrm{e}}{\Gamma \vdash_h e \Rightarrow \rho \rightrightarrows \mathrm{e}}$$

$\boxed{\Gamma \vdash^{\mathsf{inst}} e : \sigma \rightrightarrows \mathrm{e} \,;\, \overline{\pi} \rightsquigarrow \overline{\sigma} \,;\, \rho_r \rightrightarrows \mathrm{e}_r}$ *(Elaboration for instantiation)*

Elab-ITyArg

$$\frac{\begin{array}{c}\sigma' \rightrightarrows \mathrm{s}' \\ \Gamma \vdash^{\mathsf{inst}} e\,\sigma' : \sigma[\sigma' \,/\, a] \rightrightarrows \mathrm{e}\,\mathrm{s}' \,;\, \overline{\pi} \rightsquigarrow \overline{\sigma} \,;\, \rho_r \rightrightarrows \mathrm{e}_r\end{array}}{\Gamma \vdash^{\mathsf{inst}} e : \forall a.\sigma \rightrightarrows \mathrm{e} \,;\, \sigma', \overline{\pi} \rightsquigarrow \overline{\sigma} \,;\, \rho_r \rightrightarrows \mathrm{e}_r}$$

Elab-IArg

$$\frac{\begin{array}{c}\Gamma \vdash^{\forall} e' \Leftarrow \sigma_1 \rightrightarrows \mathrm{e}' \\ \Gamma \vdash^{\mathsf{inst}} e\,e' : \sigma_2 \rightrightarrows \mathrm{e}\,\mathrm{e}' \,;\, \overline{\pi} \rightsquigarrow \overline{\sigma} \,;\, \rho_r \rightrightarrows \mathrm{e}_r\end{array}}{\Gamma \vdash^{\mathsf{inst}} e : (\sigma_1 \to \sigma_2) \rightrightarrows \mathrm{e} \,;\, e', \overline{\pi} \rightsquigarrow \sigma_1, \overline{\sigma} \,;\, \rho_r \rightrightarrows \mathrm{e}_r}$$

$$\textsc{Elab-IAll}\quad \frac{\tau \rightrightarrows \mathrm{t} \qquad \overline{\pi} \neq \sigma', \overline{\pi}' \qquad \Gamma \vdash^{\mathsf{inst}} e : \sigma[\tau \,/\, a] \rightrightarrows \mathrm{e}\,\mathrm{t} \,;\, \overline{\pi} \rightsquigarrow \overline{\sigma} \,;\, \rho_r \rightrightarrows \mathrm{e}_r}{\Gamma \vdash^{\mathsf{inst}} e : \forall\, a.\sigma \rightrightarrows \mathrm{e} \,;\, \overline{\pi} \rightsquigarrow \overline{\sigma} \,;\, \rho_r \rightrightarrows \mathrm{e}_r}$$

$$\textsc{Elab-IExistCore}\quad \frac{\Gamma \vdash^{\mathsf{inst}} e : \epsilon[\lfloor \mathrm{e} \rfloor \,/\, a] \rightrightarrows \mathbf{open}\,\mathrm{e} \,;\, \overline{\pi} \rightsquigarrow \overline{\sigma} \,;\, \rho_r \rightrightarrows \mathrm{e}_r}{\Gamma \vdash^{\mathsf{inst}} e : \exists\, a.\epsilon \rightrightarrows \mathrm{e} \,;\, \overline{\pi} \rightsquigarrow \overline{\sigma} \,;\, \rho_r \rightrightarrows \mathrm{e}_r}
\qquad
\textsc{Elab-IResult}\quad \frac{}{\Gamma \vdash^{\mathsf{inst}} e : \rho_r \rightrightarrows \mathrm{e}_r \,;\, [\,] \rightsquigarrow [\,] \,;\, \rho_r \rightrightarrows \mathrm{e}_r}$$

$\boxed{\sigma \rightrightarrows \mathrm{s}}$ *(Elaboration for types)*

$$\textsc{ElabT-ForAll}\ \frac{\sigma \rightrightarrows \mathrm{s}}{\forall\, a.\sigma \rightrightarrows \forall\, a.\mathrm{s}}
\qquad
\textsc{ElabT-Exists}\ \frac{\epsilon \rightrightarrows \mathrm{t}}{\exists\, a.\epsilon \rightrightarrows \exists\, a.\mathrm{t}}
\qquad
\textsc{ElabT-Arrow}\ \frac{\sigma_1 \rightrightarrows \mathrm{s}_1 \qquad \sigma_2 \rightrightarrows \mathrm{s}_2}{\sigma_1 \to \sigma_2 \rightrightarrows \mathrm{s}_1 \to \mathrm{s}_2}
\qquad
\textsc{ElabT-Var}\ \frac{}{a \rightrightarrows a}
\qquad
\textsc{ElabT-ProjCore}\ \frac{}{\lfloor \mathrm{e} \rfloor \rightrightarrows \lfloor \mathrm{e} \rfloor}$$

$\boxed{\Gamma \rightrightarrows \mathrm{G}}$ *(Elaboration for contexts)*

$$\textsc{ElabC-Nil}\ \frac{}{\emptyset \rightrightarrows \emptyset}
\qquad
\textsc{ElabC-TyVar}\ \frac{\Gamma \rightrightarrows \mathrm{G}}{\Gamma, a \rightrightarrows \mathrm{G}, a}
\qquad
\textsc{ElabC-Var}\ \frac{\Gamma \rightrightarrows \mathrm{G} \qquad \sigma \rightrightarrows \mathrm{s}}{\Gamma, x{:}\sigma \rightrightarrows \mathrm{G}, x : \mathrm{s}}$$

In a small abuse of notation, we write (for example, in rule Elab-IAbs) a list of types in a **pack** construct to denote nested **pack**s. Formally, for e of type $\mathrm{r}[\overline{\mathrm{t}} \,/\, \overline{a}]$, with $\overline{\mathrm{t}} = \mathrm{t}_1 \ldots \mathrm{t}_n$ and $\overline{a} = a_1 \ldots a_n$, the construction is defined recursively by:

$$\mathbf{pack}\,\mathrm{t}_1 \ldots \mathrm{t}_n, \mathrm{e}\,\mathbf{as}\,\exists\, a_1 \ldots a_n.\mathrm{r} = \mathbf{pack}\,\mathrm{t}_1, (\mathbf{pack}\,\mathrm{t}_2 \ldots \mathrm{t}_n, \mathrm{e}\,\mathbf{as}\,\exists\, a_2 \ldots a_n.\mathrm{r}[\mathrm{t}_1 \,/\, a_1])\,\mathbf{as}\,\exists\, a_1\, a_2 \ldots a_n.\mathrm{r}$$

Define erasure on $\mathbb{X}$ terms by the following equations:

$$\begin{aligned}
|n| &= n \\
|x| &= x \\
|e \,::\, \sigma| &= |e| \\
|h\,\overline{\pi}, e| &= |h\,\overline{\pi}|\,|e| \\
|h\,\overline{\pi}, \sigma| &= |h\,\overline{\pi}| \\
|\lambda x.e| &= \lambda x.|e| \\
|\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2| &= \mathbf{let}\ x = |e_1|\ \mathbf{in}\ |e_2|
\end{aligned}$$

Theorem A.1 (Elaboration erasure (Theorem 7.2)).

(1) *If* $\Gamma \vdash^{\forall} e \Leftarrow \sigma \rightrightarrows \mathrm{e}$, *then* $|e| = |\mathrm{e}|$.
(2) *If* $\Gamma \vdash e \Rightarrow \rho \rightrightarrows \mathrm{e}$, *then* $|e| = |\mathrm{e}|$.
(3) *If* $\Gamma \vdash e \Leftarrow \rho \rightrightarrows \mathrm{e}$, *then* $|e| = |\mathrm{e}|$.
(4) *If* $\Gamma \vdash_h h \Rightarrow \sigma \rightrightarrows \mathrm{h}$, *then* $|h| = |\mathrm{h}|$.
(5) *If* $\Gamma \vdash^{\mathsf{inst}} e : \sigma \rightrightarrows \mathrm{e} \,;\, \overline{\pi} \rightsquigarrow \overline{\sigma} \,;\, \rho_r \rightrightarrows \mathrm{e}_0$ *and* $|e| = |\mathrm{e}|$, *then* $|e\,\overline{\pi}| = |\mathrm{e}_0|$.

Proof. By straightforward induction on the elaboration judgments. □

B PROOFS ABOUT OUR SURFACE LANGUAGE, $\mathbb{X}$

THEOREM B.1 (SOUNDNESS).

(1) *If* $\Gamma \vdash^{\forall} e \Leftarrow \sigma \rightrightarrows \mathrm{e}$*, then* $\mathrm{G} \vdash \mathrm{e} : \mathrm{s}$*, where* $\Gamma \rightrightarrows \mathrm{G}$ *and* $\sigma \rightrightarrows \mathrm{s}$.
(2) *If* $\Gamma \vdash e \Rightarrow \rho \rightrightarrows \mathrm{e}$*, then* $\mathrm{G} \vdash \mathrm{e} : \mathrm{r}$*, where* $\Gamma \rightrightarrows \mathrm{G}$ *and* $\rho \rightrightarrows \mathrm{r}$.
(3) *If* $\Gamma \vdash e \Leftarrow \rho \rightrightarrows \mathrm{e}$*, then* $\mathrm{G} \vdash \mathrm{e} : \mathrm{r}$*, where* $\Gamma \rightrightarrows \mathrm{G}$ *and* $\rho \rightrightarrows \mathrm{r}$.
(4) *If* $\Gamma \vdash_h h \Rightarrow \sigma \rightrightarrows \mathrm{h}$*, then* $\mathrm{G} \vdash \mathrm{h} : \mathrm{s}$*, where* $\Gamma \rightrightarrows \mathrm{G}$ *and* $\sigma \rightrightarrows \mathrm{s}$.
(5) *If* $\Gamma \vdash^{\mathrm{inst}} h : \sigma \rightrightarrows \mathrm{h} \,;\, \overline{\pi} \rightsquigarrow \overline{\sigma} \,;\, \rho_r \rightrightarrows \mathrm{e}_r$ *and* $\mathrm{G} \vdash \mathrm{h} : \mathrm{s}$*, then* $\mathrm{G} \vdash \mathrm{e}_r : \mathrm{r}_r$ *where* $\Gamma \rightrightarrows \mathrm{G}$*,* $\sigma \rightrightarrows \mathrm{s}$ *and* $\rho_r \rightrightarrows \mathrm{r}_r$.

PROOF. By (mutual) structural induction on the typing rule. The full set of rules can be found in Annex A.

Rule ELAB-GEN From the premise: $\Gamma, \overline{a} \vdash e \Leftarrow \rho[\overline{\tau} \,/\, \overline{b}] \rightrightarrows \mathrm{e}$, where $\overline{\tau \rightrightarrows \mathrm{t}}$ and $\rho \rightrightarrows \mathrm{r}$. By induction hypothesis, $\mathrm{G}, \overline{a} \vdash \mathrm{e} : \mathrm{r}[\overline{\mathrm{t}} \,/\, \overline{b}]$. By successive applications of rule CE-PACK we get $\mathrm{G}, \overline{a} \vdash \mathbf{pack}\, \overline{\mathrm{t}}, \mathrm{e}\, \mathbf{as}\, \exists\, \overline{b}.\mathrm{r} : \exists\, \overline{b}.\mathrm{r}$. Then by successive applications of rule CE-TABS we get the result: $\mathrm{G} \vdash \Lambda \overline{a}.\mathbf{pack}\, \overline{\mathrm{t}}, \mathrm{e}\, \mathbf{as}\, \exists\, \overline{b}.\mathrm{r} : \forall\, \overline{a}.\exists\, \overline{b}.\mathrm{r}$.

Rule ELAB-APP Inference and synthesis are treated at the same time by mutual induction. By induction hypothesis, $\mathrm{G} \vdash \mathrm{h} : \mathrm{s}$ where $\sigma \rightrightarrows \mathrm{s}$. Then by induction hypothesis (case (5)), we obtain $\mathrm{G} \vdash \mathrm{e}_r : \mathrm{r}_r$.

Rule ELAB-IABS By induction hypothesis, $\mathrm{G}, x : \mathrm{t} \vdash \mathrm{e} : \mathrm{r}$. By applications of rule CE-PACK we obtain $\mathrm{G}, x : \mathrm{t} \vdash \mathbf{pack}\, \lfloor \mathrm{r} \rfloor_x, \mathrm{e}\, \mathbf{as}\, \exists\, \overline{a}.\mathrm{r}' : \exists\, \overline{a}.\mathrm{r}'$ where $\mathrm{r}' = \mathrm{r}[\overline{a} \,/\, \lfloor \mathrm{r} \rfloor_x]$. We conclude by applying rule CE-ABS where the premise $x \notin fv(\exists\, \overline{a}.\mathrm{r}')$ is verified by construction of r' and definition of $\lfloor \mathrm{r} \rfloor_x$.

Rule ELAB-CABS By induction hypothesis and rule CE-APP.

Rule ELAB-LETCORE Inference and synthesis are treated at the same time. By induction hypothesis and rule rule CE-LET.

Rule ELAB-VAR Since $x{:}\sigma \in \Gamma$, we have $x : \mathrm{s} \in \mathrm{G}$ and we conclude by rule CE-VAR.

Rule ELAB-ANN By induction hypothesis.

Rule ELAB-INFER By induction hypothesis.

We see the instantiation judgment for elaboration as a bottom-up computation initialized, in rule ELAB-APP, by a head h such that $\mathrm{G} \vdash \mathrm{h} : \mathrm{s}$. Hence we just prove that going "up" in the derivation tree maintains the invariant that the first core expression e is well-typed (i.e. that $\Gamma \vdash^{\mathrm{inst}} e : \sigma \rightrightarrows \mathrm{e} \,;\, \overline{\pi} \rightsquigarrow \overline{\sigma} \,;\, \rho_r \rightrightarrows \mathrm{e}_r$ implies $\mathrm{G} \vdash \mathrm{e} : \mathrm{s}$ where $\sigma \rightrightarrows \mathrm{s}$).

Rule ELAB-ITYARG Assuming that $\mathrm{G} \vdash \mathrm{e} : \forall\, a.\mathrm{s}$, by rule CE-TAPP: $\mathrm{G} \vdash \mathrm{e}\, \mathrm{s}' : \mathrm{s}[\mathrm{s}' \,/\, a]$.

Rule ELAB-IARG Assuming that $\mathrm{G} \vdash \mathrm{e} : \mathrm{s}_1 \rightarrow \mathrm{s}_2$ and $\Gamma \vdash^{\forall} e' \Leftarrow \sigma_1 \rightrightarrows \mathrm{e}'$. By induction hypothesis, $\mathrm{G} \vdash \mathrm{e}' : \mathrm{s}_1$ where $\sigma_1 \rightrightarrows \mathrm{s}_1$. By rule CE-APP we obtain $\mathrm{G} \vdash \mathrm{e}\, \mathrm{e}' : \mathrm{s}_2$.

Rule ELAB-IALL Assuming that $\mathrm{G} \vdash \mathrm{e} : \forall\, a.\mathrm{s}$. By rule CE-TAPP, we obtain $\mathrm{G} \vdash \mathrm{e}\, \mathrm{t} : \mathrm{s}[\mathrm{t} \,/\, a]$.

Rule ELAB-IEXISTCORE Assuming that $\mathrm{G} \vdash \mathrm{e} : \exists\, a.\mathrm{t}$ where $\epsilon \rightrightarrows \mathrm{t}$. By rule CE-OPEN: $\mathrm{G} \vdash \mathbf{open}\, \mathrm{e} : \mathrm{t}[\lfloor \mathrm{e} \rfloor \,/\, a]$.

Finally, at the top of the derivation tree, rule ELAB-IRESULT ensures that this invariant translates to the result of the computation, that is, to the second core expression e_r and the result type ρ_r such that $\mathrm{G} \vdash \mathrm{e}_r : \mathrm{r}_r$ with $\rho_r \rightrightarrows \mathrm{r}_r$. □

THEOREM B.2 (CONSERVATIVE EXTENSION OF CLÉMENT ET AL. [1986]). *If e has no type arguments or type annotations, and* Γ, e, τ, σ *contain no existentials, then:*

(1) $(\Gamma \vdash_{HM} e : \tau)$ *implies* $(\Gamma \vdash e \Rightarrow \tau)$
(2) $(\Gamma \vdash_{HM} e : \sigma)$ *implies* $\left(\Gamma \vdash^{\forall} e \Leftarrow \sigma\right)$

where $\vdash_{HM}$ denotes typing in the Hindley-Milner type system, as described by Clément et al. [1986, Figure 3].

Proof. Proceed by induction on the length of the derivation for $\Gamma \vdash_{HM} e : \tau$ and case analysis on e.

$e = x$: The rule used is C_Var. From its premise we get $x{:}\forall\,\overline{a}.\tau' \in \Gamma$, with $\tau = \tau'[\overline{\tau}\,/\,\overline{a}]$. In our type system, we can type $\Gamma \vdash_h x \Rightarrow \forall\,\overline{a}.\tau$ with H-Var. Then the instantiation judgment gives us $\Gamma \vdash^{\mathsf{inst}} x : \forall\,\overline{a}.\tau'\ ;\ []\ \rightsquigarrow\ []\ ;\ \tau$ as the IAll rule will be used to instantiate $\forall\,\overline{a}.\tau$ with $\overline{\tau}$. Finally we apply App to obtain $\Gamma \vdash x \Rightarrow \tau$.

$e = \lambda x.e'$: Since there are no existentials in $\tau = \tau_1 \to \tau_2$, hence in τ_2, the iAbs rule is the same as the usual C_Abs rule, therefore we conclude by induction.

let $x = e_1$ in e_2: Without existentials, the Let rule is the same as applying the C_Gen and C_Let rules at the same time.

$e = h\,e_1 \ldots e_n$: The type of h is $\tau_1 \to\ \ldots\ \to \tau_n \to \tau$. By applying the induction hypothesis on the successive premises obtained by inversing the C_App rules used to type e, we get $\Gamma \vdash e_i \Rightarrow \tau_i$ for all i, hence by Theorem 7.6: $\Gamma \vdash e_i \Leftarrow \tau_i$. The instantiation judgment, given as input $h : \tau_1 \to\ \ldots\ \to \tau_n \to \tau$ and the list of arguments $e_1 \ldots e_n$, outputs the list of types $\tau_1 \ldots \tau_n$ and the return type τ. Hence we can apply App.

□

Theorem B.3 (Synthesis implies checking). *If $\Gamma \vdash e \Rightarrow \rho$ then $\Gamma \vdash e \Leftarrow \rho$.*

Proof. Proceed by induction on the typing judgment $\Gamma \vdash e \Rightarrow \rho$.

Rule iAbs: By inversion and applying the induction hypothesis, we get $\Gamma, x{:}\tau \vdash e \Leftarrow \rho$. Hence by rule Gen, $\Gamma, x{:}\tau \vdash^{\forall} e \Leftarrow \exists\,\overline{a}.\rho'$ and we conclude by rule cAbs.

Rule Let and rule App: Same rules for synthesis and checking.

□

Theorem B.4 (Order of Quantification does not matter). *Let ρ' (resp. σ') be two types that differ from ρ (resp. σ) only by the ordering of quantified type variables in their (eventual) existential types. Then:*

(1) $(\Gamma \vdash e \Rightarrow \rho)$ *if and only if* $(\Gamma \vdash e \Rightarrow \rho')$
(2) $(\Gamma \vdash^{\forall} e \Leftarrow \sigma)$ *if and only if* $(\Gamma \vdash^{\forall} e \Leftarrow \sigma')$

Proof. In inference mode, the only rule that packs existentials is rule iAbs. This rule packs all the possible type variables at the same time, hence we see that their ordering does not matter. It is trivial therefore to choose one ordering or the other, to go from type ρ to type ρ'.

In checking mode, rule Gen also does several packs at once, whose ordering does not matter. □

Lemma B.5. *If $\overline{a} \notin dom(\Gamma)$*

(1) *If $\Gamma \vdash^{\forall} e \Leftarrow \sigma$ then $\overline{a} \notin fv(e)$.*
(2) *If $\Gamma \vdash e \Rightarrow \rho$ then $\overline{a} \notin fv(e)$.*
(3) *If $\Gamma \vdash_h h \Rightarrow \sigma$ then $\overline{a} \notin fv(h)$.*

Proof. By structural induction on the derivation.

Rule Gen: By inversion, $\Gamma, \overline{a}' \vdash e \Leftarrow \rho[\overline{\tau}\,/\,\overline{b}]$. By α-equivalence, it is permissible to choose the $\overline{a}'$ fresh, such that $\overline{a}$ and $\overline{a}'$ do not intersect. Hence, we have $\overline{a} \notin dom(\Gamma, \overline{a}')$ and by induction hypothesis $\overline{a} \notin fv(e)$.

Rule App**:** By induction hypothesis, we have $\overline{a} \notin fv(h)$ as well as $\overline{a} \notin fv(e_i)$ for all i. Since $\Gamma \vdash^{\mathsf{inst}} h : \sigma ; \overline{\pi} \rightsquigarrow \overline{\sigma} ; \rho_r$, we also know thanks to the scoping rule of rule ITyArg that for every $\sigma' \in \overline{\pi}$, $fv(\sigma') \subseteq dom(\Gamma)$. So since $\overline{a} \notin dom(\Gamma)$ we conclude that $\overline{a} \notin fv(h\,\overline{\pi})$.

Rule iAbs**:** Since $fv(\tau) \subseteq dom(\Gamma)$, we have $\overline{a} \notin dom(\Gamma, x{:}\tau)$ and by induction hypothesis $\overline{a} \notin fv(e)$, which concludes.

Rule cAbs**:** Since $fv(\sigma_1) \subseteq dom(\Gamma)$, we conclude by induction hypothesis.

Rule LetCore By induction hypothesis $\overline{a} \notin fv(e_1)$. Consider $\overline{a}' = fv(\rho_1) \backslash dom(\Gamma)$ and $\Gamma, x{:}\forall\,\overline{a}'.\rho_1 \vdash e_2 \Leftrightarrow \rho_2$. By definition of the $\overline{a}'$, $\overline{a} \notin dom(\Gamma, x{:}\forall\,\overline{a}'.\rho_1)$ so by induction hypothesis $\overline{a} \notin fv(e_2)$ which concludes.

Rule H-Var**:** There are no type variables in x.

Rule H-Ann**:** The scoping condition $fv(\sigma) \subseteq dom(\Gamma)$ with the induction hypothesis ensures the result.

Rule H-Infer**:** By induction hypothesis.

□

Lemma B.6. *Assuming* $\overline{a} \notin dom(\Gamma)$ *and* $fv(\overline{\tau}) \subseteq dom(\Gamma)$.

(1) *If* $\Gamma \vdash^{\forall} e \Leftarrow \sigma \rightrightarrows \mathrm{e}$, *then* $\Gamma \vdash^{\forall} e \Leftarrow \sigma[\overline{\tau} \,/\, \overline{a}] \rightrightarrows \mathrm{e}[\overline{\mathrm{t}} \,/\, \overline{a}]$, *where* $\overline{\tau} \rightrightarrows \overline{\mathrm{t}}$.
(2) *If* $\Gamma \vdash_h h \Rightarrow \sigma \rightrightarrows \mathrm{h}$, *then* $\Gamma \vdash_h h \Rightarrow \sigma[\overline{\tau} \,/\, \overline{a}] \rightrightarrows \mathrm{h}[\overline{\mathrm{t}} \,/\, \overline{a}]$, *where* $\overline{\tau} \rightrightarrows \overline{\mathrm{t}}$.
(3) *If* $\Gamma \vdash e \Leftarrow \rho \rightrightarrows \mathrm{e}$, *then* $\Gamma \vdash e \Leftarrow \rho[\overline{\tau} \,/\, \overline{a}] \rightrightarrows \mathrm{e}[\overline{\mathrm{t}} \,/\, \overline{a}]$, *where* $\overline{\tau} \rightrightarrows \overline{\mathrm{t}}$.
(4) *If* $\Gamma \vdash_h h \Rightarrow \sigma[\overline{\tau} \,/\, \overline{a}] \rightrightarrows \mathrm{h}[\overline{\mathrm{t}} \,/\, \overline{a}]$ *where* $\overline{\tau} \rightrightarrows \overline{\mathrm{t}}$ *and* $\Gamma \vdash^{\mathsf{inst}} h : \sigma \rightrightarrows \mathrm{h} ; \overline{\pi} \rightsquigarrow \overline{\sigma} ; \rho_r \rightrightarrows \mathrm{e}_r$, *then* $\Gamma \vdash^{\mathsf{inst}} h : \sigma[\overline{\tau} \,/\, \overline{a}] \rightrightarrows \mathrm{e}[\overline{\mathrm{t}} \,/\, \overline{a}] ; \overline{\pi} \rightsquigarrow \overline{\sigma}[\overline{\tau} \,/\, \overline{a}] ; \rho_r[\overline{\tau} \,/\, \overline{a}] \rightrightarrows \mathrm{e}_r[\overline{\mathrm{t}} \,/\, \overline{a}]$ *where* $\overline{\tau} \rightrightarrows \overline{\mathrm{t}}$.

Proof. By structural induction on elaboration derivations.

Rule Elab-Gen**:** Since $\overline{a} \notin dom(\Gamma, \overline{a}')$, by induction hypothesis $\Gamma, \overline{a}' \vdash e \Leftarrow \rho[\overline{\tau}' \,/\, \overline{b}] \rightrightarrows \mathrm{e}[\overline{\mathrm{t}} \,/\, \overline{a}]$ where $\overline{\tau} \rightrightarrows \overline{\mathrm{t}}$. By rule Elab-Gen $\Gamma \vdash^{\forall} e \Leftarrow \forall\,\overline{a}'.\exists\,\overline{b}.\rho[\overline{\tau} \,/\, \overline{a}] \rightrightarrows \Lambda\overline{a}.\mathbf{pack}\,\overline{\mathrm{t}}', \mathrm{e}[\overline{\mathrm{t}} \,/\, \overline{a}]\ \mathbf{as}\ \exists\,\overline{b}.\mathrm{r}[\overline{\mathrm{t}} \,/\, \overline{a}]$ where $\overline{\tau}' \rightrightarrows \overline{\mathrm{t}}'$. Since $fv(\overline{\tau}') \subseteq dom(\Gamma, \overline{a}')$ and $\overline{a} \notin dom(\Gamma)$, $\Lambda\overline{a}.\mathbf{pack}\,\overline{\mathrm{t}}', \mathrm{e}[\overline{\mathrm{t}} \,/\, \overline{a}]\ \mathbf{as}\ \exists\,\overline{b}.\mathrm{r}[\overline{\mathrm{t}} \,/\, \overline{a}] = (\Lambda\overline{a}.\mathbf{pack}\,\overline{\mathrm{t}}', \mathrm{e}\ \mathbf{as}\ \exists\,\overline{b}.\mathrm{r})[\overline{\mathrm{t}} \,/\, \overline{a}]$ which concludes.

Rule Elab-App**:** By induction hypothesis and case (4) of the Lemma.

Rule Elab-iAbs**:** By induction hypothesis $\Gamma, x{:}\tau \vdash e \Rightarrow \rho[\overline{\tau} \,/\, \overline{a}] \rightrightarrows \mathrm{e}[\overline{\mathrm{t}} \,/\, \overline{a}]$. We find that, since $fv(\overline{\tau}) \subseteq dom(\Gamma)$, $\rho[\overline{\tau} \,/\, \overline{a}][\overline{a}' \,/\, \lfloor \rho[\overline{\tau} \,/\, \overline{a}] \rfloor_x] = \rho[\overline{a}' \,/\, \lfloor \rho \rfloor_x][\overline{\tau} \,/\, \overline{a}]$. So by rule Elab-iAbs, we obtain $\Gamma \vdash \lambda x.e \Rightarrow \tau \to \exists\,\overline{a}'.\rho'[\overline{\tau} \,/\, \overline{a}] \rightrightarrows \lambda x{:}\mathrm{t}.\mathbf{pack}\,\lfloor \mathrm{r} \rfloor_x, \mathrm{e}[\overline{\mathrm{t}} \,/\, \overline{a}]\ \mathbf{as}\ \exists\,\overline{a}'.\mathrm{r}'[\overline{\mathrm{t}} \,/\, \overline{a}]$ which concludes since $\lambda x{:}\mathrm{t}.\mathbf{pack}\,\lfloor \mathrm{r} \rfloor_x, \mathrm{e}[\overline{\mathrm{t}} \,/\, \overline{a}]\ \mathbf{as}\ \exists\,\overline{a}'.\mathrm{r}'[\overline{\mathrm{t}} \,/\, \overline{a}] = (\lambda x{:}\mathrm{t}.\mathbf{pack}\,\lfloor \mathrm{r} \rfloor_x, \mathrm{e}\ \mathbf{as}\ \exists\,\overline{a}'.\mathrm{r}')[\overline{\mathrm{t}} \,/\, \overline{a}]$.

Rule Elab-cAbs**:** By induction hypothesis. We also use $fv(\sigma_1) \subseteq dom(\Gamma)$ to prove $\lambda x{:}\mathrm{s}_1.\mathrm{e}[\overline{\mathrm{t}} \,/\, \overline{a}] = (\lambda x{:}\mathrm{s}_1.\mathrm{e})[\overline{\mathrm{t}} \,/\, \overline{a}]$.

Rule Elab-LetCore**:** After remarking that by construction of $\overline{a}' = fv(\rho_1) \backslash dom(\Gamma)$, $\forall\,\overline{a}'.\rho_1 = (\forall\,\overline{a}'.\rho_1)[\overline{\tau} \,/\, \overline{a}]$, we conclude by induction hypothesis.

Rule Elab-Var**:** Since $\overline{a} \notin dom(\Gamma)$, this means the $\overline{a}$ do not appear in σ hence $\sigma[\overline{\tau} \,/\, \overline{a}] = \sigma$ and we are done.

Rule Elab-Ann**:** By induction hypothesis, and using the fact that $fv(\overline{\tau}) \subseteq dom(\Gamma)$.

Rule Elab-Infer**:** By induction hypothesis.

To prove case (4) of the Lemma, we go through the derivation tree for $\Gamma \vdash^{\mathsf{inst}} h : \sigma \rightrightarrows \mathrm{h} ; \overline{\pi} \rightsquigarrow \overline{\sigma} ; \rho \rightrightarrows \mathrm{e}_r$ and transform it by applying the substitution $[\overline{\tau} \,/\, \overline{a}]$ at every intermediary step. We show that it is does not change the result, since this substitution does not affect the application of the rules.

Rule Elab-ITyArg**:** Since $fv(\sigma') \subseteq dom(\Gamma)$ and $\overline{a} \notin dom(\Gamma)$, we conclude by noting that $\sigma[\overline{\tau} \,/\, \overline{a}][\sigma' \,/\, a] = \sigma[\sigma' \,/\, a][\overline{\tau} \,/\, \overline{a}]$.

Rule Elab-IArg**:** By case (1) of the Lemma, from $\Gamma \vdash^{\forall} e' \Leftarrow \sigma_1 \rightrightarrows \mathrm{e}'$ we obtain $\Gamma \vdash^{\forall} e' \Leftarrow \sigma_1[\overline{\tau} \,/\, \overline{a}] \rightrightarrows \mathrm{e}'[\overline{\mathrm{t}} \,/\, \overline{a}]$. Hence we correctly have $\Gamma \vdash^{\mathsf{inst}} e\, e' : \sigma_2[\overline{\tau} \,/\, \overline{a}] \rightrightarrows (\mathrm{e}\,\mathrm{e}')[\overline{\mathrm{t}} \,/\, \overline{a}]\ ;\ \overline{\pi} \rightsquigarrow \overline{\sigma}[\overline{\tau} \,/\, \overline{a}]\ ;\ \rho_r[\overline{\tau} \,/\, \overline{a}] \rightrightarrows \mathrm{e}_r[\overline{\mathrm{t}} \,/\, \overline{a}]$.
Rule Elab-IAll**:** We just notice that, since $fv(\tau) \subseteq dom(\Gamma)$, $\sigma[\overline{\tau} \,/\, \overline{a}][\tau \,/\, a] = \sigma[\tau \,/\, a][\overline{\tau} \,/\, \overline{a}]$.
Rule Elab-IExistCore**:** The rule applies with $\Gamma \vdash^{\mathsf{inst}} e : \epsilon[\lfloor \mathrm{e}[\overline{\mathrm{t}} \,/\, \overline{a}] \rfloor \,/\, a] \rightrightarrows \mathbf{open}\, \mathrm{e}[\overline{\mathrm{t}} \,/\, \overline{a}]\ ;\ \overline{\pi} \rightsquigarrow \overline{\sigma}[\overline{\tau} \,/\, \overline{a}]\ ;\ \rho_r[\overline{\tau} \,/\, \overline{a}] \rightrightarrows \mathrm{e}_r[\overline{\mathrm{t}} \,/\, \overline{a}]$. We conclude by noting that $\epsilon[\lfloor \mathrm{e}[\overline{\mathrm{t}} \,/\, \overline{a}] \rfloor \,/\, a] = \epsilon[\lfloor \mathrm{e} \rfloor \,/\, a][\overline{\tau} \,/\, \overline{a}]$ and $\mathbf{open}\, \mathrm{e}[\overline{\mathrm{t}} \,/\, \overline{a}] = (\mathbf{open}\, \mathrm{e})[\overline{\mathrm{t}} \,/\, \overline{a}]$.
Rule Elab-IResult**:** $\Gamma \vdash^{\mathsf{inst}} e : \rho_r[\overline{\tau} \,/\, \overline{a}] \rightrightarrows \mathrm{e}_r[\overline{\mathrm{t}} \,/\, \overline{a}]\ ;\ [] \rightsquigarrow []\ ;\ \rho_r[\overline{\tau} \,/\, \overline{a}] \rightrightarrows \mathrm{e}_r[\overline{\mathrm{t}} \,/\, \overline{a}]$ is true.

□

Lemma B.7 (Free variable substitution). *Given $\overline{a} \notin dom(\Gamma)$:*

(1) *If $\Gamma \vdash^{\forall} e \Leftarrow \sigma$, then $\Gamma \vdash^{\forall} e \Leftarrow \sigma[\overline{\tau} \,/\, \overline{a}]$.*
(2) *If $\Gamma \vdash_h h \Rightarrow \sigma$, then $\Gamma \vdash_h h \Rightarrow \sigma[\overline{\tau} \,/\, \overline{a}]$.*
(3) *If $\Gamma \vdash e \Rightarrow \rho$, then $\Gamma \vdash e \Rightarrow \rho[\overline{\tau} \,/\, \overline{a}]$.*
(4) *If $\Gamma \vdash_h h \Rightarrow \sigma$ and $\Gamma \vdash^{\mathsf{inst}} h : \sigma\ ;\ \overline{\pi} \rightsquigarrow \overline{\sigma}\ ;\ \rho_r$, then $\Gamma \vdash^{\mathsf{inst}} h : \sigma[\overline{\tau} \,/\, \overline{a}]\ ;\ \overline{\pi} \rightsquigarrow \overline{\sigma}[\overline{\tau} \,/\, \overline{a}]\ ;\ \rho_r[\overline{\tau} \,/\, \overline{a}]$.*

Proof. By corollary of Lemma B.6 □

Lemma B.8 (Substitution). *Suppose $\Gamma_1 \vdash e_1 \Rightarrow \rho_1 \rightrightarrows \mathrm{e}_1$ and take $\overline{a} = fv(\rho_1) \backslash fv(\Gamma_1)$.*

(1) *If $\Gamma_1, x{:}\forall \overline{a}.\rho_1, \Gamma_2 \vdash e_2 \Rightarrow \rho_2$, then $\Gamma_1, \Gamma_2[\Lambda\overline{a}.\mathrm{e}_1 \,/\, x] \vdash e_2[e_1 \,/\, x] \Rightarrow \rho_2[\Lambda\overline{a}.\mathrm{e}_1 \,/\, x]$.*
(2) *If $\Gamma_1, x{:}\forall \overline{a}.\rho_1, \Gamma_2 \vdash^{\forall} e_2 \Leftarrow \sigma$, then $\Gamma_1, \Gamma_2[\Lambda\overline{a}.\mathrm{e}_1 \,/\, x] \vdash^{\forall} e_2[e_1 \,/\, x] \Leftarrow \sigma[\Lambda\overline{a}.\mathrm{e}_1 \,/\, x]$*
(3) *If $\Gamma_1, x{:}\forall \overline{a}.\rho_1, \Gamma_2 \vdash^{\mathsf{inst}} h : \sigma\ ;\ \overline{\pi} \rightsquigarrow \overline{\sigma}\ ;\ \rho_r$, then $\Gamma_1, \Gamma_2[\Lambda\overline{a}.\mathrm{e}_1 \,/\, x] \vdash^{\mathsf{inst}} h[e_1 \,/\, x] : \sigma[\Lambda\overline{a}.\mathrm{e}_1 \,/\, x]\ ;\ \overline{\pi}[e_1 \,/\, x] \rightsquigarrow \overline{\sigma}[\Lambda\overline{a}.\mathrm{e}_1 \,/\, x]\ ;\ \rho_r[\Lambda\overline{a}.\mathrm{e}_1 \,/\, x]$*
(4) *If $\Gamma_1, x{:}\forall \overline{a}.\rho_1, \Gamma_2 \vdash_h h \Rightarrow \sigma$, then $\Gamma_1, \Gamma_2[\Lambda\overline{a}.\mathrm{e}_1 \,/\, x] \vdash_h h[e_1 \,/\, x] \Rightarrow \sigma[\Lambda\overline{a}.\mathrm{e}_1 \,/\, x]$*

Proof. (1,2,3,4) By induction on e_2.

- $e_2 = x$**:** Then $\Gamma_1, x{:}\forall \overline{a}.\rho_1, \Gamma_2 \vdash_h x \Rightarrow \rho_2$ implies $\rho_2 = \rho_1[\overline{\tau} \,/\, \overline{a}]$. This means that $\rho_2[\Lambda\overline{a}.\mathrm{e}_1 \,/\, x] = \rho_1[\overline{\tau} \,/\, \overline{a}][\Lambda\overline{a}.\mathrm{e}_1 \,/\, x]$. Since x does not appear in ρ_1 (it is not in Γ_1, which is used to type e_1 with ρ_1), we have in fact $\rho_2[\Lambda\overline{a}.\mathrm{e}_1 \,/\, x] = \rho_1[\overline{\tau}[\Lambda\overline{a}.\mathrm{e}_1 \,/\, x] \,/\, \overline{a}]$. Thus, since $\Gamma_1 \vdash e_1 \Leftarrow \rho_1$ and $\overline{a} \notin dom(\Gamma_1)$, by Lemma B.7 we obtain $\Gamma_1 \vdash e_1 \Leftarrow \rho_2[\Lambda\overline{a}.\mathrm{e}_1 \,/\, x]$, and then we conclude by weakening.
- $e_2 = e :: \sigma$**:** By inversion on rules App and rule H-Ann, we get $\Gamma_1, x{:}\forall \overline{a}.\rho_1 \vdash^{\forall} e \Leftarrow \sigma$. By induction hypothesis, $\Gamma_1 \vdash^{\forall} e[e_1 \,/\, x] \Leftarrow \sigma[\Lambda\overline{a}.\mathrm{e}_1 \,/\, x]$. Then, since projections do not appear in type arguments, $\sigma[\Lambda\overline{a}.\mathrm{e}_1 \,/\, x] = \sigma$ and $\Gamma_1 \vdash_h e[e_1 \,/\, x] :: \sigma \Rightarrow \sigma$, and we conclude by applying rule App.
- $e_2 = \lambda y.e$**:** By inversion on rule iAbs and induction hypothesis, $\Gamma_1, y{:}\tau[\Lambda\overline{a}.\mathrm{e}_1 \,/\, x] \vdash e[e_1 \,/\, x] \Rightarrow \rho[\Lambda\overline{a}.\mathrm{e}_1 \,/\, x]$. Hence $\Gamma_1 \vdash \lambda y.e[e_1 \,/\, x] \Rightarrow (\tau \to \exists \overline{b}.\rho')[\Lambda\overline{a}.\mathrm{e}_1 \,/\, x]$.
- $e_2 = \mathbf{let}\ y = e_3\ \mathbf{in}\ e_4$ By the induction hypothesis.
- $e_2 = h\,\overline{\pi}$ **with non-empty** $\overline{\pi}$**:** By the induction hypothesis.

□

Theorem B.9 (Let-inlining). *If x is free in e_2 then:*

(1) $(\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \Rightarrow \rho)$ *implies* $(\Gamma \vdash e_2[e_1 \,/\, x] \Rightarrow \rho)$
(2) $(\Gamma \vdash^{\forall} \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \Leftarrow \sigma)$ *implies* $(\Gamma \vdash^{\forall} e_2[e_1 \,/\, x] \Leftarrow \sigma)$

Proof. *(1)* By inversion on the LetCore rule, we have

$$\begin{cases} \Gamma \vdash e_1 \Rightarrow \rho_1 \rightrightarrows \mathsf{e}_1 \\ \Gamma, x{:}\forall\,\overline{a}.\rho_1 \vdash e_2 \Rightarrow \rho' \\ \overline{a} \;=\; fv(\rho_1) \backslash dom(\Gamma) \\ \rho = \rho'[\Lambda\overline{a}.\mathsf{e}_1 \,/\, x] \end{cases}$$

By Lemma B.8 we obtain $\Gamma \vdash e_2[e_1 \,/\, x] \Rightarrow \rho'[\Lambda\overline{a}.\mathsf{e}_1 \,/\, x]$.

(2) Let $\sigma = \forall\,\overline{a}.\exists\,\overline{b}.\rho$. By inversion on rule Gen, we have $\Gamma, \overline{a} \vdash \mathbf{let}\, x = e_1 \,\mathbf{in}\, e_2 \Leftarrow \rho[\overline{\tau} \,/\, \overline{b}]$. By inversion on rule LetCore, we obtain:

$$\begin{cases} \Gamma \vdash e_1 \Rightarrow \rho_1 \rightrightarrows \mathsf{e}_1 \\ \Gamma, x{:}\forall\,\overline{a}.\rho_1 \vdash e_2 \Leftarrow \rho' \\ \overline{a} \;=\; fv(\rho_1) \backslash dom(\Gamma) \\ \rho = \rho'[\Lambda\overline{a}.\mathsf{e}_1 \,/\, x] \end{cases}$$

By Lemma B.8, we obtain $\Gamma \vdash e_2[e_1 \,/\, x] \Leftarrow \rho'[\Lambda\overline{a}.\mathsf{e}_1 \,/\, x]$ i.e. $\Gamma \vdash e_2[e_1 \,/\, x] \Leftarrow \rho$. We conclude by rule Gen. □

C DETAILS AND PROOFS ABOUT THE CORE LANGUAGE, $\mathbb{FX}$

C.1 Typing rules

$\boxed{\mathsf{G} \vdash \mathsf{e} : \mathsf{t}}$ *(Core expression typing)*

$$\textsc{CE-Var}\quad \frac{\vdash \mathsf{G}\,\mathbf{ok} \qquad x : \mathsf{t} \in \mathsf{G}}{\mathsf{G} \vdash x : \mathsf{t}} \qquad \textsc{CE-Int}\quad \frac{\vdash \mathsf{G}\,\mathbf{ok}}{\mathsf{G} \vdash n : \mathsf{Int}} \qquad \textsc{CE-Abs}\quad \frac{\mathsf{G}, x : \mathsf{t}_1 \vdash \mathsf{e} : \mathsf{t}_2 \qquad x \notin fv(\mathsf{t}_2)}{\mathsf{G} \vdash \lambda x{:}\mathsf{t}_1.\mathsf{e} : \mathsf{t}_1 \to \mathsf{t}_2} \qquad \textsc{CE-App}\quad \frac{\mathsf{G} \vdash \mathsf{e}_1 : \mathsf{t}_1 \to \mathsf{t}_2 \qquad \mathsf{G} \vdash \mathsf{e}_2 : \mathsf{t}_1}{\mathsf{G} \vdash \mathsf{e}_1\,\mathsf{e}_2 : \mathsf{t}_2}$$

$$\textsc{CE-TAbs}\quad \frac{\mathsf{G}, a \vdash \mathsf{e} : \mathsf{t}}{\mathsf{G} \vdash \Lambda a.\mathsf{e} : \forall\, a.\mathsf{t}} \qquad \textsc{CE-TApp}\quad \frac{\mathsf{G} \vdash \mathsf{e} : \forall\, a.\mathsf{t}_1 \qquad \mathsf{G} \vdash \mathsf{t}_2 : \mathbf{type}}{\mathsf{G} \vdash \mathsf{e}\,\mathsf{t}_2 : \mathsf{t}_1[\mathsf{t}_2 \,/\, a]} \qquad \textsc{CE-Pack}\quad \frac{\mathsf{G} \vdash \mathsf{t} : \mathbf{type} \qquad \mathsf{G} \vdash \exists\, a.\mathsf{t}_2 : \mathbf{type} \qquad \mathsf{G} \vdash \mathsf{e} : \mathsf{t}_2[\mathsf{t} \,/\, a]}{\mathsf{G} \vdash \mathbf{pack}\,\mathsf{t}, \mathsf{e}\,\mathbf{as}\,\exists\, a.\mathsf{t}_2 : \exists\, a.\mathsf{t}_2}$$

$$\textsc{CE-Open}\quad \frac{\mathsf{G} \vdash \mathsf{e} : \exists\, a.\mathsf{t}}{\mathsf{G} \vdash \mathbf{open}\,\mathsf{e} : \mathsf{t}[\lfloor \mathsf{e} \rfloor \,/\, a]} \qquad \textsc{CE-Let}\quad \frac{\mathsf{G} \vdash \mathsf{e}_1 : \mathsf{t}_1 \qquad \mathsf{G}, x : \mathsf{t}_1 \vdash \mathsf{e}_2 : \mathsf{t}_2}{\mathsf{G} \vdash \mathbf{let}\, x = \mathsf{e}_1 \,\mathbf{in}\, \mathsf{e}_2 : \mathsf{t}_2[\mathsf{e}_1 \,/\, x]} \qquad \textsc{CE-Cast}\quad \frac{\mathsf{G} \vdash \mathsf{e} : \mathsf{t}_1 \qquad \mathsf{G} \vdash \gamma : \mathsf{t}_1 \sim \mathsf{t}_2}{\mathsf{G} \vdash \mathsf{e} \rhd \gamma : \mathsf{t}_2}$$

$\boxed{\mathsf{G} \vdash \mathsf{t} : \mathbf{type}}$ *(Core type well-formedness)*

$$\textsc{CT-Var}\quad \frac{\vdash \mathsf{G}\,\mathbf{ok} \qquad a \in \mathsf{G}}{\mathsf{G} \vdash a : \mathbf{type}} \qquad \textsc{CT-Base}\quad \frac{\vdash \mathsf{G}\,\mathbf{ok} \qquad \overline{\mathsf{G} \vdash \mathsf{t}_i : \mathbf{type}}}{\mathsf{G} \vdash B\,\overline{\mathsf{t}} : \mathbf{type}} \qquad \textsc{CT-ForAll}\quad \frac{\mathsf{G}, a \vdash \mathsf{t} : \mathbf{type}}{\mathsf{G} \vdash \forall\, a.\mathsf{t} : \mathbf{type}} \qquad \textsc{CT-Exists}\quad \frac{\mathsf{G}, a \vdash \mathsf{t} : \mathbf{type}}{\mathsf{G} \vdash \exists\, a.\mathsf{t} : \mathbf{type}}$$

$$\textsc{CT-Proj}\quad \frac{\vdash \mathsf{G}\,\mathbf{ok} \qquad fv(\mathsf{e}) \subseteq dom(\mathsf{G})}{\mathsf{G} \vdash \lfloor \mathsf{e} \rfloor : \mathbf{type}}$$

$\boxed{G \vdash \gamma : t_1 \sim t_2}$ *(Core coercion typing)*

$$\textsc{CG-Refl}\quad \frac{G \vdash t : \textbf{type}}{G \vdash \langle t \rangle : t \sim t}$$

$$\textsc{CG-Sym}\quad \frac{G \vdash \gamma : t_1 \sim t_2}{G \vdash \textbf{sym}\,\gamma : t_2 \sim t_1}$$

$$\textsc{CG-Trans}\quad \frac{G \vdash \gamma_1 : t_1 \sim t_2 \qquad G \vdash \gamma_2 : t_2 \sim t_3}{G \vdash \gamma_1 \mathbin{;;} \gamma_2 : t_1 \sim t_3}$$

$$\textsc{CG-Base}\quad \frac{\vdash G\ \textbf{ok} \qquad \overline{G \vdash \gamma : t_1 \sim t_2}}{G \vdash B\overline{\gamma} : B\overline{t}_1 \sim B\overline{t}_2}$$

$$\textsc{CG-ForAll}\quad \frac{G, a \vdash \gamma : t_1 \sim t_2}{G \vdash \forall\, a.\gamma : (\forall\, a.t_1) \sim (\forall\, a.t_2)}$$

$$\textsc{CG-Exists}\quad \frac{G, a \vdash \gamma : t_1 \sim t_2}{G \vdash \exists\, a.\gamma : (\exists\, a.t_1) \sim (\exists\, a.t_2)}$$

$$\textsc{CG-Proj}\quad \frac{G \vdash \eta : e_1 \sim e_2}{G \vdash \lfloor \eta \rfloor : \lfloor e_1 \rfloor \sim \lfloor e_2 \rfloor}$$

$$\textsc{CG-ProjPack}\quad \frac{G \vdash \textbf{pack}\,t, e\,\textbf{as}\,t_2 : t_2}{G \vdash \textbf{projpack}\,t, e\,\textbf{as}\,t_2 : \lfloor \textbf{pack}\,t, e\,\textbf{as}\,t_2 \rfloor \sim t}$$

$$\textsc{CG-InstForAll}\quad \frac{G \vdash \gamma_1 : (\forall\, a.t_1) \sim (\forall\, a.t_2) \qquad G \vdash \gamma_2 : t_3 \sim t_4}{G \vdash \gamma_1 \mathbin{@} \gamma_2 : t_1[t_3 / a] \sim t_2[t_4 / a]}$$

$$\textsc{CG-InstExists}\quad \frac{G \vdash \gamma_1 : (\exists\, a.t_1) \sim (\exists\, a.t_2) \qquad G \vdash \gamma_2 : t_3 \sim t_4}{G \vdash \gamma_1 \mathbin{@} \gamma_2 : t_1[t_3 / a] \sim t_2[t_4 / a]}$$

$$\textsc{CG-Nth}\quad \frac{G \vdash \gamma : B\overline{t} \sim B\overline{t}'}{G \vdash \textbf{nth}_n\,\gamma : t_n \sim t'_n}$$

$\boxed{G \vdash \eta : e_1 \sim e_2}$ *(Core expression coercion typing)*

$$\textsc{CH-Coherence}\quad \frac{G \vdash e : t_1 \qquad G \vdash \gamma : t_1 \sim t_2}{G \vdash e \rhd \gamma : e \sim (e \rhd \gamma)}$$

$$\textsc{CH-Step}\quad \frac{G \vdash e : t \qquad G \vdash e' : t \qquad G \vdash e \longrightarrow e'}{G \vdash \textbf{step}\,e : e \sim e'}$$

$\boxed{\vdash G\ \textbf{ok}}$ *(Core context well-formedness)*

$$\textsc{C-Nil}\quad \frac{}{\vdash \emptyset\ \textbf{ok}}$$

$$\textsc{C-Type}\quad \frac{\vdash G\ \textbf{ok} \qquad a \notin \mathit{dom}(G)}{\vdash G, a\ \textbf{ok}}$$

$$\textsc{C-Term}\quad \frac{G \vdash t : \textbf{type} \qquad x \notin \mathit{dom}(G)}{\vdash G, x : t\ \textbf{ok}}$$

$\boxed{G \vdash e \longrightarrow e'}$ *(Core operational semantics)*

$$\textsc{CS-Beta}\quad \frac{}{G \vdash (\lambda x{:}t.e_1)\,e_2 \longrightarrow e_1[e_2 / x]}$$

$$\textsc{CS-AppCong}\quad \frac{G \vdash e_1 \longrightarrow e_1'}{G \vdash e_1\,e_2 \longrightarrow e_1'\,e_2}$$

$$\textsc{CS-AppPull}\quad \frac{v = \lambda x{:}t.e_0 \qquad \gamma_1 = \textbf{sym}\,(\textbf{nth}_0\,\gamma) \qquad \gamma_2 = \textbf{nth}_1\,\gamma}{G \vdash (v \rhd \gamma)\,e \longrightarrow (v\,(e \rhd \gamma_1)) \rhd \gamma_2}$$

$$\textsc{CS-TAbsCong}\quad \frac{G, a \vdash e \longrightarrow e'}{G \vdash \Lambda a.e \longrightarrow \Lambda a.e'}$$

$$\textsc{CS-TAbsPull}\quad \frac{}{G \vdash \Lambda a.(v \rhd \gamma) \longrightarrow (\Lambda a.v) \rhd \forall\, a.\gamma}$$

$$\textsc{CS-TBeta}\quad \frac{}{G \vdash (\Lambda a.v)\,t \longrightarrow v[t / a]}$$

$$\textsc{CS-TAppCong}\quad \frac{G \vdash e \longrightarrow e'}{G \vdash e\,t \longrightarrow e'\,t}$$

$$\textsc{CS-TAppPull}\quad \frac{G \vdash v : \forall\, a.t_0}{G \vdash (v \rhd \gamma)\,t \longrightarrow v\,t \rhd (\gamma \mathbin{@} \langle t \rangle)}$$

$$\textsc{CS-PackCong}\quad \frac{G \vdash e \longrightarrow e'}{G \vdash \textbf{pack}\,t, e\,\textbf{as}\,t_2 \longrightarrow \textbf{pack}\,t, e'\,\textbf{as}\,t_2}$$

$$\textsc{CS-OpenPack}\quad \frac{}{\mathrm{G} \vdash \mathbf{open}\,(\mathbf{pack}\,\mathrm{t}, \mathrm{v}\,\mathbf{as}\,\mathrm{t}_2) \longrightarrow \mathrm{v} \rhd \langle \mathrm{t}_2 \rangle\,@(\mathbf{sym}\,(\mathbf{projpack}\,\mathrm{t}, \mathrm{v}\,\mathbf{as}\,\mathrm{t}_2))}$$

$$\textsc{CS-OpenPackCasted}\quad \frac{}{\mathrm{G} \vdash \mathbf{open}\,(\mathbf{pack}\,\mathrm{t}, (\mathrm{v} \rhd \gamma)\,\mathbf{as}\,\mathrm{t}_2) \longrightarrow (\mathrm{v} \rhd \gamma) \rhd \langle \mathrm{t}_2 \rangle\,@(\mathbf{sym}\,(\mathbf{projpack}\,\mathrm{t}, (\mathrm{v} \rhd \gamma)\,\mathbf{as}\,\mathrm{t}_2))}$$

$$\textsc{CS-OpenCong}\quad \frac{\mathrm{G} \vdash \mathrm{e} : \mathrm{t} \qquad \mathrm{G} \vdash \mathrm{e} \longrightarrow \mathrm{e}'}{\mathrm{G} \vdash \mathbf{open}\,\mathrm{e} \longrightarrow \mathbf{open}\,\mathrm{e}' \rhd \langle \mathrm{t} \rangle\,@(\mathbf{sym}\,\lfloor \mathbf{step}\,\mathrm{e} \rfloor)}$$

$$\textsc{CS-OpenPull}\quad \frac{\mathrm{v} \;=\; \mathbf{pack}\,\mathrm{t}_1, \mathrm{v}_0\,\mathbf{as}\,\exists\, a.\mathrm{t}_0}{\mathrm{G} \vdash \mathbf{open}\,(\mathrm{v} \rhd \gamma) \longrightarrow (\mathbf{open}\,\mathrm{v}) \rhd \gamma\,@\lfloor \mathrm{v} \rhd \gamma \rfloor}$$

$$\textsc{CS-Let}\quad \frac{}{\mathrm{G} \vdash \mathbf{let}\,x = \mathrm{e}_1\,\mathbf{in}\,\mathrm{e}_2 \longrightarrow \mathrm{e}_2[\mathrm{e}_1 \,/\, x]}$$

$$\textsc{CS-CastCong}\quad \frac{\mathrm{G} \vdash \mathrm{e} \longrightarrow \mathrm{e}'}{\mathrm{G} \vdash \mathrm{e} \rhd \gamma \longrightarrow \mathrm{e}' \rhd \gamma}$$

$$\textsc{CS-CastTrans}\quad \frac{}{\mathrm{G} \vdash (\mathrm{v} \rhd \gamma_1) \rhd \gamma_2 \longrightarrow \mathrm{v} \rhd (\gamma_1 \,;;\, \gamma_2)}$$

C.2 Structural properties

Lemma C.1 (Context regularity).

(1) *If* $\mathrm{G} \vdash \mathrm{e} : \mathrm{t}$, *then* $\vdash \mathrm{G}\,\mathbf{ok}$.
(2) *If* $\mathrm{G} \vdash \mathrm{t} :\ \mathbf{type}$, *then* $\vdash \mathrm{G}\,\mathbf{ok}$.
(3) *If* $\mathrm{G} \vdash \gamma : \mathrm{t}_1 \sim \mathrm{t}_2$, *then* $\vdash \mathrm{G}\,\mathbf{ok}$.
(4) *If* $\mathrm{G} \vdash \eta : \mathrm{e}_1 \sim \mathrm{e}_2$, *then* $\vdash \mathrm{G}\,\mathbf{ok}$.

Proof. By straightforward structural induction on the typing rule, inverting a rule in the context judgment in the cases of context extension. □

Lemma C.2 (Context prefix). *If* $\vdash \mathrm{G}, \mathrm{G}'\,\mathbf{ok}$, *then* $\vdash \mathrm{G}\,\mathbf{ok}$.

Proof. Straightforward induction on the structure of G'. □

Lemma C.3 (Weakening in types). *If* $\mathrm{G} \vdash \mathrm{t} :\ \mathbf{type}$ *and* $\vdash \mathrm{G}, \mathrm{G}'\,\mathbf{ok}$, *then* $\mathrm{G}, \mathrm{G}' \vdash \mathrm{t} :\ \mathbf{type}$.

Proof. By straightforward induction on $\mathrm{G} \vdash \mathrm{t} :\ \mathbf{type}$. In the case for rule CT-Proj, we use the transitivity of $\subseteq$. □

Lemma C.4 (Permutation in types). *Suppose* G' *is a permutation of* G *and* $\vdash \mathrm{G}'\,\mathbf{ok}$. *If* $\mathrm{G} \vdash \mathrm{t} :\ \mathbf{type}$, *then* $\mathrm{G}' \vdash \mathrm{t} :\ \mathbf{type}$.

Proof. By straightforward induction on $\mathrm{G} \vdash \mathrm{t} :\ \mathbf{type}$. In the case for rule CT-Proj, we use the fact that $\subseteq$ ignores permutations. □

Lemma C.5 (Permutation in context prefixes). *Suppose* G' *is a permutation of* G. *If* $\vdash \mathrm{G}, \mathrm{G}''\,\mathbf{ok}$ *and* $\vdash \mathrm{G}'\,\mathbf{ok}$, *then* $\vdash \mathrm{G}', \mathrm{G}''\,\mathbf{ok}$.

Proof. By induction on the structure of G'', appealing to Lemma C.4. □

Lemma C.6 (Permutation in contexts (1)).

(1) *If* $\vdash \mathrm{G}, x : \mathrm{t}, a, \mathrm{G}'\,\mathbf{ok}$, *then* $\vdash \mathrm{G}, a, x : \mathrm{t}, \mathrm{G}'\,\mathbf{ok}$.
(2) *If* $\vdash \mathrm{G}, a', a, \mathrm{G}'\,\mathbf{ok}$, *then* $\vdash \mathrm{G}, a, a', \mathrm{G}'\,\mathbf{ok}$.

Proof.

(1) By Lemma C.2, we know $\vdash \mathsf{G}, x : \mathsf{t}, a\ \mathbf{ok}$. Inversion tells us that $\mathsf{G} \vdash \mathsf{t} : \mathbf{type}$. We then use rule C-Term to get $\vdash \mathsf{G}, a, x : \mathsf{t}\ \mathbf{ok}$. We are then done by Lemma C.5.

(2) By Lemma C.2, we know $\vdash \mathsf{G}, a', a\ \mathbf{ok}$. We are done by inversion, rule C-Type, and Lemma C.5

□

Lemma C.7 (Permutation in contexts). *If* $\vdash \mathsf{G}_1, \mathsf{G}_2, a, \mathsf{G}_3\ \mathbf{ok}$*, then* $\vdash \mathsf{G}_1, a, \mathsf{G}_2, \mathsf{G}_3\ \mathbf{ok}$.

Proof. By induction on the structure of G_2, appealing to Lemma C.6. □

Lemma C.8 (Strengthening in contexts). *If* $\vdash \mathsf{G}, x : \mathsf{t}, \mathsf{G}'\ \mathbf{ok}$ *and* G' *contains only type variable bindings. Then* $\vdash \mathsf{G}, \mathsf{G}'\ \mathbf{ok}$.

Proof. Straightforward induction on the structure of G'. □

Lemma C.9 (Strengthening in types). *Suppose* $\mathsf{G}, x : \mathsf{t}', \mathsf{G}' \vdash \mathsf{t} : \mathbf{type}$, $x \notin fv(\mathsf{t})$*, and* G' *contains only type variable bindings. Then* $\mathsf{G}, \mathsf{G}' \vdash \mathsf{t} : \mathbf{type}$.

Proof. By induction on the structure of $\mathsf{G}, x : \mathsf{t}', \mathsf{G}' \vdash \mathsf{t} : \mathbf{type}$.

Rule CT-Var: By appeal to Lemma C.8 and rule CT-Var.

Rule CT-Base: By the induction hypothesis and Lemma C.8.

Rule CT-ForAll: By the induction hypothesis.

Rule CT-Exists: By the induction hypothesis.

Rule CT-Proj: We use Lemma C.8 to show $\vdash \mathsf{G}, \mathsf{G}'\ \mathbf{ok}$ We know $\mathsf{t} = \lfloor \mathsf{e} \rfloor$, and that we further know that $fv(\mathsf{e}) \subseteq dom(\mathsf{G}, x : \mathsf{t}, \mathsf{G}')$. However, we also have assumed that $x \notin fv(\mathsf{e})$, and thus $fv(\mathsf{e}) \subseteq dom(\mathsf{G}, \mathsf{G}')$. We can finish with rule CT-Proj.

□

Lemma C.10 (Permutation in terms). *Suppose* G' *is a permutation of* G *and* $\vdash \mathsf{G}'\ \mathbf{ok}$.

(1) *If* $\mathsf{G} \vdash \mathsf{e} : \mathsf{t}$*, then* $\mathsf{G}' \vdash \mathsf{e} : \mathsf{t}$.

(2) *If* $\mathsf{G} \vdash \gamma : \mathsf{t}_1 \sim \mathsf{t}_2$*, then* $\mathsf{G}' \vdash \gamma : \mathsf{t}_1 \sim \mathsf{t}_2$.

(3) *If* $\mathsf{G} \vdash \eta : \mathsf{e}_1 \sim \mathsf{e}_2$*, then* $\mathsf{G}' \vdash \eta : \mathsf{e}_1 \sim \mathsf{e}_2$.

(4) *If* $\mathsf{G} \vdash \mathsf{e} \longrightarrow \mathsf{e}'$*, then* $\mathsf{G}' \vdash \mathsf{e} \longrightarrow \mathsf{e}'$.

Proof. Straightforward mutual induction on the structure of the assumed typing judgment, using Lemma C.4 in cases that refer to the well-formedness of types. □

Lemma C.11 (Weakening in terms). *Suppose* $\vdash \mathsf{G}, \mathsf{G}'\ \mathbf{ok}$.

(1) *If* $\mathsf{G} \vdash \mathsf{e} : \mathsf{t}$*, then* $\mathsf{G}, \mathsf{G}' \vdash \mathsf{e} : \mathsf{t}$.

(2) *If* $\mathsf{G} \vdash \gamma : \mathsf{t}_1 \sim \mathsf{t}_2$*, then* $\mathsf{G}, \mathsf{G}' \vdash \gamma : \mathsf{t}_1 \sim \mathsf{t}_2$.

(3) *If* $\mathsf{G} \vdash \eta : \mathsf{e}_1 \sim \mathsf{e}_2$*, then* $\mathsf{G}, \mathsf{G}' \vdash \eta : \mathsf{e}_1 \sim \mathsf{e}_2$.

(4) *If* $\mathsf{G} \vdash \mathsf{e} \longrightarrow \mathsf{e}'$*, then* $\mathsf{G}, \mathsf{G}' \vdash \mathsf{e} \longrightarrow \mathsf{e}'$.

Proof. Straightforward mutual induction on the structure of the assumed judgment, allowing variable renaming in rules CE-Abs, CE-TAbs, CE-Let, CG-ForAll, CG-Exists, and CS-TAbsCong and using Lemma C.10 in those cases. Cases using the type well-formedness judgment additionally need Lemma C.3. □

Lemma C.12 (Well-formed context types). *If* $\vdash \mathsf{G}\ \mathbf{ok}$ *and* $x : \mathsf{t} \in \mathsf{G}$ *then* $\mathsf{G} \vdash \mathsf{t} : \mathbf{type}$.

Proof. By structural induction on the structure of $\vdash \mathsf{G}\ \mathbf{ok}$.

Rule C-Nil: Not possible, by $x : \mathsf{t} \in \mathsf{G}$.

Rule C-Type: By the induction hypothesis and Lemma C.3.

Rule C-Term: If we have found the binding for x, the result comes straight from Lemma C.3. Otherwise, we use the induction hypothesis and Lemma C.3.

□

Lemma C.13 (Expression scoping).

(1) *If* $G \vdash e : t$*, then* $fv(e) \subseteq dom(G)$.
(2) *If* $G \vdash \gamma : t_1 \sim t_2$*, then* $fv(\gamma) \subseteq dom(G)$.
(3) *If* $G \vdash \eta : e_1 \sim e_2$*, then* $fv(\eta) \subseteq dom(G)$.

Proof. Straightforward mutual induction on $G \vdash e : t$, $G \vdash \gamma : t_1 \sim t_2$, and $G \vdash \eta : e_1 \sim e_2$. We must use Lemma C.12 in the case for rule CE-Abs. □

C.3 Preservation

Lemma C.14 (Type substitution in types).

(1) *If* $G_1, a, G_2 \vdash t_1 : \textbf{type}$ *and* $G_1 \vdash t_2 : \textbf{type}$*, then* $G_1, G_2[t_2 / a] \vdash t_1[t_2 / a] : \textbf{type}$.
(2) *If* $\vdash G_1, a, G_2\ \textbf{ok}$ *and* $G_1 \vdash t_2 : \textbf{type}$*, then* $\vdash G_1, G_2[t_2 / a]\ \textbf{ok}$.

Proof. By mutual induction on the structure of the typing judgments.

Rule CT-Var: Here, we know $t_1 = a'$, and inversion tells us $\vdash G_1, a, G_2\ \textbf{ok}$. The induction hypothesis tells us that $\vdash G_1, G_2[t_2 / a]\ \textbf{ok}$. We now have three cases:
- $a' \in G_1$**:** We must prove $G_1, G_2[t_2 / a] \vdash a' : \textbf{type}$. This comes straight from $\vdash G_1, G_2[t_2 / a]\ \textbf{ok}$ and $a' \in G_1$, by rule CT-Var.
- $a' = a$**:** We must prove $G_1, G_2[t_2 / a] \vdash t_2 : \textbf{type}$. We are done by Lemma C.3.
- $a' \in G_2$**:** We must prove $G_1, G_2[t_2 / a] \vdash a' : \textbf{type}$. This comes straight from $\vdash G_1, G_2[t_2 / a]\ \textbf{ok}$, and $a' \in G_2[t_2 / a]$, by rule CT-Var. (Note that substitutions do not affect type variable bindings.)

Rule CT-Base: By the induction hypothesis.
Rule CT-ForAll: By the induction hypothesis.
Rule CT-Exists: In this case, $t_1 = \exists a'.t_0$. Inversion tells us $G_1, a, G_2, a' \vdash t_0 : \textbf{type}$. We now use the induction hypothesis to get $G_1, G_2[t_2 / a], a' \vdash t_0[t_2 / a] : \textbf{type}$ and finish with rule CT-Exists to get $G_1, G_2[t_2 / a] \vdash \exists a'.t_0[t_2 / a] : \textbf{type}$ as desired.
Rule CT-Proj: We know $t_1 = \lfloor e \rfloor$, and inversion tells us that $\vdash G_1, a, G_2\ \textbf{ok}$ and $fv(e) \subseteq dom(G_1, a, G_2)$. We must prove $G_1, G_2[t_2 / a] \vdash \lfloor e[t_2 / a] \rfloor : \textbf{type}$. The induction hypothesis tells us that $\vdash G_1, G_2[t_2 / a]\ \textbf{ok}$, so (using rule CT-Proj) we must prove only that $fv(e[t_2 / a]) \subseteq dom(G_1, G_2[t_2 / a])$. This must be true, because a cannot be free in $e[t_2 / a]$ and $dom(G_2[t_2 / a]) = dom(G_2)$.
Rule C-Nil: Impossible.
Rule C-Type: We have two cases, depending on whether G_2 is empty. If G_2 is empty, our result is immediate. Otherwise, it comes from the induction hypothesis.
Rule C-Term: By the induction hypothesis.

□

Lemma C.15 (Type substitution).

(1) *If* $G_1, x : t_2, G_2 \vdash t_1 : \textbf{type}$ *and* $G_1 \vdash e_2 : t_2$*, then* $G_1, G_2[e_2 / x] \vdash t_1[e_2 / x] : \textbf{type}$.
(2) *If* $\vdash G_1, x : t_2, G_2\ \textbf{ok}$ *and* $G_1 \vdash e_2 : t_2$*, then* $\vdash G_1, G_2[e_2 / x]\ \textbf{ok}$.

Proof. By mutual induction on the typing judgments.

Rule CT-Var: We know that $t_1 = a$, and inversion of rule CT-Var gives us $\vdash G_1, x : t_2, G_2\ \textbf{ok}$ and $a \in G_1, x : t_2, G_2$. We must prove $G_1, G_2[e_2 / x] \vdash a : \textbf{type}$. The induction hypothesis gives us that $\vdash \mathrm{G}_1, \mathrm{G}_2[\mathrm{e}_2 / x]$ **ok**. And, noting that substitutions do not affect type variable bindings, we must have $a \in \mathrm{G}_1, \mathrm{G}_2[\mathrm{e}_2 / x]$. Thus we are done by rule CT-Var.

Rule CT-Base**:** By the induction hypothesis.

Rule CT-ForAll**:** By the induction hypothesis.

Rule CT-Exists**:** By the induction hypothesis.

Rule CT-Proj**:** We know that $\mathrm{t}_1 = \lfloor \mathrm{e} \rfloor$; we must prove $\mathrm{G}_1, \mathrm{G}_2[\mathrm{e}_2 / x] \vdash \lfloor \mathrm{e} \rfloor [\mathrm{e}_2 / x] : \textbf{type}$.

We know	How
$\vdash \mathrm{G}_1, x : \mathrm{t}_2, \mathrm{G}_2$ **ok**	inversion of rule CT-Proj
$fv(\mathrm{e}) \subseteq dom(\mathrm{G}_1, x : \mathrm{t}_2, \mathrm{G}_2)$	inversion of rule CT-Proj
$\vdash \mathrm{G}_1, \mathrm{G}_2[\mathrm{e}_2 / x]$ **ok**	induction hypothesis
$fv(\mathrm{e}[\mathrm{e}_2 / x]) \subseteq fv(\mathrm{e}) \cup fv(\mathrm{e}_2) \backslash \{x\}$	def'n of substitution
$fv(\mathrm{e}[\mathrm{e}_2 / x]) \subseteq dom(\mathrm{G}_1, \mathrm{G}_2[\mathrm{e}_2 / x])$	rules of $\subseteq$
$\mathrm{G}_1, \mathrm{G}_2[\mathrm{e}_2 / x] \vdash \lfloor \mathrm{e} \rfloor [\mathrm{e}_2 / x] : \textbf{type}$	rule CT-Proj

Rule C-Nil**:** Impossible, as the starting context is not empty (it has a binding for x).

Rule C-Type**:** By the induction hypothesis, noting that the substitution in contexts will not affect a type variable binding. (Type variables a and term variables x are distinct.)

Rule C-Term**:** We have two cases: either G_2 is empty or not. If it is empty, then we are done by Lemma C.1. If it is not empty, then we know that the substitution does not affect the name of the last variable in the context, and we are done by the (first) induction hypothesis.

□

Lemma C.16 (Substitution in values). *If* v *is a value, then* $\mathrm{v}[\mathrm{e} / x]$ *is also a value.*

Proof. Straightforward induction on the definition of values. □

Lemma C.17 (Substitution). *Suppose* $\mathrm{G}_1 \vdash \mathrm{e}_2 : \mathrm{t}_2$.

(1) *If* $\mathrm{G}_1, x : \mathrm{t}_2, \mathrm{G}_2 \vdash \mathrm{e}_1 : \mathrm{t}_1$*, then* $\mathrm{G}_1, \mathrm{G}_2[\mathrm{e}_2 / x] \vdash \mathrm{e}_1[\mathrm{e}_2 / x] : \mathrm{t}_1[\mathrm{e}_2 / x]$.
(2) *If* $\mathrm{G}_1, x : \mathrm{t}_2, \mathrm{G}_2 \vdash \gamma : \mathrm{t}_0 \sim \mathrm{t}_1$*, then* $\mathrm{G}_1, \mathrm{G}_2[\mathrm{e}_2 / x] \vdash \gamma[\mathrm{e}_2 / x] : \mathrm{t}_0[\mathrm{e}_2 / x] \sim \mathrm{t}_1[\mathrm{e}_2 / x]$.
(3) *If* $\mathrm{G}_1, x : \mathrm{t}_2, \mathrm{G}_2 \vdash \eta : \mathrm{e}_0 \sim \mathrm{e}_1$*, then* $\mathrm{G}_1, \mathrm{G}_2[\mathrm{e}_2 / x] \vdash \eta[\mathrm{e}_2 / x] : \mathrm{e}_0[\mathrm{e}_2 / x] \sim \mathrm{e}_1[\mathrm{e}_2 / x]$.
(4) *If* $\mathrm{G}_1, x : \mathrm{t}_2, \mathrm{G}_2 \vdash \mathrm{e}_1 \longrightarrow \mathrm{e}_1'$*, then* $\mathrm{G}_1, \mathrm{G}_2[\mathrm{e}_2 / x] \vdash \mathrm{e}_1[\mathrm{e}_2 / x] \longrightarrow \mathrm{e}_1'[\mathrm{e}_2 / x]$.

Proof. By mutual induction on the structure of $\mathrm{G}_1, x : \mathrm{t}_2, \mathrm{G}_2 \vdash \mathrm{e}_1 : \mathrm{t}_1$, $\mathrm{G}_1, x : \mathrm{t}_2, \mathrm{G}_2 \vdash \gamma : \mathrm{t}_0 \sim \mathrm{t}_1$, and $\mathrm{G}_1, x : \mathrm{t}_2, \mathrm{G}_2 \vdash \eta : \mathrm{e}_0 \sim \mathrm{e}_1$.

Rule CE-Var**:** Here, $\mathrm{e}_1 = x'$ for some x'. We have three cases:

$x' : \mathrm{t}_1 \in \mathrm{G}_1$: By Lemma C.1, we know that $x \notin dom(\mathrm{G}_1)$. Thus, $x \neq x'$. Thus, $\mathrm{e}_1[\mathrm{e}_2 / x] = \mathrm{e}_1 = x'$. We now must show that t_1 does not mention x. This comes from the fact that t_1 is well-formed within G_1 (Lemma C.12) and thus that $fv(\mathrm{t}_1) \subseteq dom(\mathrm{G}_1)$, excluding x. We have now established that $\mathrm{t}_1[\mathrm{e}_2 / x] = \mathrm{t}_1$. Our final goal is thus $\mathrm{G}_1, \mathrm{G}_2[\mathrm{e}_2 / x] \vdash x' : \mathrm{t}_1$; we know $x' : \mathrm{t}_1 \in \mathrm{G}_1$. To use rule CE-Var, we must only show $\vdash \mathrm{G}_1, \mathrm{G}_2[\mathrm{e}_2 / x]$ **ok**. This comes straight from Lemma C.15, and we are done with this case.

$x' = x$: Using Lemma C.15 to get $\vdash \mathrm{G}_1, \mathrm{G}_2[\mathrm{e}_2 / x]$ **ok**, we are done by Lemma C.11.

$x' : \mathrm{t}_1 \in \mathrm{G}_2$: We know $x \neq x'$ by the well-formedness of the context. We must show $\mathrm{G}_1, \mathrm{G}_2[\mathrm{e}_2 / x] \vdash x' : \mathrm{t}_1[\mathrm{e}_2 / x]$. Since $x' : \mathrm{t}_1 \in \mathrm{G}_2$, then it must be that $x' : \mathrm{t}_1[\mathrm{e}_2 / x] \in \mathrm{G}_2[\mathrm{e}_2 / x]$. We are thus done by rule CE-Var and Lemma C.15.

Rule CE-Int**:** Direct from Lemma C.15, noting that the substitutions in the subject and object have no effect.

Rule CE-Abs**:** Here, $\mathrm{e}_1 = \lambda x'{:}\mathrm{t}_3.\mathrm{e}_3$ for some x', t_3, and e_3. We also have $\mathrm{t}_1 = \mathrm{t}_3 \rightarrow \mathrm{t}_4$ for some t_4 such that $\mathrm{G}_1, x : \mathrm{t}_1, \mathrm{G}_2, x' : \mathrm{t}_3 \vdash \mathrm{e}_3 : \mathrm{t}_4$. The induction hypothesis tells us that $\mathrm{G}_1, \mathrm{G}_2[\mathrm{e}_2 / x], x' : \mathrm{t}_3[\mathrm{e}_2 / x] \vdash \mathrm{e}_3[\mathrm{e}_2 / x] : \mathrm{t}_4[\mathrm{e}_2 / x]$. This is exactly what we need to use rule CE-Abs, and we are thus done (noting that it must be that $fv(\mathrm{e}_2)$ does not include x', as x' is locally bound).

Rule CE-App**:** By the induction hypothesis.

Rule CE-TAbs**:** By the induction hypothesis.

Rule CE-TApp**:** By the induction hypothesis and Lemma C.15.

Rule CE-Pack**:** Here, $\mathrm{e}_1 = \mathbf{pack}\,\mathrm{t}, \mathrm{e}\,\mathbf{as}\,\exists\, a.\mathrm{t}'$, where $\mathrm{t}_1 = \exists\, a.\mathrm{t}'$. We must show $\mathrm{G}_1, \mathrm{G}_2[\mathrm{e}_2 / x] \vdash \mathbf{pack}\,\mathrm{t}[\mathrm{e}_2 / x], \mathrm{e}[\mathrm{e}_2 / x]\,\mathbf{as}\,\exists\, a.\mathrm{t}'[\mathrm{e}_2 / x] : \exists\, a.\mathrm{t}'[\mathrm{e}_2 / x]$. Lemma C.15 gives us the first two premises of rule CE-Pack. We must show $\mathrm{G}_1, \mathrm{G}_2[\mathrm{e}_2 / x] \vdash \mathrm{e}[\mathrm{e}_2 / x] : \mathrm{t}'[\mathrm{e}_2 / x][\mathrm{t}[\mathrm{e}_2 / x] / a]$. By the algebra of substitutions, the object of this judgment equals $\mathrm{t}'[\mathrm{t} / a][\mathrm{e}_2 / x]$. By inversion on our original assumption, we know $\mathrm{G}_1, x : \mathrm{t}_2, \mathrm{G}_2 \vdash \mathrm{e} : \mathrm{t}'[\mathrm{t} / a]$. We are thus done by the induction hypothesis.

Rule CE-Open**:** Here, $\mathrm{e}_1 = \mathbf{open}\,\mathrm{e}$, where $\mathrm{G}_1, x : \mathrm{t}_2, \mathrm{G}_2 \vdash \mathrm{e} : \exists\, a.\mathrm{t}$ and $\mathrm{t}_1 = \mathrm{t}[\lfloor \mathrm{e} \rfloor / a]$. We must show $\mathrm{G}_1, \mathrm{G}_2[\mathrm{e}_2 / x] \vdash \mathbf{open}\,\mathrm{e}[\mathrm{e}_2 / x] : \mathrm{t}[\lfloor \mathrm{e} \rfloor / a][\mathrm{e}_2 / x]$. The object of this judgment equals $\mathrm{t}[\mathrm{e}_2 / x][\lfloor \mathrm{e} \rfloor[\mathrm{e}_2 / x] / a]$. To use rule CE-Open, we must show $\mathrm{G}_1, \mathrm{G}_2[\mathrm{e}_2 / x] \vdash \mathrm{e}[\mathrm{e}_2 / x] : \exists\, a.\mathrm{t}[\mathrm{e}_2 / x]$. This comes directly from the induction hypothesis, and so we are done with this case.

Rule CE-Let**:** Similar to the case for rule CE-Abs.

Rule CE-Cast**:** By the induction hypothesis.

Rule CG-Refl**:** By Lemma C.15.

Rule CG-Sym**:** By the induction hypothesis.

Rule CG-Trans**:** By the induction hypothesis.

Rule CG-Base**:** By the induction hypothesis and Lemma C.15.

Rule CG-ForAll**:** By the induction hypothesis.

Rule CG-Exists**:** By the induction hypothesis.

Rule CG-Proj**:** By the induction hypothesis.

Rule CG-ProjPack**:** By the induction hypothesis.

Rule CG-InstForAll**:** By the induction hypothesis, noting that the substitutions commute, as their domains are distinct.

Rule CG-InstExists**:** By the induction hypothesis, noting that the substitutions commute, as their domains are distinct.

Rule CG-Nth**:** By the induction hypothesis.

Rule CH-Coherence**:** By the induction hypothesis.

Rule CH-Step**:** By the induction hypothesis.

Rule CS-Beta**:** We know $\mathrm{e}_1 = (\lambda x_0{:}\mathrm{t}.\mathrm{e}_3)\,\mathrm{e}_4$ and $\mathrm{e}_1' = \mathrm{e}_3[\mathrm{e}_4 / x_0]$. We must show $\mathrm{G}_1, \mathrm{G}_2[\mathrm{e}_2 / x] \vdash (\lambda x_0{:}\mathrm{t}[\mathrm{e}_2 / x].\mathrm{e}_3[\mathrm{e}_2 / x])\,\mathrm{e}_4[\mathrm{e}_2 / x] \longrightarrow \mathrm{e}_3[\mathrm{e}_4 / x_0][\mathrm{e}_2 / x]$. Rule CS-Beta tells us $\mathrm{G}_1, \mathrm{G}_2[\mathrm{e}_2 / x] \vdash (\lambda x_0{:}\mathrm{t}[\mathrm{e}_2 / x].\mathrm{e}_3[\mathrm{e}_2 / x])\,\mathrm{e}_4[\mathrm{e}_2 / x] \longrightarrow \mathrm{e}_3[\mathrm{e}_2 / x][\mathrm{e}_4[\mathrm{e}_2 / x] / x_0]$. A little algebra on substitutions (and the fact that $x \neq x_0$, renaming if necessary) shows that these judgments are the same.

Rule CS-AppCong**:** By the induction hypothesis.

Rule CS-AppPull**:** By the induction hypothesis.

Rule CS-TAbsCong**:** By the induction hypothesis.

Rule CS-TAbsPull**:** By Lemma C.16.

Rule CS-TBeta**:** Similar to the case for rule CS-Beta, with an appeal to Lemma C.16.

Rule CS-TAppCong**:** By the induction hypothesis.

Rule CS-TAppPull**:** By the induction hypothesis and Lemma C.16.

Rule CS-PackCong**:** By the induction hypothesis.

Rule CS-OpenPack**:** By Lemma C.16.

Rule CS-OpenPackCasted**:** By Lemma C.16.

Rule CS-OpenCong**:** By the induction hypothesis.
Rule CS-OpenPull**:** By the induction hypothesis, with an appeal to Lemma C.16.
Rule CS-Let**:** Similar to the case for rule CS-Beta.
Rule CS-CastCong**:** By the induction hypothesis.
Rule CS-CastTrans**:** By the induction hypothesis, with an appeal to Lemma C.16.

□

Lemma C.18 (Type substitution in terms). *Suppose* $\mathrm{G}_1 \vdash \mathrm{t}_2 : \textbf{type}$.

(1) *If* $\mathrm{G}_1, a, \mathrm{G}_2 \vdash \mathrm{e}_1 : \mathrm{t}_1$, *then* $\mathrm{G}_1, \mathrm{G}_2[\mathrm{t}_2 / a] \vdash \mathrm{e}_1[\mathrm{t}_2 / a] : \mathrm{t}_1[\mathrm{t}_2 / a]$.
(2) *If* $\mathrm{G}_1, a, \mathrm{G}_2 \vdash \gamma_1 : \mathrm{t}_0 \sim \mathrm{t}_1$, *then* $\mathrm{G}_1, \mathrm{G}_2[\mathrm{t}_2 / a] \vdash \gamma_1[\mathrm{t}_2 / a] : \mathrm{t}_0[\mathrm{t}_2 / a] \sim \mathrm{t}_1[\mathrm{t}_2 / a]$.
(3) *If* $\mathrm{G}_1, a, \mathrm{G}_2 \vdash \eta_1 : \mathrm{e}_0 \sim \mathrm{e}_1$, *then* $\mathrm{G}_1, \mathrm{G}_2[\mathrm{t}_2 / a] \vdash \eta_1[\mathrm{t}_2 / a] : \mathrm{e}_0[\mathrm{t}_2 / a] \sim \mathrm{e}_1[\mathrm{t}_2 / a]$.
(4) *If* $\mathrm{G}_1, a, \mathrm{G}_2 \vdash \mathrm{e} \longrightarrow \mathrm{e}'$, *then* $\mathrm{G}_1, \mathrm{G}_2[\mathrm{t}_2 / a] \vdash \mathrm{e}[\mathrm{t}_2 / a] \longrightarrow \mathrm{e}'[\mathrm{t}_2 / a]$.

Proof. By mutual induction on the structure of $\mathrm{G}_1, a, \mathrm{G}_2 \vdash \mathrm{e}_1 : \mathrm{t}_1$, $\mathrm{G}_1, a, \mathrm{G}_2 \vdash \gamma_1 : \mathrm{t}_0 \sim \mathrm{t}_1$, and $\mathrm{G}_1, a, \mathrm{G}_2 \vdash \eta_1 : \mathrm{e}_0 \sim \mathrm{e}_1$.

Rule CE-Var**:** Here, $\mathrm{e}_1 = x$ for some x. We have two cases:
$x : \mathrm{t}_1 \in \mathrm{G}_1$**:** Similar to the reasoning in this case in the proof of Lemma C.17, but invoking Lemma C.14.
$x : \mathrm{t}_1 \in \mathrm{G}_2$**:** Similar to the reasoning in this case in the proof of Lemma C.17, but invoking Lemma C.14.
Rule CE-Int**:** By Lemma C.14.
Rule CE-Abs**:** By the induction hypothesis.
Rule CE-App**:** By the induction hypothesis.
Rule CE-TAbs**:** By the induction hypothesis.
Rule CE-TApp**:** By the induction hypothesis and Lemma C.14.
Rule CE-Pack**:** Similar to this case in the proof of Lemma C.17, using Lemma C.14.
Rule CE-Open**:** Similar to this case in the proof of Lemma C.17.
Rule CE-Let**:** Similar to this case in the proof of Lemma C.17.
Rule CE-Cast**:** By the induction hypothesis.
Rule CG-Refl**:** By Lemma C.14.
Rule CG-Sym**:** By the induction hypothesis.
Rule CG-Trans**:** By the induction hypothesis.
Rule CG-Base**:** By the induction hypothesis and Lemma C.14.
Rule CG-ForAll**:** By the induction hypothesis.
Rule CG-Exists**:** By the induction hypothesis.
Rule CG-Proj**:** By the induction hypothesis.
Rule CG-ProjPack**:** By the induction hypothesis.
Rule CG-InstForAll**:** By the induction hypothesis, noting that the substitutions commute as their domains are distinct (renaming the local bound variable, if necessary).
Rule CG-InstExists**:** By the induction hypothesis, noting that the substitutions commute as their domains are distinct (renaming the local bound variable, if necessary).
Rule CG-Nth**:** By the induction hypothesis.
Rule CH-Coherence**:** By the induction hypothesis.
Rule CH-Step**:** By the induction hypothesis.
Cases for $\mathrm{G}_1, a, \mathrm{G}_2 \vdash \mathrm{e} \longrightarrow \mathrm{e}'$**:** Similar to these cases in the proof of Lemma C.17.

□

Lemma C.19 (Object regularity).

(1) *If* $\mathrm{G} \vdash \mathrm{e} : \mathrm{t}$, *then* $\mathrm{G} \vdash \mathrm{t} : \textbf{type}$.

(2) *If* $G \vdash \gamma : t_1 \sim t_2$*, then* $G \vdash t_1 : \textbf{type}$ *and* $G \vdash t_2 : \textbf{type}$.
(3) *If* $G \vdash \eta : e_1 \sim e_2$*, then there exist* t_1 *and* t_2 *such that* $G \vdash e_1 : t_1$ *and* $G \vdash e_2 : t_2$.

Proof. By mutual structural induction on the typing judgments. Note that we know $\vdash G\ \textbf{ok}$ by Lemma C.1.

Rule CE-Var**:** By Lemma C.12.
Rule CE-Int**:** Trivial, by rule CT-Base.
Rule CE-Abs**:** Here, we know $t = t_1 \rightarrow t_2$. We know $\vdash G, x : t_1\ \textbf{ok}$ by Lemma C.1. Thus, by Lemma C.12, we have $G \vdash t_1 : \textbf{type}$. The induction hypothesis gives us $G, x : t_1 \vdash t_2 : \textbf{type}$, but we also know that $x \notin fv(t_2)$. We can use Lemma C.9 to get $G \vdash t_2 : \textbf{type}$, and we are done by rule CT-Base.
Rule CE-App**:** By the induction hypothesis, inverting rule CT-Base.
Rule CE-TAbs**:** By the induction hypothesis and rule CT-ForAll.
Rule CE-TApp**:** Here, we know $e = e_1\, t_2$, where $t = t_1[t_2 / a]$ and $G \vdash e_1 : \forall a.t_1$ and $G \vdash t_2 : \textbf{type}$. We must show $G \vdash t_1[t_2 / a] : \textbf{type}$; we are thus done by Lemma C.14.
Rule CE-Pack**:** By inversion.
Rule CE-Open**:** We know $e = \textbf{open}\, e_0$, and (by inversion) $G \vdash e_0 : \exists a.t_0$. We must prove $G \vdash t_0[\lfloor e_0 \rfloor / a] : \textbf{type}$. The induction hypothesis tells us that $G \vdash \exists a.t_0 : \textbf{type}$. Inversion by rule CT-Exists then tells us $G, a \vdash t_0 : \textbf{type}$. To use Lemma C.14, we must now show $G \vdash \lfloor e_0 \rfloor : \textbf{type}$. To use rule CT-Proj, we must now show the following:
$\vdash G\ \textbf{ok}$**:** This is from Lemma C.1.
$\boldsymbol{fv}(e_0) \subseteq \boldsymbol{dom}(G)$**:** This is from Lemma C.13.
Rule CT-Proj gives us $G \vdash \lfloor e_0 \rfloor : \textbf{type}$ and then Lemma C.14 gives us $G \vdash t_0[\lfloor e_0 \rfloor / a] : \textbf{type}$ as desired.
Rule CE-Let**:** By the induction hypothesis and Lemma C.15.
Rule CE-Cast**:** By the induction hypothesis.
Rule CG-Refl**:** By inversion.
Rule CG-Sym**:** By the induction hypothesis.
Rule CG-Trans**:** By the induction hypothesis.
Rule CG-Base**:** By the induction hypothesis and rule CT-Base.
Rule CG-ForAll**:** By the induction hypothesis and rule CT-ForAll.
Rule CG-Exists**:** By the induction hypothesis and rule CT-Exists.
Rule CG-Proj**:** By the induction hypothesis, Lemma C.13, and rule CT-Proj.
Rule CG-ProjPack**:** Here, $\gamma = \textbf{projpack}\, t_3, e\ \textbf{as}\ t_4$, and we must show $G \vdash \lfloor \textbf{pack}\, t_3, e\ \textbf{as}\ t_4 \rfloor : \textbf{type}$ and $G \vdash t_3 : \textbf{type}$. Inversion on the typing judgment gives us $G \vdash \textbf{pack}\, t_3, e\ \textbf{as}\ t_4 : t_4$. This can be so only by rule CE-Pack. We can thus invert again to get $G \vdash t_3 : \textbf{type}$. We use Lemma C.13 and we are done by rule CT-Proj.
Rule CG-InstForAll**:** In this case, we know $\gamma = \gamma_1 @ \gamma_2$, with inversion giving us $G \vdash \gamma_1 : (\forall a.t_3) \sim (\forall a.t_4)$ and $G \vdash \gamma_2 : t_5 \sim t_6$. We must show $G \vdash t_3[t_5 / a] : \textbf{type}$ and $G \vdash t_4[t_6 / a] : \textbf{type}$. Let's focus on the first of these.

We know	How
$G \vdash \forall a.t_3 : \textbf{type}$	induction hypothesis
$G, a \vdash t_3 : \textbf{type}$	inversion of rule CT-ForAll
$G \vdash t_5 : \textbf{type}$	induction hypothesis
$G, a \vdash t_3[t_5 / a] : \textbf{type}$	Lemma C.14

The derivation for $G \vdash t_4[t_6 / a] : \textbf{type}$ is similar.

Rule CG-InstExists**:** In this case, we know $\gamma = \gamma_1 @ \gamma_2$, with inversion giving us $G \vdash \gamma_1 : (\exists a.t_3) \sim (\exists a.t_4)$ and $G \vdash \gamma_2 : t_5 \sim t_6$. We must show $G \vdash t_3[t_5 / a] : \mathbf{type}$ and $G \vdash t_4[t_6 / a] : \mathbf{type}$. Let's focus on the first of these.

We know	How
$G \vdash \exists a.t_3 : \mathbf{type}$	induction hypothesis
$G, a \vdash t_3 : \mathbf{type}$	inversion of rule CT-Exists
$G \vdash t_5 : \mathbf{type}$	induction hypothesis
$G \vdash t_3[t_5 / a] : \mathbf{type}$	Lemma C.14

The derivation for $G \vdash t_4[t_6 / a] : \mathbf{type}$ is similar.

Rule CG-Nth**:** By the induction hypothesis, followed by inverting rule CT-Base.

Rule CH-Coherence**:** By inversion, using rule CE-Cast.

Rule CH-Step**:** By inversion.

□

Theorem C.20 (Preservation). *If* $G \vdash e : t$ *and* $G \vdash e \longrightarrow e'$, *then* $G \vdash e' : t$.

Proof. By induction on the structure of $G \vdash e \longrightarrow e'$.

Rule CS-Beta**:** We have $e = (\lambda x{:}t_1.e_1)\, e_2$ and $e' = e_1[e_2 / x]$, and we know $G \vdash \lambda x{:}t_1.e_1 : t_1 \rightarrow t_2$ (with our original type t equalling t_2) and $G \vdash e_2 : t_1$. The former must be by rule CE-Abs, and we can thus conclude $G, x : t_1 \vdash e_1 : t_2$ and $x \notin fv(t_2)$. Lemma C.17 tells us $G \vdash e_1[e_2 / x] : t_2[e_2 / x]$. But since $x \notin fv(t_2)$, this reduces to $G \vdash e_1[e_2 / x] : t_2$, and we are done with this case.

Rule CS-AppCong**:** By the induction hypothesis.

Rule CS-AppPull**:** In this case, we know $e = (v \rhd \gamma)\, e_2$, where $v = \lambda x{:}t_0.e_0$.

We know	How
$t = t_2$	inversion on rule CE-App
$G \vdash (v \rhd \gamma) : t_1 \rightarrow t_2$	inversion on rule CE-App
$G \vdash e_2 : t_1$	inversion on rule CE-App
$G \vdash v : t_3$	inversion on rule CE-Cast
$t_3 = t_4 \rightarrow t_5$	inversion on rule CE-Abs (using $v = \lambda x{:}t_0.e_0$)
$G \vdash \gamma : (t_4 \rightarrow t_5) \sim (t_1 \rightarrow t_2)$	inversion on rule CE-Cast
$G \vdash \mathbf{nth}_0\, \gamma : t_4 \sim t_1$	rule CG-Nth
$G \vdash \mathbf{sym}\, (\mathbf{nth}_0\, \gamma) : t_1 \sim t_4$	rule CG-Sym
$G \vdash e_2 \rhd \mathbf{sym}\, (\mathbf{nth}_0\, \gamma) : t_4$	rule CE-Cast
$G \vdash v\, (e_2 \rhd \mathbf{sym}\, (\mathbf{nth}_0\, \gamma)) : t_5$	rule CE-App
$G \vdash \mathbf{nth}_1\, \gamma : t_5 \sim t_2$	rule CG-Nth
$G \vdash (v\, (e_2 \rhd \mathbf{sym}\, (\mathbf{nth}_0\, \gamma))) \rhd \mathbf{nth}_1\, \gamma : t_2$	rule CE-Cast

Rule CS-TAbsCong**:** By the induction hypothesis.

Rule CS-TAbsPull**:** In this case, we know $e = \Lambda a.(v \rhd \gamma)$. We must prove $G \vdash (\Lambda a.v) \rhd \forall a.\gamma : t$.

We know	How
$G \vdash \Lambda a.(v \rhd \gamma) : t$	assumption
$G, a \vdash v \rhd \gamma : t_1$	inversion of rule CE-TAbs
$t = \forall a.t_1$	inversion of rule CE-TAbs
$G, a \vdash v : t_2$	inversion of rule CE-Cast
$G, a \vdash \gamma : t_2 \sim t_1$	inversion of rule CE-Cast
$G \vdash \forall a.\gamma : (\forall a.t_2) \sim (\forall a.t_1)$	rule CG-ForAll
$G \vdash \Lambda a.v : \forall a.t_2$	rule CE-TAbs
$G \vdash (\Lambda a.v) \rhd \forall a.\gamma : \forall a.t_1$	rule CE-Cast

Rule CS-TBeta**:** We have $e = (\Lambda a.v_1)\, t_2$ and $e' = v_1[t_2 / a]$. We know $G \vdash \Lambda a.v_1 : \forall a.t_1$ (where our original type t equals $t_1[t_2 / a]$). Inversion on rule CE-TAbs gives us $G, a \vdash v_1 : t_1$. We can now use Lemma C.18 to get $G \vdash v_1[t_2 / a] : t_1[t_2 / a]$ as desired.

Rule CS-TAppCong**:** By the induction hypothesis.

Rule CS-TAppPull**:** We have $e = (v \rhd \gamma)\, t_0$ where $G \vdash v : \forall a.t_2$, and we must prove $G \vdash v\, t_0 \rhd (\gamma @\langle t_0\rangle) : t$.

We know	How
$G \vdash (v \rhd \gamma)\, t_0 : t$	assumption
$G \vdash v \rhd \gamma : \forall a.t_1$	inversion of rule CE-TApp
$G \vdash t_0 : \textbf{type}$	inversion of rule CE-TApp
$t = t_1[t_0 / a]$	inversion of rule CE-TApp
$G \vdash \gamma : (\forall a.t_2) \sim (\forall a.t_1)$	inversion of rule CE-Cast
$G \vdash \langle t_0\rangle : t_0 \sim t_0$	rule CG-Refl
$G \vdash \gamma @\langle t_0\rangle : t_2[t_0 / a] \sim t_1[t_0 / a]$	rule CG-InstForAll
$G \vdash v\, t_0 : t_2[t_0 / a]$	rule CE-TApp
$G \vdash v\, t_0 \rhd (\gamma @\langle t_0\rangle) : t_1[t_0 / a]$	rule CE-Cast

Rule CS-PackCong**:** By the induction hypothesis.

Rule CS-OpenPack**:** Here, we have $\mathrm{e} \;=\; \mathbf{open}\,(\mathbf{pack}\,\mathrm{t}_1, \mathrm{v}_0\,\mathbf{as}\,\mathrm{t}_0)$.

We know	How
$G \vdash \mathbf{open}\,(\mathbf{pack}\,\mathrm{t}_1, \mathrm{v}_0\,\mathbf{as}\,\mathrm{t}_0) : \mathrm{t}$	assumption
$G \vdash \mathbf{pack}\,\mathrm{t}_1, \mathrm{v}_0\,\mathbf{as}\,\mathrm{t}_0 : \exists\, a.\mathrm{t}_2$	inversion of rule CE-Open
$\mathrm{t} \;=\; \mathrm{t}_2[\lfloor \mathbf{pack}\,\mathrm{t}_1, \mathrm{v}_0\,\mathbf{as}\,\mathrm{t}_0 \rfloor \,/\, a]$	inversion of rule CE-Open
$G \vdash \mathrm{v}_0 : \mathrm{t}_2[\mathrm{t}_1 \,/\, a]$	inversion of rule CE-Pack
$\mathrm{t}_0 \;=\; \exists\, a.\mathrm{t}_2$	inversion of rule CE-Pack
$G \vdash \mathrm{t}_0 : \mathbf{type}$	inversion of rule CE-Pack
$G \vdash \langle \mathrm{t}_0 \rangle : (\exists\, a.\mathrm{t}_2) \sim (\exists\, a.\mathrm{t}_2)$	rule CG-Refl
$G \vdash \mathbf{projpack}\,\mathrm{t}_1, \mathrm{v}_0\,\mathbf{as}\,\mathrm{t}_0 : \lfloor \mathbf{pack}\,\mathrm{t}_1, \mathrm{v}_0\,\mathbf{as}\,\mathrm{t}_0 \rfloor \sim \mathrm{t}_1$	rule CG-ProjPack
$G \vdash \mathbf{sym}\,(\mathbf{projpack}\,\mathrm{t}_1, \mathrm{v}_0\,\mathbf{as}\,\mathrm{t}_0) : \mathrm{t}_1 \sim \lfloor \mathbf{pack}\,\mathrm{t}_1, \mathrm{v}_0\,\mathbf{as}\,\mathrm{t}_0 \rfloor$	rule CG-Sym
$G \vdash \langle \mathrm{t}_0 \rangle\, @(\mathbf{sym}\,(\mathbf{projpack}\,\mathrm{t}_1, \mathrm{v}_0\,\mathbf{as}\,\mathrm{t}_0)) : \mathrm{t}_2[\mathrm{t}_1 \,/\, a] \sim \mathrm{t}_2[\lfloor \mathbf{pack}\,\mathrm{t}_1, \mathrm{v}_0\,\mathbf{as}\,\mathrm{t}_0 \rfloor \,/\, a]$	rule CG-InstExists
$G \vdash \mathrm{v}_0 \rhd \langle \mathrm{t}_0 \rangle\, @(\mathbf{sym}\,(\mathbf{projpack}\,\mathrm{t}_1, \mathrm{v}_0\,\mathbf{as}\,\mathrm{t}_0)) : \mathrm{t}_2[\lfloor \mathbf{pack}\,\mathrm{t}_1, \mathrm{v}_0\,\mathbf{as}\,\mathrm{t}_0 \rfloor \,/\, a]$	rule CE-Cast

We thus see that the reduct has the same type as the redex, and we are done with this case.

Rule CS-OpenPackCasted**:** Similar to the previous case; note that we need rule CS-OpenPackCasted distinct from rule CS-OpenPack only to support determinism of reduction; otherwise both could be subsumed by a version of the rule that packed an expression e instead of a value.

Rule CS-OpenCong**:** We must have $\mathrm{e} \;=\; \mathbf{open}\,\mathrm{e}_0$. Inverting rule CE-Open in the derivation for $G \vdash \mathbf{open}\,\mathrm{e}_0 : \mathrm{t}$ tells us $G \vdash \mathrm{e}_0 : \exists\, a.\mathrm{t}_2$ and $\mathrm{t} \;=\; \mathrm{t}_2[\lfloor \mathrm{e}_0 \rfloor \,/\, a]$. Given $G \vdash \mathrm{e}_0 \longrightarrow \mathrm{e}_0'$, we must now show $G \vdash \mathbf{open}\,\mathrm{e}_0' \rhd \langle \exists\, a.\mathrm{t}_2 \rangle\, @(\mathbf{sym}\,\lfloor \mathbf{step}\,\mathrm{e} \rfloor) : \mathrm{t}_2[\lfloor \mathrm{e}_0 \rfloor \,/\, a]$.

We know	How
$\mathsf{G} \vdash \mathsf{e}_0' : \exists\, a.\mathsf{t}_2$	induction hypothesis
$\mathsf{G} \vdash \textbf{step}\, \mathsf{e}_0 : \mathsf{e}_0 \sim \mathsf{e}_0'$	rule CH-Step
$\mathsf{G} \vdash \lfloor \textbf{step}\, \mathsf{e}_0 \rfloor : \lfloor \mathsf{e}_0 \rfloor \sim \lfloor \mathsf{e}_0' \rfloor$	rule CG-Proj
$\mathsf{G} \vdash \textbf{sym}\, \lfloor \textbf{step}\, \mathsf{e}_0 \rfloor : \lfloor \mathsf{e}_0' \rfloor \sim \lfloor \mathsf{e}_0 \rfloor$	rule CG-Sym
$\mathsf{G} \vdash \exists\, a.\mathsf{t}_2 : \textbf{type}$	Lemma C.19
$\mathsf{G} \vdash \langle \exists\, a.\mathsf{t}_2 \rangle : (\exists\, a.\mathsf{t}_2) \sim (\exists\, a.\mathsf{t}_2)$	rule CG-Refl
$\mathsf{G} \vdash \langle \exists\, a.\mathsf{t}_2 \rangle\, @(\textbf{sym}\, \lfloor \textbf{step}\, \mathsf{e}_0 \rfloor) : \mathsf{t}_2[\lfloor \mathsf{e}_0' \rfloor / a] \sim \mathsf{t}_2[\lfloor \mathsf{e}_0 \rfloor / a]$	rule CG-InstExists
$\mathsf{G} \vdash \textbf{open}\, \mathsf{e}_0' : \mathsf{t}_2[\lfloor \mathsf{e}_0' \rfloor / a]$	rule CE-Open
$\mathsf{G} \vdash \textbf{open}\, \mathsf{e}_0' \rhd \langle \exists\, a.\mathsf{t}_2 \rangle\, @(\textbf{sym}\, \lfloor \textbf{step}\, \mathsf{e}_0 \rfloor) : \mathsf{t}_2[\lfloor \mathsf{e}_0 \rfloor / a]$	rule CE-Cast

We are done with this case.

Rule CS-OpenPull: We have $\mathsf{e} = \textbf{open}\,(\mathsf{v} \rhd \gamma)$, where $\mathsf{v} = \textbf{pack}\, \mathsf{t}_0, \mathsf{v}_0\, \textbf{as}\, \exists\, a.\mathsf{t}_1$.

We know	How
$\mathsf{G} \vdash \textbf{open}\,(\mathsf{v} \rhd \gamma) : \mathsf{t}$	assumption
$\mathsf{G} \vdash \mathsf{v} \rhd \gamma : \exists\, a.\mathsf{t}_2$	inversion of rule CE-Open
$\mathsf{t} = \mathsf{t}_2[\lfloor \mathsf{v} \rhd \gamma \rfloor / a]$	inversion of rule CE-Open
$\mathsf{G} \vdash \mathsf{v} : \mathsf{t}_3$	inversion of rule CE-Cast
$\mathsf{t}_3 = \exists\, a.\mathsf{t}_1$	inversion of rule CE-Pack
$\mathsf{G} \vdash \gamma : (\exists\, a.\mathsf{t}_1) \sim (\exists\, a.\mathsf{t}_2)$	inversion of rule CE-Cast
$\mathsf{G} \vdash \mathsf{v} \rhd \gamma : \mathsf{v} \sim \mathsf{v} \rhd \gamma$	use of rule CH-Coherence
$\mathsf{G} \vdash \lfloor \mathsf{v} \rhd \gamma \rfloor : \lfloor \mathsf{v} \rfloor \sim \lfloor \mathsf{v} \rhd \gamma \rfloor$	rule CG-Proj
$\mathsf{G} \vdash \gamma\, @\lfloor \mathsf{v} \rhd \gamma \rfloor : \mathsf{t}_1[\lfloor \mathsf{v} \rfloor / a] \sim \mathsf{t}_2[\lfloor \mathsf{v} \rhd \gamma \rfloor / a]$	rule CG-InstExists
$\mathsf{G} \vdash \textbf{open}\, \mathsf{v} : \mathsf{t}_1[\lfloor \mathsf{v} \rfloor / a]$	rule CE-Open
$\mathsf{G} \vdash \textbf{open}\, \mathsf{v} \rhd \gamma\, @\lfloor \mathsf{v} \rhd \gamma \rfloor : \mathsf{t}_2[\lfloor \mathsf{v} \rhd \gamma \rfloor / a]$	rule CE-Cast

Rule CS-Let: We have $\mathsf{e} = \textbf{let}\, x = \mathsf{e}_1\, \textbf{in}\, \mathsf{e}_2$.

We know	How
$\mathsf{G} \vdash \textbf{let}\, x = \mathsf{e}_1\, \textbf{in}\, \mathsf{e}_2 : \mathsf{t}$	assumption
$\mathsf{G} \vdash \mathsf{e}_1 : \mathsf{t}_1$	inversion of rule CE-Let
$\mathsf{G}, x : \mathsf{t}_1 \vdash \mathsf{e}_2 : \mathsf{t}_2$	inversion of rule CE-Let
$\mathsf{t} = \mathsf{t}_2[\mathsf{e}_1 / x]$	inversion of rule CE-Let
$\mathsf{G} \vdash \mathsf{e}_2[\mathsf{e}_1 / x] : \mathsf{t}_2[\mathsf{e}_1 / x]$	Lemma C.17

Rule CS-CastCong: We have $\mathsf{e} = \mathsf{e}_0 \rhd \gamma$, where $\mathsf{G} \vdash \mathsf{e}_0 \longrightarrow \mathsf{e}_0'$. We must show $\mathsf{G} \vdash \mathsf{e}_0' \rhd \gamma : \mathsf{t}$.

We know	How
$\mathsf{G} \vdash \mathsf{e}_0 : \mathsf{t}_0$	inversion of rule CE-Cast
$\mathsf{G} \vdash \gamma : \mathsf{t}_0 \sim \mathsf{t}$	inversion of rule CE-Cast
$\mathsf{G} \vdash \mathsf{e}_0' : \mathsf{t}_0$	induction hypothesis
$\mathsf{G} \vdash \mathsf{e}_0' \rhd \gamma : \mathsf{t}$	rule CE-Cast

Rule CS-CastTrans: We have $\mathsf{e} = (\mathsf{v} \rhd \gamma_1) \rhd \gamma_2$, and we must prove $\mathsf{G} \vdash \mathsf{v} \rhd (\gamma_1 \,;;\, \gamma_2) : \mathsf{t}$.

We know	How
$\mathsf{G} \vdash \mathsf{v} \rhd \gamma_1 : \mathsf{t}_1$	inversion of rule CE-Cast
$\mathsf{G} \vdash \gamma_2 : \mathsf{t}_1 \sim \mathsf{t}$	inversion of rule CE-Cast
$\mathsf{G} \vdash \mathsf{v} : \mathsf{t}_2$	inversion of rule CE-Cast (again)
$\mathsf{G} \vdash \gamma_1 : \mathsf{t}_2 \sim \mathsf{t}_1$	inversion of rule CE-Cast
$\mathsf{G} \vdash \gamma_1 \,;;\, \gamma_2 : \mathsf{t}_2 \sim \mathsf{t}$	rule CG-Trans
$\mathsf{G} \vdash \mathsf{v} \rhd (\gamma_1 \,;;\, \gamma_2) : \mathsf{t}$	rule CE-Cast

□

C.4 Progress

Definition C.21 (Rewrite relation). Define rewrite relations on types $\mathrm{t}_1 \Rightarrow \mathrm{t}_2$ and terms $\mathrm{e}_1 \Rightarrow \mathrm{e}_2$ with the rules below.

$\boxed{\mathrm{t}_1 \Rightarrow \mathrm{t}_2}$ *(Rewrite relation on types)*

$$\textsc{RT-Refl}\ \frac{}{\mathrm{t} \Rightarrow \mathrm{t}} \qquad \textsc{RT-Base}\ \frac{\overline{\mathrm{t}_i \Rightarrow \mathrm{t}_i'}}{B\,\overline{\mathrm{t}} \Rightarrow B\,\overline{\mathrm{t}}'} \qquad \textsc{RT-ForAll}\ \frac{\mathrm{t} \Rightarrow \mathrm{t}'}{\forall\, a.\mathrm{t} \Rightarrow \forall\, a.\mathrm{t}'} \qquad \textsc{RT-Exists}\ \frac{\mathrm{t} \Rightarrow \mathrm{t}'}{\exists\, a.\mathrm{t} \Rightarrow \exists\, a.\mathrm{t}'} \qquad \textsc{RT-Proj}\ \frac{\mathrm{e} \Rightarrow \mathrm{e}'}{\lfloor \mathrm{e} \rfloor \Rightarrow \lfloor \mathrm{e}' \rfloor}$$

$$\textsc{RT-ProjPack}\ \frac{\mathrm{t} \Rightarrow \mathrm{t}'}{\lfloor \textbf{pack}\, \mathrm{t}, \mathrm{e}\ \textbf{as}\ \exists\, a.\mathrm{t}_0 \rfloor \Rightarrow \mathrm{t}'}$$

$\boxed{\mathrm{e}_1 \Rightarrow \mathrm{e}_2}$ *(Rewrite relation on terms)*

$$\textsc{RE-Refl}\ \frac{}{\mathrm{e} \Rightarrow \mathrm{e}} \qquad \textsc{RE-DropCo}\ \frac{\mathrm{e} \Rightarrow \mathrm{e}'}{\mathrm{e} \rhd \gamma \Rightarrow \mathrm{e}'} \qquad \textsc{RE-AddCo}\ \frac{\mathrm{e} \Rightarrow \mathrm{e}'}{\mathrm{e} \Rightarrow \mathrm{e}' \rhd \gamma} \qquad \textsc{RE-Abs}\ \frac{\mathrm{t} \Rightarrow \mathrm{t}' \qquad \mathrm{e} \Rightarrow \mathrm{e}'}{\lambda x{:}\mathrm{t}.\mathrm{e} \Rightarrow \lambda x{:}\mathrm{t}'.\mathrm{e}'} \qquad \textsc{RE-App}\ \frac{\mathrm{e}_1 \Rightarrow \mathrm{e}_1' \qquad \mathrm{e}_2 \Rightarrow \mathrm{e}_2'}{\mathrm{e}_1\,\mathrm{e}_2 \Rightarrow \mathrm{e}_1'\,\mathrm{e}_2'}$$

$$\textsc{RE-TAbs}\ \frac{\mathrm{e} \Rightarrow \mathrm{e}'}{\Lambda a.\mathrm{e} \Rightarrow \Lambda a.\mathrm{e}'} \qquad \textsc{RE-TApp}\ \frac{\mathrm{e} \Rightarrow \mathrm{e}' \qquad \mathrm{t} \Rightarrow \mathrm{t}'}{\mathrm{e}\,\mathrm{t} \Rightarrow \mathrm{e}'\,\mathrm{t}'} \qquad \textsc{RE-Pack}\ \frac{\mathrm{t} \Rightarrow \mathrm{t}' \qquad \mathrm{e} \Rightarrow \mathrm{e}' \qquad \mathrm{t}_2 \Rightarrow \mathrm{t}_2'}{\textbf{pack}\, \mathrm{t}, \mathrm{e}\ \textbf{as}\ \mathrm{t}_2 \Rightarrow \textbf{pack}\, \mathrm{t}', \mathrm{e}'\ \textbf{as}\ \mathrm{t}_2'} \qquad \textsc{RE-Open}\ \frac{\mathrm{e} \Rightarrow \mathrm{e}'}{\textbf{open}\, \mathrm{e} \Rightarrow \textbf{open}\, \mathrm{e}'}$$

$$\textsc{RE-LetCong}\ \frac{\mathrm{e}_1 \Rightarrow \mathrm{e}_1' \qquad \mathrm{e}_2 \Rightarrow \mathrm{e}_2'}{\textbf{let}\ x = \mathrm{e}_1\ \textbf{in}\ \mathrm{e}_2 \Rightarrow \textbf{let}\ x = \mathrm{e}_1'\ \textbf{in}\ \mathrm{e}_2'} \qquad \textsc{RE-Cast}\ \frac{\mathrm{e} \Rightarrow \mathrm{e}'}{\mathrm{e} \rhd \gamma \Rightarrow \mathrm{e}' \rhd \gamma'} \qquad \textsc{RE-Beta}\ \frac{\mathrm{e}_1 \Rightarrow \mathrm{e}_1' \qquad \mathrm{e}_2 \Rightarrow \mathrm{e}_2'}{(\lambda x{:}\mathrm{t}.\mathrm{e}_1)\,\mathrm{e}_2 \Rightarrow \mathrm{e}_1'[\mathrm{e}_2' \,/\, x]}$$

$$\textsc{RE-TBeta}\ \frac{\mathrm{e} \Rightarrow \mathrm{e}' \qquad \mathrm{t} \Rightarrow \mathrm{t}'}{(\Lambda a.\mathrm{e})\,\mathrm{t} \Rightarrow \mathrm{e}'[\mathrm{t}' \,/\, a]} \qquad \textsc{RE-OpenPack}\ \frac{\mathrm{e} \Rightarrow \mathrm{e}'}{\textbf{open}\,(\textbf{pack}\, \mathrm{t}, \mathrm{e}\ \textbf{as}\ \mathrm{t}_2) \Rightarrow \mathrm{e}'} \qquad \textsc{RE-Let}\ \frac{\mathrm{e}_1 \Rightarrow \mathrm{e}_1' \qquad \mathrm{e}_2 \Rightarrow \mathrm{e}_2'}{\textbf{let}\ x = \mathrm{e}_1\ \textbf{in}\ \mathrm{e}_2 \Rightarrow \mathrm{e}_2'[\mathrm{e}_1' \,/\, x]}$$

Definition C.22. Define $\Rightarrow^*$ to be the reflexive, transitive closure of $\Rightarrow$.

Lemma C.23 (Type substitution in rewrite relation).

(1) *If* $\mathrm{t}_1 \Rightarrow \mathrm{t}_2$, *then* $\mathrm{t}_1[\mathrm{t}_3 \,/\, a] \Rightarrow \mathrm{t}_2[\mathrm{t}_3 \,/\, a]$.
(2) *If* $\mathrm{e}_1 \Rightarrow \mathrm{e}_2$, *then* $\mathrm{e}_1[\mathrm{t}_3 \,/\, a] \Rightarrow \mathrm{e}_2[\mathrm{t}_3 \,/\, a]$.

Proof. By mutual induction on the structure of $\mathrm{t}_1 \Rightarrow \mathrm{t}_2$ or $\mathrm{e}_1 \Rightarrow \mathrm{e}_2$. □

Lemma C.24 (Type substitution in transitive rewrite relation).

(1) *If* $\mathrm{t}_1 \Rightarrow^* \mathrm{t}_2$, *then* $\mathrm{t}_1[\mathrm{t}_3 \,/\, a] \Rightarrow^* \mathrm{t}_2[\mathrm{t}_3 \,/\, a]$.
(2) *If* $\mathrm{e}_1 \Rightarrow^* \mathrm{e}_2$, *then* $\mathrm{e}_1[\mathrm{t}_3 \,/\, a] \Rightarrow^* \mathrm{e}_2[\mathrm{t}_3 \,/\, a]$.

Proof. By induction on the length of the reduction. □

Lemma C.25 (Substitution in rewrite relation).

(1) *If* $\mathrm{t}_1 \Rightarrow \mathrm{t}_2$, *then* $\mathrm{t}_1[\mathrm{e}_3 \,/\, x] \Rightarrow \mathrm{t}_2[\mathrm{e}_3 \,/\, x]$.
(2) *If* $\mathrm{e}_1 \Rightarrow \mathrm{e}_2$, *then* $\mathrm{e}_1[\mathrm{e}_3 \,/\, x] \Rightarrow \mathrm{e}_2[\mathrm{e}_3 \,/\, x]$.

Proof. By mutual induction on the structure of $\mathrm{t}_1 \Rightarrow \mathrm{t}_2$ or $\mathrm{e}_1 \Rightarrow \mathrm{e}_2$. □

Lemma C.26 (Substitution in the transitive rewrite relation).

(1) *If* $\mathrm{t}_1 \Rightarrow^* \mathrm{t}_2$, *then* $\mathrm{t}_1[\mathrm{e}_3 / x] \Rightarrow^* \mathrm{t}_2[\mathrm{e}_3 / x]$.
(2) *If* $\mathrm{e}_1 \Rightarrow^* \mathrm{e}_2$, *then* $\mathrm{e}_1[\mathrm{e}_3 / x] \Rightarrow^* \mathrm{e}_2[\mathrm{e}_3 / x]$.

Proof. By induction on the length of the reduction. □

Lemma C.27 (Lifting in rewrite relation). *Assume* $\mathrm{t}_1 \Rightarrow \mathrm{t}_2$.

(1) *For every* t_3, $\mathrm{t}_3[\mathrm{t}_1 / a] \Rightarrow \mathrm{t}_3[\mathrm{t}_2 / a]$.
(2) *For every* e_3, $\mathrm{e}_3[\mathrm{t}_1 / a] \Rightarrow \mathrm{e}_3[\mathrm{t}_2 / a]$.

Proof. By mutual induction on the structure of t_3 and e_3.

$\mathrm{t}_3 = a'$: We have two cases:
  $a' = a$: We are done by assumption.
  $a' \neq a$: We are done by rule RT-Refl.
$\mathrm{t}_3 = B\,\bar{\mathrm{t}}$: By the induction hypothesis and rule RT-Base.
$\mathrm{t}_3 = \forall\, a'.\mathrm{t}_4$: By the induction hypothesis and rule RT-ForAll.
$\mathrm{t}_3 = \exists\, a'.\mathrm{t}_4$: By the induction hypothesis and rule RT-Exists.
$\mathrm{t}_3 = \lfloor \mathrm{e} \rfloor$: By the induction hypothesis and rule RT-Proj.
$\mathrm{e}_3 = x$: By rule RE-Refl.
$\mathrm{e}_3 = \lambda x{:}\mathrm{t}.\mathrm{e}$: By the induction hypothesis and rule RE-Abs.
$\mathrm{e}_3 = \mathrm{e}_1\ \mathrm{e}_2$: By the induction hypothesis and rule RE-App.
$\mathrm{e}_3 = \Lambda a.\mathrm{e}$: By the induction hypothesis and rule RE-TAbs.
$\mathrm{e}_3 = \mathrm{e}\ \mathrm{t}$: By the induction hypothesis and rule RE-TApp.
$\mathrm{e}_3 = \mathbf{pack}\ \mathrm{t}, \mathrm{e}\ \mathbf{as}\ \mathrm{t}'$: By the induction hypothesis and rule RE-Pack.
$\mathrm{e}_3 = \mathbf{open}\ \mathrm{e}$: By the induction hypothesis and rule RE-Open.
$\mathrm{e}_3 = \mathbf{let}\ x = \mathrm{e}_1\ \mathbf{in}\ \mathrm{e}_2$: By the induction hypothesis and rule RE-LetCong.
$\mathrm{e}_3 = \mathrm{e} \rhd \gamma$: By the induction hypothesis and rule RE-Cast. Note that the resulting coercion need not be related to the initial coercion.

□

Lemma C.28 (Lifting in transitive rewrite relation). *Assume* $\mathrm{t}_1 \Rightarrow^* \mathrm{t}_2$.

(1) *For every* t_3, $\mathrm{t}_3[\mathrm{t}_1 / a] \Rightarrow^* \mathrm{t}_3[\mathrm{t}_2 / a]$.
(2) *For every* e_3, $\mathrm{e}_3[\mathrm{t}_1 / a] \Rightarrow^* \mathrm{e}_3[\mathrm{t}_2 / a]$.

Proof. By induction on the length of the reduction. □

Lemma C.29 (Parallel substitution of a type). *Assume* $\mathrm{t}_1 \Rightarrow \mathrm{t}_2$.

(1) *If* $\mathrm{t}_3 \Rightarrow \mathrm{t}_4$, *then* $\mathrm{t}_3[\mathrm{t}_1 / a] \Rightarrow \mathrm{t}_4[\mathrm{t}_2 / a]$.
(2) *If* $\mathrm{e}_3 \Rightarrow \mathrm{e}_4$, *then* $\mathrm{e}_3[\mathrm{t}_1 / a] \Rightarrow \mathrm{e}_4[\mathrm{t}_2 / a]$.

Proof. By mutual induction on $\mathrm{t}_3 \Rightarrow \mathrm{t}_4$ or $\mathrm{e}_3 \Rightarrow \mathrm{e}_4$.

Rule RT-Refl: By Lemma C.27.
Rule RT-Base: By the induction hypothesis.
Rule RT-ForAll: By the induction hypothesis.
Rule RT-Exists: By the induction hypothesis.
Rule RT-Proj: By the induction hypothesis.
Rule RT-ProjPack: By the induction hypothesis.
Rule RE-Refl: By Lemma C.27.
Rule RE-DropCo: By the induction hypothesis.
Rule RE-AddCo: By the induction hypothesis.
Rule RE-Abs: By the induction hypothesis.

Rule RE-App: By the induction hypothesis.
Rule RE-TAbs: By the induction hypothesis.
Rule RE-TApp: By the induction hypothesis.
Rule RE-Pack: By the induction hypothesis.
Rule RE-Open: By the induction hypothesis.
Rule RE-LetCong: By the induction hypothesis.
Rule RE-Cast: By the induction hypothesis.
Rule RE-Beta: By the induction hypothesis.
Rule RE-TBeta: By the induction hypothesis, noting that the bound variable in the rule can be considered distinct from the variable being substituted.
Rule RE-OpenPack: By the induction hypothesis.
Rule RE-Let: By the induction hypothesis.

□

Lemma C.30 (Parallel substitution). *Assume* $e_1 \Rrightarrow e_2$.

(1) *If* $t_3 \Rightarrow t_4$*, then* $t_3[e_1 / x] \Rightarrow t_4[e_2 / x]$.
(2) *If* $e_3 \Rightarrow e_4$*, then* $e_3[e_1 / x] \Rightarrow e_4[e_2 / x]$.

Proof. Similar to previous proof. □

Lemma C.31 (Local diamond).

(1) *If* $t_1 \Rightarrow t_2$ *and* $t_1 \Rightarrow t_3$*, then there exists* t_4 *such that* $t_2 \Rightarrow t_4$ *and* $t_3 \Rightarrow t_4$.
(2) *If* $e_1 \Rightarrow e_2$ *and* $e_1 \Rightarrow e_3$*, then there exists* e_4 *such that* $e_2 \Rightarrow e_4$ *and* $e_3 \Rightarrow e_4$.

Proof. By mutual induction on the derivation for $t_1 \Rightarrow t_2$ or $e_1 \Rightarrow e_2$. In all cases, if $t_1 \Rightarrow t_3$ or $e_1 \Rightarrow e_3$ is by rule RT-Refl or rule RE-Refl, then we are done, with the common reduct being t_2 or e_2. We thus ignore the possibility that $t_1 \Rightarrow t_3$ can be by rule RT-Refl or that $e_1 \Rightarrow e_3$ can be by rule RE-Refl. Similarly, the use of rule RE-AddCo to rewrite $e_1 \Rightarrow e_3$ can be countered by a use of rule RE-DropCo in $e_3 \Rightarrow e_4$, keeping the rest of the case untouched; we thus ignore the possibility of rule RE-AddCo for $e_1 \Rightarrow e_3$.

Rule RT-Refl: In this case, $t_2 = t_1$ and t_3 can be the common reduct.
Rule RT-Base: The rewrite $t_1 \Rightarrow t_3$ must also be by rule RT-Base. We are done by applying the induction hypothesis.
Rule RT-ForAll: The rewrite $t_1 \Rightarrow t_3$ must also be by rule RT-ForAll. We are done by applying the induction hypothesis.
Rule RT-Exists: The rewrite $t_1 \Rightarrow t_3$ must also be by rule RT-Exists. We are done by applying the induction hypothesis.
Rule RT-Proj: We have two cases, depending on how $t_1 \Rightarrow t_3$ was rewritten:
Rule RT-Proj: By the induction hypothesis.
Rule RT-ProjPack: We have $t_1 = \lfloor \textbf{pack}\, t, e\, \textbf{as}\, \exists\, a.t_0 \rfloor$ and $t_2 = \lfloor e'_0 \rfloor$, where $\textbf{pack}\, t, e\, \textbf{as}\, \exists\, a.t_0 \Rightarrow e'_0$. We further have $t_3 = t'$ where $t \Rightarrow t'$.

We know	How
$e'_0 = \textbf{pack}\, t'', e''\, \textbf{as}\, \exists\, a.t''_0$	inversion of rule RE-Pack
$t \Rightarrow t''$	inversion of rule RE-Pack
t''' such that $t' \Rightarrow t'''$ and $t'' \Rightarrow t'''$	induction hypothesis
choose $t_4 = t'''$	
$t_2 \Rightarrow t'''$	rule RT-ProjPack

Rule RT-ProjPack: We have two cases, depending on how $t_1 \Rightarrow t_3$ was rewritten:
Rule RT-Proj: Like the rule RT-Proj/rule RT-ProjPack case above.

Rule RT-ProjPack**:** We are done by the induction hypothesis.

Rule RE-Refl**:** In this case, $e_2 = e_1$ and e_3 can be the common reduct.

Rule RE-DropCo**:** We have two cases, depending on how $e_1 \Rrightarrow e_3$ was rewritten:

Rule RE-DropCo**:** By the induction hypothesis.

Rule RE-Cast**:** In this case, $e_1 = e \rhd \gamma$, $e \Rrightarrow e_2$, and $e_3 = e' \rhd \gamma'$ where $e \Rrightarrow e'$. The induction hypothesis gives us e_0 such that $e_2 \Rrightarrow e_0$ and $e' \Rrightarrow e_0$. Choose $e_4 = e_0$. We see that $e_2 \Rrightarrow e_4$ (from the induction hypothesis) and $e_3 \Rrightarrow e_4$ by rule RE-Coherence.

Rule RE-AddCo**:** In this case, $e_2 = e' \rhd \gamma$ where $e_1 \Rrightarrow e'$. Use the induction hypothesis to get e_5 such that $e' \Rrightarrow e_5$ and $e_3 \Rrightarrow e_5$. Choose $e_4 = e_5$. We conclude that $e_2 \Rrightarrow e_4$ by rule RE-DropCo.

Rule RE-Abs**:** By the induction hypothesis.

Rule RE-App**:** We have two cases, depending on how $e_1 \Rrightarrow e_3$ was rewritten:

Rule RE-App**:** By the induction hypothesis.

Rule RE-Beta**:** We have $e_1 = (\lambda x{:}t_1.e_5)\, e_6$, $e_2 = (\lambda x{:}t_2.e_7)\, e_8$ (where $t_1 \Rrightarrow t_2$, $e_5 \Rrightarrow e_7$, and $e_6 \Rrightarrow e_8$ (inverting rule RE-Abs)), and $e_3 = e_9[e_{10} / x]$ (where $e_5 \Rrightarrow e_9$ and $e_6 \Rrightarrow e_{10}$).

We know	How
e_{11} such that $e_7 \Rrightarrow e_{11}$ and $e_9 \Rrightarrow e_{11}$	induction hypothesis
e_{12} such that $e_8 \Rrightarrow e_{12}$ and $e_{10} \Rrightarrow e_{12}$	induction hypothesis
Choose $e_4 = e_{11}[e_{12} / x]$	
$e_2 \Rrightarrow e_4$	rule RE-Beta
$e_3 \Rrightarrow e_4$	Lemma C.30

Rule RE-TAbs**:** By the induction hypothesis.

Rule RE-TApp**:** Similar to the rule RE-App case, but referring to rule RE-TBeta and Lemma C.29.

Rule RE-Pack**:** By the induction hypothesis.

Rule RE-Open**:** Similar to the rule RE-DropCo case, but referring to rule RE-OpenPack.

Rule LetCong**:** Similar to the rule RE-App case, but referring to rule RE-Let. This case uses Lemma C.30.

Rule Cast**:** By the induction hypothesis or following the logic in the case for rules RE-DropCo and RE-Cast.

Rule Beta**:** We have two cases, depending on how $e_1 \Rrightarrow e_3$ was rewritten.

Rule RE-App**:** See the case above about rules RE-App and RE-Beta.

Rule RE-Beta**:** We have $e_1 = (\lambda x{:}t_1.e_5)\, e_6$, $e_2 = e_7[e_8 / x]$ (where $e_5 \Rrightarrow e_7$ and $e_6 \Rrightarrow e_8$), and $e_3 = e_9[e_{10} / x]$ (where $e_5 \Rrightarrow e_9$ and $e_6 \Rrightarrow e_{10}$).

We know	How
e_{11} such that $e_7 \Rrightarrow e_{11}$ and $e_9 \Rrightarrow e_{11}$	induction hypothesis
e_{12} such that $e_8 \Rrightarrow e_{12}$ and $e_{10} \Rrightarrow e_{12}$	induction hypothesis
Choose $e_4 = e_{11}[e_{12} / x]$.	
$e_2 \Rrightarrow e_4$	Lemma C.30
$e_3 \Rrightarrow e_4$	Lemma C.30

Rule RE-TBeta**:** Like the case for rule RE-Beta, but referring to rule RE-TApp and Lemma C.29.

Rule RE-OpenPack**:** By the induction hypothesis or following the logic in the case for rules RE-Open and RE-OpenPack.

Rule RE-Let**:** Like the case for rule RE-Beta, but referring to rule RE-LetCong. This case uses Lemma C.30.

□

Lemma C.32 (Confluence). *If* $t_1 \Rightarrow^* t_2$ *and* $t_1 \Rightarrow^* t_3$*, then there exists* t_4 *such that* $t_2 \Rightarrow^* t_4$ *and* $t_3 \Rightarrow^* t_4$.

Proof. Corollary of Lemma C.31. (See e.g. Baader and Nipkow [1998, Lemma 2.7.4].) □

Lemma C.33 (Rewriting existentials). *If* $\exists\, a.t_1 \Rightarrow^* t_3$ *and* $\exists\, a.t_2 \Rightarrow^* t_3$*, then there exists* t_4 *such that* $t_1 \Rightarrow^* t_4$ *and* $t_2 \Rightarrow^* t_4$.

Proof. Ignoring reflexivity, the only rule that applies to $\exists\, a.t_1$ and $\exists\, a.t_2$ is rule RT-Exists. Accordingly, an inductive argument shows that t_3 must have the form $\exists\, a.t_4$ for some t_4. Furthermore, the argument that reveals t_4 also shows that $t_1 \Rightarrow^* t_4$ and $t_2 \Rightarrow^* t_4$ as desired. □

Lemma C.34 (Rewriting existentials). *If* $\forall\, a.t_1 \Rightarrow^* t_3$ *and* $\forall\, a.t_2 \Rightarrow^* t_3$*, then there exists* t_4 *such that* $t_1 \Rightarrow^* t_4$ *and* $t_2 \Rightarrow^* t_4$.

Proof. Similar to proof of Lemma C.33. □

Lemma C.35 (Rewriting base types). *If* $B\,\overline{t} \Rightarrow^* t_0$ *and* $B\,\overline{t}' \Rightarrow^* t_0$*, then, for each* i*, there exists* t_i'' *such that* $t_i \Rightarrow^* t_i''$ *and* $t_i' \Rightarrow t_i''$.

Proof. Similar to proof of Lemma C.33. □

Lemma C.36 (Rewriting subsumes reduction). *If* $G \vdash e_1 \longrightarrow e_2$*, then* $e_1 \Rightarrow e_2$.

Proof. By induction on the structure of $G \vdash e_1 \longrightarrow e_2$. (We leave out uses of rule RE-Refl throughout.)

Rule CS-Beta**:** By rule RE-Beta.
Rule CS-AppCong**:** By the induction hypothesis and rule RE-App.
Rule CS-AppPull**:** By rules RE-AddCo, RE-App, RE-DropCo, and RE-AddCo.
Rule CS-TAbsCong**:** By the induction hypothesis and rule RE-TAbs.
Rule CS-TAbsPull**:** By rules RE-AddCo, RE-TAbs, and RE-DropCo.
Rule CS-TBeta**:** By rule RE-TBeta.
Rule CS-TAppCong**:** By the induction hypothesis and rule RE-TApp.
Rule CS-TAppPull**:** By rules RE-AddCo, RE-TApp, and RE-DropCo.
Rule CS-PackCong**:** By the induction hypothesis and rule RE-Pack.
Rule CS-OpenPack**:** By rules RE-OpenPack and RE-AddCo.
Rule CS-OpenPackCasted**:** By rules RE-OpenPack and RE-AddCo.
Rule CS-OpenCong**:** By the induction hypothesis and rule RE-Open.
Rule CS-OpenPull**:** By rules RE-AddCo, RE-Open, and RE-DropCo.
Rule CS-Let**:** By rule RE-Let.
Rule CS-CastCong**:** By the induction hypothesis and rule RE-Cast.
Rule CS-CastTrans**:** by rules RE-Cast and RE-DropCo.

□

Lemma C.37 (Completeness of the rewrite relation). *If* $G \vdash \gamma : t_1 \sim t_2$*, then there exists* t_3 *such that* $t_1 \Rightarrow^* t_3$ *and* $t_2 \Rightarrow^* t_3$.

Proof. By induction on the structure of the typing judgment.

Rule CG-Refl**:** Trivial.
Rule CG-Sym**:** By the induction hypothesis.

Rule CG-Trans**:** We have $\gamma = \gamma_1 \,;;\, \gamma_2$.

We know	How
$\mathrm{G} \vdash \gamma_1 : \mathrm{t}_1 \sim \mathrm{t}_4$	inversion of rule CG-Trans
$\mathrm{G} \vdash \gamma_2 : \mathrm{t}_4 \sim \mathrm{t}_2$	inversion of rule CG-Trans
t_5 such that $\mathrm{t}_1 \Rightarrow^* \mathrm{t}_5$ and $\mathrm{t}_4 \Rightarrow^* \mathrm{t}_5$	induction hypothesis
t_6 such that $\mathrm{t}_4 \Rightarrow^* \mathrm{t}_6$ and $\mathrm{t}_2 \Rightarrow^* \mathrm{t}_6$	induction hypothesis
t_7 such that $\mathrm{t}_5 \Rightarrow^* \mathrm{t}_7$ and $\mathrm{t}_6 \Rightarrow^* \mathrm{t}_7$	Lemma C.32

We are done, as $\mathrm{t}_1 \Rightarrow^* \mathrm{t}_7$ and $\mathrm{t}_2 \Rightarrow^* \mathrm{t}_7$.

Rule CG-Base**:** By the induction hypothesis and rule RT-Base.

Rule CG-ForAll**:** By the induction hypothesis and rule RT-ForAll.

Rule CG-Exists**:** By the induction hypothesis and rule RT-Exists.

Rule CG-Proj**:** We have $\gamma = \lfloor \eta \rfloor$, where $\mathrm{G} \vdash \eta : \mathrm{e}_1 \sim \mathrm{e}_2$. We must show that $\lfloor \mathrm{e}_1 \rfloor$ and $\lfloor \mathrm{e}_2 \rfloor$ are joinable. We have two cases, depending on the rule used to prove $\mathrm{G} \vdash \eta : \mathrm{e}_1 \sim \mathrm{e}_2$:

Rule CH-Coherence**:** In this case, $\mathrm{e}_2 = \mathrm{e}_1 \rhd \gamma'$. The common reduct is $\lfloor \mathrm{e}_1 \rfloor$, and we are done by rule RE-DropCo.

Rule CH-Step**:** In this case, $\mathrm{G} \vdash \mathrm{e}_1 \longrightarrow \mathrm{e}_2$. Lemma C.36 tells us $\mathrm{e}_1 \Rightarrow \mathrm{e}_2$; we are done by rule RE-Proj.

Rule CG-ProjPack**:** We are done by rule RT-ProjPack and rule RT-Refl.

Rule CG-InstForAll**:** Similar to the case below, but using Lemma C.34.

Rule CG-InstExists**:** We have $\gamma = \gamma_1 @ \gamma_2$.

We know	How
$\mathrm{G} \vdash \gamma_1 : (\exists\, a.\mathrm{t}_4) \sim (\exists\, a.\mathrm{t}_5)$	inversion of rule CG-InstExists
$\mathrm{G} \vdash \gamma_2 : \mathrm{t}_6 \sim \mathrm{t}_7$	inversion of rule CG-InstExists
t_8 that is the join of $\exists\, a.\mathrm{t}_4$ and $\exists\, a.\mathrm{t}_5$	induction hypothesis
t_9 that is the join of t_6 and t_7	induction hypothesis
t_{10} that is the join of t_4 and t_5	Lemma C.33
$\mathrm{t}_4[\mathrm{t}_6 / a] \Rightarrow^* \mathrm{t}_{10}[\mathrm{t}_6 / a]$	Lemma C.24
$\mathrm{t}_5[\mathrm{t}_7 / a] \Rightarrow^* \mathrm{t}_{10}[\mathrm{t}_7 / a]$	Lemma C.24
$\mathrm{t}_{10}[\mathrm{t}_6 / a] \Rightarrow^* \mathrm{t}_{10}[\mathrm{t}_9 / a]$	Lemma C.28
$\mathrm{t}_{10}[\mathrm{t}_7 / a] \Rightarrow^* \mathrm{t}_{10}[\mathrm{t}_9 / a]$	Lemma C.28
$\mathrm{t}_{10}[\mathrm{t}_9 / a]$ is the join of $\mathrm{t}_4[\mathrm{t}_6 / a]$ and $\mathrm{t}_5[\mathrm{t}_7 / a]$	transitivity

Rule CG-Nth**:** By the induction hypothesis and Lemma C.35.

□

Definition C.38 (Value type). If t is a *value type*, then t is one of the following:

(1) a base type $B\,\overline{\mathrm{t}}'$
(2) a universal type $\forall\, a.\mathrm{t}'$
(3) an existential type $\exists\, a.\mathrm{t}'$

Definition C.39 (Type head). If t is a value type, then define **head**(t) by the following equations:

$$\begin{aligned} \mathbf{head}(B\,\overline{\mathrm{t}}) &= B \\ \mathbf{head}(\forall\, a.\mathrm{t}) &= \forall \\ \mathbf{head}(\exists\, a.\mathrm{t}) &= \exists \end{aligned}$$

Lemma C.40 (Value types). *If* $\mathrm{G} \vdash \mathrm{v} : \mathrm{t}$, *then* t *is a value type.*

Proof. Straightforward case analysis on the structure of v. □

Lemma C.41 (Preservation of value types). *If* t *is a value type and* $\mathrm{t} \Rightarrow^* \mathrm{t}'$*, then* t' *is a value type and* $\mathbf{head}(\mathrm{t}) = \mathbf{head}(\mathrm{t}')$.

Proof. By induction over the length of the chain $\mathrm{t} \Rightarrow^* \mathrm{t}'$.

Zero steps: Trivial.

$n + 1$ **steps:** We have t_0 such that $\mathrm{t} \Rightarrow^* \mathrm{t}_0$ in n steps and that $\mathrm{t}_0 \Rightarrow \mathrm{t}'$. The induction hypothesis tells us that t_0 is a value type and that $\mathbf{head}(\mathrm{t}) = \mathbf{head}(\mathrm{t}_0)$. Analyzing how t_0 rewrites to t', we see it must be by rule RT-Base, rule RT-ForAll, or rule RT-Exists. In any of these cases t' is a value type such that $\mathbf{head}(\mathrm{t}_0) = \mathbf{head}(\mathrm{t}')$.

□

Lemma C.42 (Consistency). *If* $\mathrm{G} \vdash \gamma : \mathrm{t}_1 \sim \mathrm{t}_2$ *and both* t_1 *and* t_2 *are value types, then* $\mathbf{head}(\mathrm{t}_1) = \mathbf{head}(\mathrm{t}_2)$.

Proof. Lemma C.37 gives us t_3 such that $\mathrm{t}_1 \Rightarrow^* \mathrm{t}_3$ and $\mathrm{t}_2 \Rightarrow^* \mathrm{t}_3$. Lemma C.41 then tells us that t_3 is a value type with $\mathbf{head}(\mathrm{t}_3) = \mathbf{head}(\mathrm{t}_1)$. Another use of Lemma C.41 tells us that $\mathbf{head}(\mathrm{t}_3) = \mathbf{head}(\mathrm{t}_2)$. By transitivity of equality, $\mathbf{head}(\mathrm{t}_1) = \mathbf{head}(\mathrm{t}_2)$. □

Lemma C.43 (Canonical forms).

(1) *If* $\mathrm{G} \vdash \mathrm{v} : \mathrm{t}_1 \longrightarrow \mathrm{t}_2$*, then there exist* x *and* e *such that* $\mathrm{v} = \lambda x{:}\mathrm{t}_1.\mathrm{e}$.
(2) *If* $\mathrm{G} \vdash \mathrm{v} : \forall\, a.\mathrm{t}$*, then there exists* v_0 *such that* $\mathrm{v} = \Lambda a.\mathrm{v}_0$.
(3) *If* $\mathrm{G} \vdash \mathrm{v} : \exists\, a.\mathrm{t}$*, then either:*
   (a) *there exists* $\mathrm{t}_0, \mathrm{v}_0$*, and* t_1 *such that* $\mathrm{v} = \mathbf{pack}\ \mathrm{t}_0, \mathrm{v}_0\ \mathbf{as}\ \mathrm{t}_1$*, or*
   (b) *there exists* $\mathrm{t}_0, \mathrm{v}_0, \gamma_0$*, and* t_1 *such that* $\mathrm{v} = \mathbf{pack}\ \mathrm{t}_0, (\mathrm{v}_0 \rhd \gamma_0)\ \mathbf{as}\ \mathrm{t}_1$

Proof.

(1) Straightforward case analysis on the structure of v.

□

Theorem C.44 (Progress). *If* $\mathrm{G} \vdash \mathrm{e} : \mathrm{t}$*, where* G *contains only type variable bindings, then one of the following is true:*

(1) *there exists* e' *such that* $\mathrm{G} \vdash \mathrm{e} \longrightarrow \mathrm{e}'$*;*
(2) e *is a value* v*; or*
(3) e *is a casted value* $\mathrm{v} \rhd \gamma$.

Proof. By induction on the structure of the typing judgment.

Rule CE-Var**:** Impossible, as G contains only type variable bindings.
Rule CE-Int**:** Here, $\mathrm{e} = n$, a value.
Rule CE-Abs**:** Here, $\mathrm{e} = \lambda x{:}\mathrm{t}_1.\mathrm{e}_1$, a value.
Rule CE-App**:** We know $\mathrm{e} = \mathrm{e}_1\ \mathrm{e}_2$, with $\mathrm{G} \vdash \mathrm{e}_1 : \mathrm{t}_1 \longrightarrow \mathrm{t}_2$ and $\mathrm{G} \vdash \mathrm{e}_2 : \mathrm{t}_1$. Applying the induction hypothesis on the first of these yields three possibilities:
  There exists e_1' **such that** $\mathrm{G} \vdash \mathrm{e}_1 \longrightarrow \mathrm{e}_1'$**:** In this case, $\mathrm{e}_1\ \mathrm{e}_2$ steps by rule CS-AppCong.
  $\mathrm{e}_1 = \mathrm{v}_1$: Lemma C.43 tells us that $\mathrm{v}_1 = \lambda x{:}\mathrm{t}_1.\mathrm{e}_0$. Thus, our original expression is $\mathrm{e} = (\lambda x{:}\mathrm{t}_1.\mathrm{e}_0)\ \mathrm{e}_2$, which can reduce by rule CS-Beta.
  $\mathrm{e}_1 = \mathrm{v}_1 \rhd \gamma_1$: Thus, our original expression is $\mathrm{e} = (\mathrm{v}_1 \rhd \gamma_1)\ \mathrm{e}_2$. In order to use rule CS-AppPull, we need only prove $\mathrm{v}_1 = \lambda x{:}\mathrm{t}_3.\mathrm{e}_0$ for some t_3 and e_0.

We know	How
$G \vdash (v_1 \rhd \gamma_1)\, e_2 : t$	assumption
$G \vdash v_1 \rhd \gamma_1 : t_4 \to t$	inversion of rule CE-App
$G \vdash v_1 : t_5$	inversion of rule CE-Cast
$G \vdash \gamma_1 : t_5 \sim (t_4 \to t)$	inversion of rule CE-Cast
t_5 is a value type	Lemma C.40
$t_5 = t_6 \to t_7$	Lemma C.42
$v_1 = \lambda x{:}t_3.e_0$	Lemma C.43

We can thus use rule CS-AppPull, and we are done with this case.

Rule CE-TAbs: Here, $e = \Lambda a.e_0$, where $G, a \vdash e_0 : t_0$ and $t = \forall\, a.t_0$. Using the induction hypothesis on e_0 gives us three possibilities:

There exists e_0' such that $G, a \vdash e_0 \longrightarrow e_0'$**:** We are done by rule CS-TAbsCong.

$e_0 = v_0$**:** The expression $e = \Lambda a.v_0$ is a value.

$e_0 = v_0 \rhd \gamma_0$**:** We are done by rule CS-TAbsPull.

Rule CE-TApp: We know $e = e_0\, t_0$, with $G \vdash e_0 : \forall\, a.t_1$ and $G \vdash t_0 :$ **type**. A use of the induction hypothesis on e_0 yields three cases:

There exists e_0' such that $G \vdash e_0 \longrightarrow e_0'$**:** We are done by rule CS-TAppCong.

$e_0 = v_0$**:** We have $e = v_0\, t_0$. Lemma C.43 tells us that $v_0 = \Lambda a.v_1$, and thus that $e = (\Lambda a.v_1)\, t_0$. We are done by rule CS-TBeta.

$e_0 = v_0 \rhd \gamma_0$**:** We have $e = (v_0 \rhd \gamma_0)\, t_0$. To use rule CS-TAppPull, we must show $G \vdash v_0 : \forall\, a.t_1$.

We know	How
$G \vdash (v_0 \rhd \gamma_0)\, t_0 : t$	assumption
$G \vdash v_0 \rhd \gamma_0 : \forall\, a.t_3$	inversion of rule CE-TApp
$G \vdash v_0 : t_4$	inversion of rule CE-Cast
$G \vdash \gamma_0 : t_4 \sim \forall\, a.t_3$	inversion of rule CE-Cast
t_4 is a value type	Lemma C.40
$t_4 = \forall\, a.t_1$	Lemma C.42

We can now use rule CS-TAppPull, and so we are done with this case.

Rule CE-Pack: We know $e =$ **pack** t_0, e_0 **as** $\exists\, a.t_1$, where $G \vdash e_0 : t_1[t_0 / a]$. We use the induction hypothesis on e_0 to get three cases:

There exists e_0' such that $G \vdash e_0 \longrightarrow e_0'$**:** We are done by rule CS-PackCong.

$e_0 = v_0$**:** Then $e =$ **pack** t_0, v_0 **as** $\exists\, a.t_1$ is a value.

$e_0 = v_0 \rhd \gamma_0$**:** In this case, we have $e =$ **pack** $t_0, (v_0 \rhd \gamma_0)$ **as** $\exists\, a.t_1$, which is a value.

Rule CE-Open: We know $e =$ **open** e_0, where $G \vdash e_0 : \exists\, a.t_0$. Using the induction hypothesis on e_0 gives us three possibilities:

There exists e_0' such that $G \vdash e_0 \longrightarrow e_0'$**:** We are done by rule CS-OpenCong.

$e_0 = v_0$**:** Lemma C.43 gives us two cases, depending on whether the packed value is casted. If it is not, we are done by rule CS-OpenPack; if it is, we are done by rule CS-OpenPackCasted.

$e_0 = v_0 \rhd \gamma_0$**:** In this case, we have $e =$ **open** $(v_0 \rhd \gamma_0)$. To use rule CS-OpenPull, we must show only that $v_0 =$ **pack** t_1, v_1 **as** $\exists\, a.t_0$.

We know	How
$\mathsf{G} \vdash \textbf{open}\ (\mathsf{v}_0 \rhd \gamma_0) : \mathsf{t}$	assumption
$\mathsf{G} \vdash \mathsf{v}_0 \rhd \gamma_0 : \exists\, a.\mathsf{t}_2$	inversion of rule CE-Open
$\mathsf{t} \ = \ \mathsf{t}_2[\lfloor \mathsf{v}_0 \rhd \gamma_0 \rfloor \,/\, a]$	inversion of rule CE-Open
$\mathsf{G} \vdash \mathsf{v}_0 : \mathsf{t}_3$	inversion of rule CE-Cast
$\mathsf{G} \vdash \gamma_0 : \mathsf{t}_3 \sim \exists\, a.\mathsf{t}_2$	inversion of rule CE-Cast
t_3 is a value type	Lemma C.40
$\mathsf{t}_3 \ = \ \exists\, a.\mathsf{t}_4$	Lemma C.42
$\mathsf{v}_0 \ = \ \textbf{pack}\, \mathsf{t}_1, \mathsf{v}_1\ \textbf{as}\ \exists\, a.\mathsf{t}_0$	Lemma C.43

We are thus done by rule CS-OpenPull.

Rule CE-Let: We are done by rule CS-Let.

Rule CE-Cast: We know $\mathsf{e} \ = \ \mathsf{e}_0 \rhd \gamma_0$, where $\mathsf{G} \vdash \mathsf{e}_0 : \mathsf{t}_0$. We use the induction hypothesis on e_0 to get three cases:

There exists e_0' such that $\mathsf{G} \vdash \mathsf{e}_0 \longrightarrow \mathsf{e}_0'$: We are done by rule CS-CastCong.

$\mathsf{e}_0 \ = \ \mathsf{v}_0$**:** Then e is a casted value $\mathsf{v}_0 \rhd \gamma_0$ and we are done.

$\mathsf{e}_0 \ = \ \mathsf{v}_0 \rhd \gamma_1$**:** We are done by rule CS-CastTrans.

□

C.5 Erasure

An erased expression $\breve{e}$ is defined with the following grammar:

$$\begin{array}{rcl} \breve{e} & ::= & x \mid \lambda x.\breve{e} \mid \breve{e}_1\ \breve{e}_2 \mid \textbf{let}\ x = \breve{e}_1\ \textbf{in}\ \breve{e}_2 \mid n \\ \breve{v} & ::= & \lambda x.\breve{e} \mid n \end{array}$$

Define the erasure function over core expressions with the following equations:

$$\begin{array}{rcl} |x| & = & x \\ |\lambda x{:}\mathsf{t}.\mathsf{e}| & = & \lambda x.|\mathsf{e}| \\ |\mathsf{e}_1\ \mathsf{e}_2| & = & |\mathsf{e}_1|\ |\mathsf{e}_2| \\ |\Lambda a.\mathsf{e}| & = & |\mathsf{e}| \\ |\mathsf{e}\ \mathsf{t}| & = & |\mathsf{e}| \\ |\textbf{pack}\ \mathsf{t}, \mathsf{e}\ \textbf{as}\ \mathsf{t}_2| & = & |\mathsf{e}| \\ |\textbf{open}\ \mathsf{e}| & = & |\mathsf{e}| \\ |\textbf{let}\ x = \mathsf{e}_1\ \textbf{in}\ \mathsf{e}_2| & = & \textbf{let}\ x = |\mathsf{e}_1|\ \textbf{in}\ |\mathsf{e}_2| \\ |\mathsf{e} \rhd \gamma| & = & |\mathsf{e}| \\ |n| & = & n \end{array}$$

The single-step operational semantics of erased expressions is given by these rules:

$\boxed{\breve{e} \longrightarrow \breve{e}'}$ *(Single-step operational semantics)*

$$\text{ES-Beta}\ \frac{}{(\lambda x.\breve{e}_1)\ \breve{e}_2 \longrightarrow \breve{e}_1[\breve{e}_2 \,/\, x]} \qquad \text{ES-App}\ \frac{\breve{e}_1 \longrightarrow \breve{e}_1'}{\breve{e}_1\ \breve{e}_2 \longrightarrow \breve{e}_1'\ \breve{e}_2} \qquad \text{ES-Let}\ \frac{}{\textbf{let}\ x = \breve{e}_1\ \textbf{in}\ \breve{e}_2 \longrightarrow \breve{e}_2[\breve{e}_1 \,/\, x]}$$

Lemma C.45 (Erasure substitution). *For all expressions* e_1 *and* e_2, $|\mathsf{e}_1[\mathsf{e}_2 \,/\, x]| \ = \ |\mathsf{e}_1|[|\mathsf{e}_2| \,/\, x]$.

Proof. Straightforward induction on the structure of e_1. □

Lemma C.46 (Erasure type substitution). *For all expressions* e *and types* t, $|e[t / a]| = |e|$.

Proof. Straightforward induction on the structure of e. □

Lemma C.47 (Single-step erasure (⇒)). *If* $G \vdash e \longrightarrow e'$, *then either* $|e| = |e'|$ *or* $|e| \longrightarrow |e'|$.

Proof. By induction on the structure of $G \vdash e \longrightarrow e'$.

Rule CS-Beta**:** By rule ES-Beta and Lemma C.45.
Rule CS-AppCong**:** By the induction hypothesis and rule ES-App.
Rule CS-AppPull**:** Here, $|e| = |e'|$.
Rule CS-TAbsCong**:** By the induction hypothesis.
Rule CS-TAbsPull**:** Here, $|e| = |e'|$.
Rule CS-TBeta**:** By Lemma C.46.
Rule CS-TAppCong**:** By the induction hypothesis.
Rule CS-TAppPull**:** Here, $|e| = |e'|$.
Rule CS-PackCong**:** By the induction hypothesis.
Rule CS-OpenPack**:** Here, $|e| = |e'|$.
Rule CS-OpenPackCasted**:** Here, $|e| = |e'|$.
Rule CS-OpenCong**:** By the induction hypothesis.
Rule CS-OpenPull**:** Here, $|e| = |e'|$.
Rule CS-Let**:** By rule ES-Let and Lemma C.45.
Rule CS-CastCong**:** By the induction hypothesis.
Rule CS-CastTrans**:** Here, $|e| = |e'|$.

□

Theorem C.48 (Erasure). *If* $G \vdash e \longrightarrow^* e'$, *then* $|e| \longrightarrow^* |e'|$.

Proof. By induction on the length of the reduction, appealing to Lemma C.47. □